Spoq: Scaling Machine-Checkable Systems Verification in Coq

Xupeng Li, Xuheng Li, Wei Qiang, Ronghui Gu, Jason Nieh

Columbia University

Motivation

Everybody wants …

Absolutely Correct

Operating System

File System

Cloud Hypervisor

…

Motivation

Formal Verification …

Absolutely Correct √

Operating System

File System

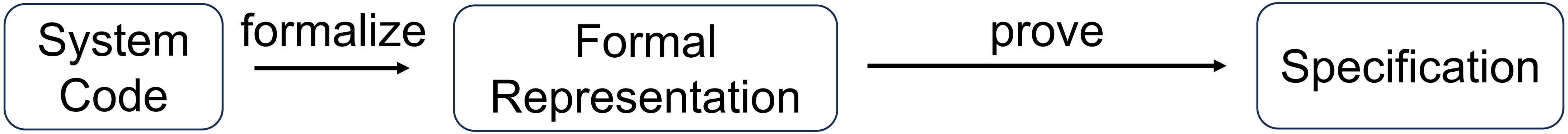
Cloud Hypervisor

…

$\sqsubseteq$

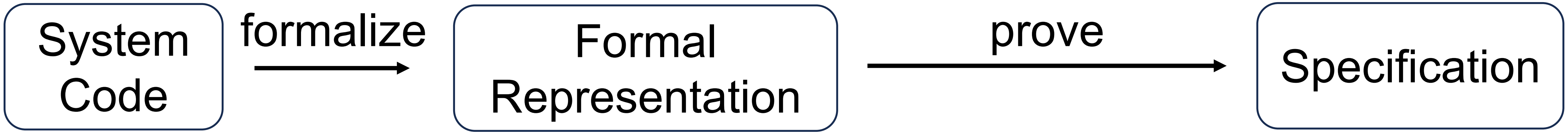
Expectation
(Formal Specification)

Workflow of System Verification



“Functional Correctness”

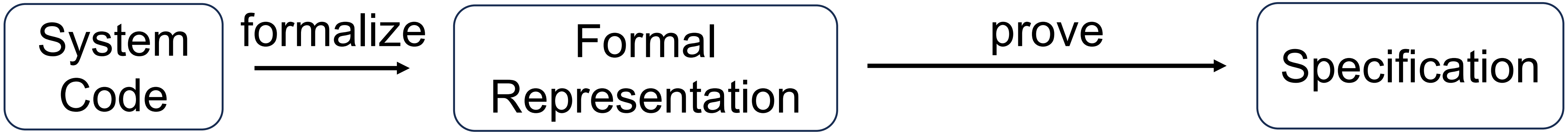
Workflow of System Verification



“Functional Correctness”

- IMPORTANT Implementation satisfies specification

Workflow of System Verification



System Code

formalize

Formal Representation

prove

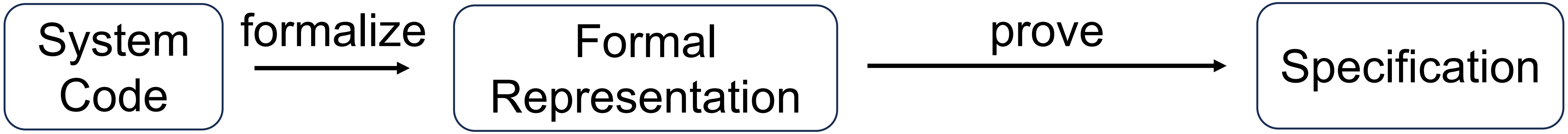
Specification

“Functional Correctness”

- IMPORTANT Implementation satisfies specification
- specification satisfy higher-level properties (i.e. security)

→ properties hold on implementation

Workflow of System Verification



System Code

formalize

Formal Representation

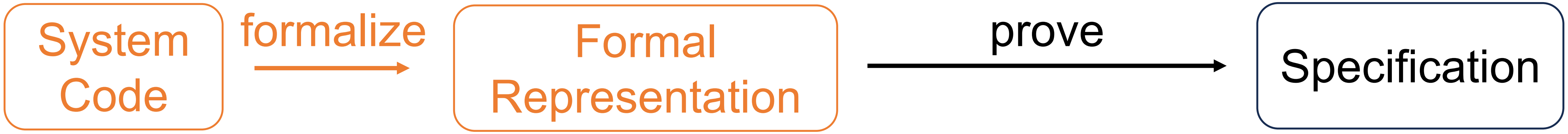
prove

Specification

“Functional Correctness”

- IMPORTANT Implementation satisfies specification
- specification satisfy higher-level properties (i.e. security)
    - → properties hold on implementation
- Most challenging

Challenge 1: Intractable Original Code



System Code

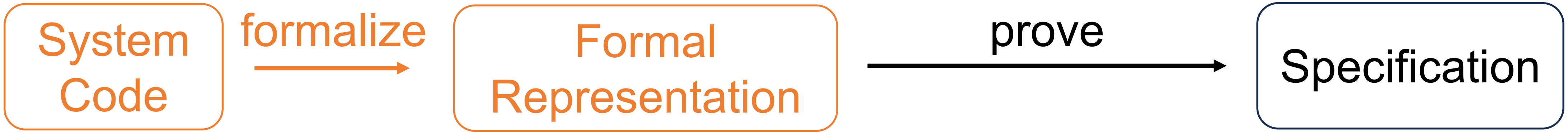
formalize

Formal Representation

prove

Specification

Challenge 1: Intractable Original Code



System Code

formalize

Formal Representation

prove

Specification

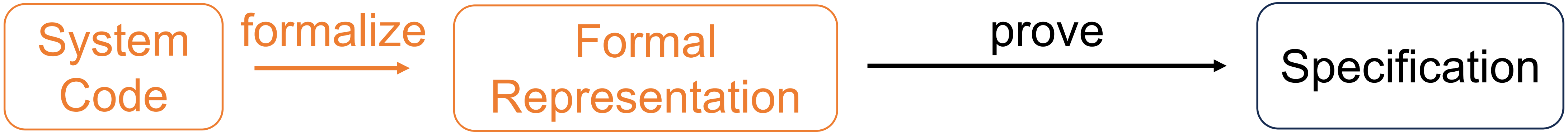
✗ Compiler Derivatives

```
#define __hyp_text __section(.hyp.text) notrace
u32 ~~__hyp_text~~ mem_region_search(u64 addr)
```

✗ Statement expression: GNU extension

```
#define readl_relaxed(c)                                  \
  ({ u32 __r =                                            \
     le32_to_cpu((__force __le32)__raw_readl(c));         \
     __r;})
```

Challenge 1: Intractable Original Code



✗ Compiler Derivatives

```
#define __hyp_text __section(.hyp.text) notrace
u32 ~~__hyp_text~~ mem_region_search(u64 addr)
```

ClightGen

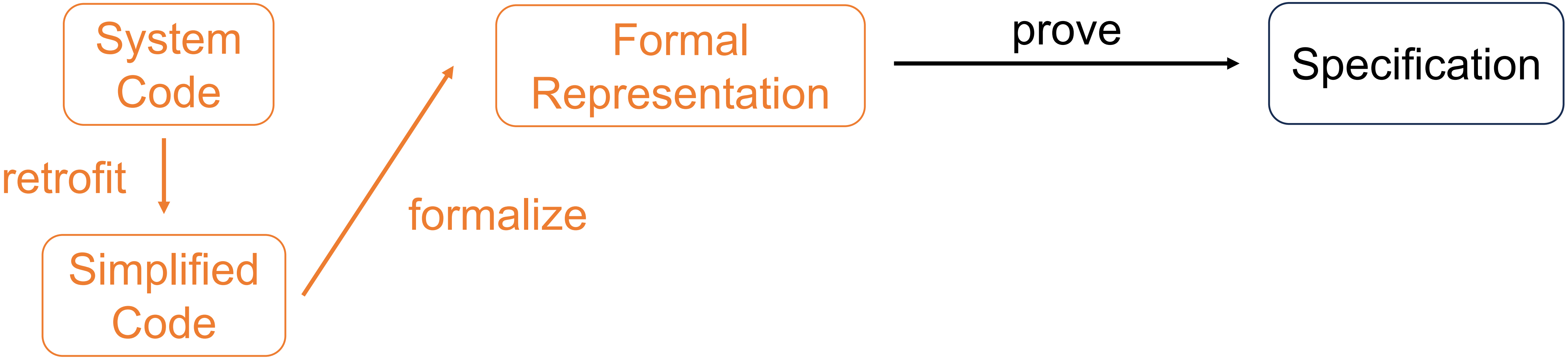
✗ Statement expression: GNU extension

```
#define readl_relaxed(c)                              \
  ({ u32 __r =                                        \
     le32_to_cpu((__force __le32)__raw_readl(c));    \
     __r;})
```

~~Linux~~
~~mbedtls~~
~~Memcached~~
~~OpenSSL~~
~~Redis~~

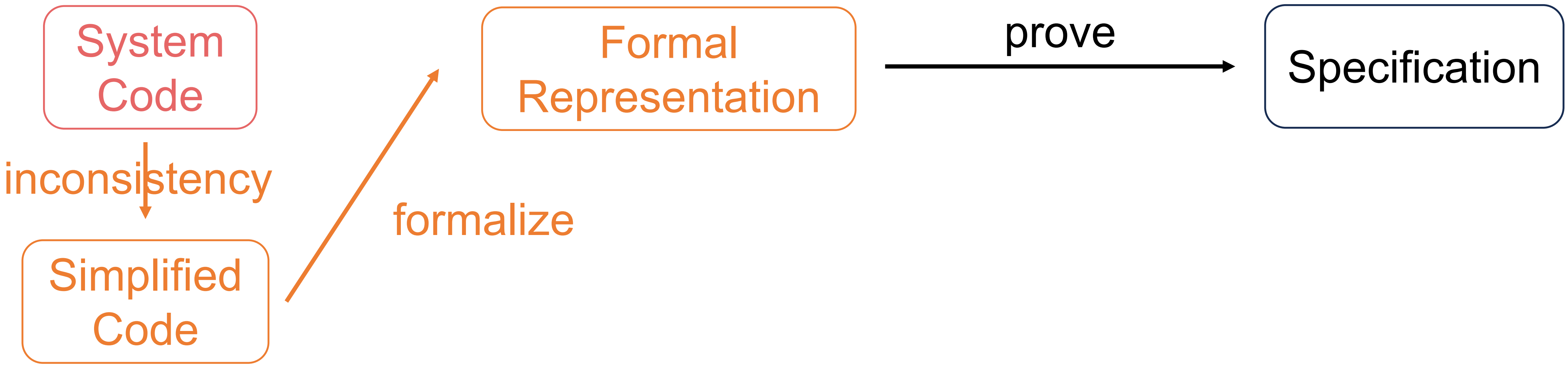
99% of Linux code fails

Challenge 1: Intractable Original Code



Challenge 1: Intractable Original Code

NO Guarantee



System Code

inconsistency

Simplified Code

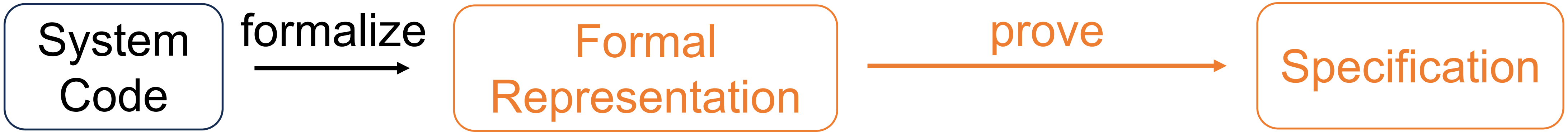
formalize

Formal Representation

prove

Specification

Challenge 2: Huge Proof Effort


System Code

formalize

Formal Representation

prove

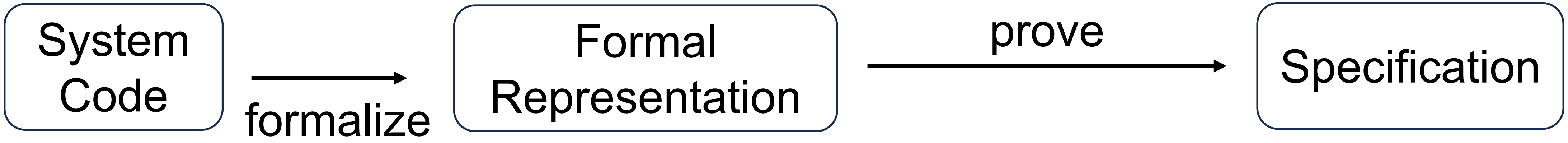
Specification

	Code LOC	Spec & Proof Loc	Spec & Proof / Code
sel4 SOSP’09	8.7K	203K	**23.4**
CertiKOS OSDI’16	6.5K	100K	**15.4**
SeKVM SOSP’21	3.8K	33K	**8.7**
Komodo SOSP’17	2.7K	23K	**8.5**
DaisyNFS OSDI’22	5.7K	46K	**8.0**
VeriBetrKV OSDI’20	6.4K	46K	**7.2**
CCA OSDI’22	3.5K	21K	**6.0**

Spoq -- Scaling Proofs in Coq

- Verify C Systems Code
- Make it easier to use Coq

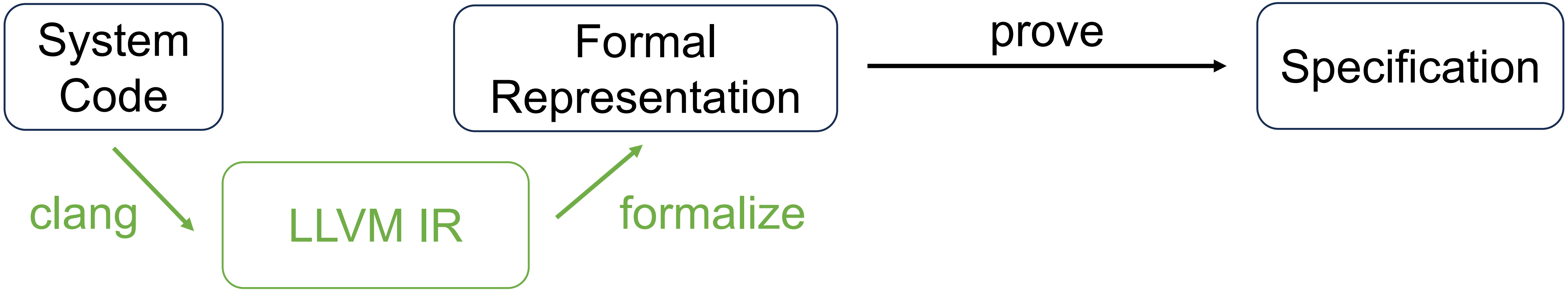
Spoq -- Scaling Proofs in Coq



- Challenge 1: Intractable Original Code

- Challenge 2: Huge Proof Effort

Spoq -- Scaling Proofs in Coq



System Code

clang

LLVM IR

formalize

Formal Representation

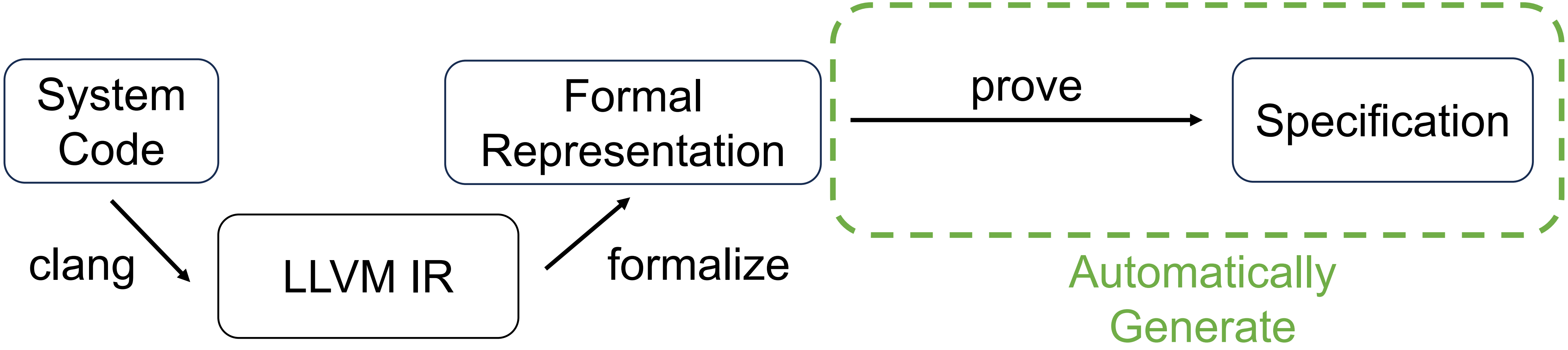
prove

Specification

- Solution 1: Formalize “Intractable” Original Code
  - Rule-based reconstruction algorithm
  - Support 99% of Linux code

