

# **Completeness Thresholds for Memory Safety: Unbounded Guarantees via Bounded Proofs**

**Tobias Reinhard, Justus Fasse, Bart Jacobs**  
**KU Leuven**

October 18, 2023

# What This Work Is About

- Connection between bounded & unbounded proofs
- Ideas to increase trust in bounded model checking

# What This Work Is About

- Connection between bounded & unbounded proofs
- Ideas to increase trust in bounded model checking
- When is a bounded “proof” a proof?

# Model Checking: Easy Off-by-1 Error

- WHILE language with pointer arithmetic
- Targeted property: Memory safety
- Memory assumption `array( $a, s$ ):`  
 $a[0] \dots a[s-1]$  allocated

```
for i in [0 : s-1] do  
  !a[i+1]
```

# Model Checking: Easy Off-by-1 Error

- WHILE language with pointer arithmetic
- Targeted property: Memory safety
- Memory assumption `array(a, s)`:  
 $a[0] \dots a[s-1]$  allocated

```
for i in [0 :  $s$ -1] do  
    !a[i+1]
```

Which bounds should we choose for  $s$ ?

- $s = 0$ : No error
- $s = 1$ : Error

# Model Checking: “Harder” Off-by-N Error

Memory assumption:  
`array(a, s)`

```
for i in [0 : s-2] do  
  !a[i+2]
```

Which bounds should we choose for *s*?

# Model Checking: “Harder” Off-by-N Error

Memory assumption:  
`array(a, s)`

`for i in [0 : s-2] do`  
`!a[i+2]`

Which bounds should we choose for *s*?

- *s* = 0: No error
- *s* = 1: No error
- *s* = 2: Error

# Model Checking: No Off-by-N Error

Memory assumption:  
array( $a$ ,  $s$ )

```
for i in [0 :  $s-1$ ] do  
  !a[i]
```

Which  $s$  can convince us?



# Model Checking: No Off-by-N Error

Memory assumption:  
`array(a, s)`

```
for i in [0 : s-1] do  
  !a[i]
```

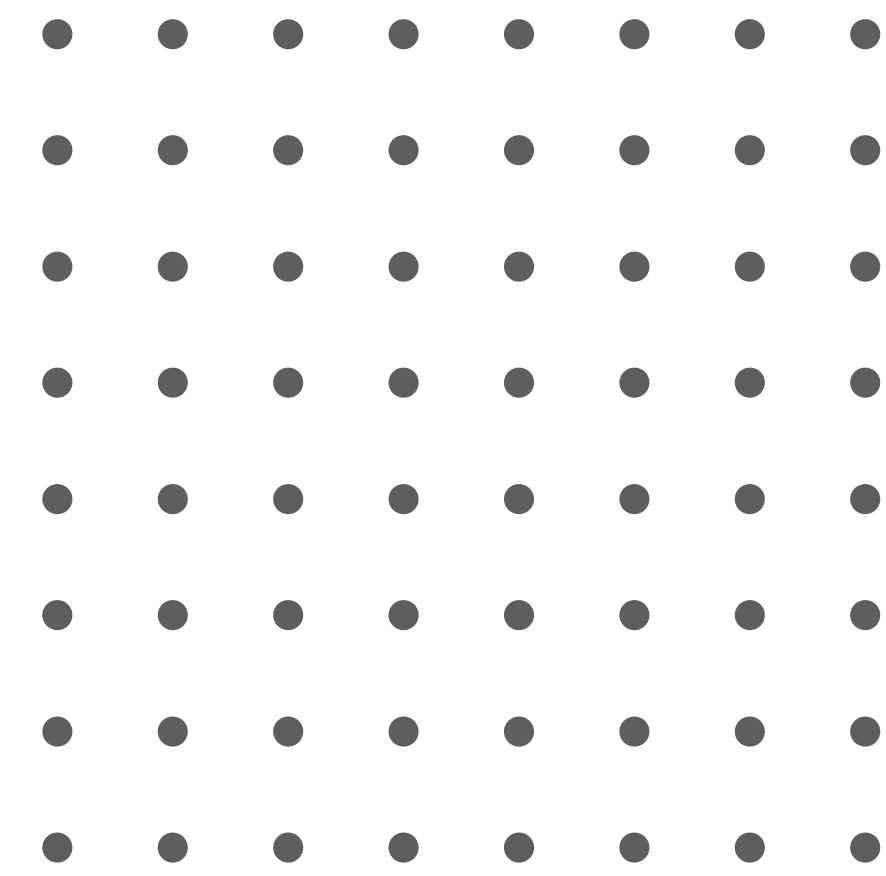
Which *s* can convince us?

- *s* = 0: No error
- *s* = 1: No error
- *s* = 2: No error
- *s* = 3: No error
- ⋮

⇒ Which size bound is large enough?

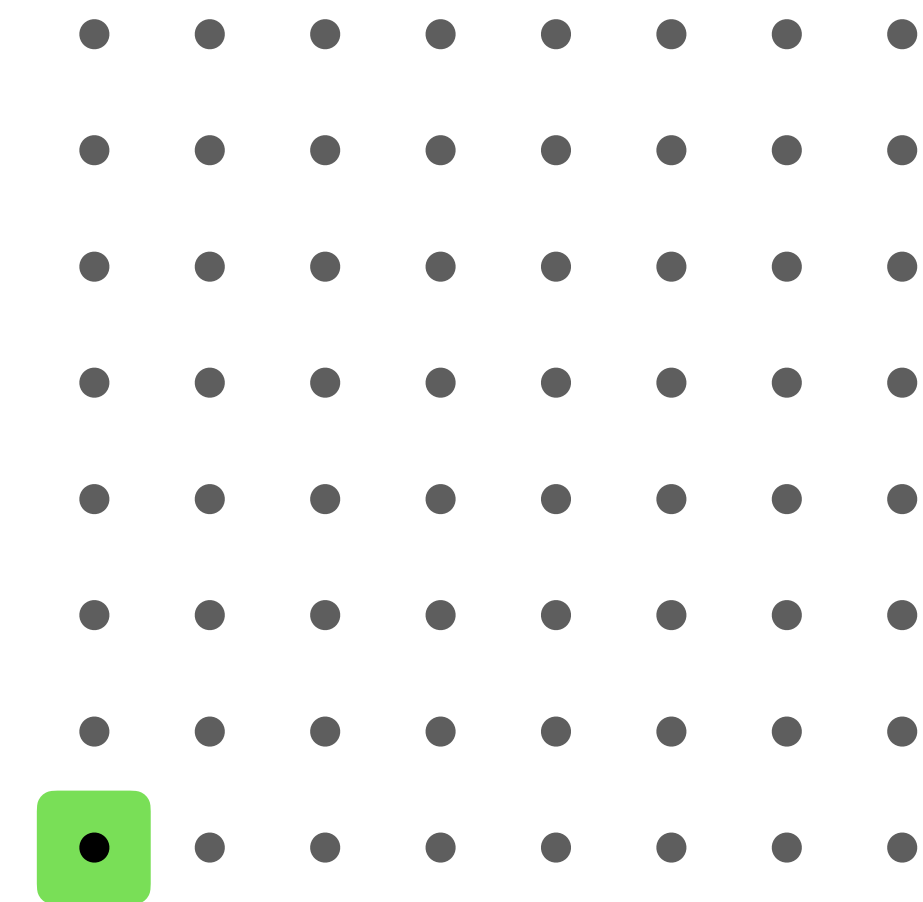
# Model Checking Finite Systems

- Finite state transition system  $T$
- Prove property  $Gp$   
 $G \approx$  globally  $\approx p$  holds in every state
- Approach:  
Prove  $Gp$  for all paths up to length  $k$   
 $T \models_k Gp$



