CS 6115: Certified Software Systems

Parsing in Coq

Fall 2024

Status Check

Last Time

- Regular Expression Derivatives

Status Check

Last Time

- Regular Expression Derivatives

Today

- Parsing in Coq
  - RockSalt [PLDI ’12]
  - LeapFrog [PLDI ’22]
- Symbolic bisimulations
- Bisimulations “up-to”

RockSalt

RockSalt: Better, Faster, Stronger SFI for the x86

Greg Morrisett *
greg@eecs.harvard.edu

Gang Tan
gtan@cse.lehigh.edu

Joseph Tassarotti
tassarotti@college.harvard.edu

Jean-Baptiste Tristan
tristan@seas.harvard.edu

Edward Gan
egan@college.harvard.edu

Abstract

Software-based fault isolation (SFI), as used in Google's Native Client (NaCl), relies upon a conceptually simple machine-code analysis to enforce a security policy. But for complicated architectures such as the x86, it is all too easy to get the details of the analysis wrong. We have built a new checker that is smaller, faster, and has a much reduced trusted computing base when compared to Google's original analysis. The key to our approach is automatically generating the bulk of the analysis from a declarative description which we relate to a formal model of a subset of the x86 instruction set architecture. The x86 model, developed in Coq, is of independent interest and should be usable for a wide range of machine-level verification tasks.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms security, verification

Keywords software fault isolation, domain-specific languages

1. Introduction

Native Client (NaCl) is a new service provided by Google's Chrome browser that allows native executable code to be run directly in the context of the browser [37]. To prevent buggy or malicious code from corrupting the browser's state, leaking information, or directly accessing system resources, the NaCl loader checks that the binary code respects a *sandbox* security policy. The sandbox policy is meant to ensure that, when loaded and executed, the untrusted code (a) will only read or write data in specified segments of memory, (b) will only execute code from a specified segment of memory, disjoint from the data segments, (c) will not execute a specific class of instructions (*e.g.*, system calls), and (d) will only communicate with the browser through a well-defined set of entry points.

Ensuring the correctness of the NaCl checker is crucial for preventing vulnerabilities, yet early versions had bugs that attackers could exploit, as demonstrated by a contest that Google ran [25]. A high-level goal of this work is to produce a high-assurance checker for the NaCl sandbox policy. Thus far, we have managed to construct a new NaCl checker for the 32-bit x86 (IA-32) processor (minus floating-point) which we call RockSalt. The RockSalt checker is smaller, marginally faster, and easier to modify than Google's original code. Furthermore, the core of RockSalt is automatically generated from a higher-level specification, and this generator has been proven correct with respect to a model of the x86 using the Coq proof assistant [9].

We are not the first to address assurance for SFI using formal methods. In particular, Zhao *et al*. [38] built a provably correct verifier for a sandbox policy similar to NaCl's. Specifically, building upon a model of the ARM processor in HOL [13], they constructed a program logic and a provably correct verification condition generator, which when coupled with an abstract interpretation, generates proofs that assembly code respects the policy.

Our work has two key differences: First, there is no formal model for the subset of x86 that NaCl supports. Consequently, we have constructed a new model for the x86 in Coq. We believe that this model is an important contribution of our work, as it can be used to validate reasoning about the behavior of x86 machine code in other contexts (*e.g.*, for verified compilers).

Second, Zhao *et al*.'s approach takes about 2.5 hours to check a 300 instruction program, whereas RockSalt checks roughly 1M instructions per second. Instead of a general-purpose theorem prover, RockSalt only relies upon a set of tables that encode a deterministic finite-state automaton (DFA) and a few tens of lines of (trusted) `C` code. Consequently, the checker is extremely fast, has a much smaller run-time trusted computing base, and can be easily integrated into the NaCl runtime.

1.1 Overview

This paper has two major parts: the first part describes our model of the x86 in Coq and the second describes the RockSalt NaCl checker and its proof of correctness with respect to the model.

The x86 architecture is notoriously complicated, and our fragment includes a parser for over 130 different instructions with semantic definitions for over 70 instructions[1]. This includes support for operands that include byte and word immediates, registers, and complicated addressing modes (*e.g.*, scaled index plus offset). Furthermore, the x86 allows prefix bytes, such as operand size override, locking, and string repeat, that can be combined in many different ways to change the behavior of an instruction. Finally, the instruction set architecture is so complex, that it is unlikely that we can produce a faithful model from documentation, so we must be able to validate our model against implementations.

To address these issues, we have constructed a pair of domain-specific languages (DSLs), inspired by the work on SLED [30] and λ-RTL [29] (as well as more recent work [11, 19]), for specifying

* This research was sponsored in part by NSF grants CCF-0915030, CCF-0915157, CNS-0910660, CCF-1149211, AFOSR MURI grant FA9550-09-1-0539, and a gift from Google.

[1] Some instructions have numerous encodings. For example, there are fourteen different opcode forms for the `ADC` instruction, but we count this as a single instruction.

PLDI'12, June 11–16, 2012, Beijing, China.

Context: Software Fault Isolation (SFI)

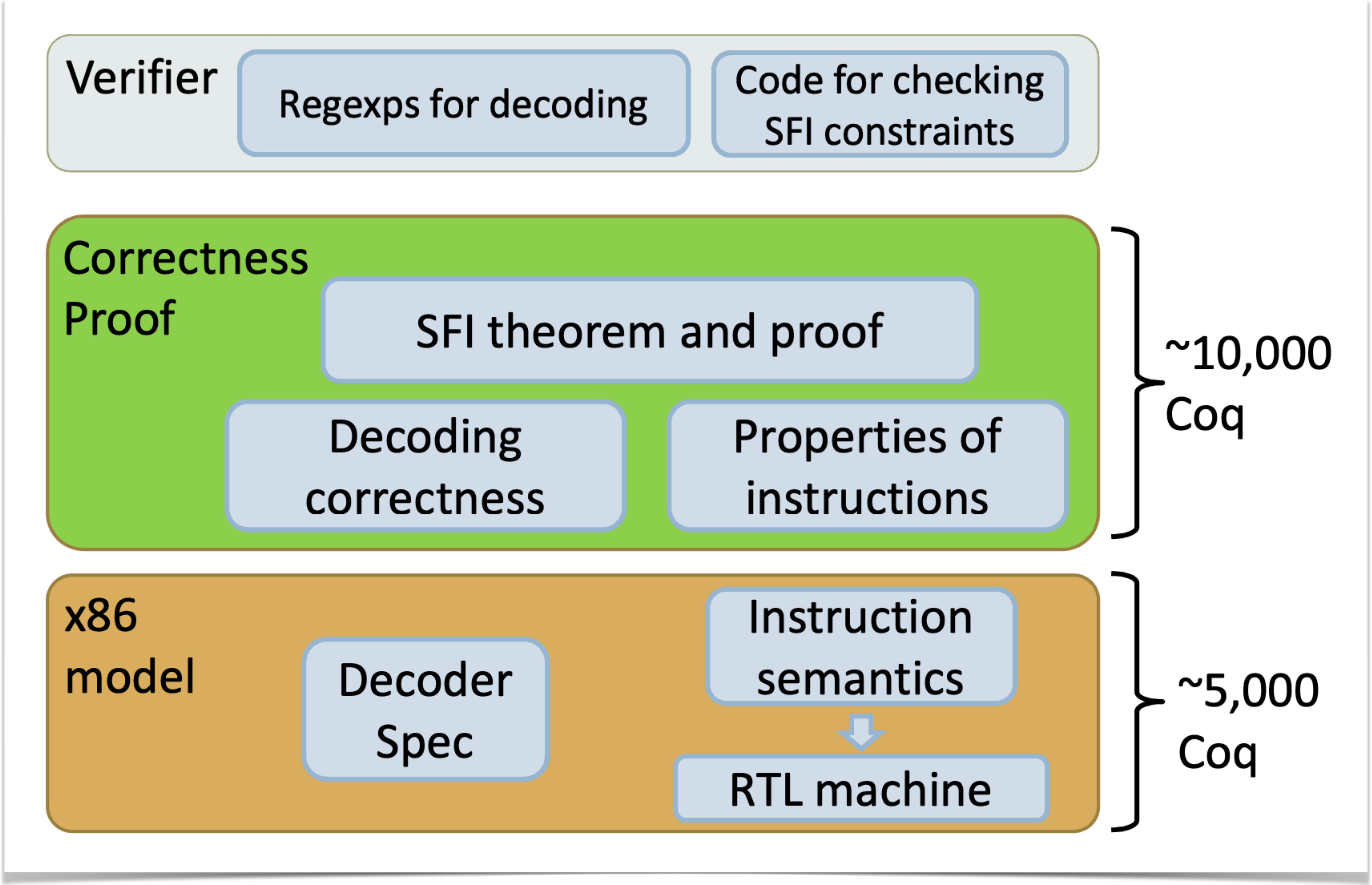
Idea: use an inlined reference monitor (IRM) to enforce safety properties for low-level code

Challenge: how to specify and parse x86 in a declarative way?

Context: Software Fault Isolation (SFI)

Idea: use an inlined reference monitor (IRM) to enforce safety properties for low-level code

Challenge: how to specify and parse x86 in a declarative way?



Slide credit: Gang Tan

Parsing via Grammars

```
Inductive grammar : Type → Type
| Char: char → grammar char
| Any: grammar char
| Eps: grammar unit
| Cat:∀T1 T2,grammar T1 → grammar T2 → grammar (T1*T2)
| Void: ∀T, grammar T
| Alt: ∀T, grammar T → grammar T → grammar T
| Star: ∀T, grammar T → grammar (list T)
| Map: ∀T1 T2, (T1 → T2) → grammar T1 → grammar T2
```

$g_1 \,\texttt{||}\, g_2$	$:=$	$\texttt{Alt}\; g_1\; g_2$	$g_1 \,\texttt{\$}\, g_2$	$:=$	$\texttt{Cat}\; g_1\; g_2$
$g \,\texttt{@}\, f$	$:=$	$\texttt{Map}\; f\; g$	$g_1 \,\texttt{\$\$}\, g_2$	$:=$	$(g_1 \,\texttt{\$}\, g_2) \,\texttt{@}\, \texttt{snd}$

Example: Parsing CALL instructions

```
Definition CALL_p : grammar instr :=
   "1110" $$ "1000" $$ word @
   (fun w => CALL true false (Imm_op w) None)
|| "1111" $$ "1111" $$ ext_op_modrm2 "010" @
   (fun op => CALL true true op None)
|| "1001" $$ "1010" $$ halfword $ word @
   (fun p => CALL false false (Imm_op (snd p))
              (Some (fst p)))
|| "1111" $$ "1111" $$ ext_op_modrm2 "011" @
   (fun op => CALL false true op None).
```

Figure 2. Parsing Specification for the CALL instruction

Semantics of Grammars

$$
\begin{aligned}
[\![\texttt{Char}\ c]\!] &= \{(c :: \texttt{nil}, c)\} \\
[\![\texttt{Any}]\!] &= \bigcup_c \{(c :: \texttt{nil}, c)\} \\
[\![\texttt{Eps}]\!] &= \{(\texttt{nil}, \texttt{tt})\} \\
[\![\texttt{Void}]\!] &= \emptyset \\
[\![\texttt{Alt}\ g_1\ g_2]\!] &= [\![g_1]\!] \cup [\![g_2]\!] \\
[\![\texttt{Cat}\ g_1\ g_2]\!] &= \{((s_1 s_2), (v_1, v_2)) \mid (s_i, v_i) \in [\![g_i]\!]\}
\end{aligned}
$$

Quiz: can you write down the semantics of Map f g and Star g?

Semantics of Grammars

$$
\begin{aligned}
[\![\texttt{Char}\, c]\!] &= \{(c :: \texttt{nil}, c)\} \\
[\![\texttt{Any}]\!] &= \textstyle\bigcup_c \{(c :: \texttt{nil}, c)\} \\
[\![\texttt{Eps}]\!] &= \{(\texttt{nil}, \texttt{tt})\} \\
[\![\texttt{Void}]\!] &= \emptyset \\
[\![\texttt{Alt}\, g_1\, g_2]\!] &= [\![g_1]\!] \cup [\![g_2]\!] \\
[\![\texttt{Cat}\, g_1\, g_2]\!] &= \{((s_1 s_2), (v_1, v_2)) \mid (s_i, v_i) \in [\![g_i]\!]\} \\
[\![\texttt{Map}\, f\, g]\!] &= \{(s, f(v)) \mid (s, v) \in [\![g]\!]\} \\
[\![\texttt{Star}\, g]\!] &= [\![\texttt{Map}\,(\lambda\, \_.\, \texttt{nil})\, \texttt{Eps}]\!] \cup \\
&\quad\; [\![\texttt{Map}\,(::)\,(\texttt{Cat}\, g\,(\texttt{Star}\, g))]\!]
\end{aligned}
$$

Derivative Specification

Quiz: what should the specification for the (semantic) derivative operation be?

Derivative Specification

Quiz: what should the specification for the (semantic) derivative operation be?

$$\texttt{deriv}_c\, g = \{(s, v) \mid (c :: s, v) \in [\![g]\!]\}$$

Derivative

$$\mathtt{deriv}_c\, g = \{(s, v) \mid (c :: s, v) \in [\![g]\!]\}$$

$$\begin{array}{rcll}
\mathtt{deriv}_c\, \mathtt{Any} & = & \mathtt{Map}\,(\lambda\, _.\, c)\, \mathtt{Eps} & \\
\mathtt{deriv}_c\, (\mathtt{Char}\, c) & = & \mathtt{Map}\,(\lambda\, _.\, c)\, \mathtt{Eps} & \\
\mathtt{deriv}_c\, (\mathtt{Alt}\, g_1\, g_2) & = & \mathtt{Alt}\,(\mathtt{deriv}_c\, g_1)\,(\mathtt{deriv}_c\, g_2) & \\
\mathtt{deriv}_c\, (\mathtt{Star}\, g) & = & \mathtt{Map}\,(::)\,(\mathtt{Cat}(\mathtt{deriv}_c\, g)\,(\mathtt{Star}\, g)) & \\
\mathtt{deriv}_c\, (\mathtt{Cat}\, g_1\, g_2) & = & \mathtt{Alt}(\mathtt{Cat}\,(\mathtt{deriv}_c\, g_1)\, g_2) & \\
 & & \quad (\mathtt{Cat}\,(\mathtt{null}\, g_1)\,(\mathtt{deriv}_c\, g_2)) & \\
\mathtt{deriv}_c\, (\mathtt{Map}\, f\, g) & = & \mathtt{Map}\, f\,(\mathtt{deriv}_c\, g) & \\
\mathtt{deriv}_c\, g & = & \mathtt{Void} & \text{otherwise}
\end{array}$$

Nullable

$$
\begin{aligned}
\texttt{null}\,\texttt{Eps} &= \texttt{Eps} \\
\texttt{null}\,(\texttt{Alt}\,g_1\,g_2) &= \texttt{Alt}\,(\texttt{null}\,g_1)\,(\texttt{null}\,g_2) \\
\texttt{null}\,(\texttt{Cat}\,g_1\,g_2) &= \texttt{Cat}\,(\texttt{null}\,g_1)\,(\texttt{null}\,g_2) \\
\texttt{null}\,(\texttt{Star}\,g) &= \texttt{Map}\,(\lambda\,\_.\,\texttt{nil})\,\texttt{Eps} \\
\texttt{null}\,(\texttt{Map}\,f\,g) &= \texttt{Map}\,f\,(\texttt{null}\,g) \\
\texttt{null}\,g &= \texttt{Void} \qquad \text{otherwise}
\end{aligned}
$$

Extraction

$$
\begin{array}{rcl}
\texttt{extract Eps} & = & \{\texttt{tt}\} \\
\texttt{extract}\ (\texttt{Star}\, g) & = & \{\texttt{nil}\} \\
\texttt{extract}\ (\texttt{Alt}\, g_1\ g_2) & = & (\texttt{extract}\ g_1) \cup (\texttt{extract}\ g_2) \\
\texttt{extract}\ (\texttt{Cat}\, g_1\ g_2) & = & \{(v_1, v_2) \mid v_i \in \texttt{extract}\ g_i\} \\
\texttt{extract}\ (\texttt{Map}\, f\ g) & = & \{f(v) \mid v \in \texttt{extract}\ g\} \\
\texttt{extract}\ g & = & \emptyset \qquad\qquad \text{otherwise}
\end{array}
$$

Smart Constructors

Cat g Eps	$\to$	g	Cat Eps g	$\to$	g
Cat g Void	$\to$	Void	Cat Void g	$\to$	Void
Alt g Void	$\to$	g	Alt Void g	$\to$	g
Star (Star g)	$\to$	Star g	Alt g g	$\to$	g

(**Aside:** these are all theorems in Kleene Algebra)

Leapfrog

Leapfrog: Certified Equivalence for Protocol Parsers

Ryan Doenges
Cornell University
USA
rhd89@cornell.edu

Tobias Kappé
ILLC, University of Amsterdam
Netherlands
t.kappe@uva.nl

John Sarracino
Cornell University
USA
jsarracino@cornell.edu

Nate Foster
Cornell University
USA
jnfoster@cs.cornell.edu

Greg Morrisett
Cornell University
USA
jgm19@cornell.edu

Abstract

We present Leapfrog, a Coq-based framework for verifying equivalence of network protocol parsers. Our approach is based on an automata model of P4 parsers, and an algorithm for symbolically computing a compact representation of a bisimulation, using "leaps." Proofs are powered by a certified compilation chain from first-order entailments to low-level bitvector verification conditions, which are discharged using off-the-shelf SMT solvers. As a result, parser equivalence proofs in Leapfrog are fully automatic and push-button.

We mechanically prove the core metatheory that underpins our approach, including the key transformations and several optimizations. We evaluate Leapfrog on a range of practical case studies, all of which require minimal configuration and no manual proof. Our largest case study uses Leapfrog to perform translation validation for a third-party compiler from automata to hardware pipelines. Overall, Leapfrog represents a step towards a world where all parsers for critical network infrastructure are verified. It also suggests directions for follow-on efforts, such as verifying relational properties involving security.

CCS Concepts: • **Theory of computation → Automata extensions**; • **Software and its engineering → Software verification**.

Keywords: P4, network protocol parsers, Coq, automata, equivalence, foundational verification, certified parsers

ACM Reference Format:
Ryan Doenges, Tobias Kappé, John Sarracino, Nate Foster, and Greg Morrisett. 2022. Leapfrog: Certified Equivalence for Protocol Parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3519939.3523715

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523715

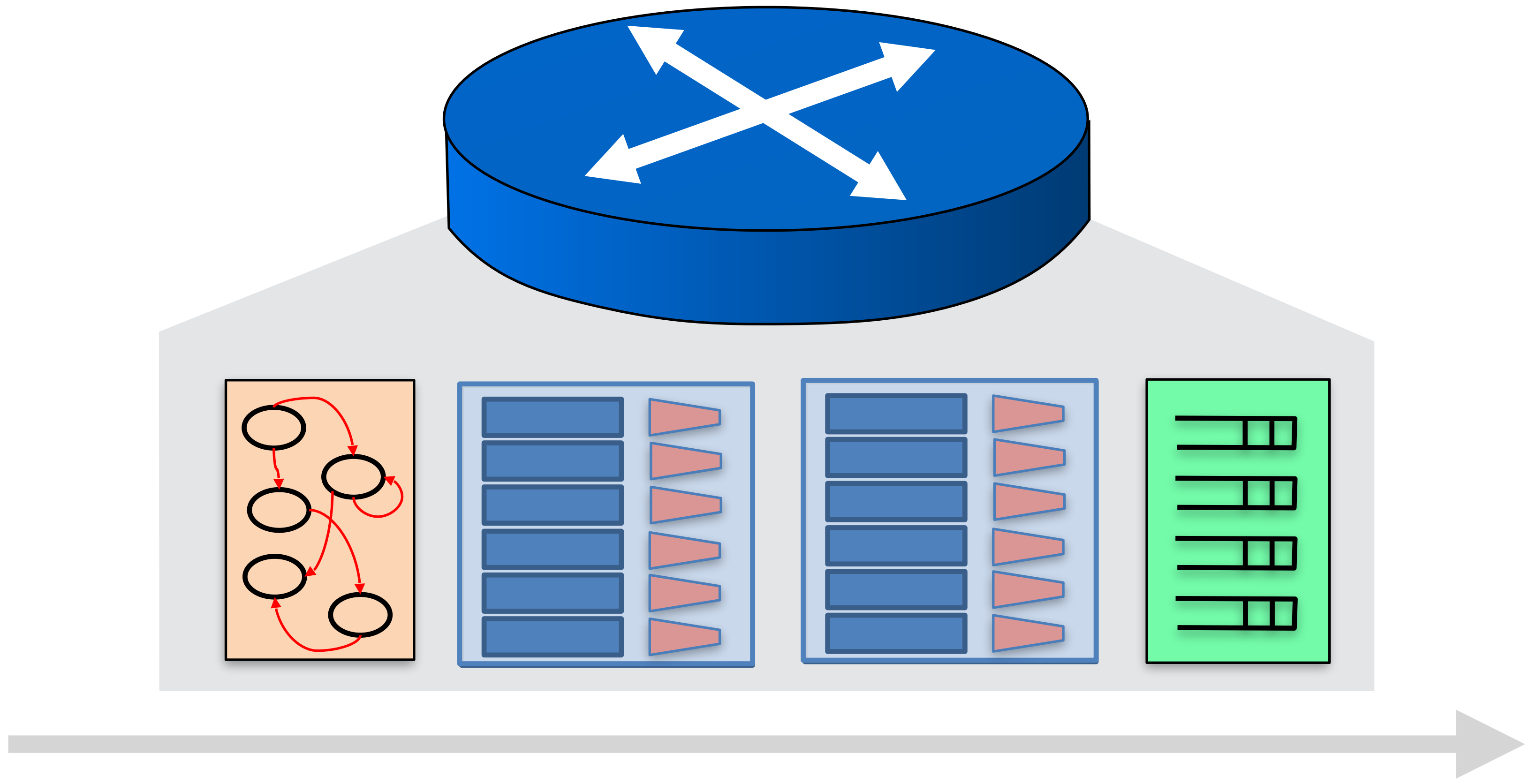
1 Introduction

Devices like routers, firewalls and network interface cards as well as operating system kernels occupy a critical role in modern communications infrastructure. Each of these implements parsing for a cornucopia of networking protocols in its *protocol parser*. The parser is the network's first line of defense, responsible for organizing and filtering unstructured and often untrusted data as it arrives from the outside world. Due to their crucial role, bugs in parsers are a significant source of crashes, vulnerabilities, and other faults [48].

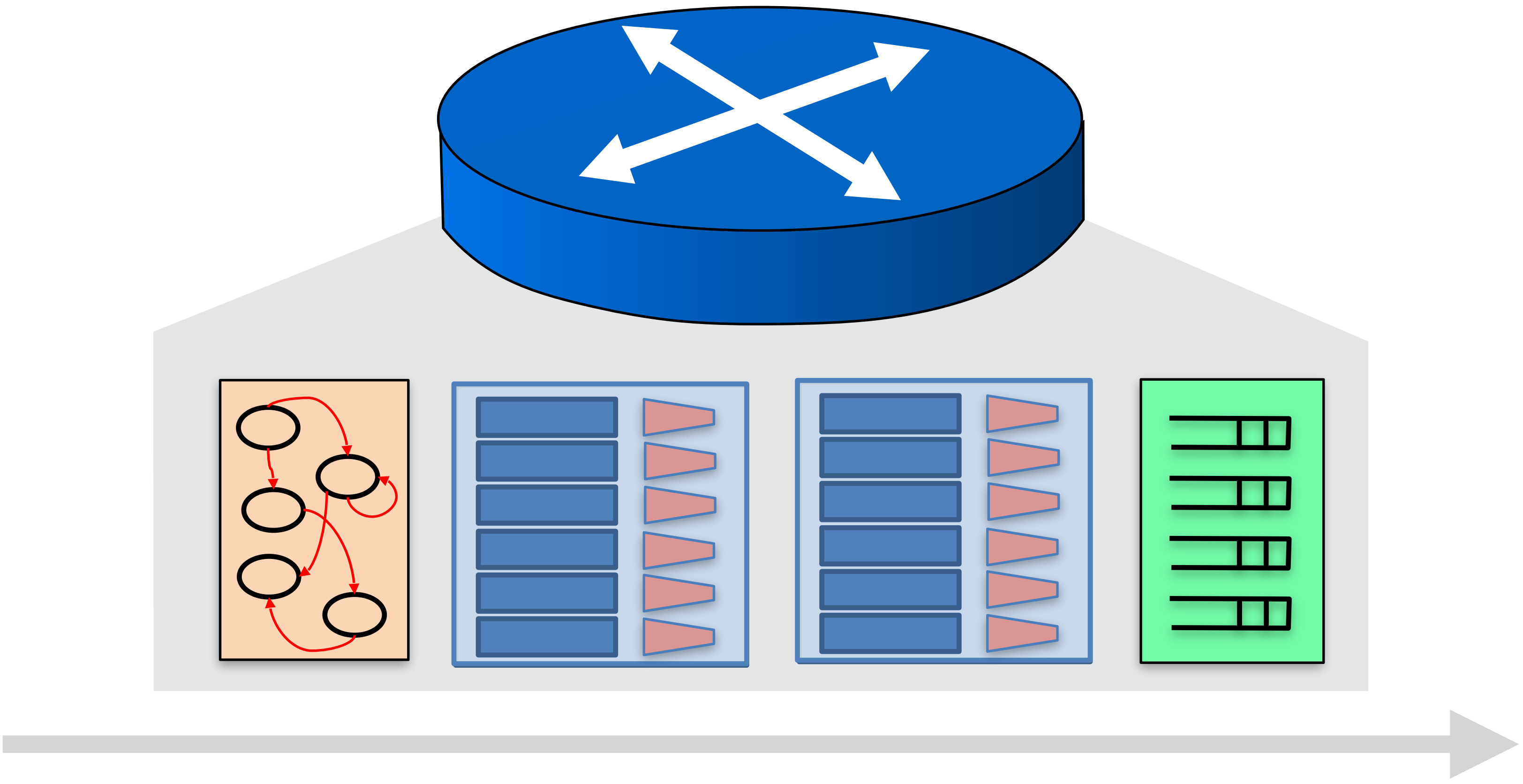
Example Router Bug. Consider the following bug, which was present in a commercial router developed by a leading equipment vendor several years ago. Internally, the router was organized around a high-throughput pipeline, which most packets traversed in a single pass. However some packets had to be *recirculated*, meaning they took additional passes through the pipeline before being sent back out on the wire. The router used an internal state variable to decide whether a packet should be recirculated. Usually this state variable was initialized by vendor-supplied code. But, as was discovered by a customer, it could also be erroneously initialized from data in non-standard, malformed packets. Hence, crafted packets could bypass the vendor-supplied initialization code, resulting in an infinite recirculation loop—a denial-of-service (DoS) attack on the router and its peers. In the presence of broadcast traffic, such a "packet storm" would monopolize the router's resources, rendering it unusable until it was rebooted.

An easy way to avoid this bug would be to modify the router's parser to filter away malformed packets, while still accepting valid packets. However, to have full confidence in the new parser, one would need to prove that it is *equivalent* to the original, modulo malformed packets. Although parsers tend to be simple, this would likely be a challenging verification task—it requires reasoning about a *relational* property across two distinct programs.

Network Acceleration

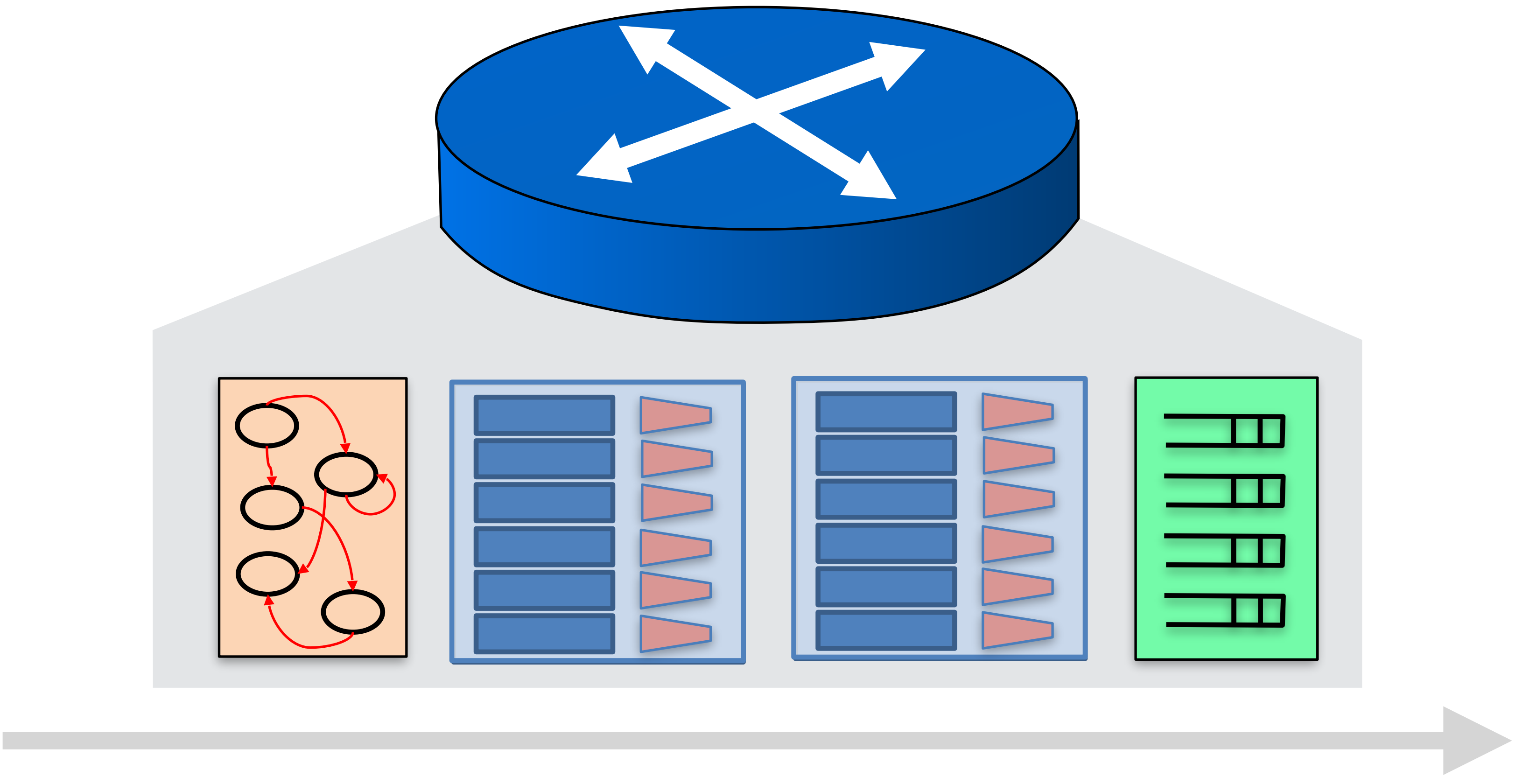


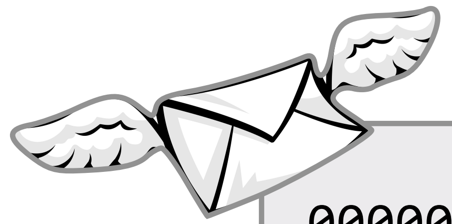
Network Acceleration



```
00000001 00000010 00000000 00000011
00000000 00000011 00000001 10101010
10111011 01010000 01100101 01110100
01110010 00110100 00100000 01101001
01110011 00100000 01100001 01110111
01100101 01110011 01101111 01101101
01100101 00100001
```

Network Acceleration

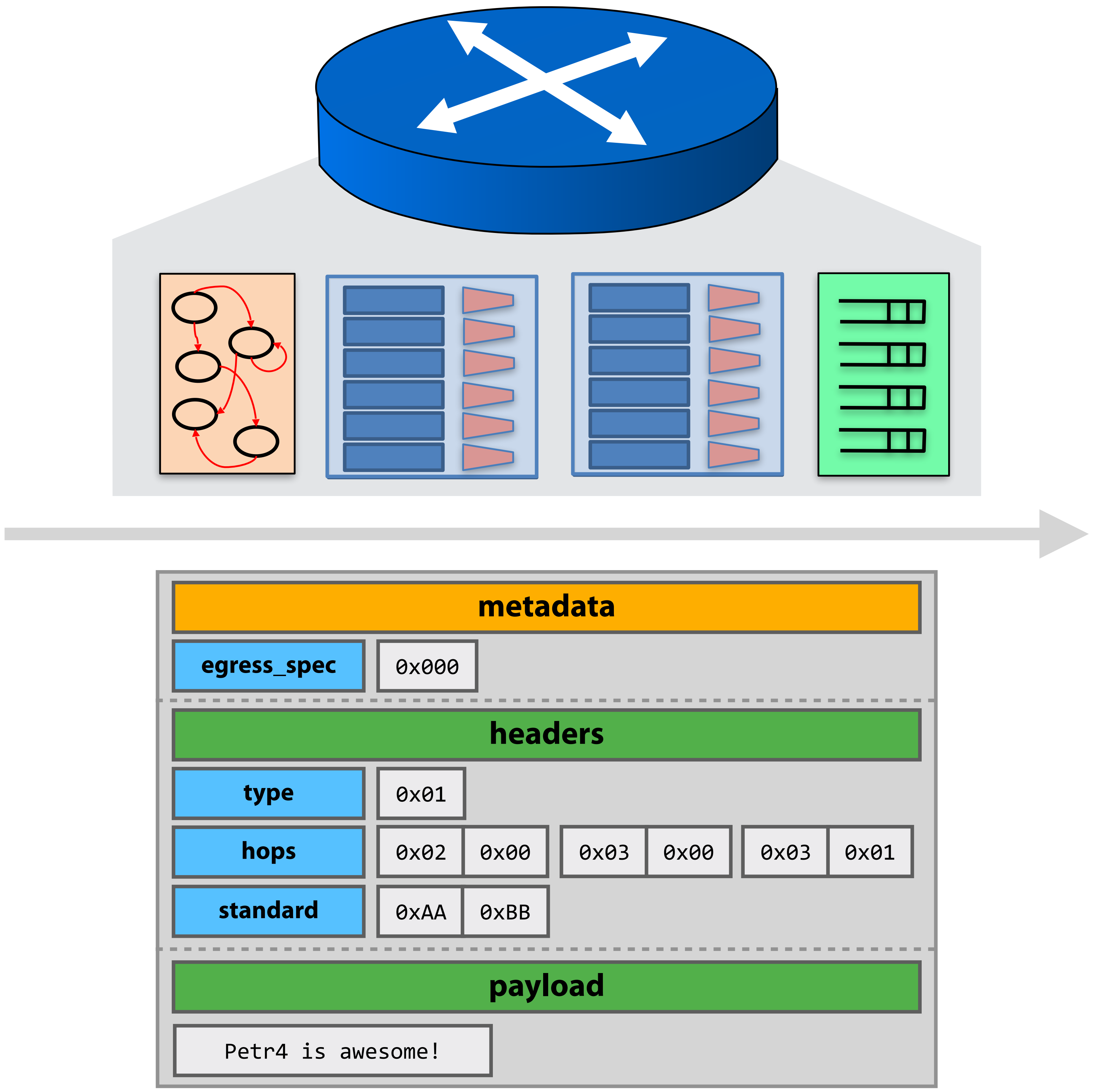




```
00000001 00000010 00000000 00000011
00000000 00000011 00000001 10101010
10111011 01010000 01100101 01110100
01110010 00110100 00100000 01101001
01110011 00100000 01100001 01110111
01100101 01110011 01101111 01101101
01100101 00100001
```

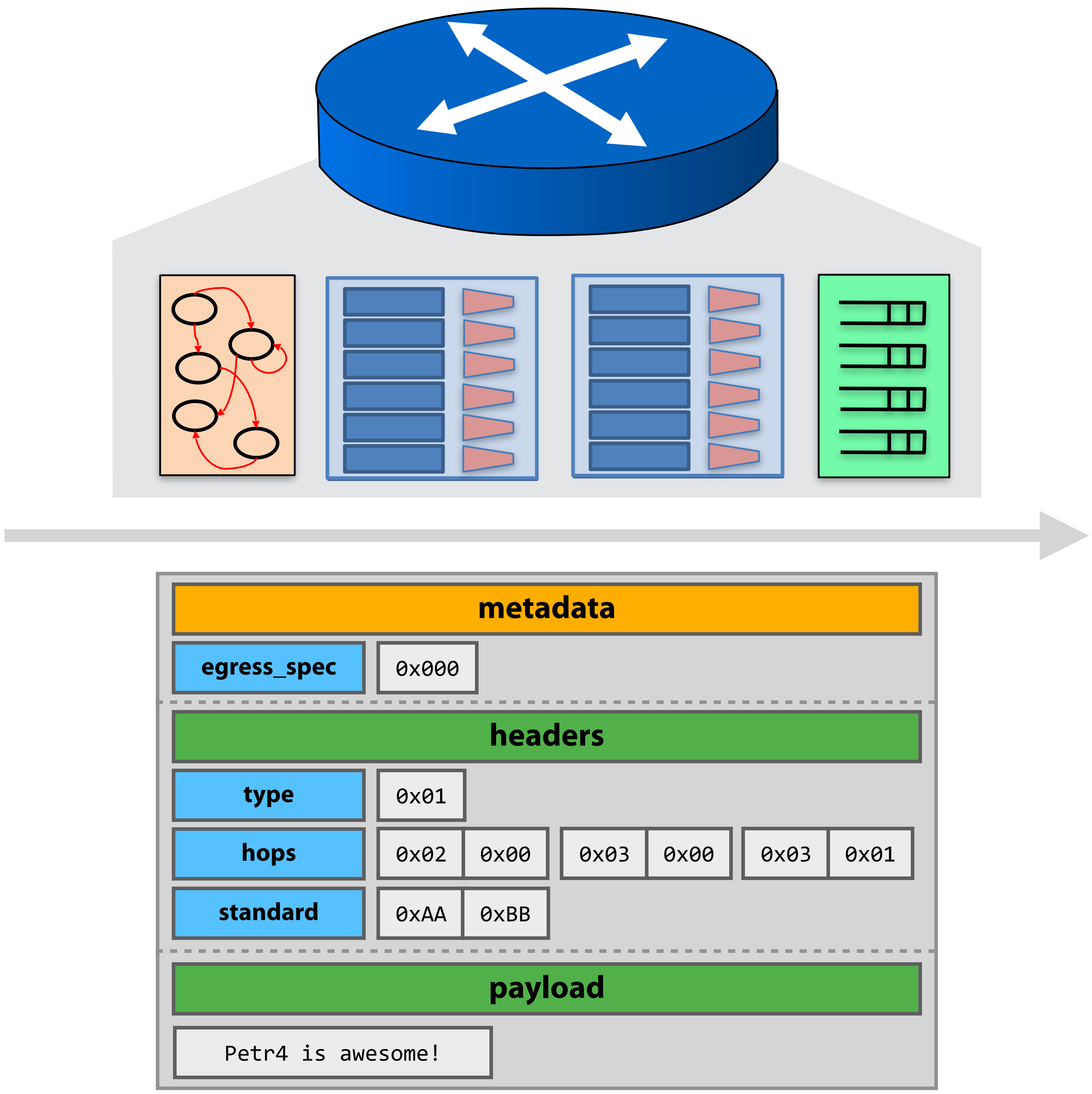
1. Parse
Extract typed representation of packet data