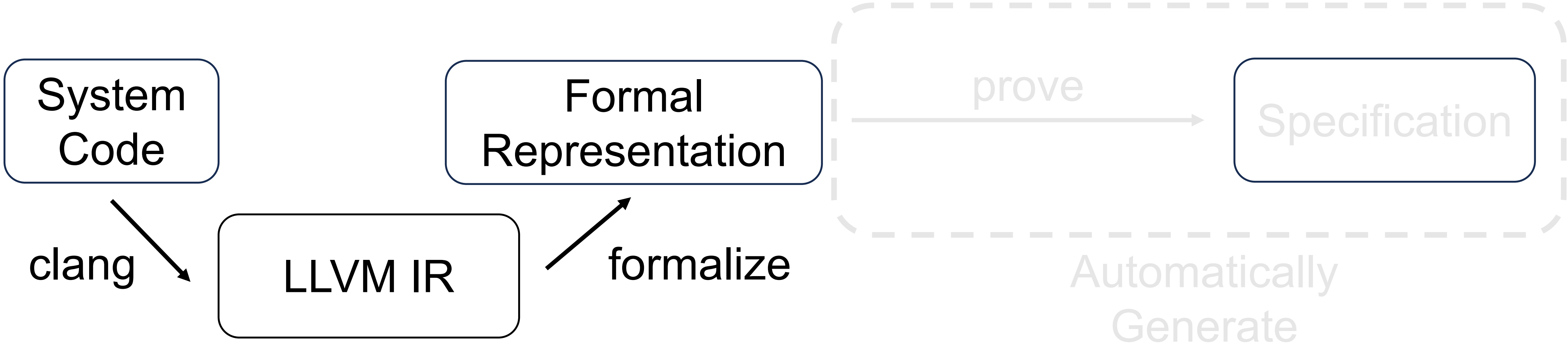
- Challenge 2: Huge Proof Effort

Spoq -- Scaling Proofs in Coq



- Solution 1: Formalize “Intractable” Original Code
  - Rule-based reconstruction algorithm
  - Support 99% of Linux code
- Solution 2: Automate Huge Proof Effort
  - Reduce 80% manual proof effort

Spoq -- Scaling Proofs in Coq



- Formalize “Intractable” Original Code
  - Rule-based reconstruction algorithm
  - Support 99% of Linux code

- Automate Huge Proof Effort
  - Reduce 80% manual proof effort

Formalized LLVM IR

- Compiled from the **original code,** no inconsistency
- **Clean** syntax and semantics

C Code

Implicit type casting
Undefined evaluation order
Macros
GNU C extensions
Compiler Derivatives
……

LLVM IR Code

None of them

Formalized LLVM IR

- **No program structure (i.e. no ifs, no loops), hard to verify**

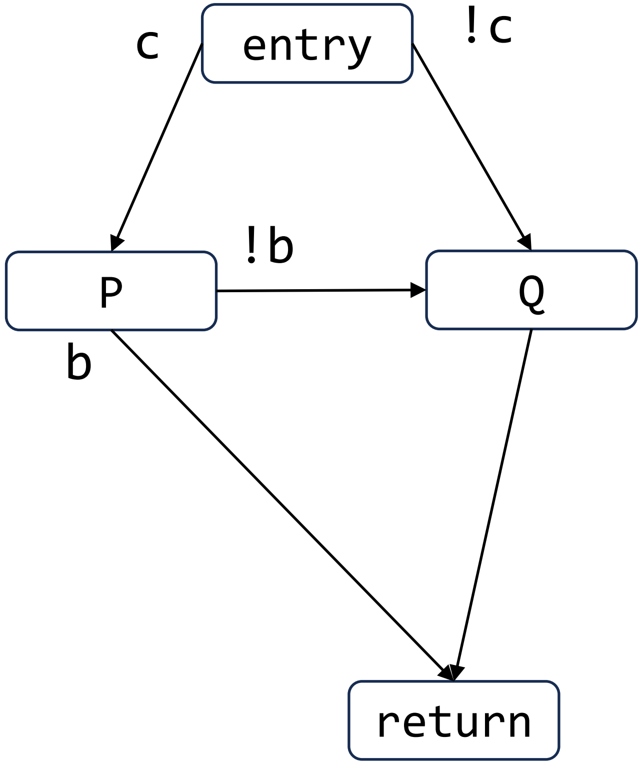
```
entry:
  …
  br %c %P %Q

P:
  …
  br %b %return %Q

Q:
  …
  br %return

return:
  ret
```

Formalized LLVM IR

- **Reconstruct program structure**

```
entry:
  …
  br %c %P %Q

P:
  …
  br %b %return %Q

Q:
  …
  br %return

return:
  ret
```

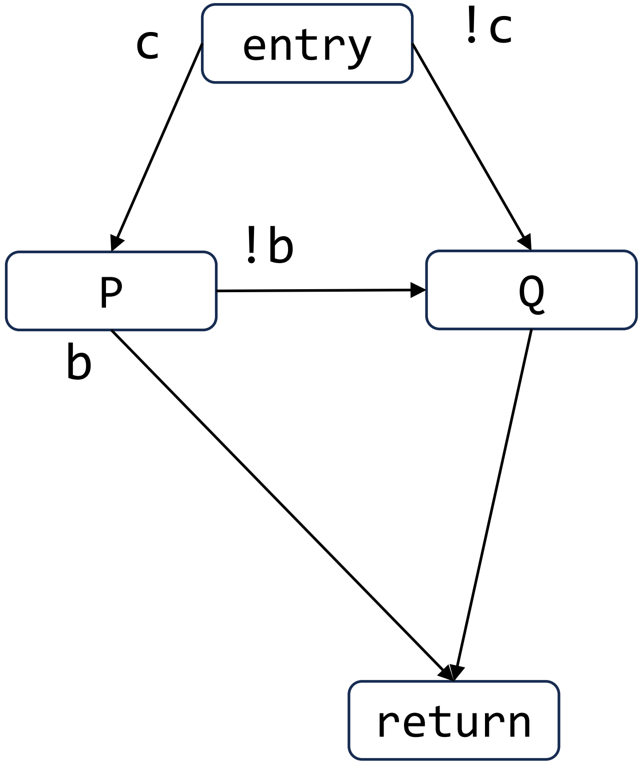
⟹



Formalized LLVM IR

- **Reconstruct program structure**

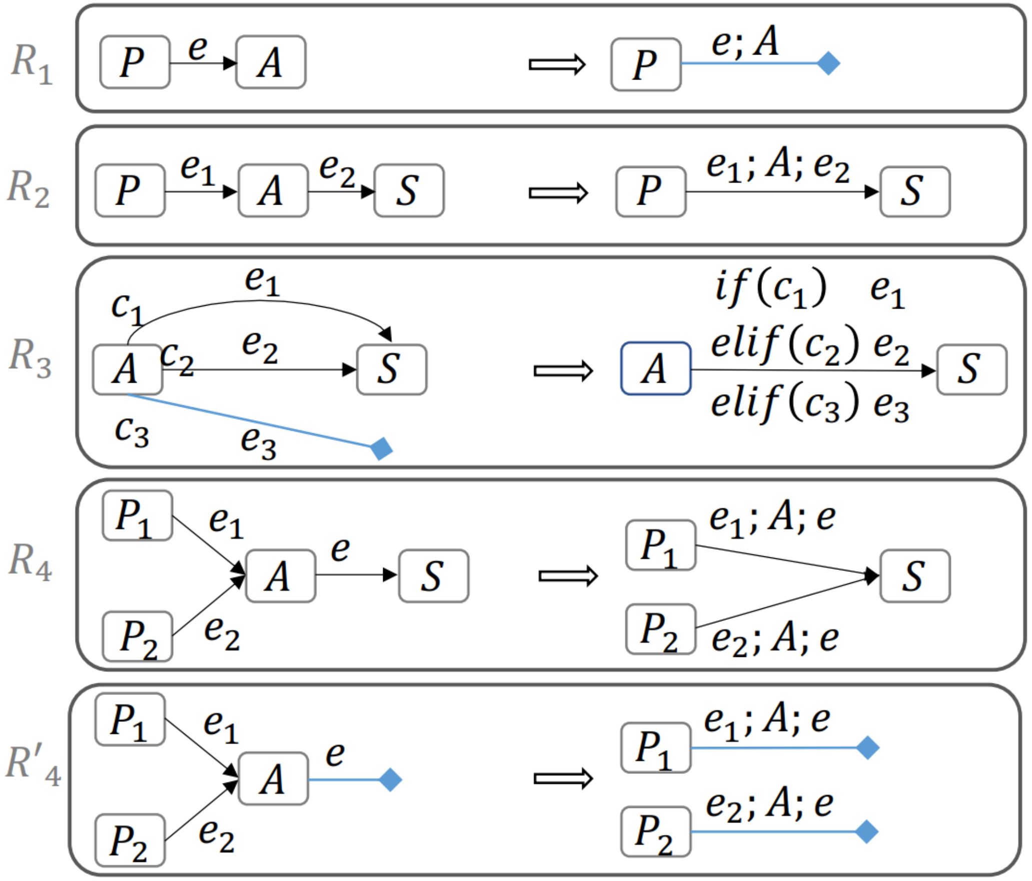
```
entry:
   …
   br %c %P %Q

P:
   …
   br %b %return %Q

Q:
   …
   br %return

return:
   ret
```



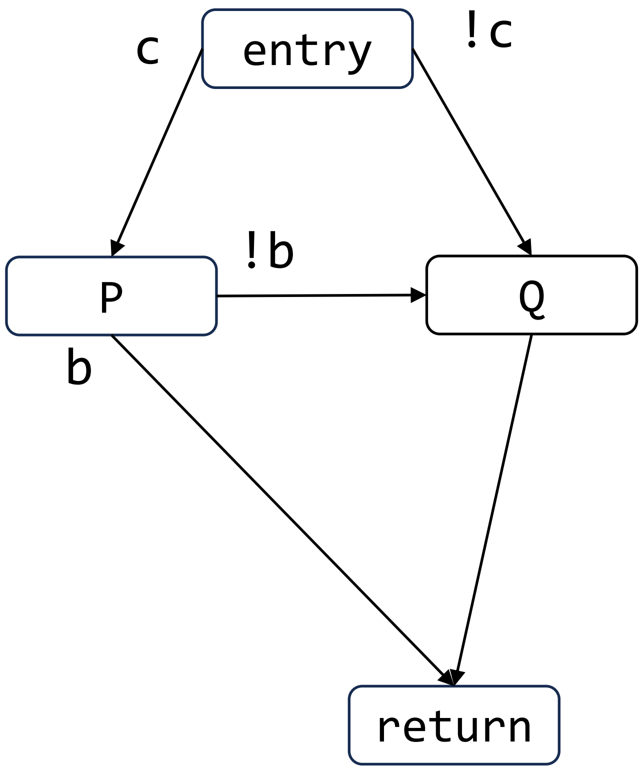
```
entry;
if (c) {
  P;
  if (!b) Q;
}
else Q;
return
```

Program Structure Reconstruction

- **Rule-**based **transformation** algorithm:

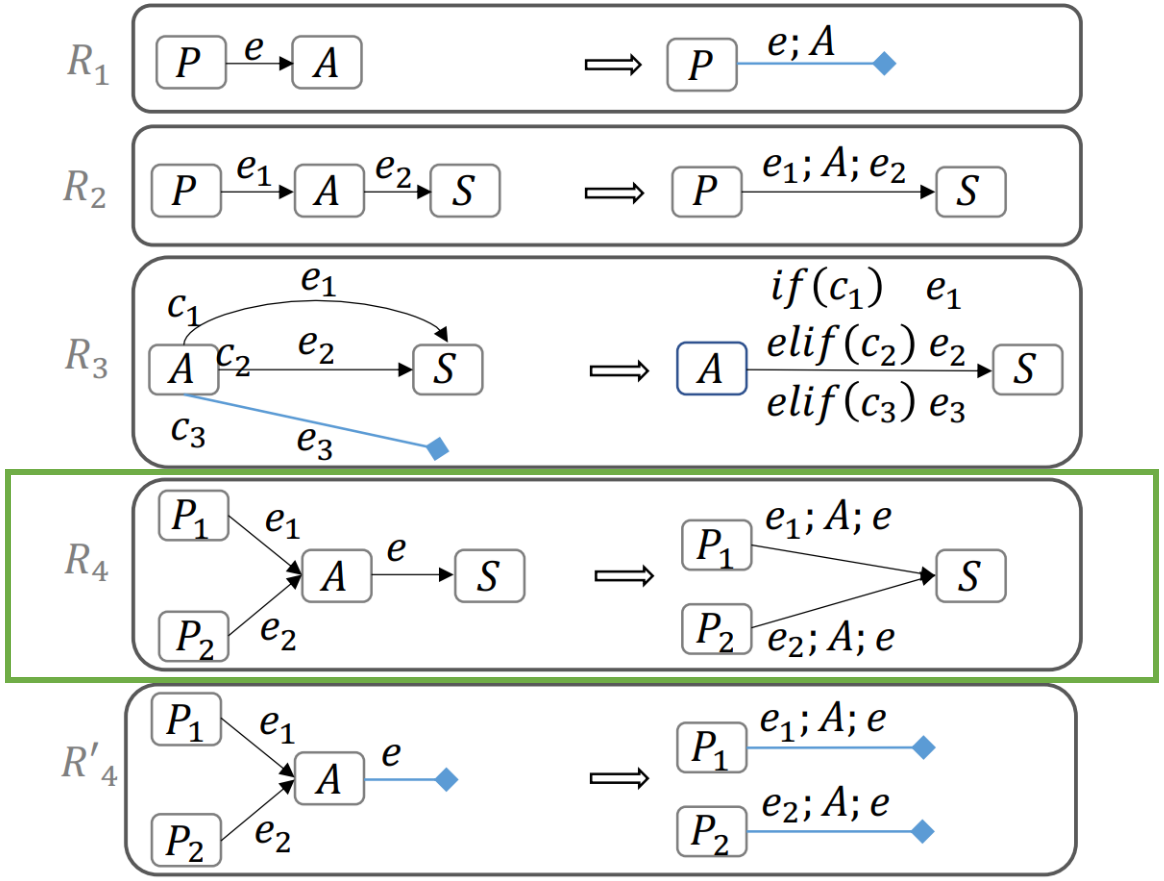


R_1
P e A ⟹ P $e; A$
R_2
P e_1 A e_2 S ⟹ P $e_1; A; e_2$ S
R_3
c_1 e_1 A c_2 e_2 S c_3 e_3 ⟹ A $if(c_1)$ e_1 $elif(c_2)$ e_2 $elif(c_3)$ e_3 S
R_4
P_1 e_1 A e S P_2 e_2 ⟹ P_1 $e_1; A; e$ S P_2 $e_2; A; e$
R'_4
P_1 e_1 A e P_2 e_2 ⟹ P_1 $e_1; A; e$ P_2 $e_2; A; e$