# Model Checking Finite Systems

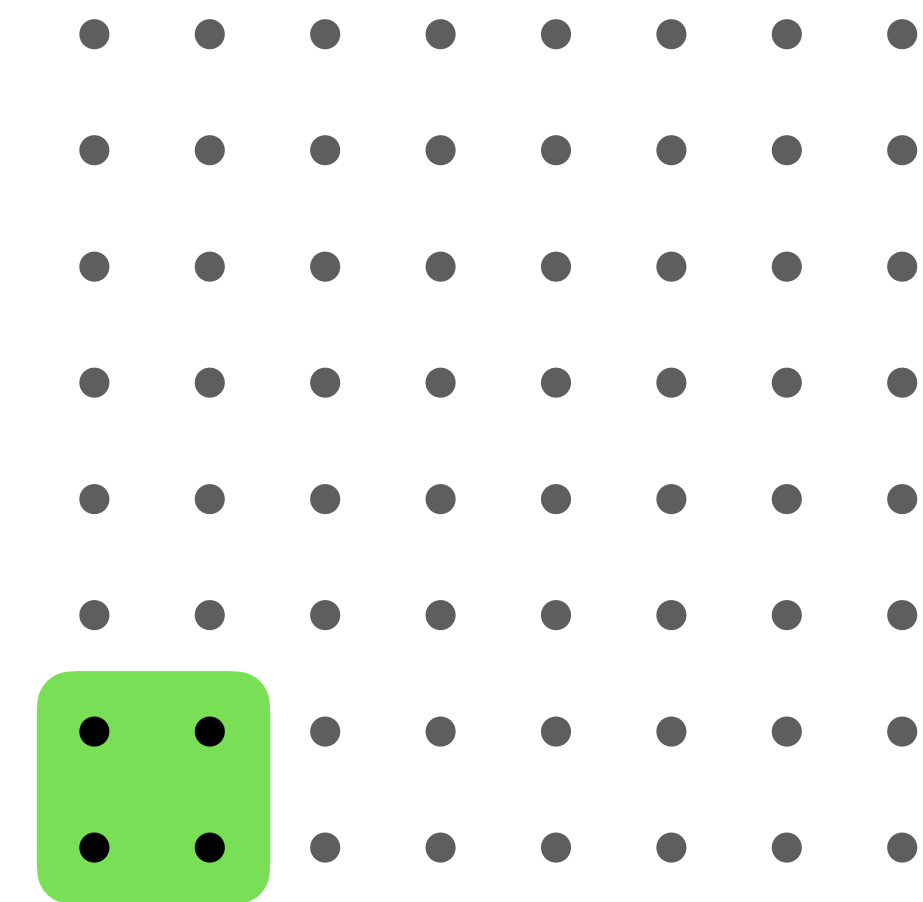
- Finite state transition system  $T$
- Prove property  $Gp$   
 $G \approx$  globally  $\approx p$  holds in every state
- Approach:  
Prove  $Gp$  for all paths up to length  $k$   
 $T \models_k Gp$



$$T \models_0 Gp$$

# Model Checking Finite Systems

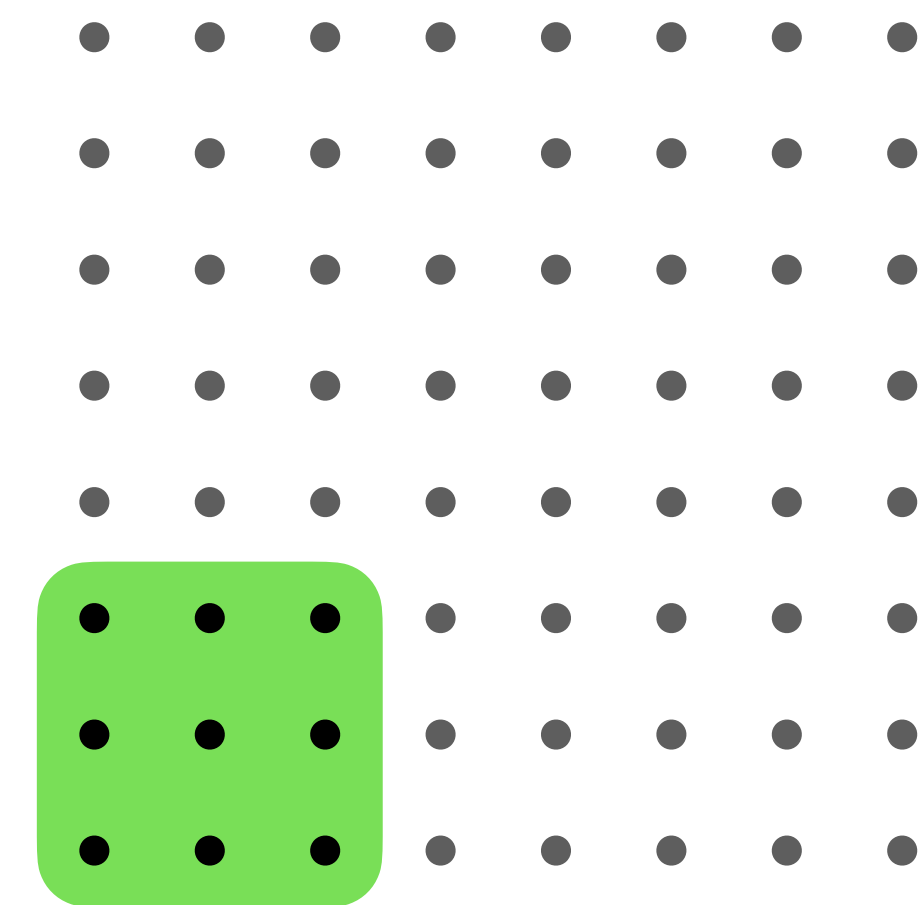
- Finite state transition system  $T$
- Prove property  $Gp$   
 $G \approx$  globally  $\approx p$  holds in every state
- Approach:  
Prove  $Gp$  for all paths up to length  $k$   
 $T \models_k Gp$



$$T \models_1 Gp$$

# Model Checking Finite Systems

- Finite state transition system  $T$
- Prove property  $Gp$   
 $G \approx$  globally  $\approx p$  holds in every state
- Approach:  
Prove  $Gp$  for all paths up to length  $k$   
 $T \models_k Gp$



$$T \models_2 Gp$$

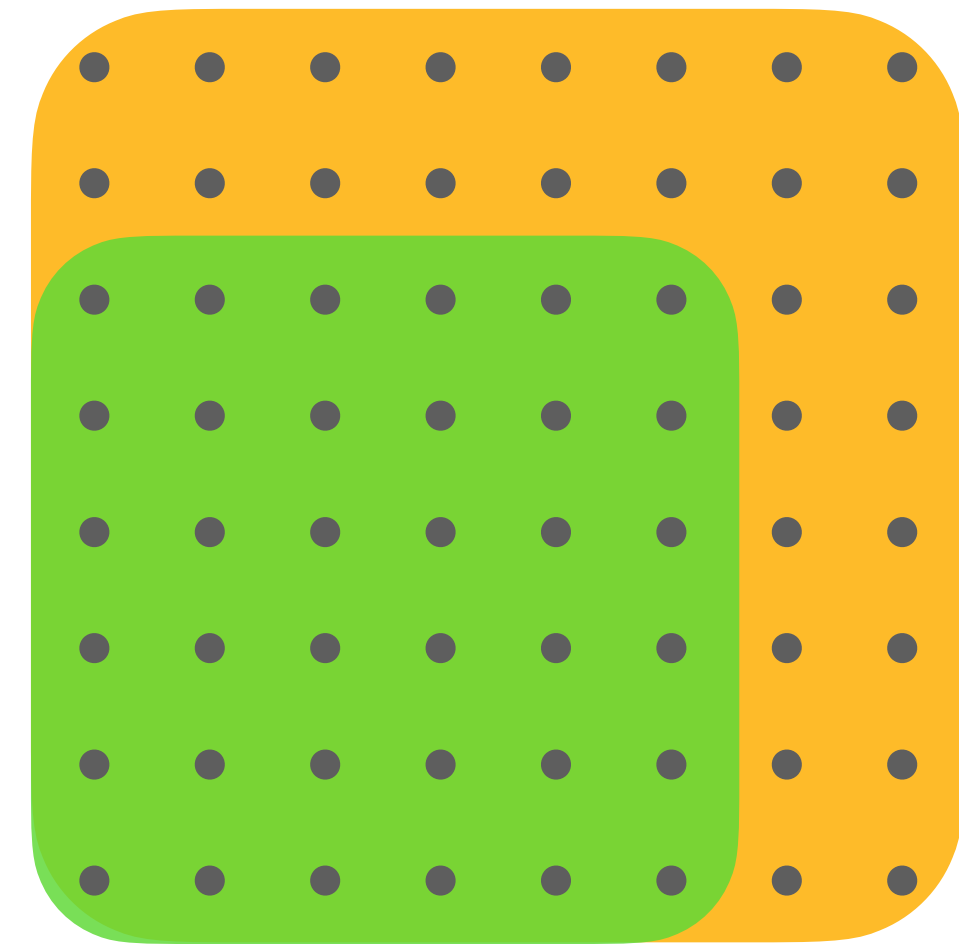
When should we stop?