Network Acceleration



1. Parse
Extract typed representation of packet data

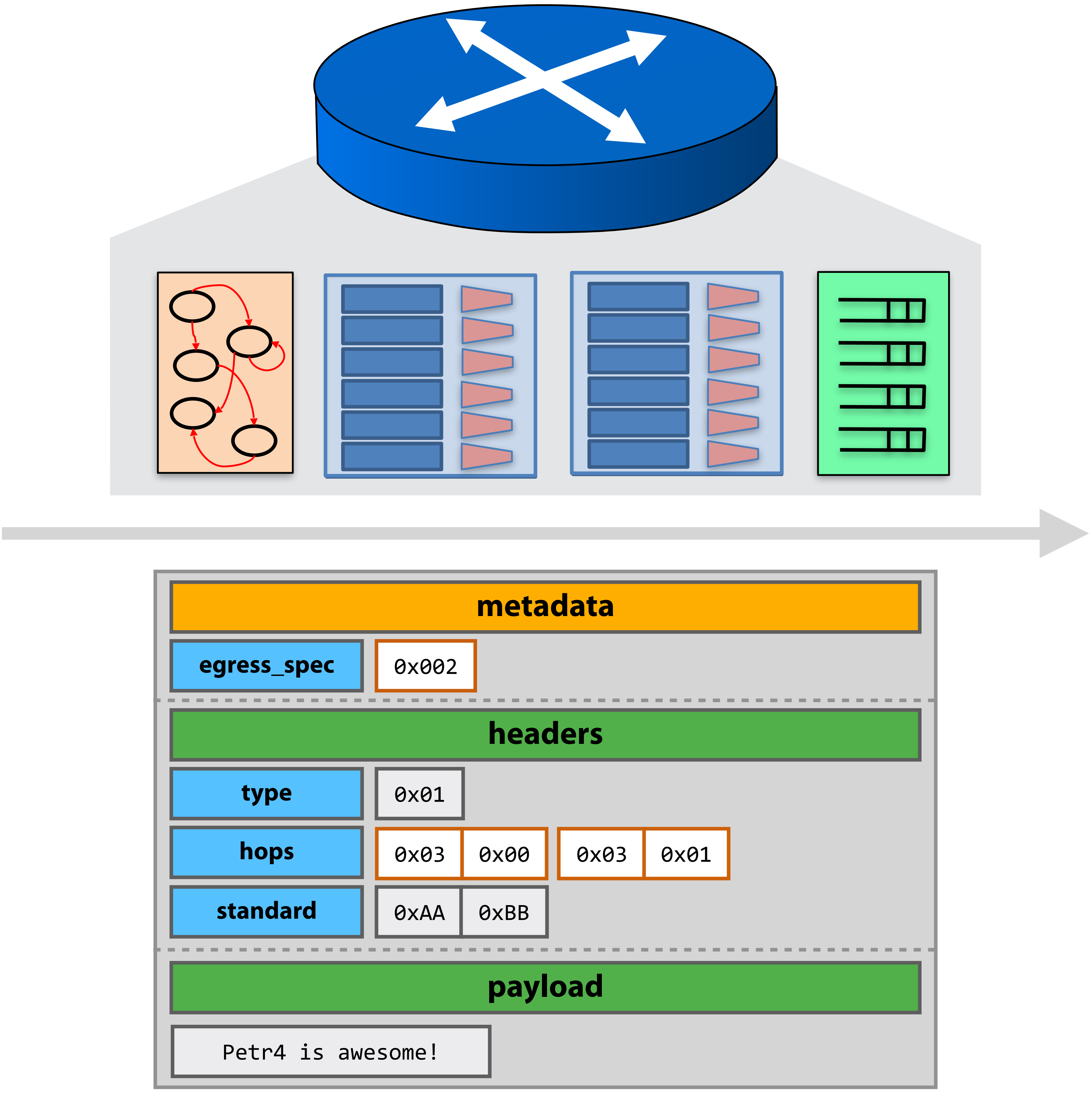
Network Acceleration



1. Parse
Extract typed representation of packet data

2. Transform
Make forwarding decision, compute outputs

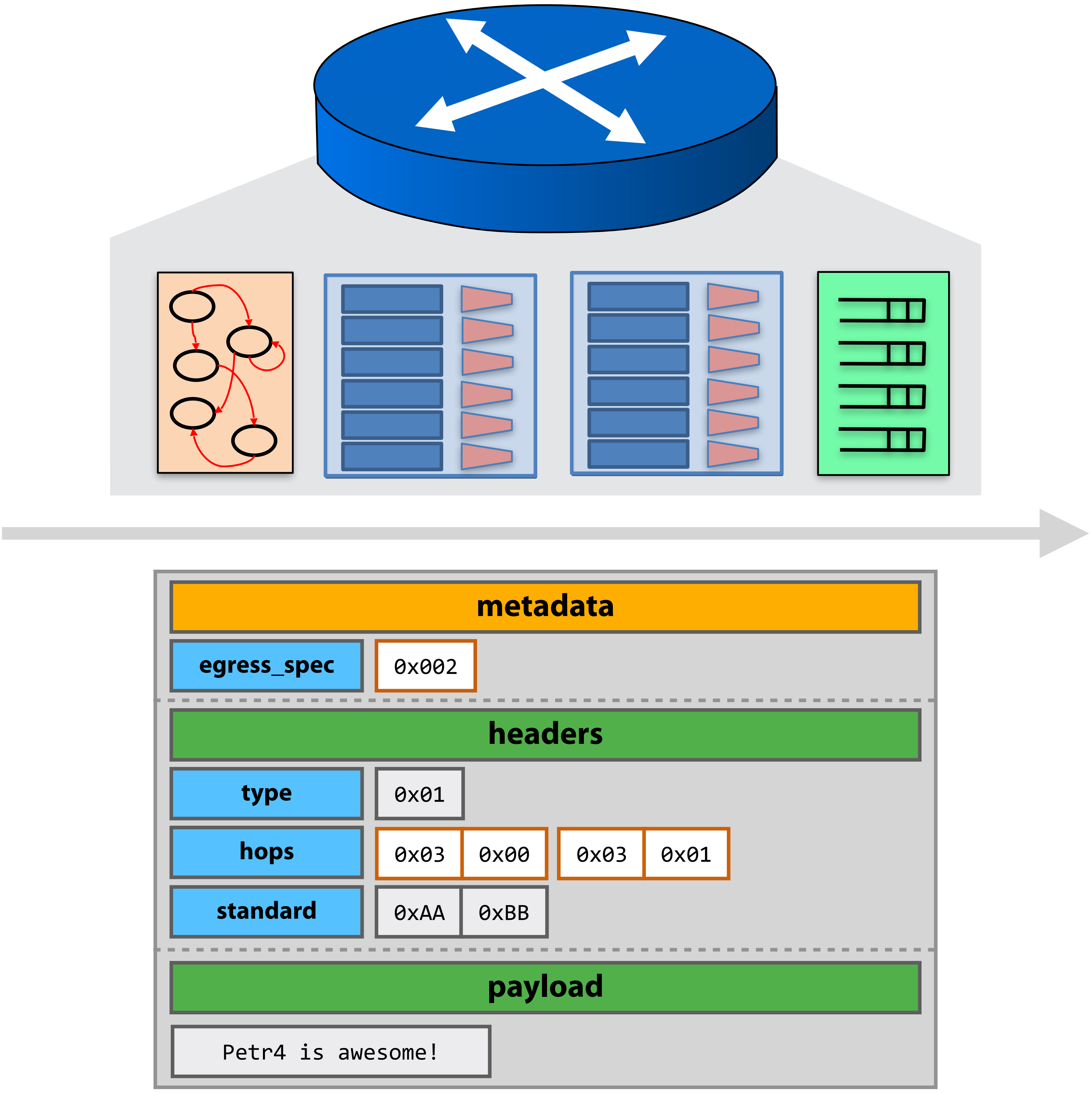
Network Acceleration



1. Parse
Extract typed representation of packet data

2. Transform
Make forwarding decision, compute outputs

Network Acceleration

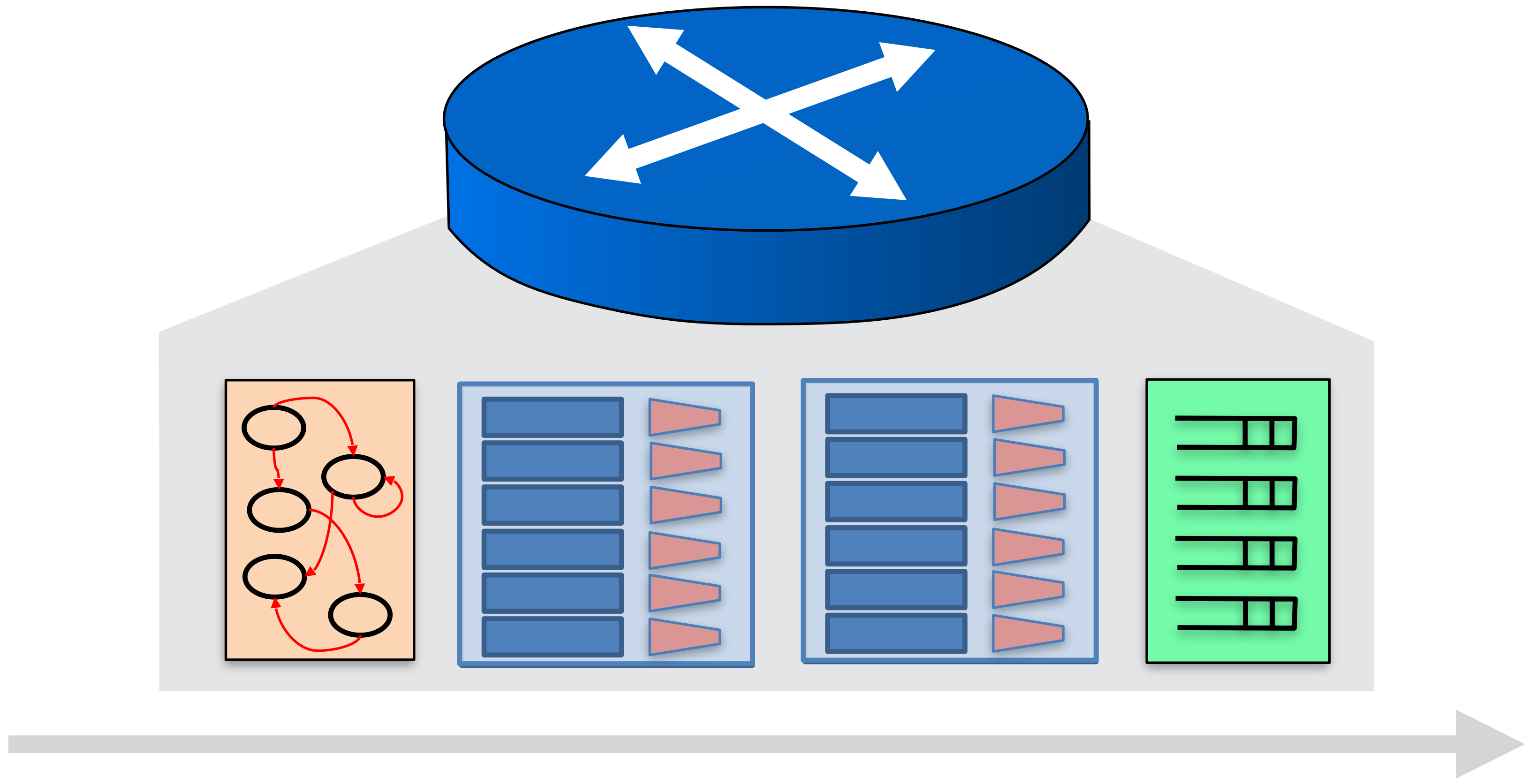


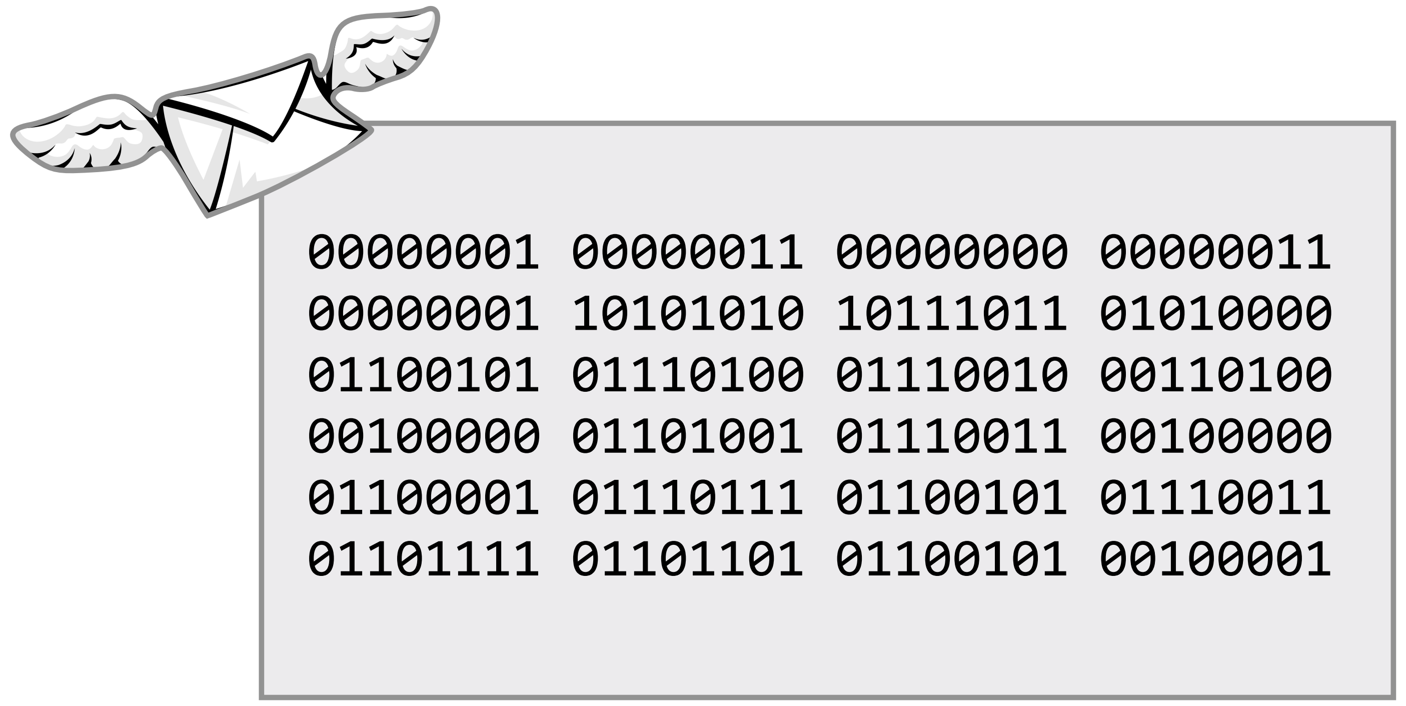
1. Parse
Extract typed representation of packet data

2. Transform
Make forwarding decision, compute outputs

3. Deparse
Map packet back into binary representation

Network Acceleration





1. Parse
Extract typed representation of packet data

2. Transform
Make forwarding decision, compute outputs

3. Deparse
Map packet back into binary representation

P4: Programming Packet Processing Pipelines

Really three languages for parsing, configurable processing, and deparsing glued together.

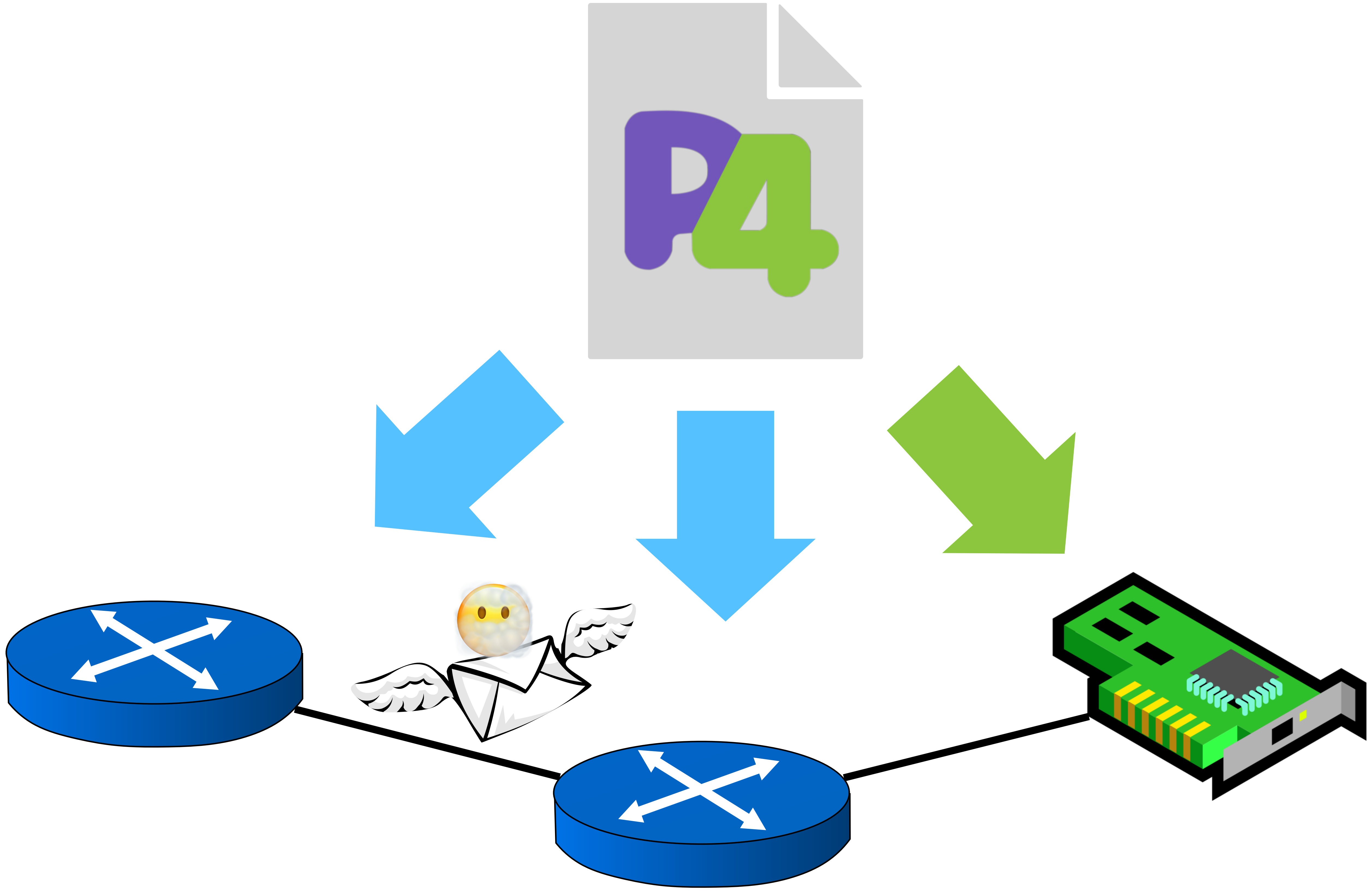
Interesting bits are state machines and match-action tables.

```
state start {
  pkt.extract(hdr.eth);
  transition select(hdr.eth.typ) {
    0x0800: parse_ip;
    default: accept;
  }
}
state parse_ip { ... }
```

```
action set_port(inout bit<8> p) {
  meta.port = p;
}
action nop() { }
table fwd_eth {
  key = { hdr.eth.dst : exact; }
  actions = { set_port; nop; }
  default_action = nop();
}
```

Security Implications

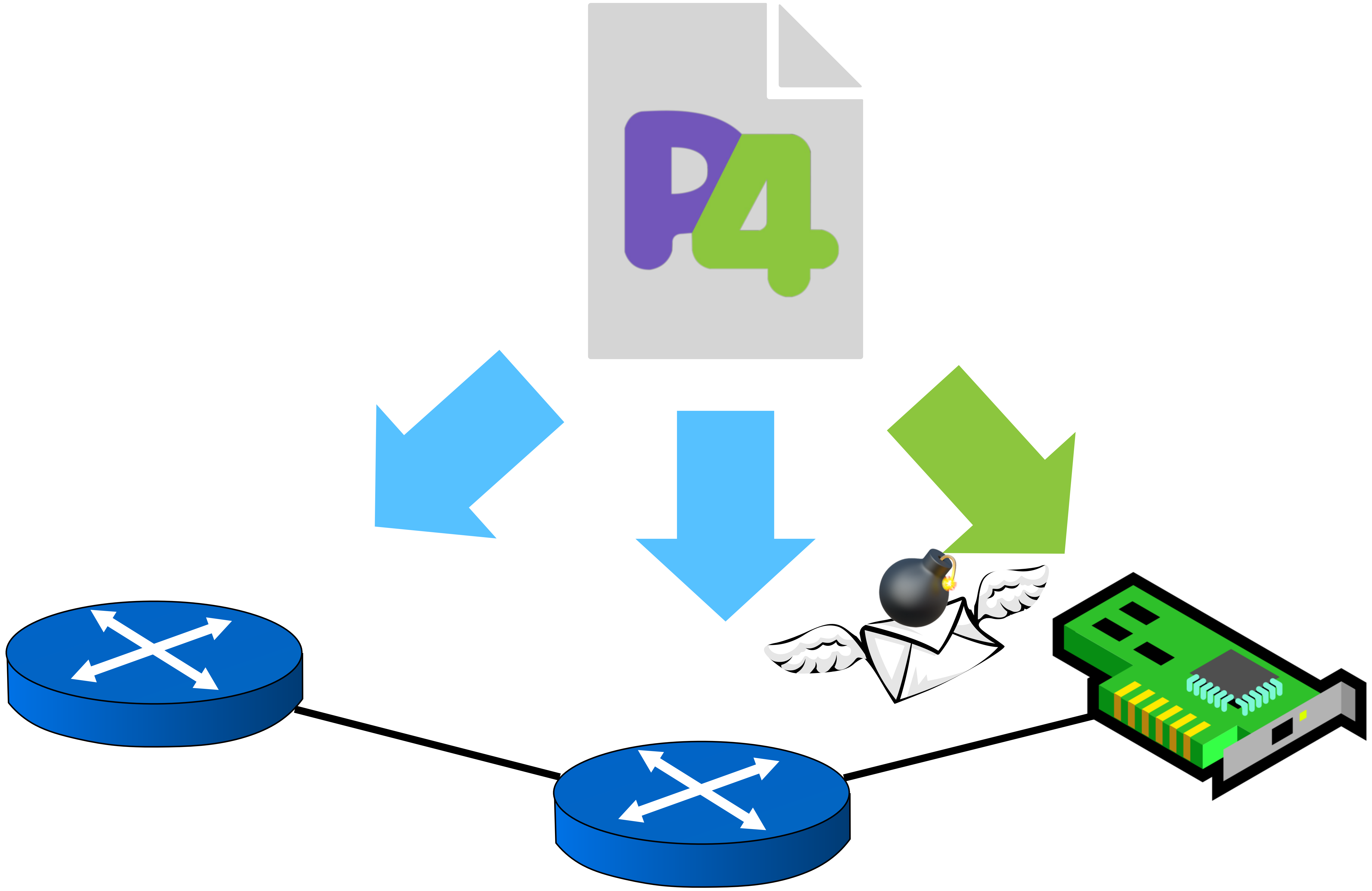
https://www.cs.dartmouth.edu/~sergey/langsec/occupy/



Protocol wire formats should be parsed in the same way by all network devices.

Security Implications

https://www.cs.dartmouth.edu/~sergey/langsec/occupy/



Protocol wire formats should be parsed in the same way by all network devices.

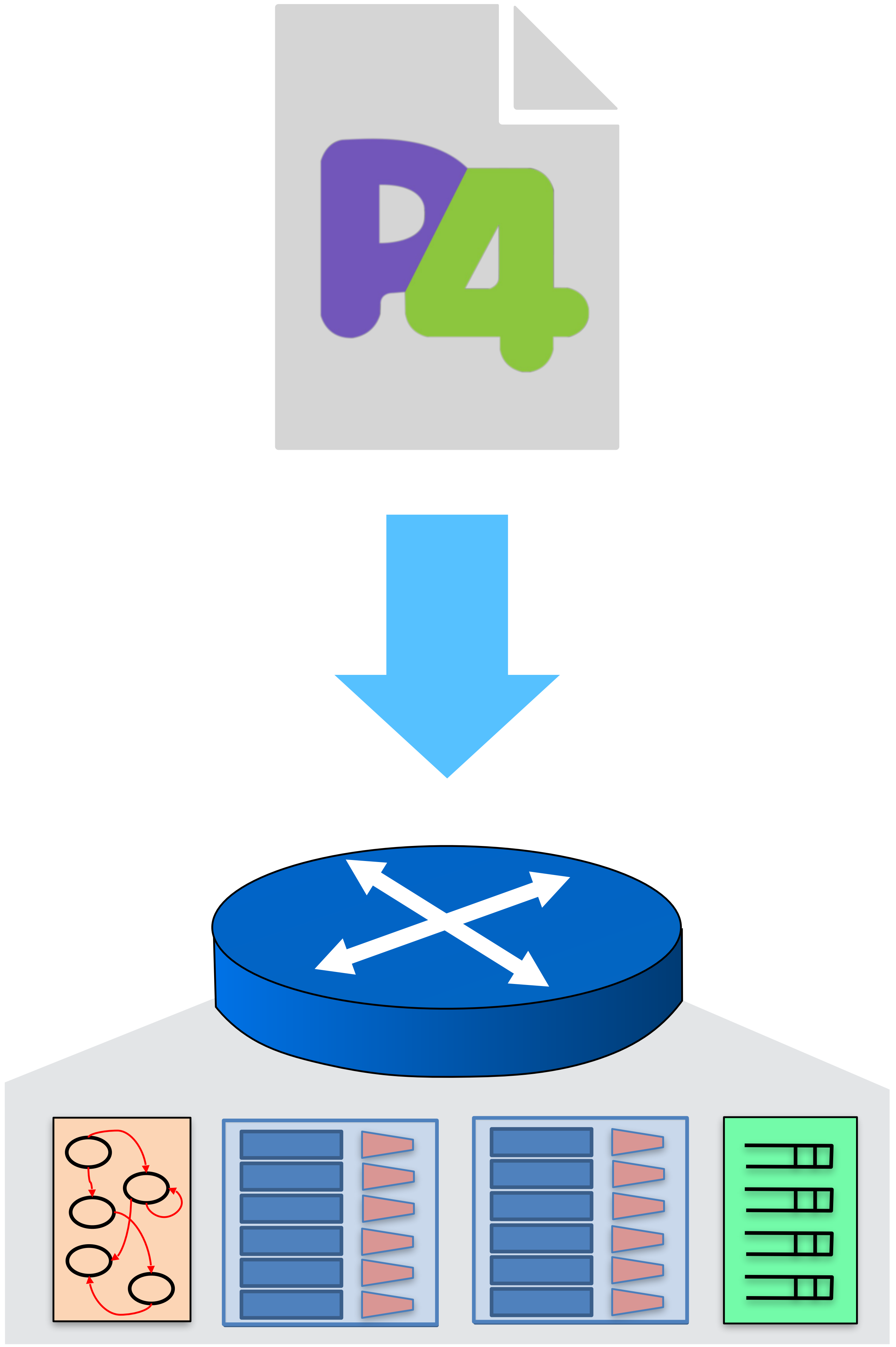
P4 Optimization is Not Optional

Precise throughput requirements

> "A 64 x 10Gb/s Ethernet switch must parse one billion packets per second"
> [Gibb et al. 2013]

Non-negotiable resource limits

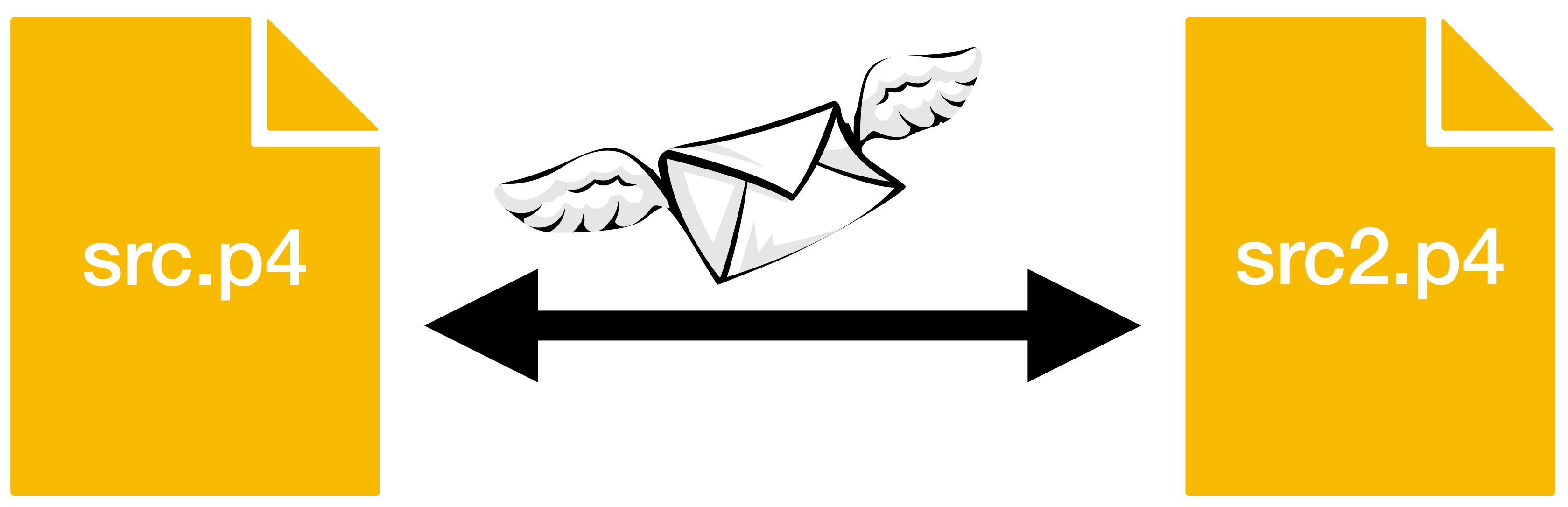
> "Unlike register allocation, there is no option to spill to memory..."
> [Jose et al. 2015]



Equivalence in P4 compilation

Your P4 programs should parse the same way, ideally according to an RFC or spec.

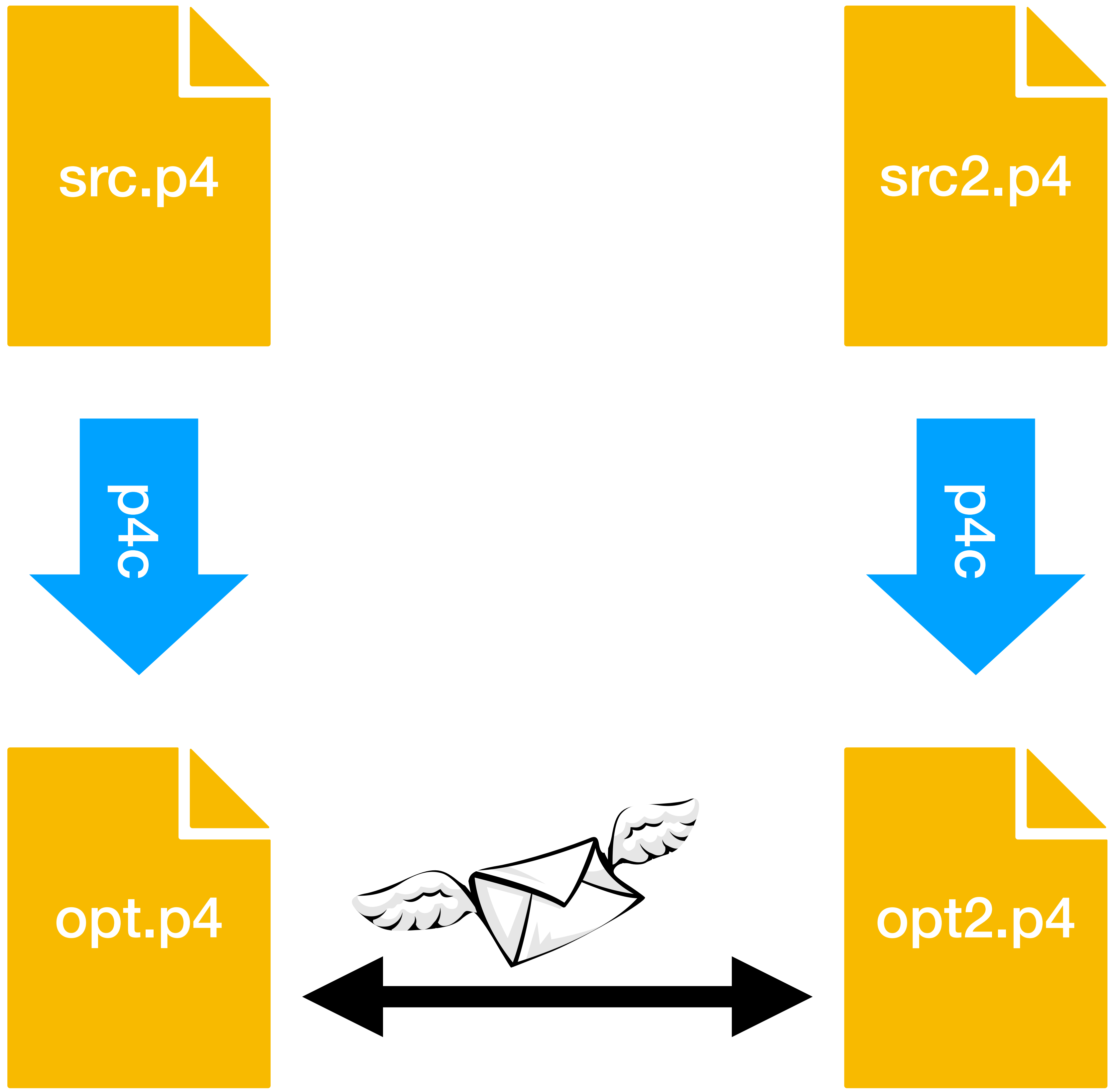
The P4 compiler should preserve parser behavior even as it optimizes them for throughput and resource usage.



Equivalence in P4 compilation

Your P4 programs should parse the same way, ideally according to an RFC or spec.

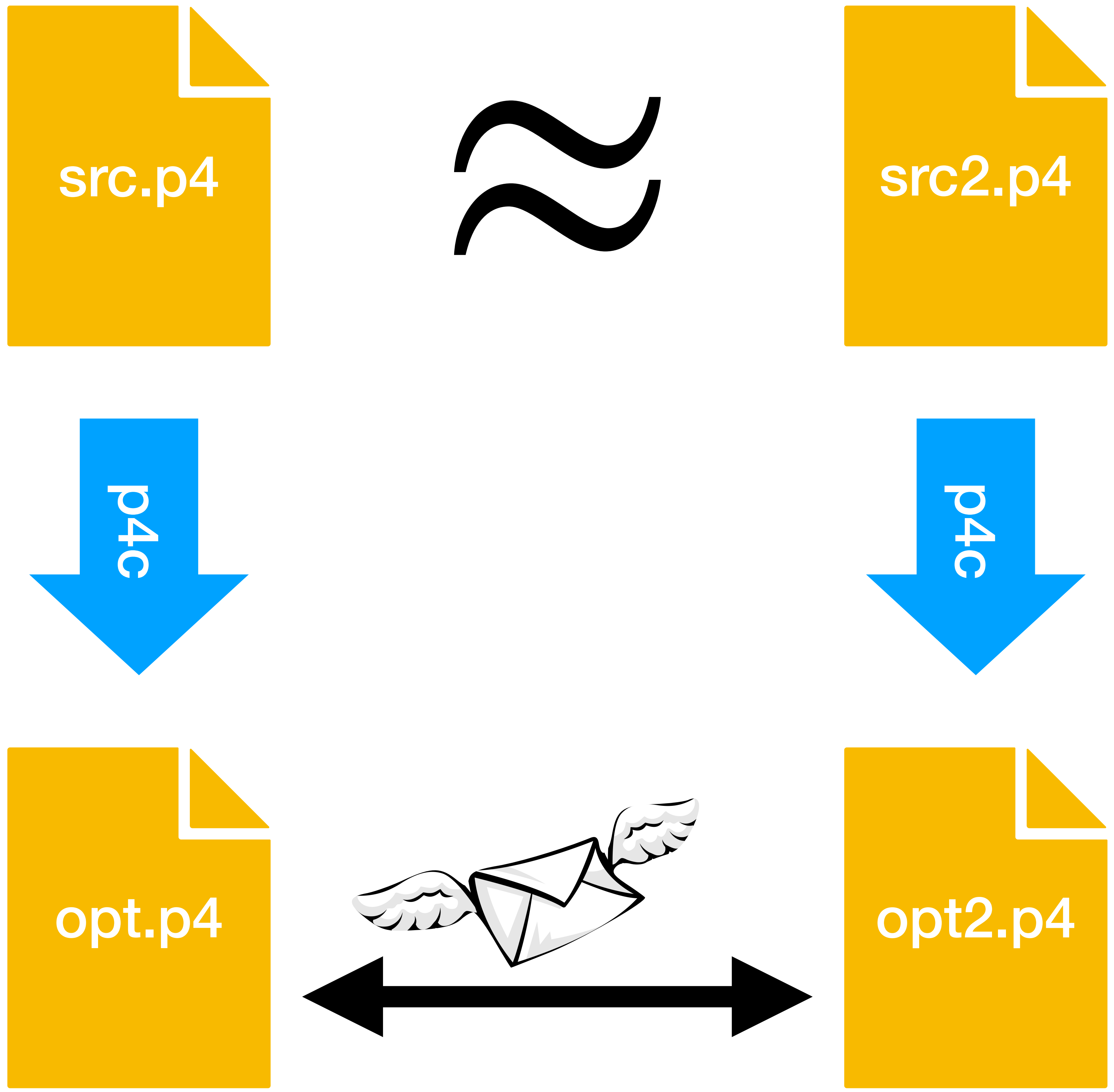
The P4 compiler should preserve parser behavior even as it optimizes them for throughput and resource usage.



