Secure Programs via Game-based Synthesis

Somesh Jha, Tom Reps, and **Bill Harris**

THE UNIVERSITY of WISCONSIN MADISON

One-slide summary

- Secure programming on a conventional OS is *intractable*
- Privilege-aware OS's take secure programming from *intractable* to *challenging*
- Our program rewriter takes secure programming from *challenging* to *simple*

Outline

1. Motivation, problem statement
2. Previous work: Capsicum [CAV '12, Oakland '13]
3. Ongoing work: HiStar
4. Open challenges

Outline

1. Motivation, problem statement

Secure Programming is *Intractable*

- 81 exploits in CVE since Sept. 2013
- Many exploit a software bug
  to carry out undesirable system operations
  - 2013-5751: exploit SAP NetWeaver
    to traverse a directory
  - 2013-5979: exploit bad filename handling in
    Xibo to read arbitrary files
  - 2013-5725: exploit ByWord
    to overwrite files

How to Carry Out an Exploit

software vulnerability

+

OS privilege

=

security exploit

The Conventional-OS Solution

software vulnerability

+

OS privilege

=

security exploit

The Conventional-OS Solution

software vulnerability

+

~~OS privilege~~

=

security exploit

The Conventional-OS Solution

software vulnerability

+

~~OS privilege~~

≠

security exploit

The Program-Verification Solution

software vulnerability

+

OS privilege

=

security exploit

The Program-Verification Solution

~~software vulnerability~~

+

OS privilege

=

security exploit

The Program-Verification Solution

~~software vulnerability~~

+

OS privilege

≠

security exploit

Priv.-aware OS

- Introduce **explicit** privileges over all system objects, **primitives** that update privileges
- Programs call primitives to manage privilege

The Priv.-aware OS Solution

software vulnerability

+

OS privilege

=

security exploit

The Priv.-aware OS Solution

(software vulnerability

+

primitives)

+

OS privilege monitor

=

security exploit

The Priv.-aware OS Solution

$$\left(\begin{array}{c}\text{software vulnerability}\\ +\\ \text{primitives}\end{array}\right)$$

\+

OS privilege monitor

≠

security exploit

The Capsicum Priv.-aware OS [Watson '10]

- Privilege: **ambient authority (Amb)** to open descriptors to system objects
- Primitives: program calls `cap_enter()` to manage Amb

Rules of Capsicum's Amb

Rules of Capsicum's Amb

1. When a process is created, it has the Amb value of its parent

Rules of Capsicum's Amb

1. When a process is created,
   it has the Amb value of its parent
2. After a process calls `cap_enter()`,
   it does not have Amb

Rules of Capsicum's Amb

1. When a process is created, it has the Amb value of its parent
2. After a process calls `cap_enter()`, it does not have Amb
3. If a process does not have Amb, then it can never obtain Amb

gzip

```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f);
L1: compress(in, out);
}
```

gzip

```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f);
L1: compress(in, out);
}
```

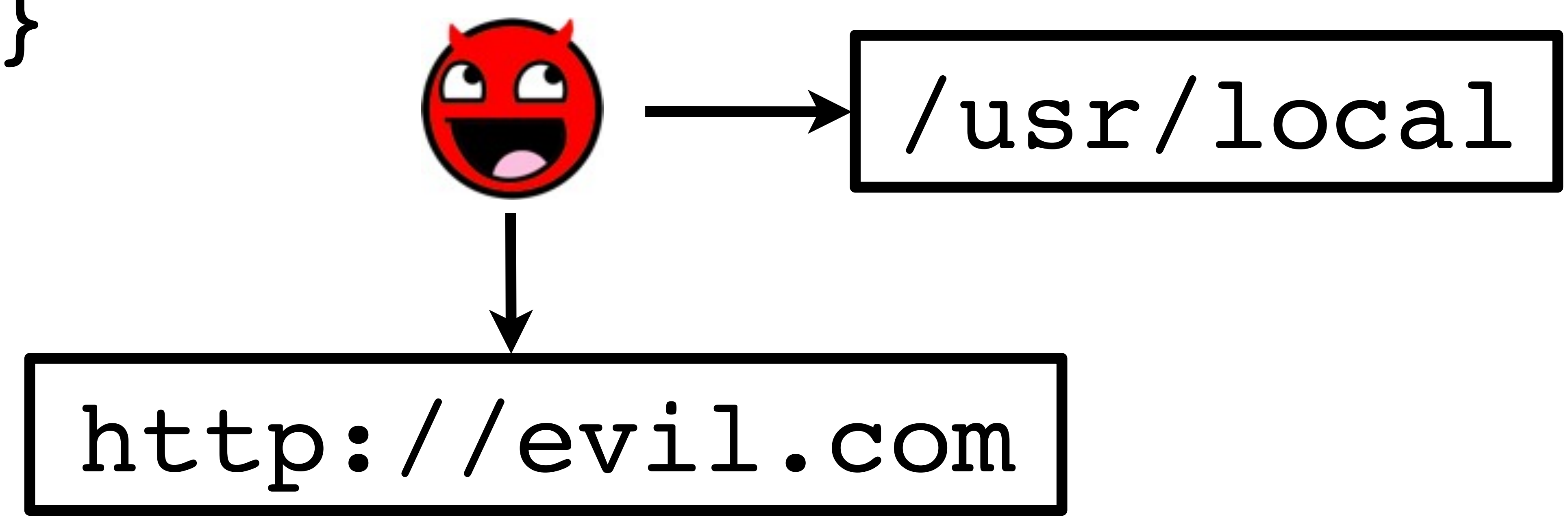
gzip

```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f);
L1: compress(in, out);
}
```

http://evil.com

gzip

```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f);
L1: compress(in, out);
}
```



A simple `gzip` policy

- When `gzip` calls `open2()` at `L0`, it should be able to open descriptors
- When `gzip` calls `compress()` at `L1`, it should not able to open descriptors

A simple `gzip` policy with AMB

- When `gzip` calls `open2()` at `L0`, it should have AMB
- When `gzip` calls `compress()` at `L1`, it should not have AMB

`gzip` with AMB

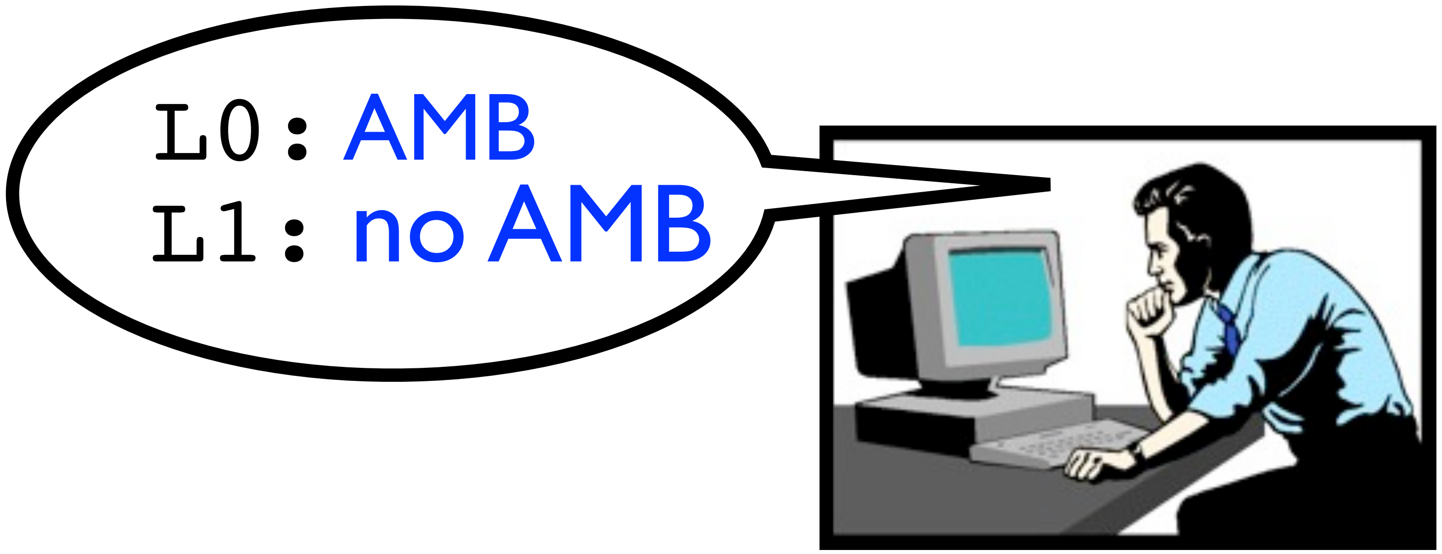
```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f);
L1: compress(in, out);
}
```

gzip with AMB

```
main() {
  file_nms = parse_cl();
  for (f in file_nms):
L0: (in, out) = open2(f);
L1: compress(in, out);
}
```

gzip with AMB

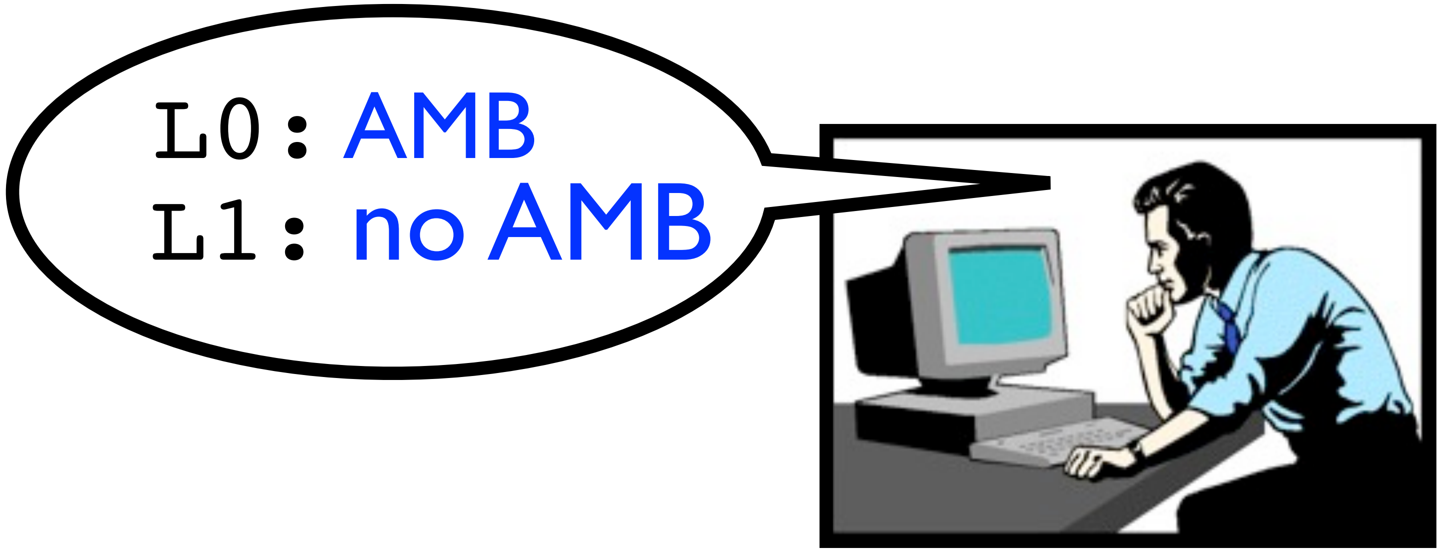
```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f);
L1: compress(in, out);
}
```



gzip with AMB

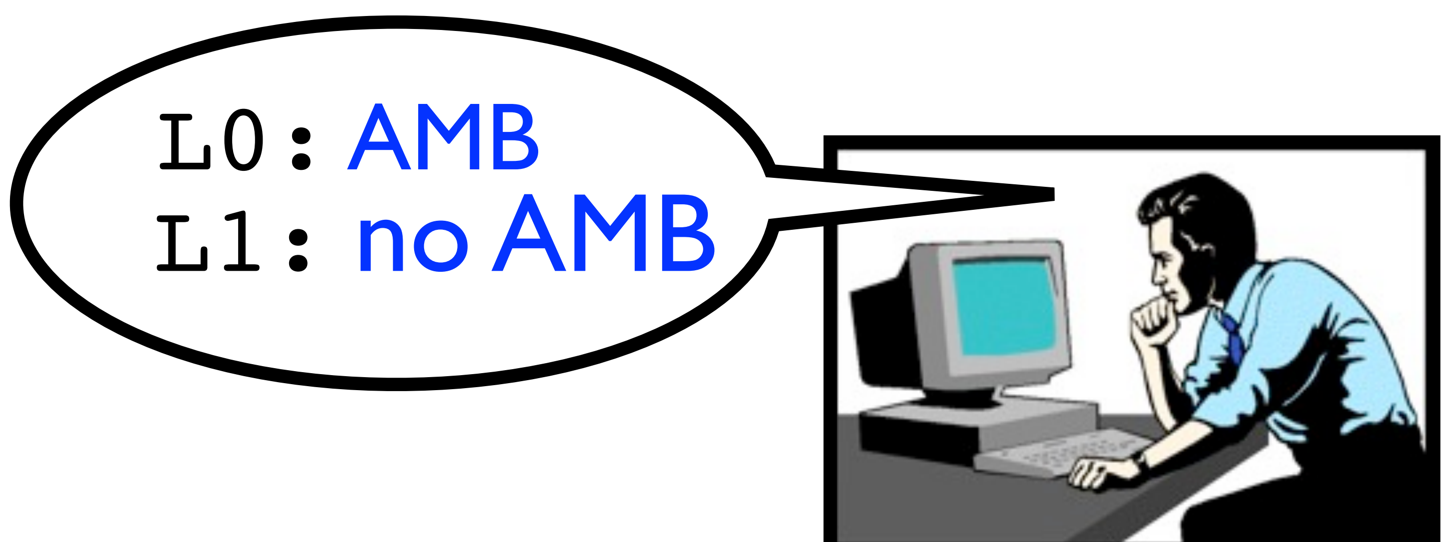
```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f);
L1: compress(in, out);
}
```

cap_enter()



gzip with AMB

```
main() {
   file_nms = parse_cl();
   for (f in file_nms):        ?  cap_enter()
L0: (in, out) = open2(f);      ?
L1: compress(in, out);
}
```



Capsicum Programming Challenges

1. Amb policies are not explicit
2. cap_enter primitive has subtle temporal effects

gzip
Programming Challenges

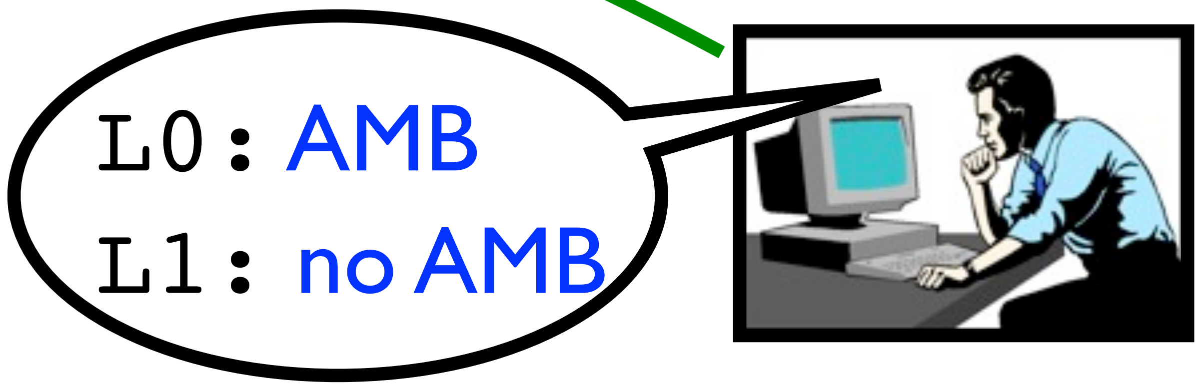
```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f);

L1: compress(in, out);
}
```

L0: AMB
L1: no AMB

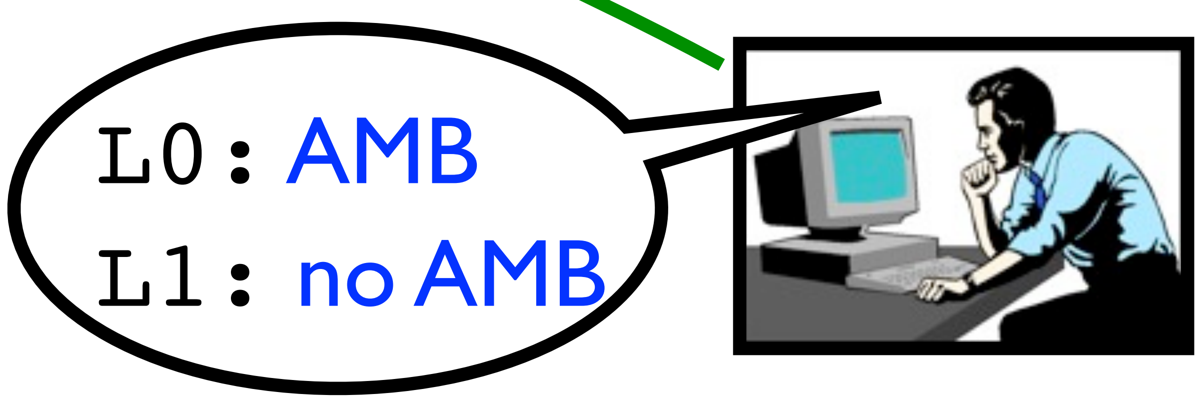
gzip
Programming Challenges

```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f);
     cap_enter();
L1: compress(in, out);
}
```



gzip Programming Challenges

```
main() {
   file_nms = parse_cl(); AMB
   for (f in file_nms):
L0: (in, out) = open2(f);
     cap_enter();
L1: compress(in, out);
}
```



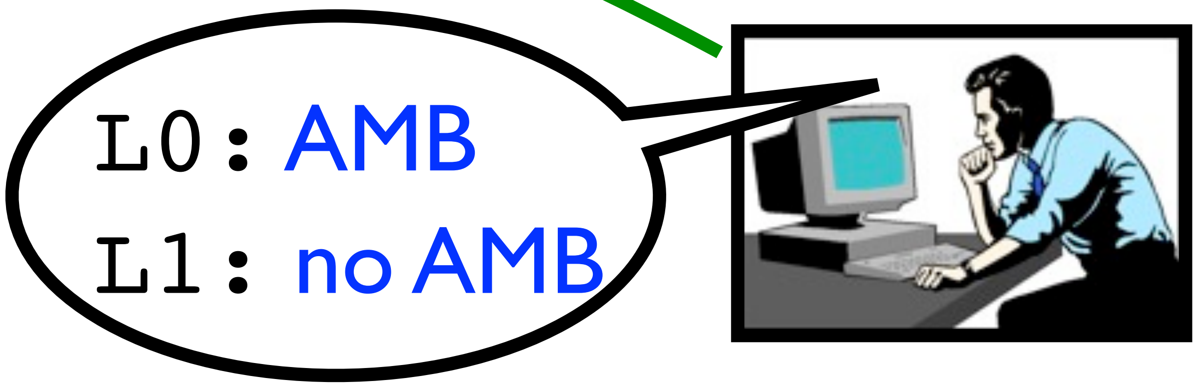
gzip
Programming Challenges

```
main() {
   file_nms = parse_cl();
   for (f in file_nms):       AMB
L0: (in, out) = open2(f);
     cap_enter();
L1: compress(in, out);
}
```

L0: AMB
L1: no AMB

gzip
Programming Challenges

```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f); AMB
     cap_enter();
L1: compress(in, out);
}
```



gzip
Programming Challenges

```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f);
     cap_enter();
L1: compress(in, out);
}
```

no AMB

L0: AMB
L1: no AMB

gzip Programming Challenges

```
main() {
  file_nms = parse_cl();
  for (f in file_nms):
L0: (in, out) = open2(f);
    cap_enter();
L1: compress(in, out);
}
```

no AMB

L0: AMB

L1: no AMB

gzip
Programming Challenges

```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f);  no AMB
     cap_enter();
L1: compress(in, out);
}
```

L0: AMB
L1: no AMB

Rules of Capsicum's Amb

1. When a process is created, it has the AMB value of its parent
2. After a process calls `cap_enter()`, it never has AMB
3. If a process does not have Amb, then it can never obtain Amb

Rules of Capsicum's Amb

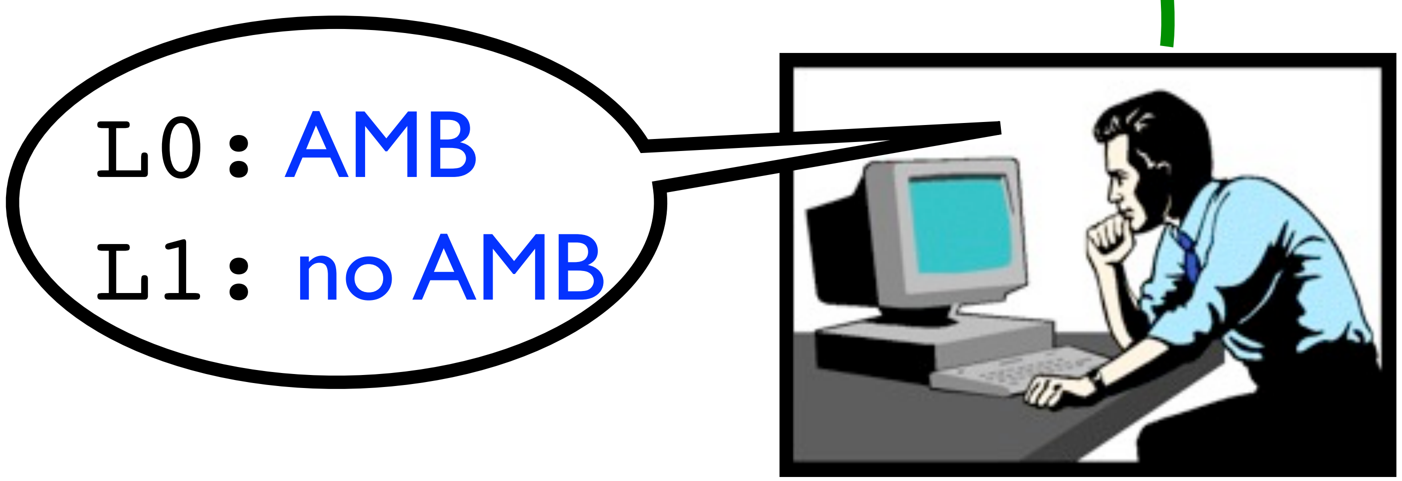
1. When a process is created,
   it has the AMB value of its parent

Instrumenting `gzip`

```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f);

     cap_enter();
L1: compress(in, out);

}
```



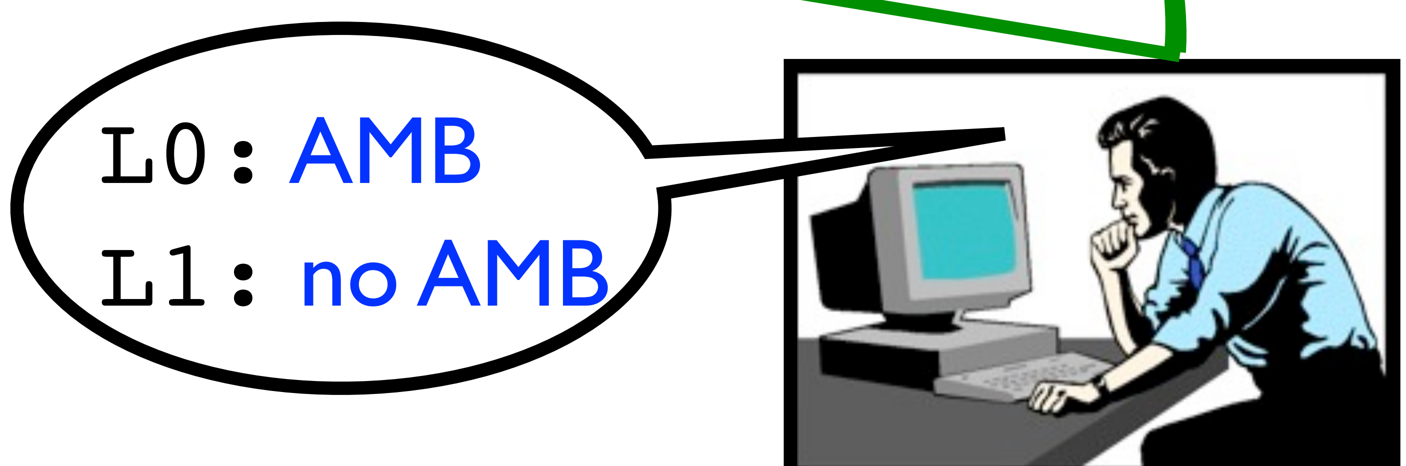
Instrumenting `gzip`

```
main() {
  file_nms = parse_cl();
  for (f in file_nms):
L0: (in, out) = open2(f);
    sync_fork();
    cap_enter();
L1: compress(in, out);
    sync_join();
}
```

L0: AMB
L1: no AMB

Instrumenting `gzip`

```
main() {
   file_nms = parse_cl(); AMB
   for (f in file_nms):
L0: (in, out) = open2(f);
      sync_fork();
      cap_enter();
L1: compress(in, out);
      sync_join();
}
```



Instrumenting `gzip`

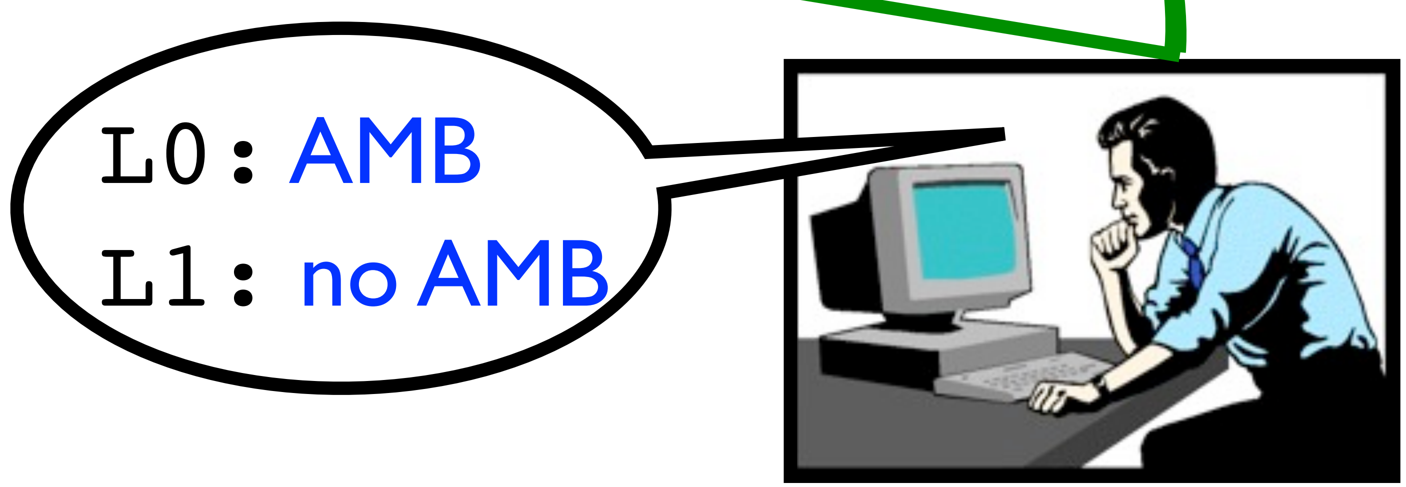
```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f);
      sync_fork();
      cap_enter();
L1: compress(in, out);
      sync_join();
}
```

AMB

L0: AMB
L1: no AMB

Instrumenting gzip

```
main() {
  file_nms = parse_cl();
  for (f in file_nms):
L0: (in, out) = open2(f); AMB
      sync_fork();
      cap_enter();
L1: compress(in, out);
      sync_join();
}
```



Instrumenting `gzip`

```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f);
      sync_fork();
      cap_enter();
L1: compress(in, out);
      sync_join();
}
```

no AMB

L0: AMB
L1: no AMB

Instrumenting `gzip`

```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f);
       sync_fork();
       cap_enter();
L1: compress(in, out);
       sync_join();
}
```

AMB

L0: AMB
L1: no AMB

Instrumenting `gzip`

```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f); AMB
      sync_fork();
      cap_enter();
L1: compress(in, out);
      sync_join();
}
```

L0: AMB
L1: no AMB

Capsicum Challenges Not Appearing in This Talk

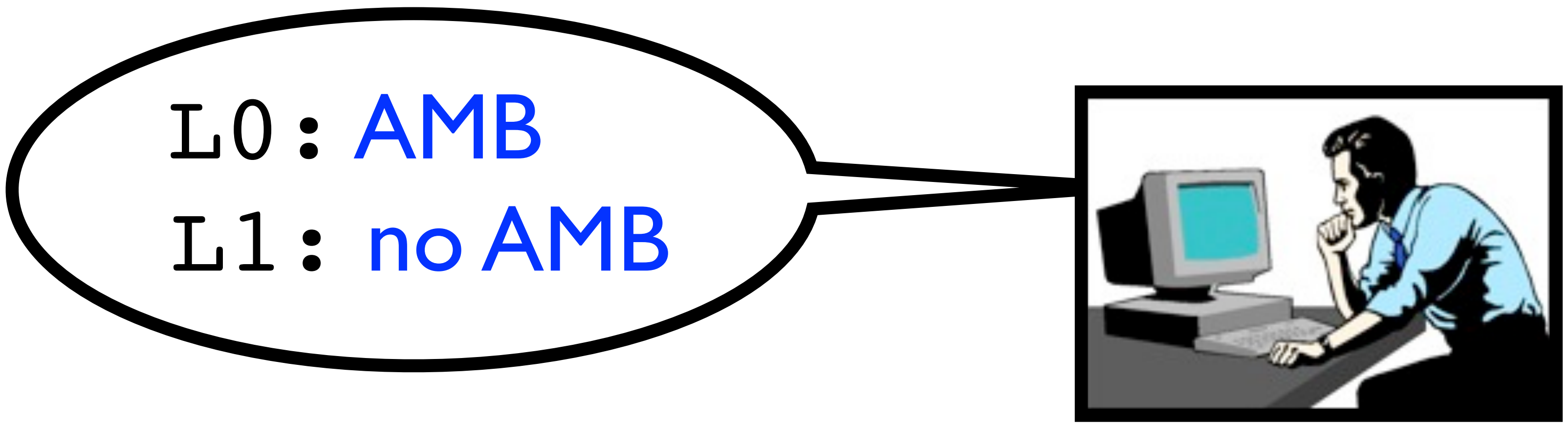
- Program can construct **capability** from each UNIX descriptor
- Capability has a vector of 63 **access rights** (~1 for every system call on a descriptor)
- Programs can assume new capabilities via a Remote Procedure Call (RPC)

Instrumenting Programs with CapWeave

1. Programmer writes an **explicit** Amb policy
2. **CapWeave** instruments program to invoke primitives so that it satisfies the policy

gzip with CapWeave

```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f);
L1: compress(in, out);
}
```



gzip with CapWeave

```
main() {
   file_nms = parse_cl();
   for (f in file_nms):
L0: (in, out) = open2(f);
L1: compress(in, out);
}
```

Policy

```
Cur(p) => (pc[L0](p) => AMB(p)
          & (pc[L1](p) => !AMB(p))
```

```
main() {
  file_nms = parse_cl();
  for (f in file_nms):
L0: (in, out) = open2(f);
L1: compress(in, out);
}
```

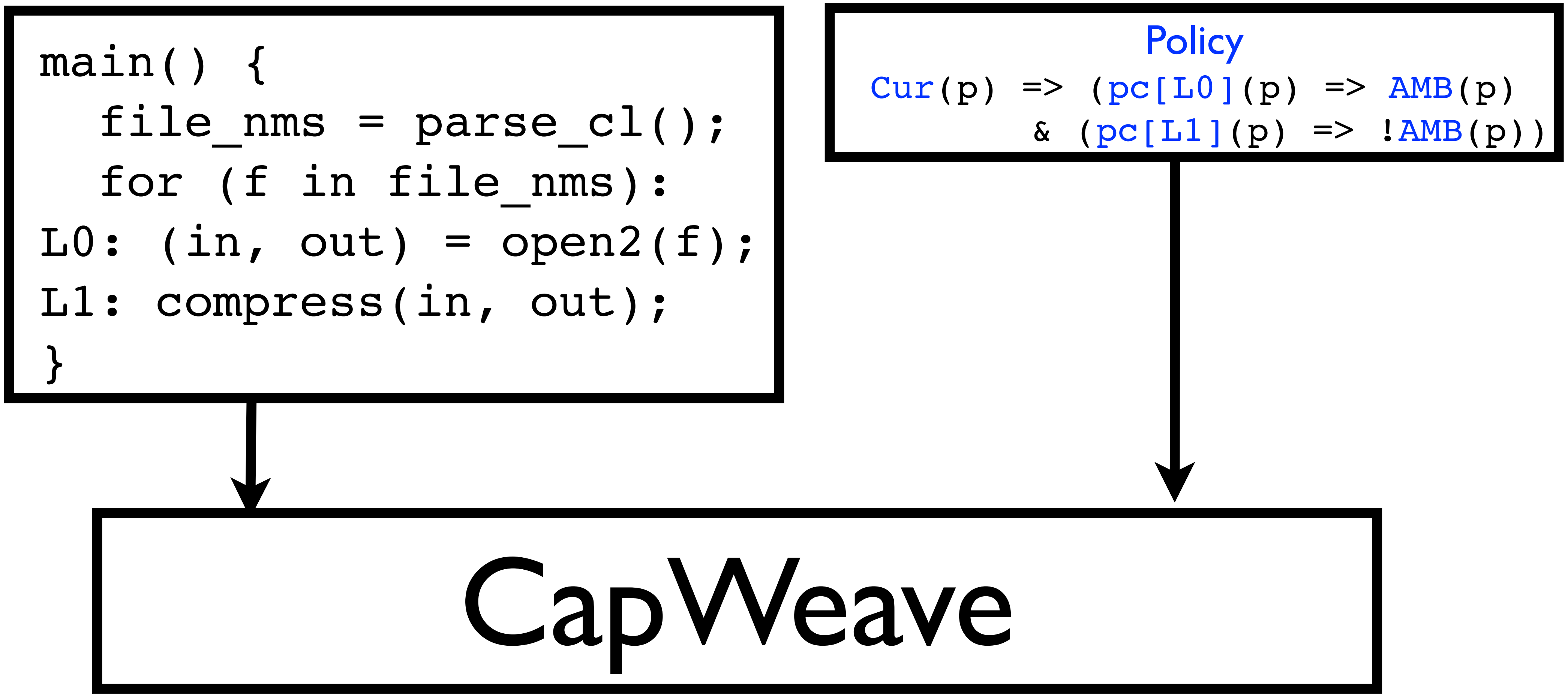
Policy

```
Cur(p) => (pc[L0](p) => AMB(p)
         & (pc[L1](p) => !AMB(p))
```

```
main() {
  file_nms = parse_cl();
  for (f in file_nms):
L0: (in, out) = open2(f);
L1: compress(in, out);
}
```