# Completeness Thresholds for Finite Systems

- $k$  is completeness thresholds (CT) iff

$$T \models_k \phi \Rightarrow T \models \phi$$

- For specific  $\phi$ :  
Can over-approximate CT via of key props of  $T$

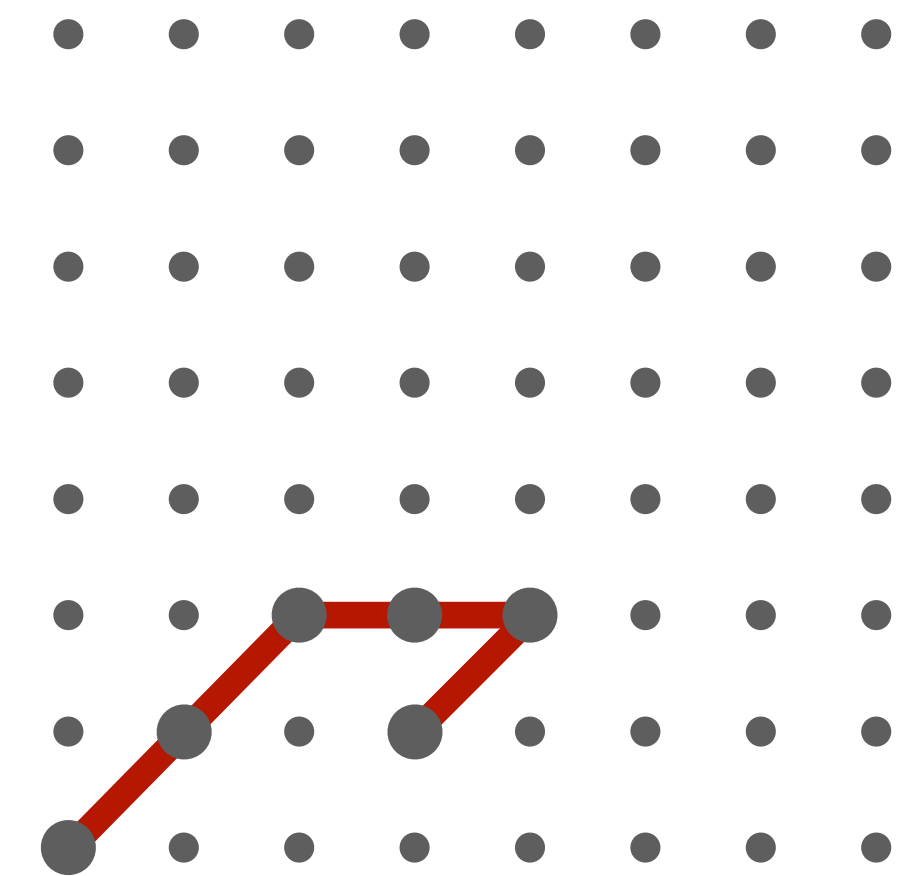


# Completeness Thresholds for Finite Systems

- $k$  is completeness thresholds (CT) iff

$$T \models_k \phi \Rightarrow T \models \phi$$

- For specific  $\phi$ :  
Can over-approximate CT via of key props of  $T$
- For  $\phi = Gp$  we know  
 $CT(T, Gp) = \text{recurrence\_diameter}(T)$   
(length of longest loop-free path)



$$\text{recurrence\_diameter}(T) = 5$$

# Completeness Thresholds for Finite Systems

- $k$  is completeness thresholds (CT) iff

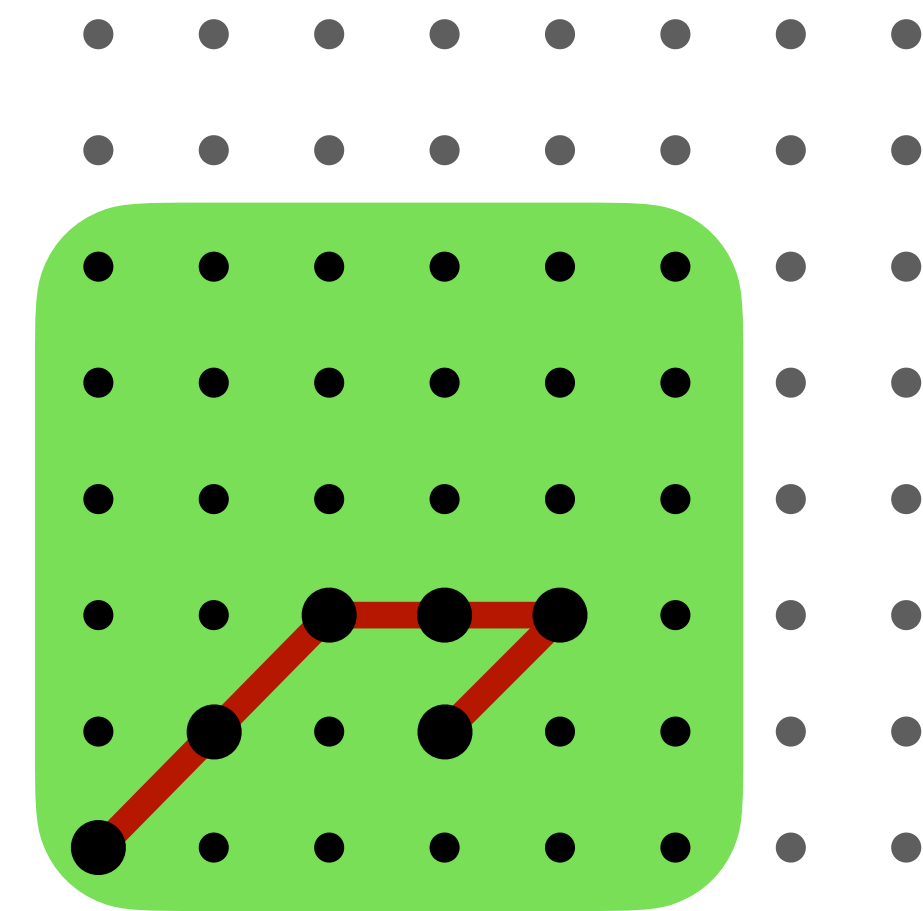
$$T \models_k \phi \Rightarrow T \models \phi$$

- For specific  $\phi$ :

Can over-approximate CT via of key props of  $T$

- For  $\phi = Gp$  we know

$CT(T, Gp) = \text{recurrence\_diameter}(T)$   
(length of longest loop-free path)