Equivalence in P4 compilation

Your P4 programs should parse the same way, ideally according to an RFC or spec.

The P4 compiler should preserve parser behavior even as it optimizes them for throughput and resource usage.



Proving Equivalence for Parsers

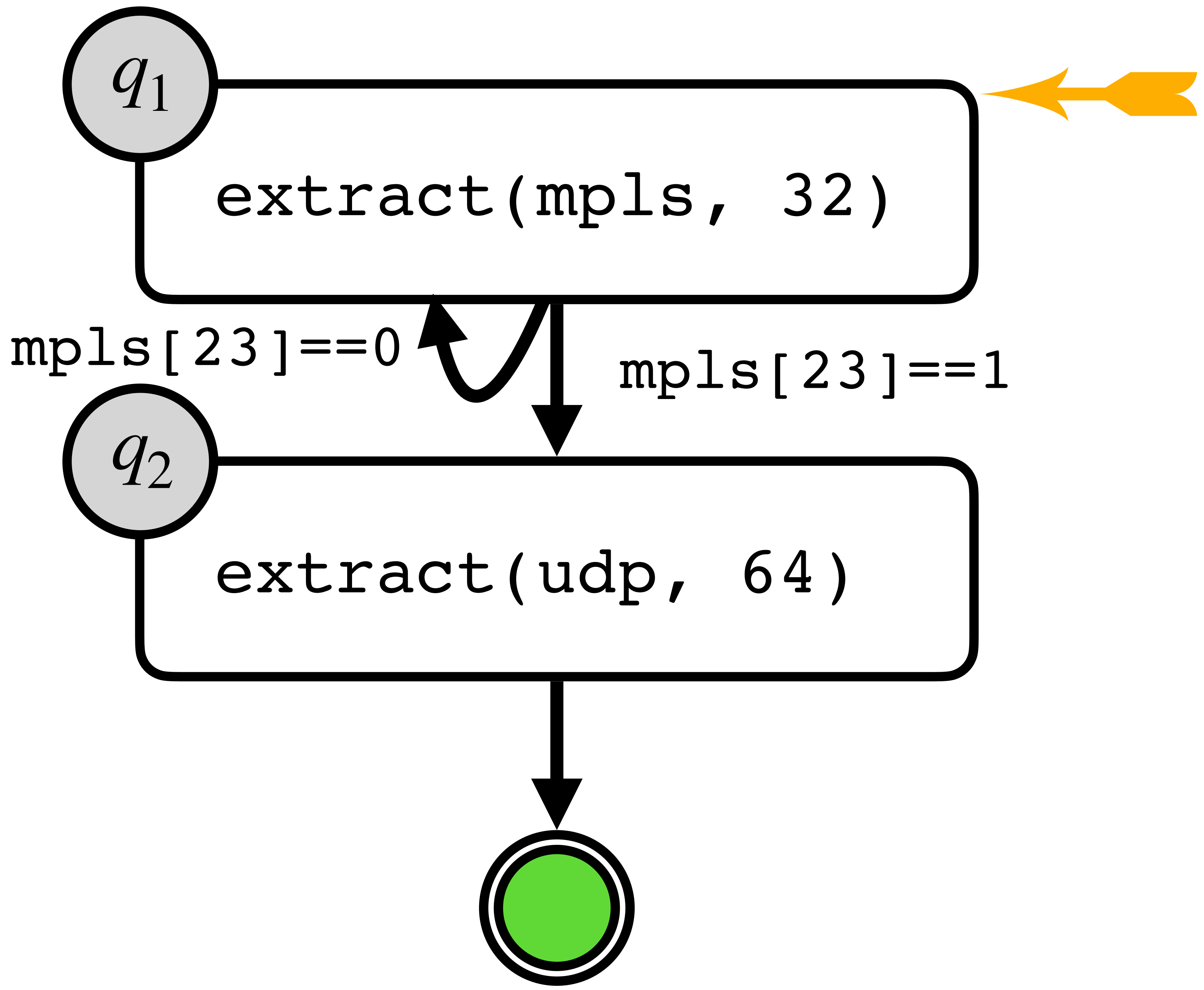
Useful when verifying hand-optimized code

$$\frac{\{P\}\, q\, \{Q\} \qquad \vdash p \equiv q}{\{P\}\, p\, \{Q\}}\ \textsc{Rewrite}$$

Useful for translation validation

```
let q := opt p in
if ⊢ p ≡ q then Ok(q)
else Err "miscompiled -___-"
```

Example: Parsing MPLS



Packet

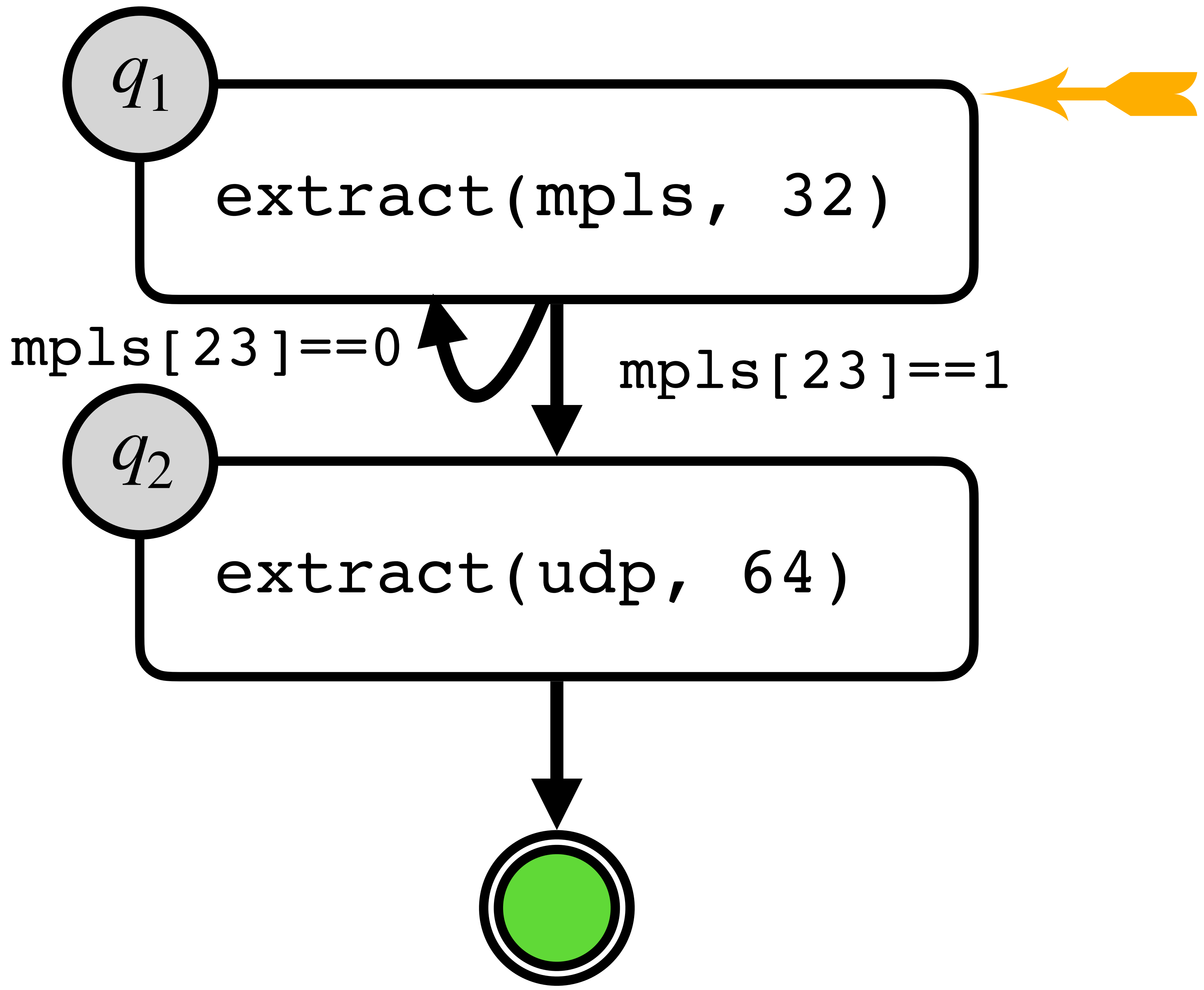
```
35 FD 00 9D 12 58 01 70
91 D1 A5 94 29 DA FA 7B
```

Store

```
mpls = 00 00 00 00
 udp = 00 00 00 00 00 00 00 00
```

To implement a parser in P4, programmers write state machines like this one.

Parsing MPLS with a P4 Automaton



Packet

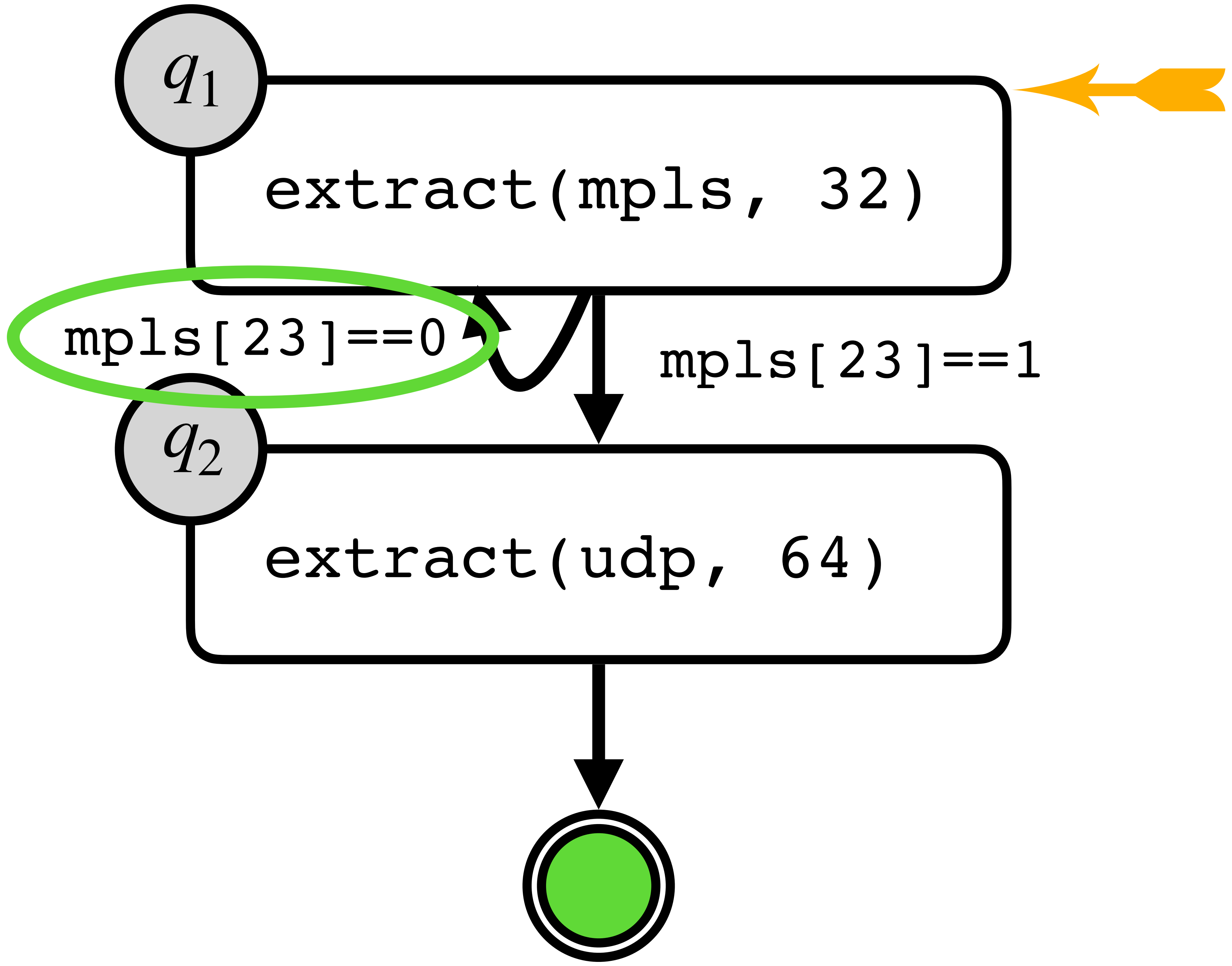
```
            12 58 01 70
91 D1 A5 94 29 DA FA 7B
```

Store

```
mpls = 35 FD 00 9D
 udp = 00 00 00 00 00 00 00 00
```

To implement a parser in P4, programmers write state machines like this one.

Parsing MPLS with a P4 Automaton



Packet

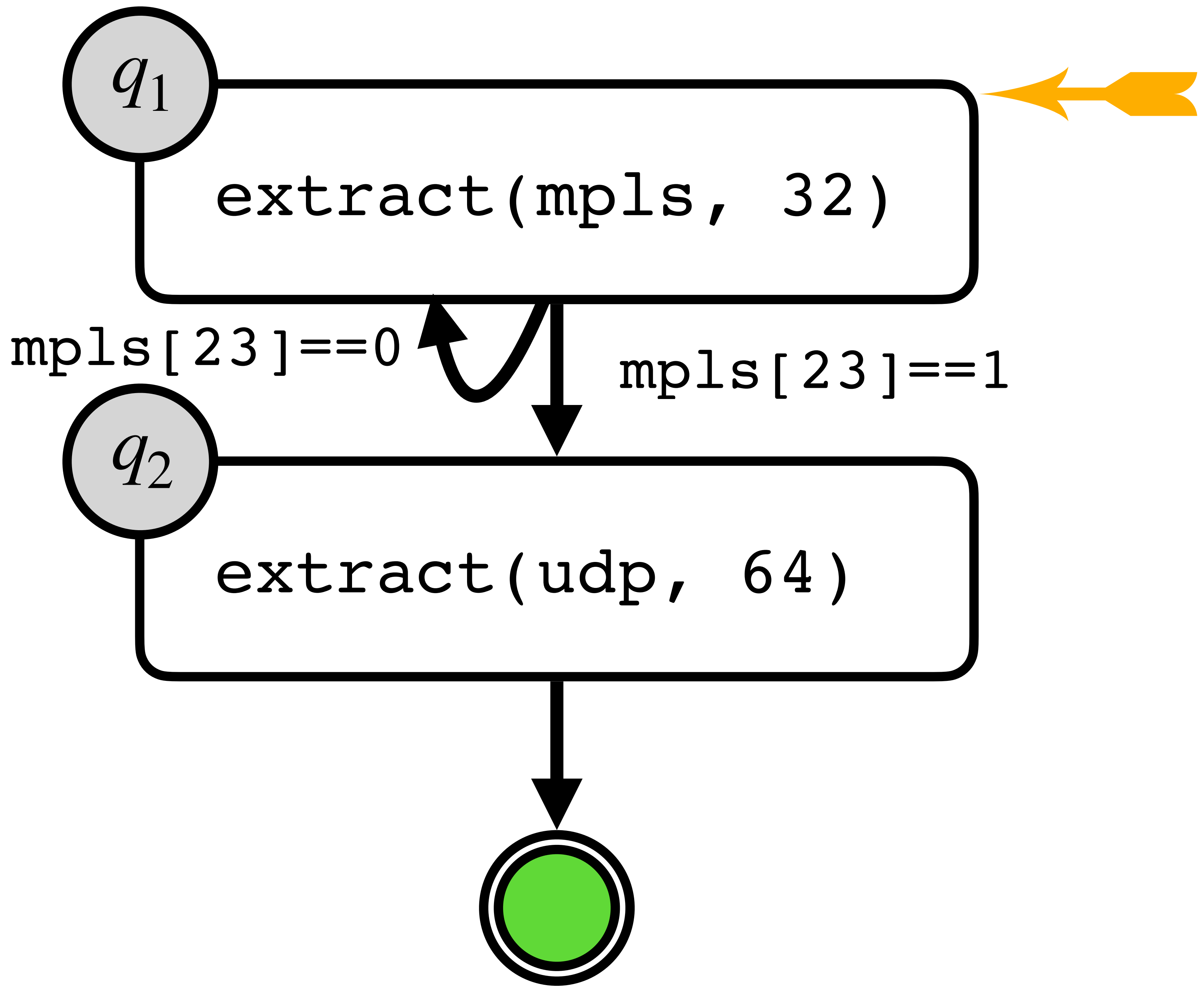
```
            12 58 01 70
91 D1 A5 94 29 DA FA 7B
```

Store

```
mpls = 35 FD 00 9D
 udp = 00 00 00 00 00 00 00 00
```

To implement a parser in P4, programmers write state machines like this one.

Parsing MPLS with a P4 Automaton



Packet

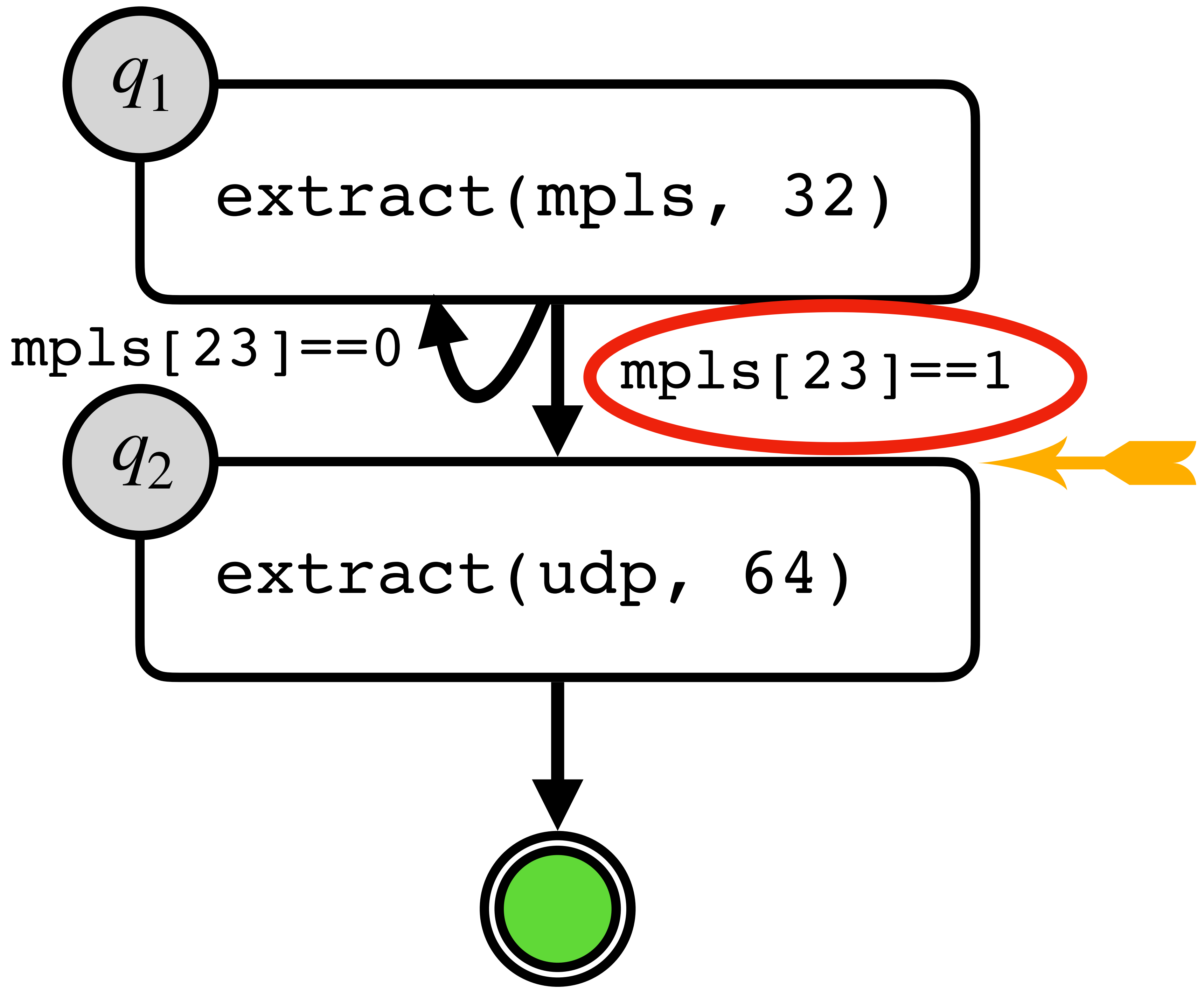
```
91 D1 A5 94 29 DA FA 7B
```

Store

```
mpls = 12 58 01 70
 udp = 00 00 00 00 00 00 00 00
```

To implement a parser in P4, programmers write state machines like this one.

Parsing MPLS with a P4 Automaton



Packet

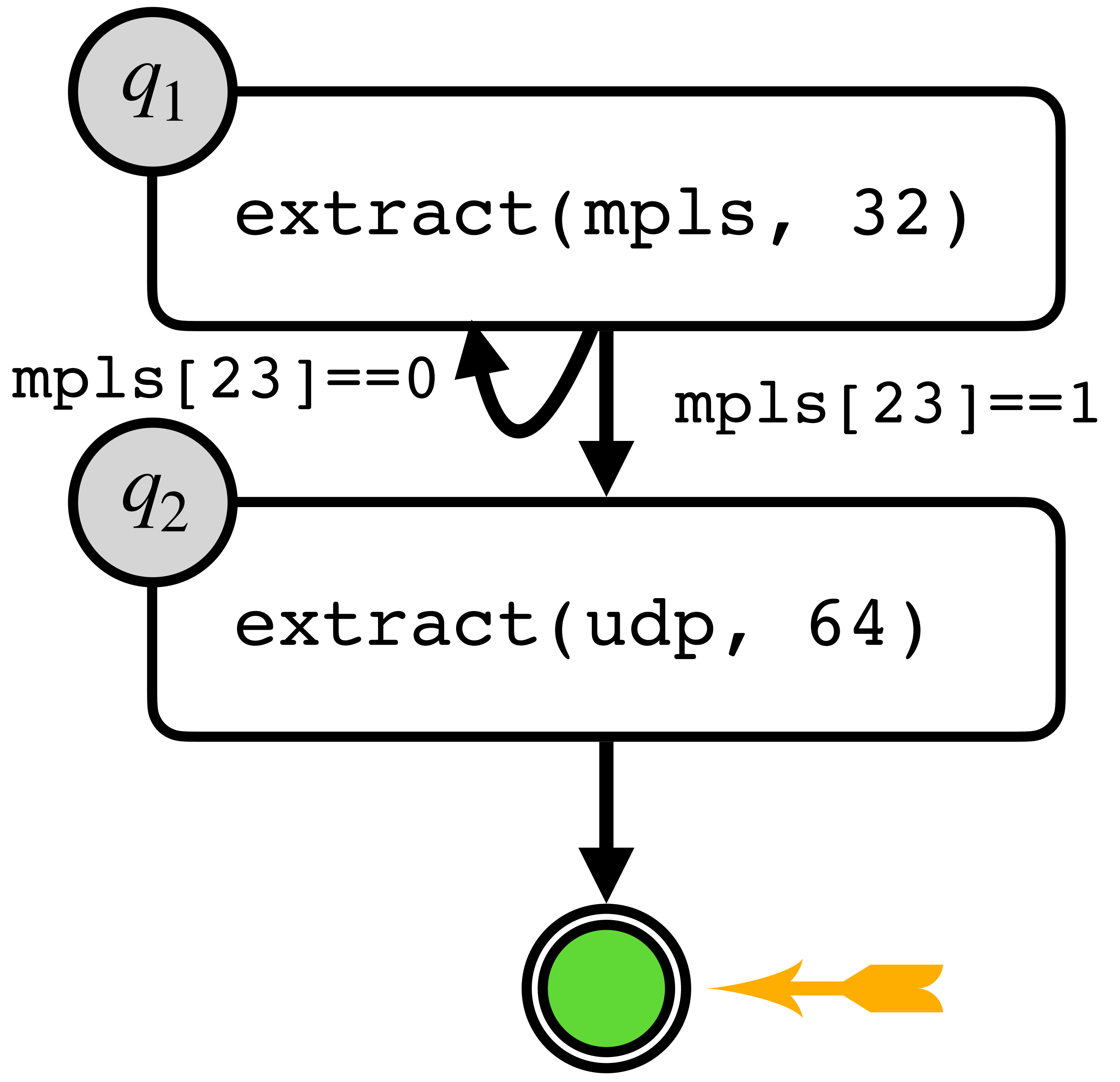
```
91 D1 A5 94 29 DA FA 7B
```

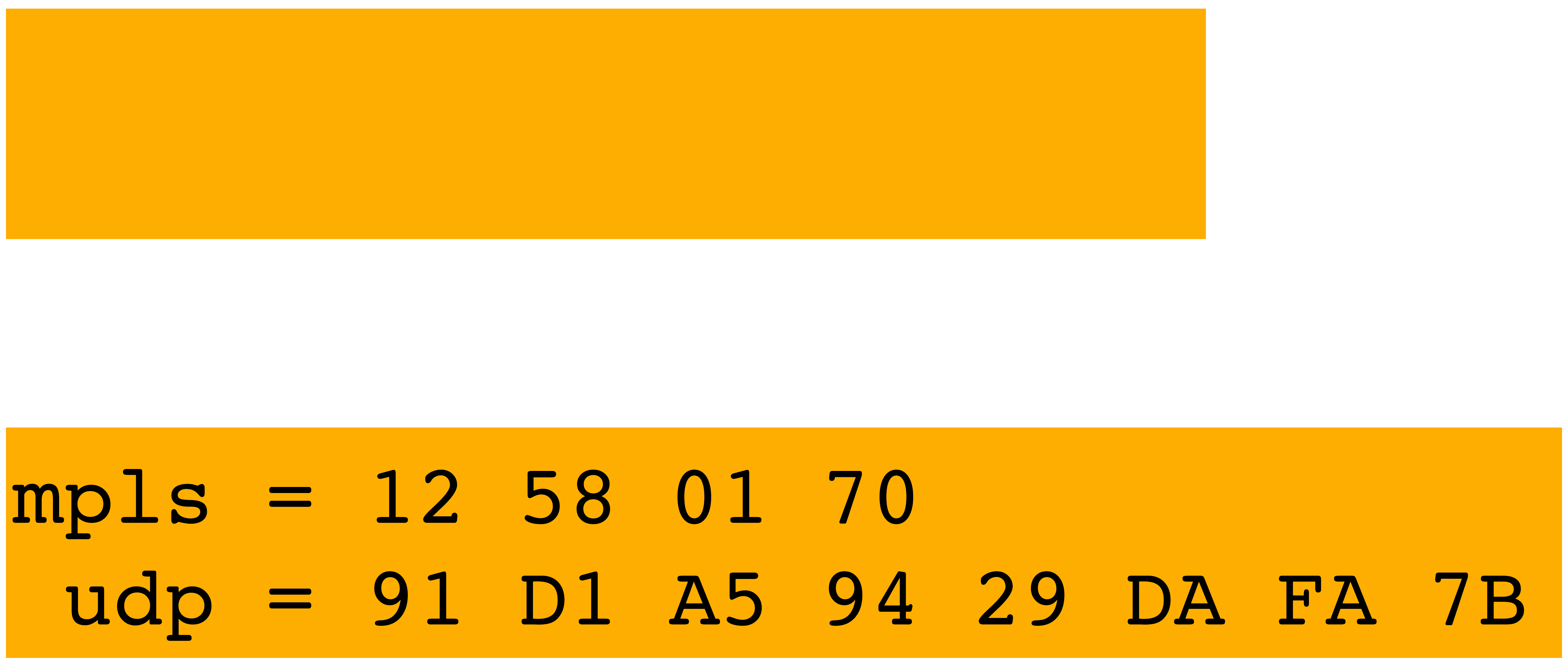
Store

```
mpls = 12 58 01 70
 udp = 00 00 00 00 00 00 00 00
```

To implement a parser in P4, programmers write state machines like this one.

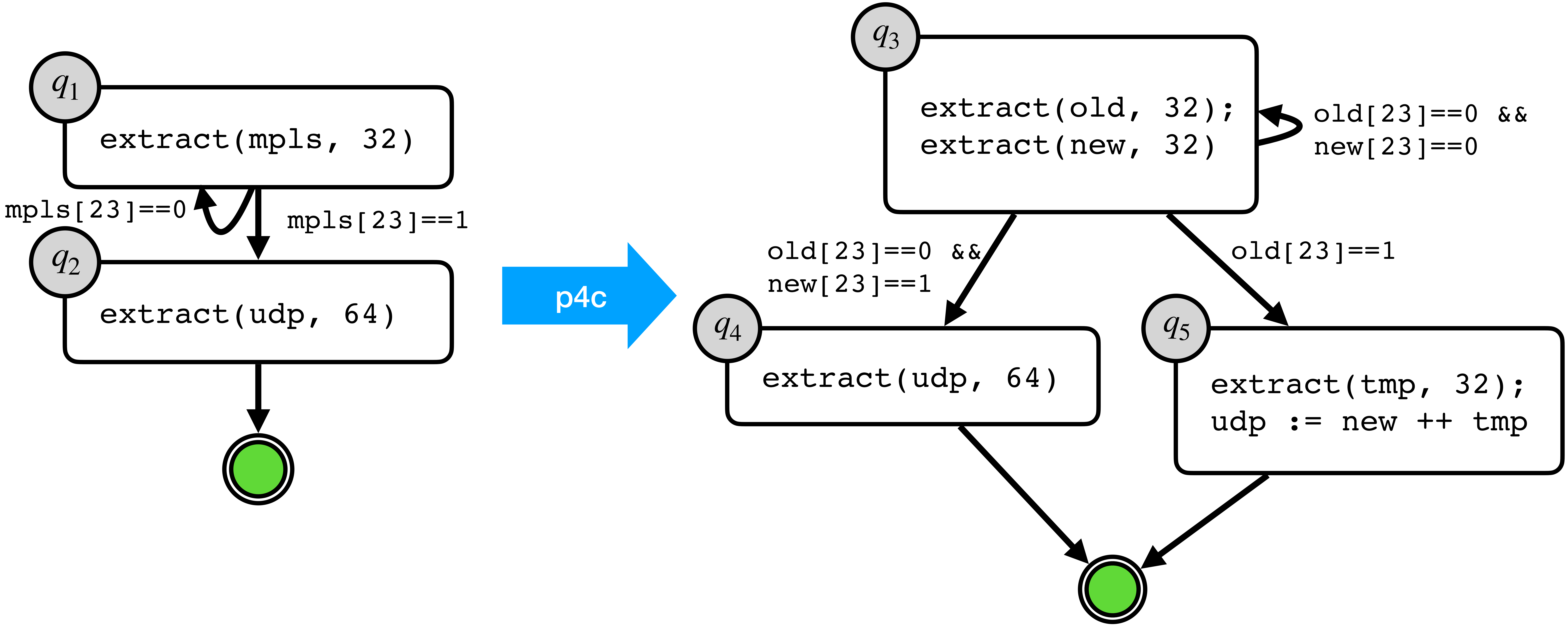
Parsing MPLS with a P4 Automaton



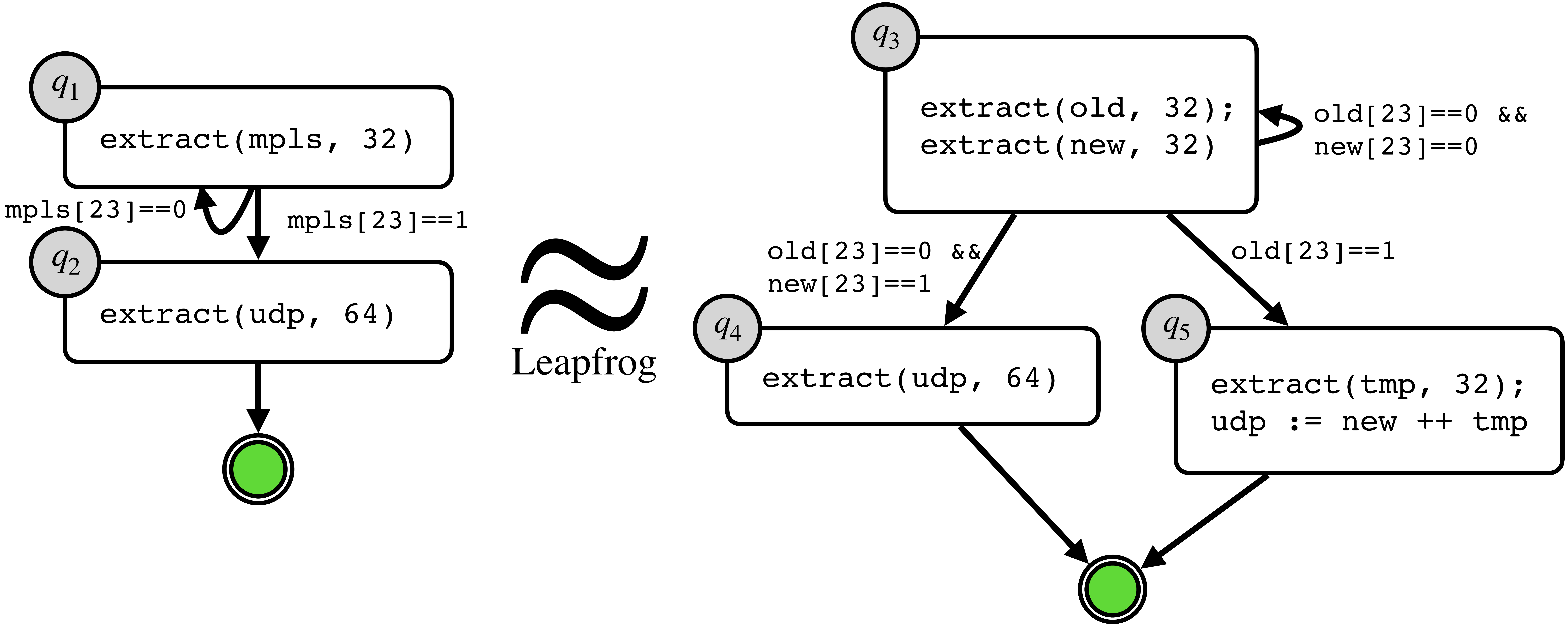


To implement a parser in P4, programmers write state machines like this one.