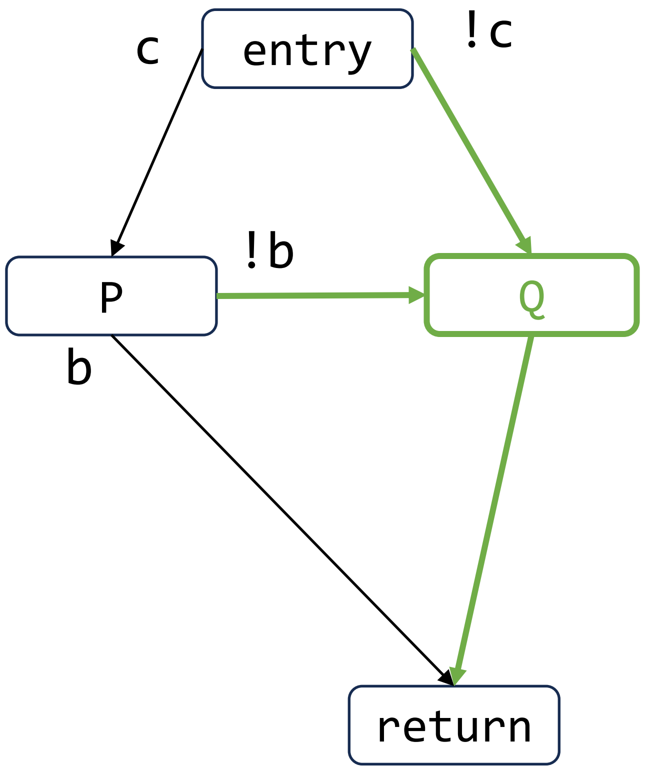


Program Structure Reconstruction

- **Rule-**based **transformation** algorithm:

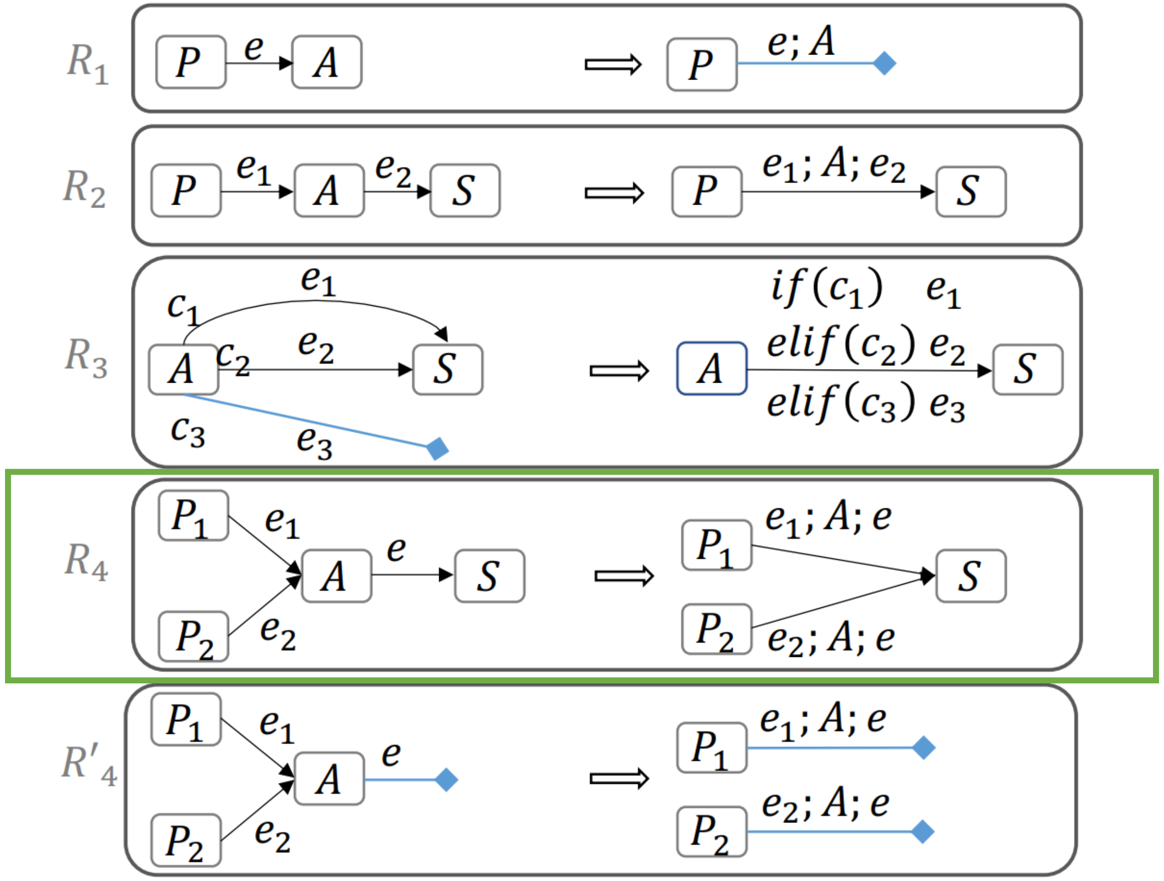


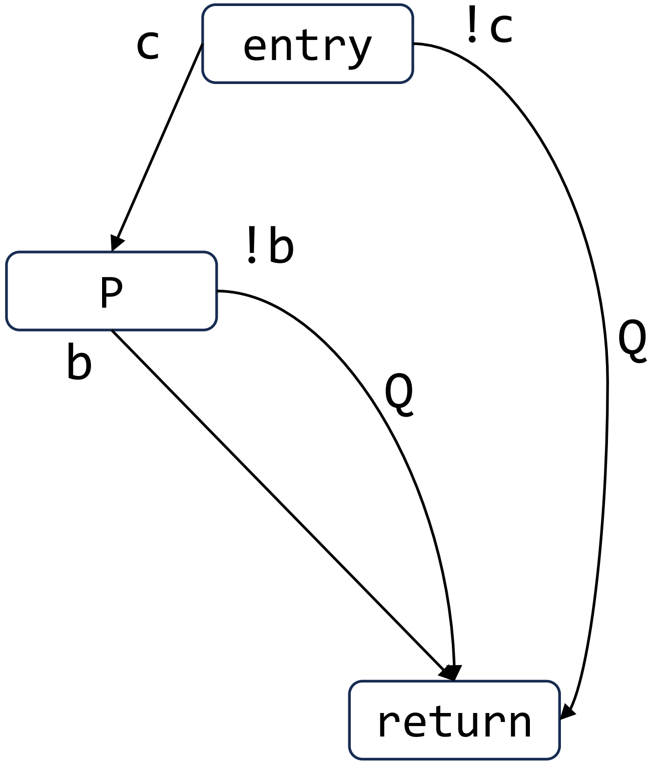
R_1: $P \xrightarrow{e} A \Longrightarrow P \xrightarrow{e; A}$ ◆
R_2: $P \xrightarrow{e_1} A \xrightarrow{e_2} S \Longrightarrow P \xrightarrow{e_1; A; e_2} S$
R_3: A c_1 e_1 c_2 e_2 S c_3 e_3 ◆ $\Longrightarrow$ A $if(c_1)\ e_1$ $elif(c_2)\ e_2$ $elif(c_3)\ e_3$ S
R_4: P_1 e_1 P_2 e_2 $A \xrightarrow{e} S$ $\Longrightarrow$ P_1 $e_1; A; e$ P_2 $e_2; A; e$ S
R'_4: P_1 e_1 P_2 e_2 $A \xrightarrow{e}$ ◆ $\Longrightarrow$ $P_1 \xrightarrow{e_1; A; e}$ ◆ $P_2 \xrightarrow{e_2; A; e}$ ◆



Program Structure Reconstruction

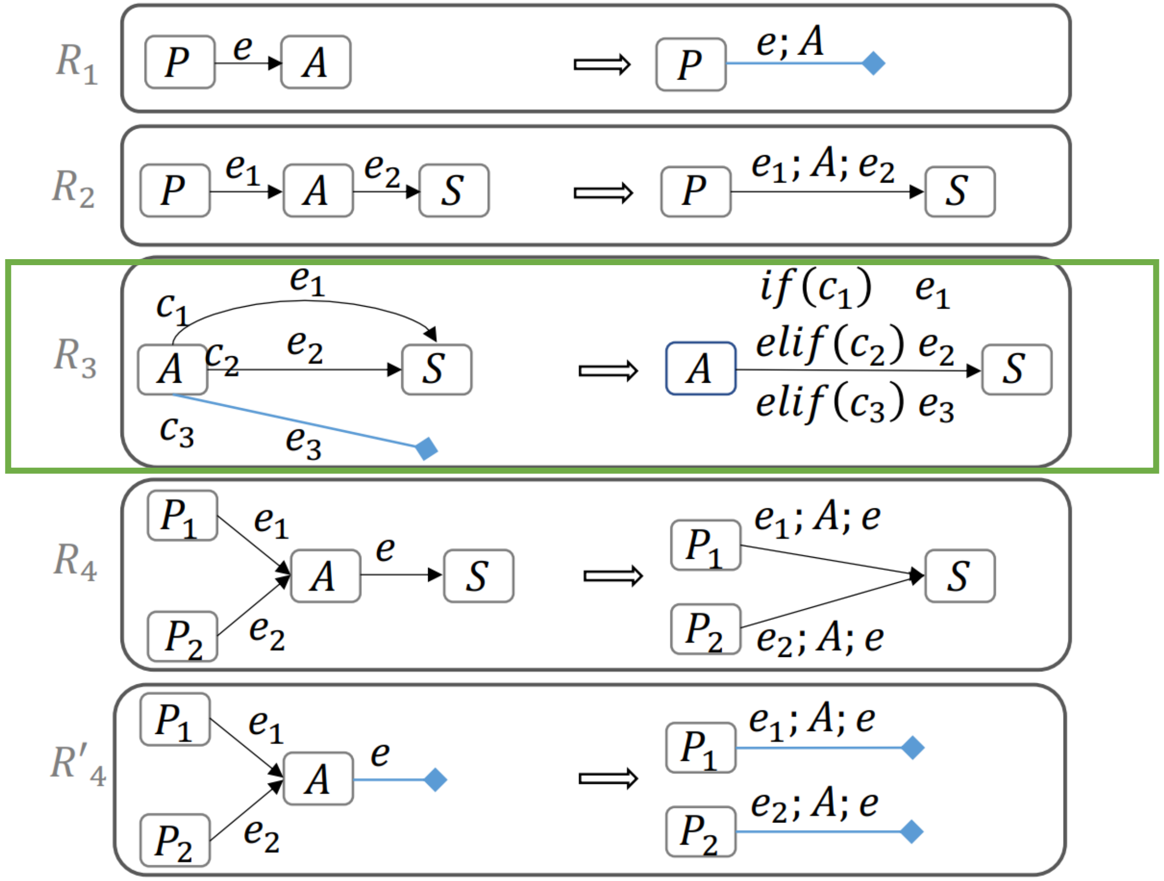
- **Rule-**based **transformation** algorithm:


R_1
$P \xrightarrow{e} A \Longrightarrow P$ $e; A$
R_2
$P \xrightarrow{e_1} A \xrightarrow{e_2} S \Longrightarrow P \xrightarrow{e_1; A; e_2} S$
R_3
c_1 e_1
A c_2 e_2 S
c_3 e_3
$\Longrightarrow A$ $if(c_1)\ e_1$ $elif(c_2)\ e_2$ $elif(c_3)\ e_3$ S
R_4
P_1 e_1 A e S P_2 e_2
$\Longrightarrow P_1$ $e_1; A; e$ S P_2 $e_2; A; e$
R'_4
P_1 e_1 A e P_2 e_2
$\Longrightarrow P_1$ $e_1; A; e$ P_2 $e_2; A; e$

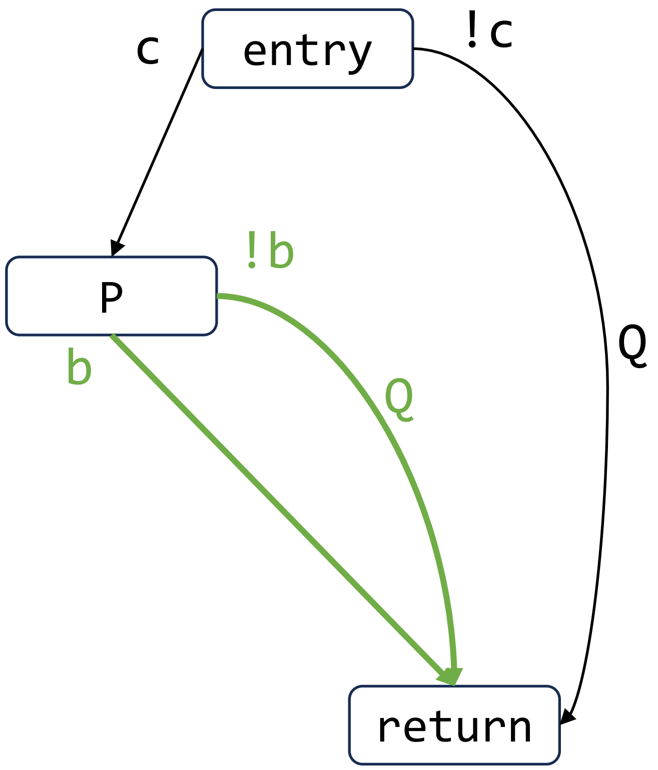


Program Structure Reconstruction

- **Rule-**based **transformation** algorithm:

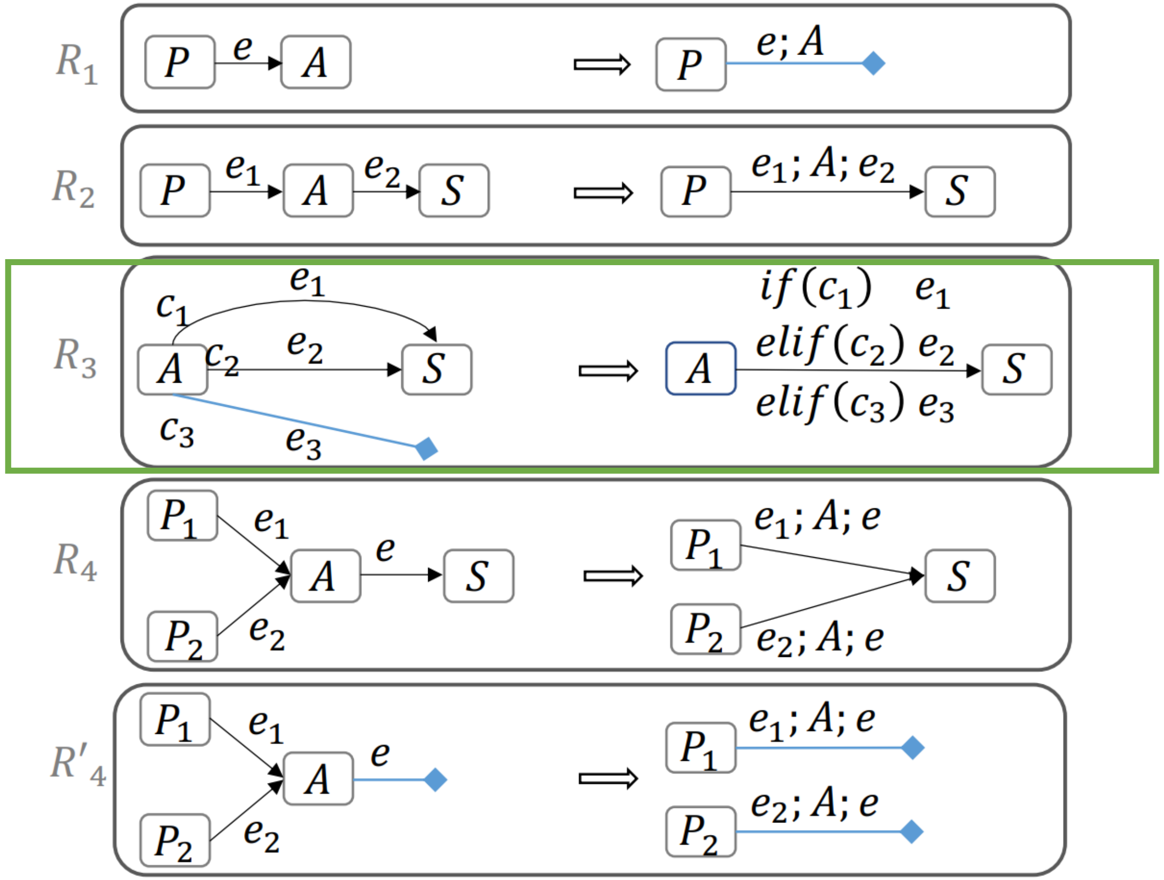


R_1
$P \xrightarrow{e} A \Longrightarrow P \xrightarrow{e; A}$
R_2
$P \xrightarrow{e_1} A \xrightarrow{e_2} S \Longrightarrow P \xrightarrow{e_1; A; e_2} S$
R_3
c_1 e_1 c_2 e_2 c_3 e_3 A S
$\Longrightarrow$
A $if(c_1)\ e_1$ $elif(c_2)\ e_2$ $elif(c_3)\ e_3$ S
R_4
P_1 e_1 P_2 e_2 A e S
$\Longrightarrow$
P_1 $e_1; A; e$ P_2 $e_2; A; e$ S
R'_4
P_1 e_1 P_2 e_2 A e
$\Longrightarrow$
P_1 $e_1; A; e$ P_2 $e_2; A; e$

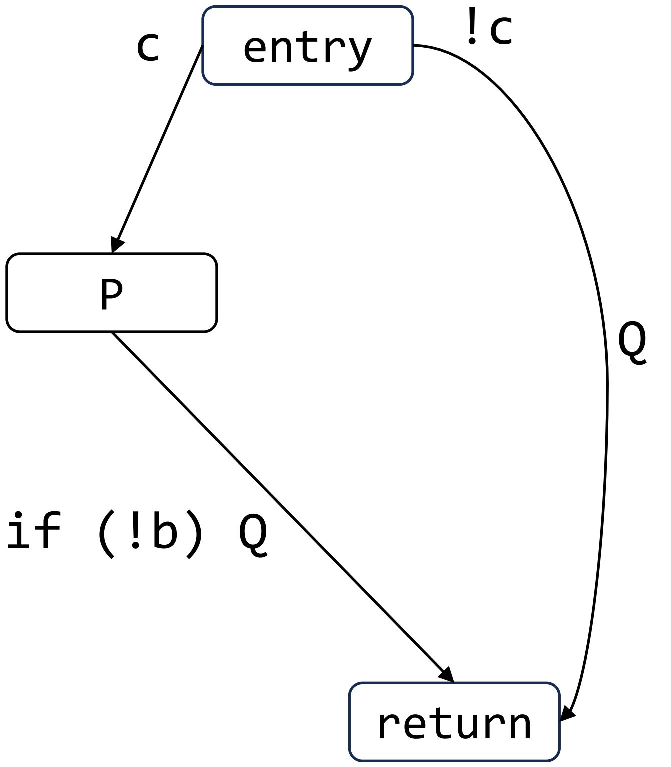


Program Structure Reconstruction

- **Rule-**based **transformation** algorithm:

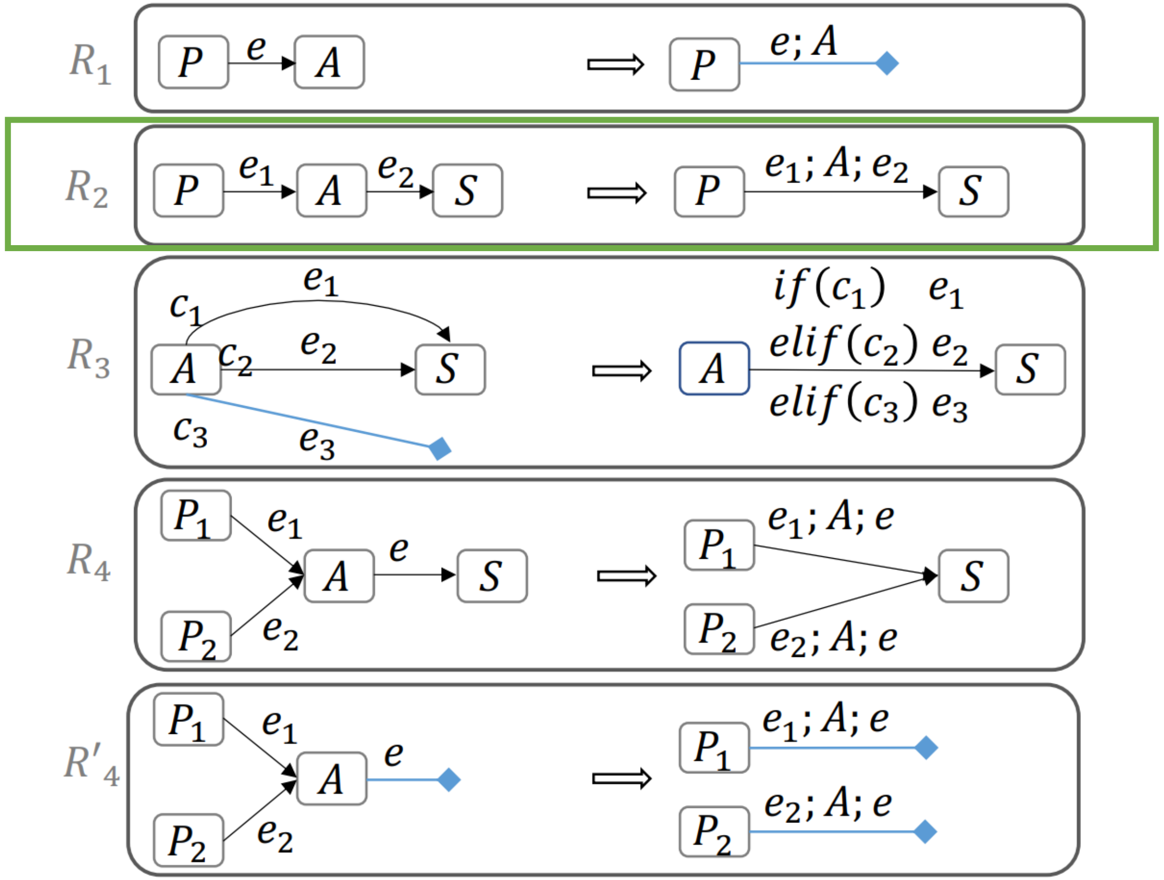


R_1
$P \xrightarrow{e} A \Longrightarrow P \xrightarrow{e; A}$
R_2
$P \xrightarrow{e_1} A \xrightarrow{e_2} S \Longrightarrow P \xrightarrow{e_1; A; e_2} S$
R_3
A c_1 e_1 c_2 e_2 c_3 e_3 S $\Longrightarrow$ A $if(c_1)\ e_1$ $elif(c_2)\ e_2$ $elif(c_3)\ e_3$ S
R_4
P_1 e_1 P_2 e_2 A e S $\Longrightarrow$ P_1 $e_1; A; e$ P_2 $e_2; A; e$ S
R'_4
P_1 e_1 P_2 e_2 A e $\Longrightarrow$ P_1 $e_1; A; e$ P_2 $e_2; A; e$

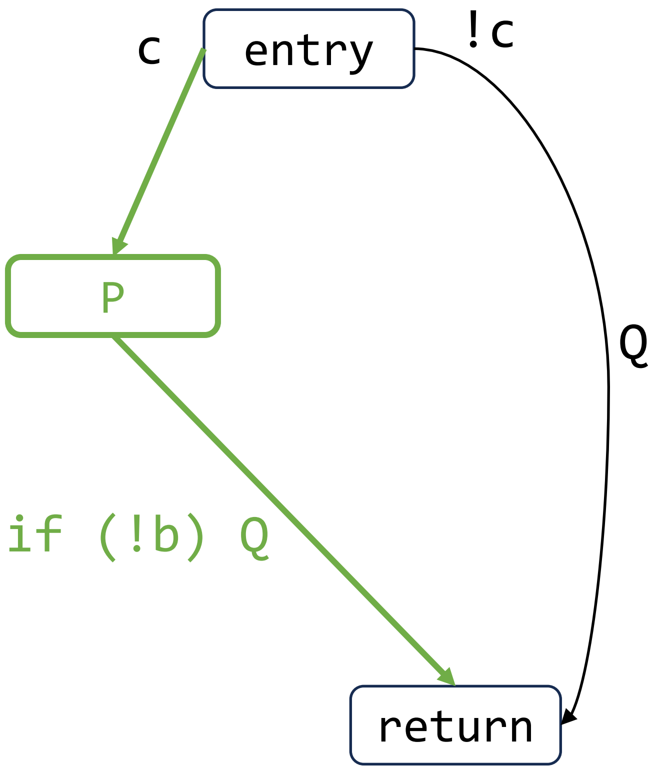


Program Structure Reconstruction

- **Rule-**based **transformation** algorithm:

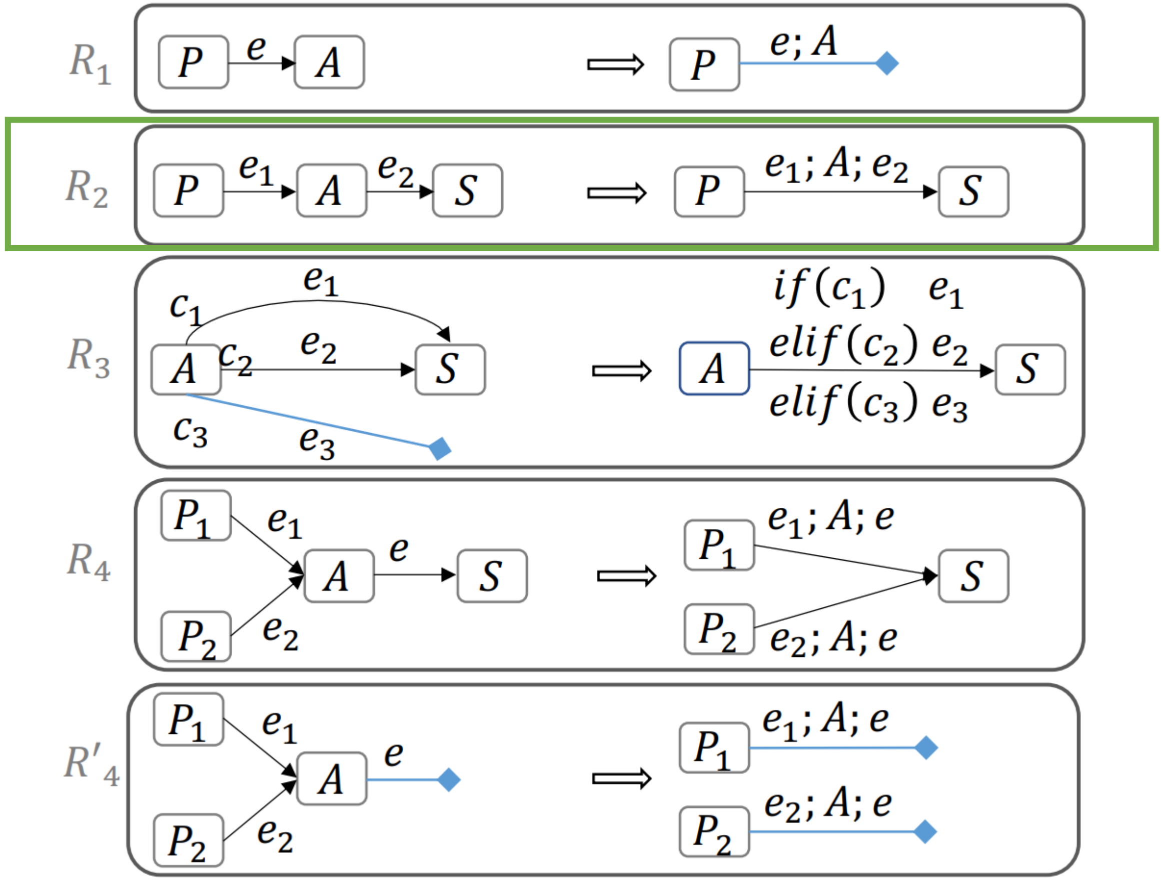


R_1
$P \xrightarrow{e} A \implies P \xrightarrow{e; A}$
R_2
$P \xrightarrow{e_1} A \xrightarrow{e_2} S \implies P \xrightarrow{e_1; A; e_2} S$
R_3
c_1 e_1 c_2 e_2 c_3 e_3 A S $\implies$ A $if(c_1)\ e_1$ $elif(c_2)\ e_2$ $elif(c_3)\ e_3$ S
R_4
P_1 e_1 P_2 e_2 A e S $\implies$ P_1 $e_1; A; e$ P_2 $e_2; A; e$ S
R'_4
P_1 e_1 P_2 e_2 A e $\implies$ P_1 $e_1; A; e$ P_2 $e_2; A; e$