Policy

```
Cur(p) => (pc[L0](p) => AMB(p)
        & (pc[L1](p) => !AMB(p))
```


main() {
file_nms = parse_cl();
for (f in file_nms):
L0: (in, out) = open2(f);
L1: compress(in, out);
}
Policy
Cur(p) => (pc[L0](p) => AMB(p)
& (pc[L1](p) => !AMB(p))
CapWeave

```
main() {
  file_nms = parse_cl();
  for (f in file_nms):
L0: (in, out) = open2(f);
L1: compress(in, out);
}
```

Policy

```
Cur(p) => (pc[L0](p) => AMB(p)
        & (pc[L1](p) => !AMB(p))
```

CapWeave

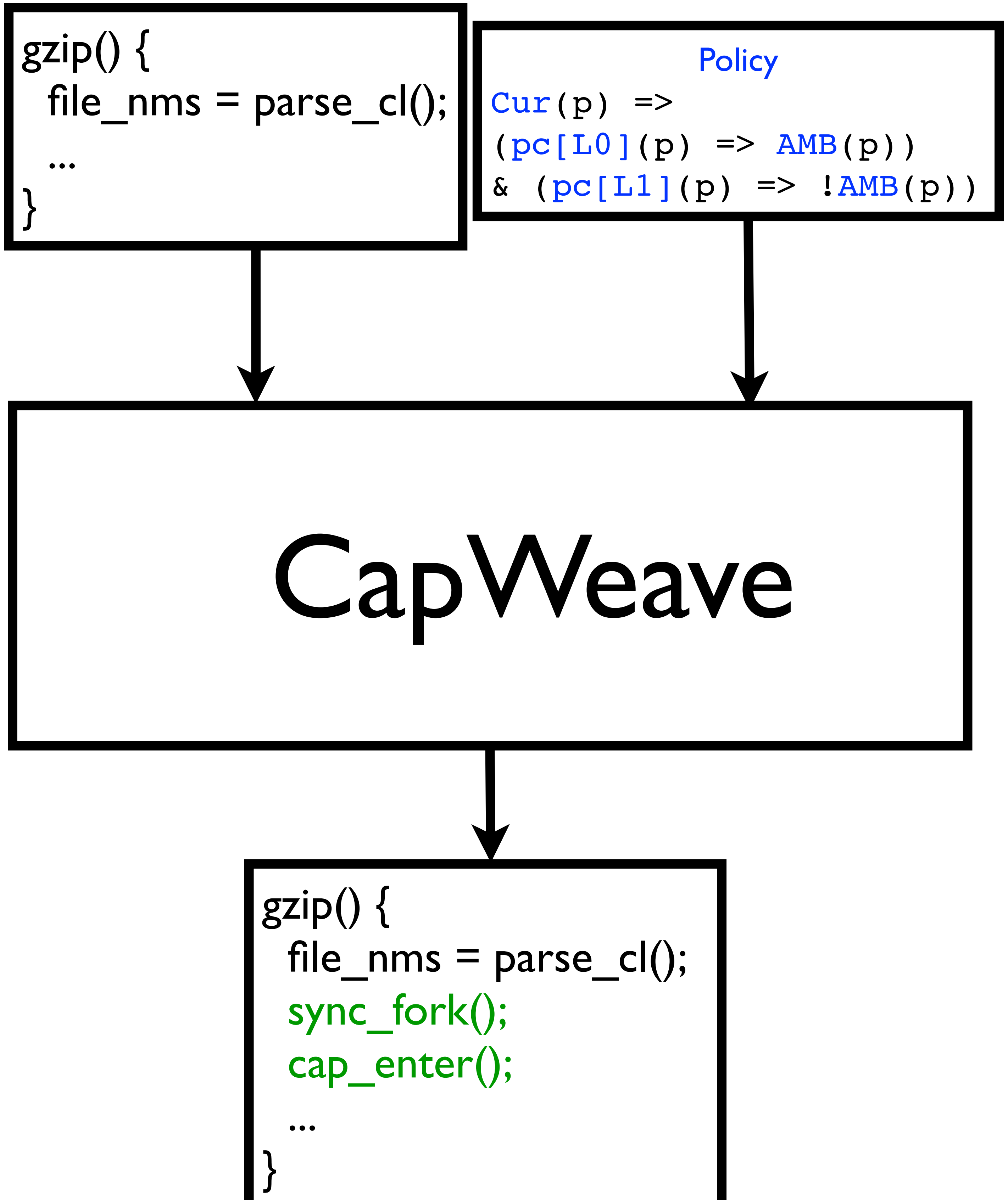
Instrumented Program

```
void main() {
  L0: open2(...);
  sync_fork();
  cap_enter();
  L1: compress();
  sync_join();
}
```

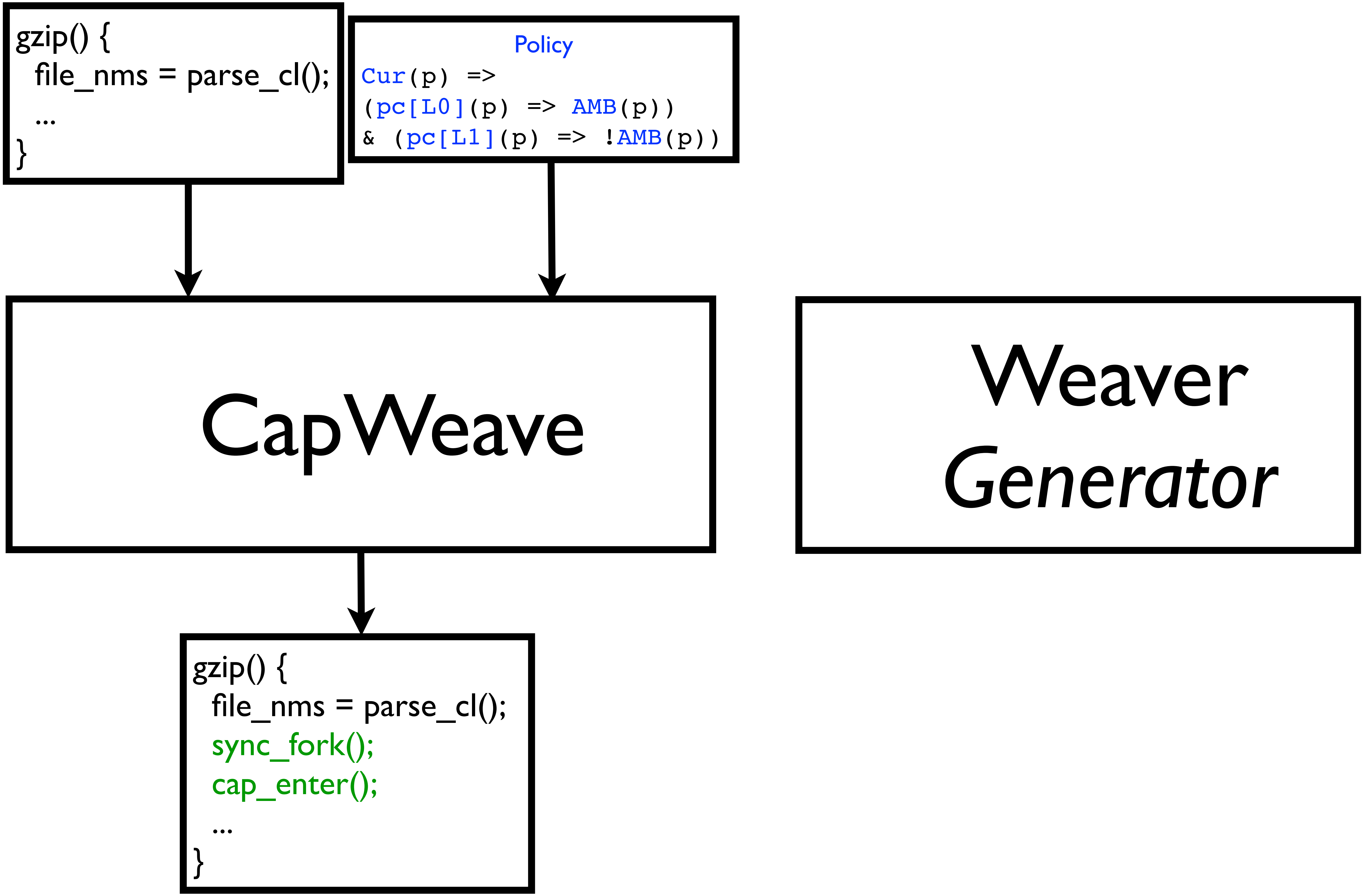
The Next 700 Policy Weavers

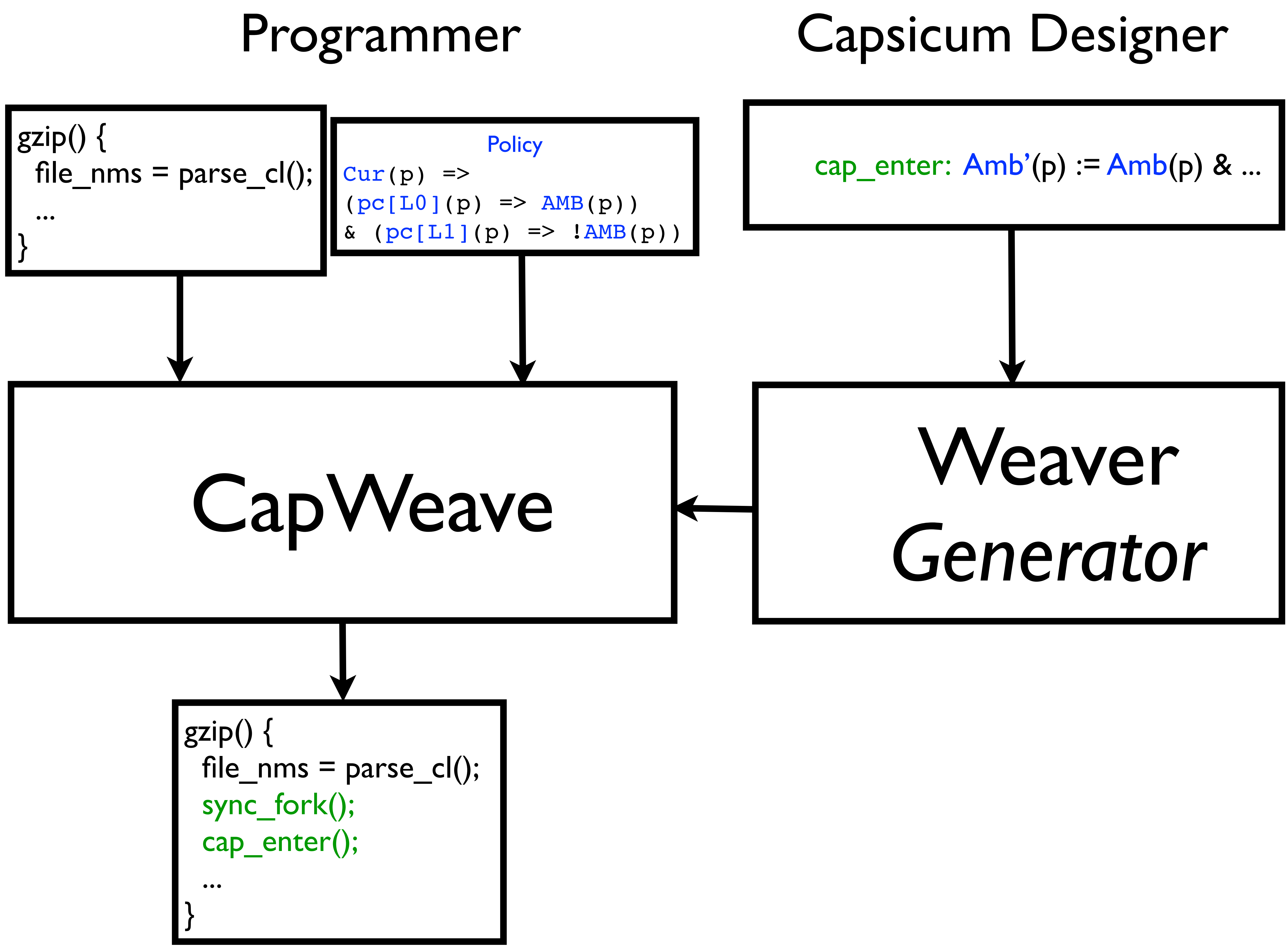
Analogous challenges with **Decentralized Information Flow Control (DIFC)**

- Asbestos [Efstathopoulos ‘05]
- HiStar [Zeldovich ’06]
- Flume [Krohn ‘07]


gzip() {
file_nms = parse_cl();
...
}
Policy
Cur(p) =>
(pc[L0](p) => AMB(p))
& (pc[L1](p) => !AMB(p))
CapWeave
gzip() {
file_nms = parse_cl();
sync_fork();
cap_enter();
...
}

Programmer

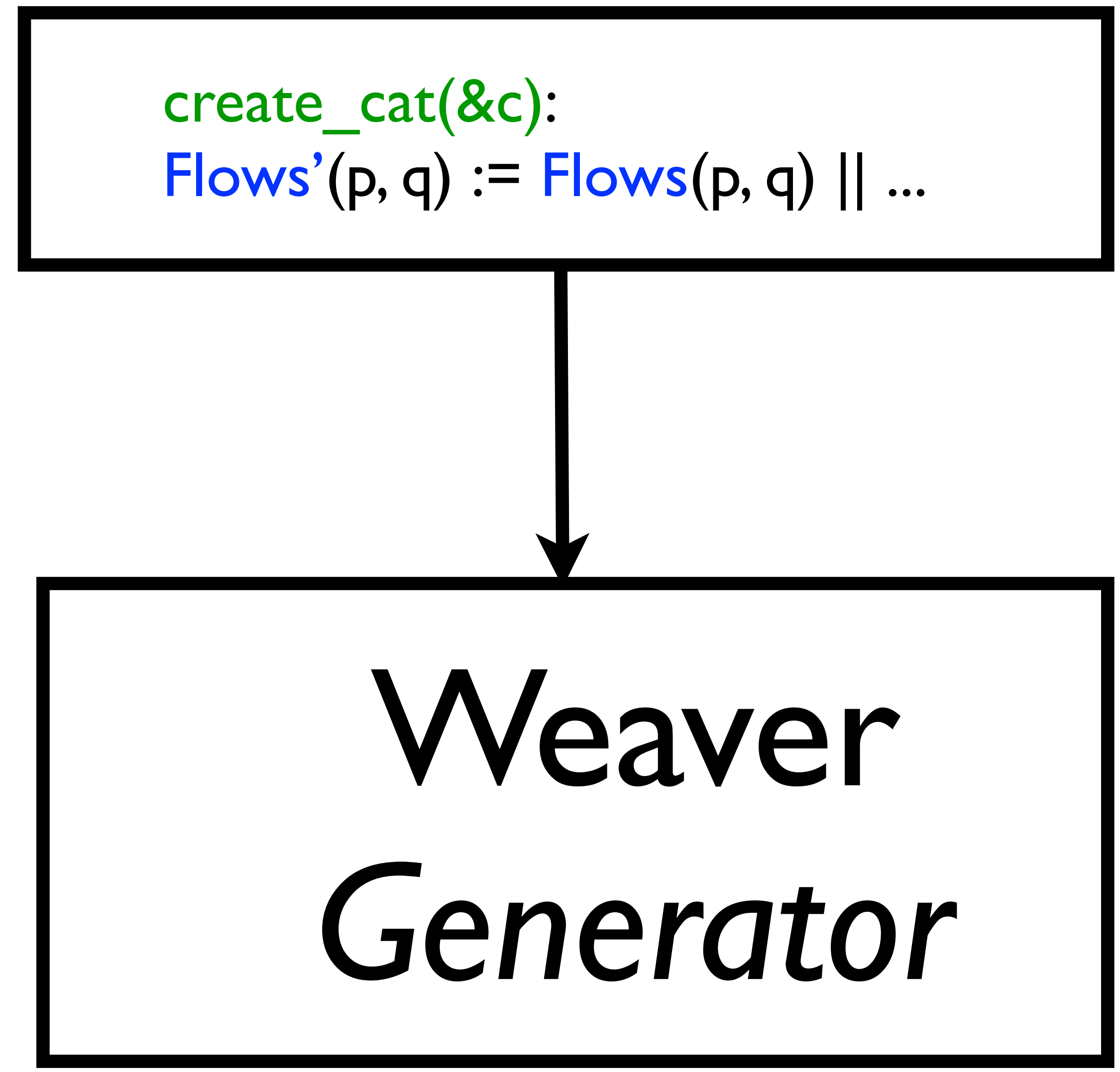

gzip() {
file_nms = parse_cl();
...
}
Policy
Cur(p) =>
(pc[L0](p) => AMB(p))
& (pc[L1](p) => !AMB(p))
CapWeave
Weaver
Generator
gzip() {
file_nms = parse_cl();
sync_fork();
cap_enter();
...
}

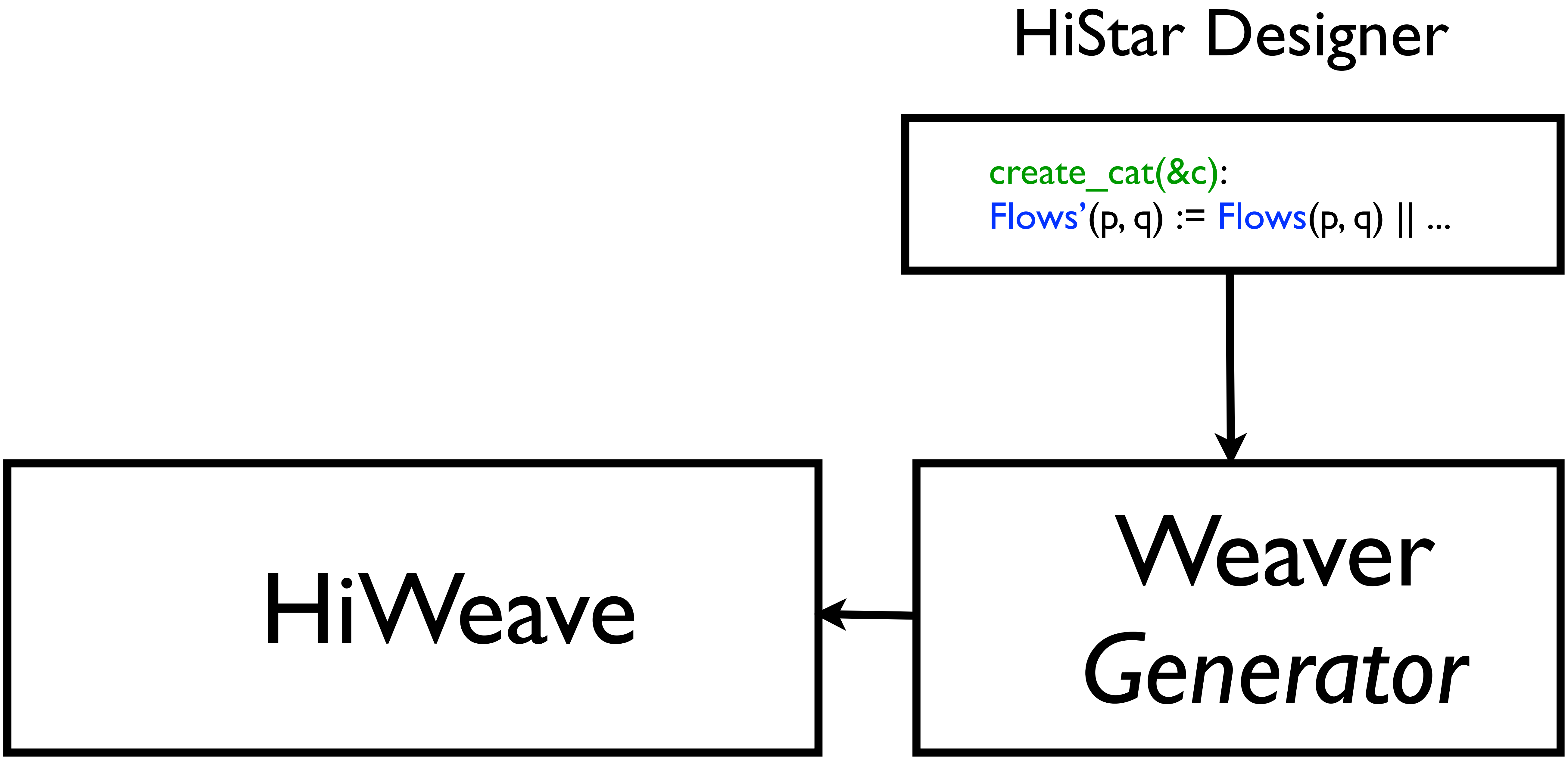

Programmer
Capsicum Designer
gzip() {
file_nms = parse_cl();
...
}
Policy
Cur(p) =>
(pc[L0](p) => AMB(p))
& (pc[L1](p) => !AMB(p))
cap_enter: Amb'(p) := Amb(p) & ...
CapWeave
Weaver
Generator
gzip() {
file_nms = parse_cl();
sync_fork();
cap_enter();
...
}

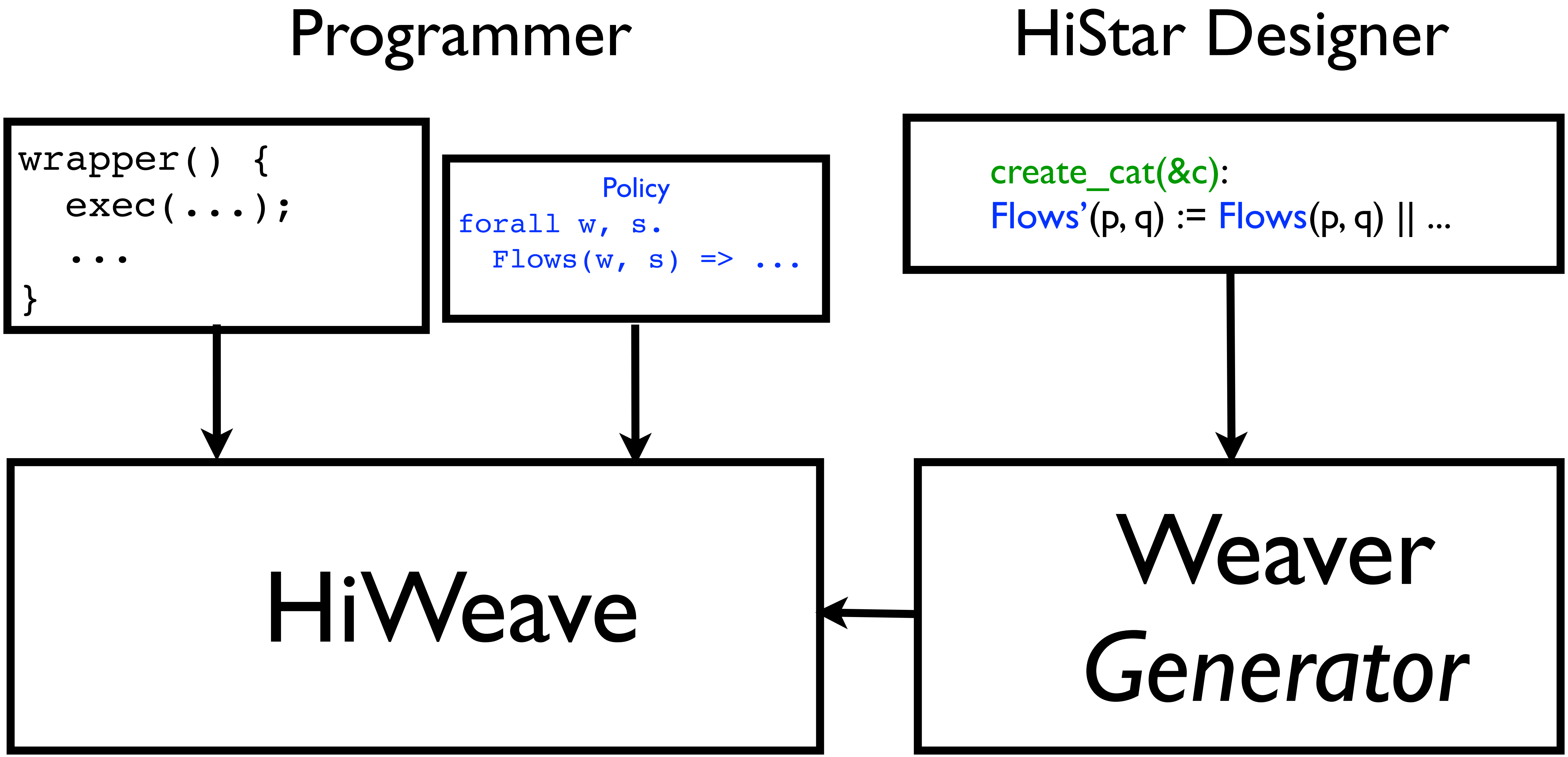
Weaver

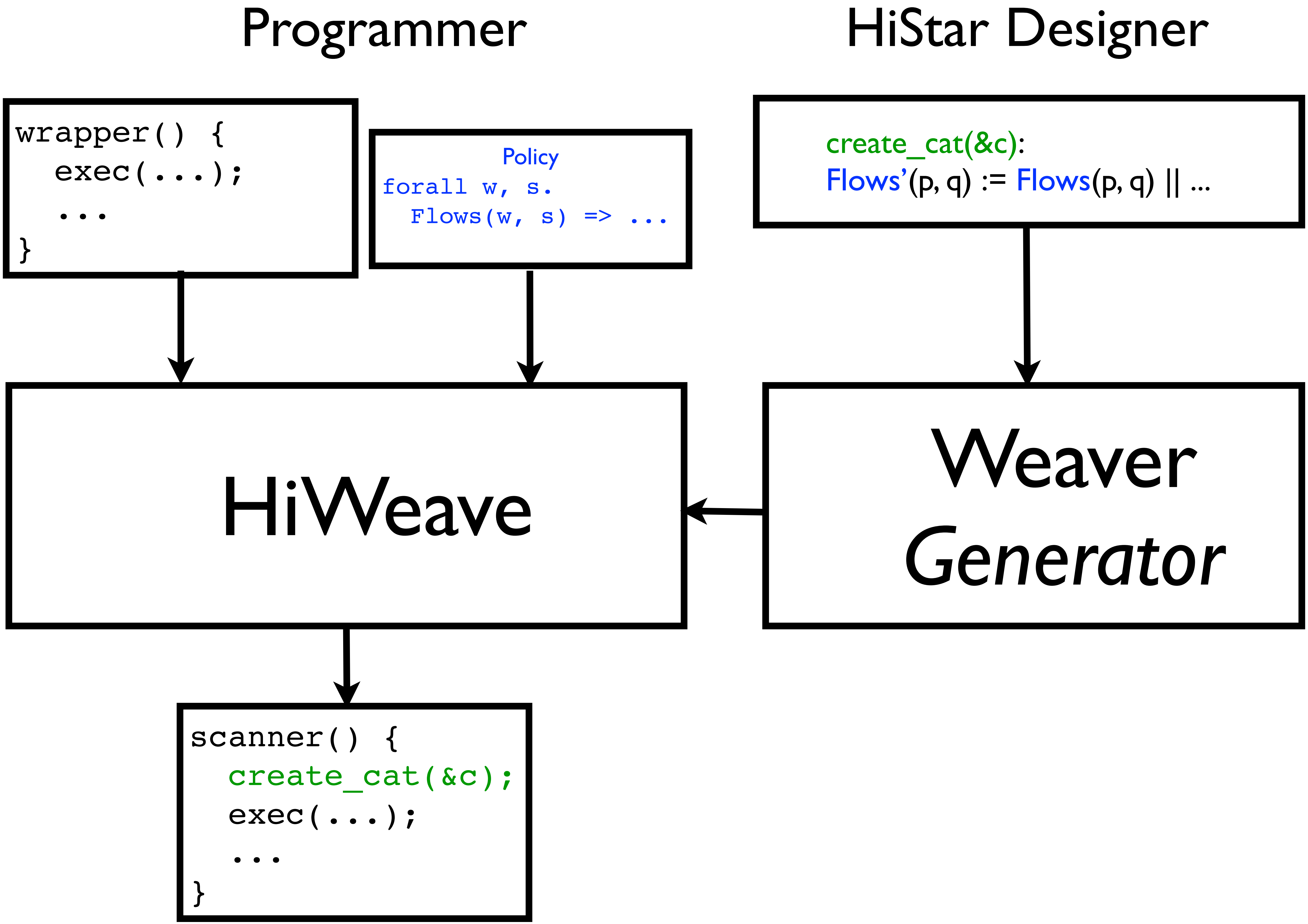
Generator

HiStar Designer




HiStar Designer
create_cat(&c):
Flows'(p, q) := Flows(p, q) || ...
HiWeave
Weaver
Generator


Programmer
HiStar Designer
wrapper() {
exec(...);
...
}
Policy
forall w, s.
Flows(w, s) => ...
create_cat(&c):
Flows'(p, q) := Flows(p, q) || ...
HiWeave
Weaver
Generator


Programmer
HiStar Designer
wrapper() {
exec(...);
...
}
Policy
forall w, s.
Flows(w, s) => ...
create_cat(&c):
Flows'(p, q) := Flows(p, q) || ...
HiWeave
Weaver
Generator
scanner() {
create_cat(&c);
exec(...);
...
}

Outline

1. Motivation, problem statement
2. Previous work: Capsicum
3. Ongoing work: HiStar
4. Open challenges

Outline

2. Previous work: Capsicum

CapWeave Algorithm

CapWeave Algorithm

Inputs: Program P, Amb Policy Q

CapWeave Algorithm

Inputs: Program P, Amb Policy Q

Output: Instrumentation of P that always satisfies Q

CapWeave Algorithm

Inputs: Program P, Amb Policy Q

Output: Instrumentation of P that always satisfies Q

1. Build finite **IP#** $\supseteq$ instrumented runs that violate Q

1. Building IP#: Inputs

Program

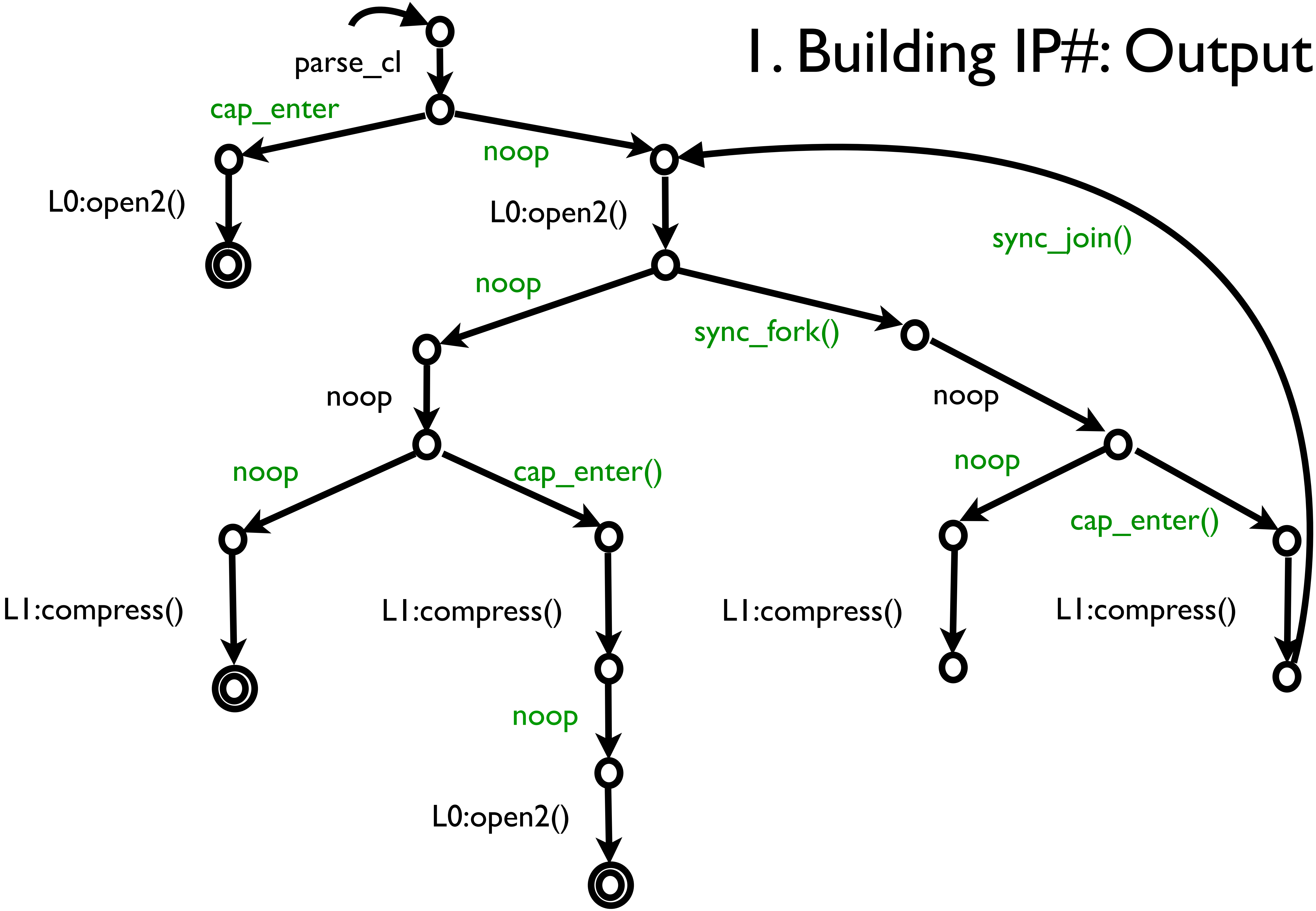
```
main() {
  file_nms = parse_cl();
  for (f in file_nms):
L0: (in, out) = open2(f);
L1: compress(in, out);
}
```