Loop Unrolling Optimization

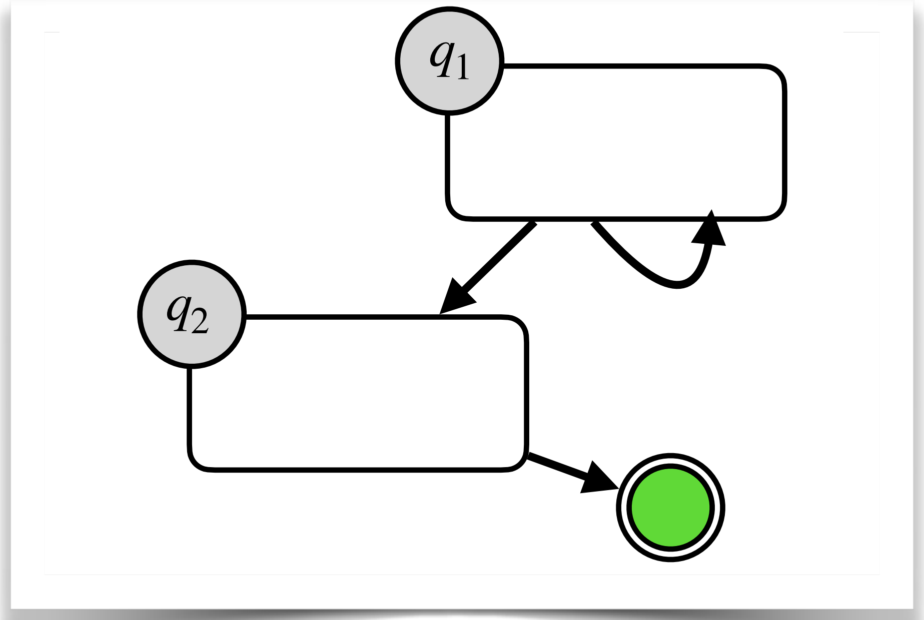


Translation Validation



Leapfrog

P4 Automata (P4A)



- Syntax
- Semantics
- Equivalence

Algorithm

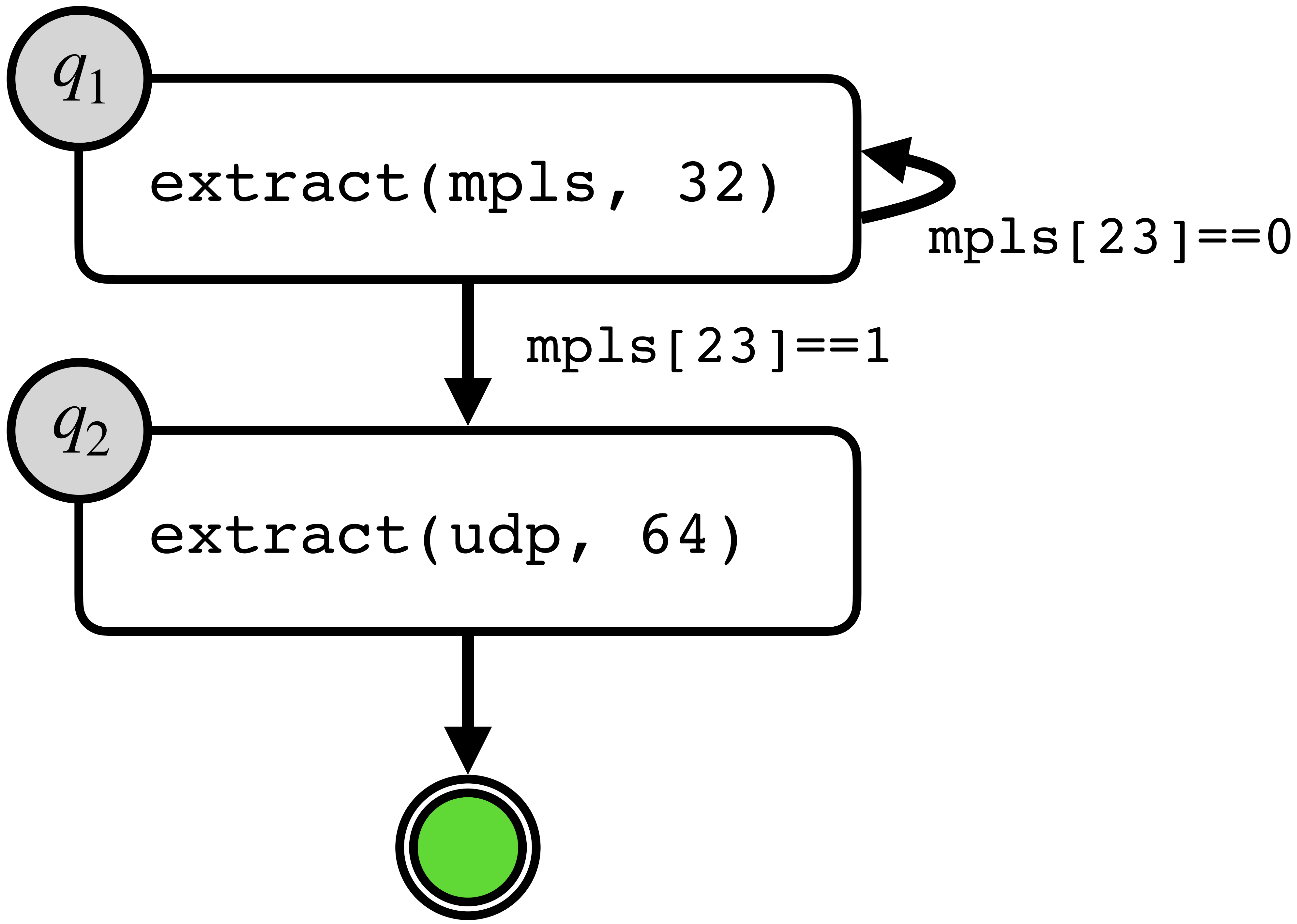
$$A \approx B$$

- Symbolic bisimulations
- Algorithm for finding symbolic bisimulations
- Bisimulations with leaps
- More optimizations (see paper)

Coq Implementation

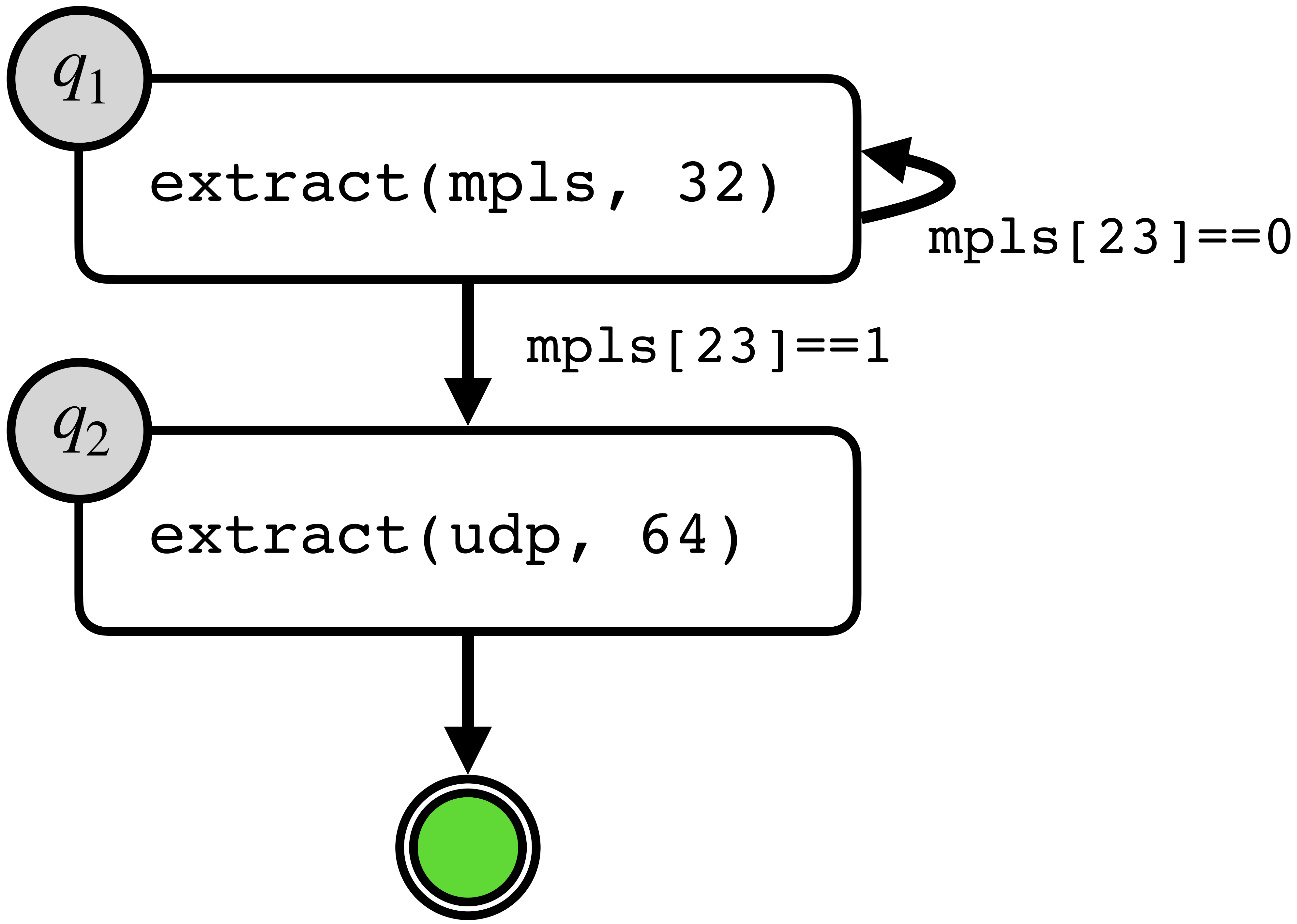
- Semi-decision procedure
- Coq certificates of equivalence
- SMT interface
- Evaluation on a range of examples

P4A: Syntax



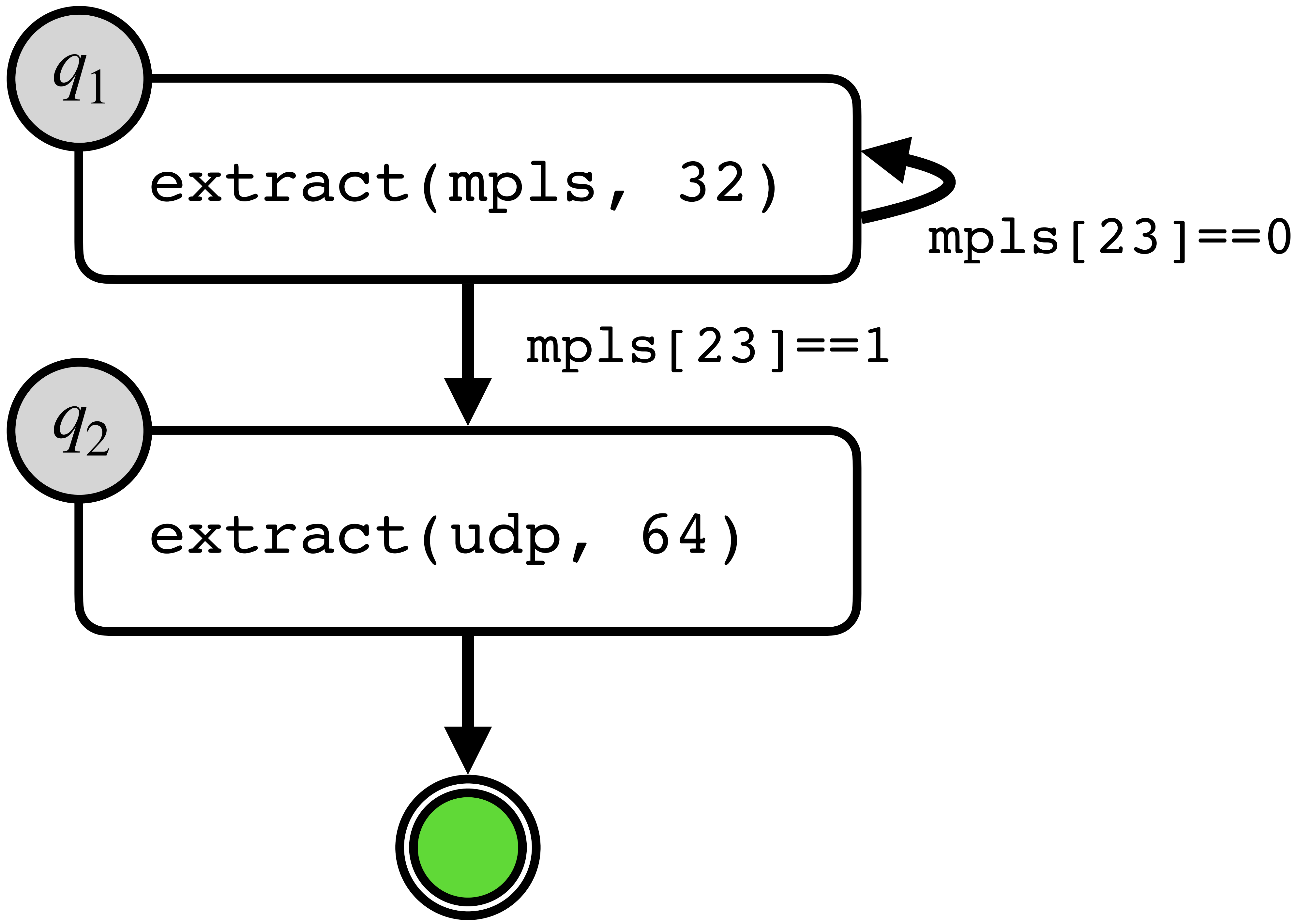
h	$\in$	H	header names
n	$\in$	$\mathbb{N}$	natural numbers
bv	$\in$	$\{0,1\}^*$	bitvector
e	$::=$	h	headers
	$\mid$	bv	bitvectors
	$\mid$	$e[n_1 : n_2]$	bitslices
	$\mid$	$e_1 +\!+\ e_2$	concatenation

P4A: Syntax



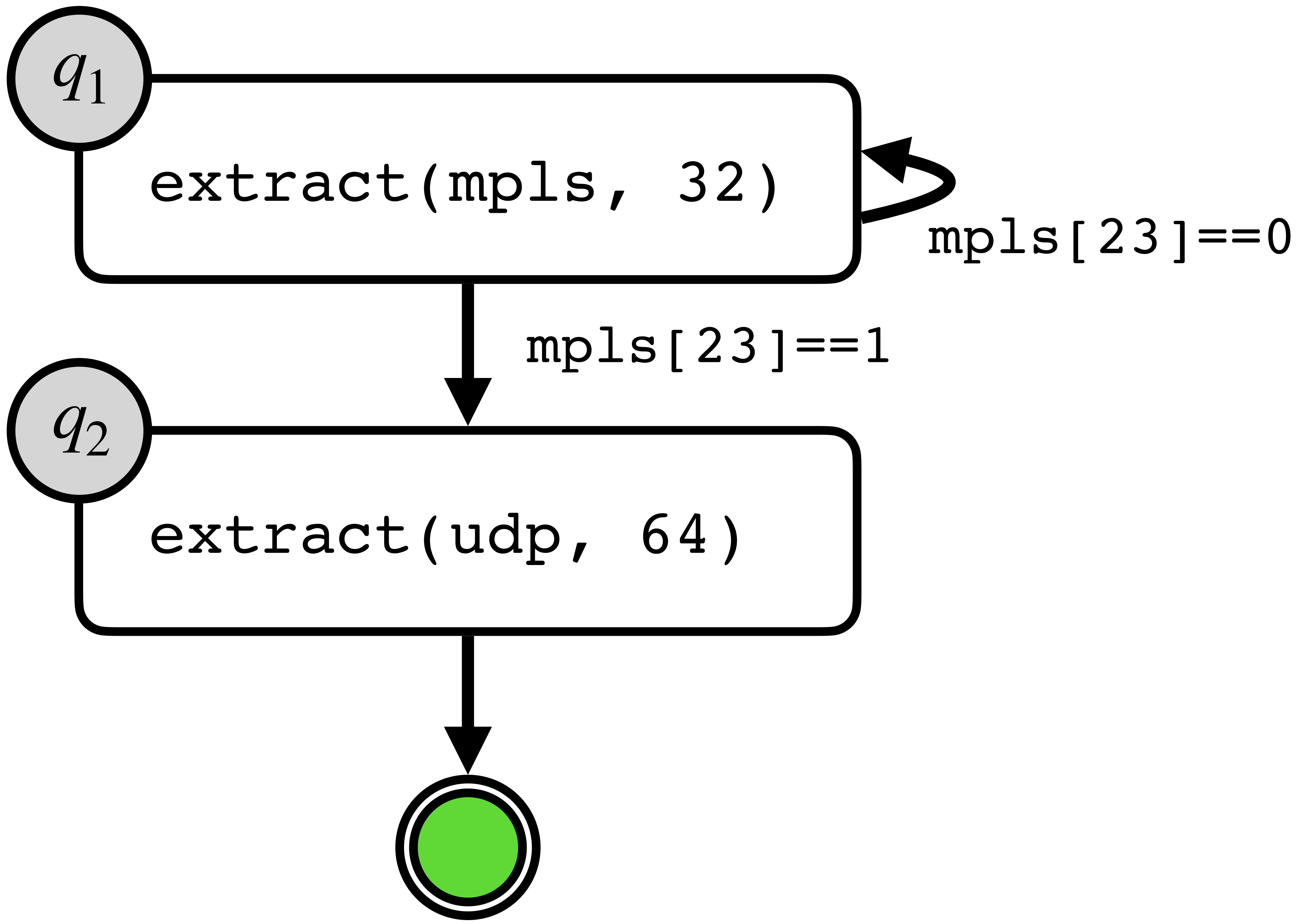
h	$\in$	H	header names
n	$\in$	$\mathbb{N}$	natural numbers
bv	$\in$	$\{0, 1\}^*$	bitvector
e	$::=$	h	headers
	$\mid$	bv	bitvectors
	$\mid$	$e[n_1 : n_2]$	bitslices
	$\mid$	$e_1 +\!\!+\ e_2$	concatenation
pat	$::=$	bv	exact match
	$\mid$	$\_$	wildcard
q	$\in$	$Q \cup \{\text{accept}, \text{reject}\}$	state names
c	$::=$	$\overline{pat} \Rightarrow q$	select case
tz	$::=$	$\texttt{goto}(q)$	direct
	$\mid$	$\texttt{select}(\overline{e})\{\overline{c}\}$	select

P4A: Syntax



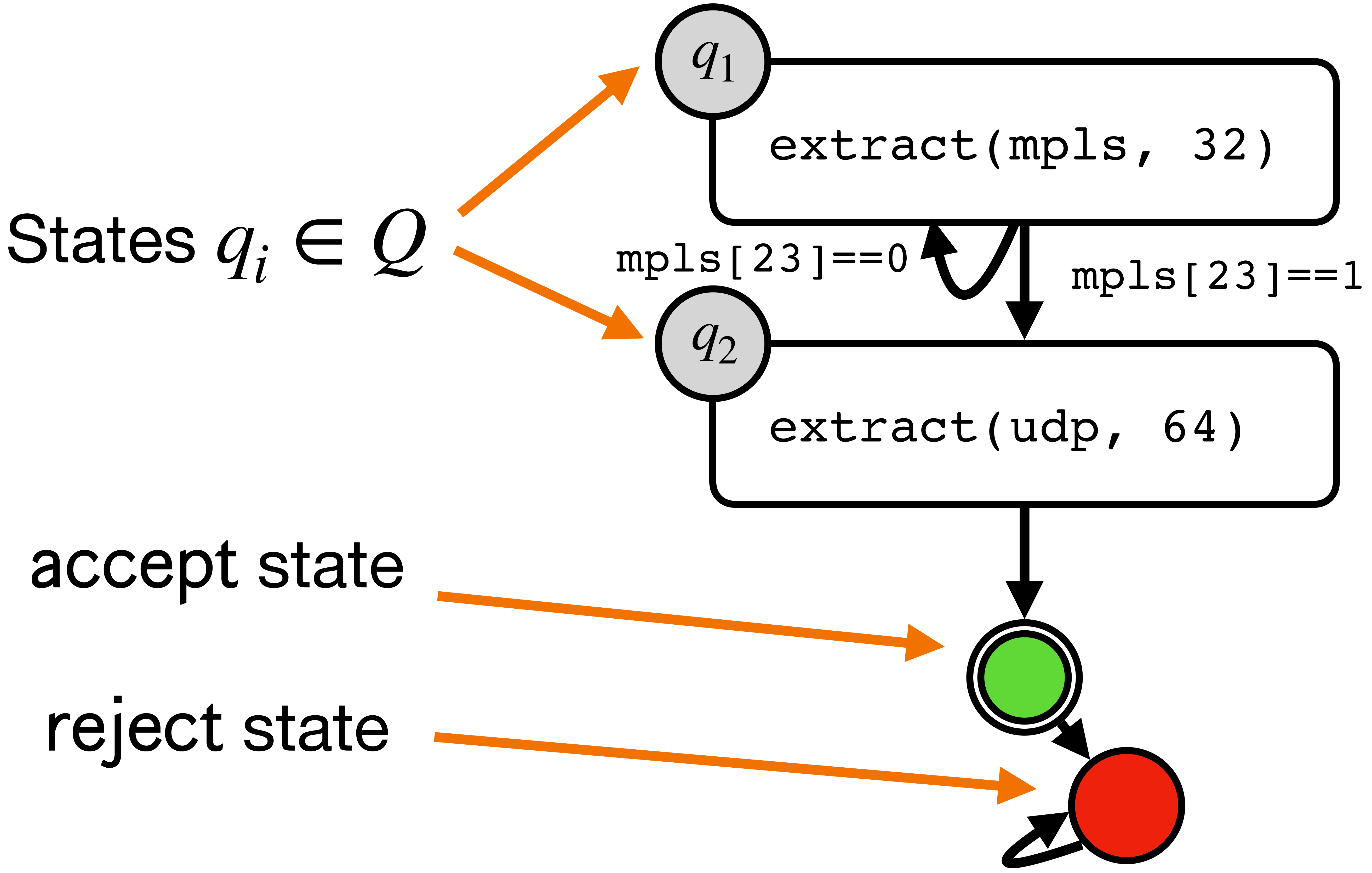
h	$\in$	H	header names
n	$\in$	$\mathbb{N}$	natural numbers
bv	$\in$	$\{0, 1\}^*$	bitvector
e	$::=$	h	headers
	$\mid$	bv	bitvectors
	$\mid$	$e[n_1 : n_2]$	bitslices
	$\mid$	$e_1 \mathbin{++} e_2$	concatenation
pat	$::=$	bv	exact match
	$\mid$	$\_$	wildcard
q	$\in$	$Q \cup \{\text{accept}, \text{reject}\}$	state names
c	$::=$	$\overline{pat} \Rightarrow q$	select case
tz	$::=$	$\texttt{goto}(q)$	direct
	$\mid$	$\texttt{select}(\overline{e})\{\overline{c}\}$	select
op	$::=$	$\texttt{extract}(h)$	extract
	$\mid$	$h := e$	assign
	$\mid$	$op_1;\ op_2$	sequence

P4A: Syntax

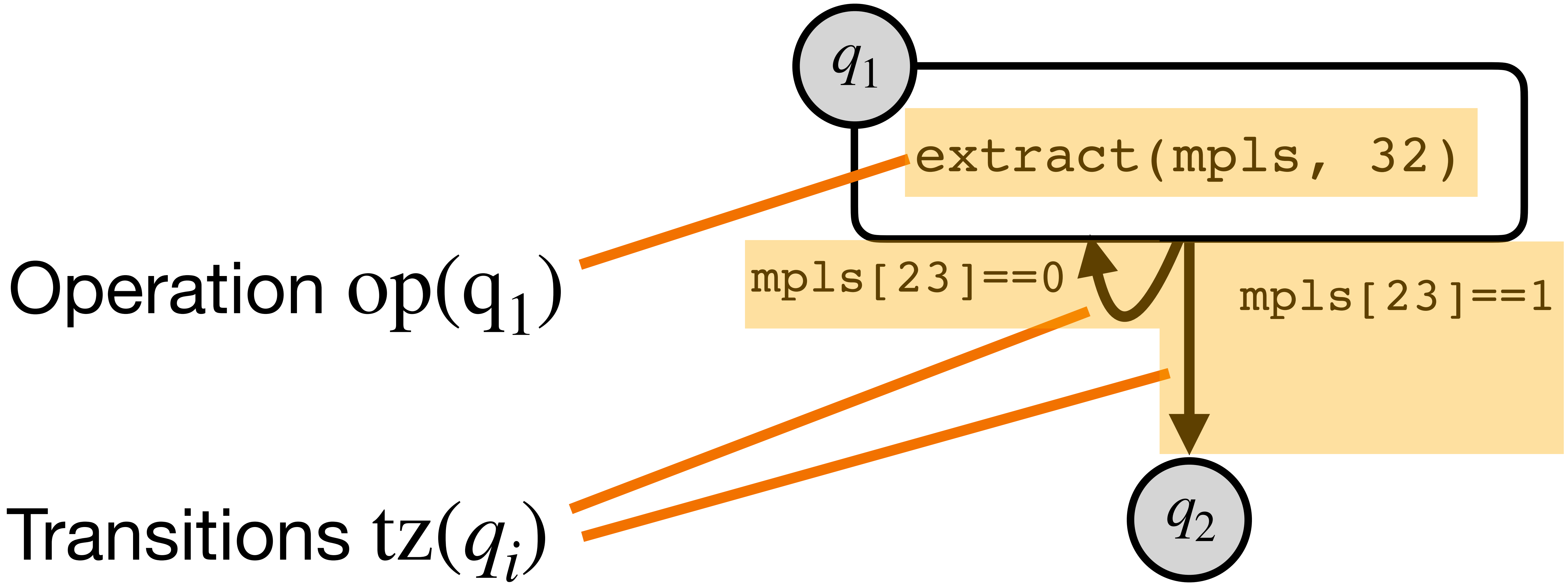


h	$\in$	H	header names
n	$\in$	$\mathbb{N}$	natural numbers
bv	$\in$	$\{0, 1\}^*$	bitvector
e	$::=$	h	headers
	$\mid$	bv	bitvectors
	$\mid$	$e[n_1 : n_2]$	bitslices
	$\mid$	$e_1 \mathbin{++} e_2$	concatenation
pat	$::=$	bv	exact match
	$\mid$	$\_$	wildcard
q	$\in$	$Q \cup \{\text{accept}, \text{reject}\}$	state names
c	$::=$	$\overline{pat} \Rightarrow q$	select case
tz	$::=$	$\texttt{goto}(q)$	direct
	$\mid$	$\texttt{select}(\overline{e})\{\overline{c}\}$	select
op	$::=$	$\texttt{extract}(h)$	extract
	$\mid$	$h := e$	assign
	$\mid$	$op_1;\ op_2$	sequence
st	$::=$	$q\ \{op; tz\}$	states ($q \in Q$)
aut	$::=$	$\overline{st}$	P4 automaton

Anatomy of a P4A



Anatomy of a State



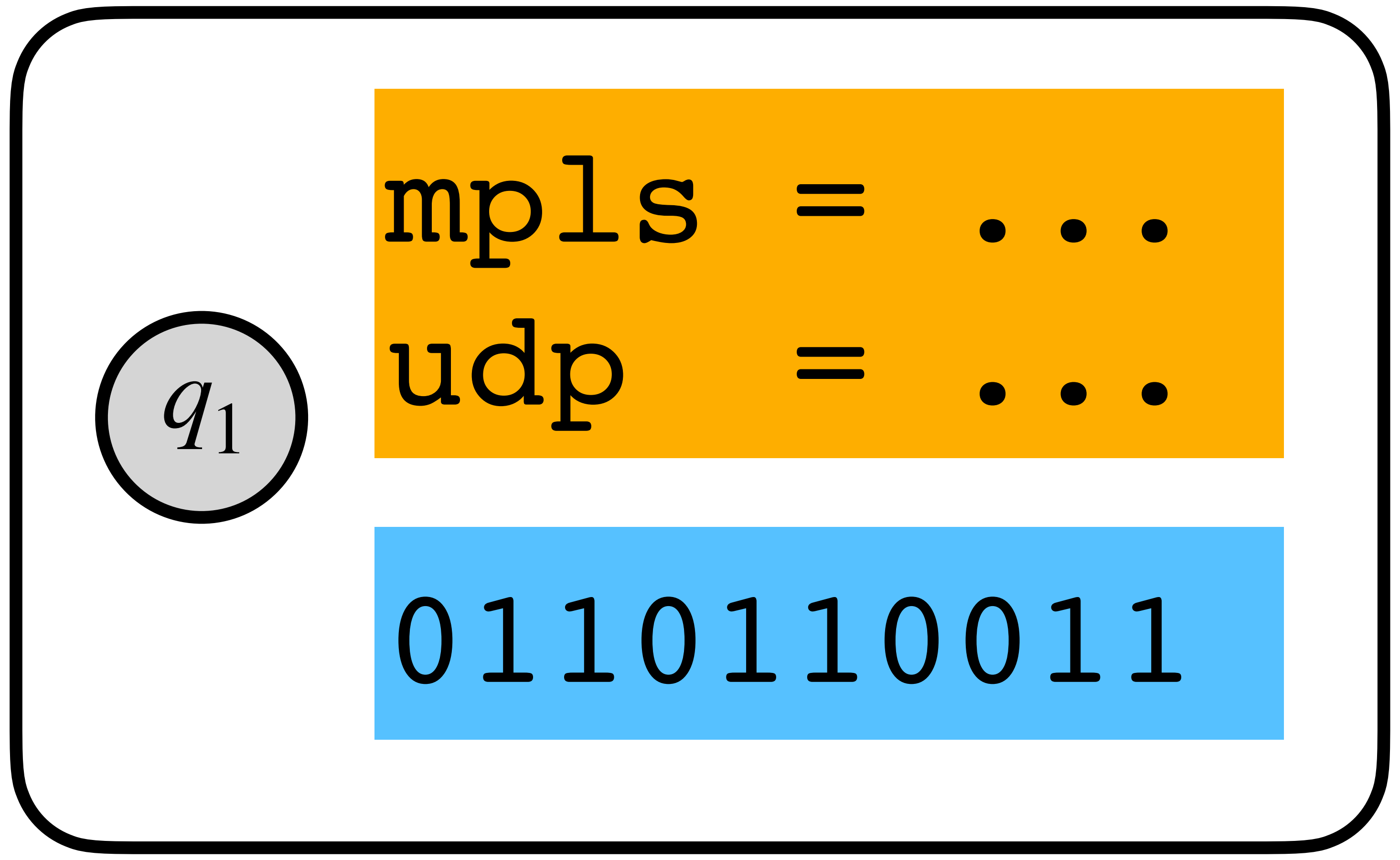
Semantics

P4A are really flowchart programs, not automata.

A finite automaton has

- C, a finite set of configurations
- $F \subseteq C$, a set of accepting ("final") configurations
- $\delta : C \times \{0,1\} \to C$, a transition function

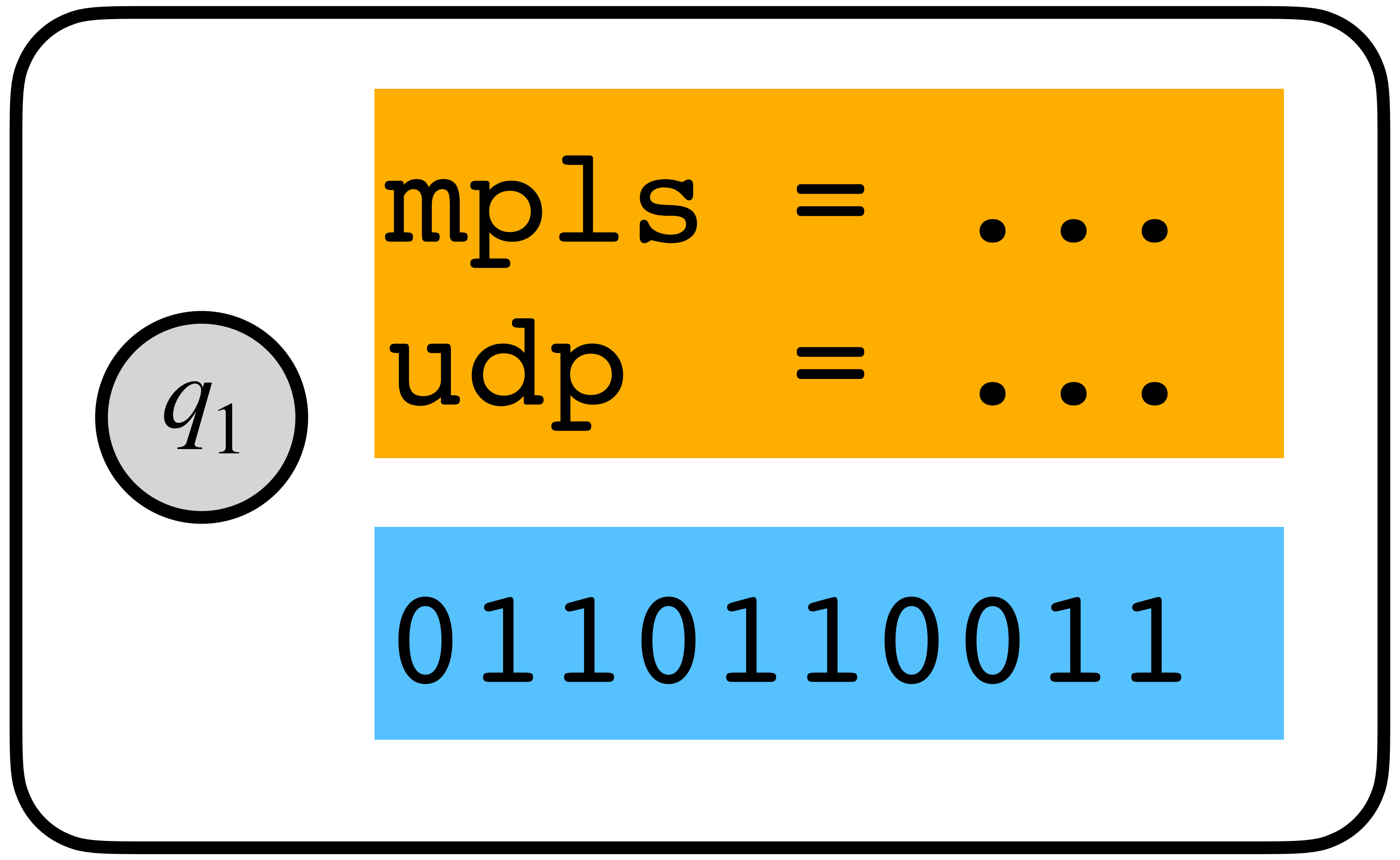
Automata Semantics: Configurations



A configuration is a tuple $\langle q, s, w \rangle$ with

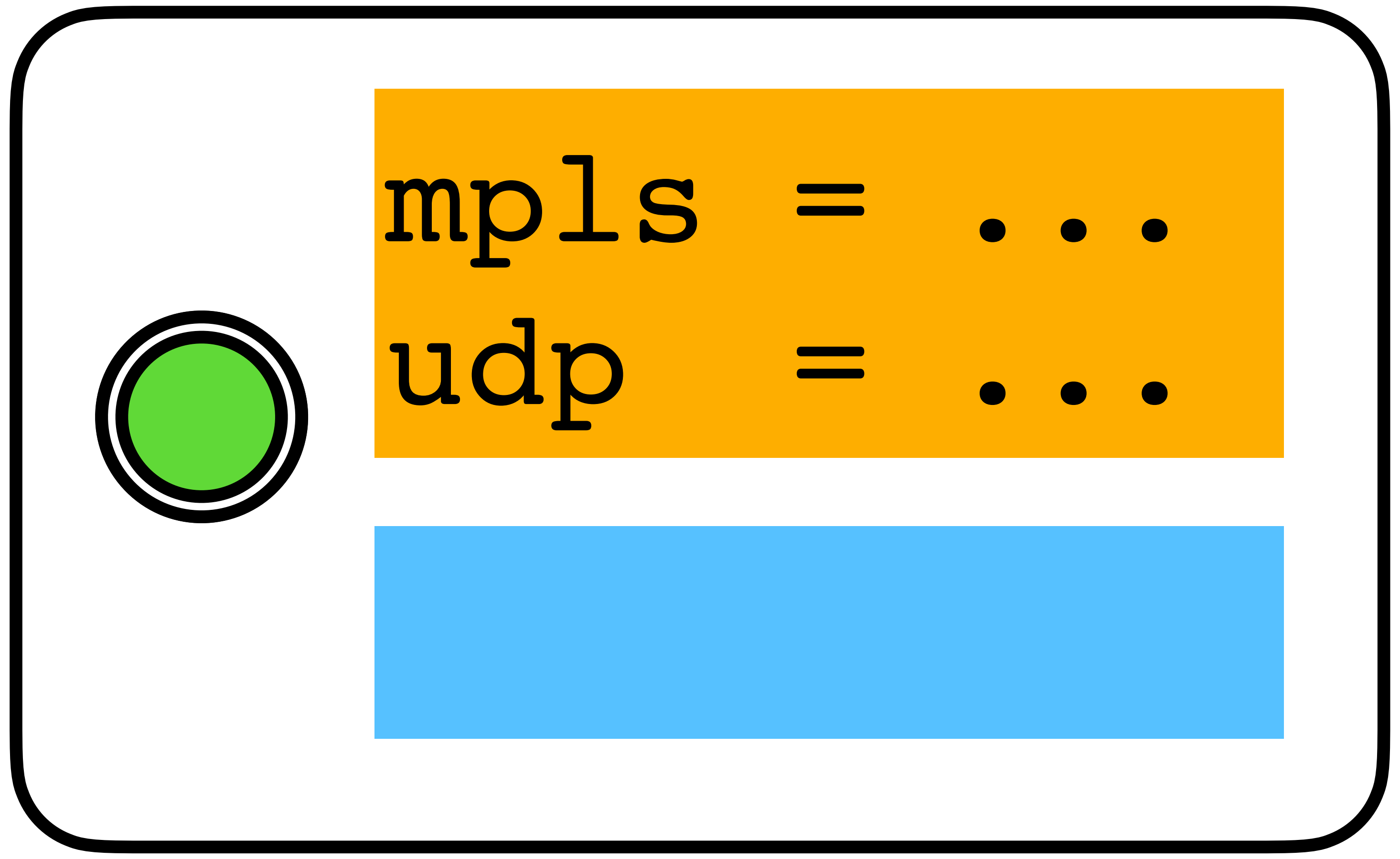
- A state $q \in Q \cup \{\text{accept, reject}\}$
- A store $s \in S$
- A buffer $w \in \{0,1\}^*$ with $|w| < |\text{op}(q)|$.

Automata Semantics: Configurations



A configuration is a tuple $\langle q, s, w \rangle$ with

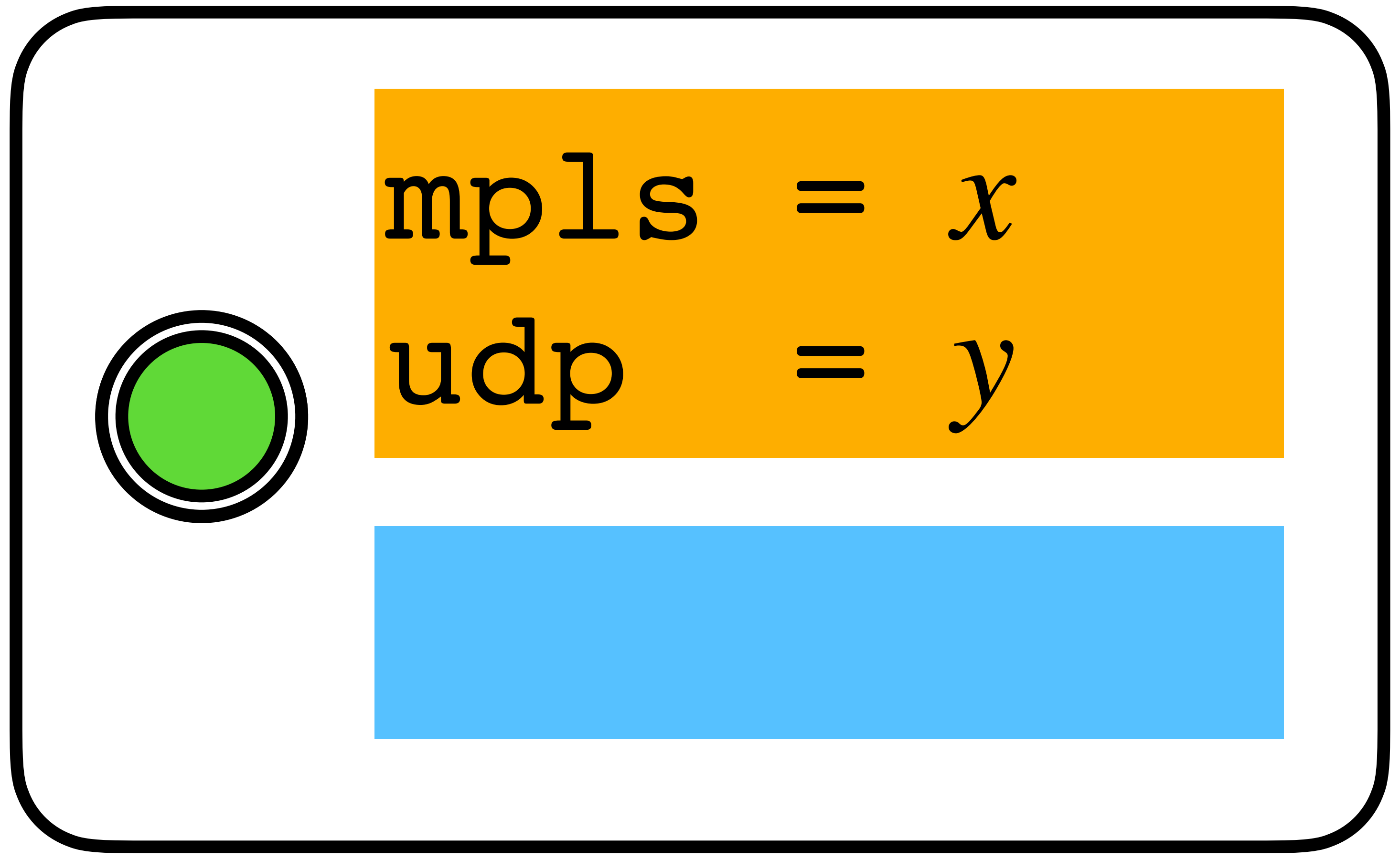
- A state $q \in Q \cup \{$accept, reject$\}$
- A store $s \in S$
- A buffer $w \in \{0,1\}^*$ with $|w| < |\mathrm{op}(q)|$.