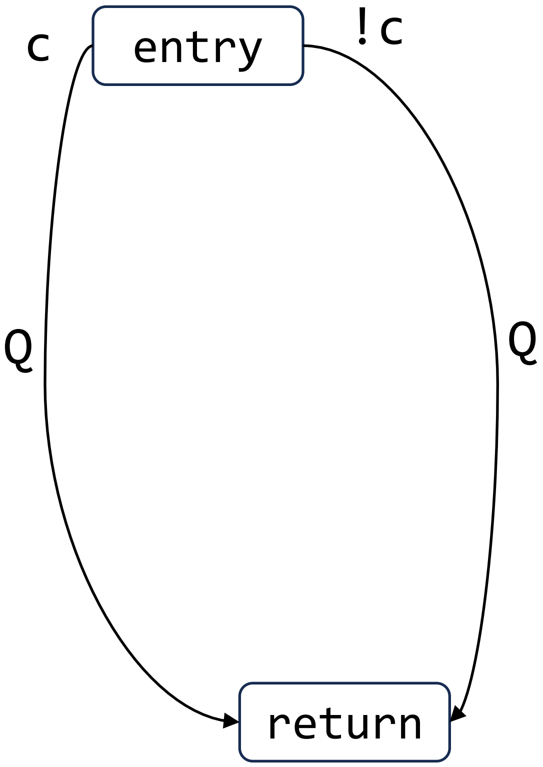


Program Structure Reconstruction

- **Rule-**based **transformation** algorithm:

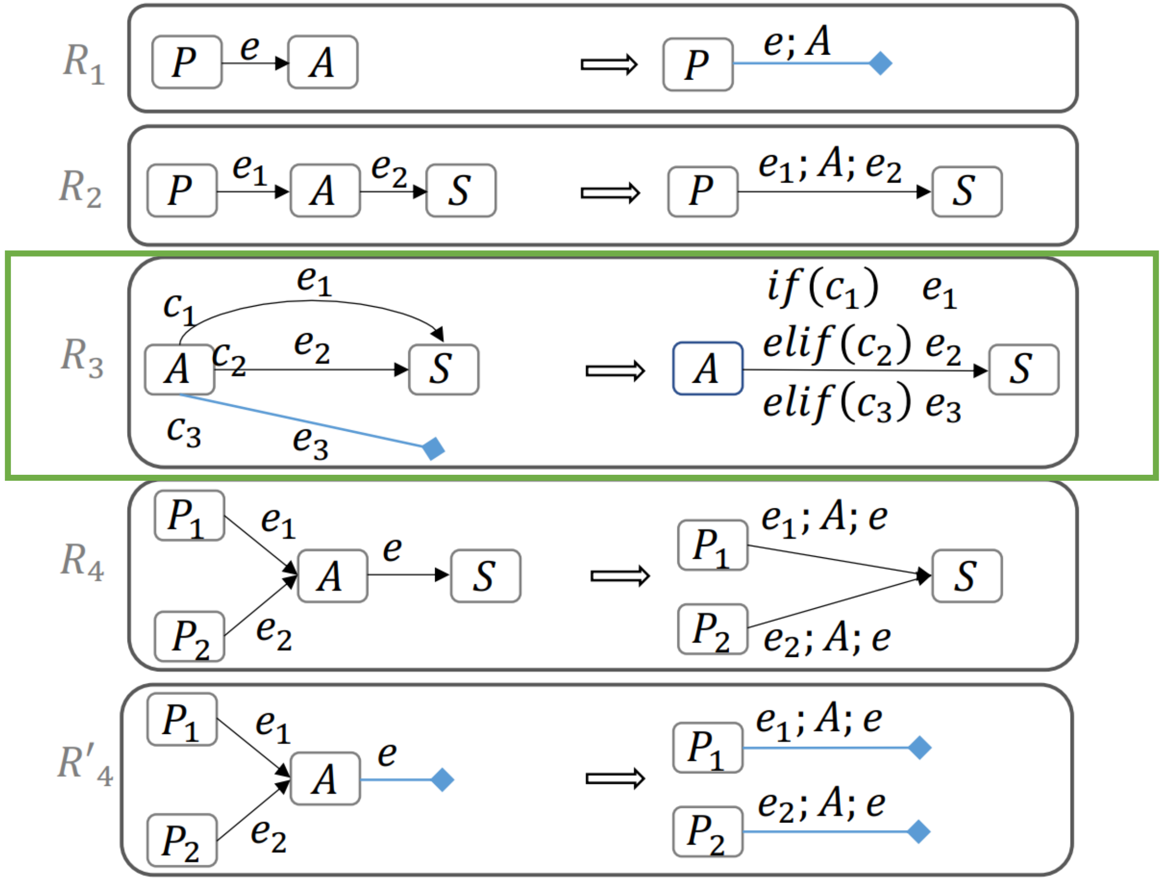

R_1
$P \xrightarrow{e} A \Longrightarrow P \xrightarrow{e; A}$
R_2
$P \xrightarrow{e_1} A \xrightarrow{e_2} S \Longrightarrow P \xrightarrow{e_1; A; e_2} S$
R_3
c_1 e_1 c_2 e_2 c_3 e_3 A S
$\Longrightarrow$
A $if(c_1)\ e_1$ $elif(c_2)\ e_2$ $elif(c_3)\ e_3$ S
R_4
P_1 e_1 P_2 e_2 A e S
$\Longrightarrow$
P_1 $e_1; A; e$ P_2 $e_2; A; e$ S
R'_4
P_1 e_1 P_2 e_2 A e
$\Longrightarrow$
P_1 $e_1; A; e$ P_2 $e_2; A; e$

```
P;
if (!b) Q
```



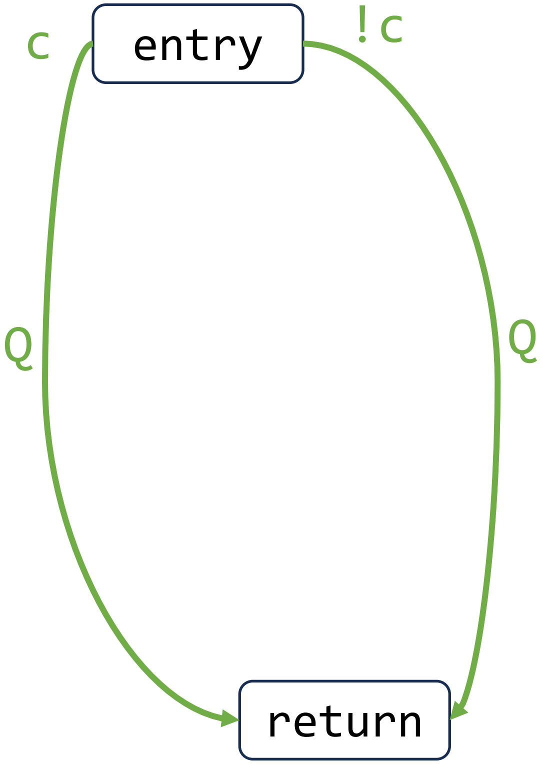
Program Structure Reconstruction

- **Rule-**based **transformation** algorithm:



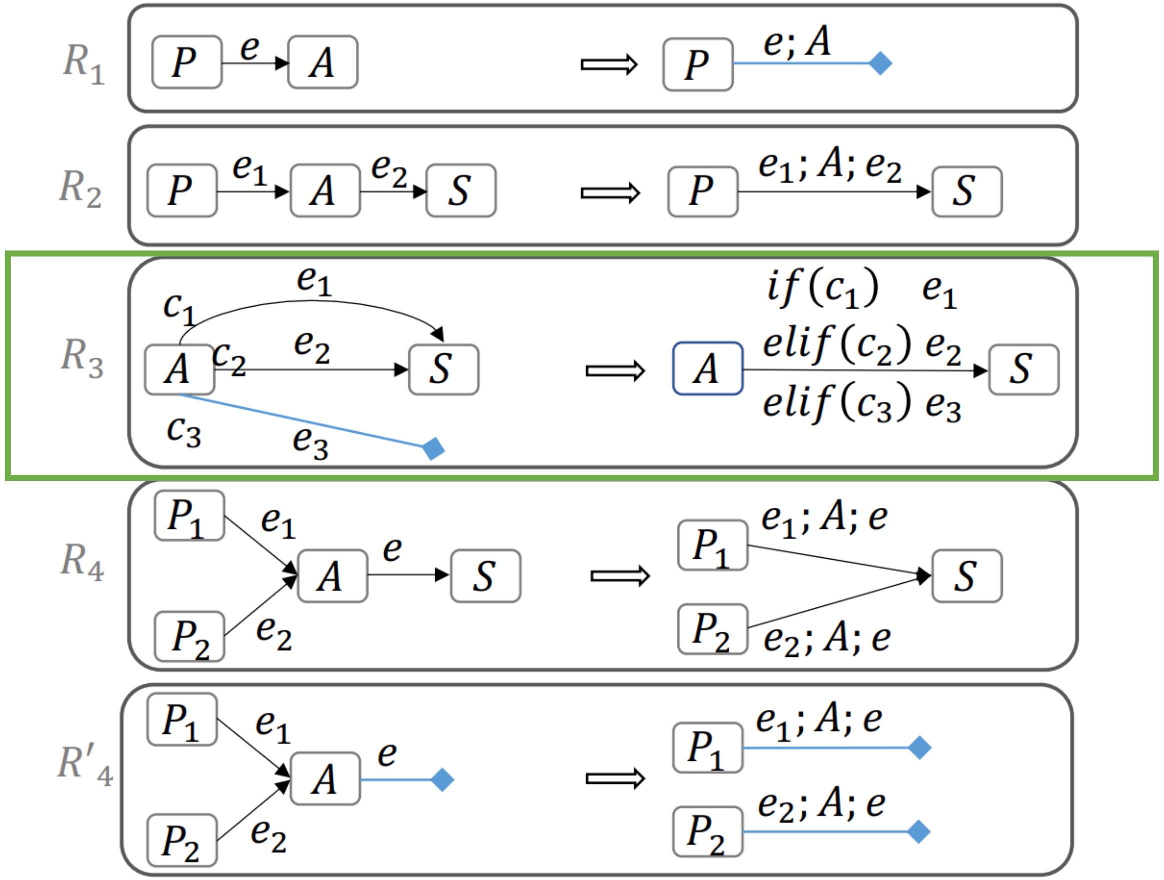
R_1
$P \xrightarrow{e} A \Longrightarrow P \xrightarrow{e; A}$
R_2
$P \xrightarrow{e_1} A \xrightarrow{e_2} S \Longrightarrow P \xrightarrow{e_1; A; e_2} S$
R_3
A c_1 e_1 c_2 e_2 S c_3 e_3
$\Longrightarrow A$ $if(c_1)\ e_1$ $elif(c_2)\ e_2$ $elif(c_3)\ e_3$ S
R_4
$P_1 \xrightarrow{e_1} A$, $P_2 \xrightarrow{e_2} A$, $A \xrightarrow{e} S$
$\Longrightarrow P_1 \xrightarrow{e_1; A; e} S$, $P_2 \xrightarrow{e_2; A; e} S$
R'_4
$P_1 \xrightarrow{e_1} A$, $P_2 \xrightarrow{e_2} A$, $A \xrightarrow{e}$
$\Longrightarrow P_1 \xrightarrow{e_1; A; e}$, $P_2 \xrightarrow{e_2; A; e}$

```
P;
if (!b) Q
```



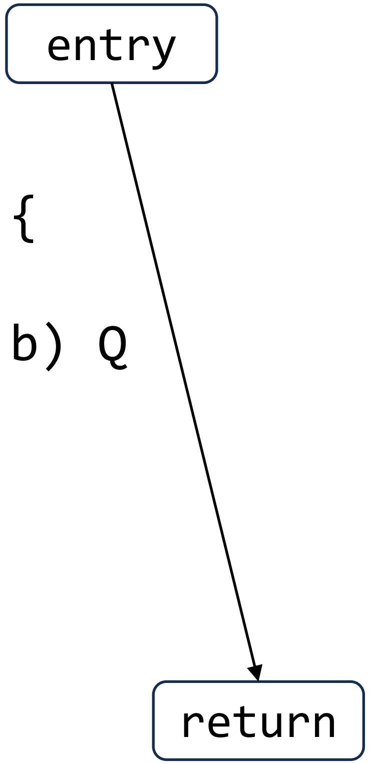
Program Structure Reconstruction

- **Rule-**based **transformation** algorithm:



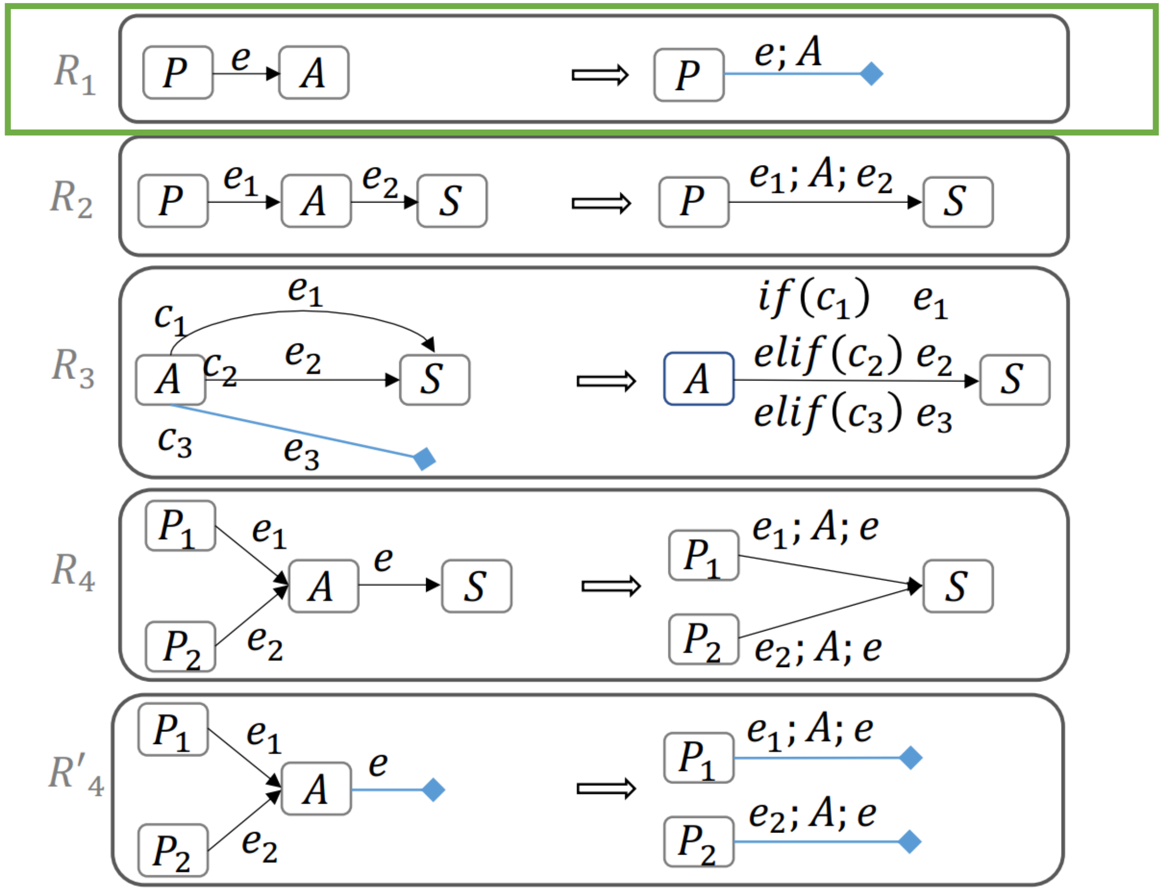
R_1
$P \xrightarrow{e} A \Longrightarrow P \xrightarrow{e; A}$
R_2
$P \xrightarrow{e_1} A \xrightarrow{e_2} S \Longrightarrow P \xrightarrow{e_1; A; e_2} S$
R_3
c_1 e_1 c_2 e_2 c_3 e_3
A S
$\Longrightarrow$
$if(c_1)\ e_1$
$elif(c_2)\ e_2$
$elif(c_3)\ e_3$
A S
R_4
P_1 e_1 P_2 e_2 A e S
$\Longrightarrow$
P_1 $e_1; A; e$ P_2 $e_2; A; e$ S
R'_4
P_1 e_1 P_2 e_2 A e
$\Longrightarrow$
P_1 $e_1; A; e$
P_2 $e_2; A; e$

```
if (c) {
  P;
  if (!b) Q
}
else Q
```