$\text{recurrence\_diameter}(T) = 5$



$T \models_5 Gp$



# Completeness Thresholds for Finite Systems

- $k$  is completeness thresholds (CT) iff

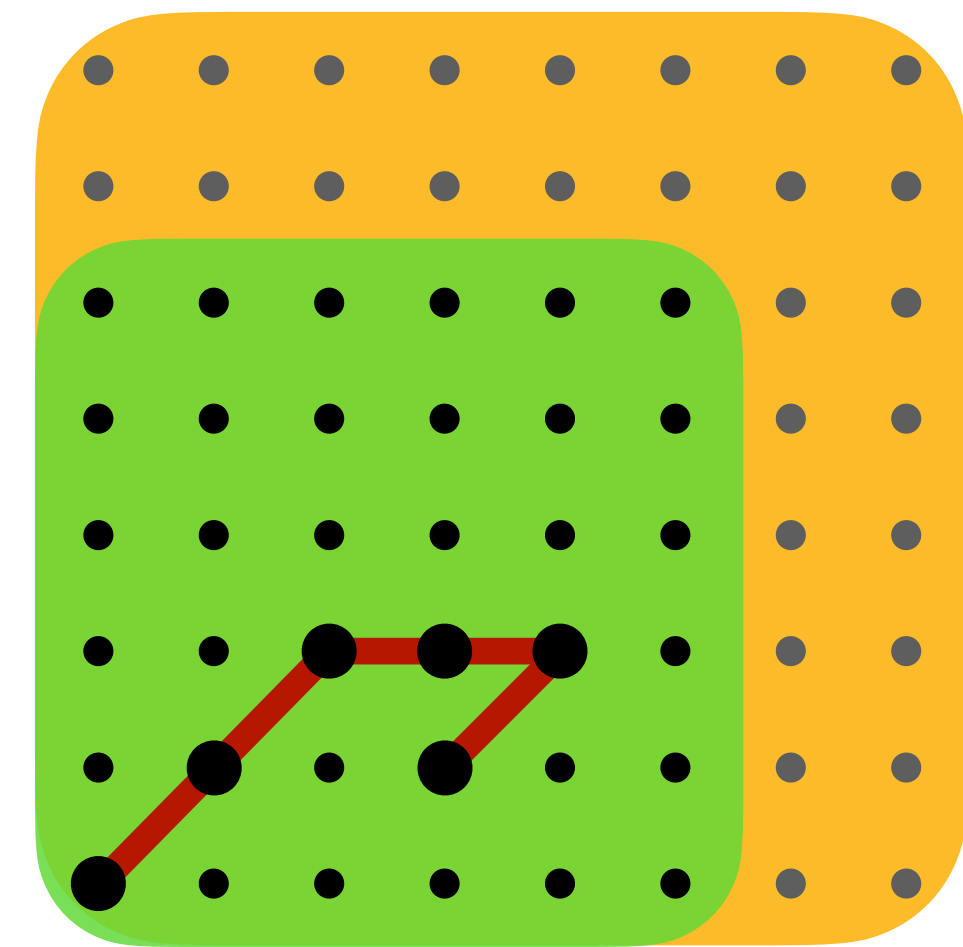
$$T \models_k \phi \Rightarrow T \models \phi$$

- For specific  $\phi$ :

Can over-approximate CT via of key props of  $T$

- For  $\phi = Gp$  we know

$CT(T, Gp) = \text{recurrence\_diameter}(T)$   
(length of longest loop-free path)



$\text{recurrence\_diameter}(T) = 5$



$T \models_5 Gp$

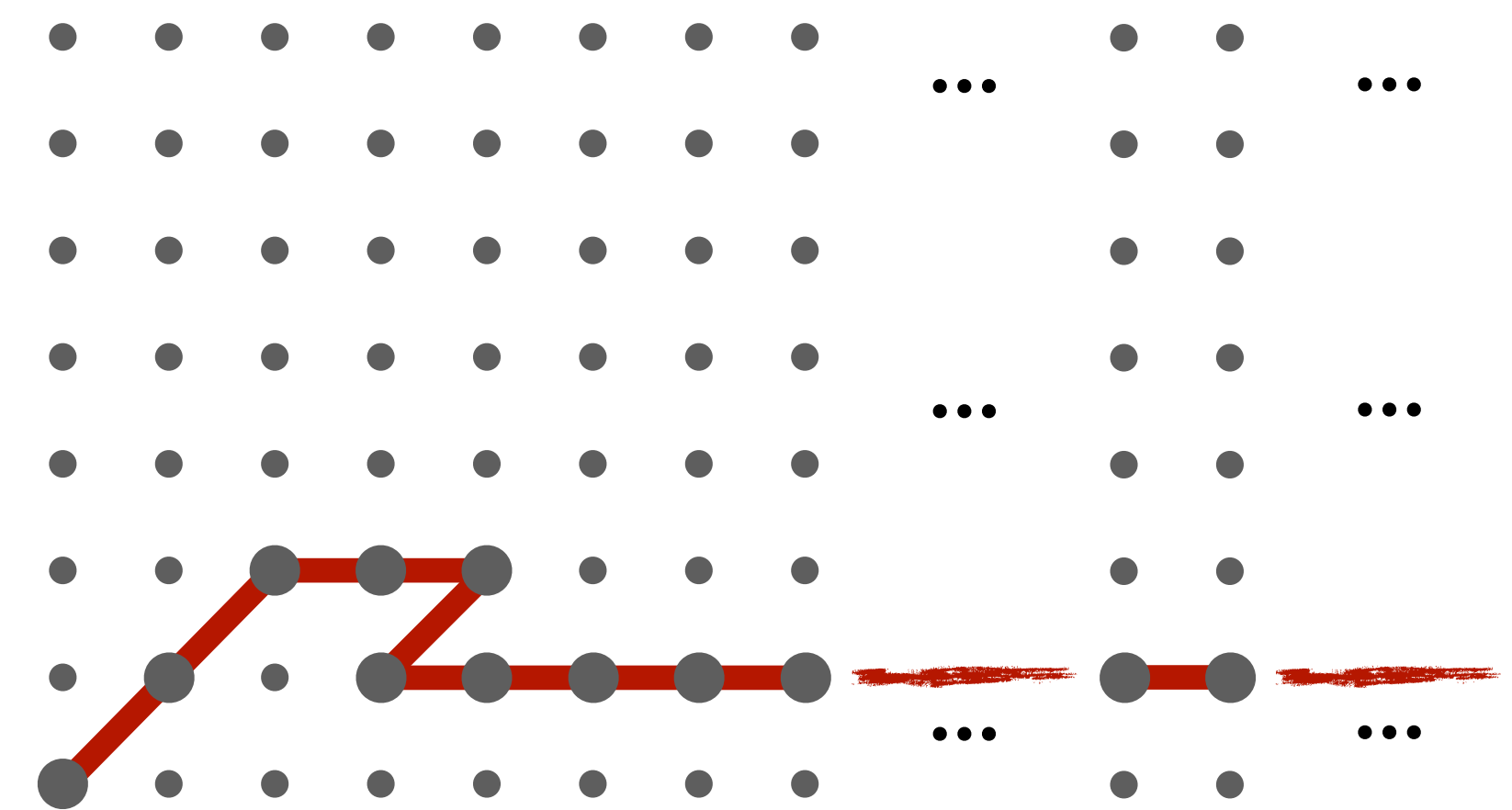


$T \models Gp$

# CTs for Infinite Systems?

## Problem

Key properties used to describe CTs may be  $\infty$



$$\text{recurrence\_diameter}(T) = \infty$$

# CTs for Infinite Systems?

## Problem

Key properties used to describe CTs may be  $\infty$

## Our Approach

Analyse program's *verification conditions*  
instead of transition system

# How Does the Array Size Affect Memory Safety?

Memory assumption:  
array( $a$ ,  $s$ )

for  $i$  in [ $L$  :  $s-R$ ] do  
  ! $a[i+Z]$

Only source for memory errors

# How Does the Array Size Affect Memory Safety?

Memory assumption:  
 $\text{array}(a, s)$

for  $i$  in  $[L : s-R]$  do  
   $!a[i+Z]$

Range  $L, \dots, s-R$  empty?

Yes

$$s^- < L + R$$

✓ No iteration  
⇒ Memory safe

# How Does the Array Size Affect Memory Safety?

Memory assumption:  
 $\text{array}(a, s)$

for  $i$  in  $[L : s-R]$  do  
   $!a[i+Z]$

Range  $L, \dots, s-R$  empty?

Yes

$$s^- < L + R$$

No need to check

# How Does the Array Size Affect Memory Safety?

Memory assumption:  
 $\text{array}(a, s)$

for  $i$  in  $[L : s-R]$  do  
   $!a[i+Z]$

Range  $L, \dots, s-R$  empty?

Yes

$$s^- < L + R$$

No need to check

No

$$s^+ \geq L + R$$

$a[L+Z] \dots a[s-R+Z]$  allocated?

# How Does the Array Size Affect Memory Safety?

Memory assumption:  
 $\text{array}(a, s)$

for  $i$  in  $[L : s-R]$  do  
   $!a[i+Z]$

Range  $L, \dots, s-R$  empty?

Yes

$$s^- < L + R$$

No need to check

No

$$s^+ \geq L + R$$

$a[L+Z] \dots a[s-R+Z]$  allocated?

$$\iff 0 \leq L + Z \quad \wedge \quad s^+ - R + Z < s^+ \quad ?$$



# How Does the Array Size Affect Memory Safety?

Memory assumption:  
 $\text{array}(a, s)$

for  $i$  in  $[L : s-R]$  do  
   $!a[i+Z]$

Range  $L, \dots, s-R$  empty?