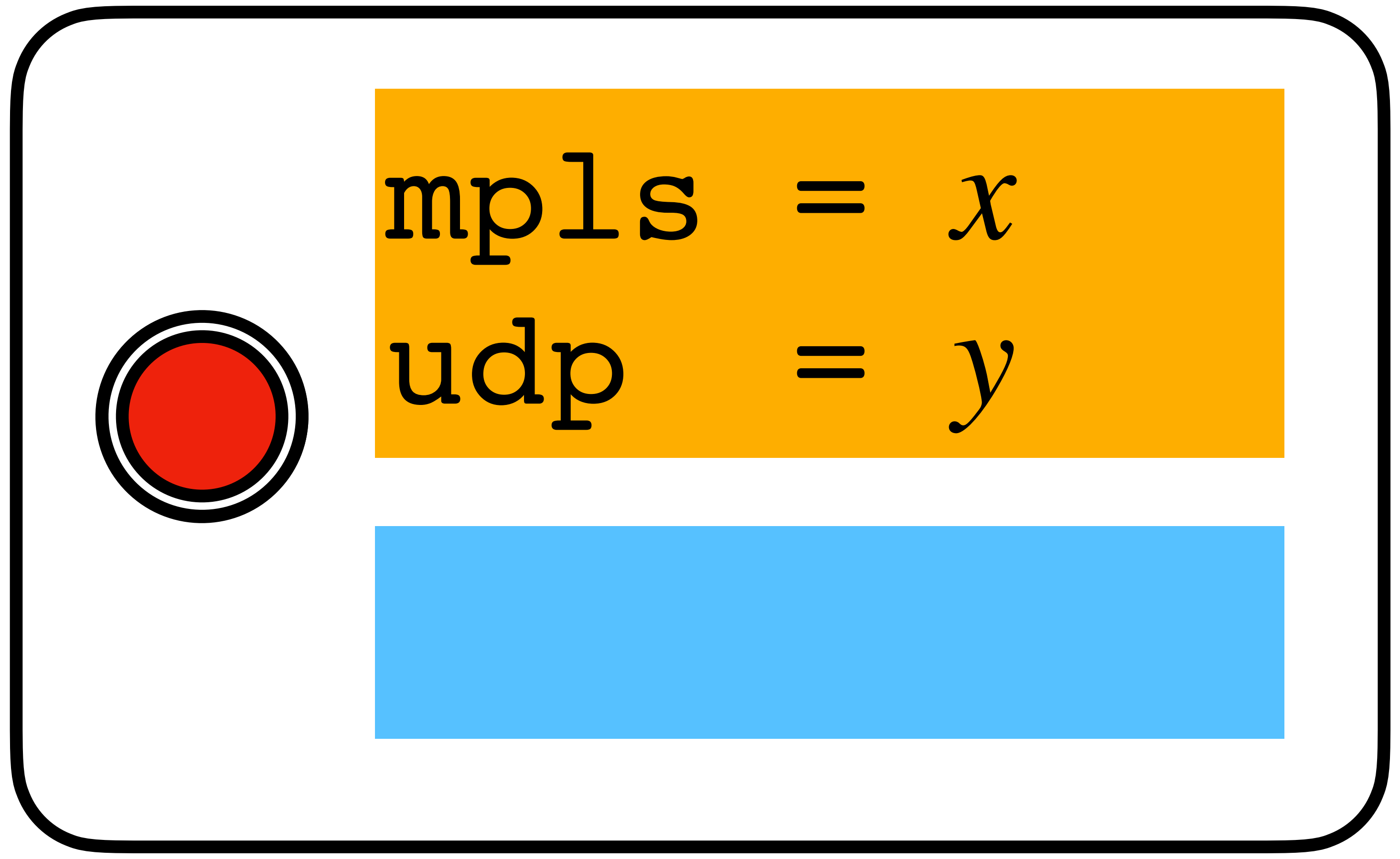
The final configurations are
$F = \{\langle q, s, \epsilon \rangle : q = \text{accept}\}$.

Steps

Defining a total function $\delta : C \times \{0,1\} \to C$.



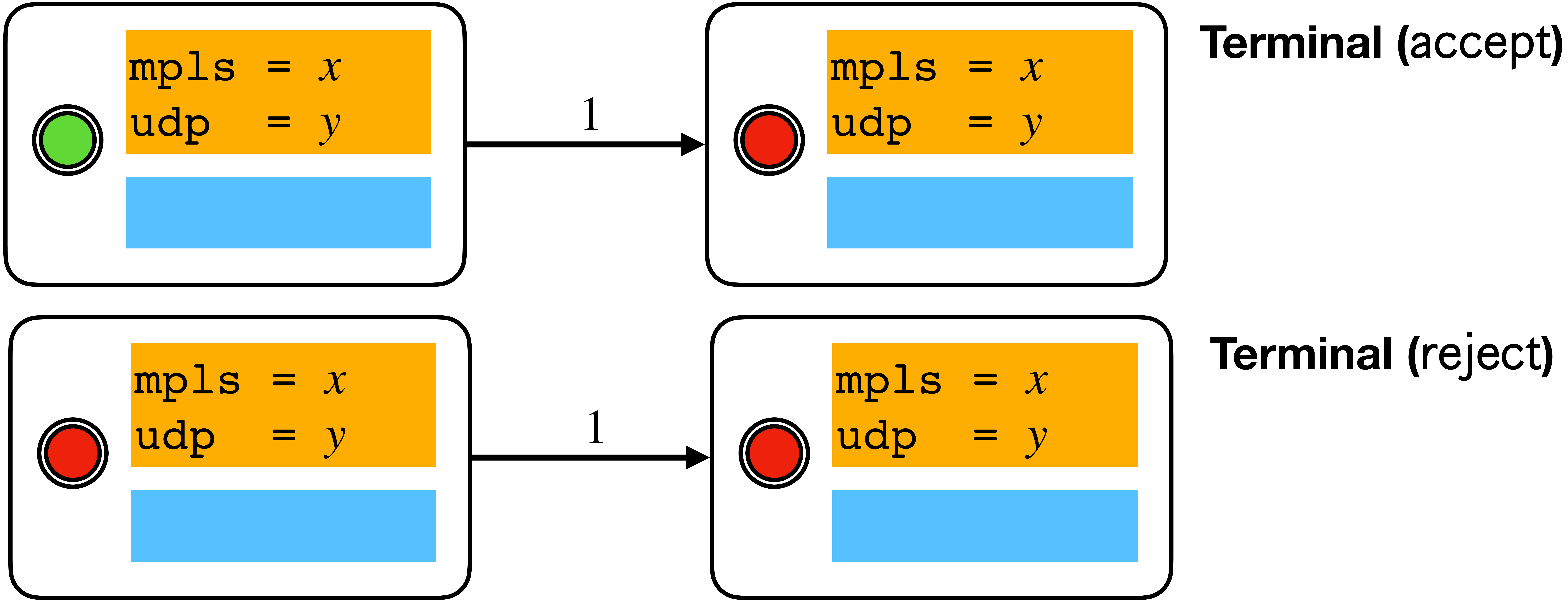
Terminal (accept)



Terminal (reject)

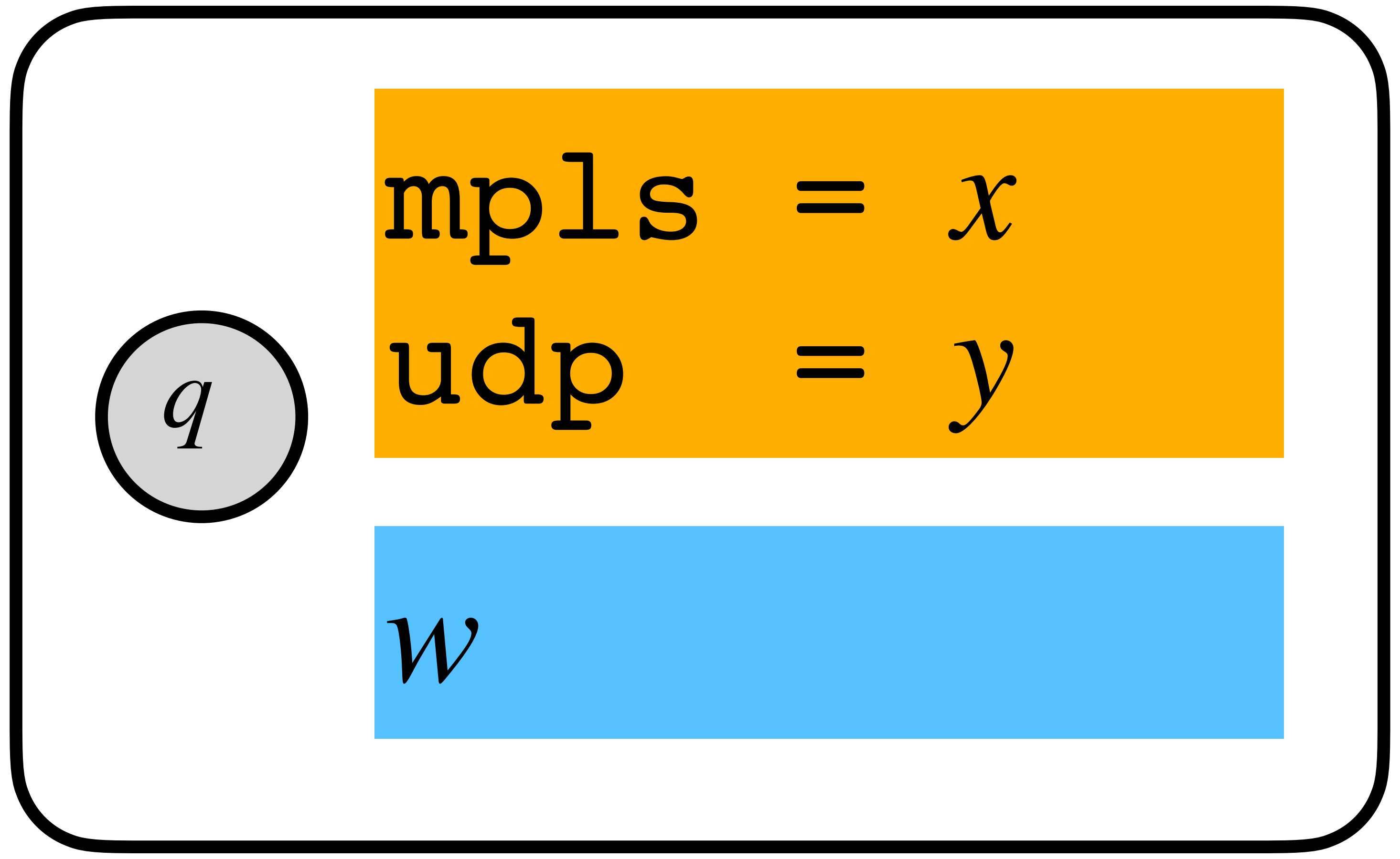
Steps

Defining a total function $\delta : C \times \{0,1\} \rightarrow C$.



Steps

Defining a total function $\delta : C \times \{0,1\} \rightarrow C$.

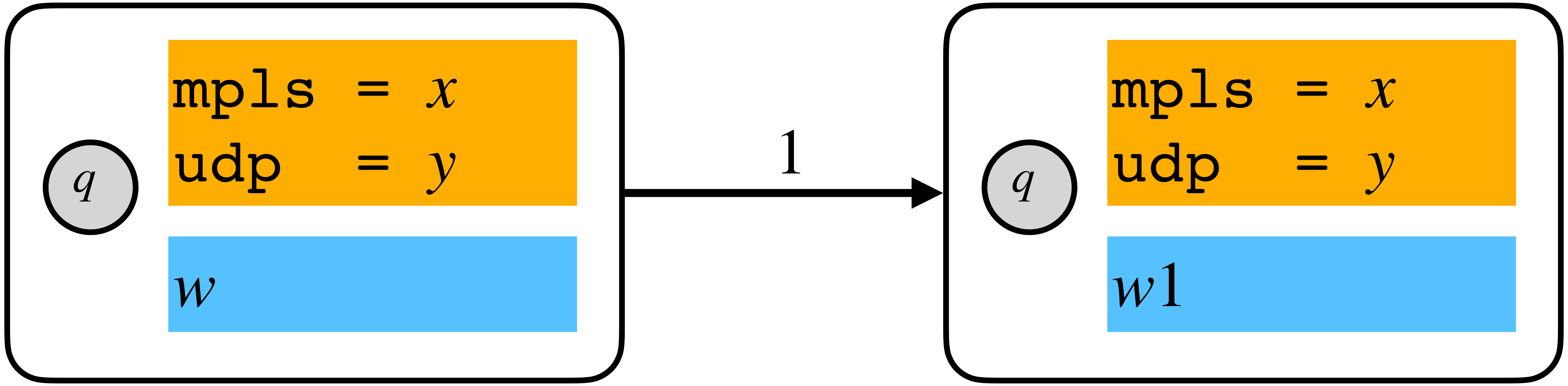


Buffering

$|w| + 1 < |\mathrm{op}(q)|$

Steps

Defining a total function $\delta : C \times \{0,1\} \to C$.

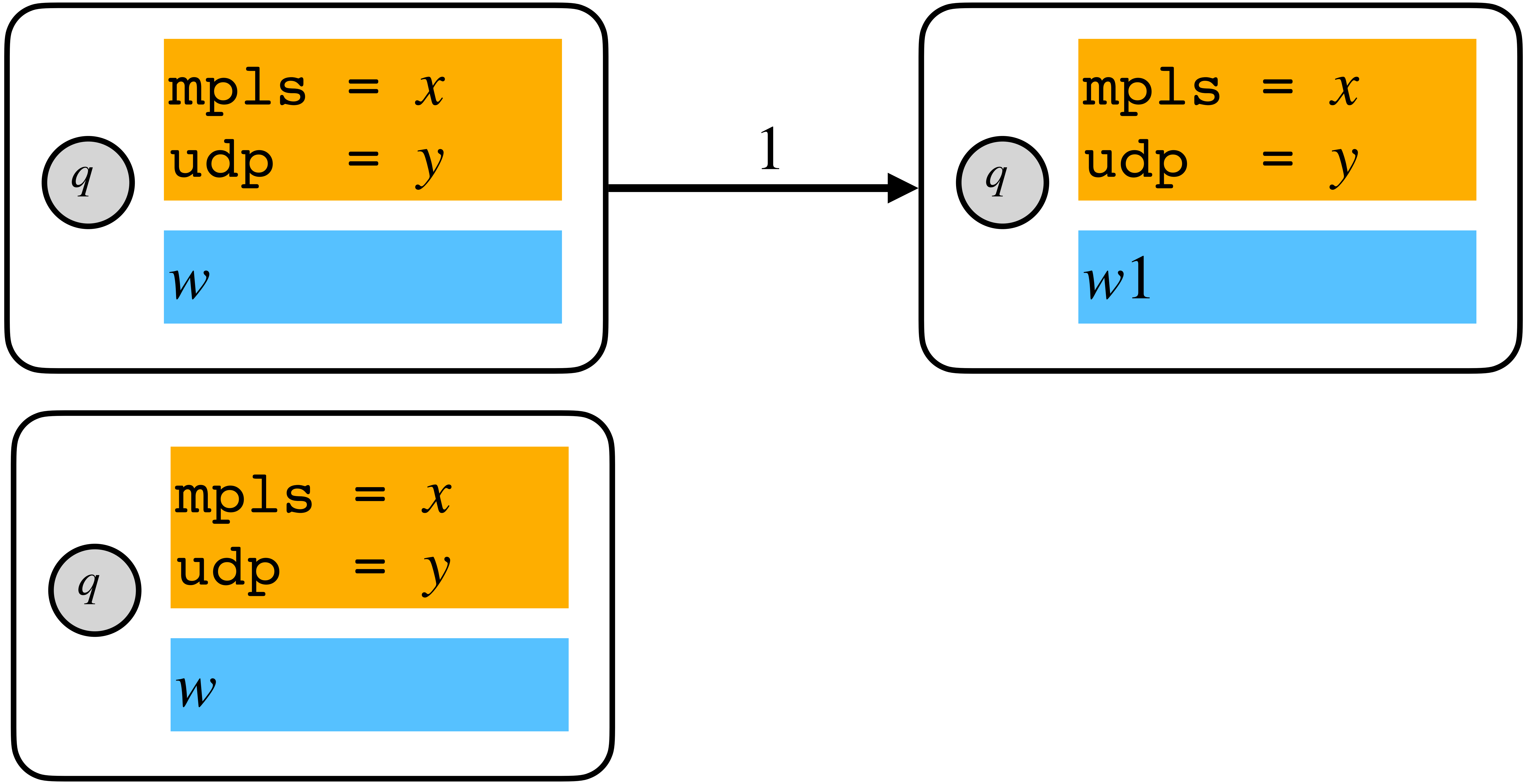


Buffering

$|w| + 1 < |\text{op}(q)|$

Steps

Defining a total function $\delta : C \times \{0,1\} \rightarrow C$.



Buffering

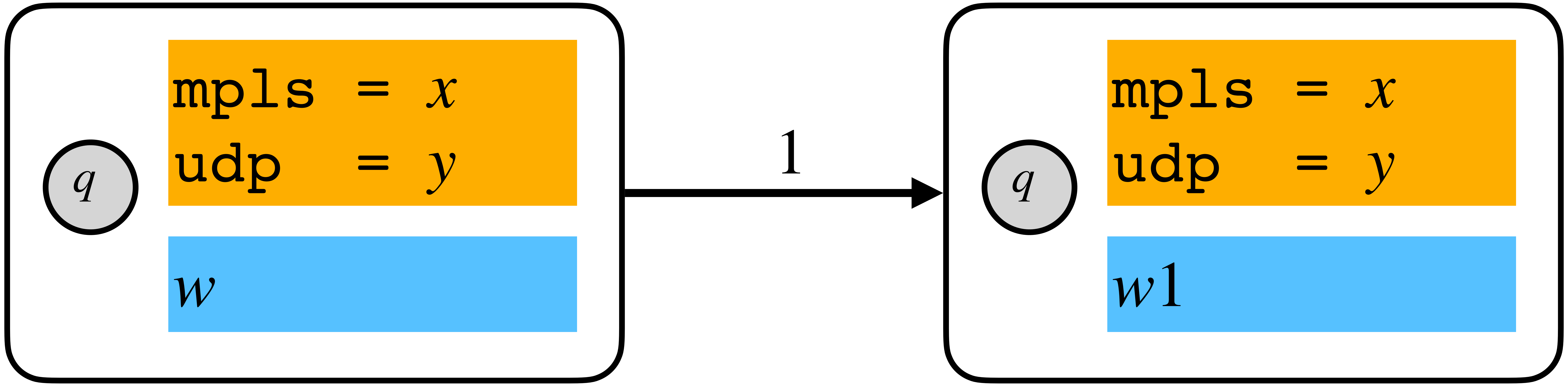
$|w| + 1 < |\text{op}(q)|$

State Change

$|w| + 1 = |\text{op}(q)|$

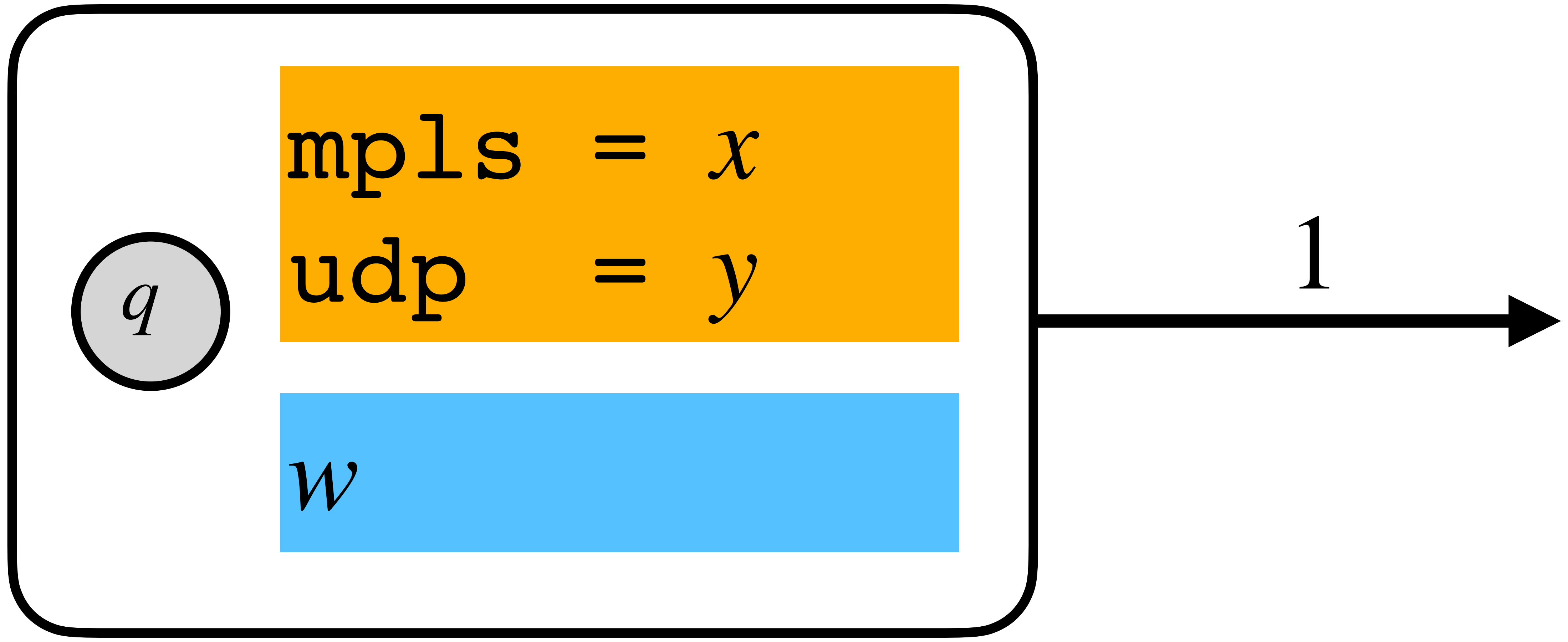
Steps

Defining a total function $\delta : C \times \{0,1\} \rightarrow C$.



Buffering

$|w| + 1 < |\mathrm{op}(q)|$



State Change

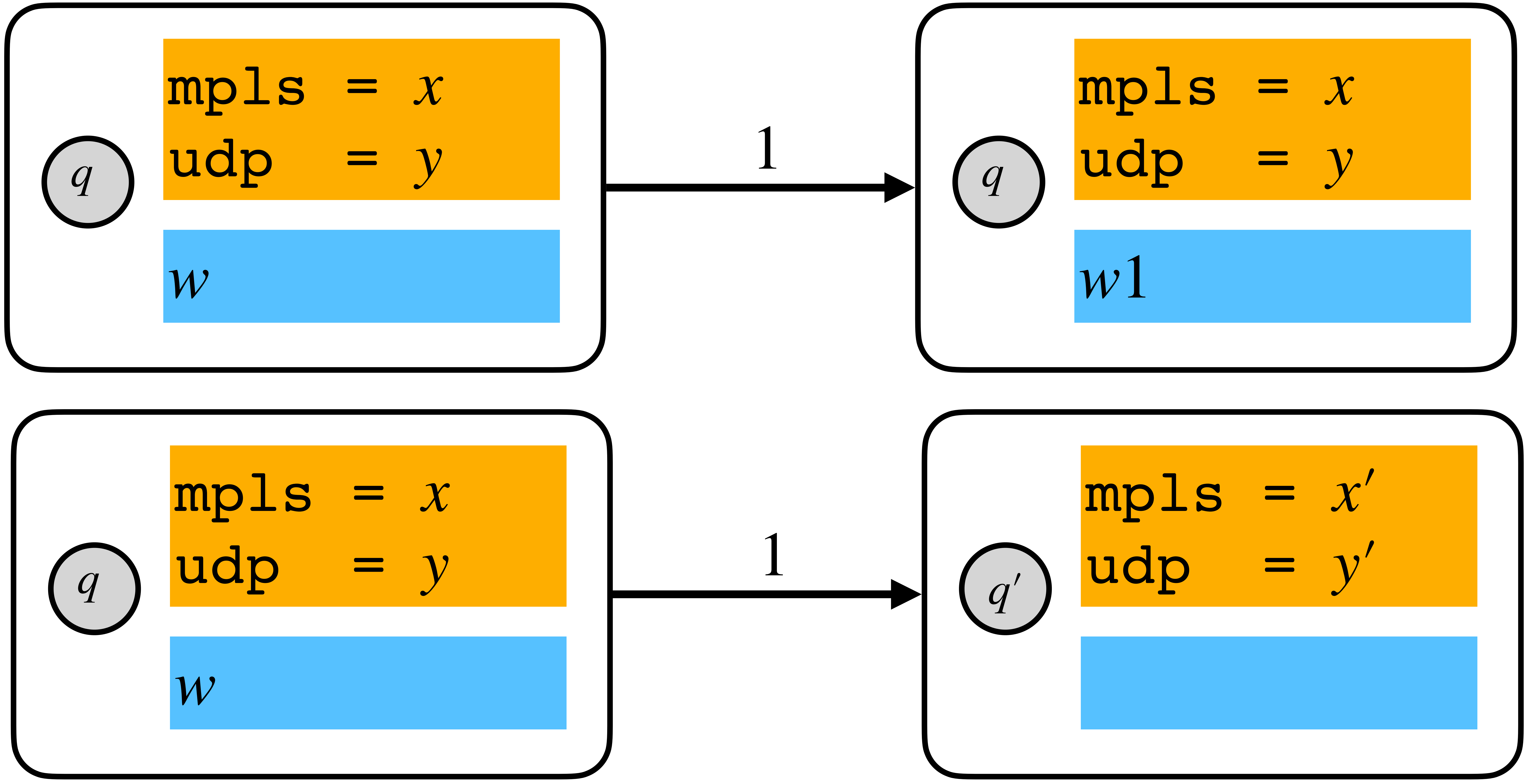
$|w| + 1 = |\mathrm{op}(q)|$

$s' = \mathrm{op}(q)(s, w1)$

$q' = \mathrm{tz}(q)(s')$

Steps

Defining a total function $\delta : C \times \{0,1\} \rightarrow C$.



Buffering

$|w| + 1 < |\text{op}(q)|$

State Change

$|w| + 1 = |\text{op}(q)|$

$s' = \text{op}(q)(s, w1)$

$q' = \text{tz}(q)(s')$

Defining Equivalence

⚠️ We'll stick to **language** equivalence, not full **program** equivalence.

So: view P4A P, Q as DFAs and decide

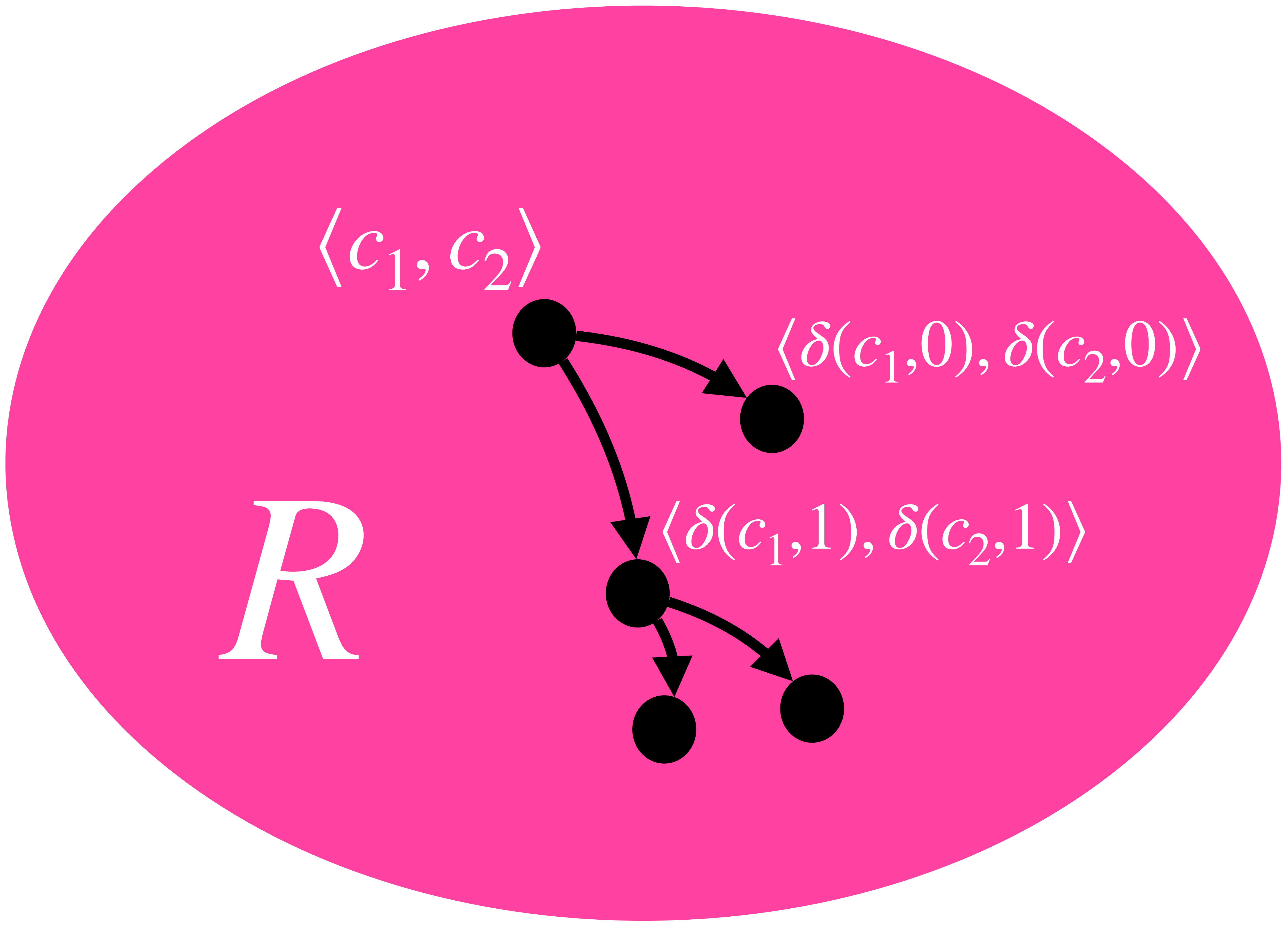
$$L_P(\mathsf{start}) = L_Q(\mathsf{start}).$$

Proving Equivalence: Bisimilarity

$R \subseteq C \times C$ is a bisimulation if it's

- closed under steps
- only relates final configs to other final configs.

If R relates two configs, then they are language equivalent.



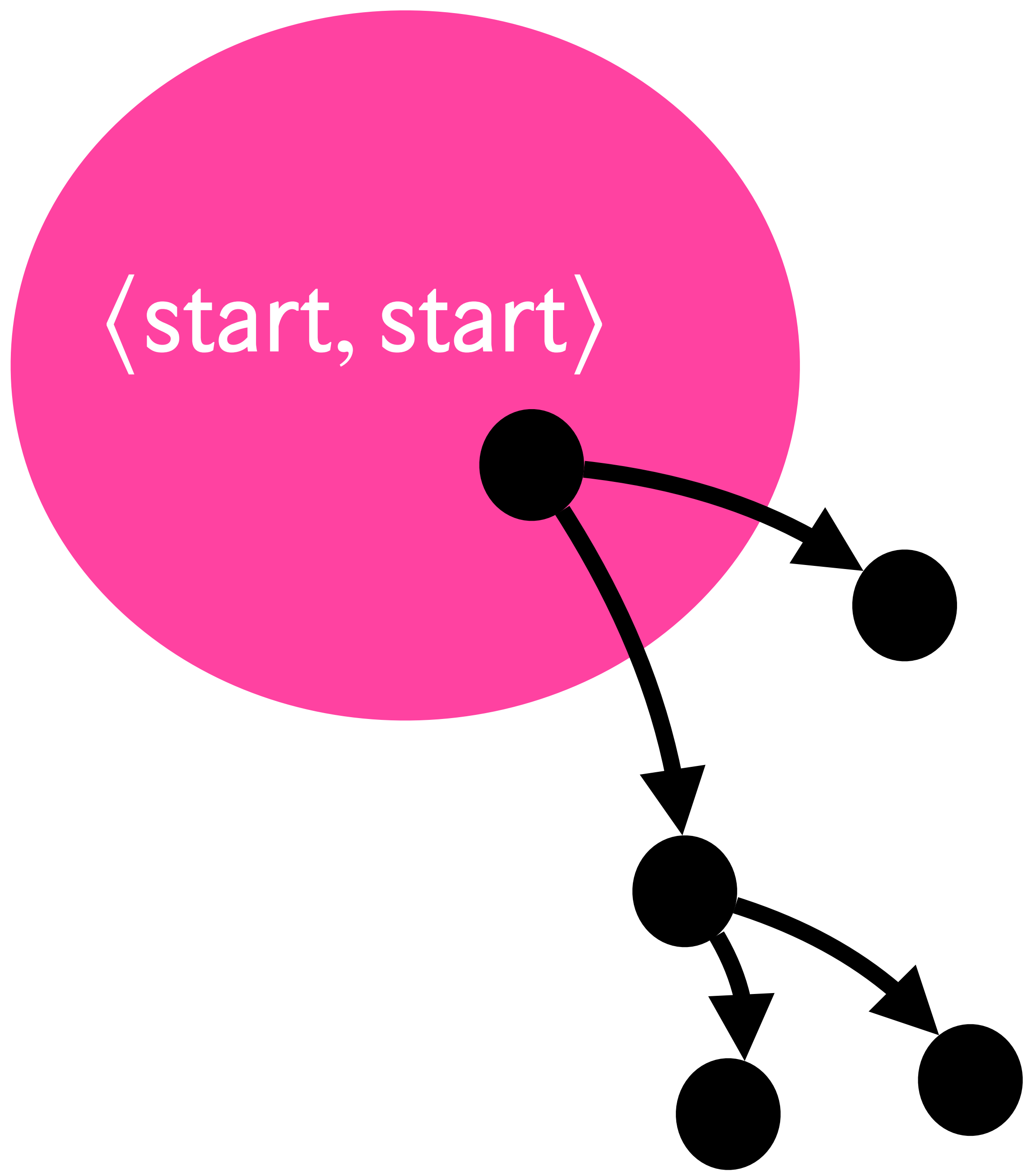
(product state space)

Constructing a Bisimulation from Below

Begin with $R = I$, the set of pairs of initial states.

Close R under parallel steps.

This produces the least bisimulation.

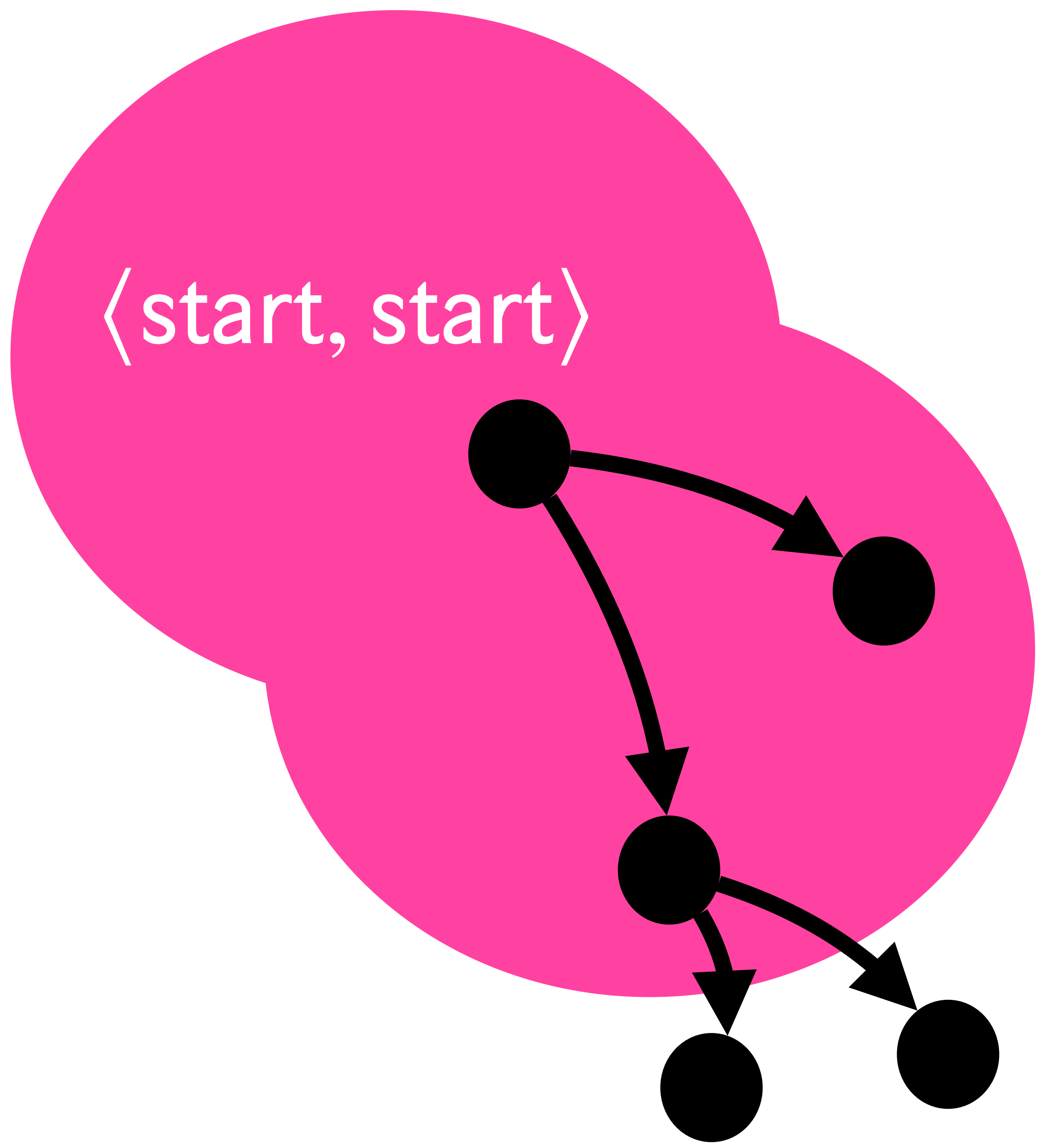


Constructing a Bisimulation from Below

Begin with $R = I$, the set of pairs of initial states.

Close R under parallel steps.

This produces the least bisimulation.