Amb Policy

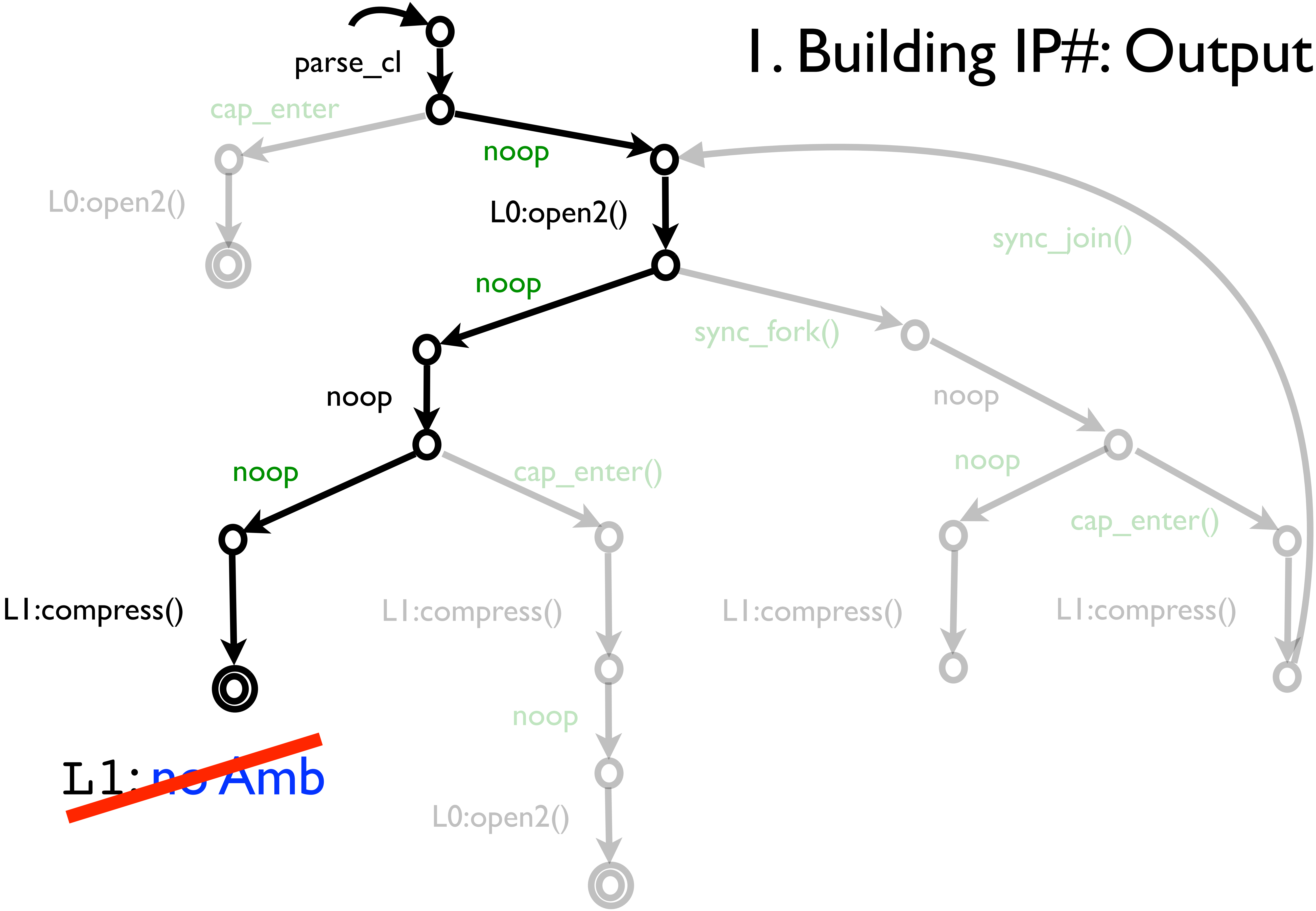
`L0`: Amb

`L1`: no Amb

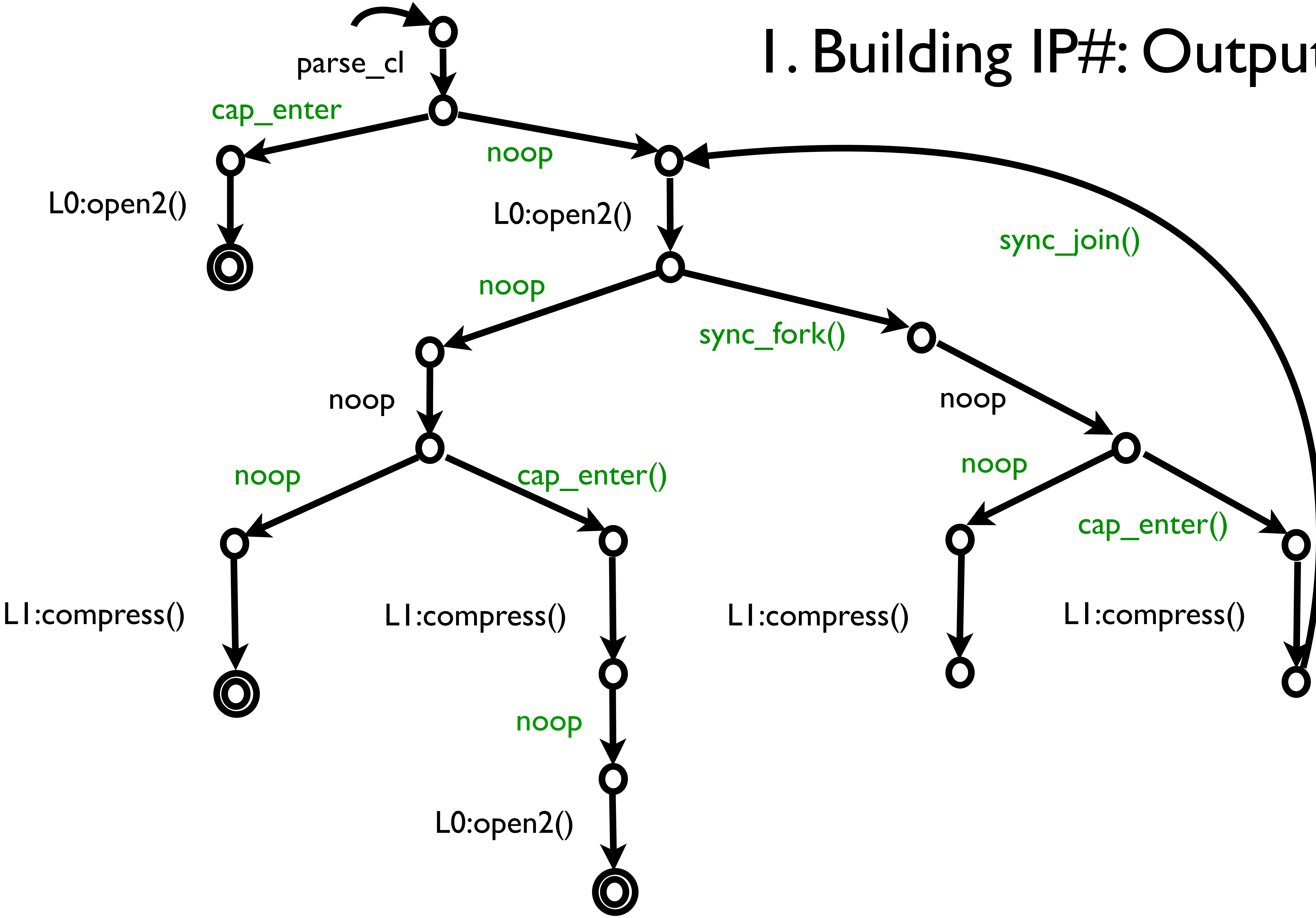
1. Building IP#: Output



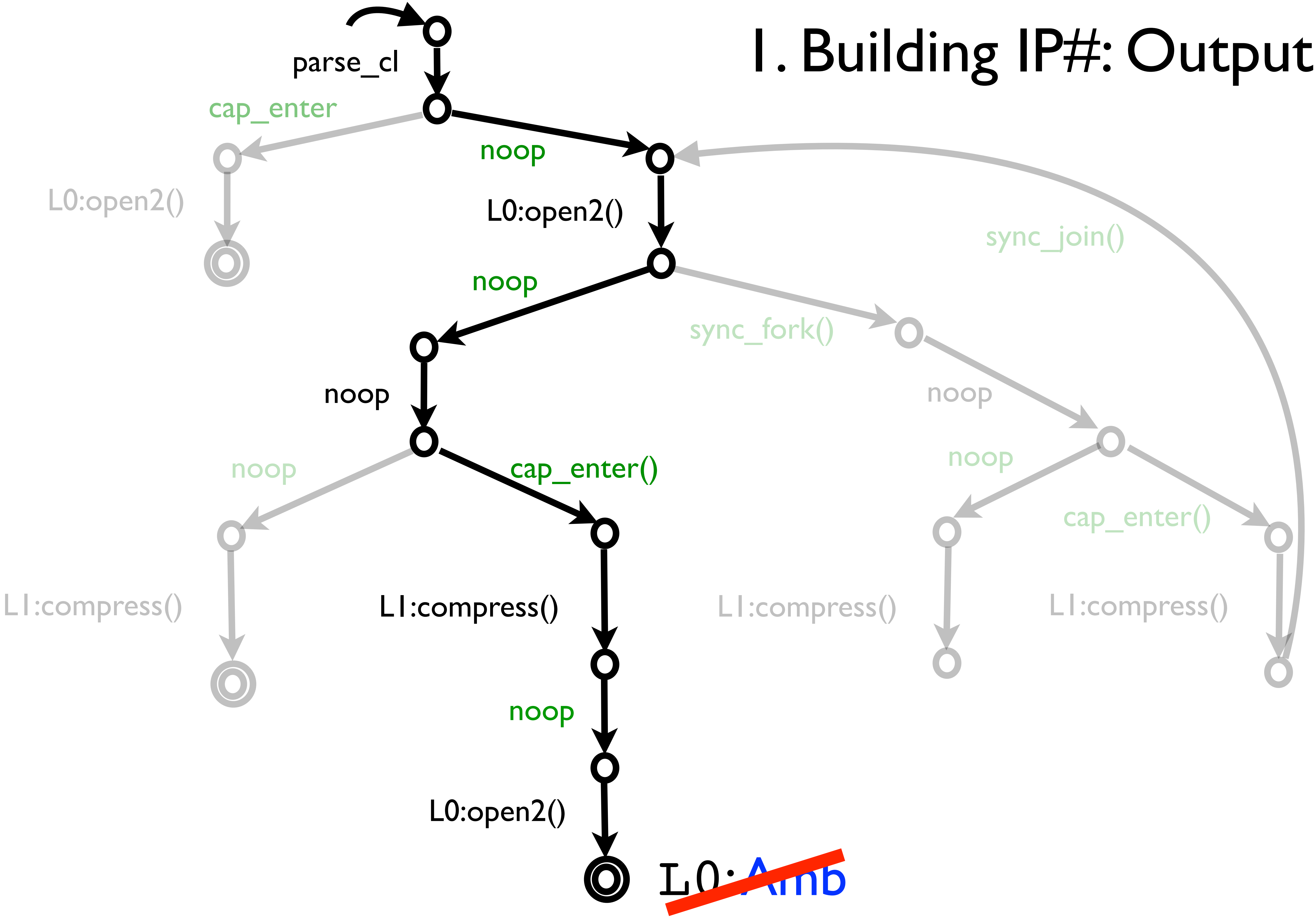
1. Building IP#: Output



1. Building IP#: Output



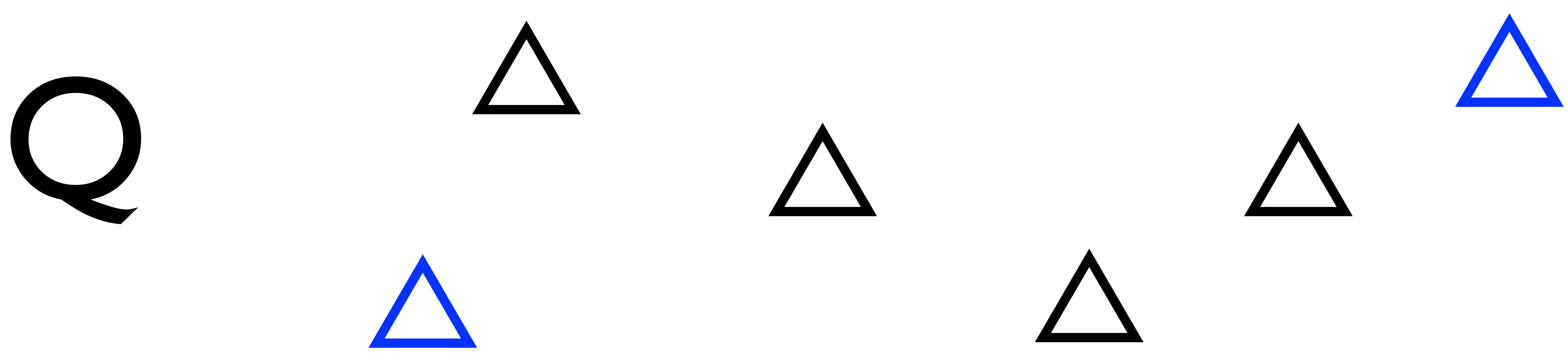
1. Building IP#: Output



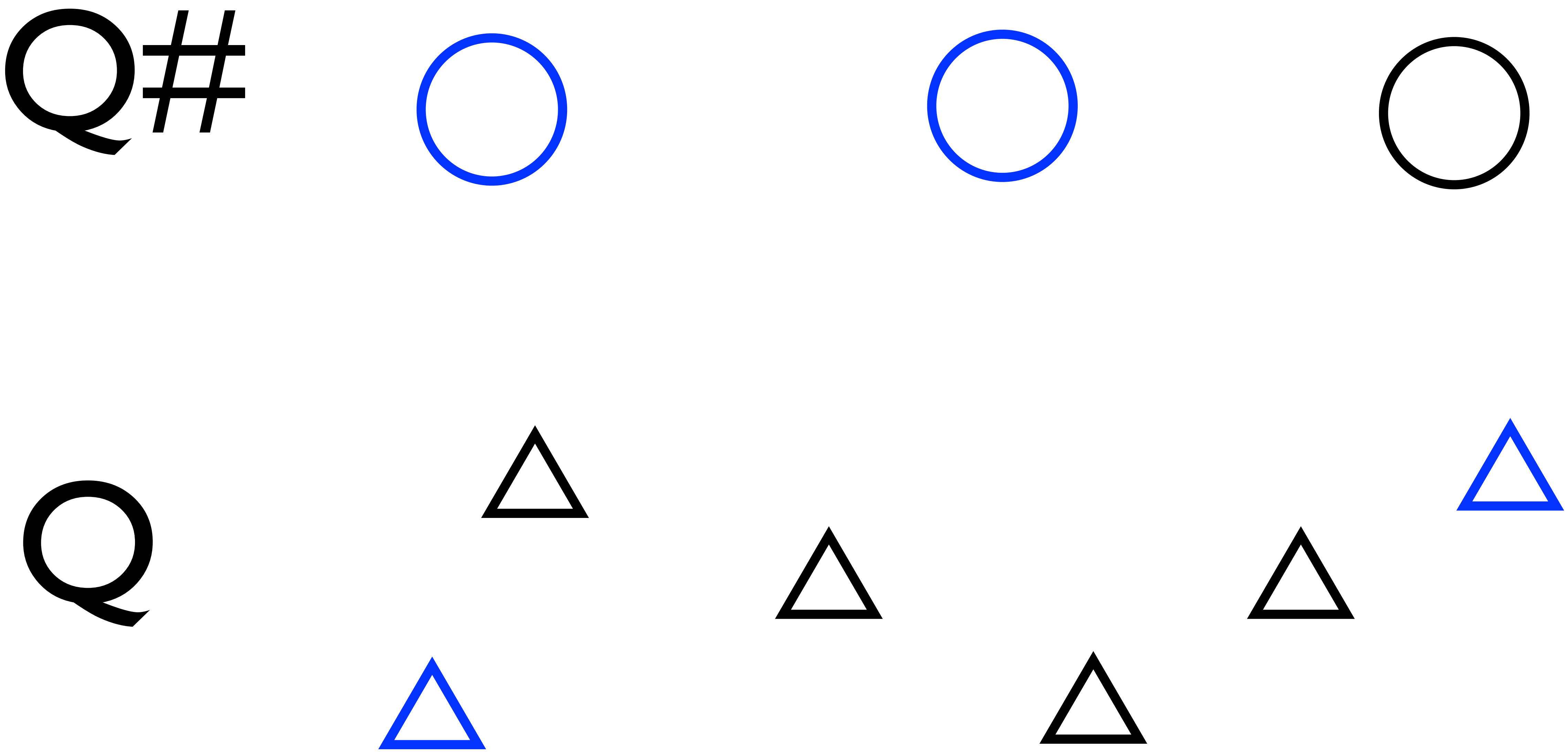
Building IP#

Basic idea: construct IP# as a forward exploration of an abstract state space

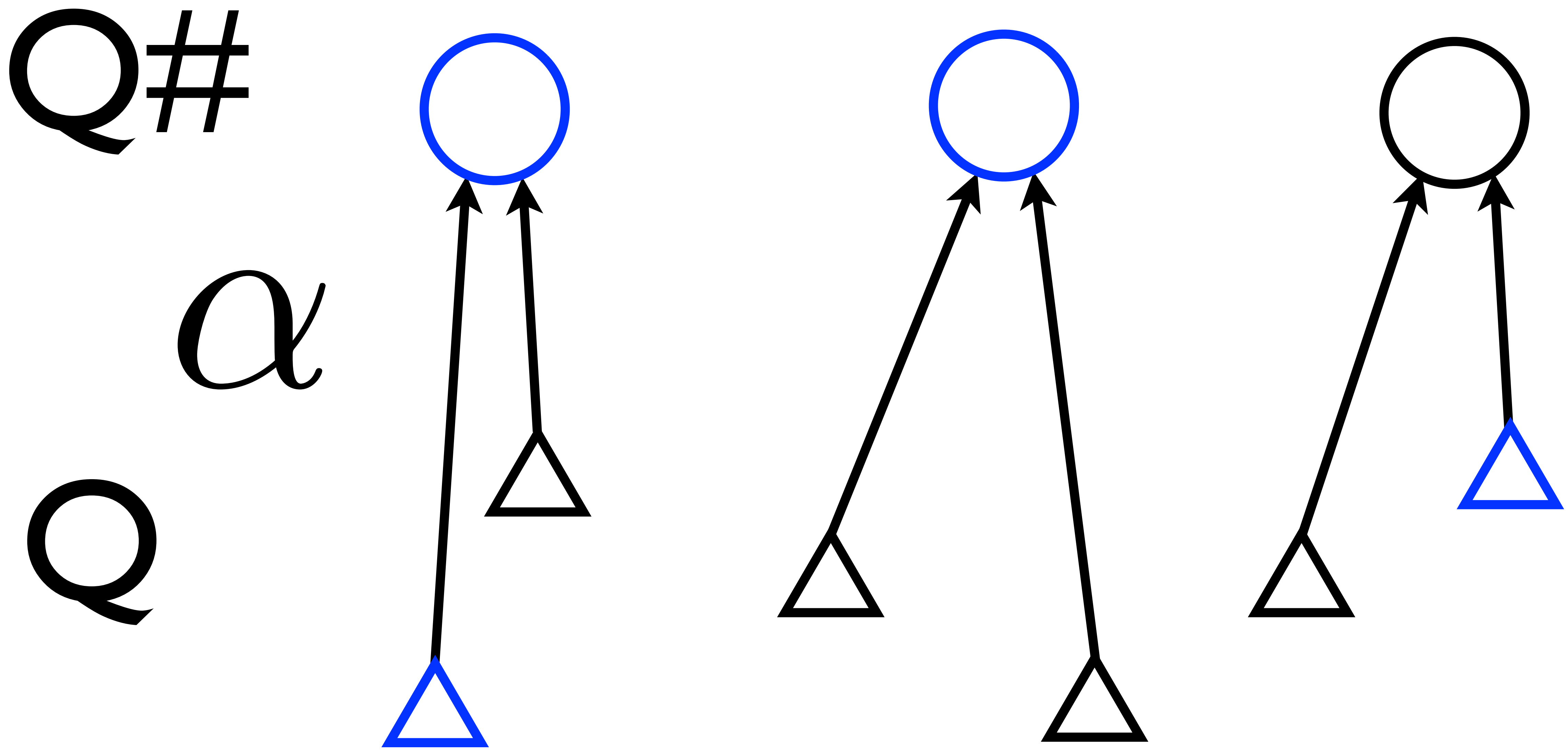
1(a). IP#: Define Abstract State-space



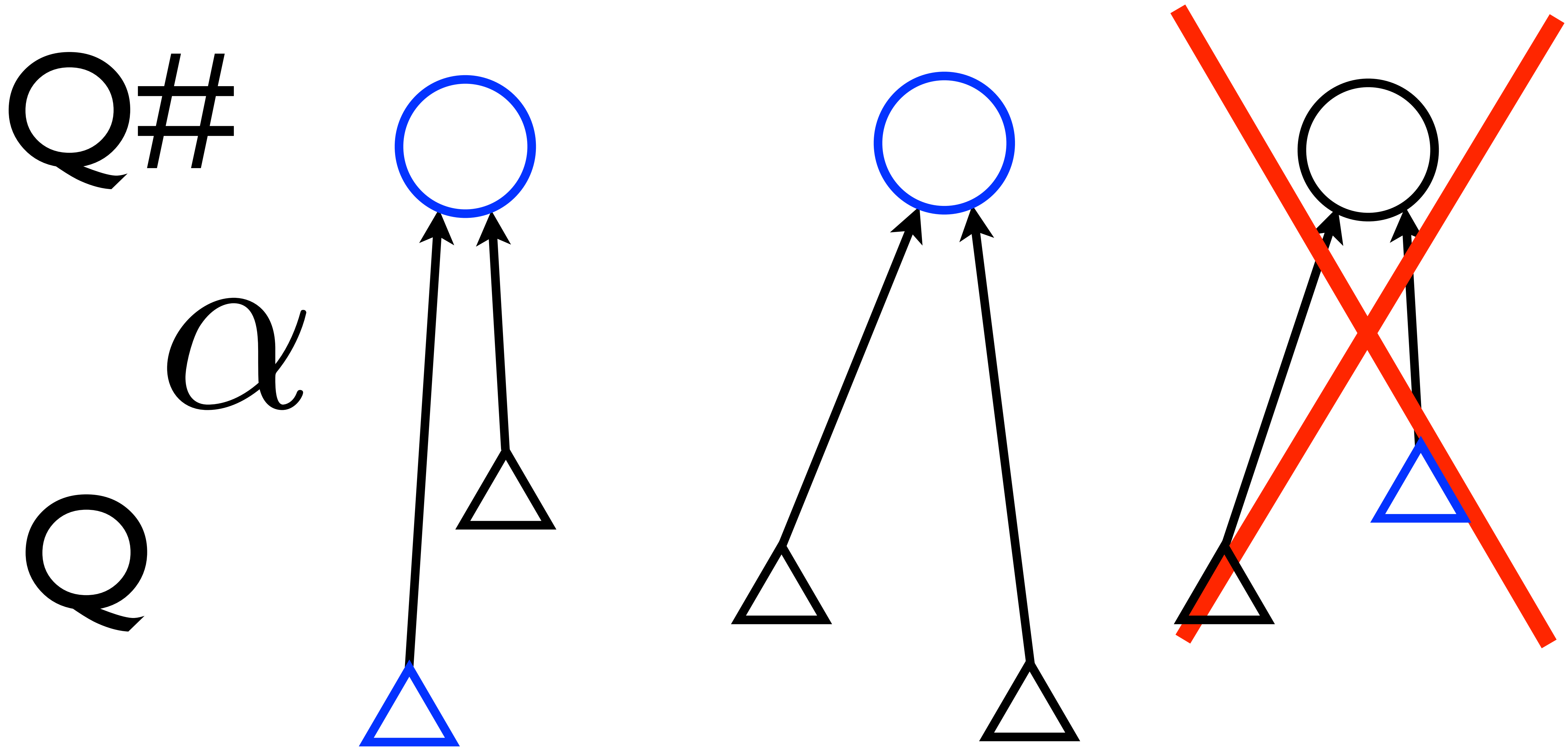
1(a). IP#: Define Abstract State-space



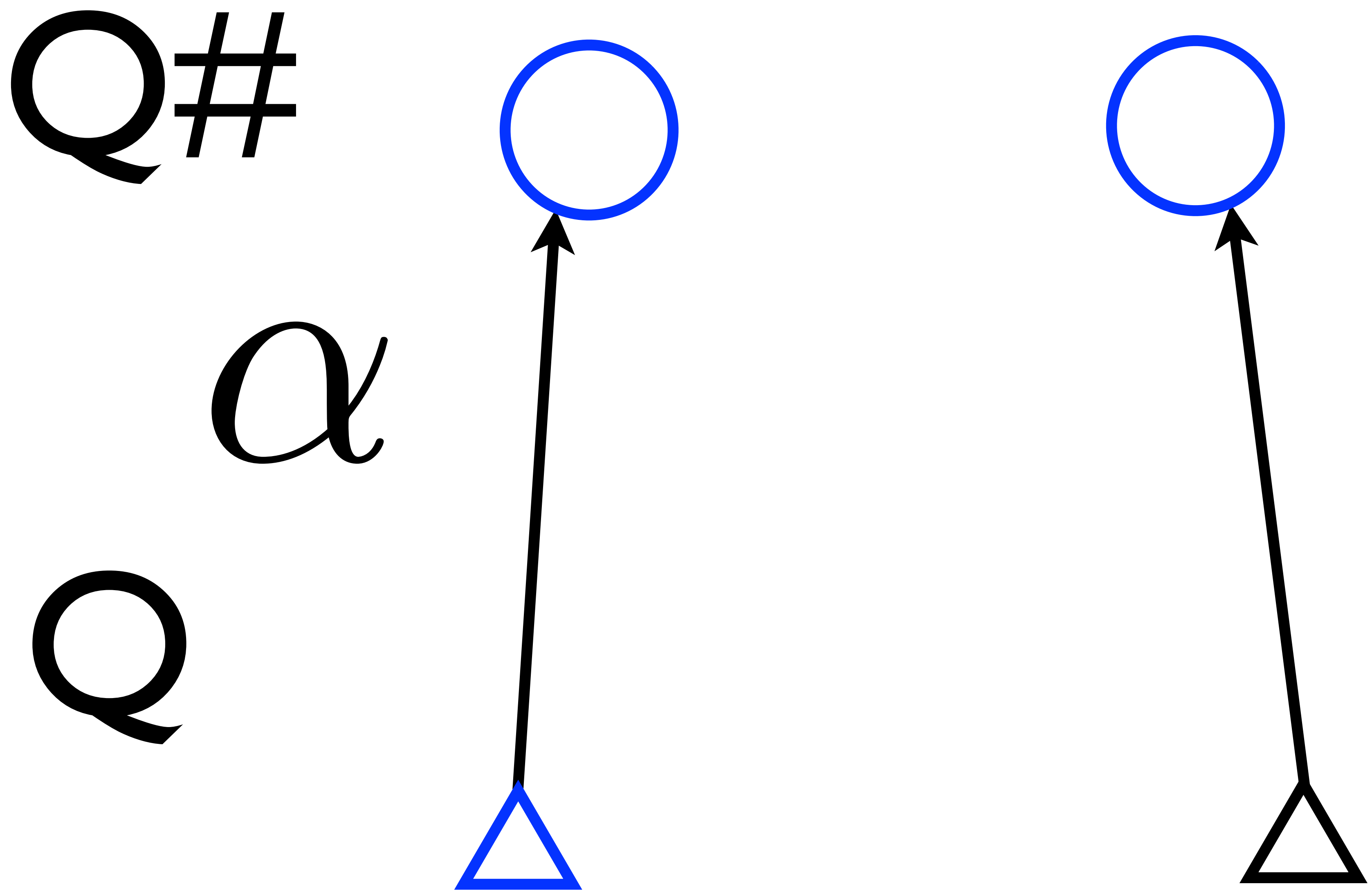
1(a). IP#: Define Abstract State-space



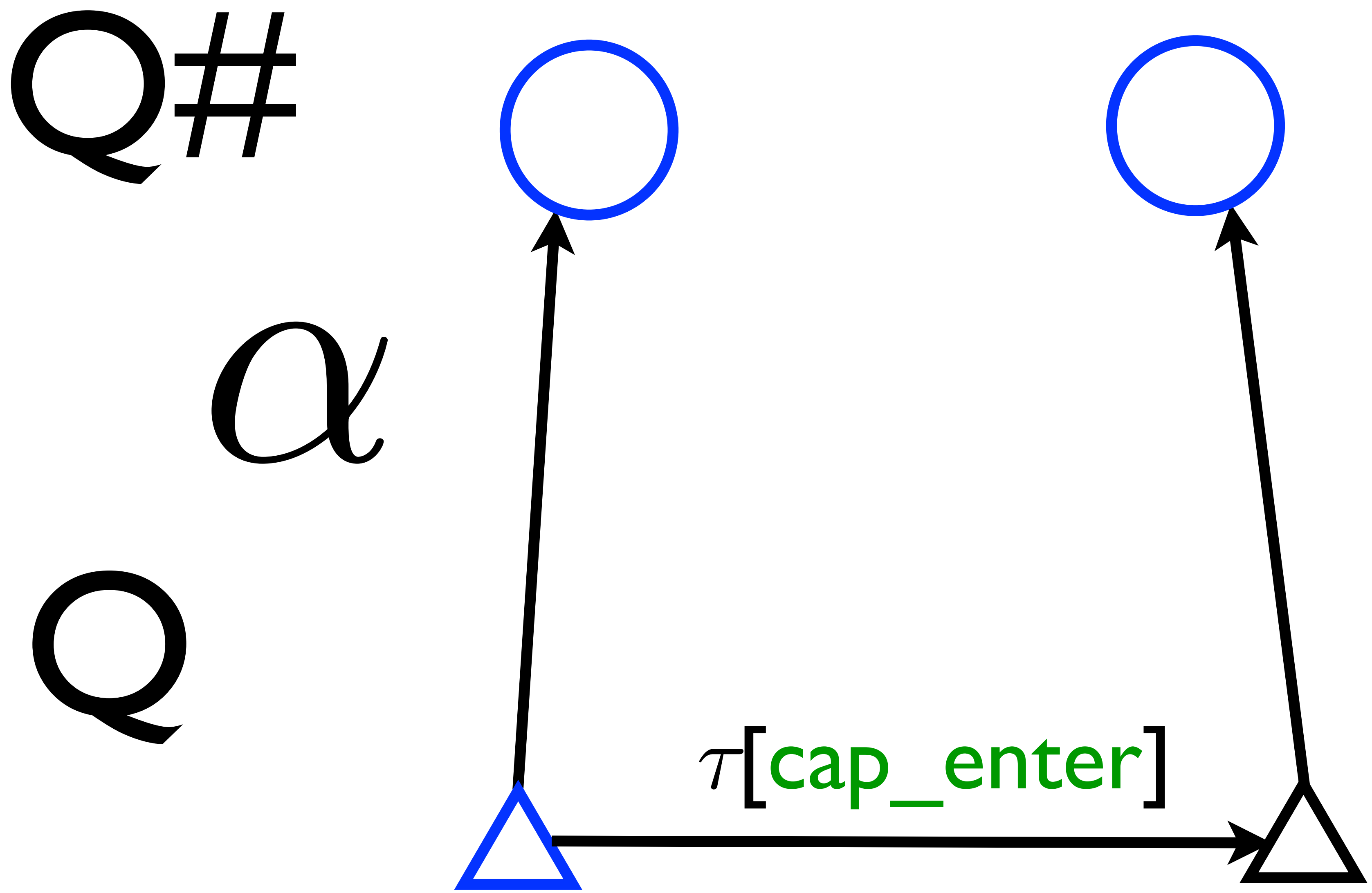
1(a). IP#: Define Abstract State-space



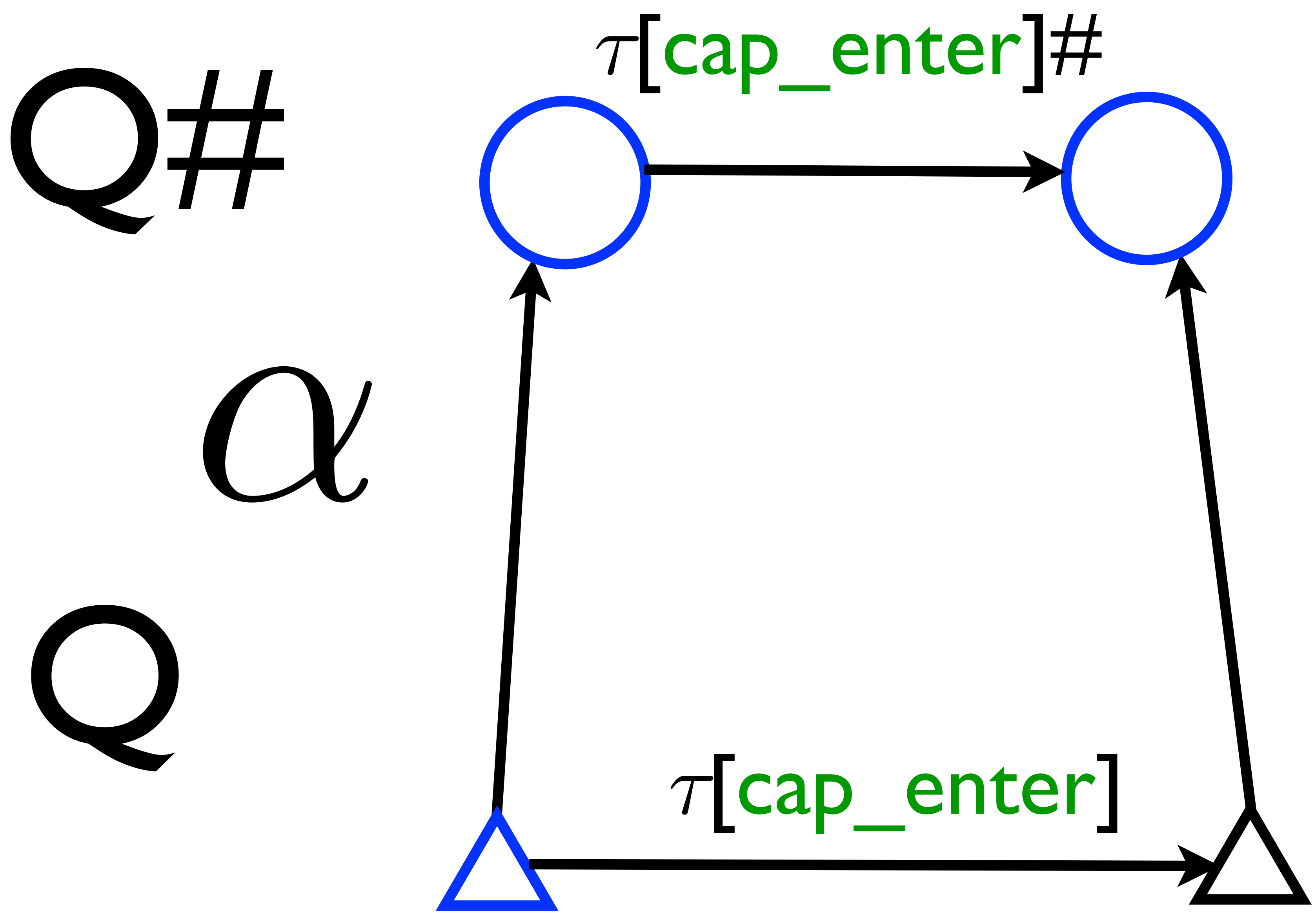
1(b). IP#: Define Abstract Transformers



1(b). IP#: Define Abstract Transformers



1(b). IP#: Define Abstract Transformers

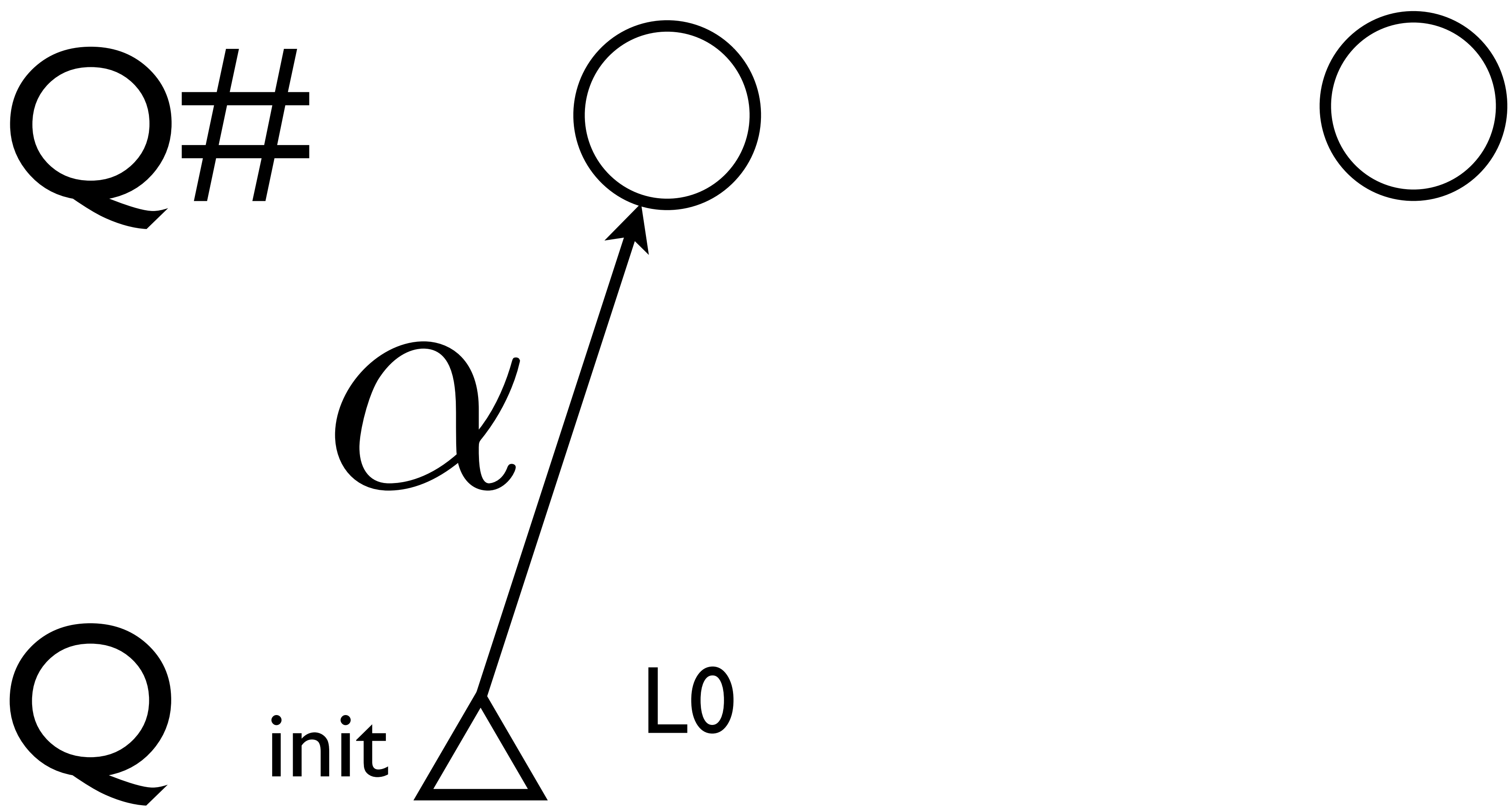


1(c). Explore Abstract State Space

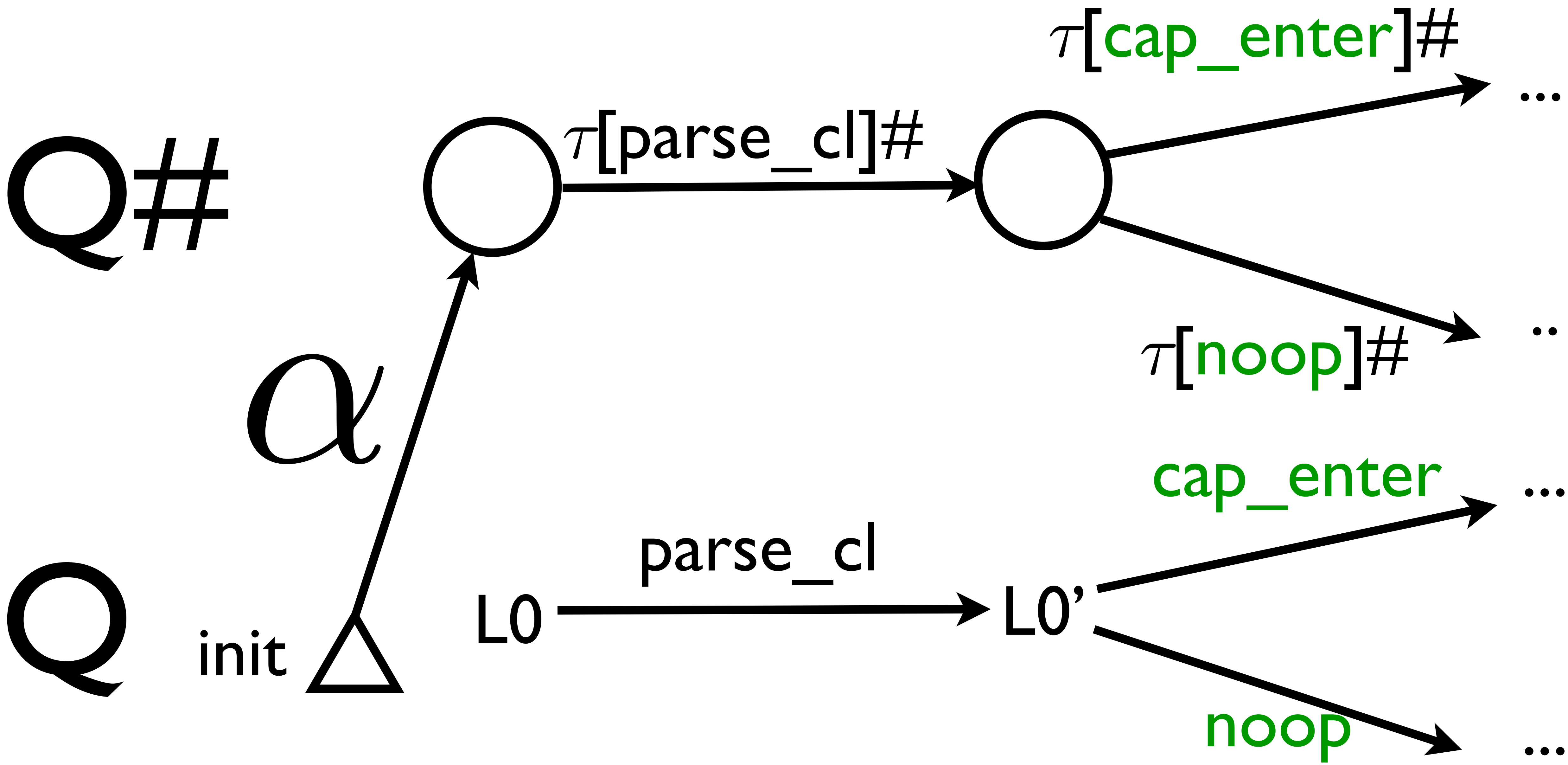
Q#   ○   ○

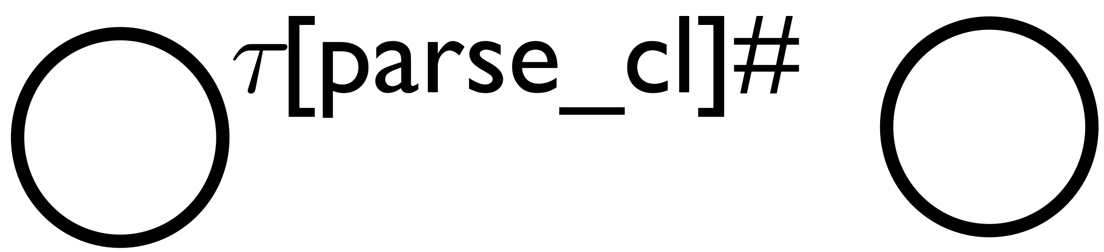
Q   △

1(c). Explore Abstract State Space

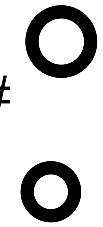


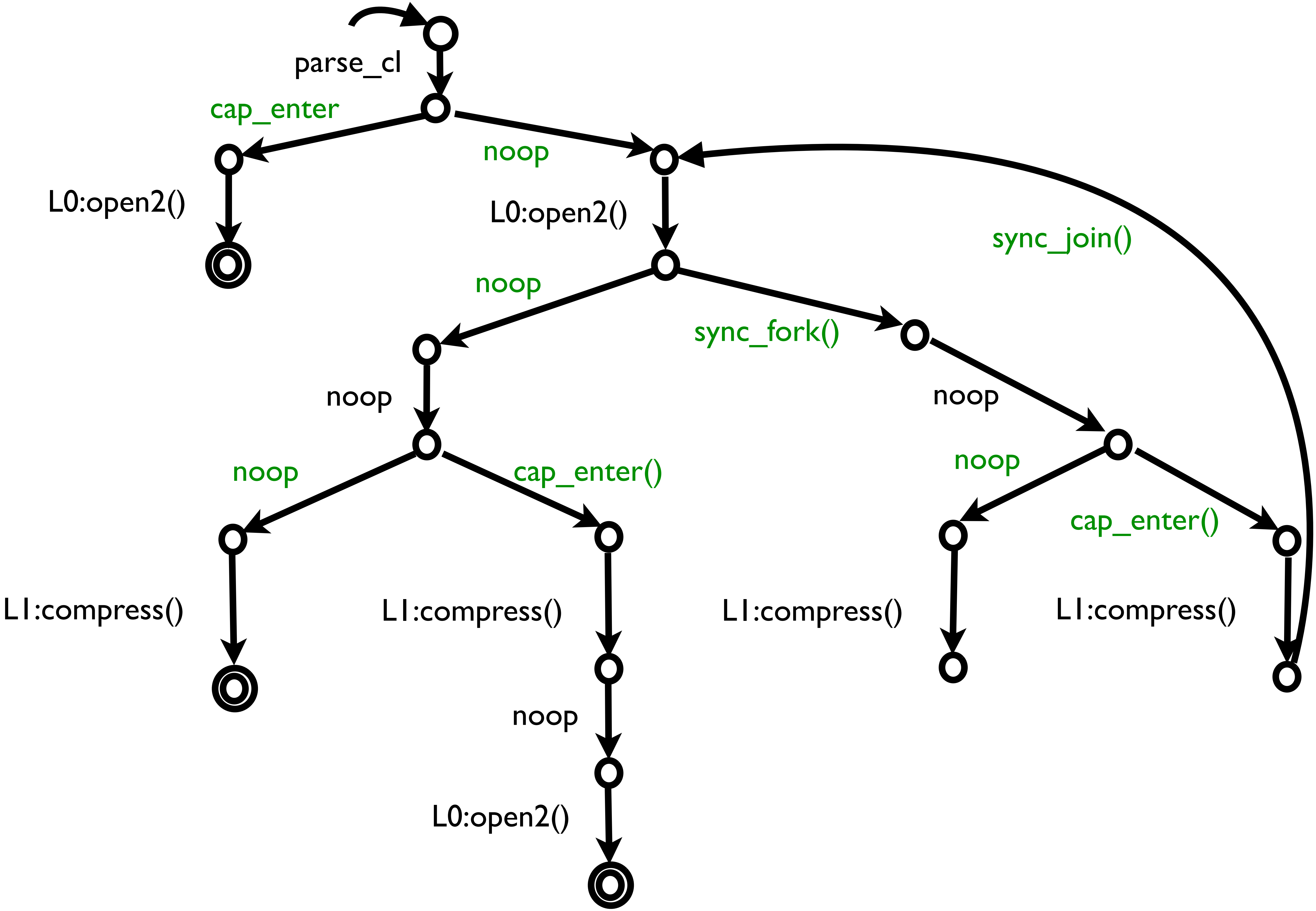
1(c). Explore Abstract State Space




τ[parse_cl]#

τ[parse_cl]#




parse_cl
cap_enter
noop
L0:open2()
L0:open2()
sync_join()
noop
sync_fork()
noop
noop
noop
cap_enter()
noop
cap_enter()
L1:compress()
L1:compress()
L1:compress()
L1:compress()
noop
L0:open2()

State-Structure Exploration

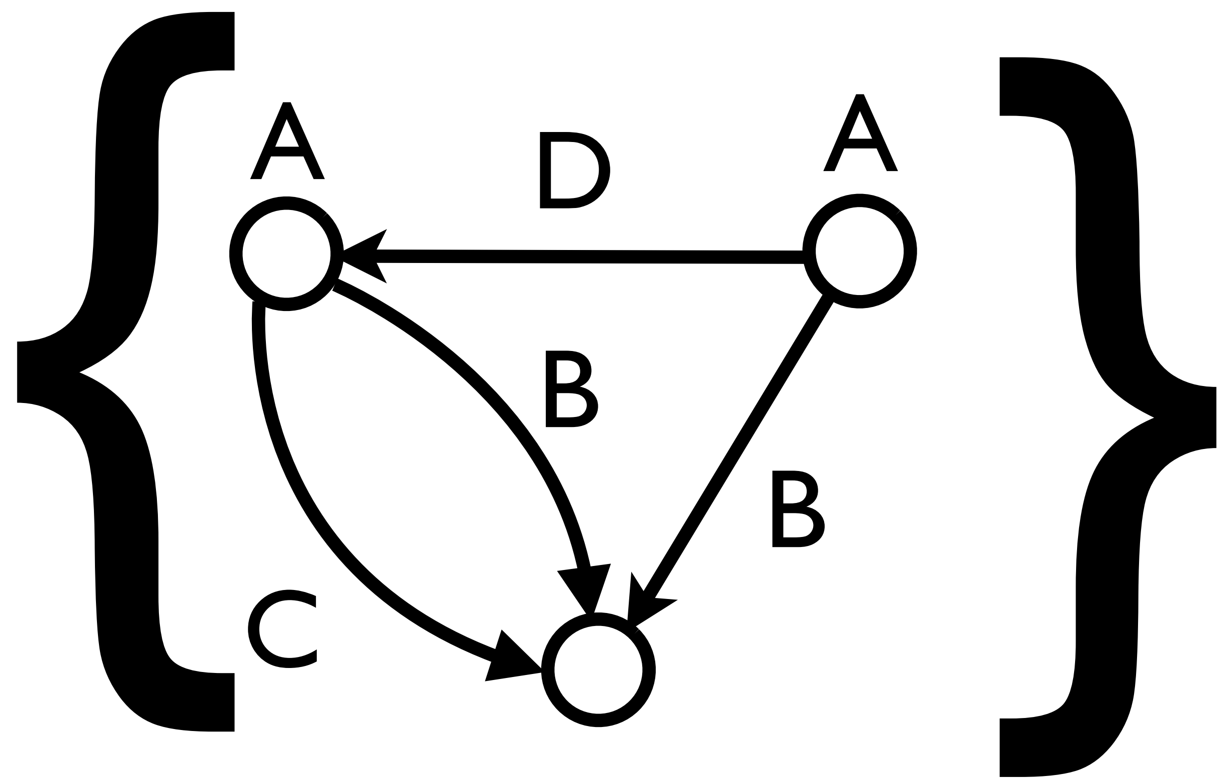
If a concrete state is a logical structure, ...

Q

State-Structure Exploration

If a concrete state is a logical structure, ...

$Q \equiv$



State-Structure Exploration

properties are FOL formulas, ...

$$\forall p. A(p) \Rightarrow ((B(p) \Rightarrow C(p)) \wedge (D(p) \Rightarrow \neg C(p)))$$

State-Structure Exploration