Yes

$$s^- < L + R$$

No need to check

No

$$s^+ \geq L + R$$

$a[L+Z] \dots a[s-R+Z]$  allocated?

$$\iff 0 \leq L + Z \quad \wedge \quad \cancel{s^+} - R + Z < \cancel{s^+} ?$$

$$\iff 0 \leq L + Z \quad \wedge \quad -R + Z < 0 ?$$

No  $s^+ \Rightarrow$  Can check any  $s^+ \geq L + R$

# How Does the Array Size Affect Memory Safety?

Memory assumption:  
 $\text{array}(a, s)$

for  $i$  in  $[L : s-R]$  do  
   $!a[i+Z]$

Range  $L, \dots, s-R$  empty?

Yes

$$s^- < L + R$$

No need to check

No

$$s^+ \geq L + R$$

Can check any  $s^+ \geq L + R$

# How Does the Array Size Affect Memory Safety?

Memory assumption:  
 $\text{array}(a, s)$

for  $i$  in  $[L : s-R]$  do  
   $!a[i+Z]$

Range  $L, \dots, s-R$  empty?

Yes

$$s^- < L + R$$

No need to check

No

$$s^+ \geq L + R$$

Can check any  $s^+ \geq L + R$

Found CT:  $\{s^+\}$

# Completeness Thresholds

- Program variable  $x$  with domain  $X$
- Specification  $\forall x \in X. Spec(c)$

# Completeness Thresholds

- Program variable  $x$  with domain  $X$
- Specification  $\forall x \in X. Spec(c)$
- Subdomain  $Q \subseteq X$  is a CT for  $x$  in  $\forall x \in X. Spec(c)$  iff
$$\models \forall x \in Q. Spec(c) \Rightarrow \models \forall x \in X. Spec(c)$$
- For us: CT are subdomains, not depths

# Verification Conditions

- Logical formula  $vc$  is VC for any spec  $Spec(c)$  iff

$$\models vc \Rightarrow \models Spec(c)$$

- Can verify VC instead of program
- In general: VCs are over-approximations, i.e.,  
possible that  $\not\models vc$  but  $\models Spec(c)$

# How to Prove CTs

- Generate VC:  $Spec(c) \rightsquigarrow \forall x \in X. vc(x)$

# How to Prove CTs

- Generate VC:  $Spec(c) \rightsquigarrow \forall x \in X. vc(x)$
- Identify subdomain  $Y \subseteq X$  where choice  $x \in Y$  does not influence validity of  $vc(x)$

$$\left( \models vc(x) \iff \models vc' \text{ with } x \notin \text{free}(vc') \right)$$

$\implies$  Found CT:  $(X \setminus Y) \cup \{y\}$  (for any choice of  $y \in Y$ )



# Proving CT in Action

Memory assumption:  
array( $a$ ,  $s$ )

for  $i$  in [ $L$  :  $s-R$ ] do  
  ! $a[i+Z]$

# Proving CT in Action

Memory assumption:

$\text{array}(a, s)$

for  $i$  in  $[L : s-R]$  do

$!a[i+Z]$

Generate VC

(fully automated)

VC  $vc_0 := \forall s. \text{array}(a, s) \rightarrow \forall i \in \{L, \dots, s-R\}. a[i+Z] \text{ alloc}$

# Proving CT in Action

$$\text{VC } vc_0 := \forall s. \text{array}(a, s) \rightarrow \forall i \in \{L, \dots, s - R\} . a[i+Z] \text{ alloc}$$

Range  $L, \dots, s-R$  empty?

# Proving CT in Action

VC  $vc_0 := \forall s. \text{array}(a, s) \rightarrow \forall i \in \{L, \dots, s - R\}. a[i + Z] \text{ alloc}$

Range  $L, \dots, s - R$  empty?

Yes

$$s^- < L + R$$

Simplify VC!

$$\begin{aligned} vc_0 &\equiv \forall s^-. \dots \rightarrow \forall i \in \emptyset. \dots \\ &\equiv \text{True} \end{aligned}$$

# Proving CT in Action

VC  $vc_0 := \forall s. \text{array}(a, s) \rightarrow \forall i \in \{L, \dots, s - R\}. a[i + Z] \text{ alloc}$

Range  $L, \dots, s - R$  empty?

Yes

$s^- < L + R$

Simplify VC!

No need to check

# Proving CT in Action

VC  $vc_0 := \forall s. \text{array}(a, s) \rightarrow \forall i \in \{L, \dots, s - R\}. a[i+Z] \text{ alloc}$

Range  $L, \dots, s-R$  empty?

Yes

$$s^- < L + R$$

No need to check

Simplify VC!

No

$$s^+ \geq L + R$$

$$vc_0 \equiv \forall i. (L \leq i < s^+ - R) \rightarrow (0 \leq i+Z < s^+)$$

# Proving CT in Action

$$\text{VC } vc_0 := \forall s. \text{array}(a, s) \rightarrow \forall i \in \{L, \dots, s - R\}. a[i+Z] \text{ alloc}$$

Range  $L, \dots, s-R$  empty?

Yes

$$s^- < L + R$$

No need to check

Simplify VC!

No

$$s^+ \geq L + R$$

$$\begin{aligned} vc_0 &\equiv \forall i. (L \leq i < s^+ - R) \rightarrow (0 \leq i+Z < s^+) \\ &\equiv \forall i. (L \leq i \rightarrow 0 \leq i+Z) \\ &\quad \wedge (i \leq -R) \rightarrow i+Z < 0) \end{aligned}$$

$\Rightarrow$  Validity does not depend on size

# Proving CT in Action

VC  $vc_0 := \forall s. \text{array}(a, s) \rightarrow \forall i \in \{L, \dots, s - R\} . a[i+Z] \text{ alloc}$

Range  $L, \dots, s-R$  empty?