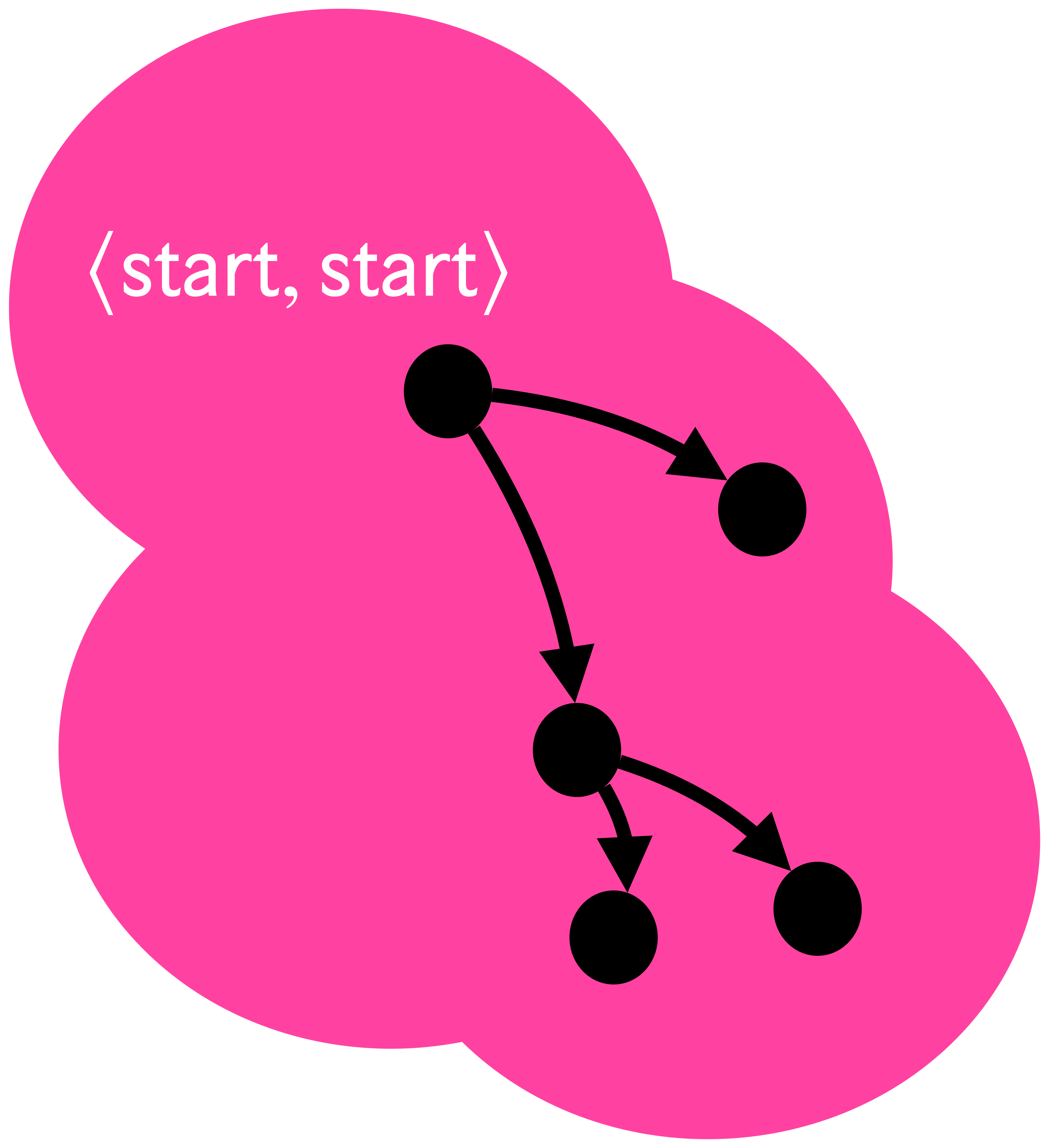


Constructing a Bisimulation from Below

Begin with $R = I$, the set of pairs of initial states.

Close R under parallel steps.

This produces the least bisimulation.

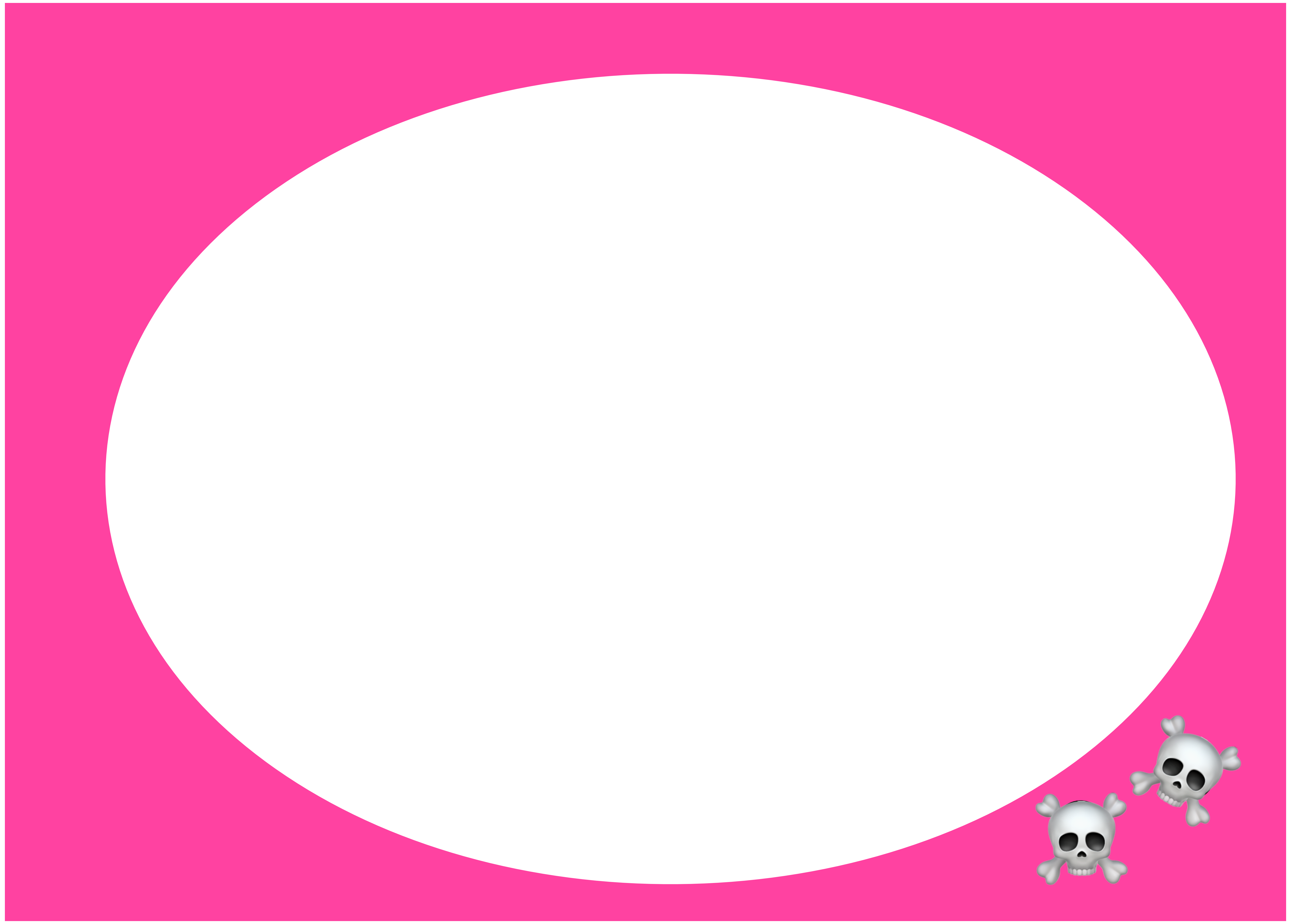


Constructing a Bisimulation From Above

Set $S = (F \times F^c) \cup (F^c \times F)$

Search **backwards** through the transition system until S is closed under backward steps.

The complement $R = S^c$ is the greatest bisimulation.

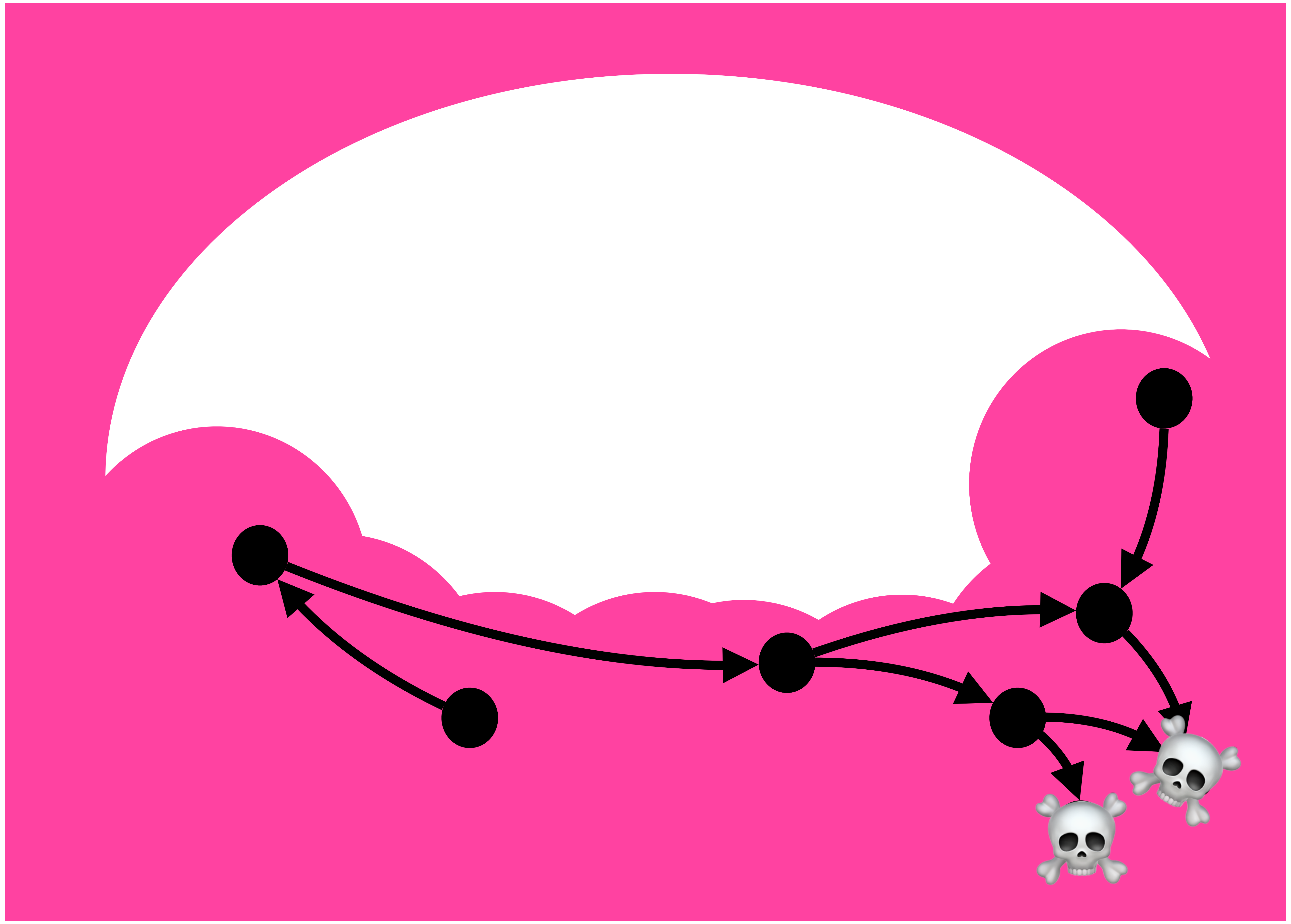


Constructing a Bisimulation From Above

Set $S = (F \times F^c) \cup (F^c \times F)$

Search **backwards** through the transition system until S is closed under backward steps.

The complement $R = S^c$ is the greatest bisimulation.

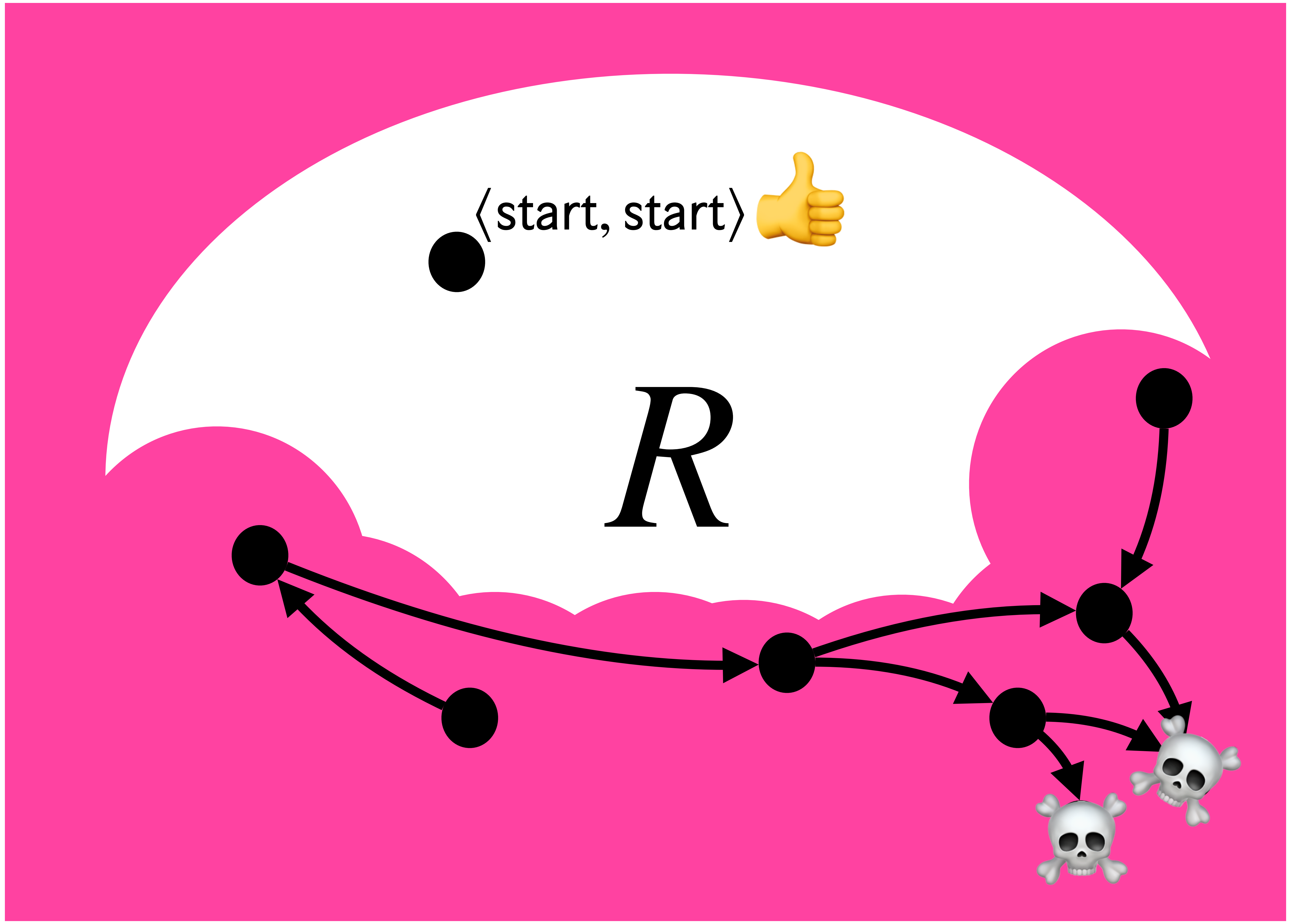


Constructing a Bisimulation From Above

Set $S = (F \times F^c) \cup (F^c \times F)$

Search **backwards** through the transition system until S is closed under backward steps.

The complement $R = S^c$ is the greatest bisimulation.

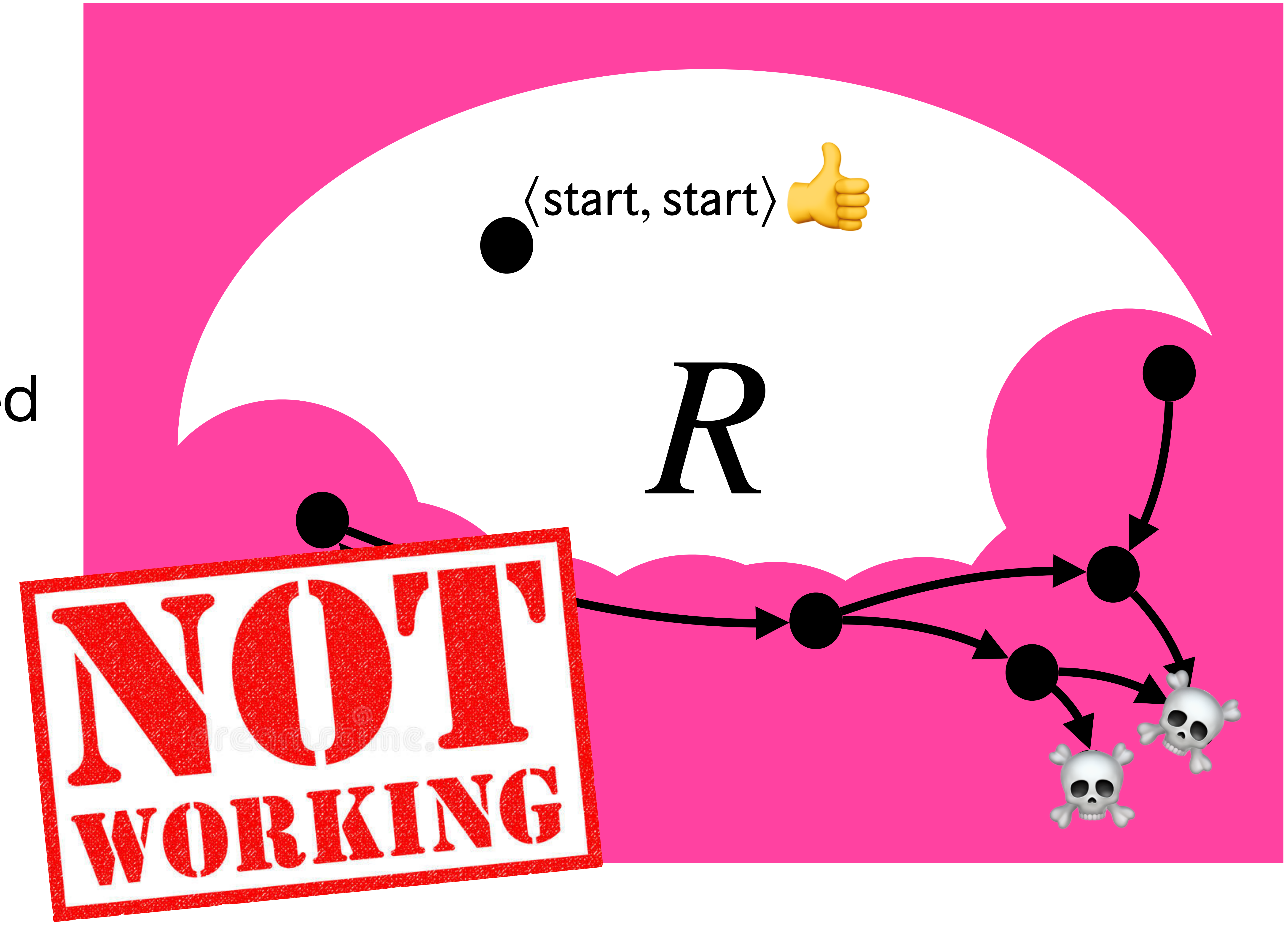


Constructing a Bisimulation From Above

Set $S = (F \times F^c) \cup (F^c \times F)$

Search **backwards** through the transition system until S is closed under backward steps.

The complement $R = S^c$ is the greatest bisimulation.



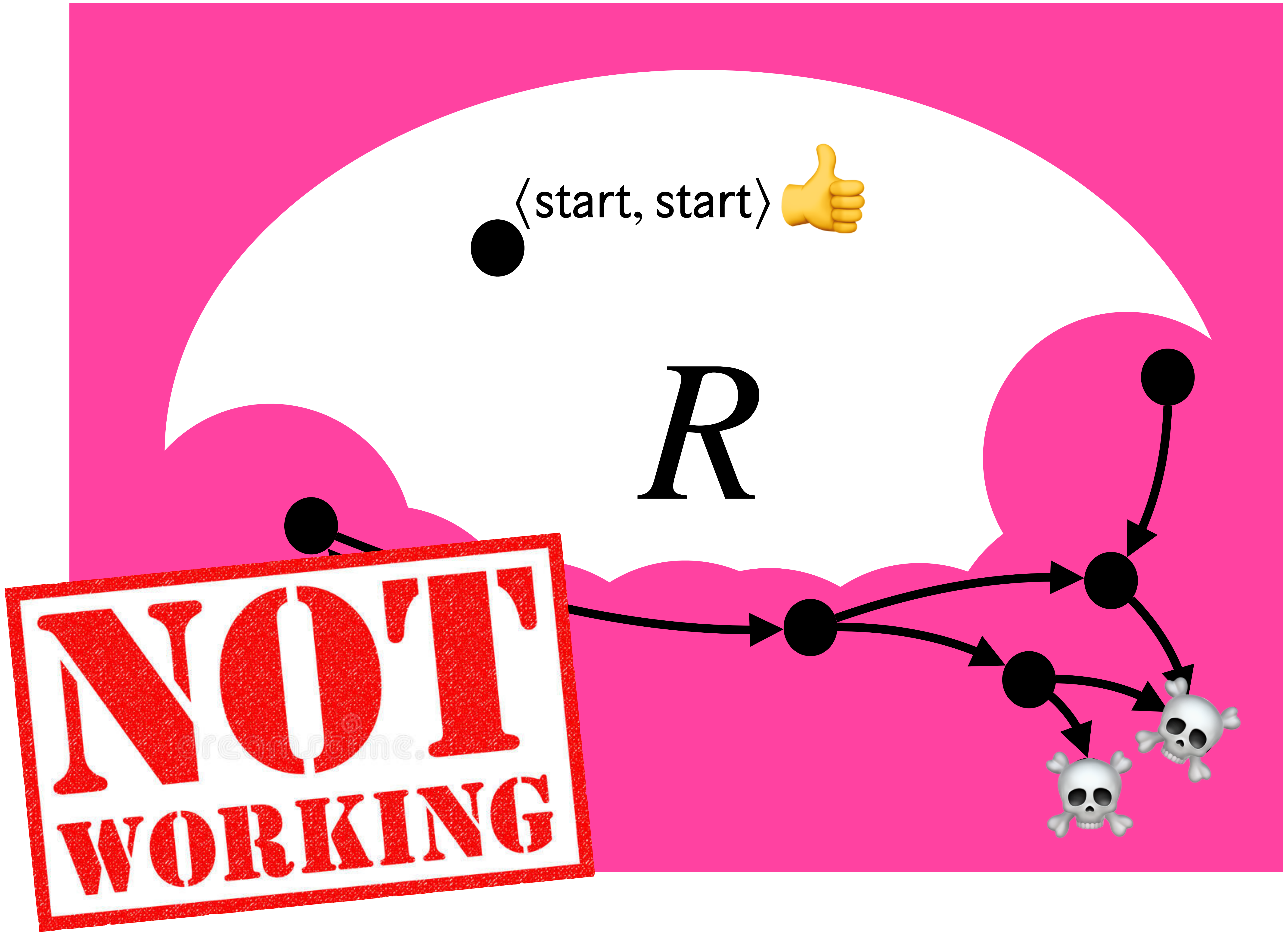
Constructing a Bisimulation From Above

There are 2^{128} config pairs for the MPLS+UDP example!

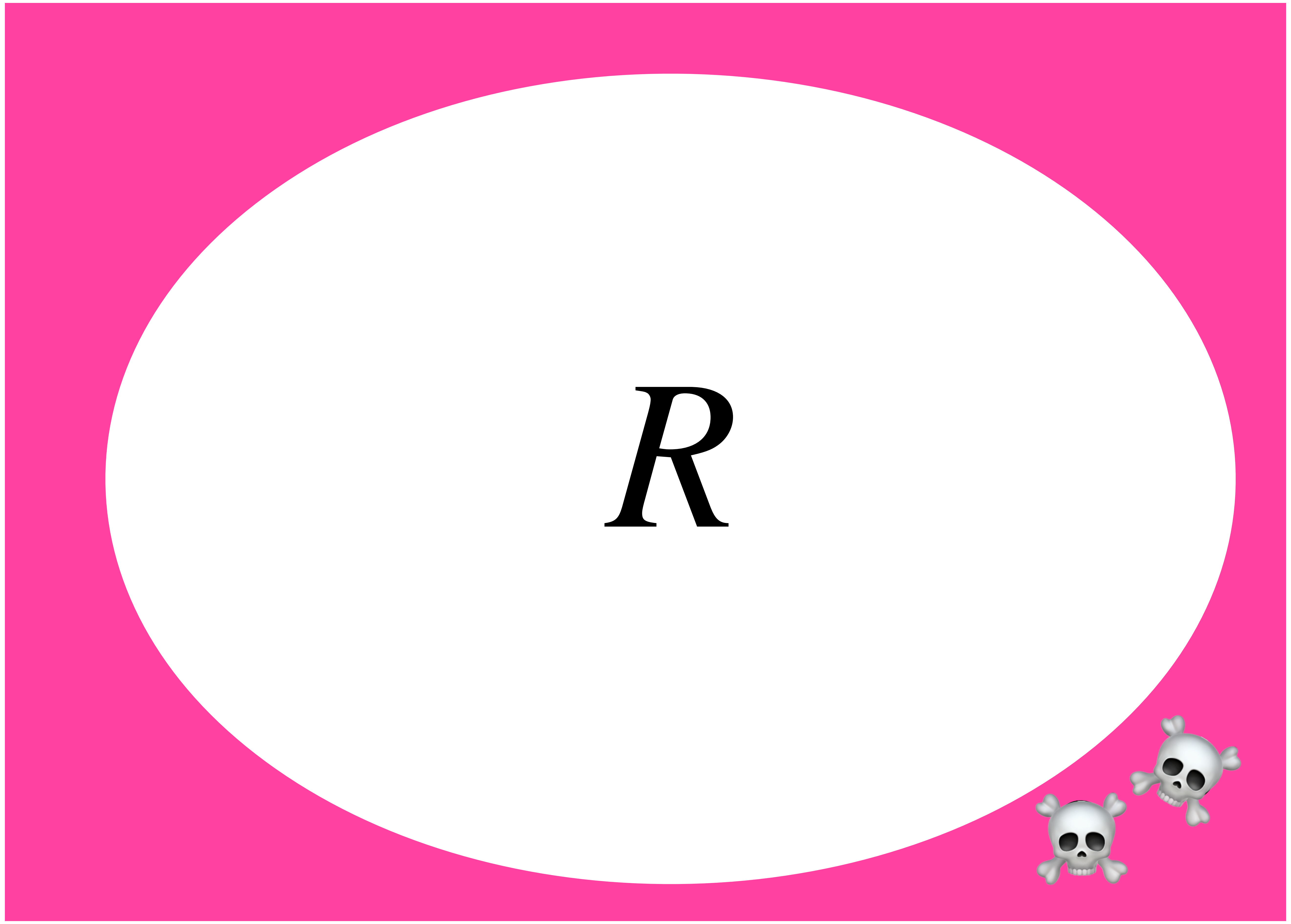
The concrete algorithm

😨 Represents S as a concrete set of pairs.

😱 Searches that space one step at a time.



Constructing a Symbolic Bisimulation

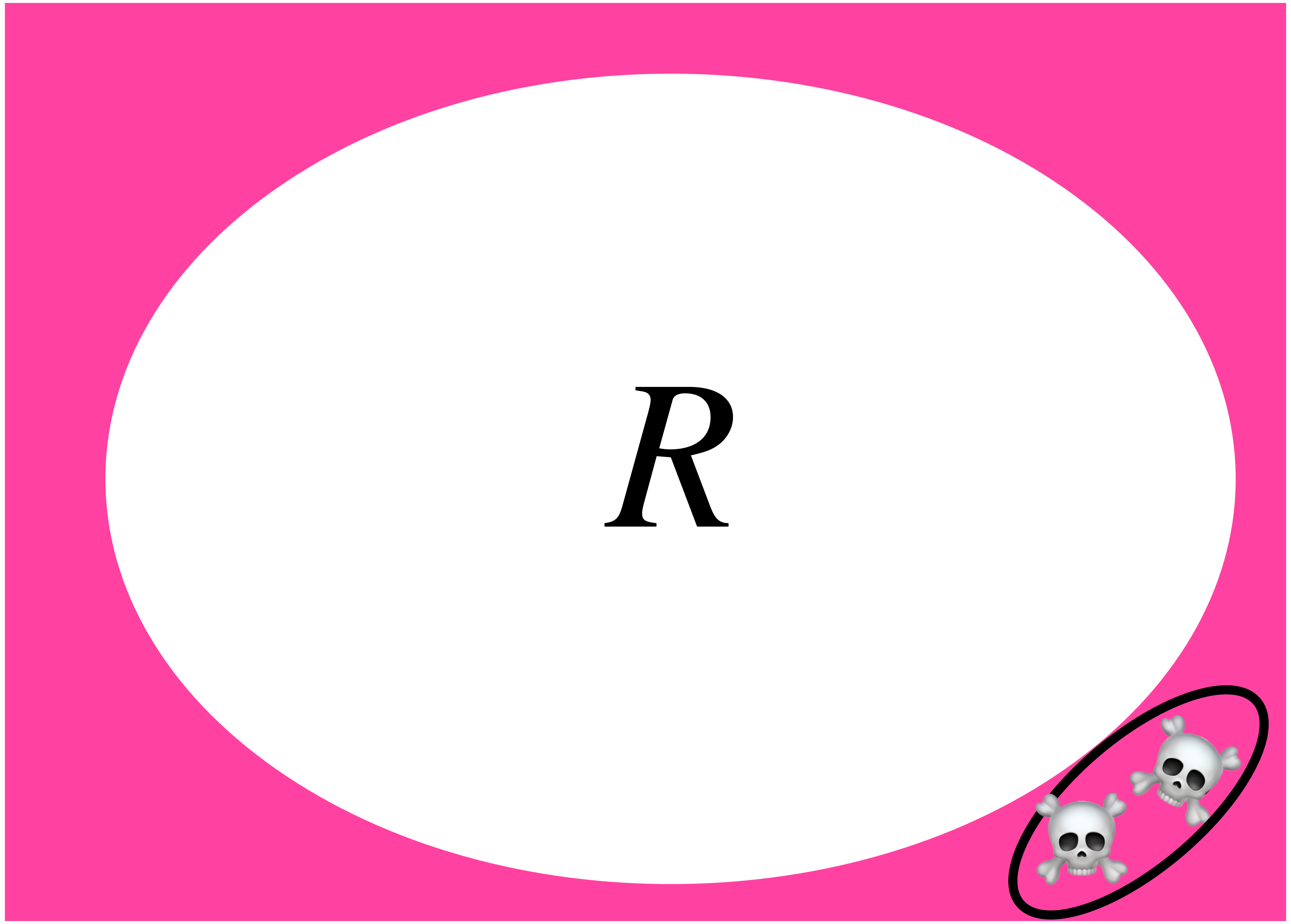


Use **symbolic** relations $S = \varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_N$.

Instead of backward steps, compute **weakest preconditions**.

At the end, $R = \neg S$ is the greatest bisimulation.

Constructing a Symbolic Bisimulation

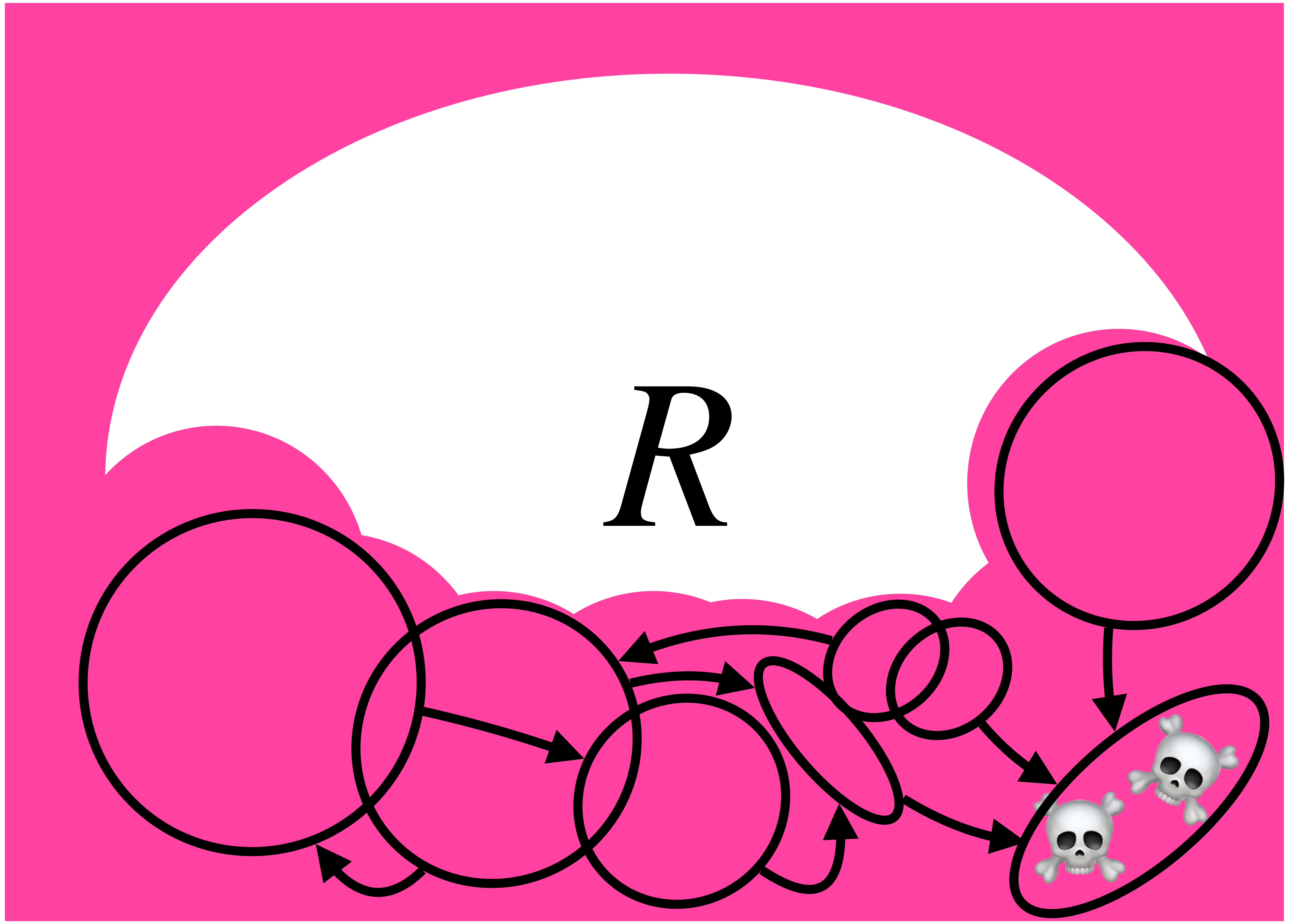


Use **symbolic** relations
$S = \varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_N$.

Instead of backward steps,
compute **weakest preconditions.**

At the end, $R = \neg S$ is the
greatest bisimulation.

Constructing a Symbolic Bisimulation

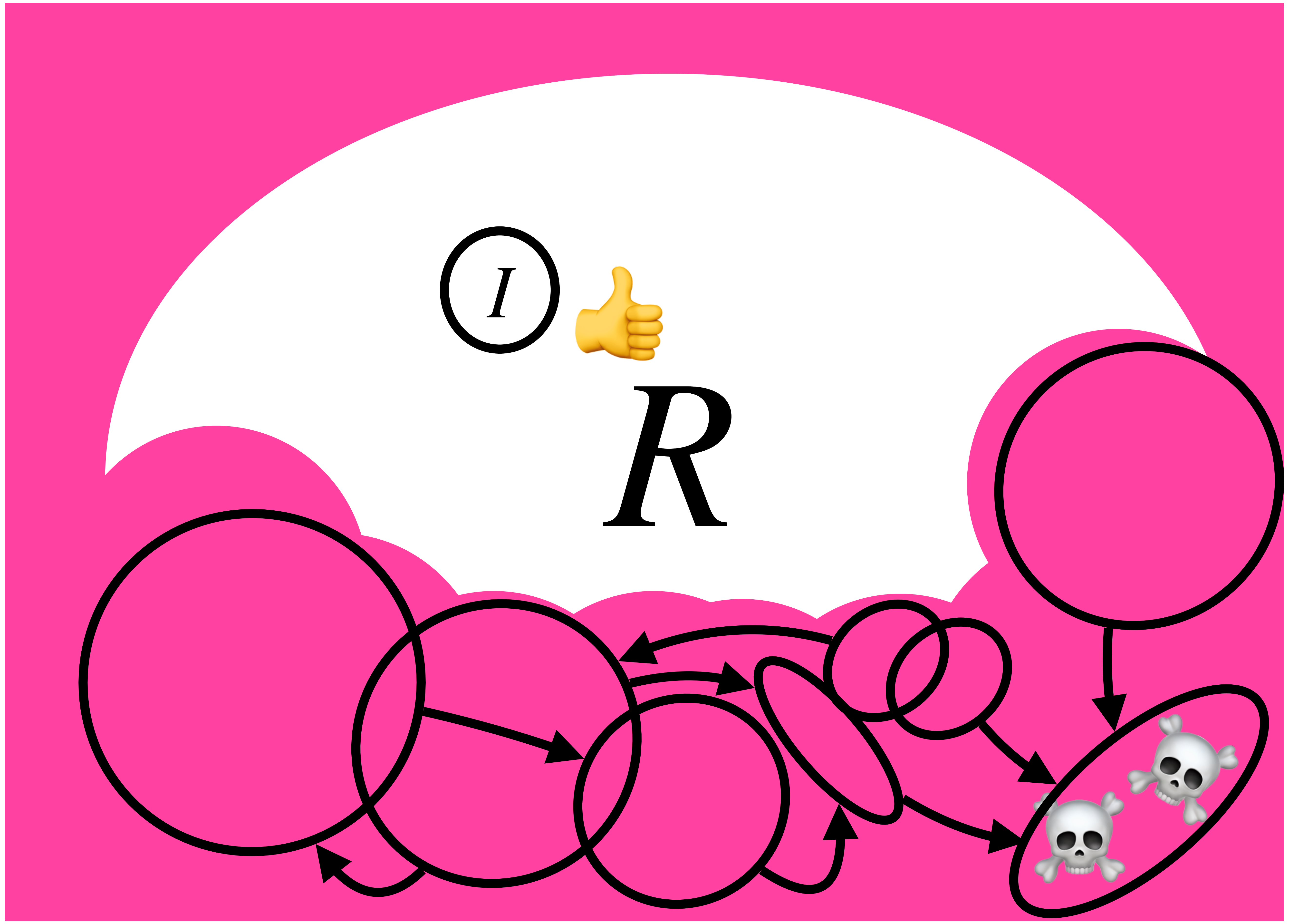


Use **symbolic** relations $S = \varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_N$.