...and semantics is given as predicate updates, ...

$$\tau[\text{action}] \equiv \begin{array}{l} A'(x) = A(x) \vee \exists\, y.\, C(y) \wedge B(q, p) \\ B'(x, y) = B(x, y) \vee (C(x) \wedge D(y)) \\ C'(x) = \ldots \\ D'(x) = \ldots \end{array}$$

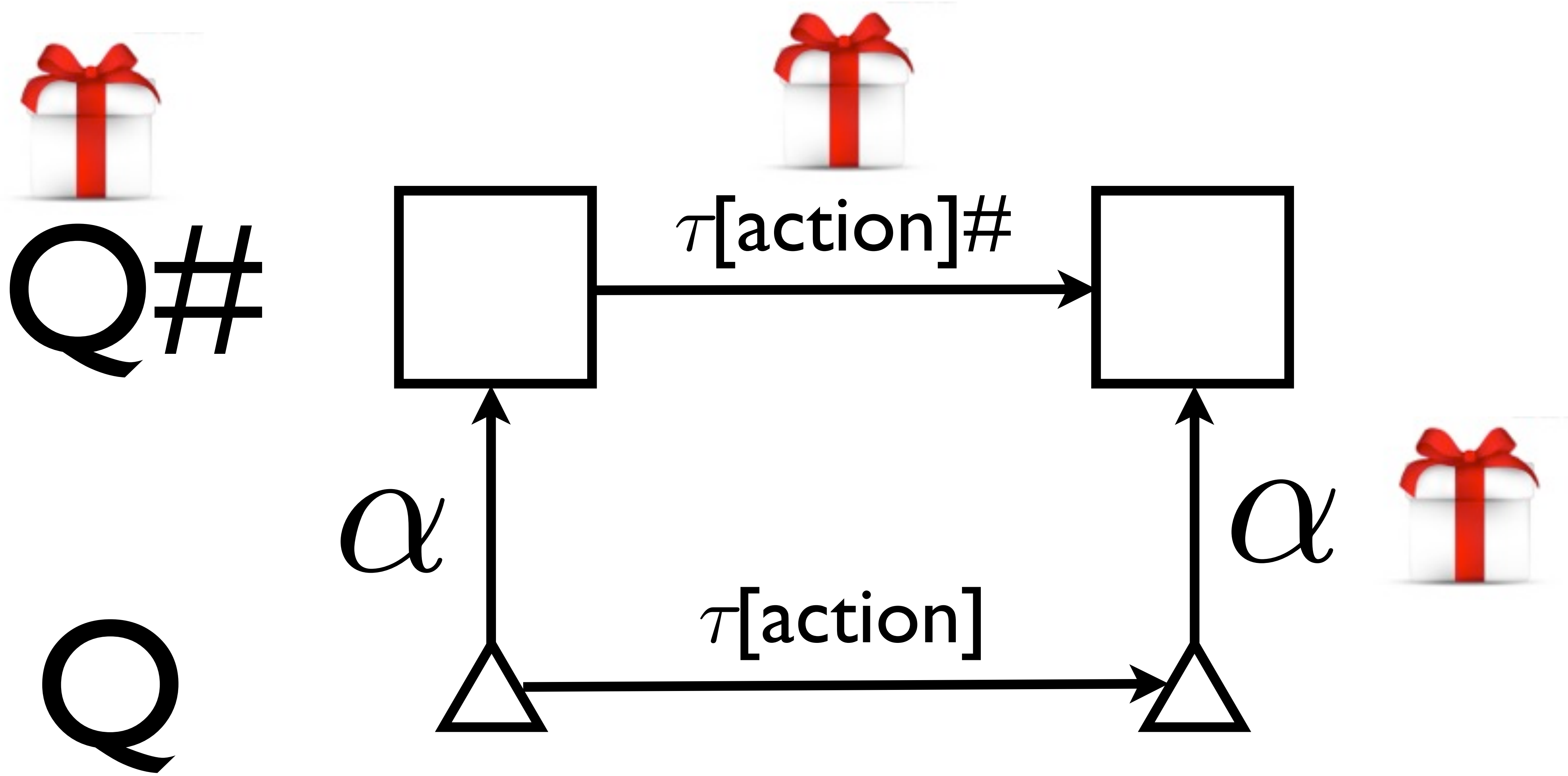
State-Structure Exploration

...then abstract space and transformers
can be generated automatically [Sagiv ’99]

Q △ △

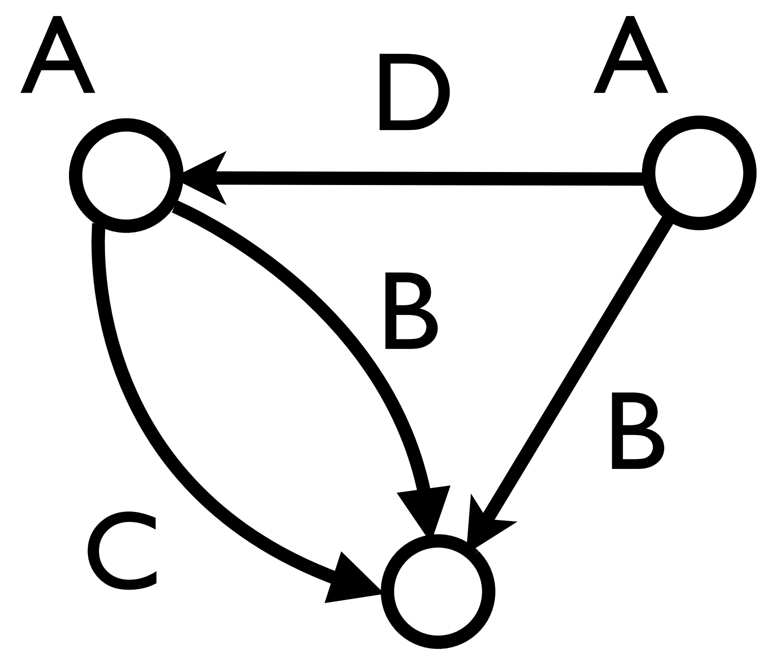
State-Structure Exploration

...then abstract space and transformers
can be generated automatically [Sagiv ’99]



Capsicum Semantics

1. $Q \equiv$



2. τ[action] $\equiv$

A'(x) = A(x) $\vee$ $\exists$ y. C(y) $\wedge$ B(q, p)

B'(x, y) = B(x, y) $\vee$ (C(x) $\wedge$ D(y))

C'(x) = ...

D'(x) = ...

Capsicum State as Structure

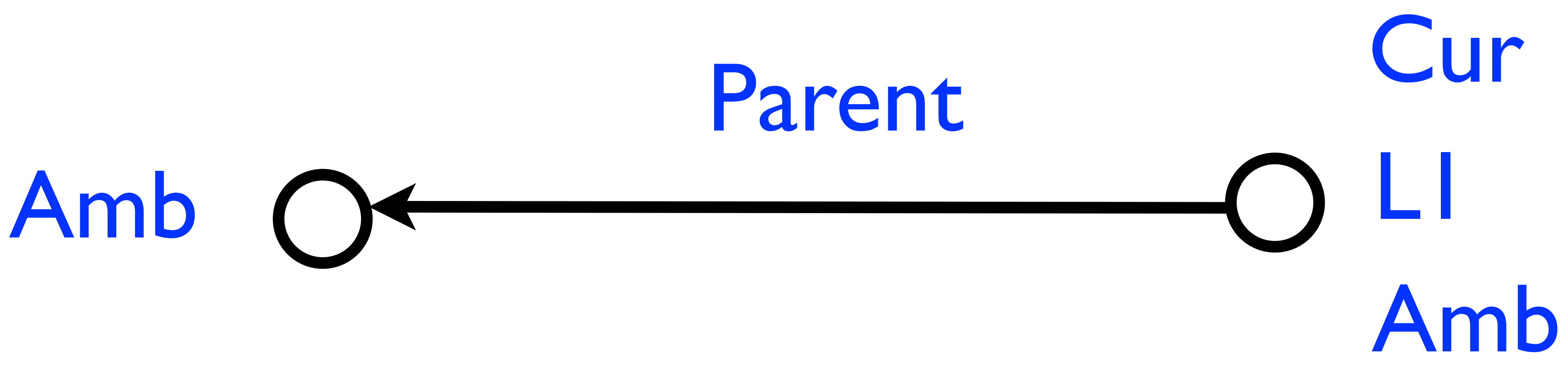
Capsicum State as Structure

Capsicum State as Structure

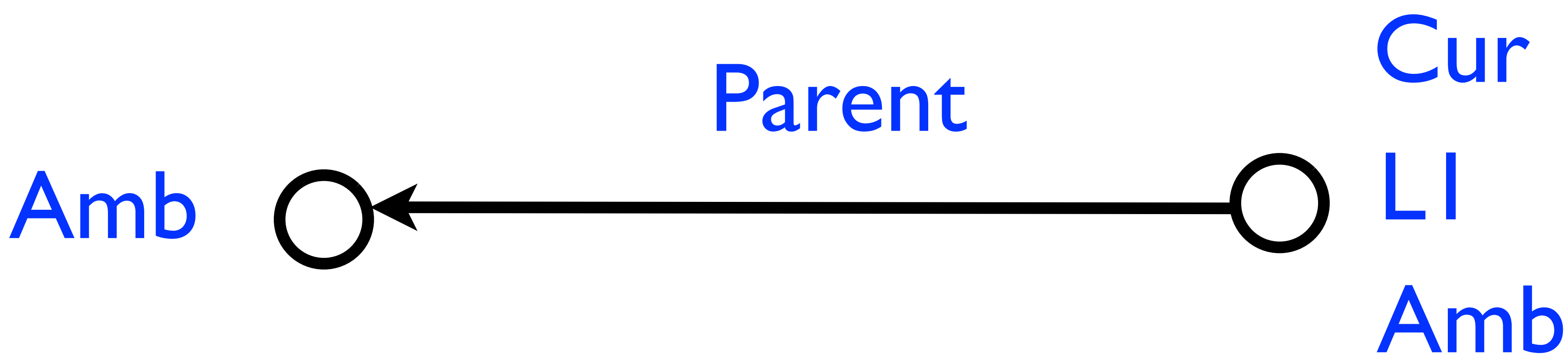
Capsicum State as Structure

Capsicum State as Structure

Capsicum State as Structure

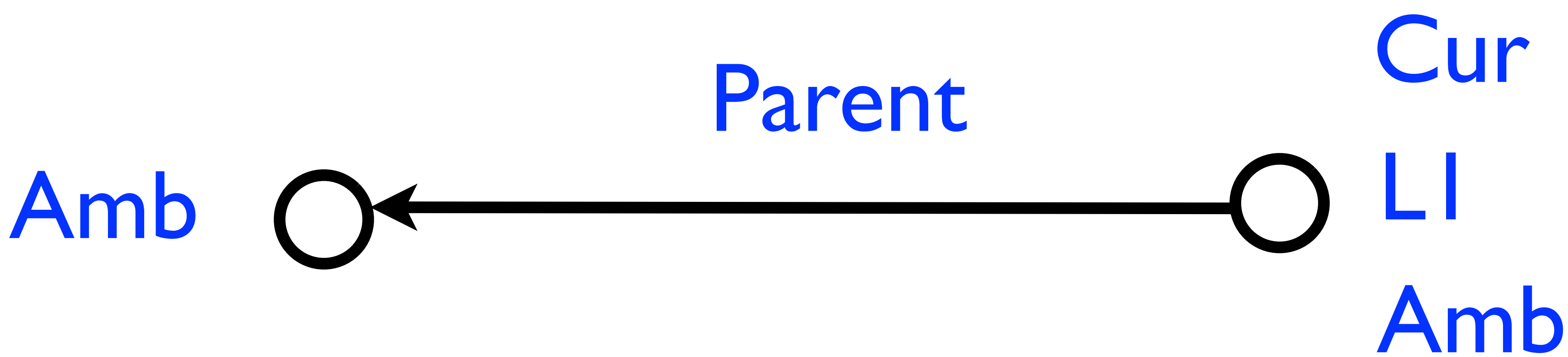


Capsicum State as Structure



$$\forall p.\ \mathrm{Cur}(p) \wedge \mathrm{L1}(p) \Rightarrow \neg\ \mathrm{Amb}(p)$$

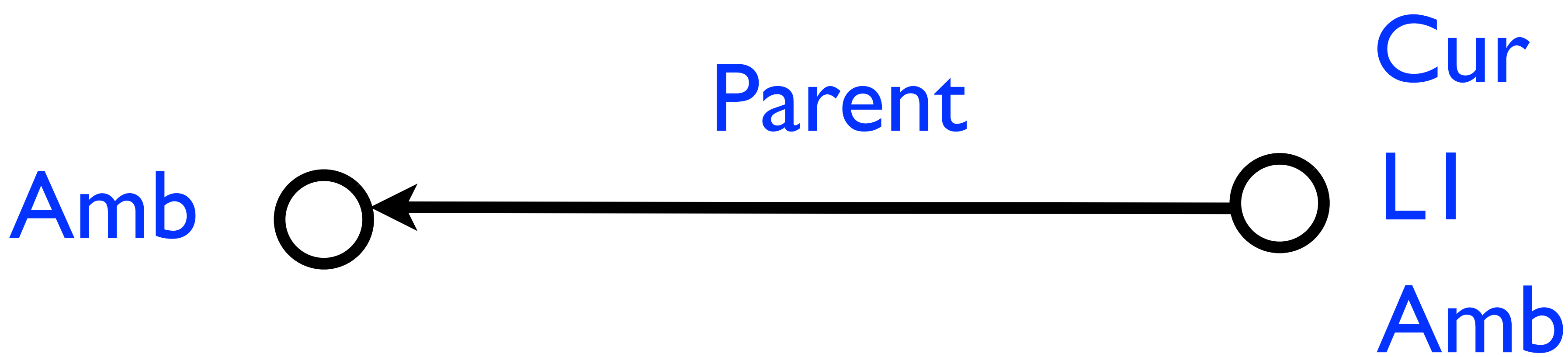
Capsicum State as Structure



$$\not\models$$

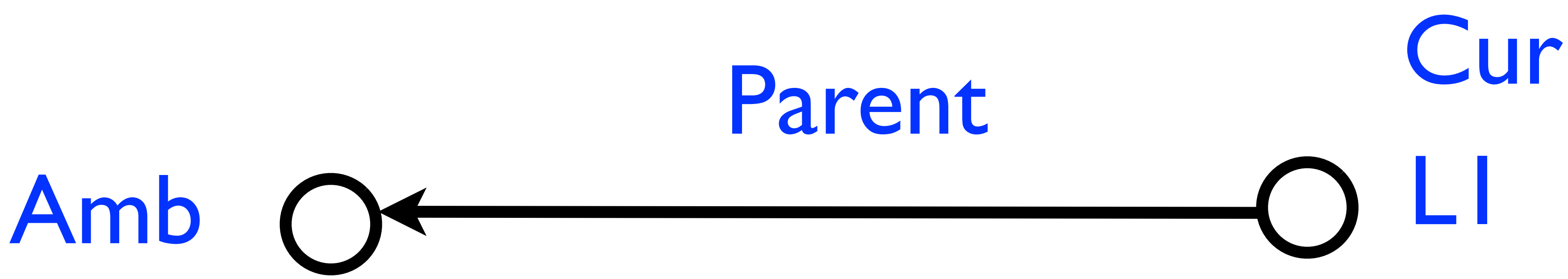
$$\forall\, p.\ \mathrm{Cur}(p) \wedge \mathrm{L1}(p) \Rightarrow \neg\ \mathrm{Amb}(p)$$

Capsicum State as Structure



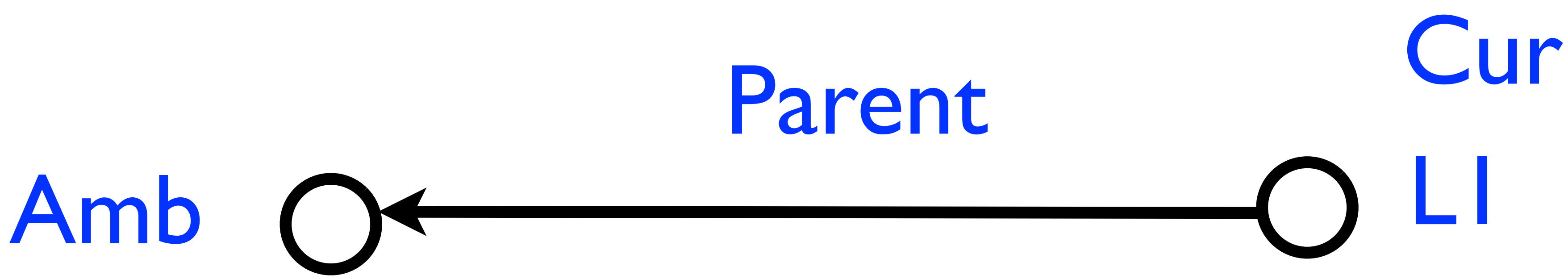
$$\forall\, p.\ \mathrm{Cur}(p) \wedge \mathrm{L1}(p) \Rightarrow \neg\, \mathrm{Amb}(p)$$

Capsicum State as Structure



$$\forall\, p.\ \mathrm{Cur}(p) \wedge \mathrm{L1}(p) \Rightarrow \neg\, \mathrm{Amb}(p)$$

Capsicum State as Structure



$$\models$$

$$\forall\, p.\ \mathrm{Cur}(p) \wedge \mathrm{L1}(p) \Rightarrow \neg\, \mathrm{Amb}(p)$$

Capsicum Structure Transformers

Action	
sync_fork()	

Capsicum Structure Transformers

Cur
Amb ○

Action	
sync_fork()	