Yes

$$s^- < L + R$$

No need to check

No

$$s^+ \geq L + R$$

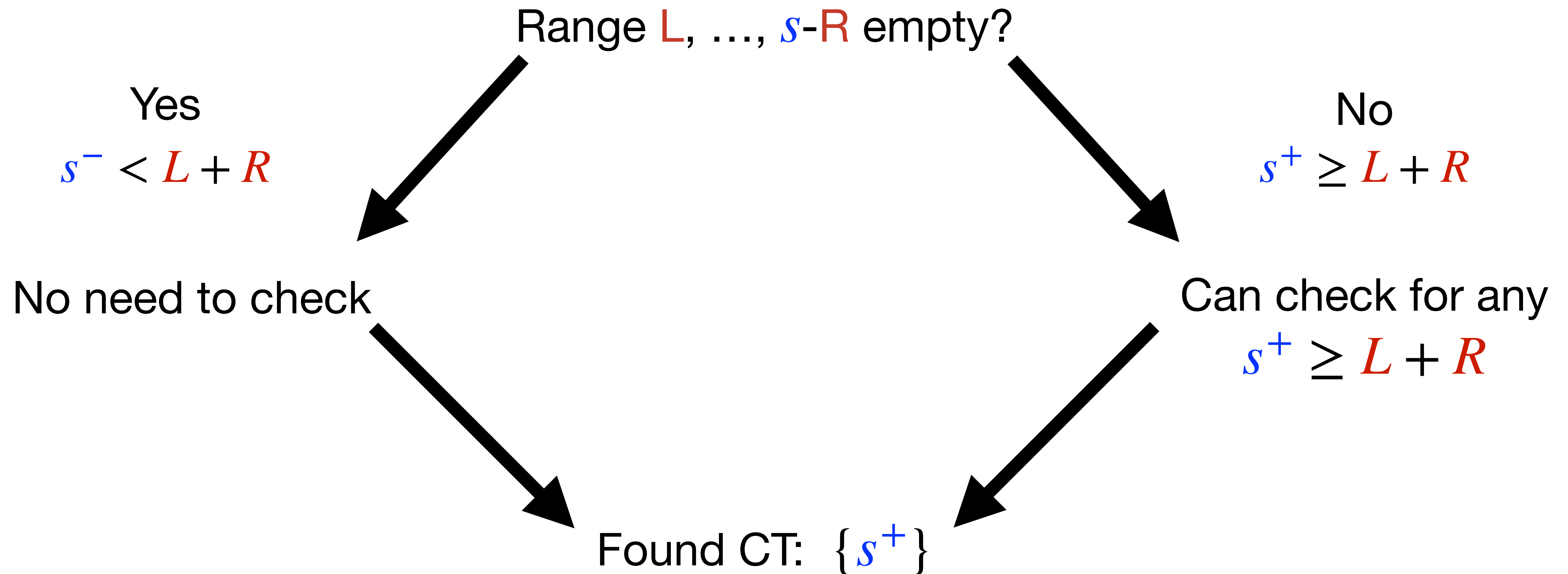
Can check for any

$$s^+ \geq L + R$$



# Proving CT in Action

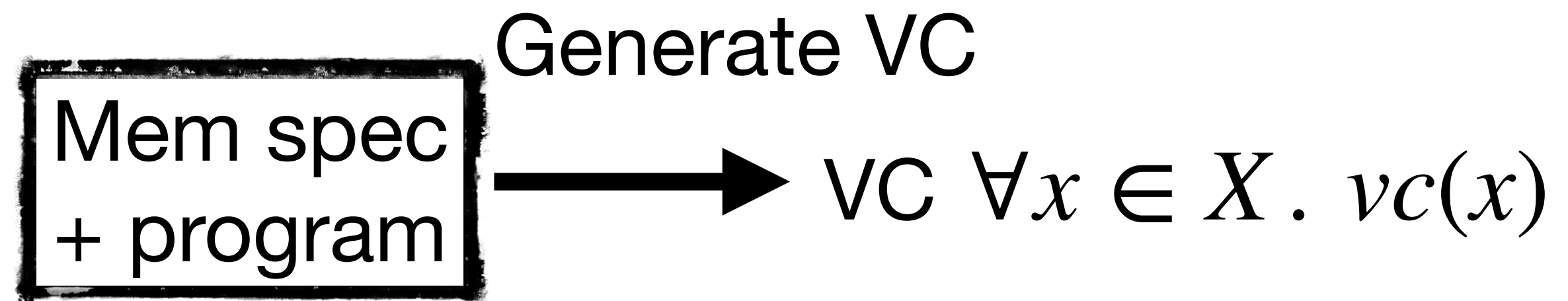
VC  $vc_0 := \forall s. \text{array}(a, s) \rightarrow \forall i \in \{L, \dots, s - R\}. a[i+Z] \text{ alloc}$