Instead of backward steps, compute **weakest preconditions**.

At the end, $R = \neg S$ is the greatest bisimulation.

Constructing a Symbolic Bisimulation



Use **symbolic** relations $S = \varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_N$.

Instead of backward steps, compute **weakest preconditions.**

At the end, $R = \neg S$ is the greatest bisimulation.

Algorithm 1: Symbolic equivalence checking.

Input: A formula ϕ representing initial states.

Input: A set of formulas I s.t. for all $c_1, c_2 \in C$,

$$[\forall \psi \in I.\ c_1 \ [\![\psi]\!] \ c_2] \Leftrightarrow [c_1 \in F \Leftrightarrow c_2 \in F]$$

Input: A function WP s.t. for all ψ, and $c_1, c_2 \in C$,

$$[\forall b \in \{0,1\}.\ \delta(c_1, b) \ [\![\psi]\!] \ \delta(c_2, b)] \Leftrightarrow c_1 \ \left[\!\!\left[\bigwedge \mathsf{WP}(\psi) \right]\!\!\right] \ c_2$$

Output: **true** if and only if for all $c_1, c_2 \in C$ with $c_1 \ [\![\phi]\!]_{\mathcal{L}} \ c_2$, it holds that $L(c_1) = L(c_2)$

$R \leftarrow \emptyset;\ T \leftarrow I$

while $T \neq \emptyset$ **do**

    pop ψ from T

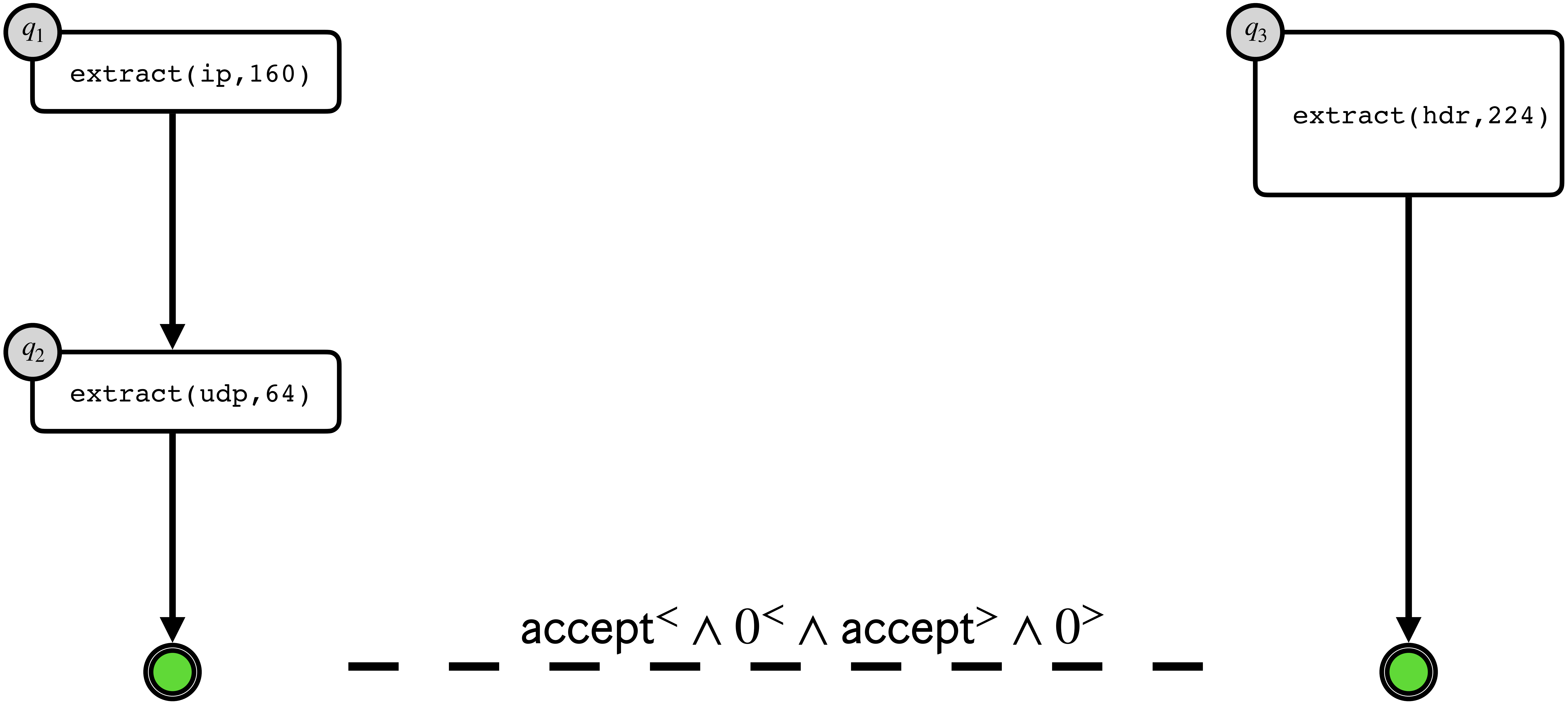
    if not $\bigwedge R \vDash \psi$ **then**

        $R \leftarrow R \cup \{\psi\}$

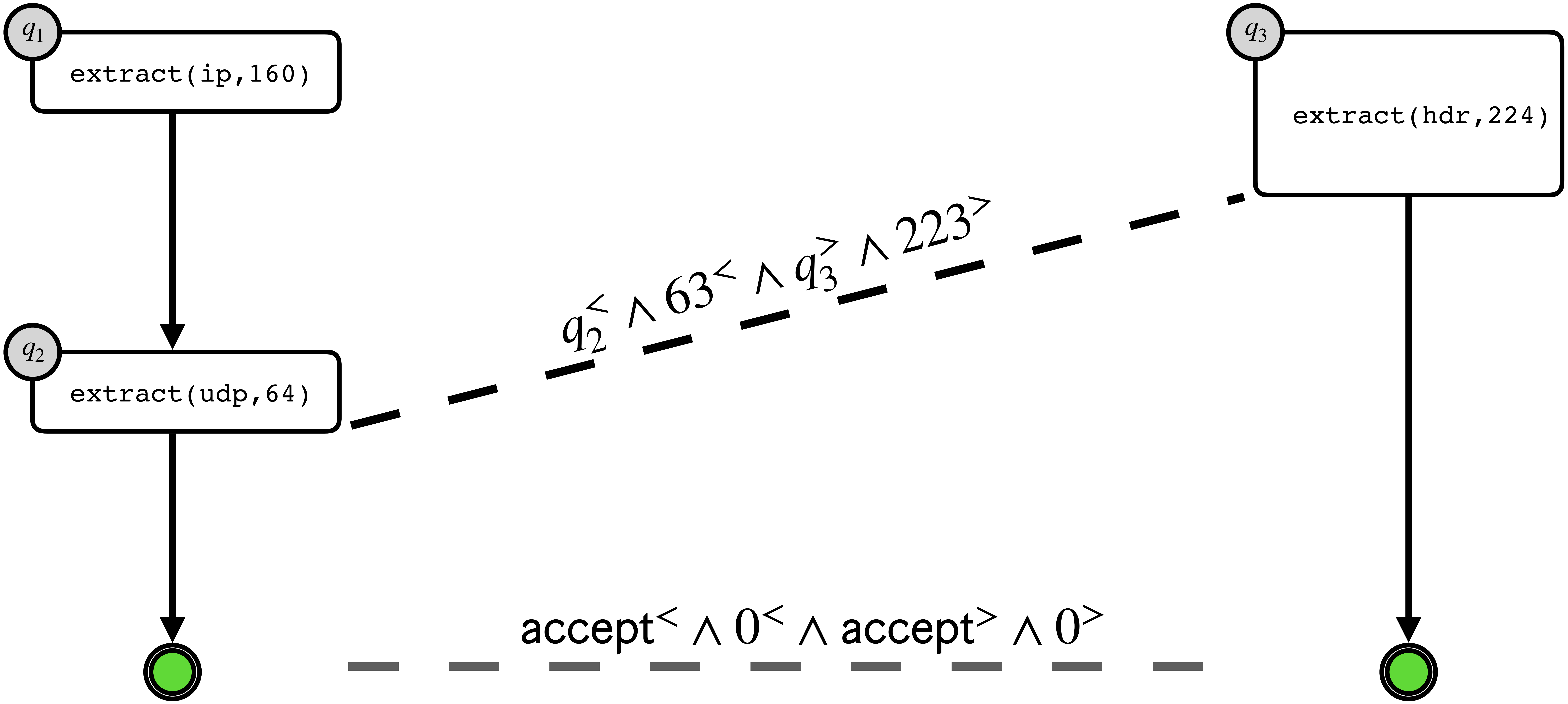
        $T \leftarrow T \cup \mathsf{WP}(\psi)$

return true if $\phi \vDash \bigwedge R$, otherwise **false**

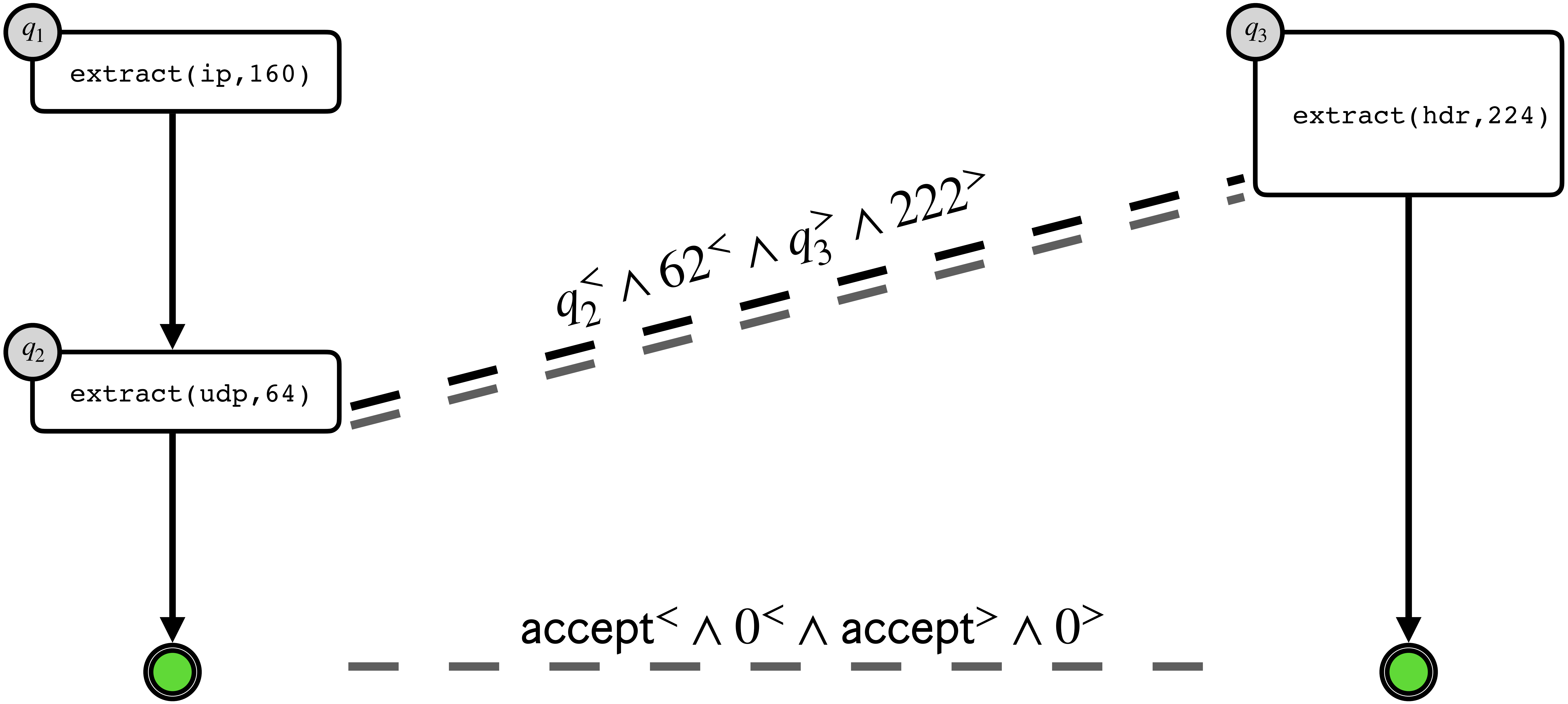
The Need for Leaps



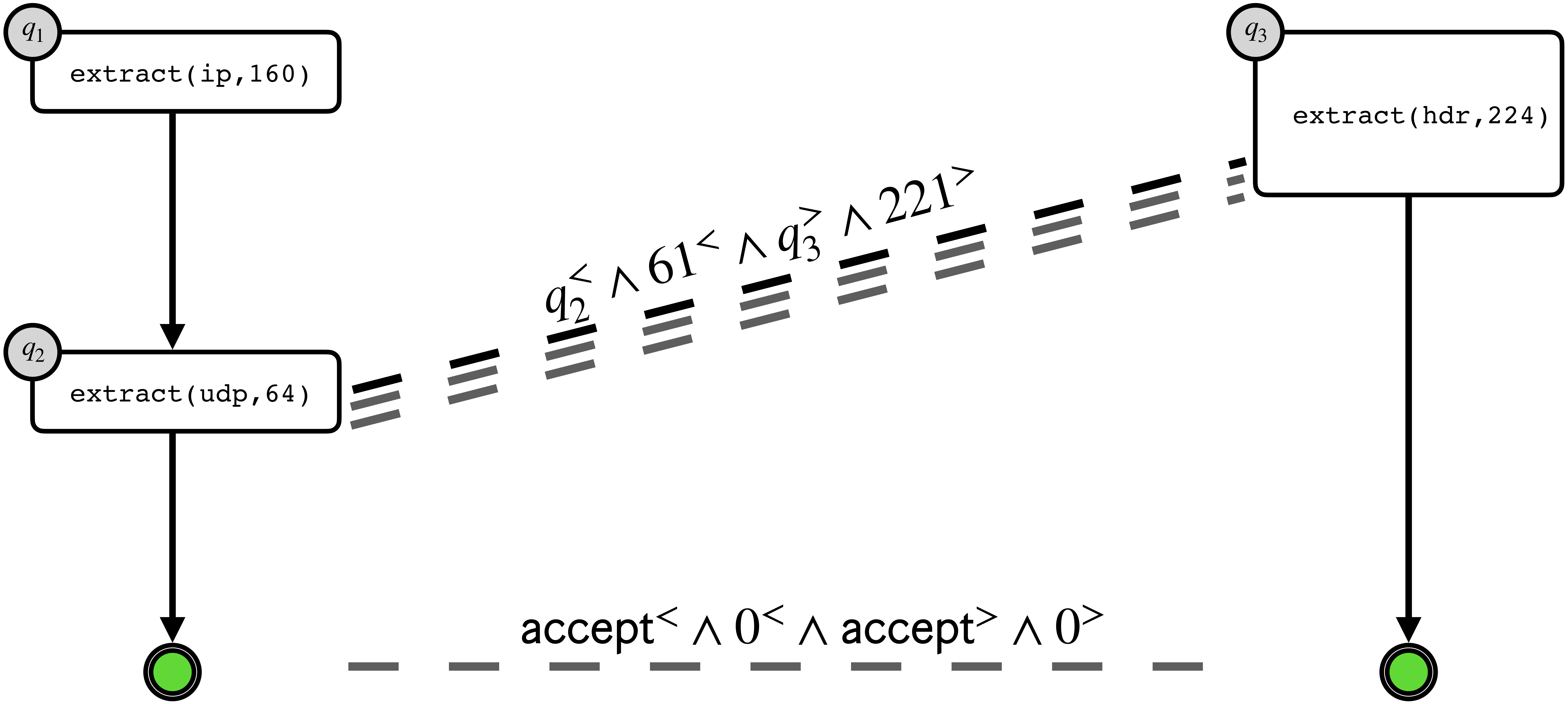
The Need for Leaps



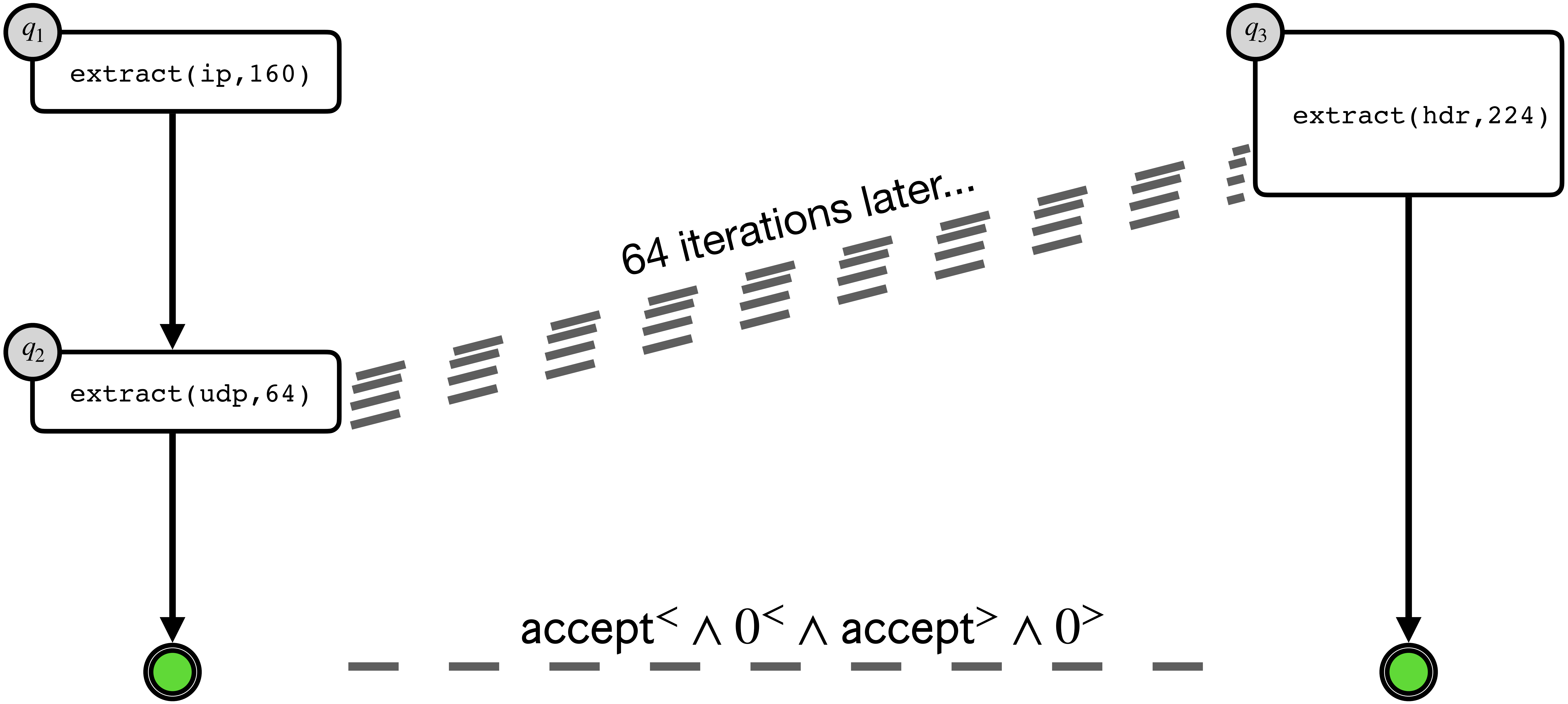
The Need for Leaps



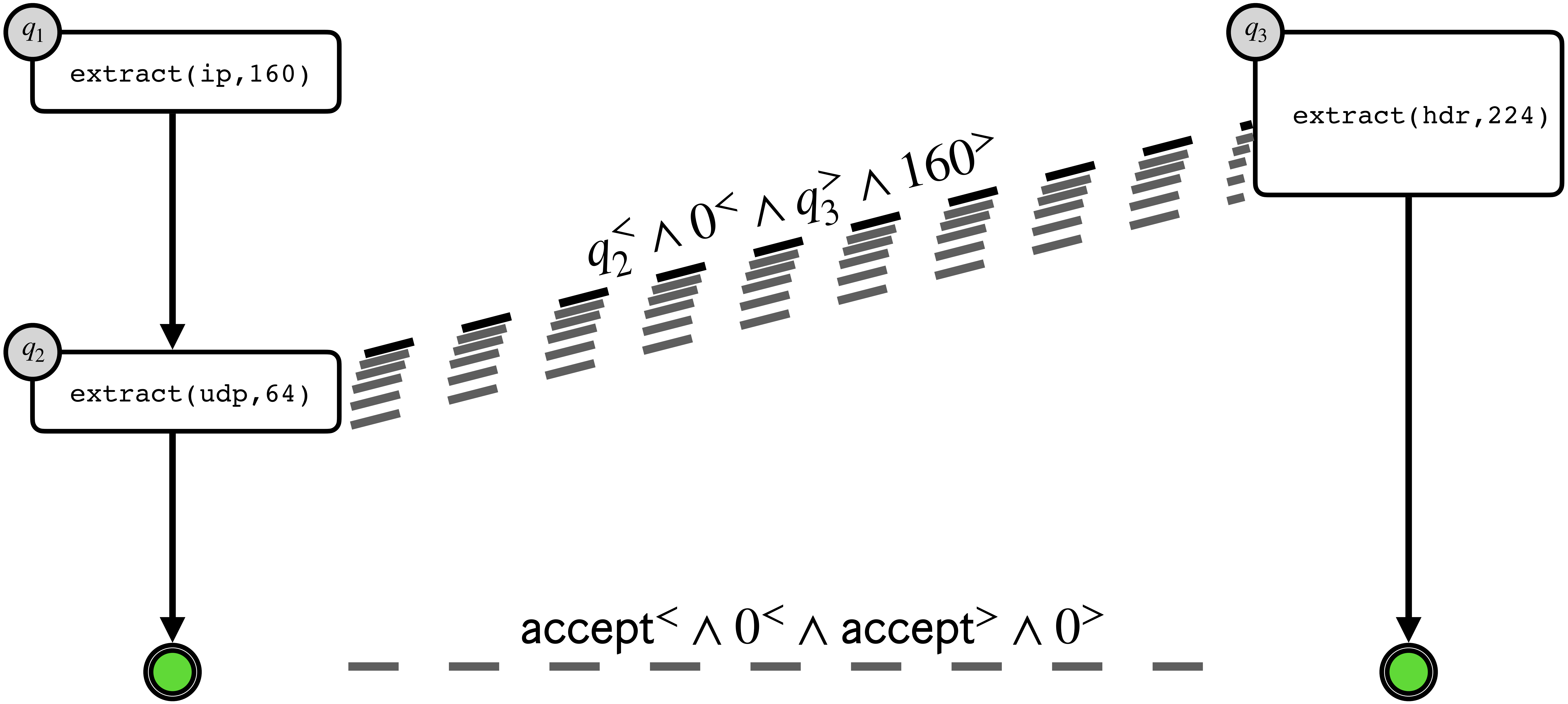
The Need for Leaps



The Need for Leaps



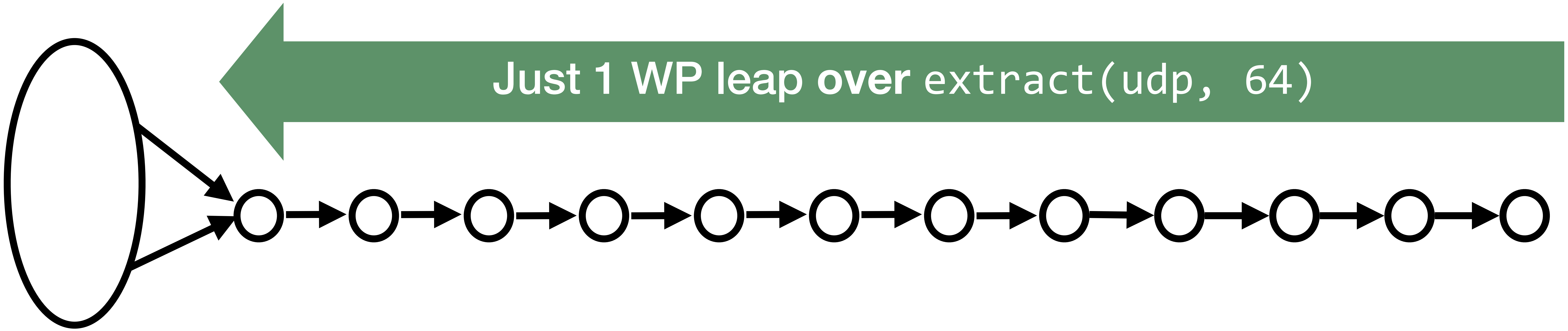
The Need for Leaps



Accelerating WP

Quiz: ideas for making this faster?

Solution: Leaps



Leaps

Definition 5.3 (Leap size). Let $c_1, c_2 \in C$ and $c_i = \langle q_i, s_i, w_i \rangle$; we define the *leap size* $\sharp(c_1, c_2) \in \mathbb{N}$ as follows:

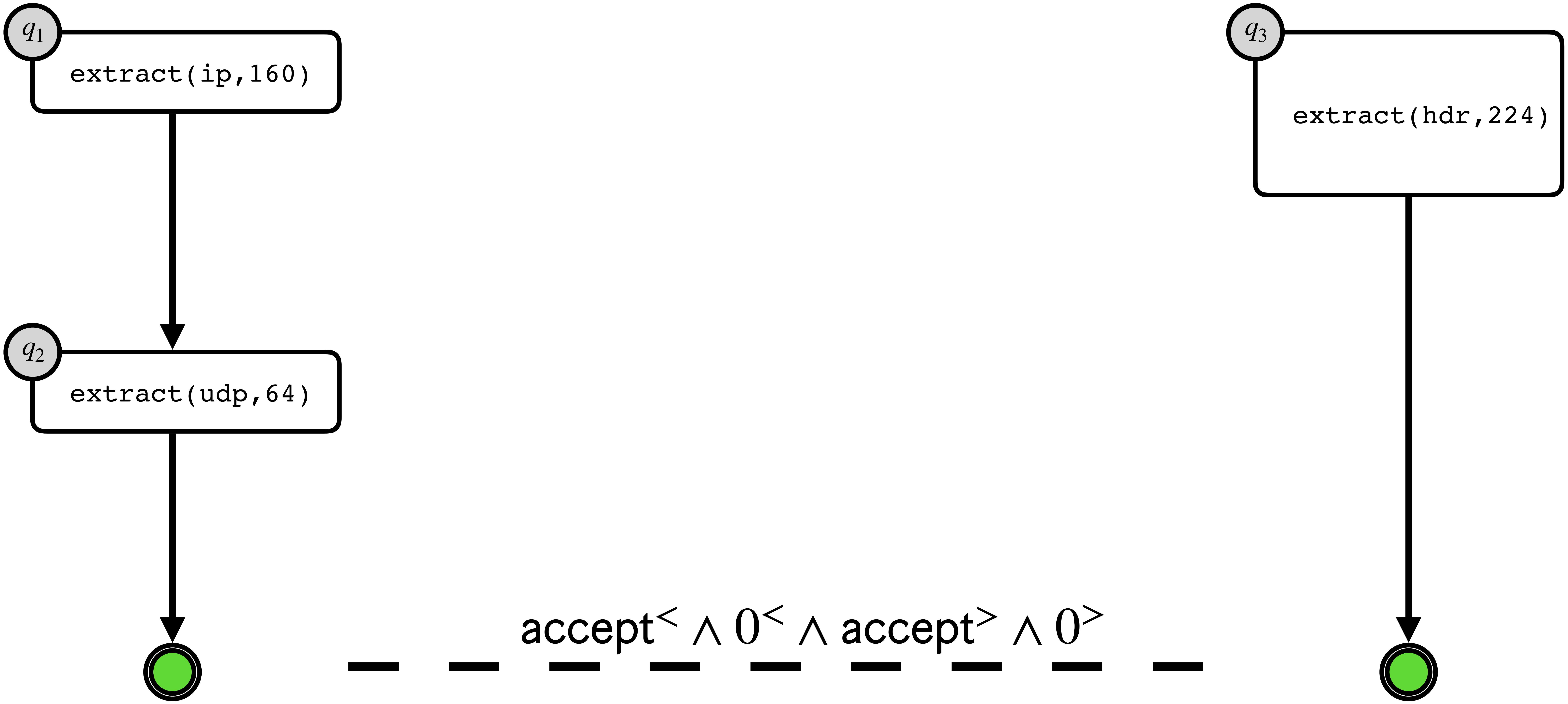
$$\sharp(c_1, c_2) = \begin{cases} 1 & q_1, q_2 \notin Q \\ \|tz(q_1)\| - |w_1| & q_1 \in Q, q_2 \notin Q \\ \|tz(q_2)\| - |w_2| & q_1 \notin Q, q_2 \in Q \\ \min(\|tz(q_1)\| - |w_1|, \\ \qquad \|tz(q_2)\| - |w_2|) & q_1, q_2 \in Q \end{cases}$$

Bisimulations with Leaps

Definition 5.4 (Bisimulation with leaps). A *bisimulation with leaps* is a relation $R \subseteq C \times C$, such that for all $c_1 \; R \; c_2$, (1) $c_1 \in F$ if and only if $c_2 \in F$, and (2) $\delta^*(c_1, w) \; R \; \delta^*(c_2, w)$ for all $w \in \{0, 1\}^{\sharp(c_1, c_2)}$. A *symbolic bisimulation with leaps* is a formula ϕ such that $[\![\phi]\!]_{\mathcal{L}}$ is a bisimulation with leaps.

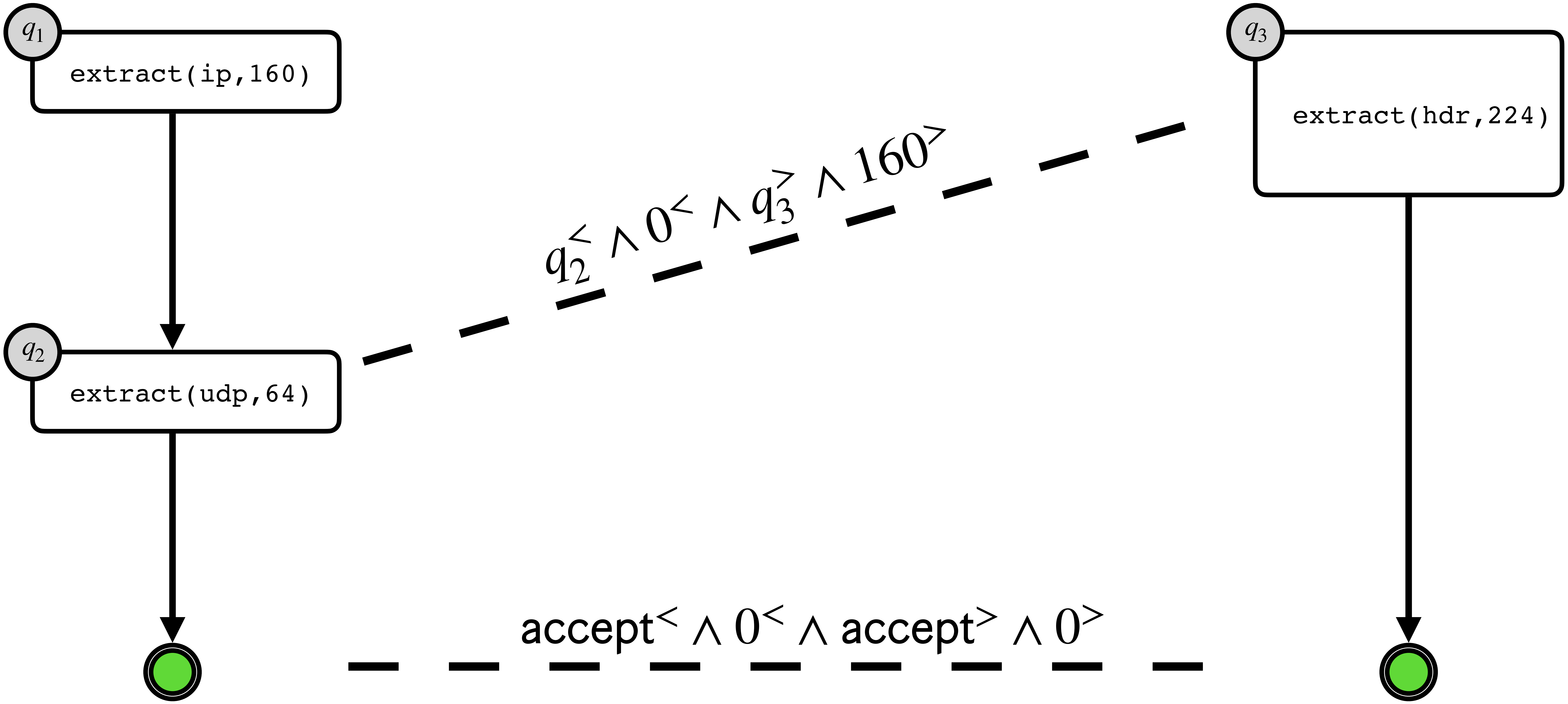
Sound and complete as an instance of the coalgebraic technique of *bisimulation up to* (here, up to leaps).