Capsicum Structure Transformers

Cur Amb

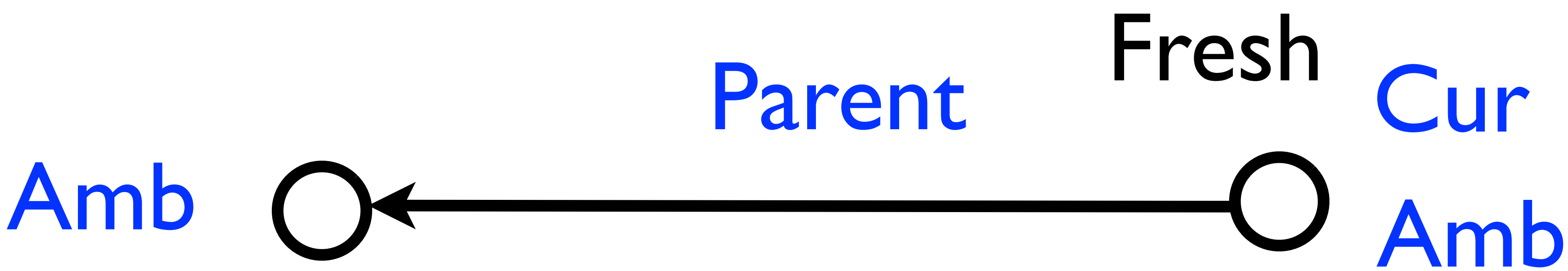
Fresh

Action	
sync_fork()	

Capsicum Structure Transformers

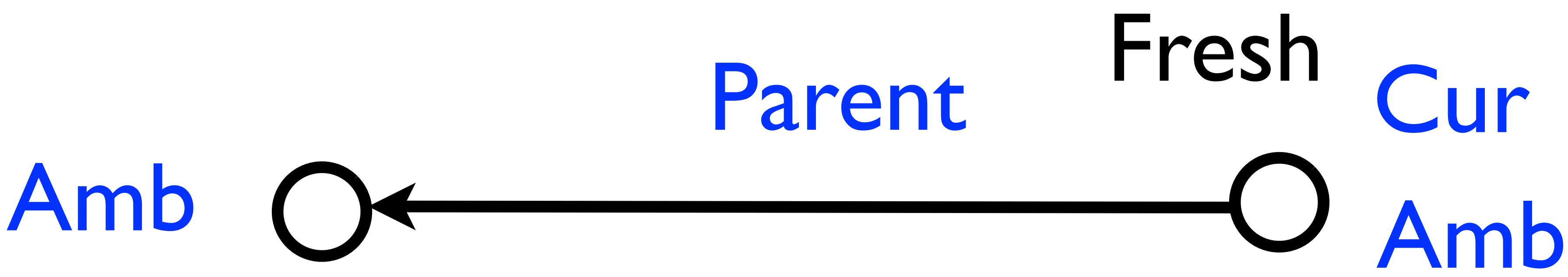
Action	
sync_fork()	

Capsicum Structure Transformers



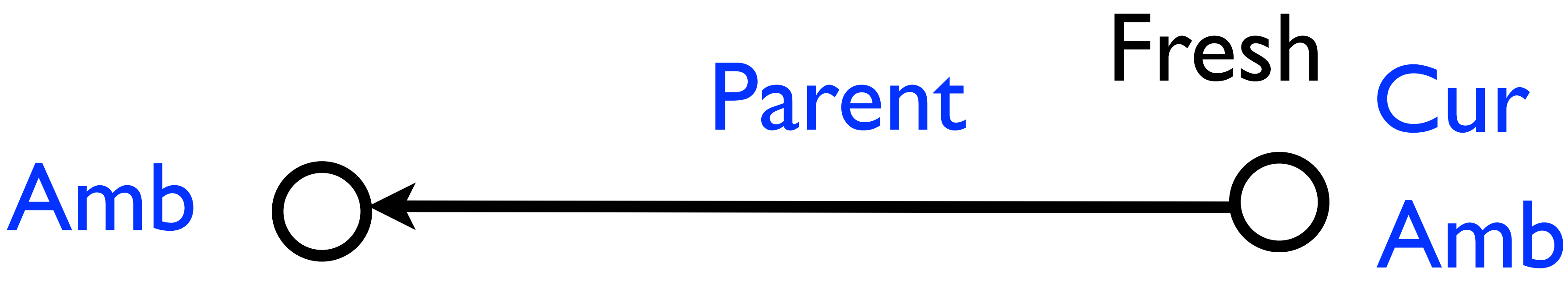
Action	
sync_fork()	

Capsicum Structure Transformers



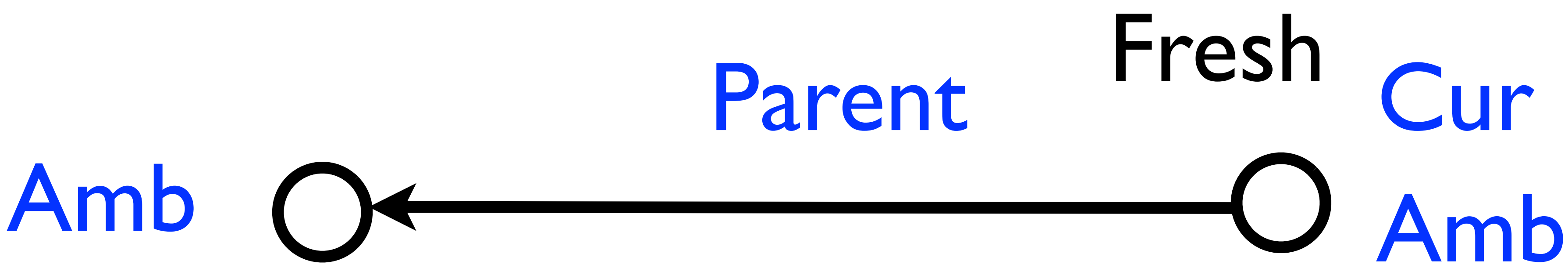
Action	Structure Transformer
sync_fork()	

Capsicum Structure Transformers



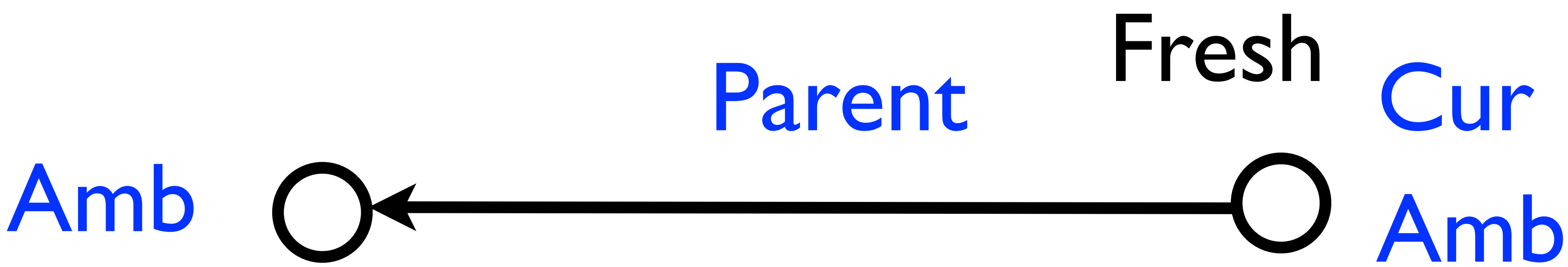
Action	Structure Transformer
sync_fork()	Intro Fresh Amb'(p) := Amb(p) $\vee$ (Fresh(p) $\wedge \exists$ q. Cur(q) $\wedge$ Amb(q))

Capsicum Structure Transformers



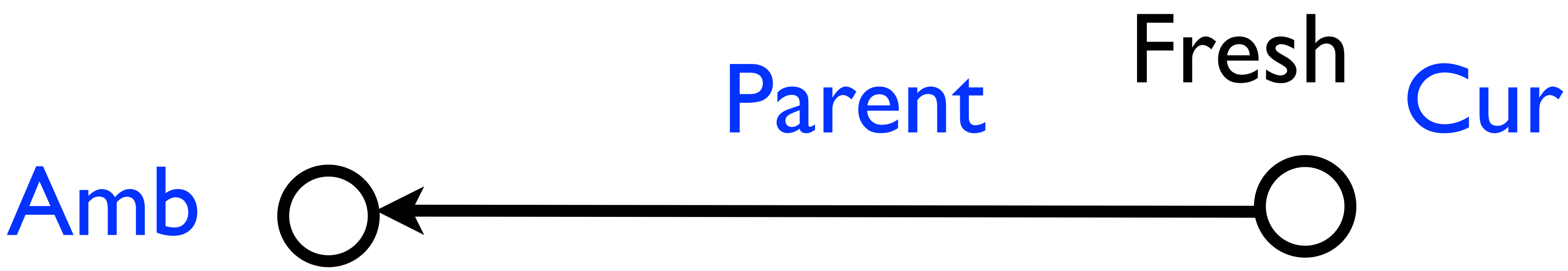
Action	Structure Transformer

Capsicum Structure Transformers



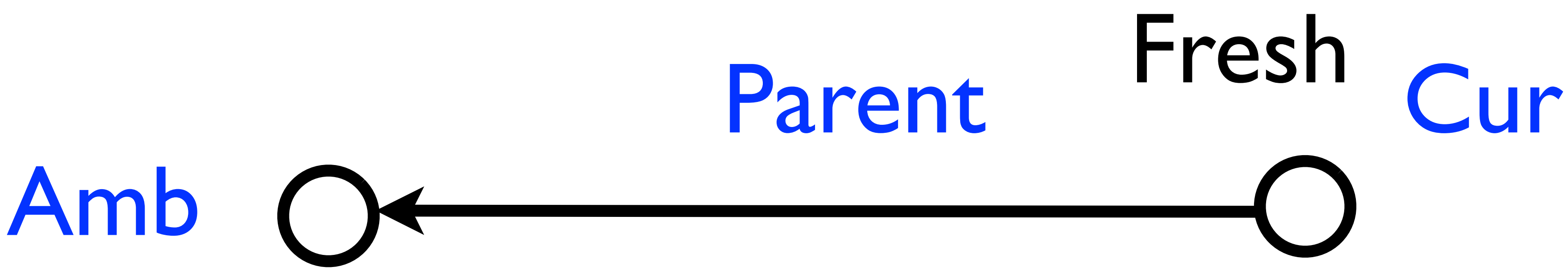
Action	Structure Transformer
cap_enter()	

Capsicum Structure Transformers



Action	Structure Transformer
cap_enter()	

Capsicum Structure Transformers



Action	**Structure Transformer**
cap_enter()	Amb'(p) := Amb(p) $\wedge$ $\neg$ Cur(p)

Building IP#: Summary

- If semantics is given as transforms of logical structures, we can generate an approximation of runs that cause a violation
- Capsicum semantics can be modeled as structure transforms

CapWeave Algorithm

Inputs: Program P, Amb Policy Q
Output: Instrumentation of P that always satisfies Q

1. Build finite **IP#** $\supseteq$ instrumented runs that violate Q

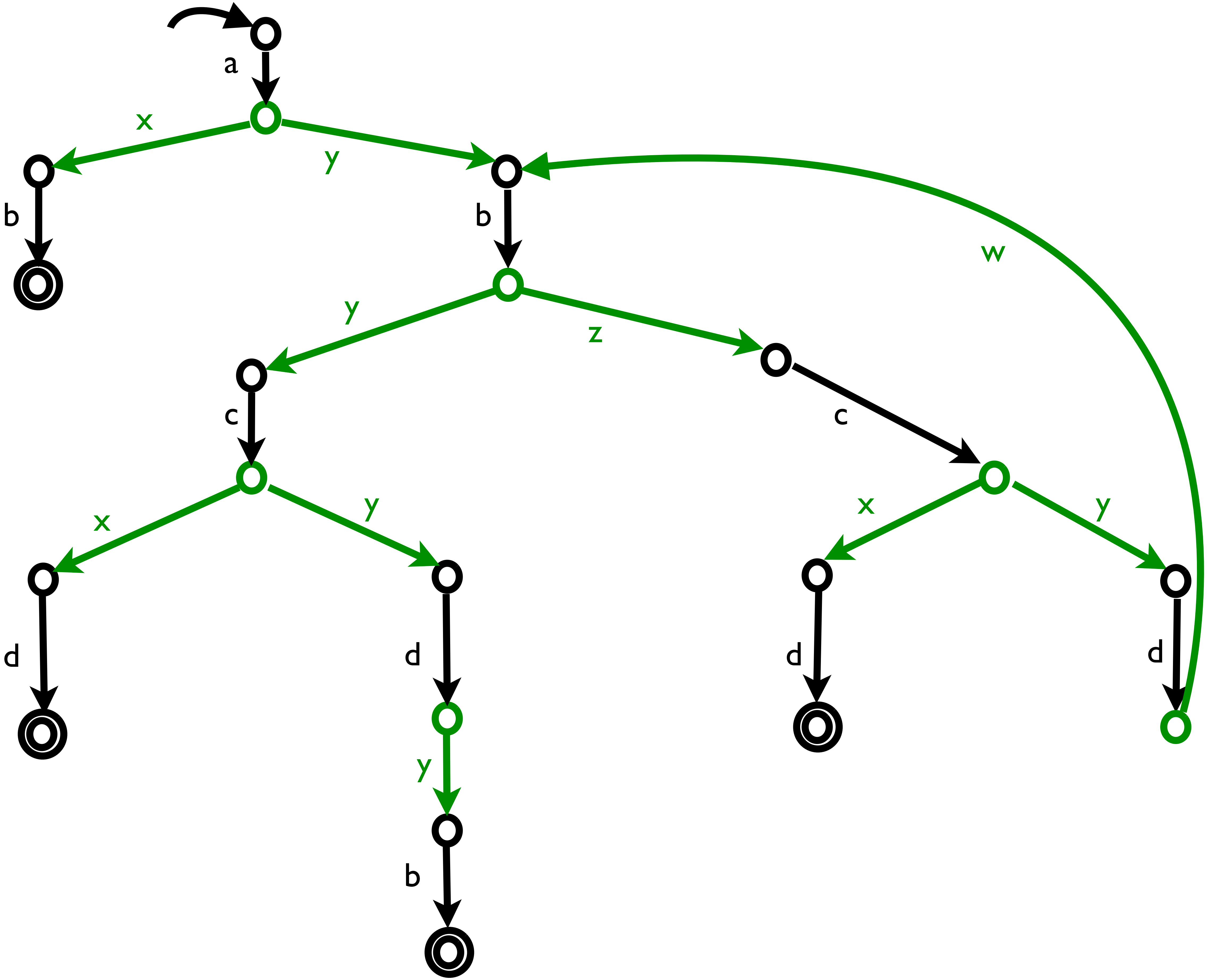
CapWeave Algorithm

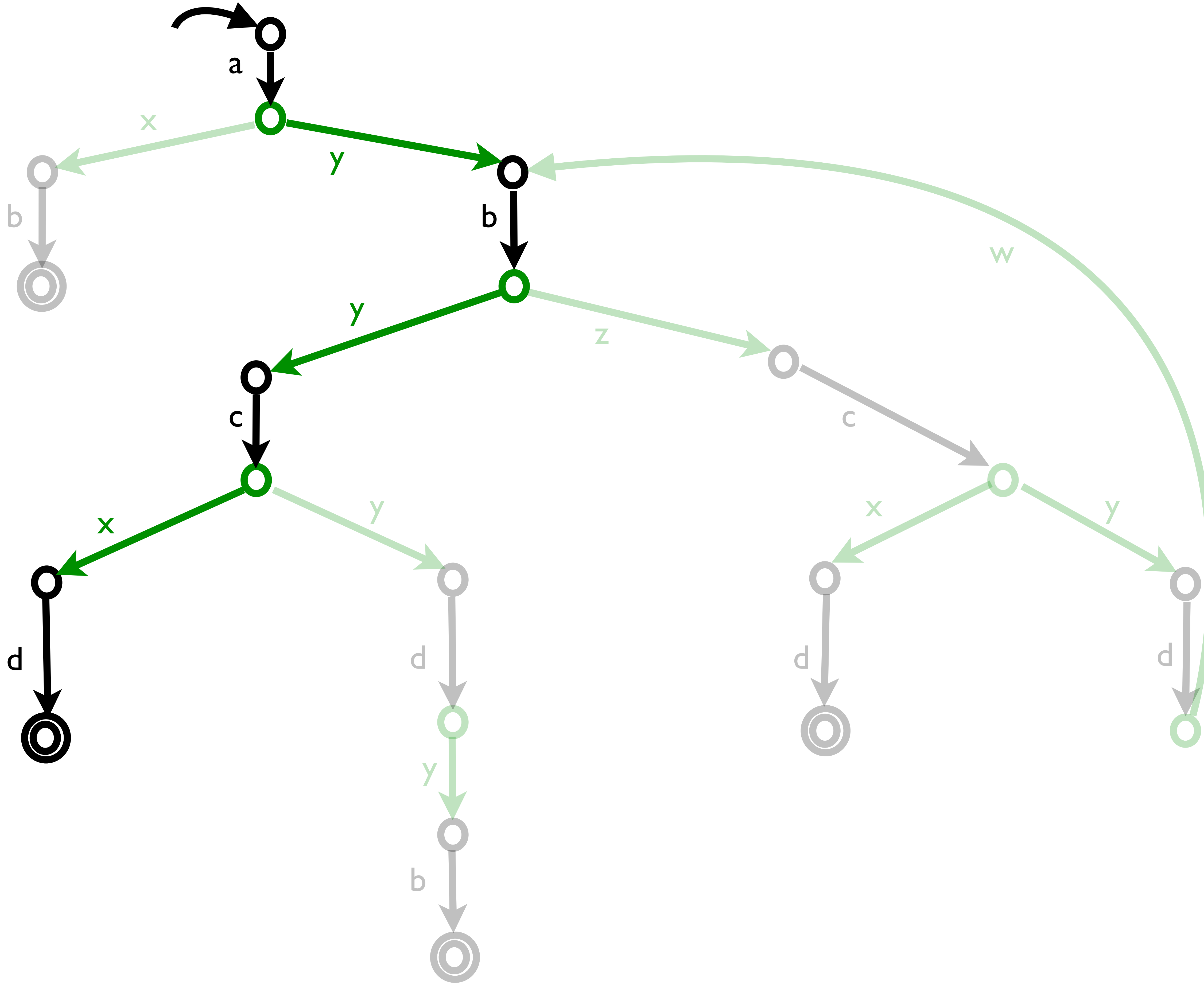
Inputs: Program P, Amb Policy Q
Output: Instrumentation of P that always satisfies Q

1. Build finite **IP#** $\supseteq$ instrumented runs that violate Q
2. From IP#, build **safety game G** won by violations of Q

Two-Player Safety Games

- In an Attacker state,
  the Attacker chooses the next input
- In a Defender state,
  the Defender chooses the next input
- Attacker wants to reach an accepting state


a
x
y
b
b
w
y
z
c
c
x
y
x
y
d
d
d
d
y
b

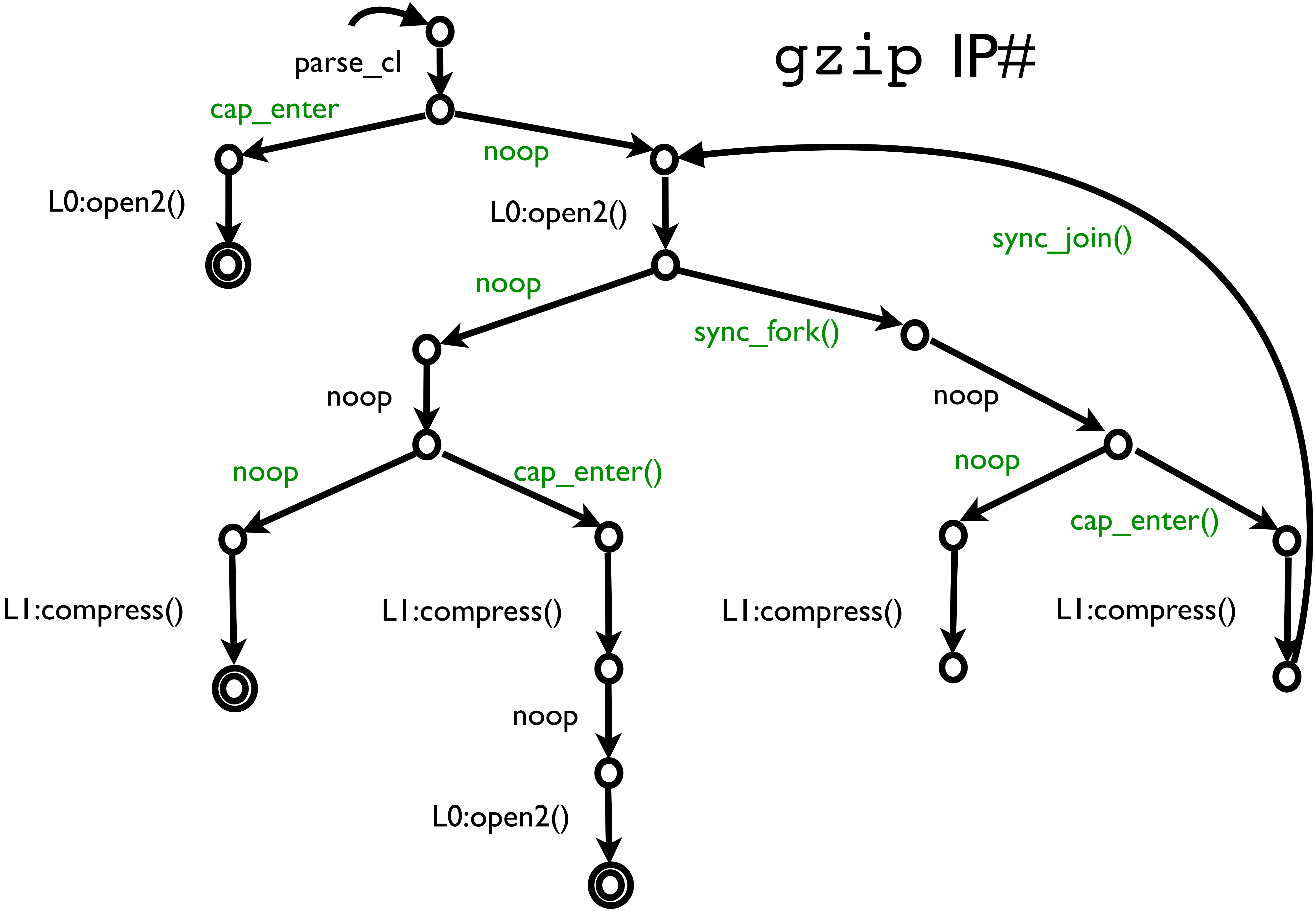


a
x
y
b
b
w
y
z
c
c
x
y
x
y
d
d
d
d
y
b

Instrumentation as a Game

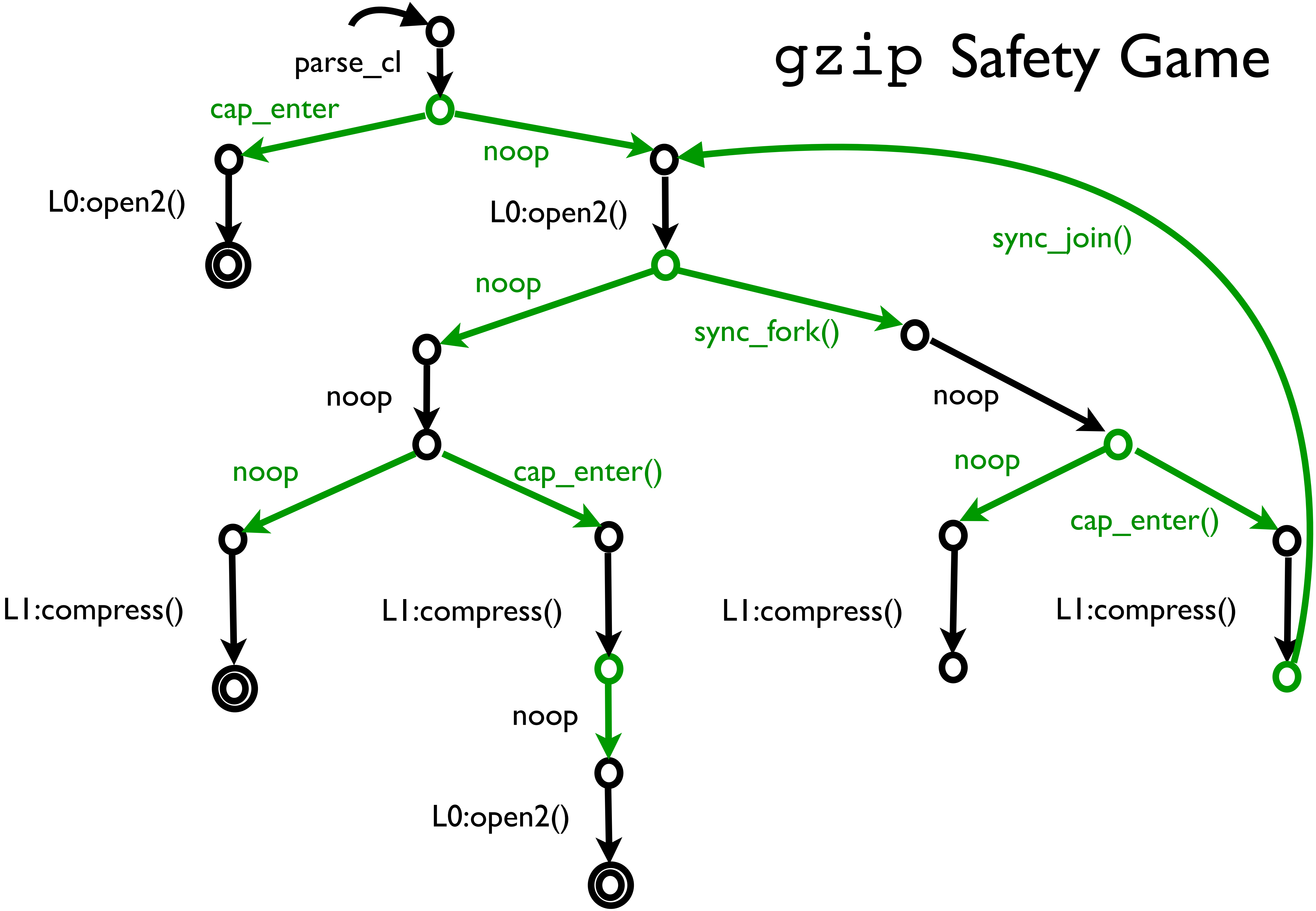
Capsicum Instrumentation	Two-player Games
Program instructions	Attacker actions
Capsicum primitives	Defender actions
Policy violations	Attacker wins
Satisfying instrumentation	Winning Defender strategy

gzip IP#

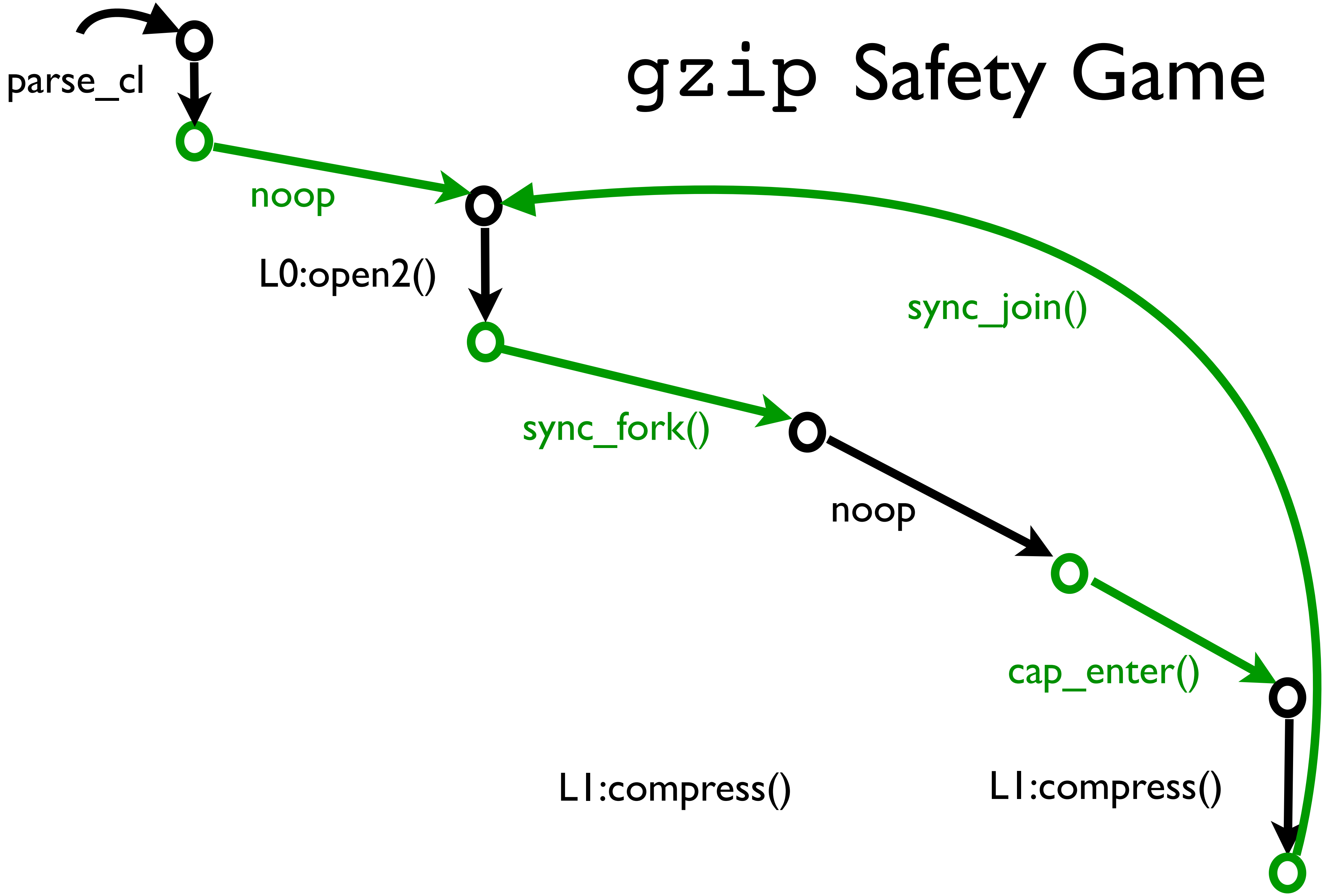


parse_cl
cap_enter
noop
L0:open2()
L0:open2()
sync_join()
noop
sync_fork()
noop
noop
noop
cap_enter()
noop
cap_enter()
L1:compress()
L1:compress()
L1:compress()
L1:compress()
noop
L0:open2()

gzip Safety Game



`gzip` Safety Game



CapWeave Algorithm

Inputs: Program P, Amb Policy Q
Output: Instrumentation of P that always satisfies Q

1. Build finite **IP#** $\supseteq$ instrumented runs that violate Q
2. From IP#, build **safety game G** won by violations of Q

CapWeave Algorithm

Inputs: Program P, Amb Policy Q
Output: Instrumentation of P that always satisfies Q

1. Build finite **IP#** $\supseteq$ instrumented runs that violate Q
2. From IP#, build **safety game G** won by violations of Q
3. From winning strategy for G, generate primitive controller for P

CapWeave Performance

Name	Program kLoC	Policy LoC	Weaving Time
bzip2-1.0.6	8	70	4m57s
gzip-1.2.4	9	68	3m26s
php-cgi-5.3.2	852	114	46m36s
tar-1.25	108	49	0m08s
tcpdump-4.1.1	87	52	0m09s
wget-1.12	64	35	0m10s

Performance on Included Tests

Name	Base Time	Hand Overhd	capweave Overhd	Diff. Overhd (%)
bzip2-1.0.6	0.593s	0.909	1.099	20.90
gzip-1.2.4	0.036s	1.111	1.278	15.03
php-cgi-5.3.2	0.289s	1.170	1.938	65.64
tar-1.25	0.156s	13.301	21.917	64.78
tcpdump-4.1.1	1.328s	0.981	1.224	24.77
wget-1.12	4.539s	1.906	1.106	0.91

Outline

1. Motivation, problem statement
2. Previous work: Capsicum
3. Ongoing work: HiStar
4. Open challenges

Outline

3. Ongoing work: HiStar

The HiStar Priv-aware OS [Zeldovich '06]

- Privilege: OS allows flow between processes
- Primitives: system calls update labels, which define allowed flows
- Very powerful: mutually untrusting login (?!)