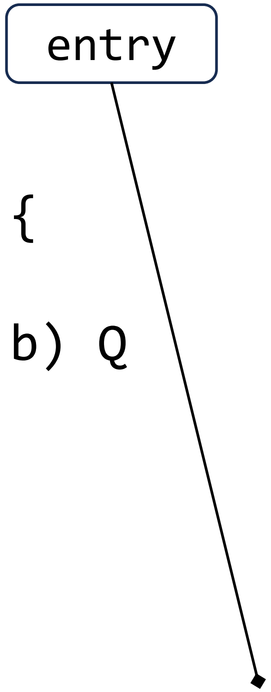


Program Structure Reconstruction

- **Rule-**based **transformation** algorithm:

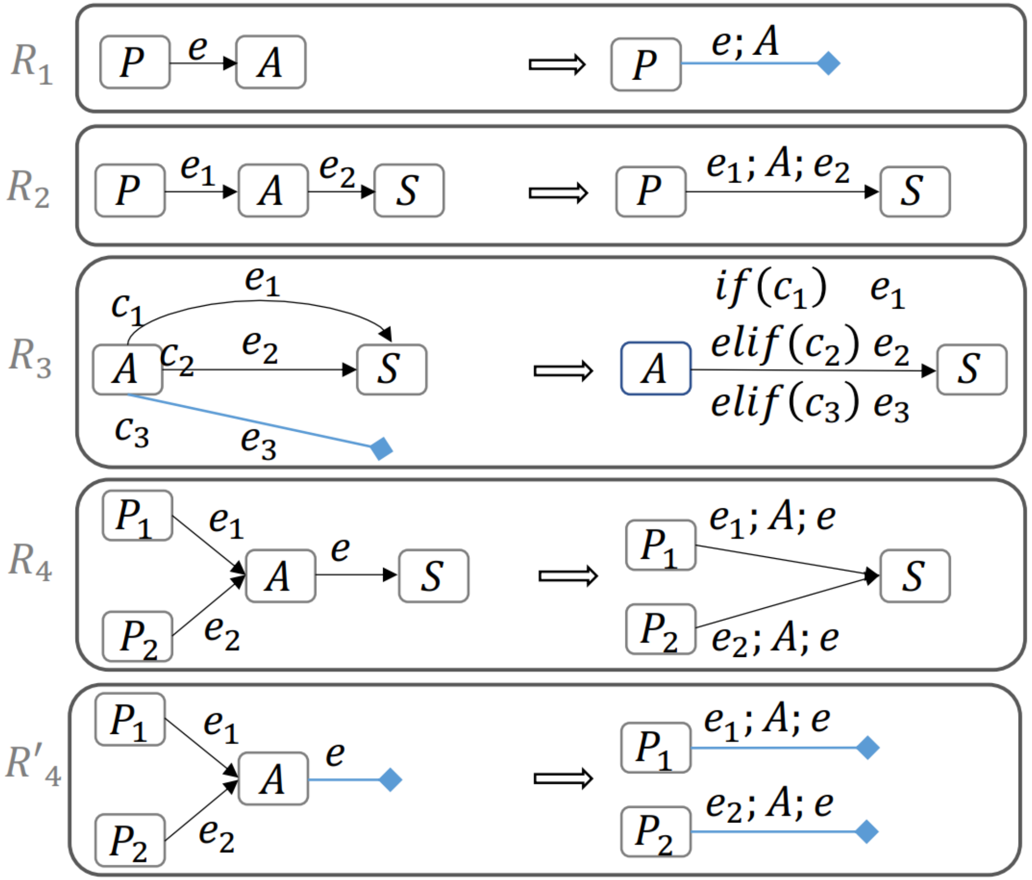

R_1
$P \xrightarrow{e} A \Longrightarrow P \xrightarrow{e; A}$
R_2
$P \xrightarrow{e_1} A \xrightarrow{e_2} S \Longrightarrow P \xrightarrow{e_1; A; e_2} S$
R_3
c_1 e_1 A c_2 e_2 S c_3 e_3
$\Longrightarrow$ A $if(c_1)$ e_1 $elif(c_2)$ e_2 $elif(c_3)$ e_3 S
R_4
P_1 e_1 A e S P_2 e_2
$\Longrightarrow$ P_1 $e_1; A; e$ S P_2 $e_2; A; e$
R'_4
P_1 e_1 A e P_2 e_2
$\Longrightarrow$ P_1 $e_1; A; e$ P_2 $e_2; A; e$

```
if (c) {
  P;
  if (!b) Q
}
else Q;
return
```



Program Structure Reconstruction

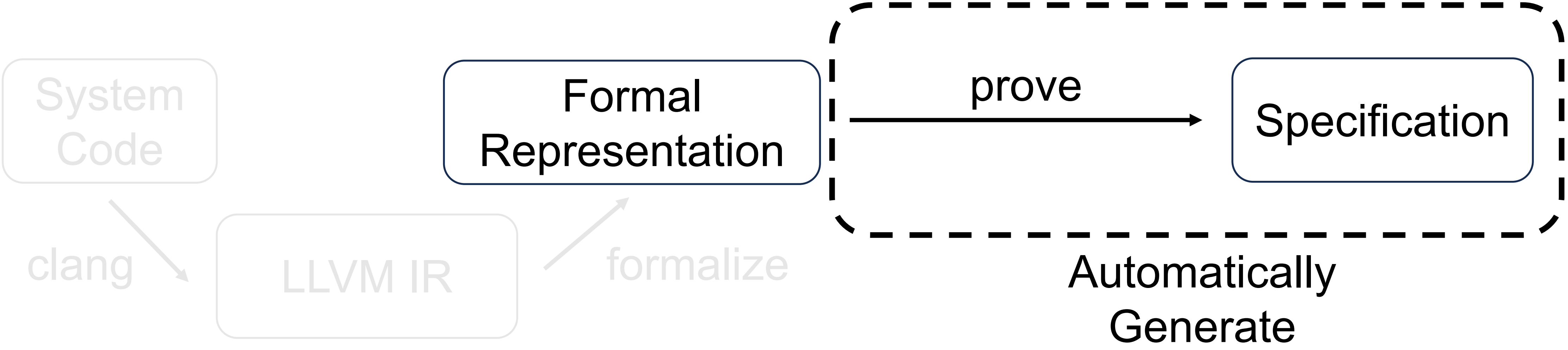
- **Rule-**based **transformation** algorithm:



R_1
$P \xrightarrow{e} A \implies P \xrightarrow{e; A}$
R_2
$P \xrightarrow{e_1} A \xrightarrow{e_2} S \implies P \xrightarrow{e_1; A; e_2} S$
R_3
A, c_1, e_1, c_2, e_2, S, c_3, e_3
$\implies A$ $if(c_1)$ e_1 $elif(c_2)$ e_2 $elif(c_3)$ e_3 S
R_4
$P_1 \xrightarrow{e_1} A$, $P_2 \xrightarrow{e_2} A$, $A \xrightarrow{e} S$
$\implies P_1 \xrightarrow{e_1; A; e} S$, $P_2 \xrightarrow{e_2; A; e} S$
R'_4
$P_1 \xrightarrow{e_1} A$, $P_2 \xrightarrow{e_2} A$, $A \xrightarrow{e}$
$\implies P_1 \xrightarrow{e_1; A; e}$, $P_2 \xrightarrow{e_2; A; e}$

```
entry;
if (c) {
  P;
  if (!b) Q
}
else Q;
return
```

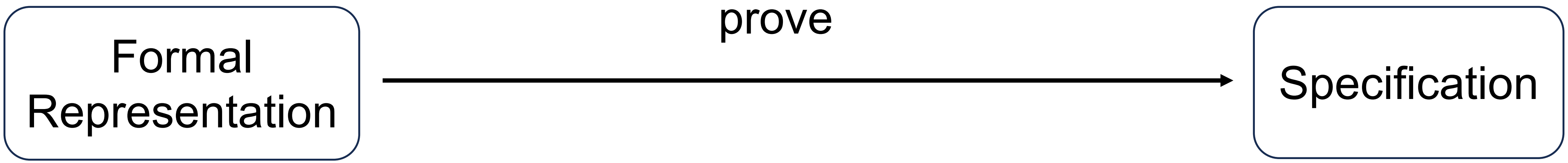
Spoq -- Scaling Proofs in Coq



- Formalize “Intractable” Original Code
  - Rule-based reconstruction algorithm
  - Support 99% of Linux code

- Automate Huge Proof Effort
  - Reduce 80% manual proof effort

Automate Spec Def & Proof

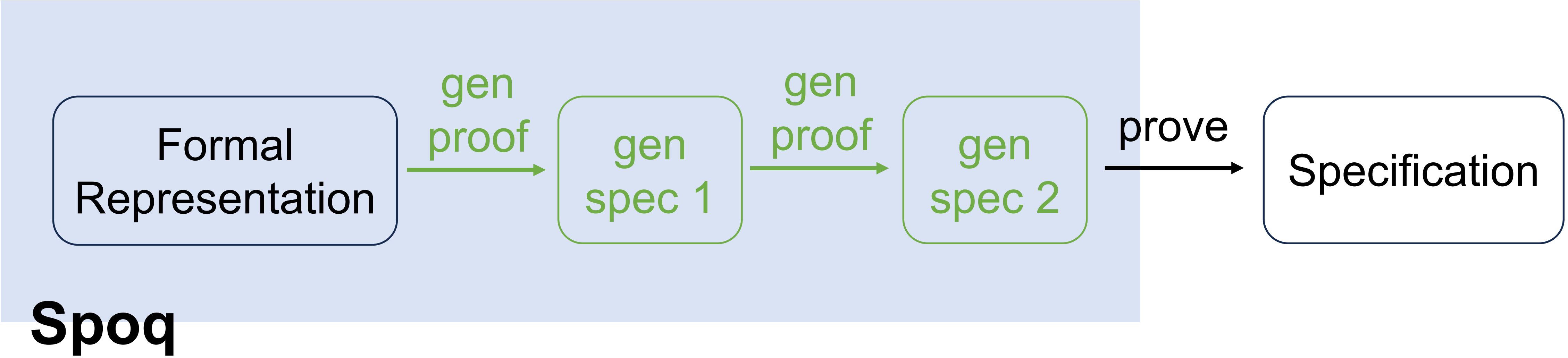


Formal Representation

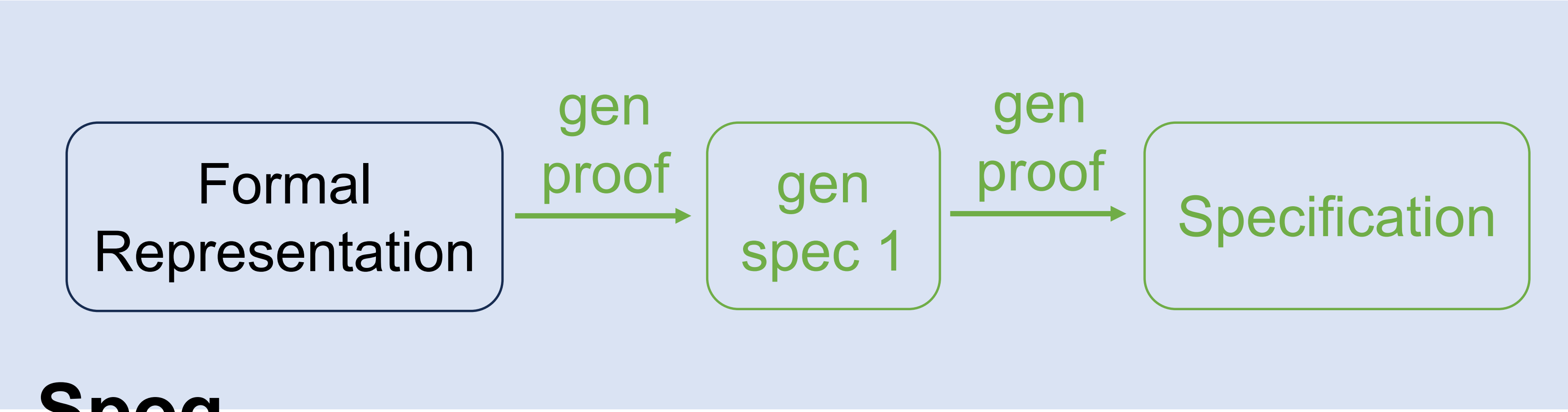
prove

Specification

Automate Spec Def & Proof



Automate Spec Def & Proof



Spoq

Synthesize the first intermediate spec

C code

```
if (x < y) {
  x = x + y;
  y = x - y;
  x = x - y;
}
else {
  x++; y--;
}
return x + y;
```

⇒

Formal Repr

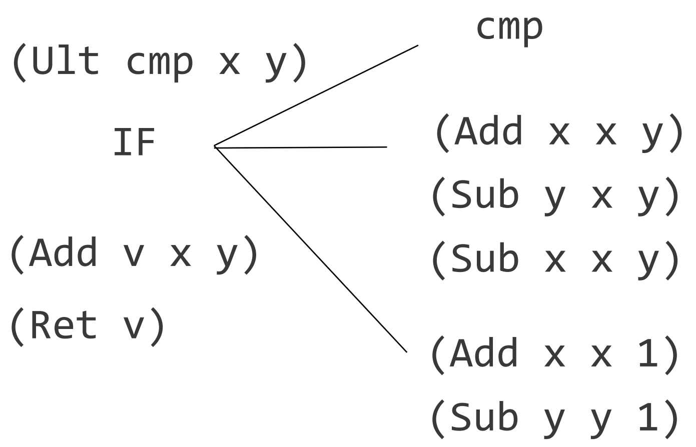
```
(Seq
  (Ult cmp x y)
  (If cmp
    (Seq (Add x x y)
         (Sub y x y)
         (Sub x x y))
    (Seq (Add x x 1)
         (Sub y y 1)))
    (Add v x y)
    (Ret v))
```

Synthesize the first intermediate spec

C code

```
if (x < y) {
  x = x + y;
  y = x - y;
  x = x - y;
}
else {
  x++; y--;
}
return x + y;
```