Leaps

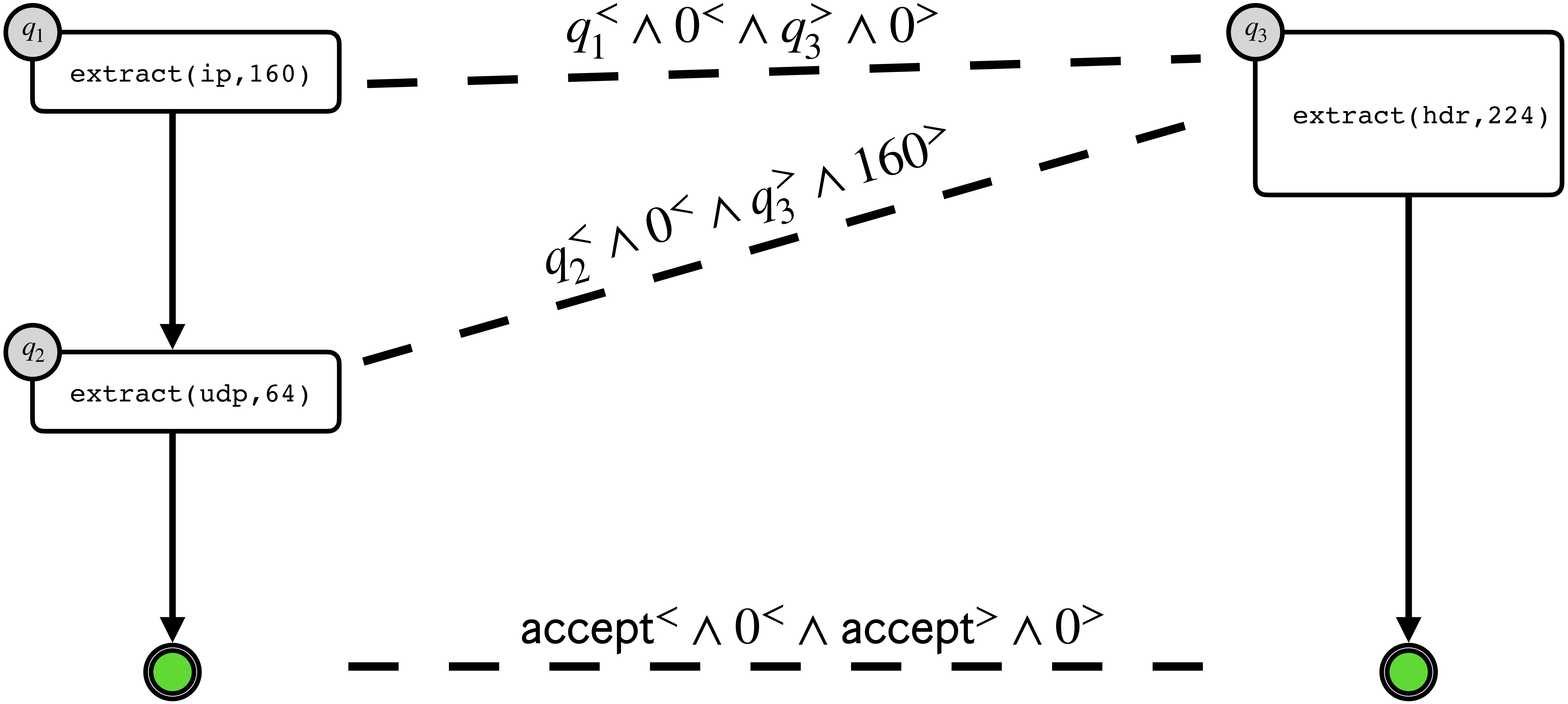


q1
extract(ip,160)
q2
extract(udp,64)
q3
extract(hdr,224)
accept< ∧ 0< ∧ accept> ∧ 0>

Leaps

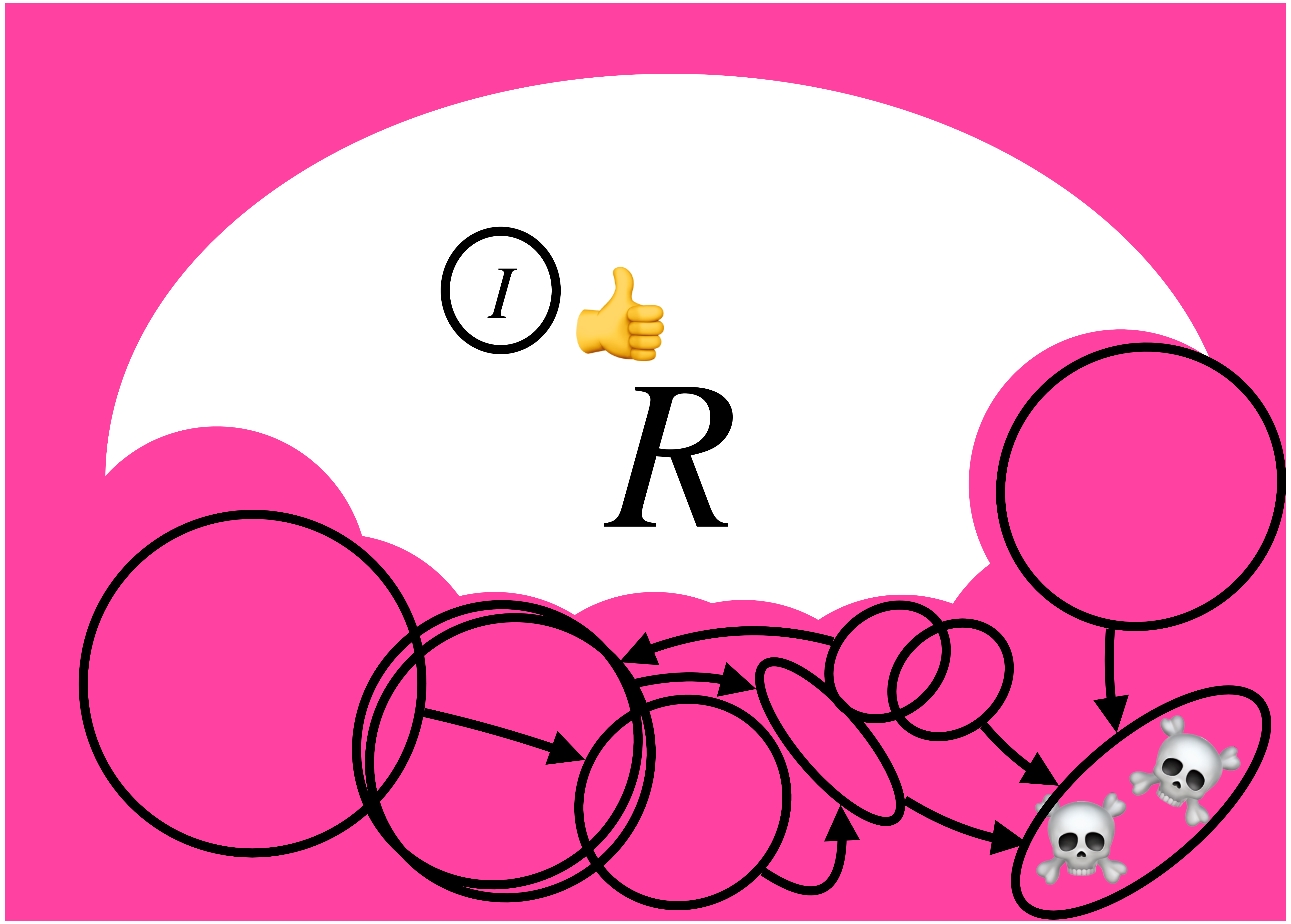

q_1
extract(ip,160)
q_3
extract(hdr,224)
$q_2^< \wedge 0^< \wedge q_3^> \wedge 160^>$
q_2
extract(udp,64)
$\text{accept}^< \wedge 0^< \wedge \text{accept}^> \wedge 0^>$

Leaps



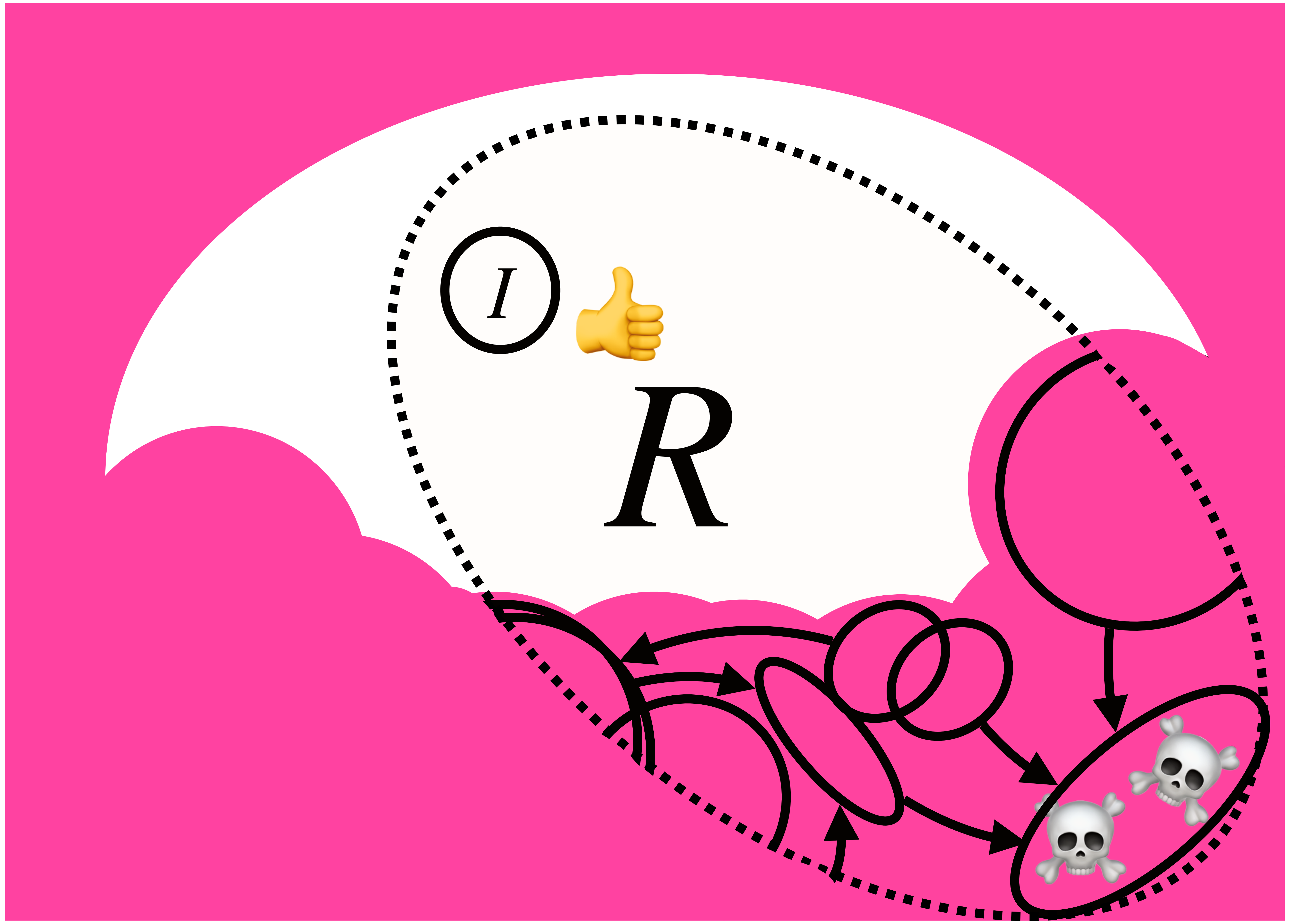
q_1
extract(ip,160)
$q_1^< \wedge 0^< \wedge q_3^> \wedge 0^>$
q_3
extract(hdr,224)
$q_2^< \wedge 0^< \wedge q_3^> \wedge 160^>$
q_2
extract(udp,64)
$accept^< \wedge 0^< \wedge accept^> \wedge 0^>$

Reachability Analysis



Problem: The WP operator is way too precise!

Reachability Analysis



Problem: The WP operator is way too precise!

Solution: Over-approximate the reachable state space and stop searching when you reach that boundary.

Gives you an intermediate bisimulation instead of the greatest bisimulation.

Implementation

SplitStateProof.v

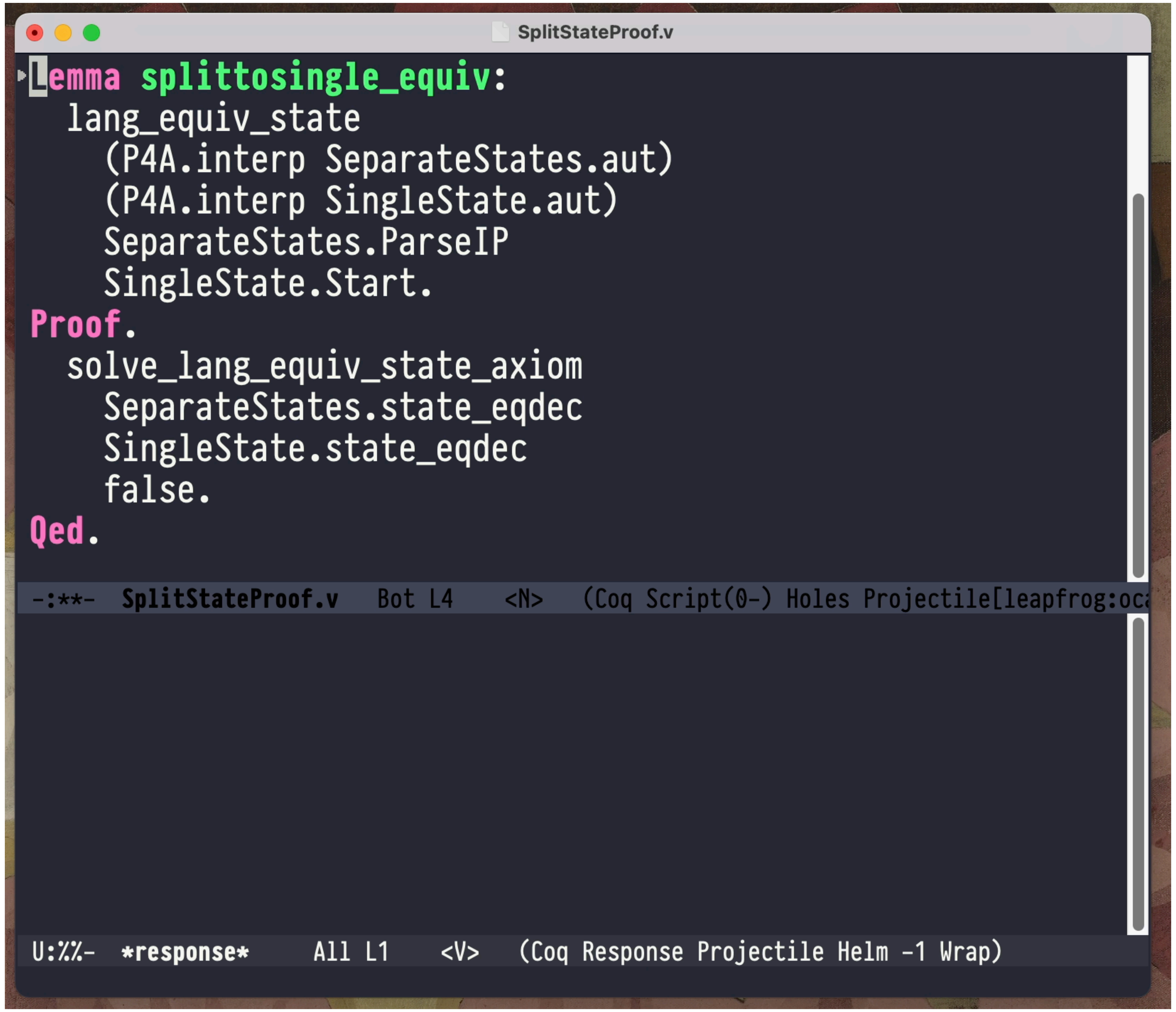
```
Lemma splittosingle_equiv:
  lang_equiv_state
    (P4A.interp SeparateStates.aut)
    (P4A.interp SingleState.aut)
    SeparateStates.ParseIP
    SingleState.Start.
Proof.
  solve_lang_equiv_state_axiom
    SeparateStates.state_eqdec
    SingleState.state_eqdec
    false.
Qed.
```

```
-:**-  SplitStateProof.v   Bot L4    <N>   (Coq Script(0-) Holes Projectile[leapfrog:oca
```

```
U:%%-  *response*     All L1    <V>   (Coq Response Projectile Helm -1 Wrap)
```

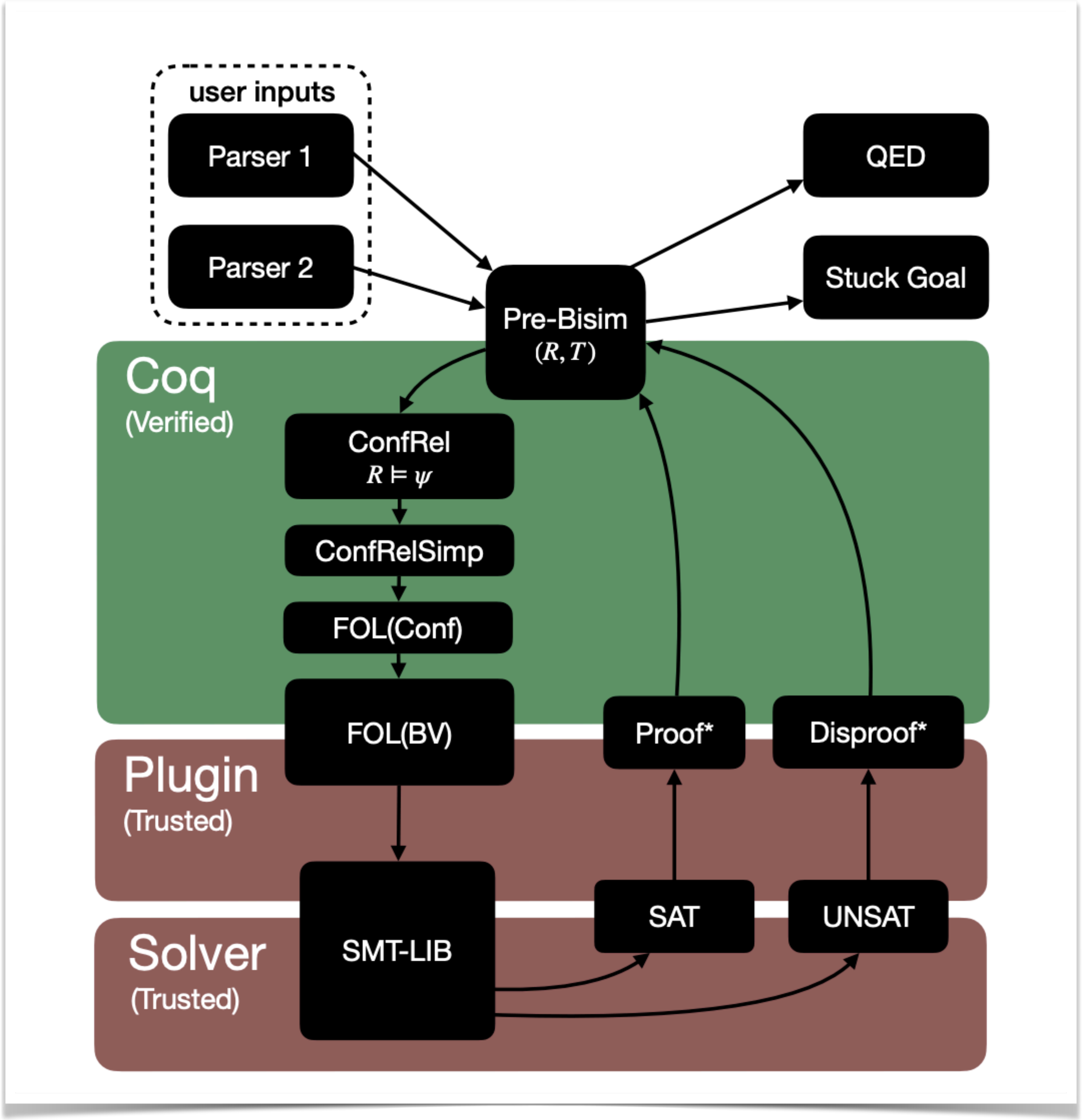
- ~11k lines of Coq, ~1k lines of OCaml
- Library of P4A syntax and semantics
- Push-button **tactic** interface
- Coq plugin for invoking Z3/CVC4
- Logic for verification conditions + **verified lowering** to theory of bitvectors
- ***Soundness proofs*** for all optimizations

Implementation

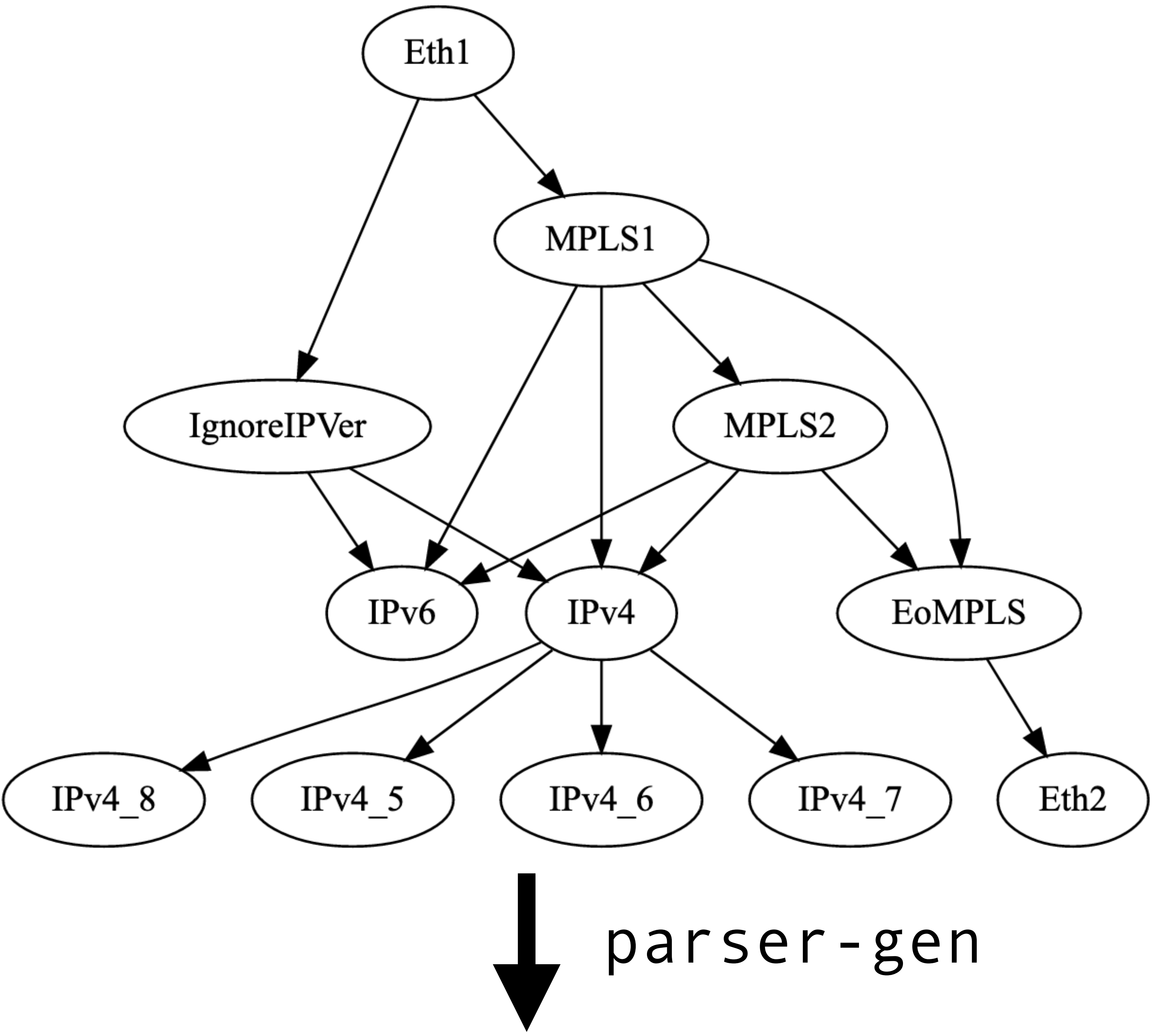


- ~11k lines of Coq, ~1k lines of OCaml
- Library of P4A syntax and semantics
- Push-button **tactic** interface
- Coq plugin for invoking Z3/CVC4
- Logic for verification conditions + **verified lowering** to theory of bitvectors
- **Soundness proofs** for all optimizations

Coq Implementation

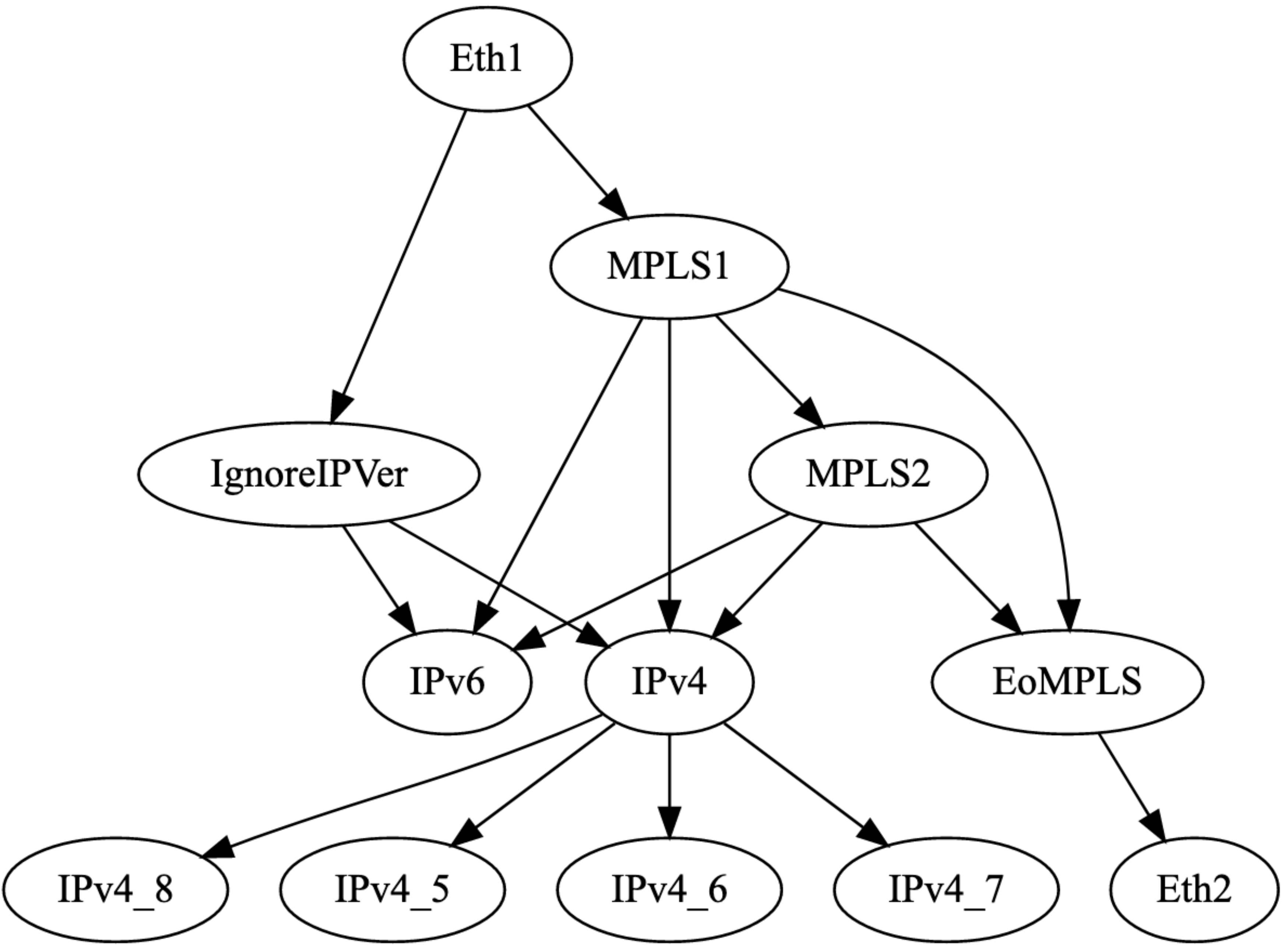


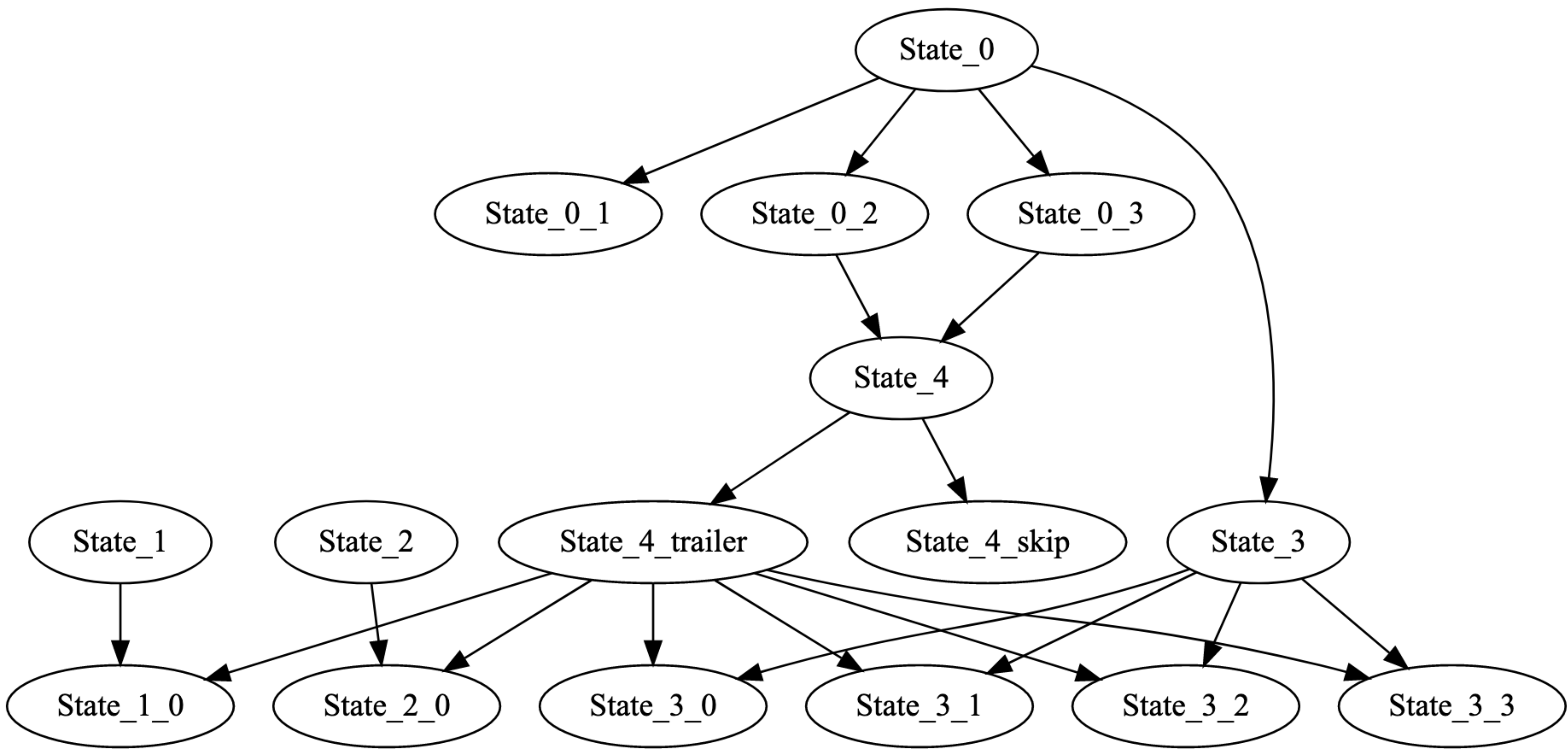
Case study: parser-gen [Gibb et al. 2013]




```
Match: ([ff, 00, 00, ff, ff, 00, 00, 00, 00], [00, 00, 00, 08, 00, 00, 00, 00, 00])  Next-State: 3/255   Adv:  14  Next-Lookup: [0, 0, 0, 0]
Match: ([ff, 00, 00, ff, ff, 00, 00, 00, 00], [00, 00, 00, 88, 47, 00, 00, 00, 00])  Next-State: 4/255   Adv:  16  Next-Lookup: [0, 2, 4, 6]
Match: ([ff, 01, 00, 00, 00, 01, 00, f0, 00], [04, 00, 00, 00, 00, 01, 00, 00, 00])  Next-State: 1/255   Adv:   6  Next-Lookup: [0, 0, 0, 0]
Match: ([ff, 01, 00, f0, 00, 00, 00, 00, 00], [04, 01, 00, 00, 00, 00, 00, 00, 00])  Next-State: 1/255   Adv:   2  Next-Lookup: [0, 0, 0, 0]
...
Match: ([ff, 00, 00, 00, 00, 00, 00, 00, 00], [04, 00, 00, 00, 00, 00, 00, 00, 00])  Next-State: 255/255 Adv:  2   Next-Lookup: [0, 0, 0, 0]
```

Case study: parser-gen [Gibb et al. 2013]



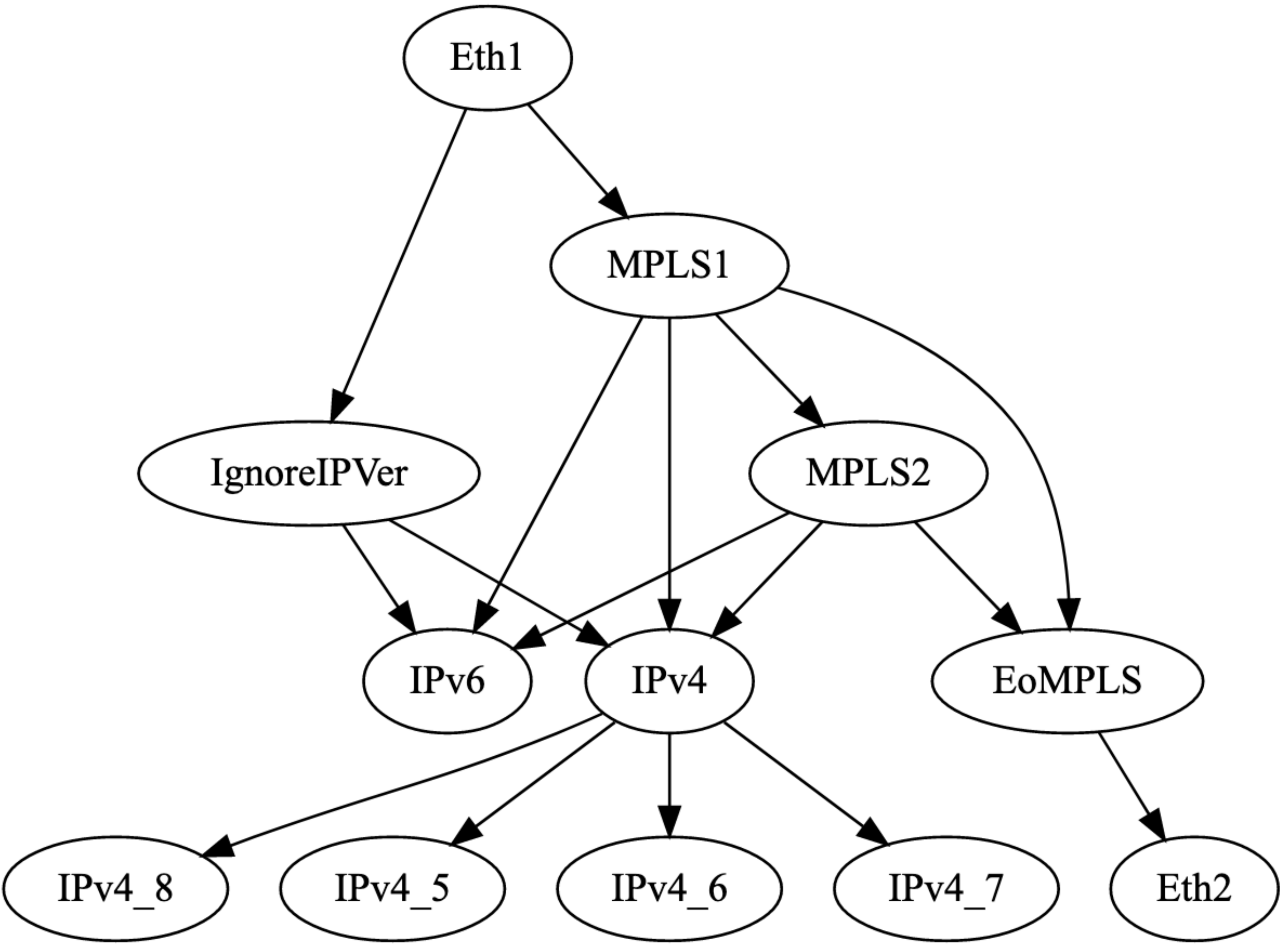


↓ parser-gen

↑ Back-translation

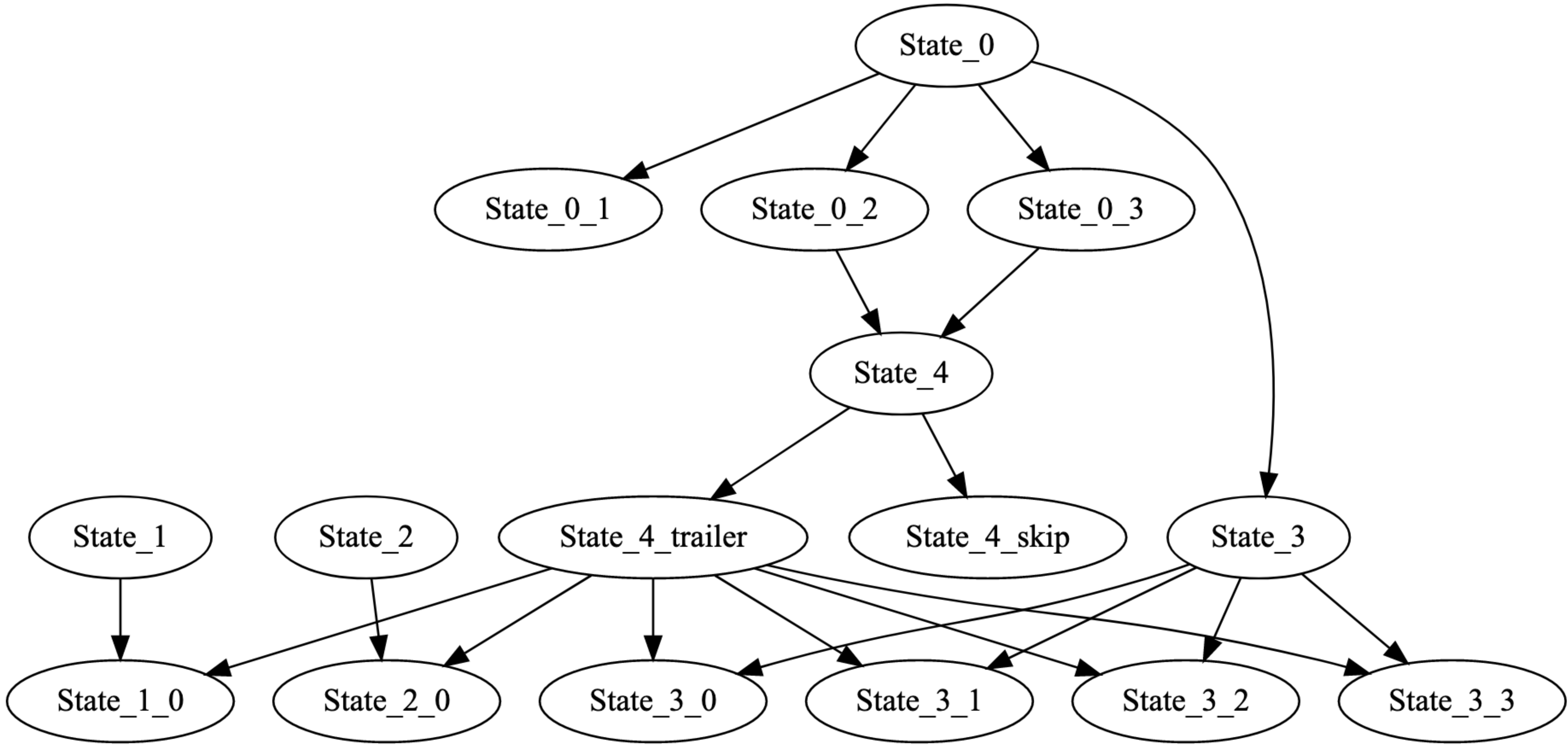
```
Match: ([ff, 00, 00, ff, ff, 00, 00, 00, 00], [00, 00, 00, 08, 00, 00, 00, 00, 00])  Next-State: 3/255   Adv:  14  Next-Lookup: [0, 0, 0, 0]
Match: ([ff, 00, 00, ff, ff, 00, 00, 00, 00], [00, 00, 00, 88, 47, 00, 00, 00, 00])  Next-State: 4/255   Adv:  16  Next-Lookup: [0, 2, 4, 6]
Match: ([ff, 01, 00, 00, 00, 01, 00, f0, 00], [04, 00, 00, 00, 00, 01, 00, 00, 00])  Next-State: 1/255   Adv:   6  Next-Lookup: [0, 0, 0, 0]
Match: ([ff, 01, 00, f0, 00, 00, 00, 00, 00], [04, 01, 00, 00, 00, 00, 00, 00, 00])  Next-State: 1/255   Adv:   2  Next-Lookup: [0, 0, 0, 0]
...
Match: ([ff, 00, 00, 00, 00, 00, 00, 00, 00], [04, 00, 00, 00, 00, 00, 00, 00, 00])  Next-State: 255/255 Adv:  2   Next-Lookup: [0, 0, 0, 0]
```

Case study: parser-gen [Gibb et al. 2013]



≈

Leapfrog



↓ `parser-gen`

↑ Back-translation

```
Match: ([ff, 00, 00, ff, ff, 00, 00, 00, 00], [00, 00, 00, 08, 00, 00, 00, 00, 00])  Next-State: 3/255   Adv:  14  Next-Lookup: [0, 0, 0, 0]
Match: ([ff, 00, 00, ff, ff, 00, 00, 00, 00], [00, 00, 00, 88, 47, 00, 00, 00, 00])  Next-State: 4/255   Adv:  16  Next-Lookup: [0, 2, 4, 6]
Match: ([ff, 01, 00, 00, 00, 01, 00, f0, 00], [04, 00, 00, 00, 00, 01, 00, 00, 00])  Next-State: 1/255   Adv:   6  Next-Lookup: [0, 0, 0, 0]
Match: ([ff, 01, 00, f0, 00, 00, 00, 00, 00], [04, 01, 00, 00, 00, 00, 00, 00, 00])  Next-State: 1/255   Adv:   2  Next-Lookup: [0, 0, 0, 0]
...
Match: ([ff, 00, 00, 00, 00, 00, 00, 00, 00], [04, 00, 00, 00, 00, 00, 00, 00, 00])  Next-State: 255/255 Adv:  2   Next-Lookup: [0, 0, 0, 0]
```