Sandboxing a Virus Scanner

```
launcher() {
  exec(“/bin/scanner”);
}
wrapper() {
  child = sync_fork(&launcher);
  while (true) {
    read(child, buf);
    sanitize(buf);
    write(netd, buf); }
}
```

A Flow Policy for a Virus Scanner

- Information should never transitively flow from the scanner to the network, unless it goes through the wrapper
- Information should always flow from the scanner to the wrapper
- Information should always flow from the wrapper to the network

Rules for HiStar's Flow

- A process's **label** maps each category to **low** or **high**
- If process p calls create_cat, then each process is low in c, and p can declassify c
- Each process may raise its level at each category
- Each process may relinquish declassification

Rules for HiStar's Flow

Information can flow from p to q if for each category:

- The level of p is lower than the level of q at c, or
- p can declassify c

Sandboxing a Virus Scanner

```
launcher() {

 exec("/bin/scanner"); }
wrapper() {



    child = sync_fork(&launcher);
    while (true) {
      read(child, buf);
      sanitize(buf);
      write(netd, buf); } }
```

Sandboxing a Virus Scanner

```
launcher() {

 exec("/bin/scanner"); }
wrapper() {
  create_cat(&x);
  raise(x);
  child = sync_fork(&launcher);
  while (true) {
    read(child, buf);
    sanitize(buf);
    write(netd, buf); } }
```

Sandboxing a Virus Scanner

```
launcher() {
 drop_declass(x);
 exec(“/bin/scanner”); }
wrapper() {
  create_cat(&x);
  raise(x);
  child = sync_fork(&launcher);
  while (true) {
    read(child, buf);
    sanitize(buf);
    write(netd, buf); } }
```

HiStar Challenges Not Appearing in This Talk

- There are actually **four** levels
- Each process has to manage its **clearance**
- Processes can create **labeled closures** (calling a closure implicitly performs two label operations and three ordering checks)

CapWeave Algorithm

Inputs: Program P, Amb Policy Q

Output: Instrumentation of P that satisfies Q

1. Build **IP#** ⊇ instrumented runs that violate Q (using Capsicum semantics)
2. From IP#, build **safety game G** won by violations of Q
3. From winning strategy for G, generate primitive controller for P

CapWeave Algorithm

Inputs: Program P, Amb Policy Q
Output: Instrumentation of P that satisfies Q

1. Build **IP#** ⊇ instrumented runs that violate Q (using Capsicum semantics)
2. From IP#, build **safety game G** won by violations of Q
3. From winning strategy for G, generate primitive controller for P

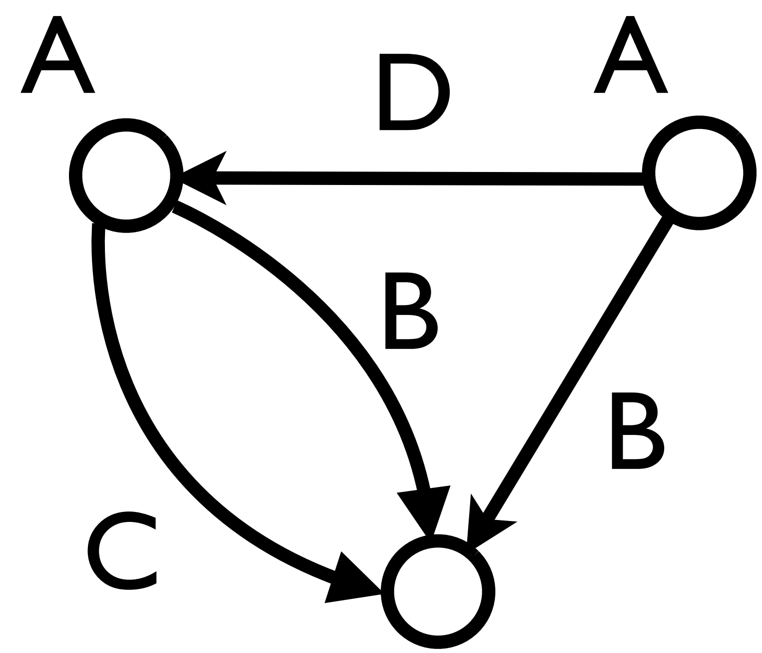
HiWeave Algorithm

Inputs: Program P, Flow Policy Q
Output: Instrumentation of P that satisfies Q

1. Build **IP#** ⊇ instrumented runs that violate Q (using HiStar semantics)
2. From IP#, build **safety game G** won by violations of Q
3. From winning strategy for G, generate primitive controller for P

Capsicum Semantics

1. $Q \equiv$



2. τ[action] $\equiv$

A'(x) = A(x) $\vee$ $\exists$ y. C(y) $\wedge$ B(q, p)

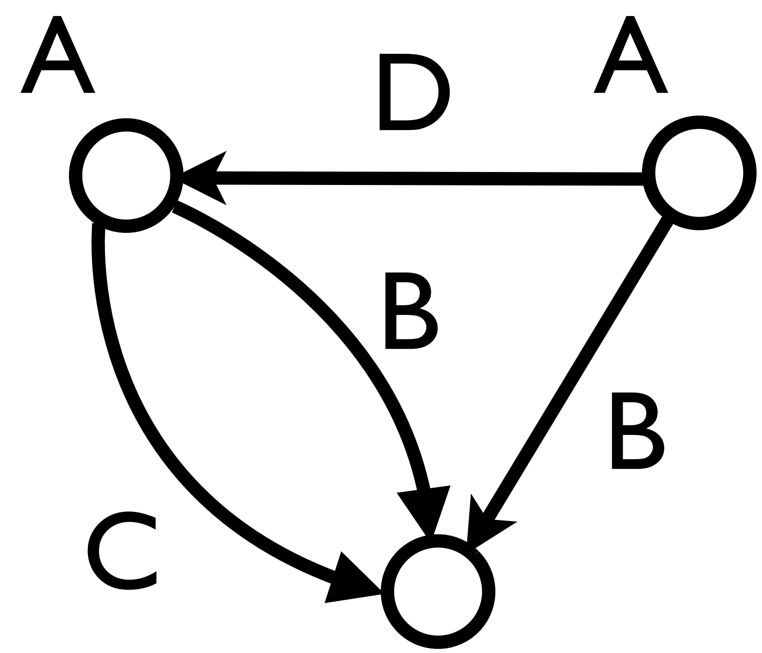
B'(x, y) = B(x, y) $\vee$ (C(x) $\wedge$ D(y))

C'(x) = ...

D'(x) = ...

HiStar Semantics

1. $Q \equiv$

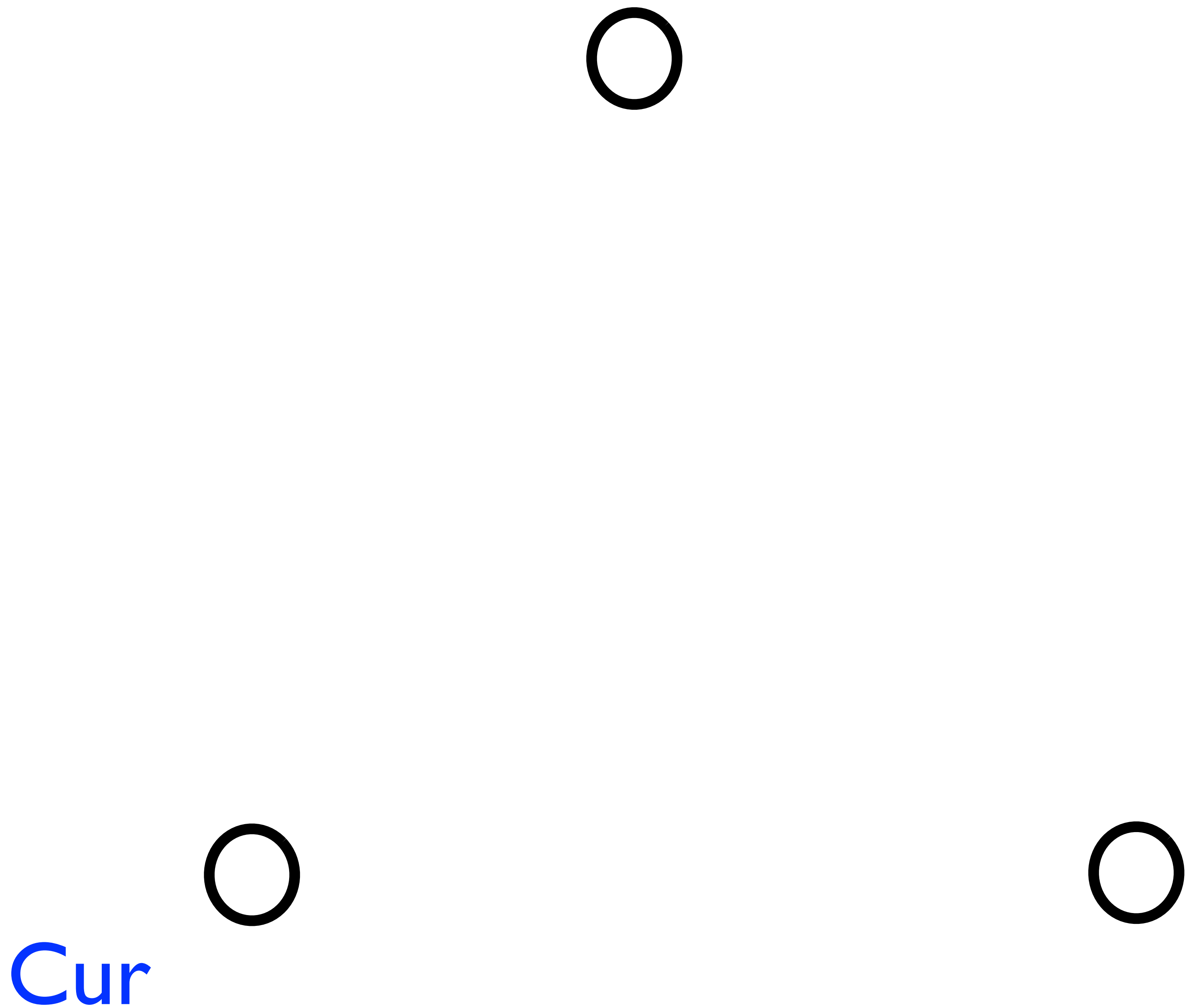


2. τ[action] $\equiv$

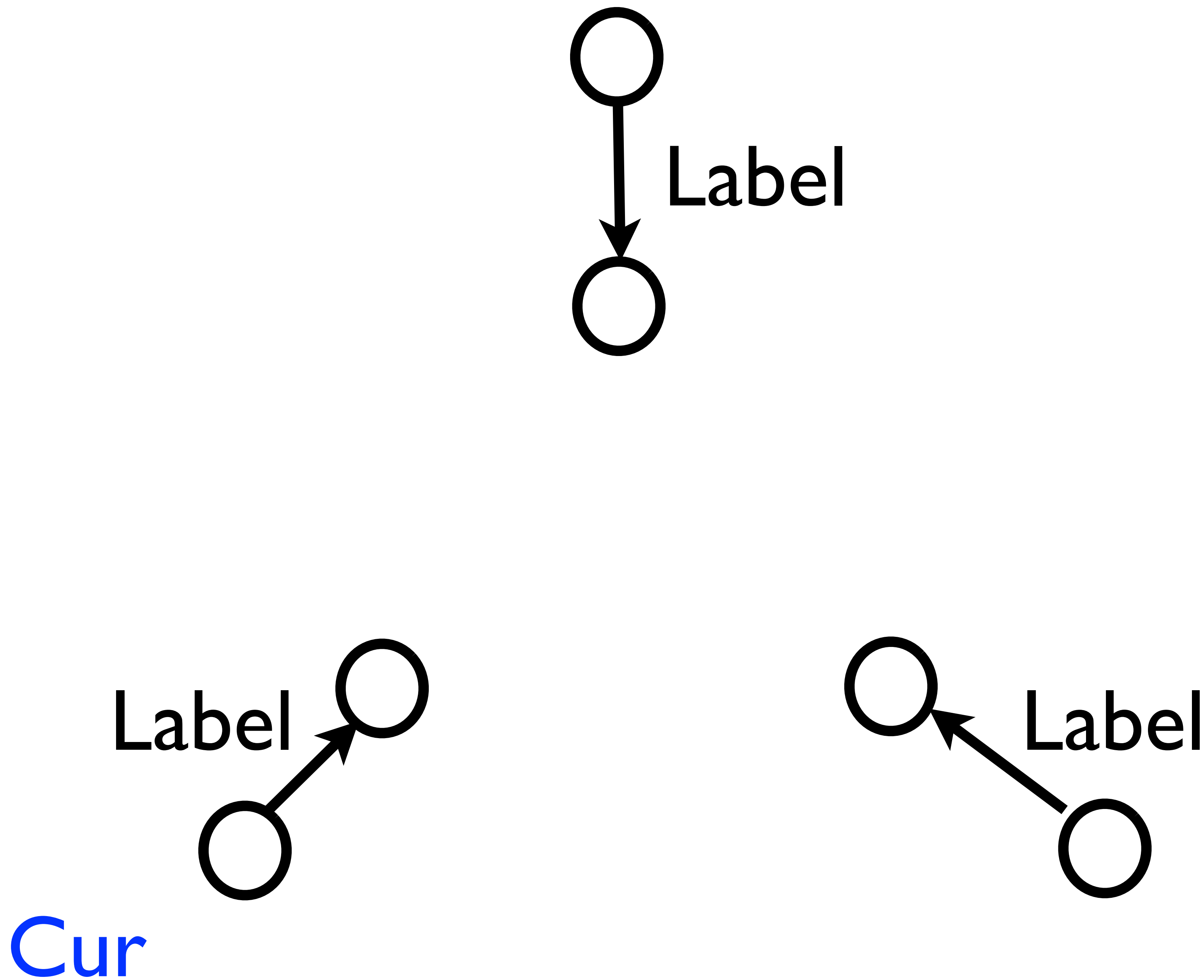
A'(x) = A(x) $\vee$ $\exists$ y. C(y) $\wedge$ B(q, p)
B'(x, y) = B(x, y) $\vee$ (C(x) $\wedge$ D(y))
C'(x) = ...
D'(x) = ...

HiStar State as Structure

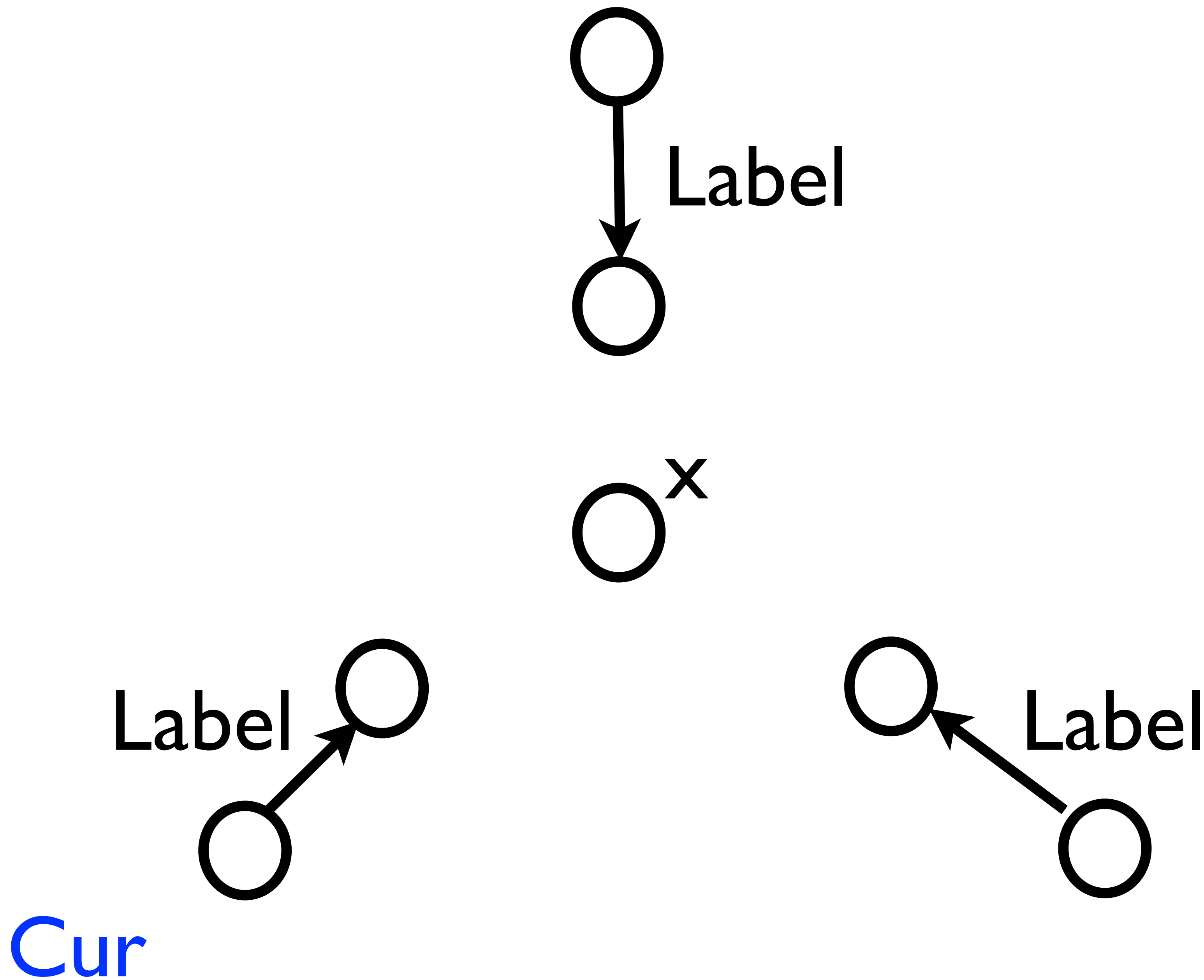
HiStar State as Structure



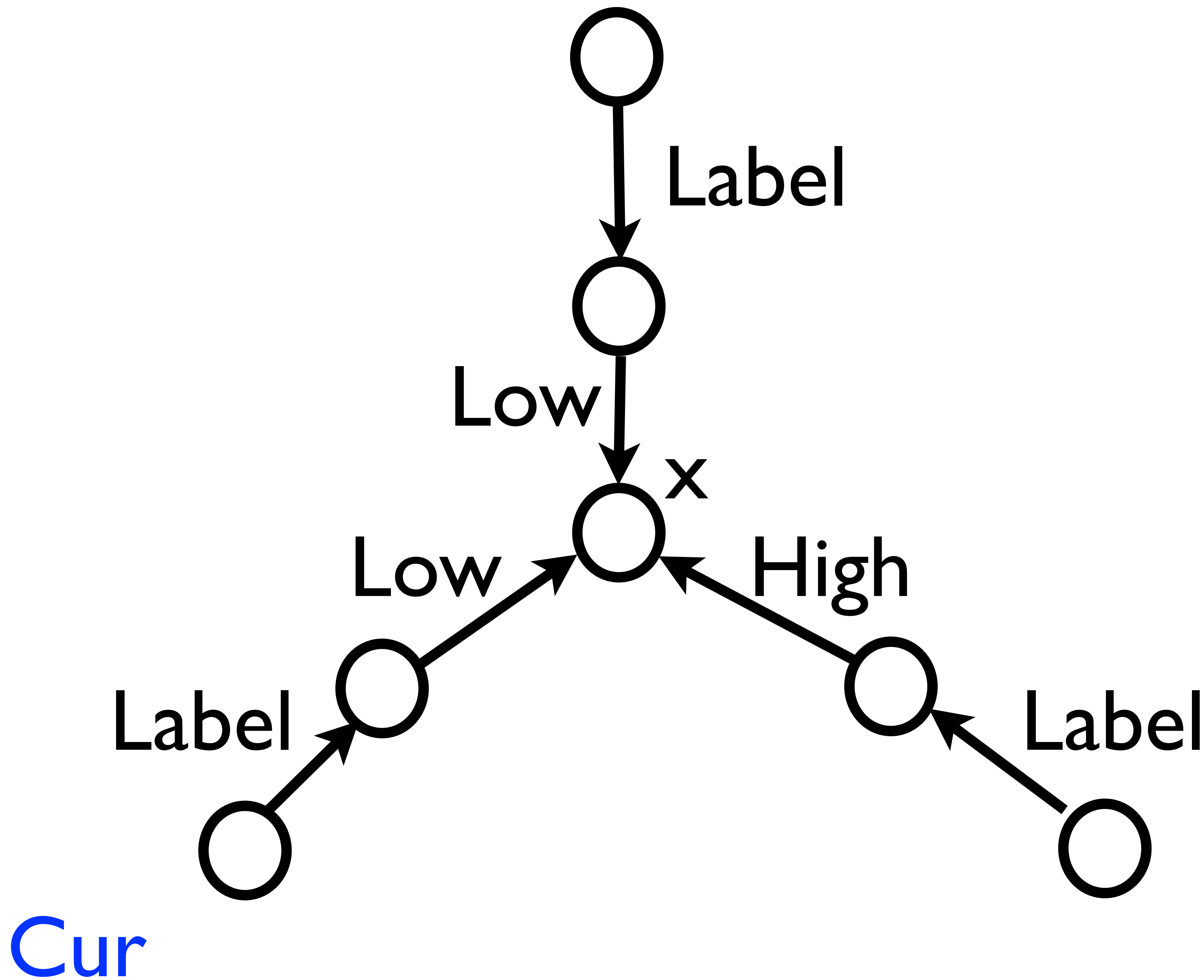
HiStar State as Structure



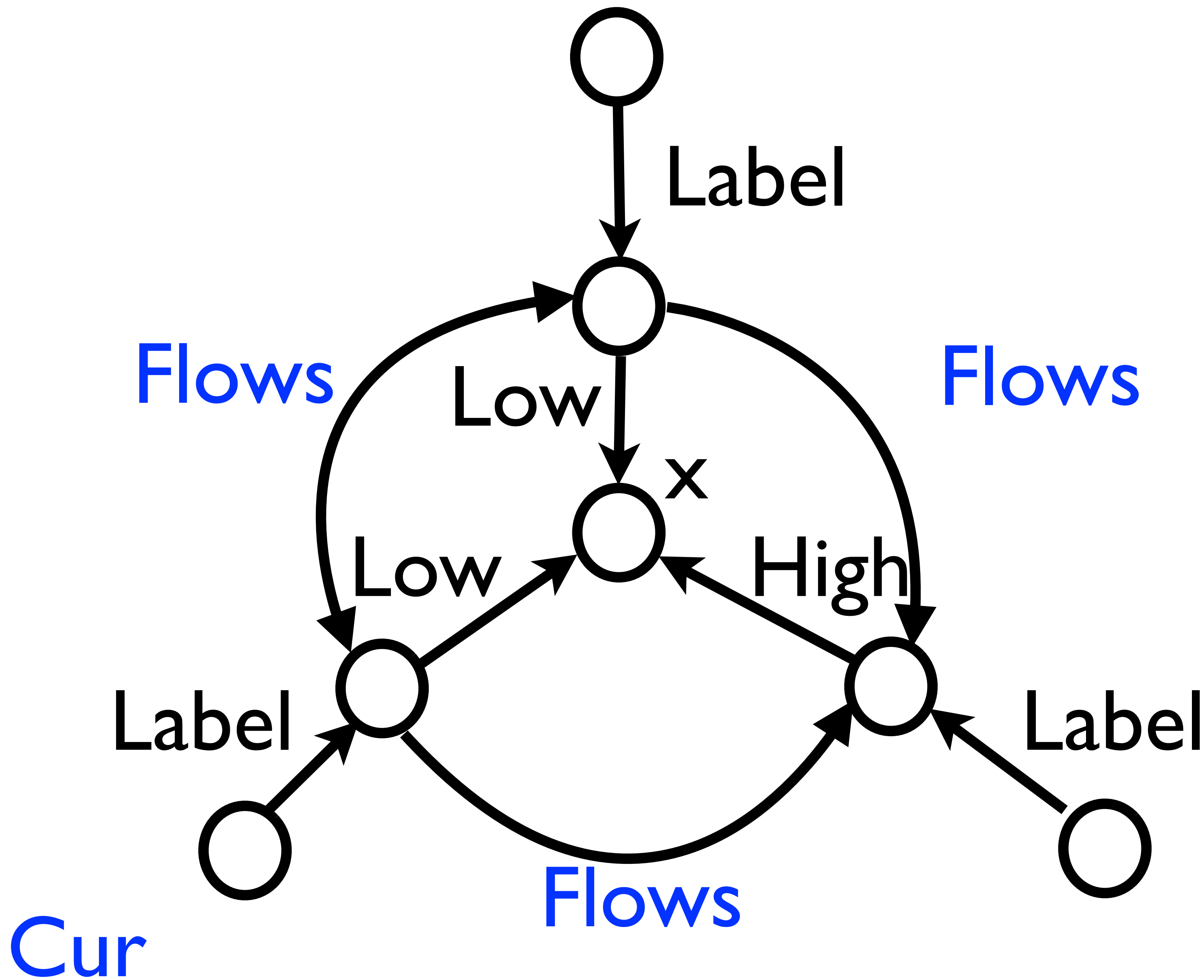
HiStar State as Structure



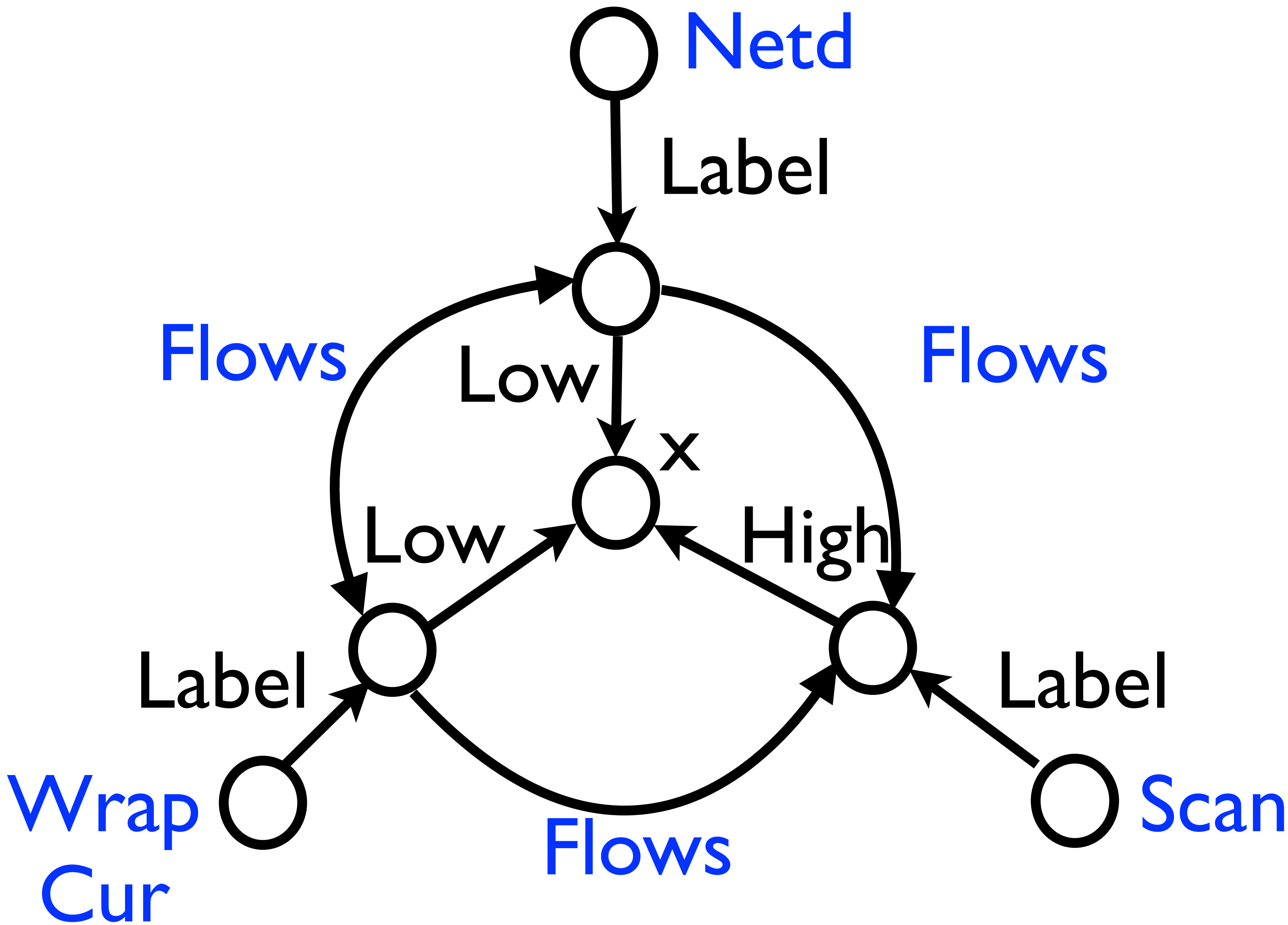
HiStar State as Structure



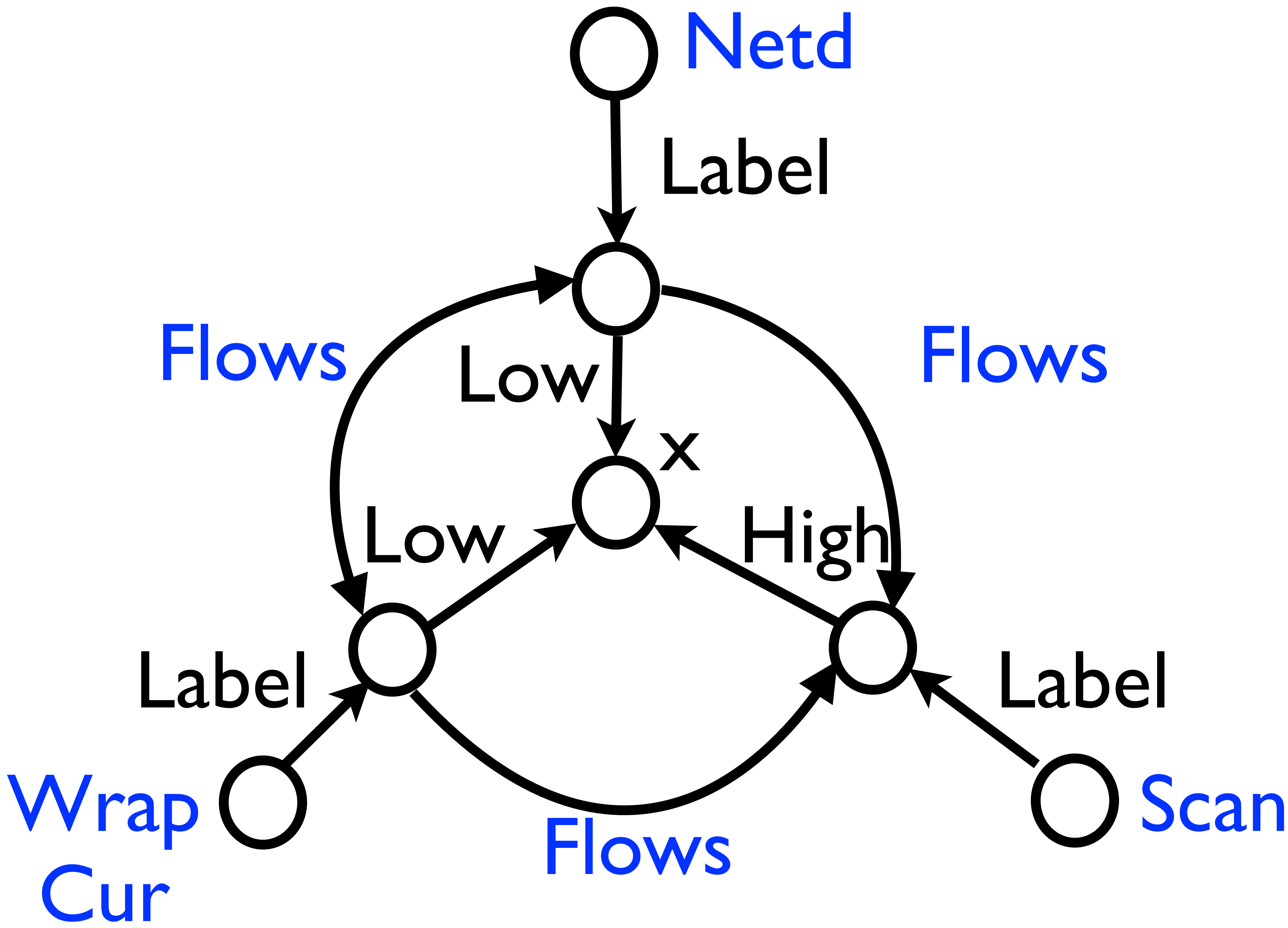
HiStar State as Structure



HiStar State as Structure



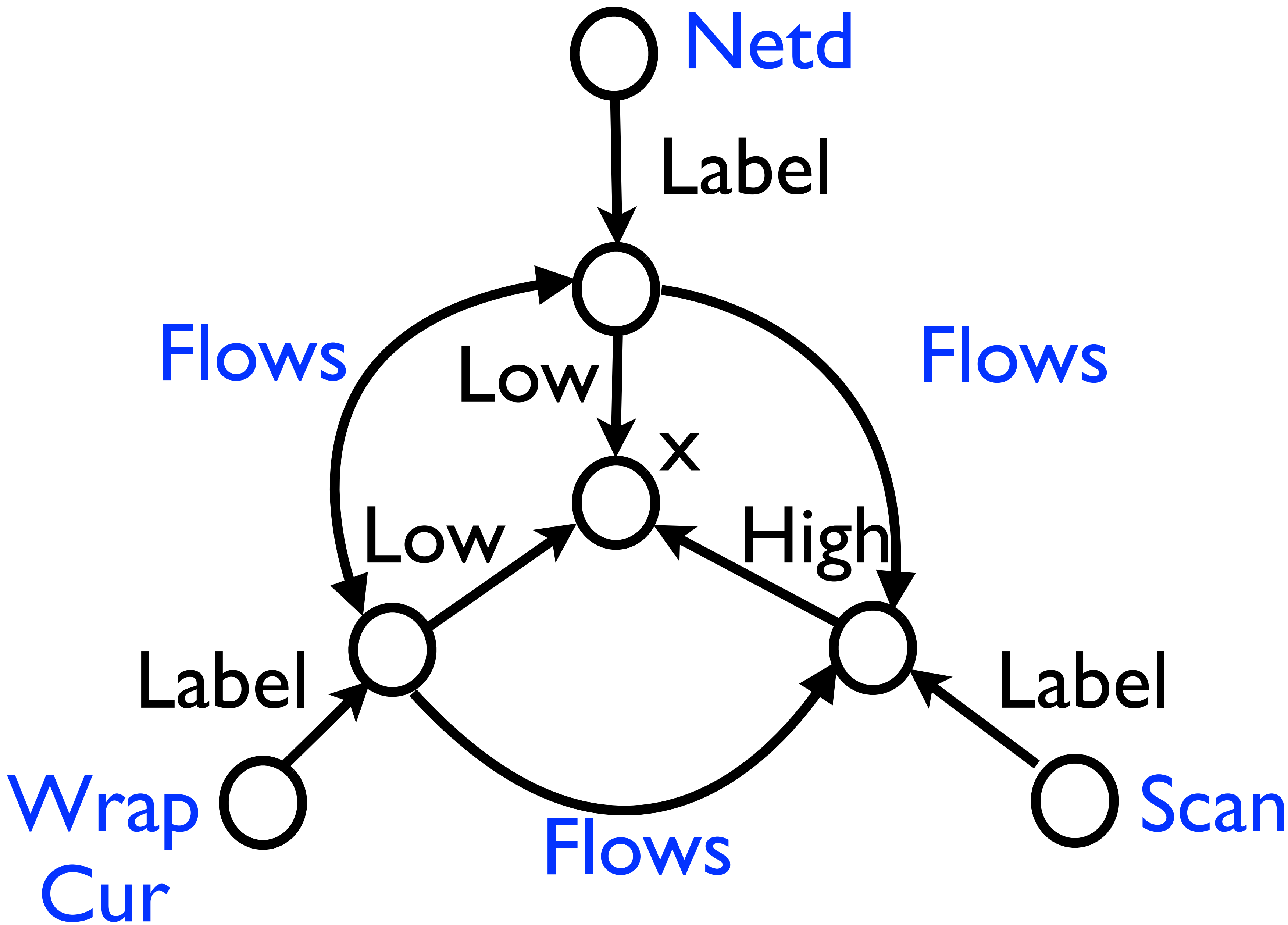
HiStar State as Structure



$$\forall\ w, s, n.\ \mathrm{Wrap}(w) \wedge \mathrm{Scan}(s) \wedge \mathrm{Netd}(n) \Rightarrow$$