Formal Repr



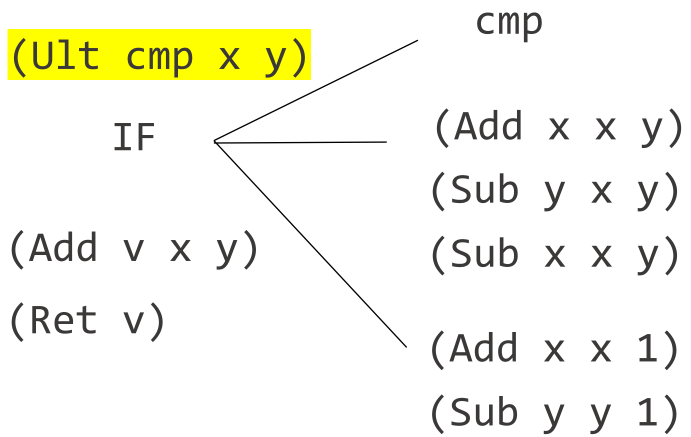
```
(Ult cmp x y)
IF
  cmp
  (Add x x y)
  (Sub y x y)
  (Sub x x y)
  (Add x x 1)
  (Sub y y 1)
(Add v x y)
(Ret v)
```

Synthesize the first intermediate spec

C code

```
if (x < y) {
  x = x + y;
  y = x - y;
  x = x - y;
}
else {
  x++; y--;
}
return x + y;
```

Formal Repr



```
(Ult cmp x y)
IF
(Add v x y)
(Ret v)
cmp
(Add x x y)
(Sub y x y)
(Sub x x y)
(Add x x 1)
(Sub y y 1)
```

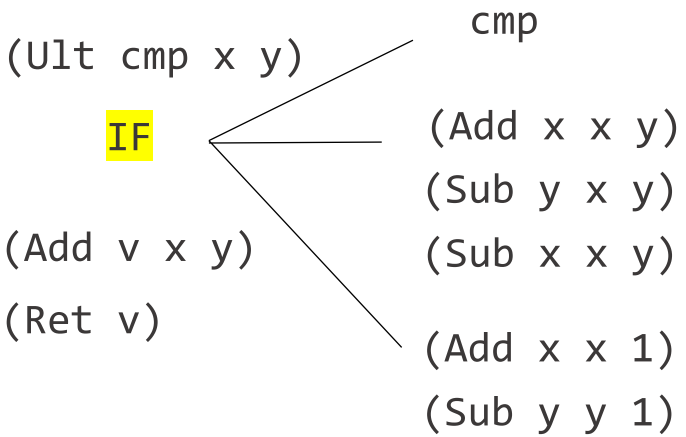
```
let cmp := x <? y in
```

Synthesize the first intermediate spec

C code

```
if (x < y) {
  x = x + y;
  y = x - y;
  x = x - y;
}
else {
  x++; y--;
}
return x + y;
```

Formal Repr



```
(Ult cmp x y)
     IF
(Add v x y)
(Ret v)
```

```
cmp

(Add x x y)
(Sub y x y)
(Sub x x y)

(Add x x 1)
(Sub y y 1)
```

```
let cmp := x <? y in
let (x, y) :=
  if  ???  then
    (x,
     y)
  else
    (x, y)
in
```

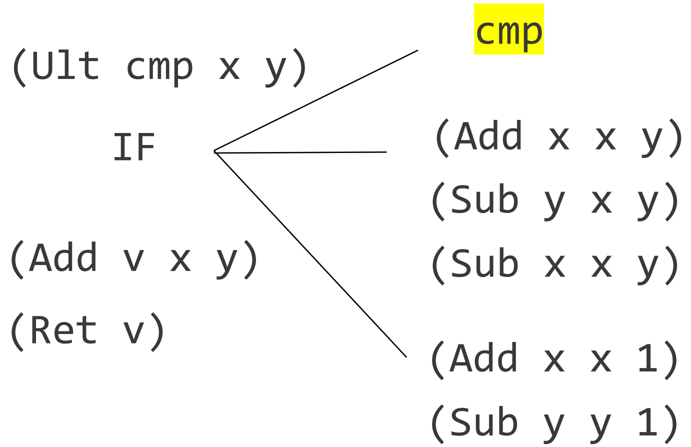
Synthesize the first intermediate spec

C code

```
if (x < y) {
  x = x + y;
  y = x - y;
  x = x - y;
}
else {
  x++; y--;
}
return x + y;
```

⟹

Formal Repr



```
(Ult cmp x y)
     IF          cmp
                 (Add x x y)
                 (Sub y x y)
                 (Sub x x y)

                 (Add x x 1)
                 (Sub y y 1)
(Add v x y)
(Ret v)
```

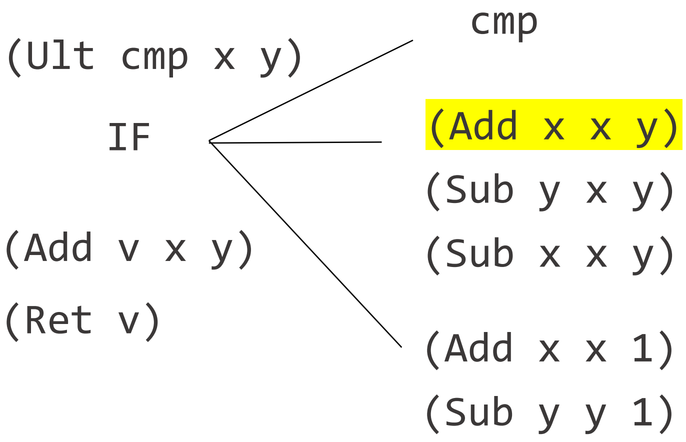
```
let cmp := x <? y in
let (x, y) :=
  if  cmp  then
    (x,
     y)
  else
    (x, y)
in
```

Synthesize the first intermediate spec

C code

```
if (x < y) {
  x = x + y;
  y = x - y;
  x = x - y;
}
else {
  x++; y--;
}
return x + y;
```

Formal Repr



```
(Ult cmp x y)
     IF
(Add v x y)
(Ret v)
```

```
cmp
(Add x x y)
(Sub y x y)
(Sub x x y)
(Add x x 1)
(Sub y y 1)
```

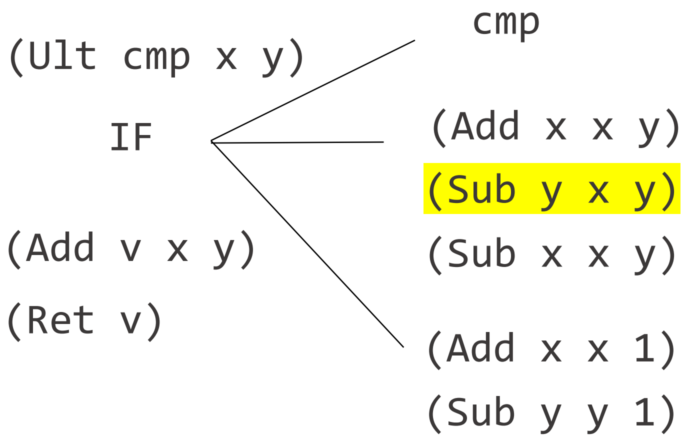
```
let cmp := x <? y in
let (x, y) :=
  if  cmp  then
    (x+y,
     y)
  else
    (x, y)
in
```

Synthesize the first intermediate spec

C code

```
if (x < y) {
  x = x + y;
  y = x - y;
  x = x - y;
}
else {
  x++; y--;
}
return x + y;
```

Formal Repr



```
(Ult cmp x y)
     IF
(Add v x y)
(Ret v)
```

```
cmp
(Add x x y)
(Sub y x y)
(Sub x x y)
(Add x x 1)
(Sub y y 1)
```

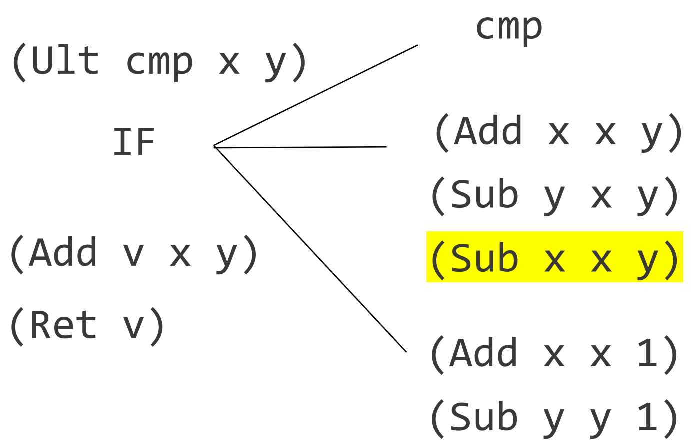
```
let cmp := x <? y in
let (x, y) :=
  if  cmp  then
    (x+y,
    (x+y)-y)
  else
    (x, y)
in
```

Synthesize the first intermediate spec

C code

```
if (x < y) {
  x = x + y;
  y = x - y;
  x = x - y;
}
else {
  x++; y--;
}
return x + y;
```

Formal Repr



```
(Ult cmp x y)
      IF
(Add v x y)
(Ret v)
```

```
cmp

(Add x x y)
(Sub y x y)
(Sub x x y)

(Add x x 1)
(Sub y y 1)
```

```
let cmp := x <? y in
let (x, y) :=
  if  cmp  then
    ((x+y)-((x+y)-y),
    (x+y)-y)
  else
    (x, y)
in
```

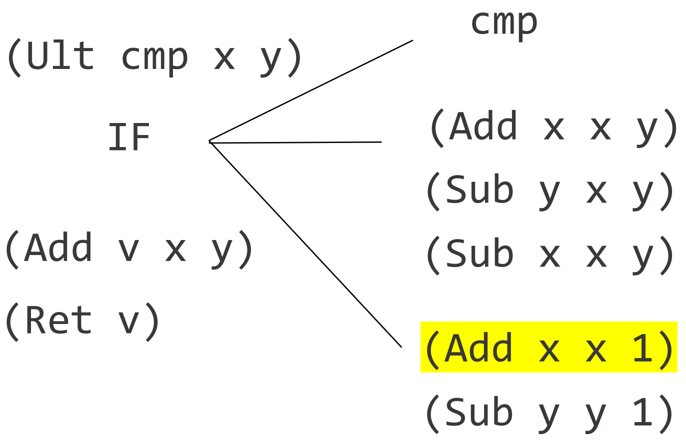
Synthesize the first intermediate spec

C code

```
if (x < y) {
  x = x + y;
  y = x - y;
  x = x - y;
}
else {
  x++; y--;
}
return x + y;
```

⟹

Formal Repr



```
(Ult cmp x y)
     IF          cmp
                 (Add x x y)
                 (Sub y x y)
                 (Sub x x y)
(Add v x y)
(Ret v)          (Add x x 1)
                 (Sub y y 1)
```

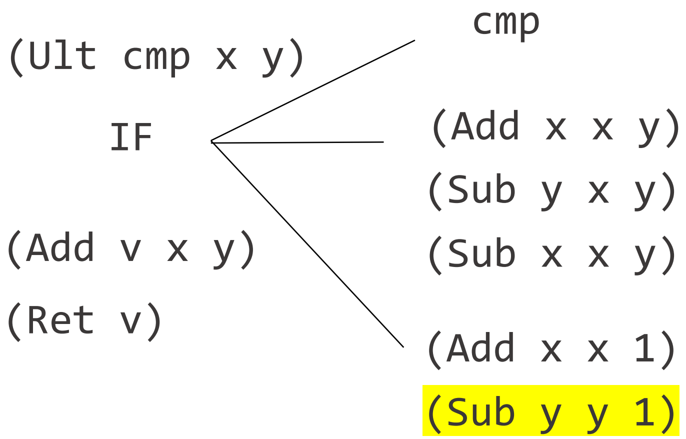
```
let cmp := x <? y in
let (x, y) :=
  if  cmp  then
    ((x+y)-((x+y)-y),
    (x+y)-y)
  else
    (x+1, y)
in
```

Synthesize the first intermediate spec

C code

```
if (x < y) {
  x = x + y;
  y = x - y;
  x = x - y;
}
else {
  x++; y--;
}
return x + y;
```

Formal Repr



```
(Ult cmp x y)
IF
(Add v x y)
(Ret v)

cmp
(Add x x y)
(Sub y x y)
(Sub x x y)
(Add x x 1)
(Sub y y 1)
```

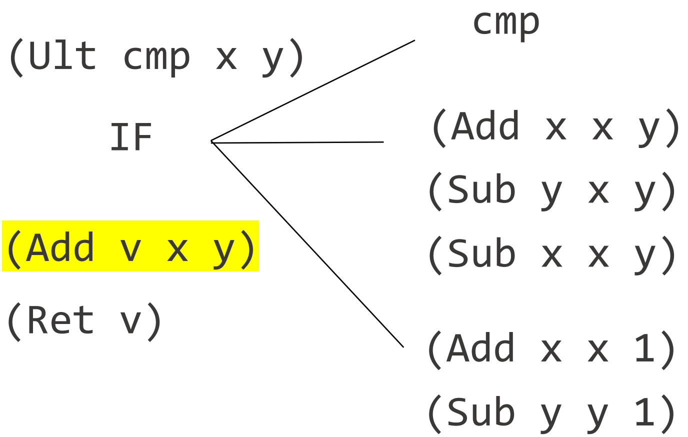
```
let cmp := x <? y in
let (x, y) :=
  if  cmp  then
    ((x+y)-((x+y)-y),
    (x+y)-y)
  else
    (x+1, y-1)
in
```

Synthesize the first intermediate spec

C code

```
if (x < y) {
  x = x + y;
  y = x - y;
  x = x - y;
}
else {
  x++; y--;
}
return x + y;
```

Formal Repr



```
(Ult cmp x y)          cmp
     IF                (Add x x y)
                       (Sub y x y)
                       (Sub x x y)
(Add v x y)
(Ret v)                (Add x x 1)
                       (Sub y y 1)
```

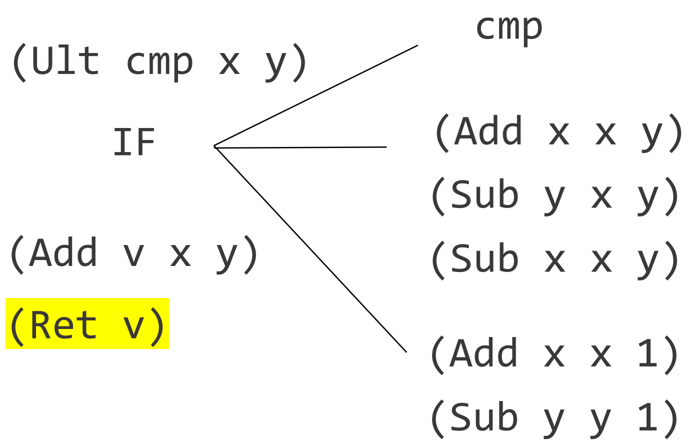
```
let cmp := x <? y in
let (x, y) :=
  if  cmp  then
    ((x+y)-((x+y)-y),
    (x+y)-y)
  else
    (x+1, y-1)
in
let v := x + y in
```

Synthesize the first intermediate spec

C code

```
if (x < y) {
  x = x + y;
  y = x - y;
  x = x - y;
}
else {
  x++; y--;
}
return x + y;
```

Formal Repr



```
(Ult cmp x y)

    IF          cmp

                (Add x x y)
                (Sub y x y)
                (Sub x x y)

(Add v x y)     (Add x x 1)
(Ret v)         (Sub y y 1)
```