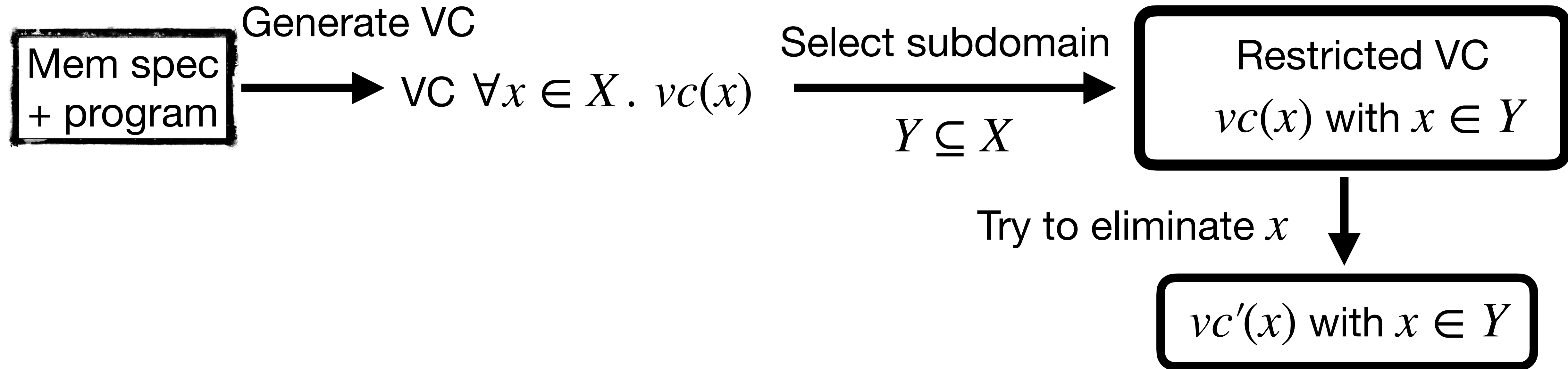
# Workflow: How to Find CTs

Mem spec  
+ program

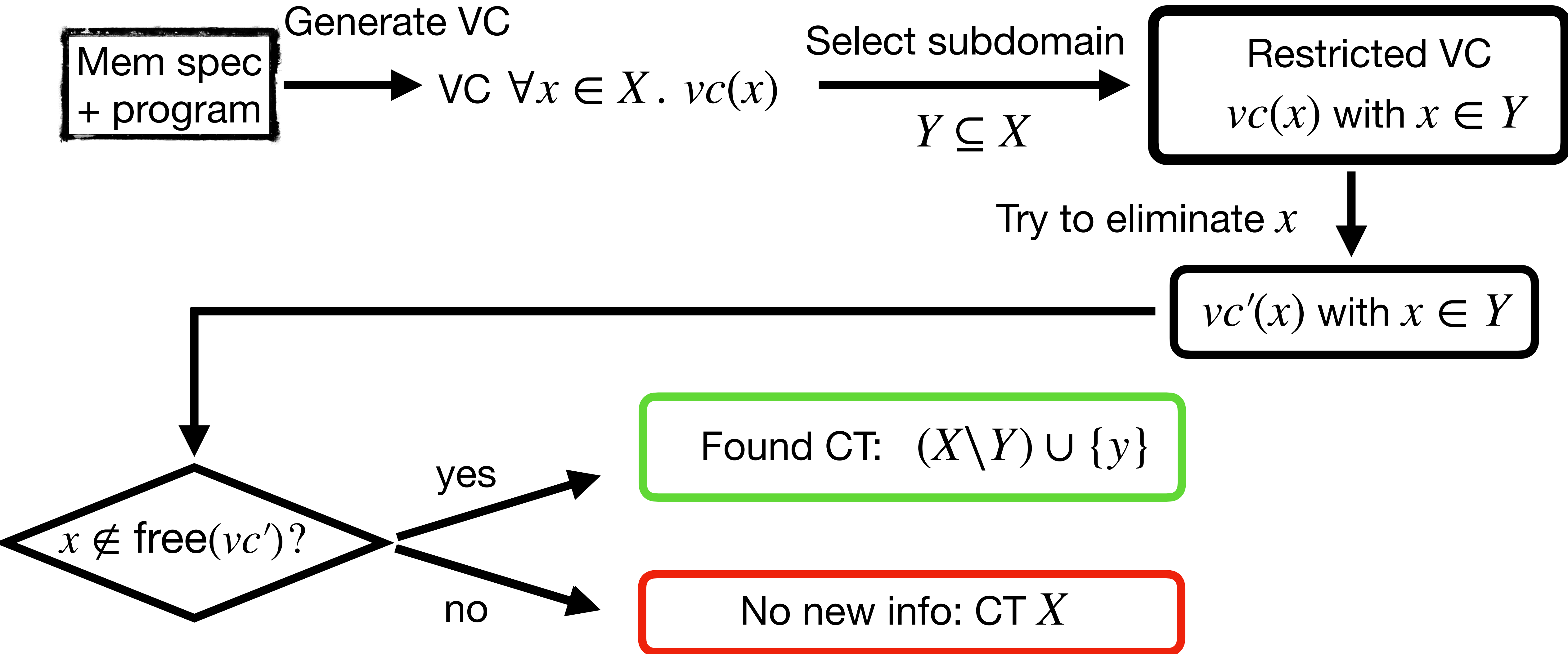
# Workflow: How to Find CTs



# Workflow: How to Find CTs

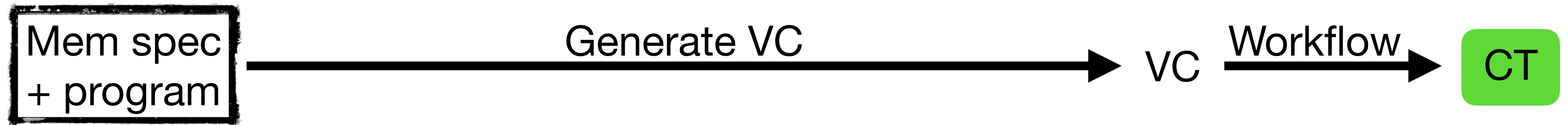


# Workflow: How to Find CTs



# Scalability

## Program Slicing



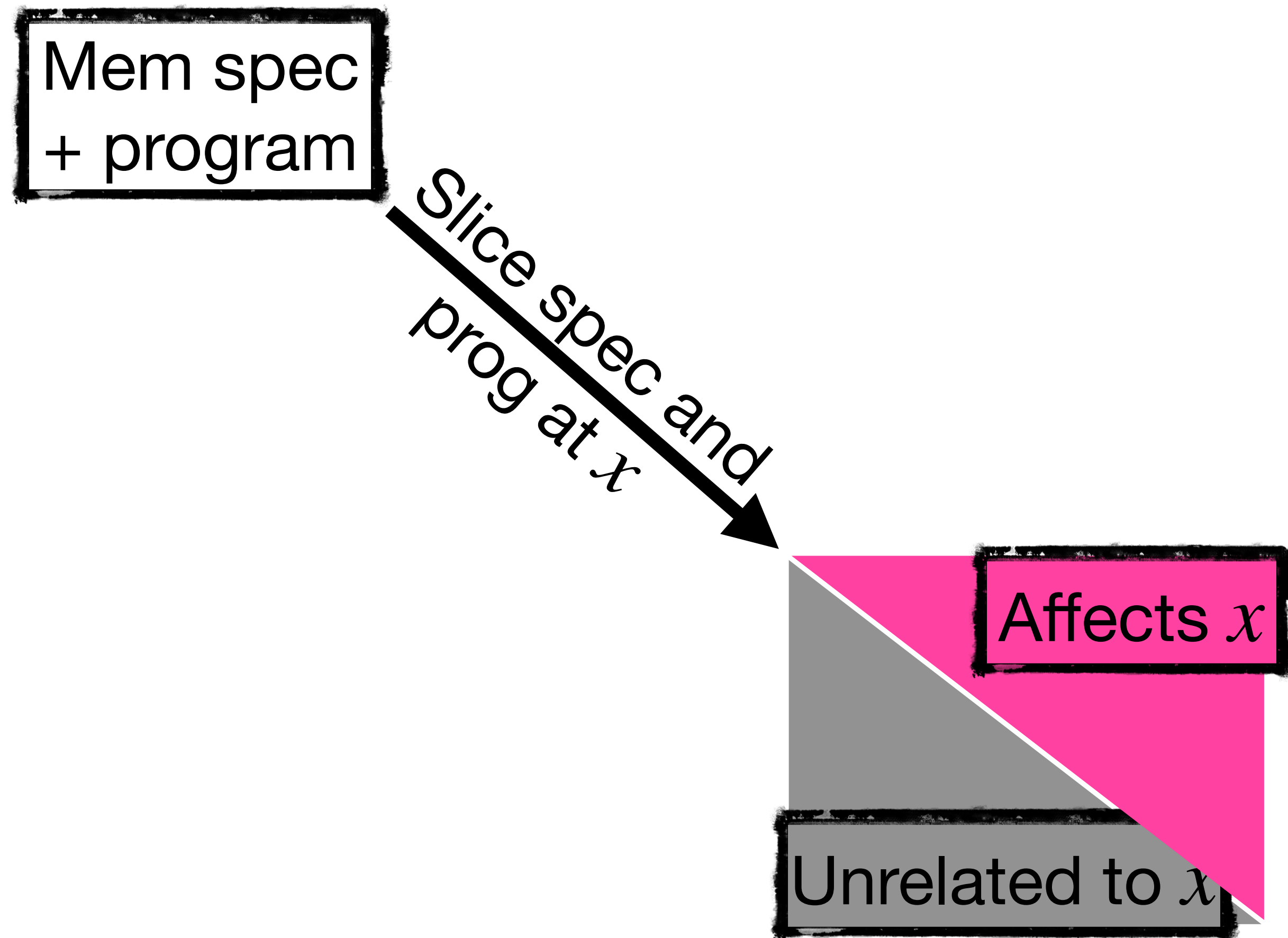
# Scalability

## Program Slicing



# Scalability

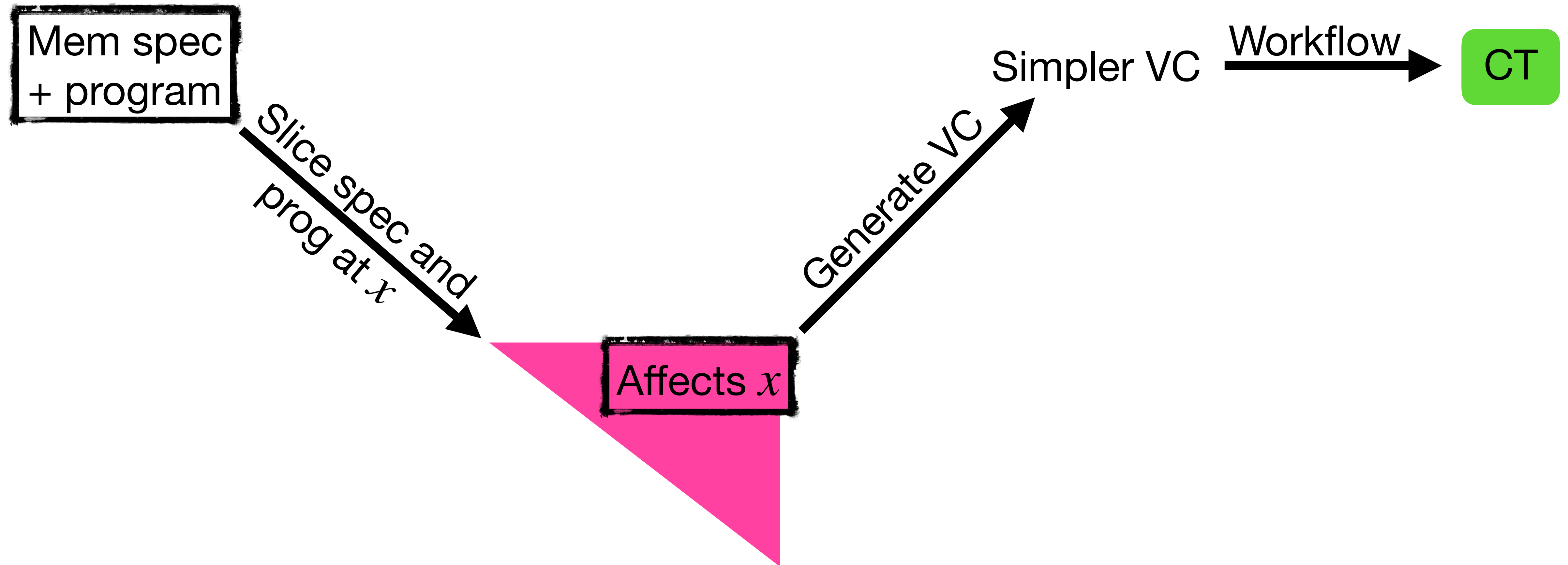
## Program Slicing



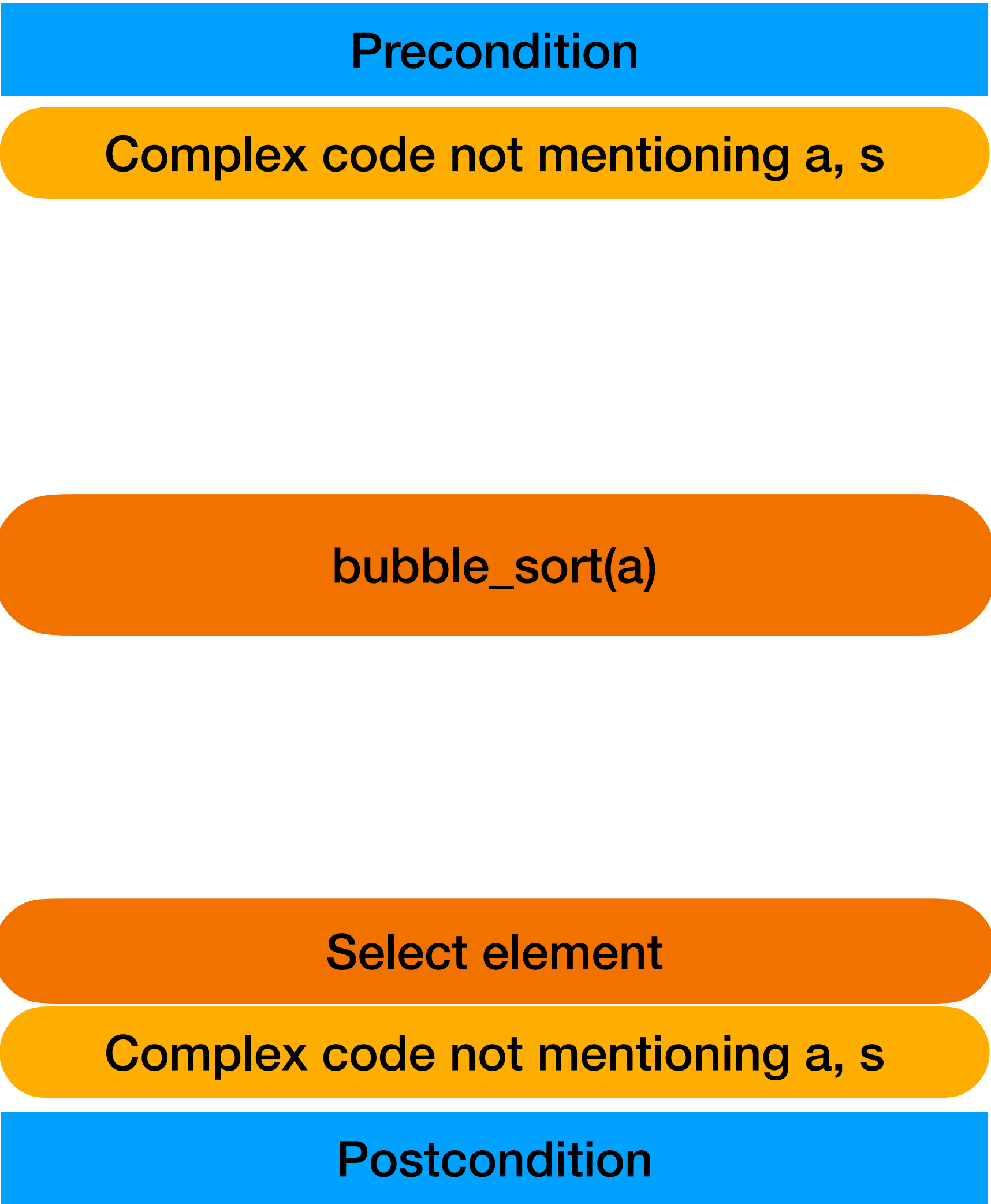
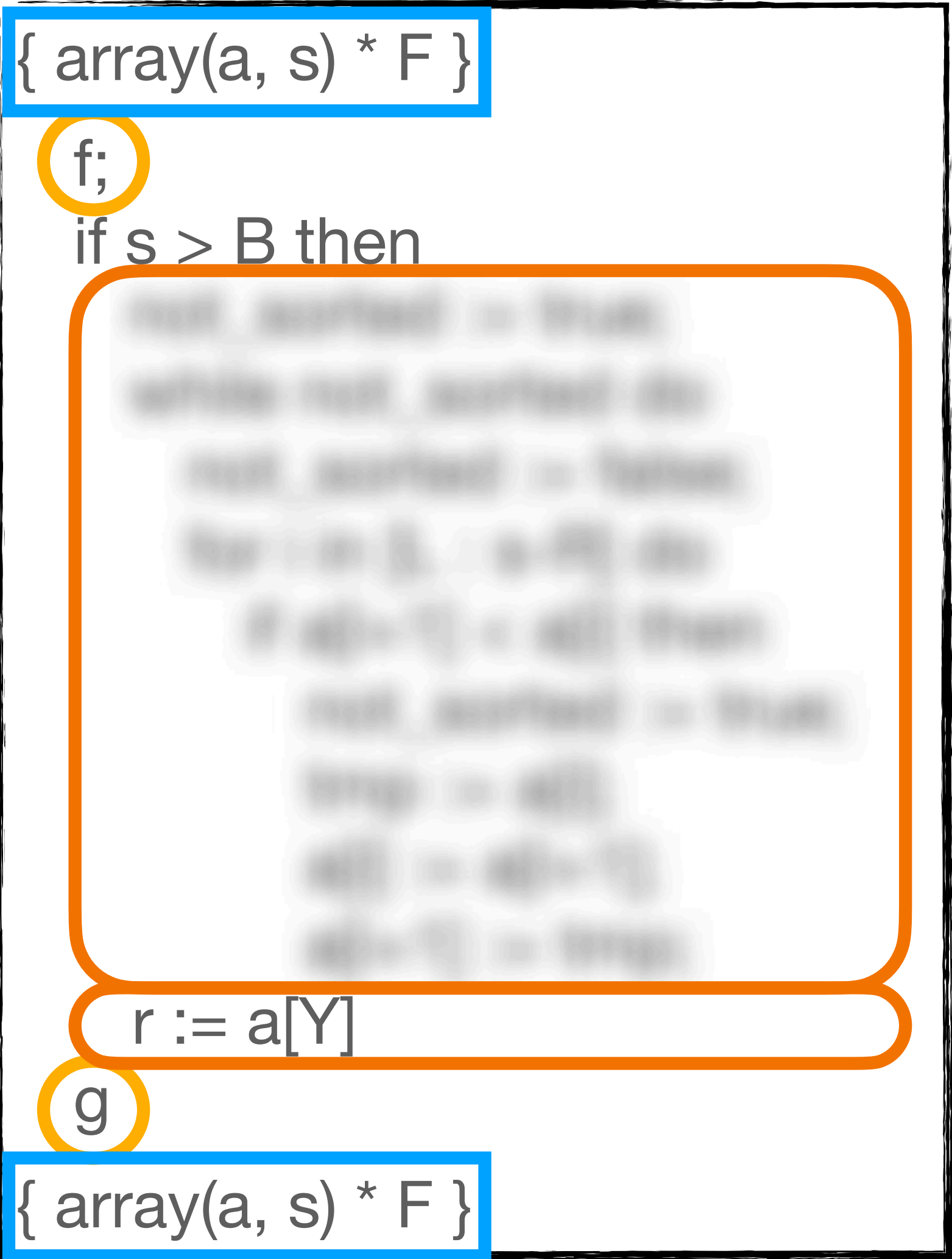


# Scalability

## Program Slicing



# Program Slicing In Action



# Program Slicing In Action

{ array(a, s) \* F }

f;

if s > B then

```
not_sorted := true;
while not_sorted do
  not_sorted := false;
  for i in [L : s-R] do
    if a[i+1] < a[i] then
      not_sorted := true;
      tmp := a[i];
      a[i] := a[i+1];
      a[i+1] := tmp;
```

r := a[Y]

g

{ array(a, s) \* F }

Precondition

Complex code not mentioning a, s

bubble\_sort(a)

Select element

Complex code not mentioning a, s

Postcondition