$$\mathrm{Flows}(s, w) \wedge \mathrm{Flows}(w, n)$$

HiStar State as Structure

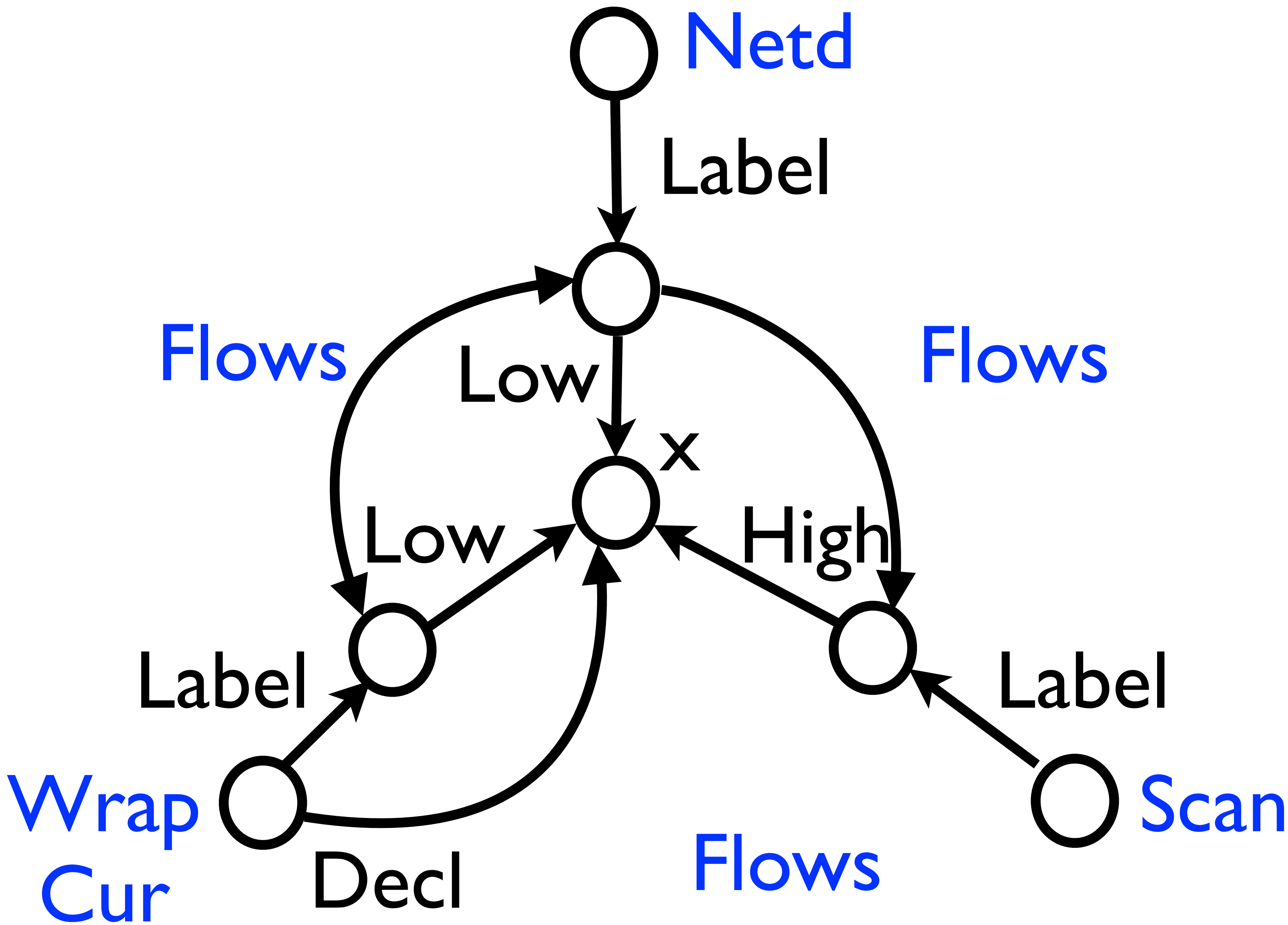


$$\not\models$$

$$\forall\ w, s, n.\ \text{Wrap}(w) \wedge \text{Scan}(s) \wedge \text{Netd}(n) \Rightarrow$$

$$\text{Flows}(s, w) \wedge \text{Flows}(w, n)$$

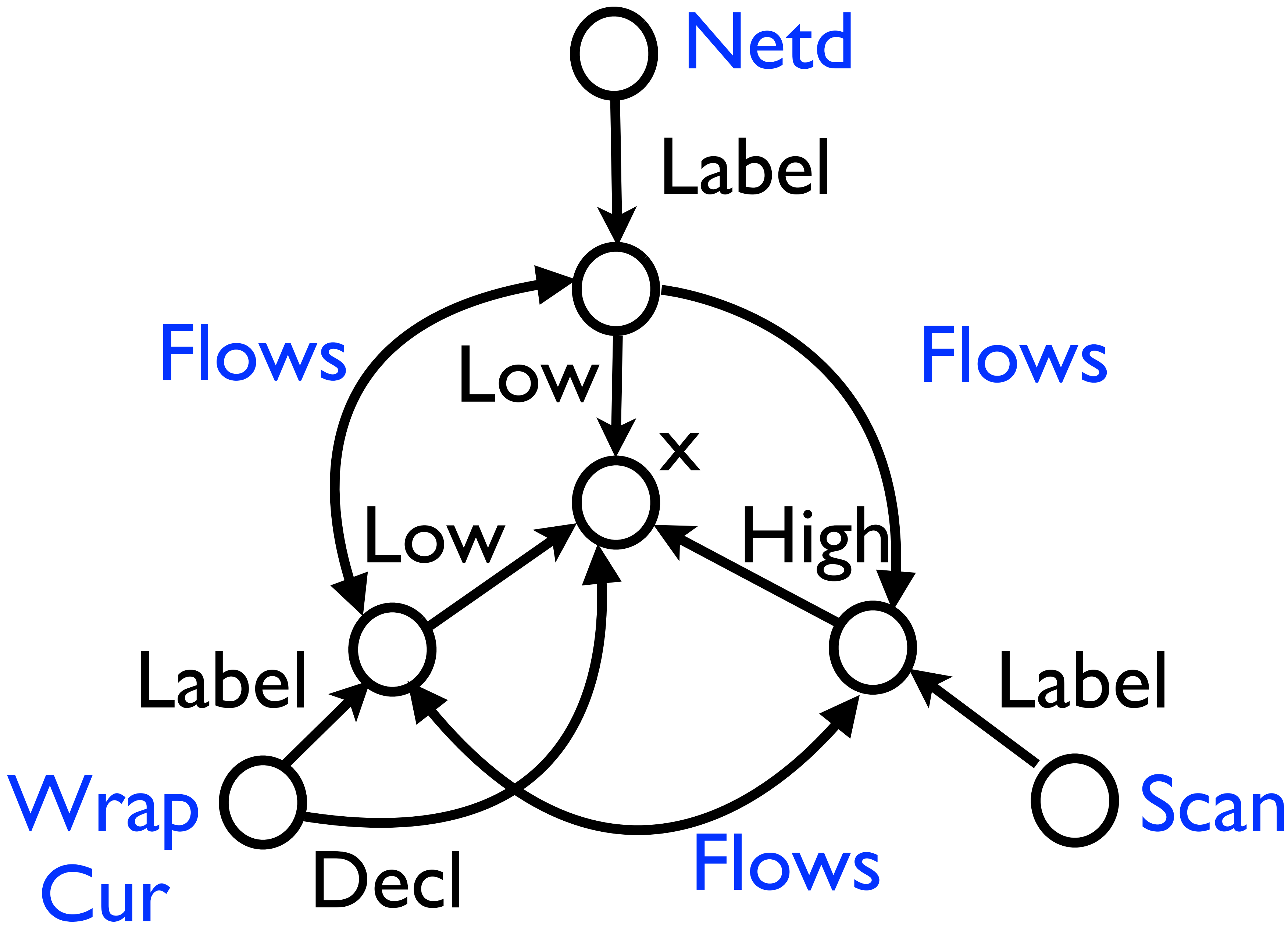
HiStar State as Structure



$$\forall\, w, s, n.\ \mathrm{Wrap}(w) \wedge \mathrm{Scan}(s) \wedge \mathrm{Netd}(n) \Rightarrow$$

$$\mathrm{Flows}(s, w) \wedge \mathrm{Flows}(w, n)$$

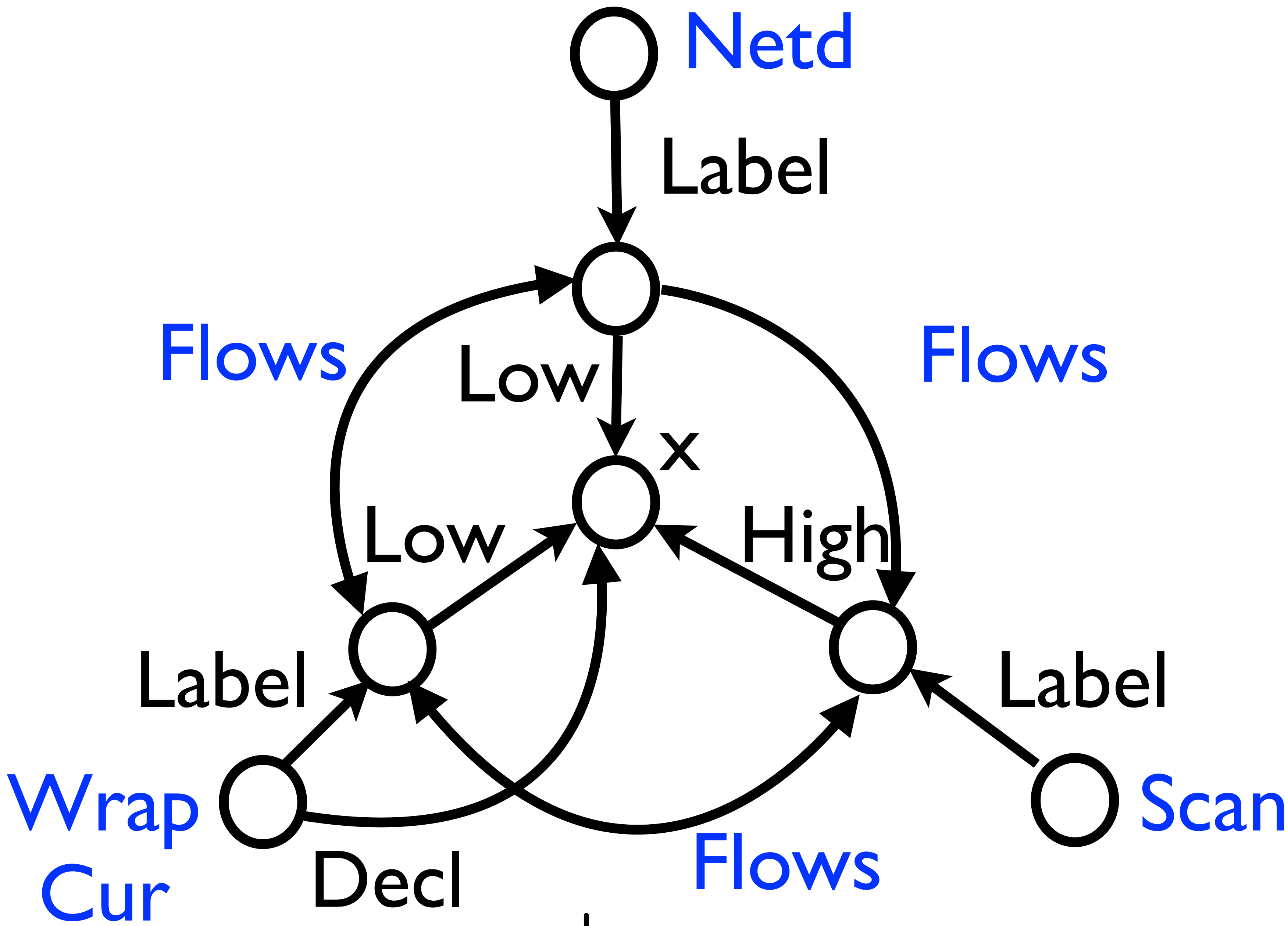
HiStar State as Structure



$$\forall\ w, s, n.\ \mathrm{Wrap}(w) \land \mathrm{Scan}(s) \land \mathrm{Netd}(n) \Rightarrow$$

$$\mathrm{Flows}(s, w) \land \mathrm{Flows}(w, n)$$

HiStar State as Structure



$$\models$$

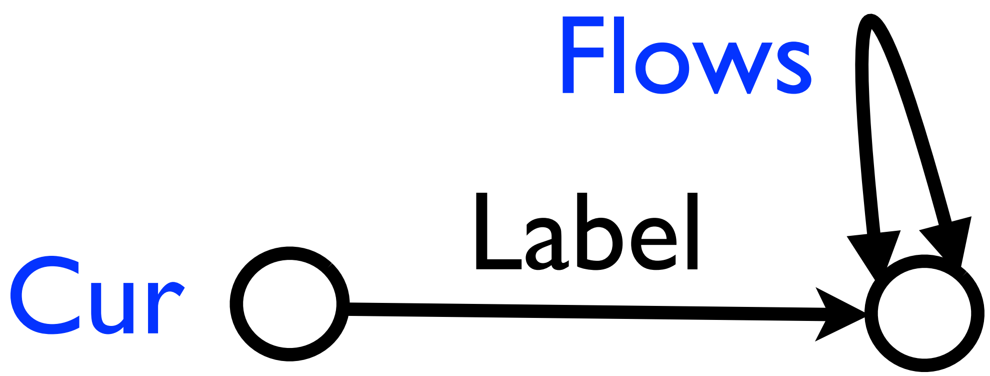
$$\forall\ w, s, n.\ \mathrm{Wrap}(w) \land \mathrm{Scan}(s) \land \mathrm{Netd}(n) \Rightarrow$$

$$\mathrm{Flows}(s, w) \land \mathrm{Flows}(w, n)$$

HiStar State Transformers

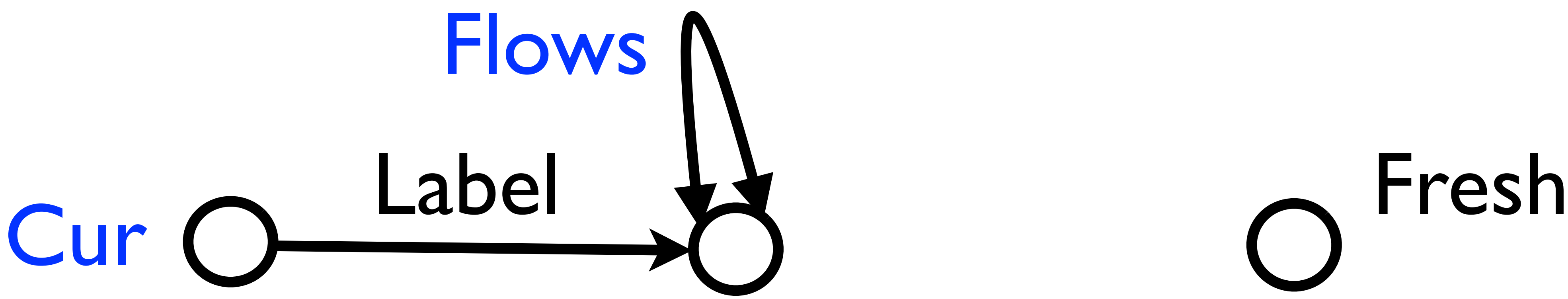
Action	
create_cat(&x)	

HiStar State Transformers



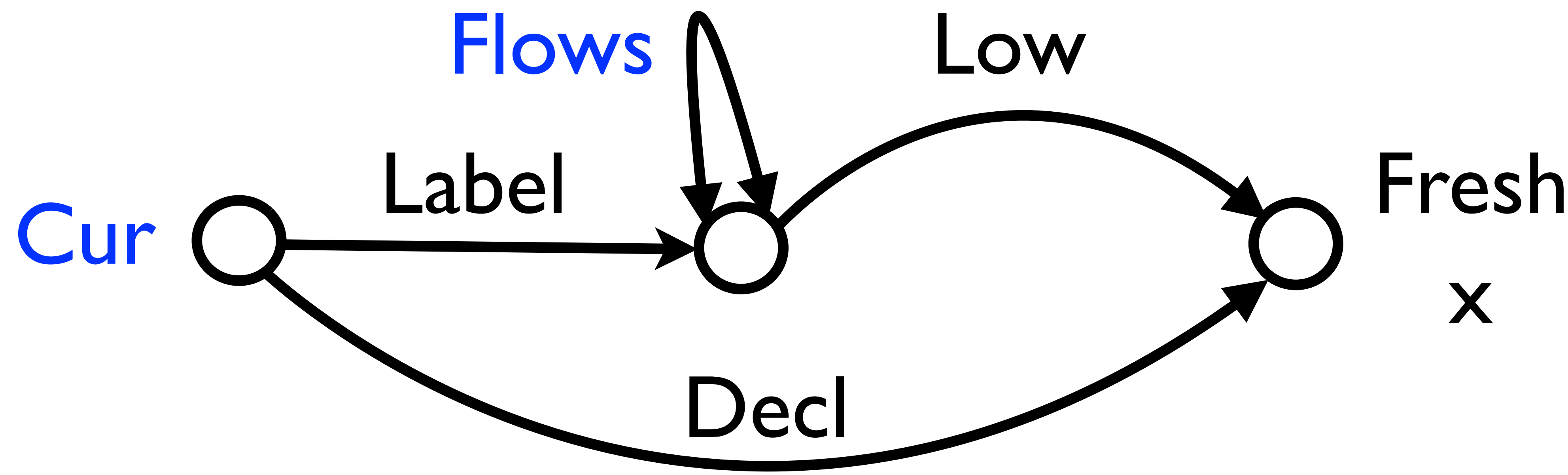
Action	
create_cat(&x)	

HiStar State Transformers



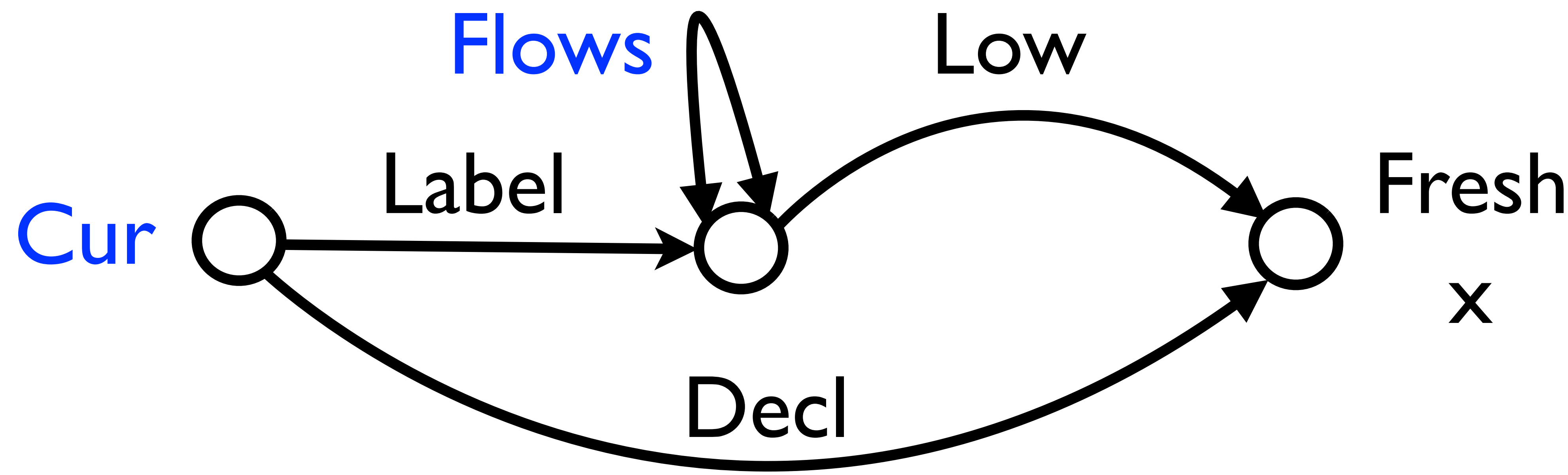
Action	
create_cat(&x)	

HiStar State Transformers



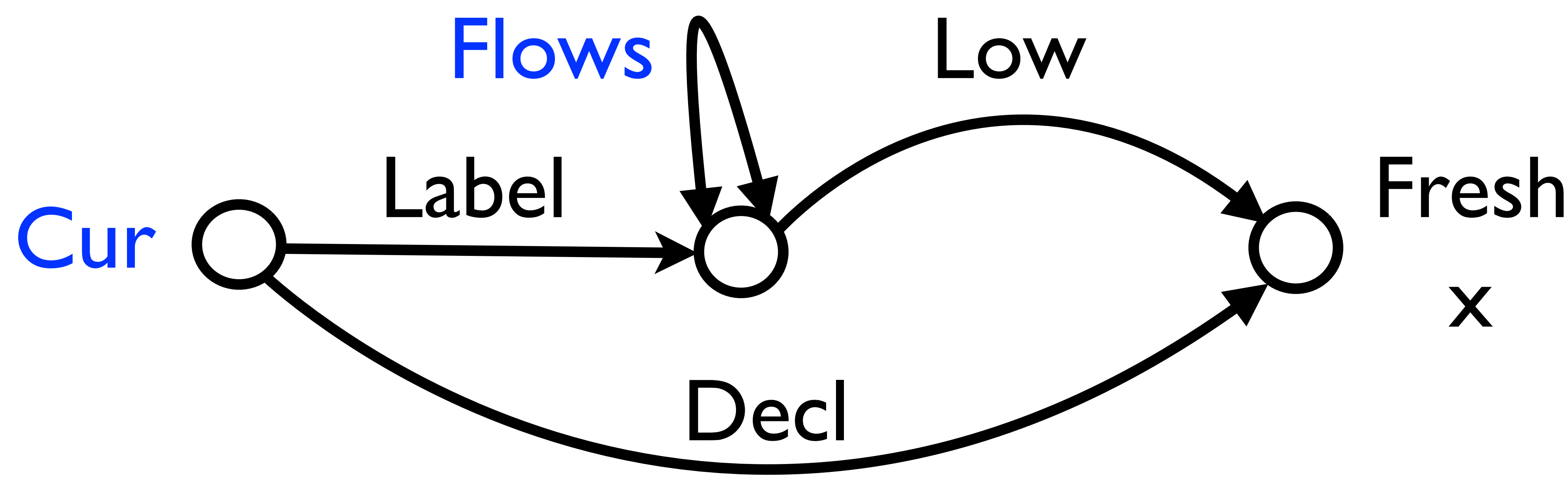
Action	
create_cat(&x)	

HiStar State Transformers



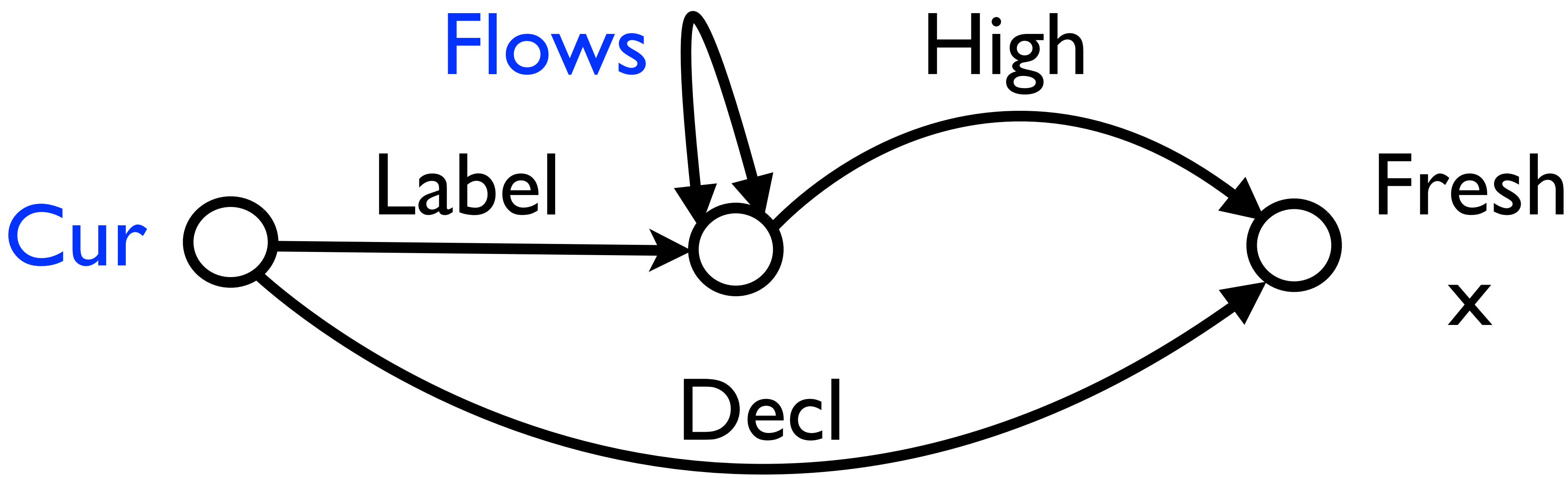
Action	Structure Transform
create_cat(&x)	Intro Fresh Decl'(p, c) := Decl(p, c) $\vee$ (Cur(p) $\wedge$ Fresh(c))

HiStar State Transformers



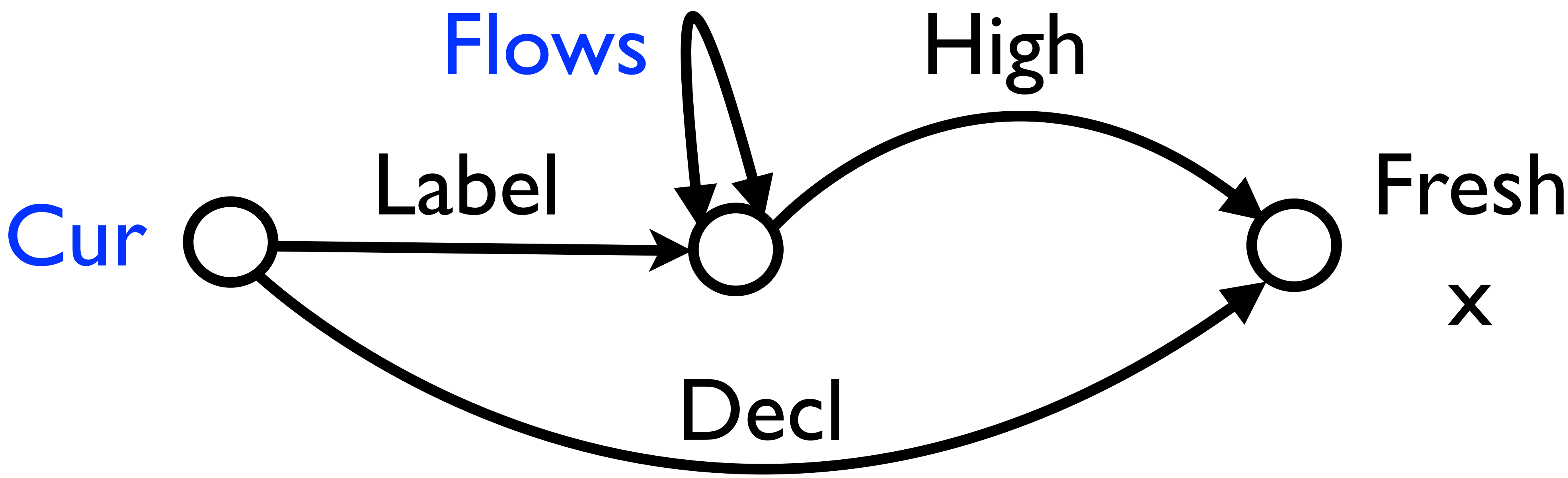
Action	
raise(&x)	

HiStar State Transformers



Action	
raise(&x)	

HiStar State Transformers

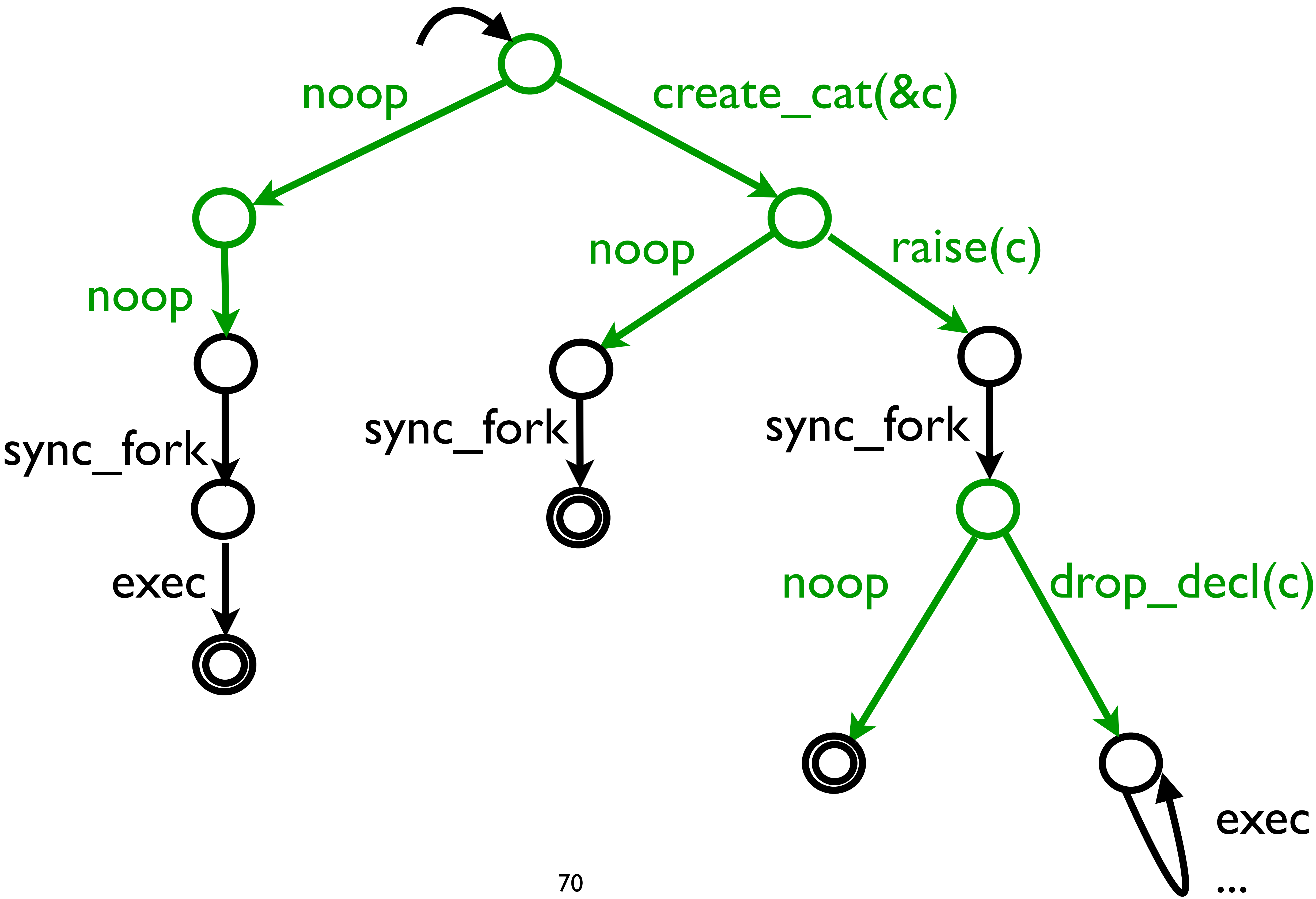


Action	Structure Transform
raise(&x)	Intro Fresh High'(l, c) := High(l, c) $\vee$ $\exists$ p. Cur(p) & Label(p, l) & x(c)

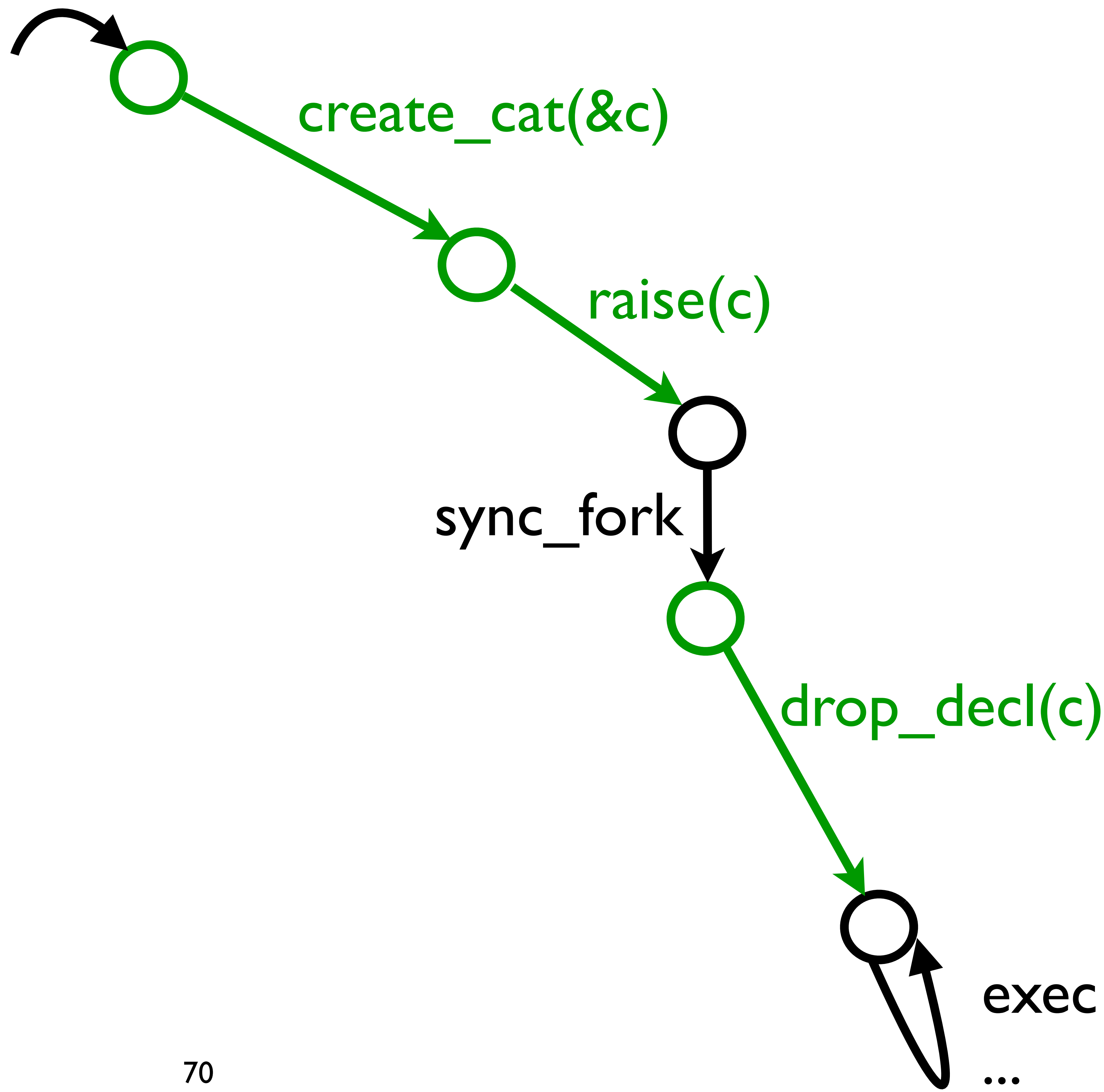
Summary: HiStar Semantics

- We can define the HiStar semantics as FOL predicate transforms and automatically generate a weaver for HiStar
- FOL predicate transforms can describe capability **and DIFC** semantics