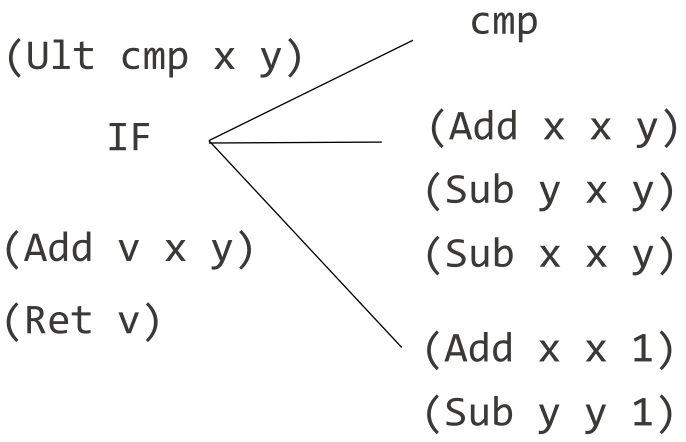
```
let cmp := x <? y in
let (x, y) :=
  if  cmp  then
    ((x+y)-((x+y)-y),
    (x+y)-y)
  else
    (x+1, y-1)
in
let v := x + y in v
```

Synthesize the first intermediate spec

C code

```
if (x < y) {
  x = x + y;
  y = x - y;
  x = x - y;
}
else {
  x++; y--;
}
return x + y;
```

Formal Repr

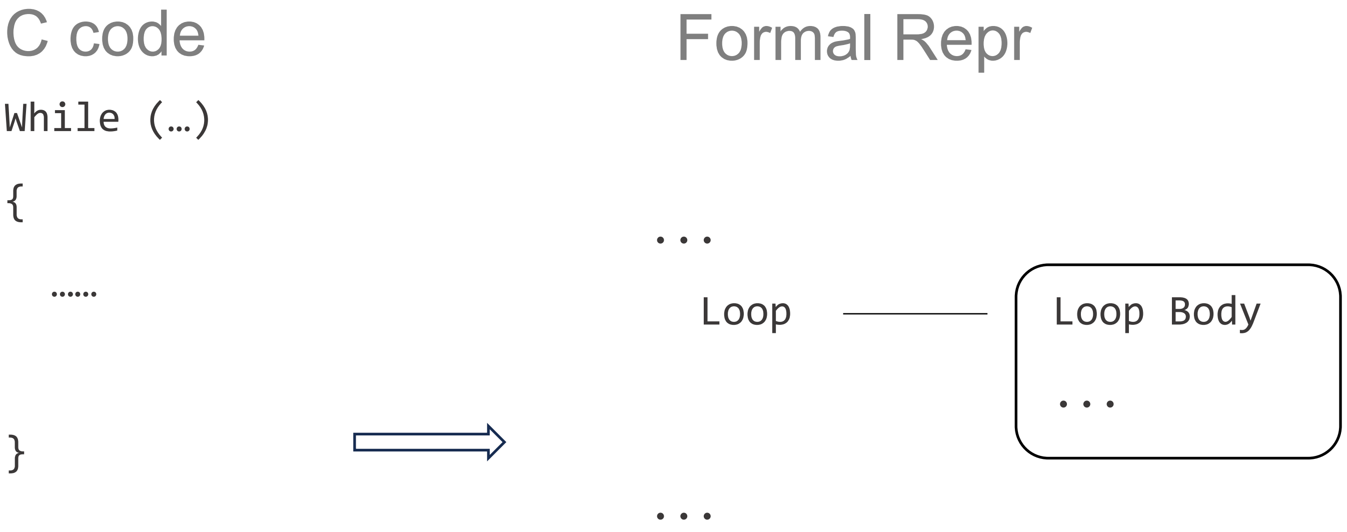


```
(Ult cmp x y)
IF
(Add v x y)
(Ret v)

cmp
(Add x x y)
(Sub y x y)
(Sub x x y)
(Add x x 1)
(Sub y y 1)
```

Initial Spec

```
let cmp := x <? y in
let (x, y) :=
  if  cmp  then
    ((x+y)-((x+y)-y),
    (x+y)-y)
  else
    (x+1, y-1)
in
let v := x + y in v
```

Synthesize the first intermediate spec



C code

```
While (…)
{
  ……

}
```

Formal Repr

```
...
  Loop ——— Loop Body
           ...
...
```

Simplifying the intermediate specs

```
let cmp := x <? y in
let (x, y) :=
  if  cmp  then
    ((x+y)-((x+y)-y),
    (x+y)-y)
  else
    (x+1, y-1)
In
let v := x + y in v
```

Simplifying the intermediate specs

```
let cmp := x <? y in
let (x, y) :=
  if  cmp  then
    ((x+y)-((x+y)-y),
    (x+y)-y)
  else
    (x+1, y-1)
in
let v := x + y in v
```

Let elimination ⟹

Simplifying the intermediate specs

```
let cmp := x <? y in
let (x, y) :=
  if  cmp  then
    ((x+y)-((x+y)-y),
    (x+y)-y)
  else
    (x+1, y-1)
in
let v := x + y in v
```

Let elimination

```
if x <? y  then
  let x := (x+y)-((x+y)-y) in
  let y := (x+y)-y in
  x + y
else
  let x := x+1 in
  let y := y-1 in
  x + y
```

Simplifying the intermediate specs

```
let cmp := x <? y in
let (x, y) :=
  if  cmp  then
    ((x+y)-((x+y)-y),
    (x+y)-y)
  else
    (x+1, y-1)
in
let v := x + y in v
```

Let elimination ⟹

```
if x <? y  then
  let x := (x+y)-((x+y)-y) in
  let y := (x+y)-y in
  x + y
else
  let x := x+1 in
  let y := y-1 in
  x + y
```

Simplifying the intermediate specs

```
let cmp := x <? y in
let (x, y) :=
  if  cmp  then
    ((x+y)-((x+y)-y),
    (x+y)-y)
  else
    (x+1, y-1)
in
let v := x + y in v
```

Let elimination

```
if x <? y  then
  let x := (x+y)-((x+y)-y) in
  let y := (x+y)-y in
  x + y
else
  let x := x+1 in
  let y := y-1 in
  x + y
```

Let elimination

```
if x <? y  then
  (x+y)-((x+y)-y) +
  ((x+y)-y)
else
  (x+1) + (y-1)
```

Simplifying the intermediate specs

```
if x <? y  then
  (x+y)-((x+y)-y) +
  ((x+y)-y)
else
  (x+1) + (y-1)
```

Simplifying the intermediate specs

```
if x <? y  then
  (x+y)-((x+y)-y) +
  ((x+y)-y)
else
  (x+1) + (y-1)
```

$$\xRightarrow{\text{Z3}}$$

```
if x <? y  then
  y + x
else
  x + y
```

Proof Hints from Z3:

```
(x+y)-((x+y)-y) + ((x+y)-y) = y + x
(x+1) + (y-1) = x + y
```

Simplifying the intermediate specs

```
if x <? y  then
  (x+y)-((x+y)-y) +
  ((x+y)-y)
else
  (x+1) + (y-1)
```

$\xrightarrow{\text{Z3}}$

```
if x <? y  then
  y + x
else
  x + y
```

$\xrightarrow{\text{Z3}}$

```
x + y
```

Proof Hints from Z3:

```
(x+y)-((x+y)-y) + ((x+y)-y) = y + x
(x+1) + (y-1) = x + y
```

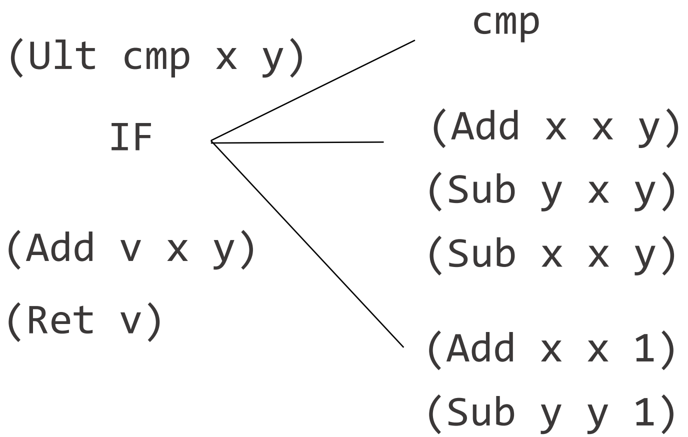
```
y + x = x + y
```

Simplifying the intermediate specs

C code

```
if (x < y) {
  x = x + y;
  y = x - y;
  x = x - y;
}
else {
  x++; y--;
}
return x + y;
```

Formal Repr

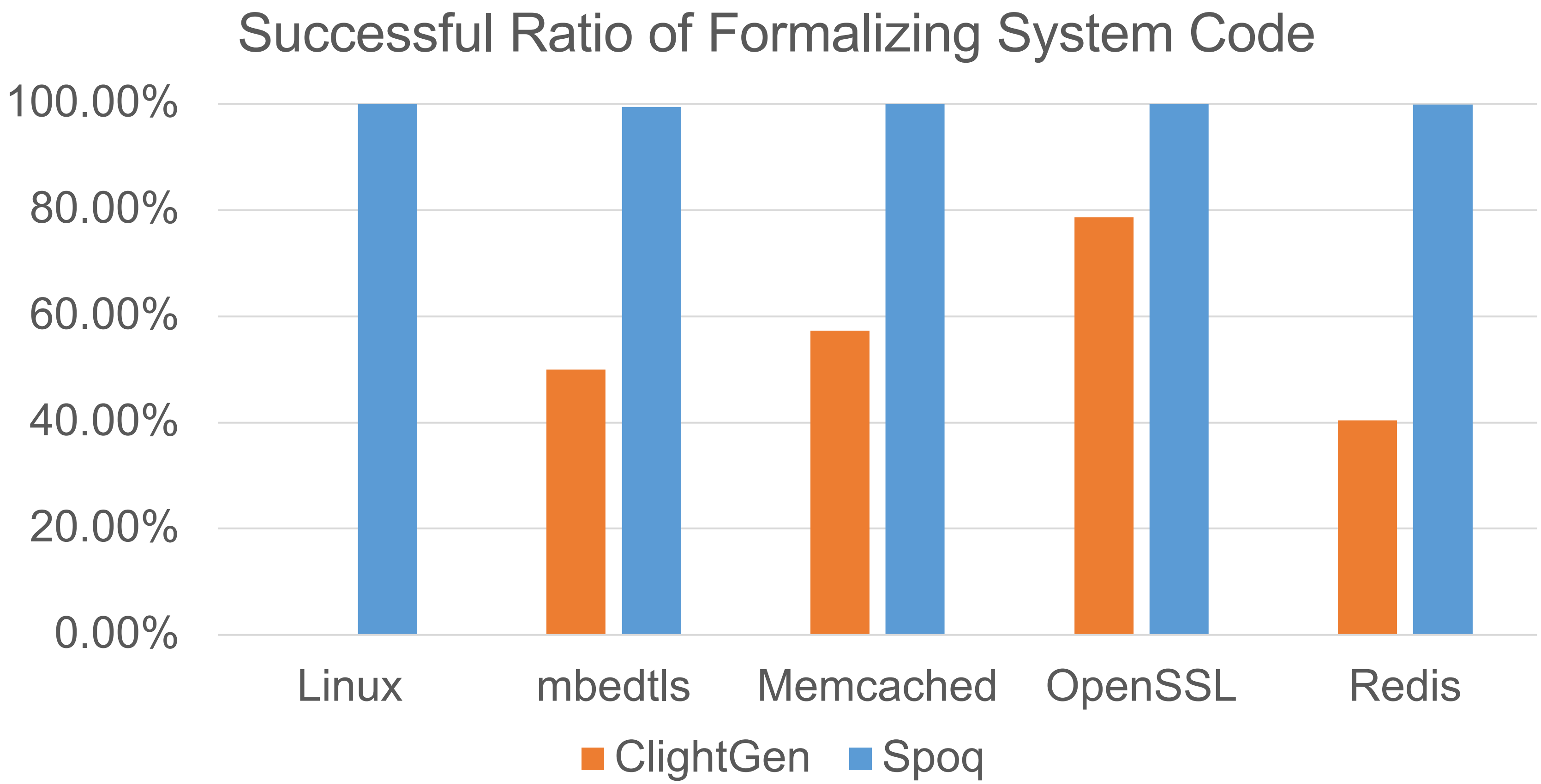


```
(Ult cmp x y)
IF
(Add v x y)
(Ret v)

cmp

(Add x x y)
(Sub y x y)
(Sub x x y)

(Add x x 1)
(Sub y y 1)
```

Gen Spec

x + y

Evaluation: Formalizing system code



Evaluation: Verify SeKVM Using Spoq

SeKVM: multiprocessor KVM Hypervisor (3.8K LOC)

Evaluation: Verify SeKVM Using Spoq

SeKVM: multiprocessor KVM Hypervisor (3.8K LOC)

Without Spoq

Verified simplified code

Manual Efforts

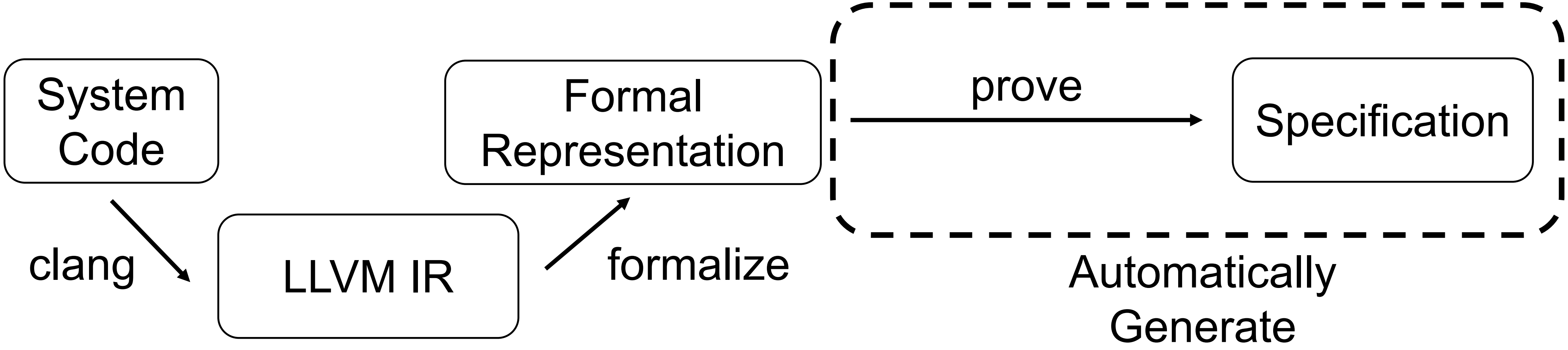
- Spec: 13.4K
- Proof: 20.1K
- Total: 33.5K

Evaluation: Verify SeKVM Using Spoq

SeKVM: multiprocessor KVM Hypervisor (3.8K LOC)

Without Spoq		**With Spoq**
Verified simplified code		Verified original code
Manual Efforts		Manual Efforts
- Spec: 13.4K	—— 86% ⬇ →	- Spec: 1.9K
- Proof: 20.1K	—— 74% ⬇ →	- Proof: 5.1K
- Total: 33.5K	—— 79% ⬇ →	- Total: 7.0K (1.8 proof/code ratio)

Summary



- <u>Formalize "Intractable" Original Code</u>
  - Rule-based reconstruction algorithm
  - Support 99% of Linux code
- <u>Automate Huge Proof Effort</u>
  - Reduce 80% manual proof effort

Spoq