Scanner Game

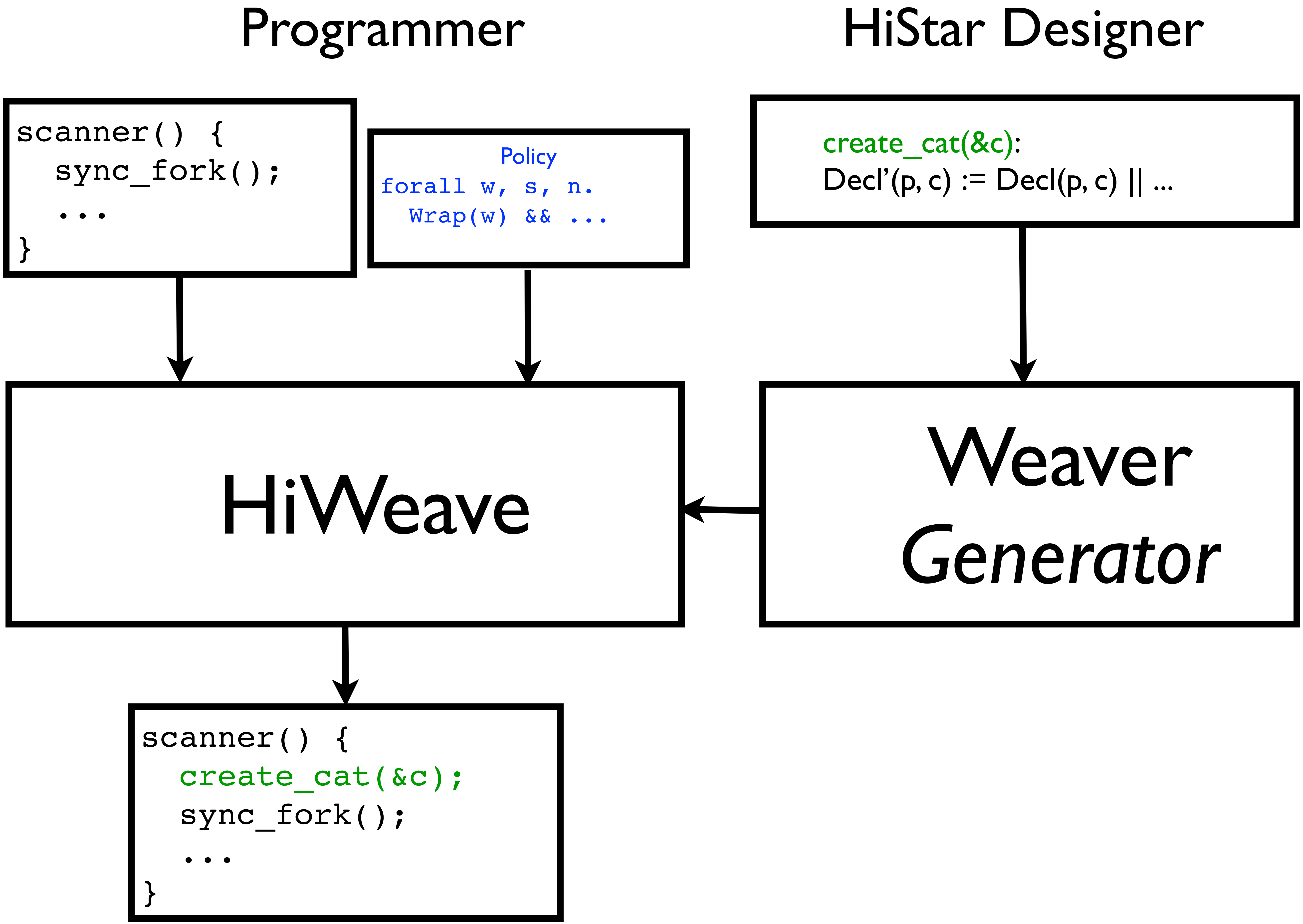


Scanner Game



HiWeave α Performance

Generates code for `clamwrap` in < 3 mins


Programmer
HiStar Designer
scanner() {
sync_fork();
...
}
Policy
forall w, s, n.
Wrap(w) && ...
create_cat(&c):
Decl'(p, c) := Decl(p, c) || ...
HiWeave
Weaver
Generator
scanner() {
create_cat(&c);
sync_fork();
...
}

Outline

1. Motivation, problem statement
2. Previous work: Capsicum
3. Ongoing work: HiStar
4. Open challenges

Open Challenges

- Automating abstraction refinement
- Automating error diagnosis
- Compositional synthesis
- Optimizing generated code
- Designing a policy logic

Automating Abstraction Refinement

- Picking the right abstraction predicates requires a lot of design effort
- Can we refine the abstraction predicates via counter-strategies?

Automating Error Diagnosis

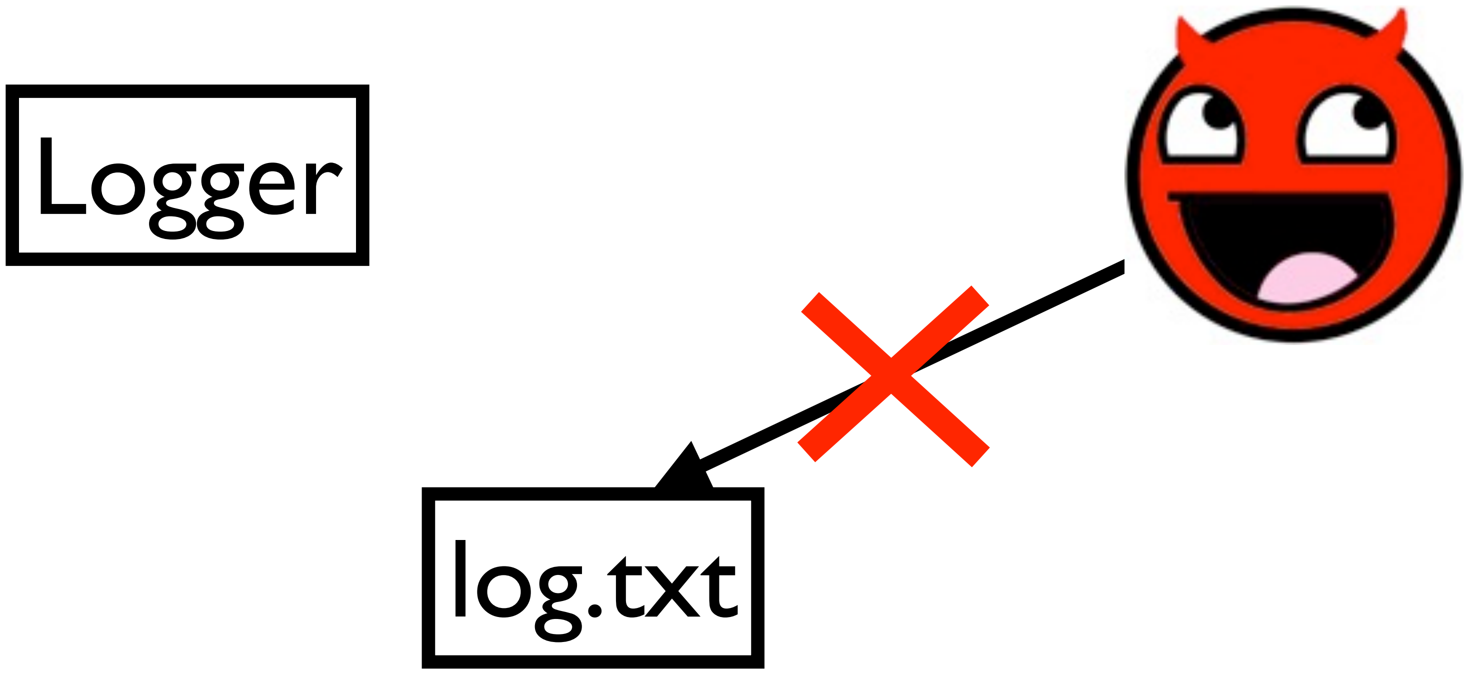
- When weaver fails, it has a counter-strategy
- How can we simplify these when presenting them to the user?

Compositional Synthesis

- Real programs are structured
  as a composition of processes
- Policies are expressed naturally as conjunction of local, global policies
- Can we adapt compositional verification?
  [Long, '89]

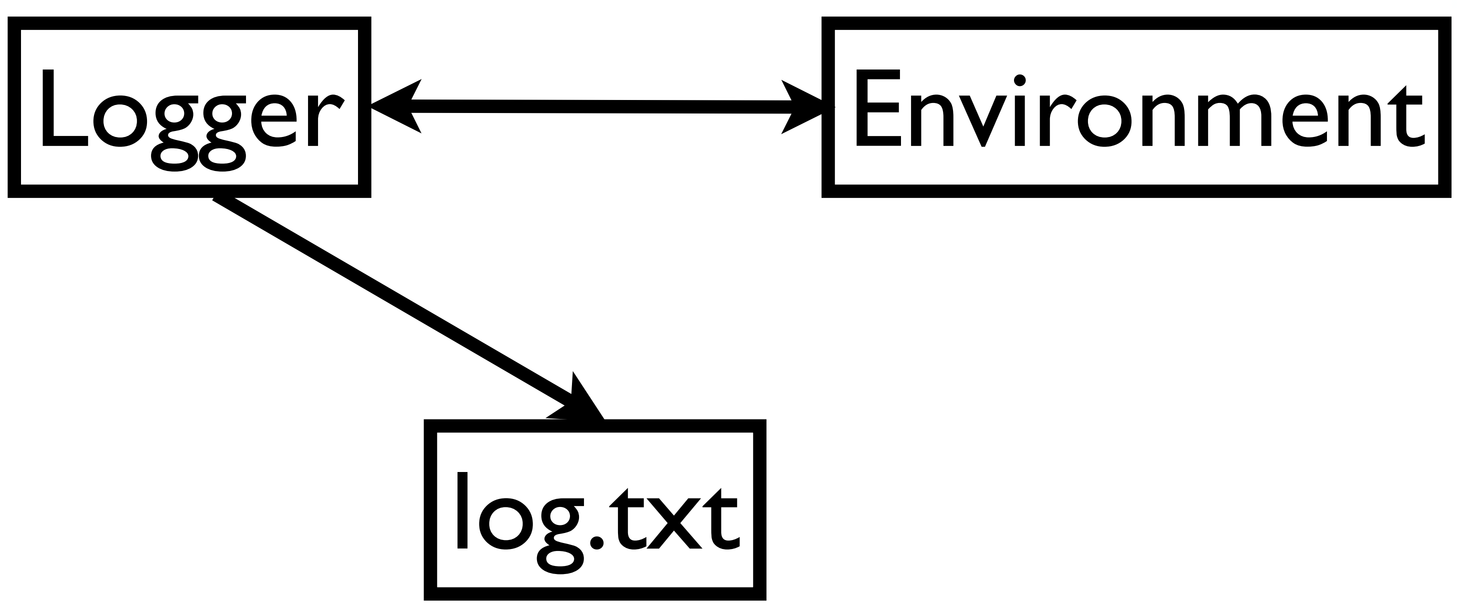
HiStar Logger

Local (security) policy: only Logger should be able to modify log



HiStar Logger

Global (functionality) policy: under certain conditions, Logger will append log on behalf of Environment



Optimizing Generated Code

- Mean-payoff games present an appealing cost model, but have high complexity in general
- Can we apply any domain specific optimizations?

Designing a Policy Logic

- The weaver generator allows a policy writer to declare policies purely over privileges
- What logic over privileges is easiest for a policy writer to understand?
- How do we evaluate value added?

Our Collaborators

Capsicum-dev

Pawel Jakub Dawidek

Khilan Gudka

Ben Laurie

Peter Neumann

MIT-LL

Jeffrey Seibert

Michael Zhivich

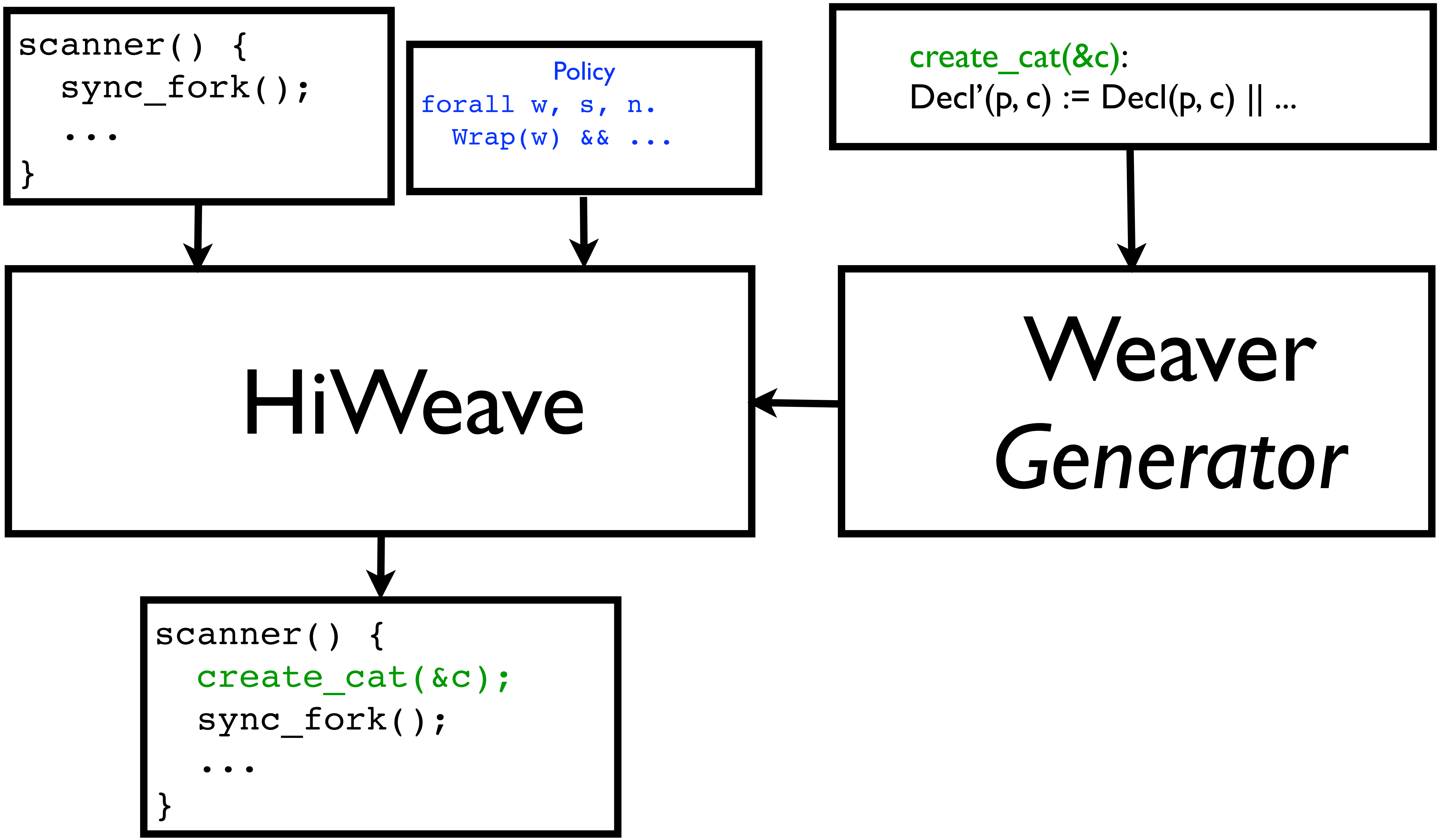
HiStar

Nickolai Zeldovich

TVLA

Mooly Sagiv

Questions?



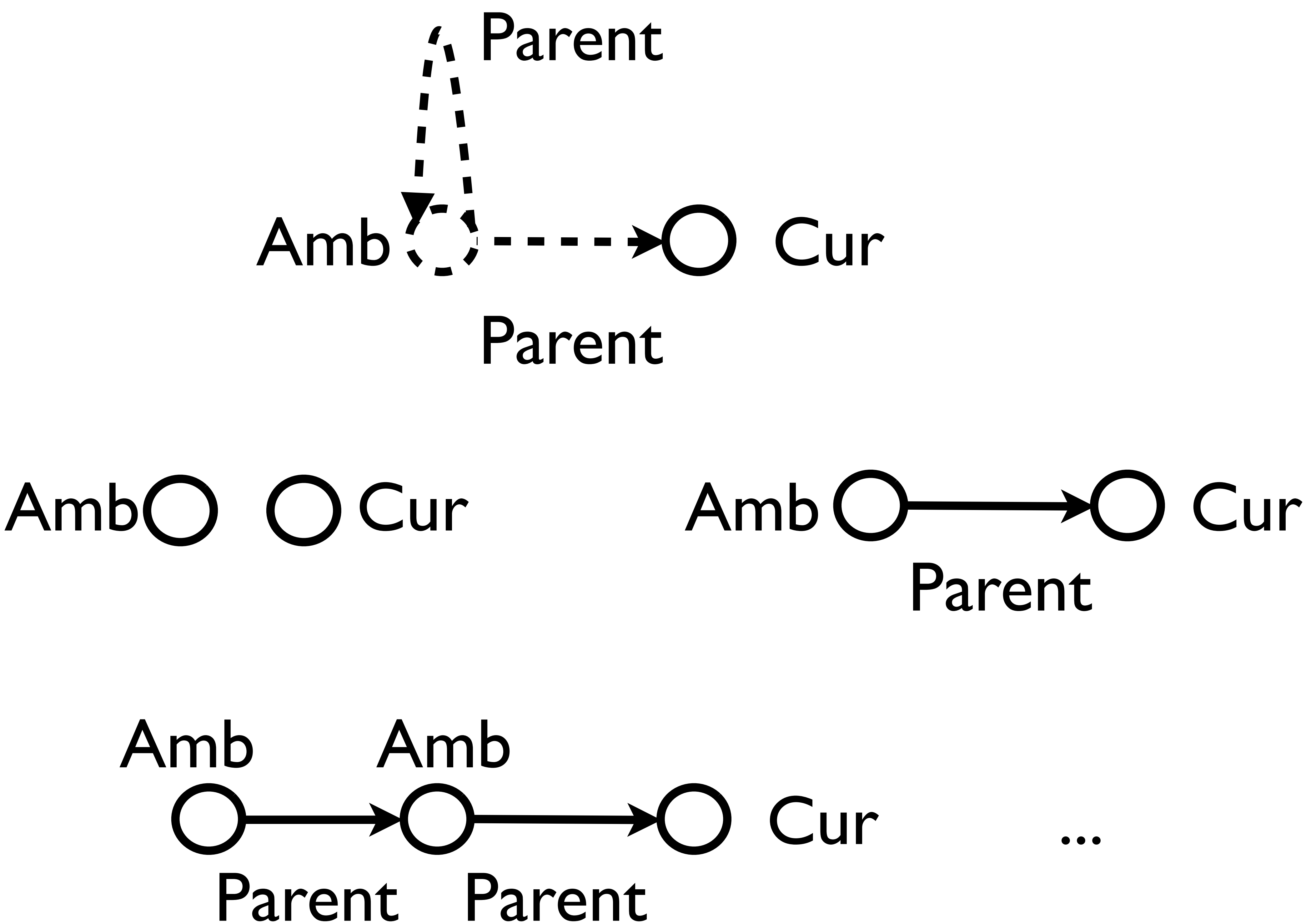
scanner() {
sync_fork();
...
}
Policy
forall w, s, n.
Wrap(w) && ...
create_cat(&c):
Decl'(p, c) := Decl(p, c) || ...
HiWeave
Weaver
Generator
scanner() {
create_cat(&c);
sync_fork();
...
}

Extra Slides

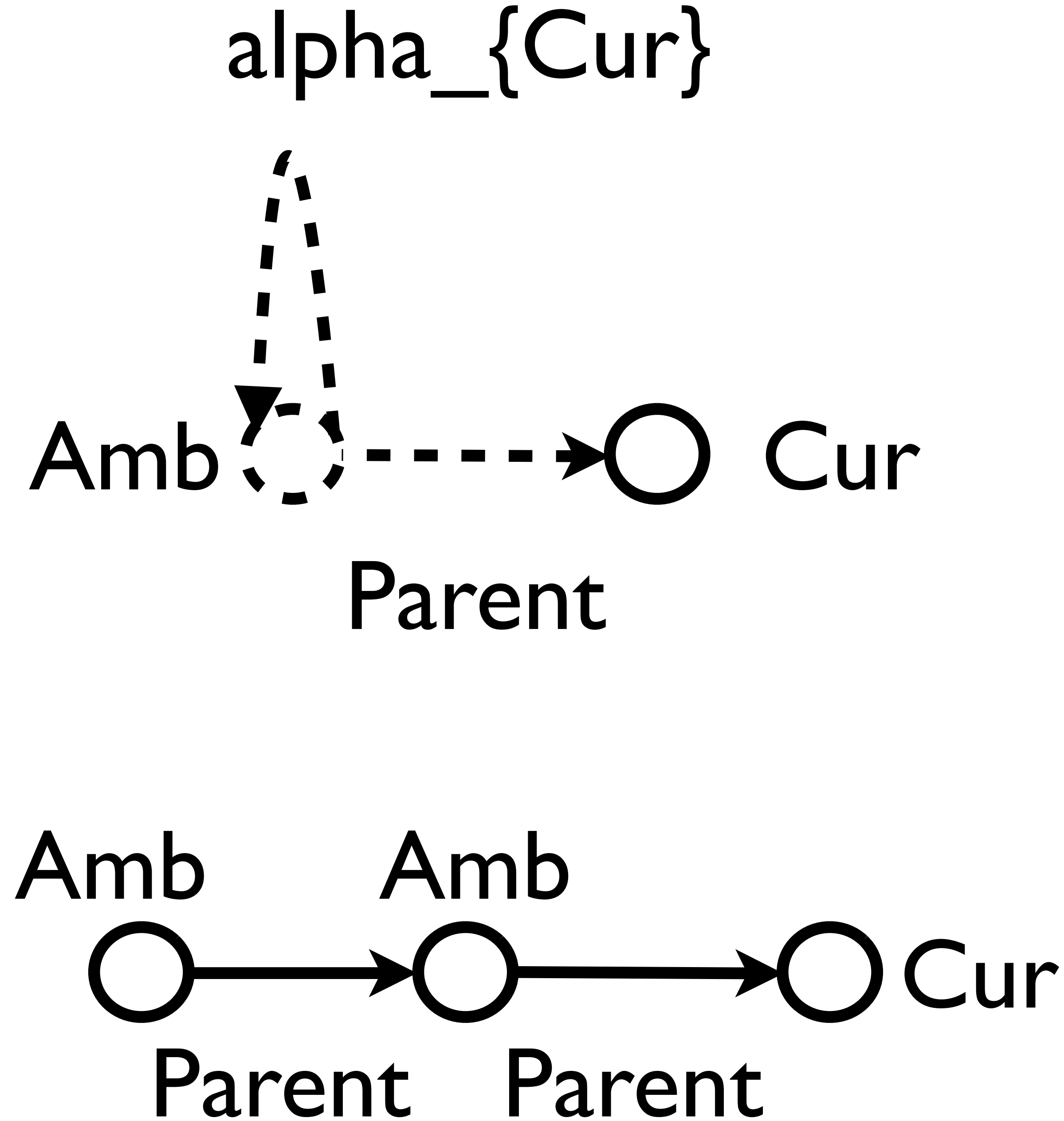
Three-valued logic

- Values: true, false, and *unknown*
- true & *unknown* = *unknown*
- false & *unknown* = false

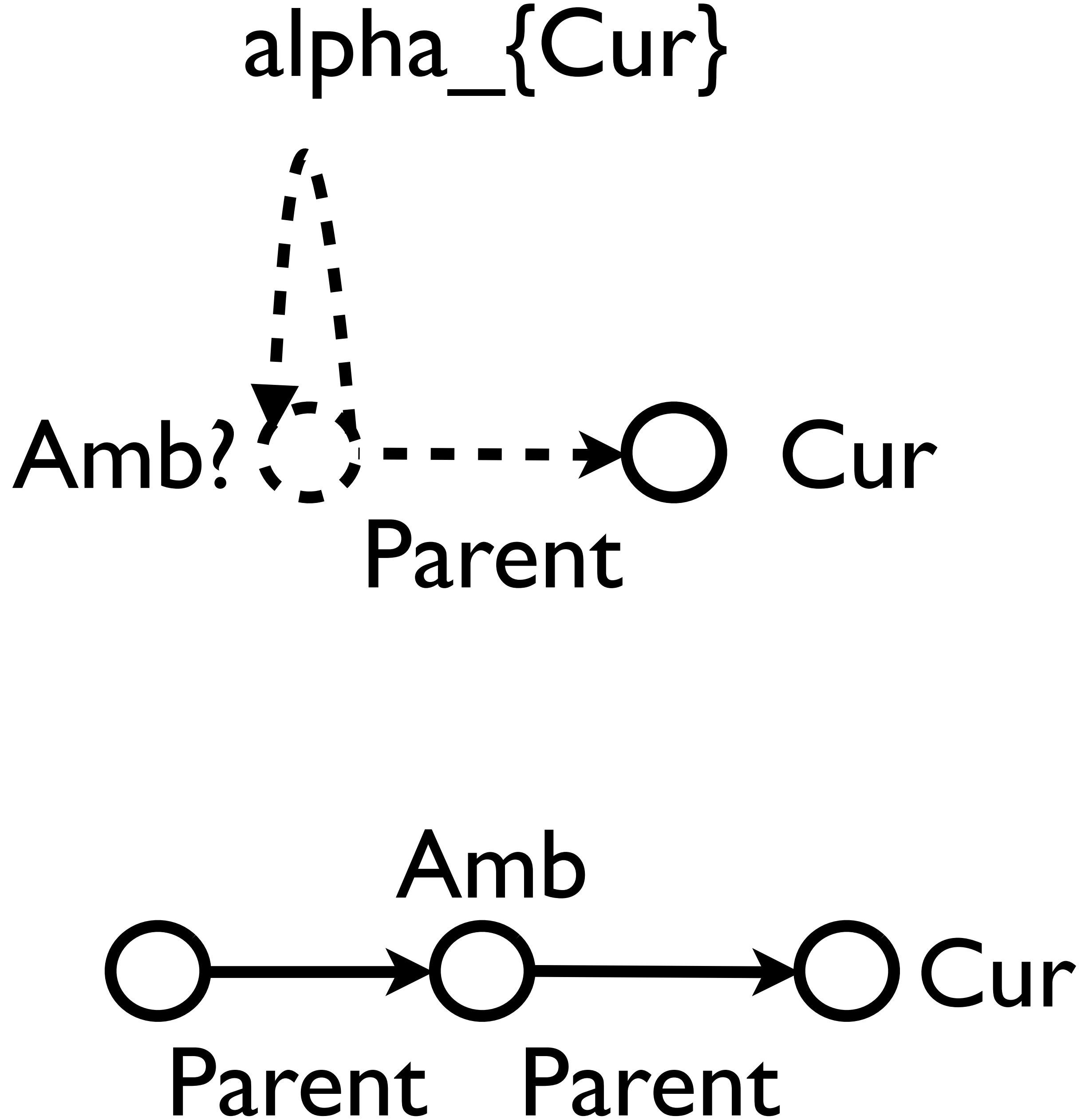
Three-valued Structures



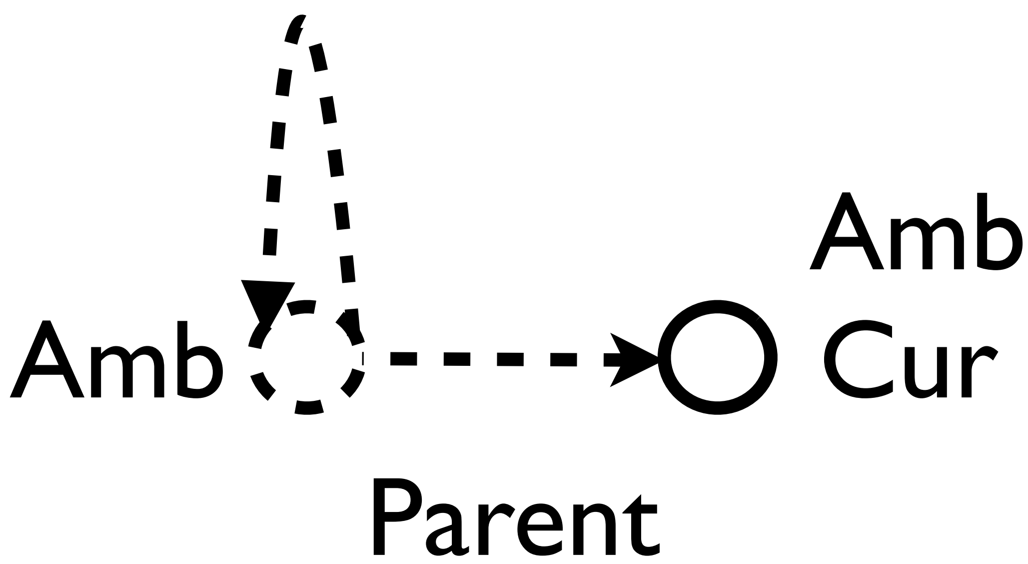
Abstraction Function

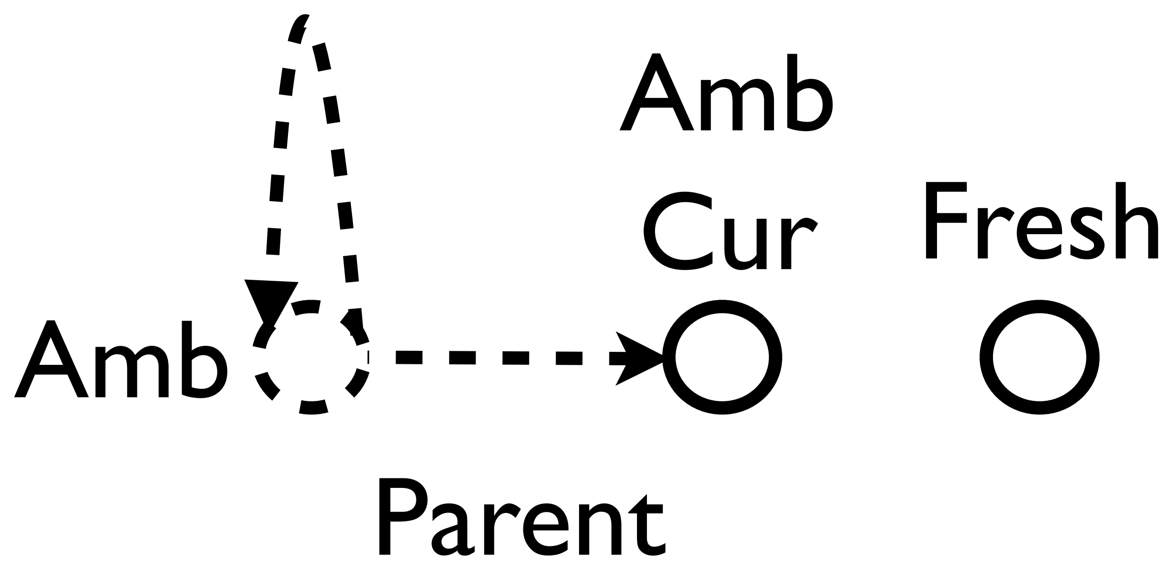


Abstraction Function



Abstract Fork (def)





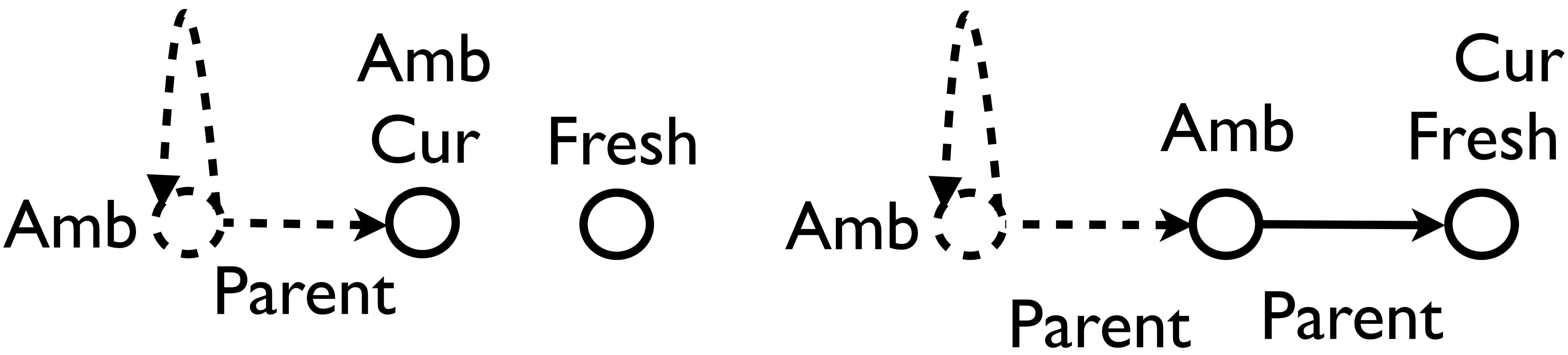
Action

Semantics

fork()

Intro Fresh
Amb(p) := Amb(p)
|| (Fresh(p) & E q. Cur(q) & Amb(q))

Abstract Fork (definite)



Action

Semantics

fork()

Intro Fresh
Amb(p) := Amb(p)
|| (Fresh(p) & E q. Cur(q) & Amb(q))