# Program Slicing In Action

~~{ array(a, s) \* }~~

~~X~~

if s > B then

~~not s > B = true;~~

~~while s > B do~~

~~not s > B = false;~~

for i in [L : s-R] do

if a[i+1] < a[i] then

~~not s > B = true;~~

~~X~~ t := a[i];

a[i] := a[i+1];

a[i+1] := t; ~~X~~

~~X~~

:= a[Y]

~~{ array(a, s) \* }~~

Precondition

~~Complex condition mentioning a, s~~

bubble\_sort(a)

Select element

~~Complex condition mentioning a, s~~

Postcondition

# Program Slicing In Action

```
{ array(a, s) }  
  if s > B then  
    for i in [L : s-R] do  
      if a[i+1] < a[i] then  
        a[i];  
        a[i] := a[i+1];  
        a[i+1];  
      a[Y]  
{ array(a, s) }
```

# Further Refinement via Static Analysis

```
{ array(a, s) }  
  if s > B then  
    for i in [L : s-R] do  
      if a[i+1] < a[i] then  
        a[i];  
        a[i] := a[i+1];  
        a[i+1];  
      a[Y]  
    { array(a, s) }
```

Comparison does not  
depend on s

⇒ Can ignore “if”

# Further Refinement via Static Analysis

```
{ array(a, s) }  
  if s > B then  
    for i in [L : s-R] do  
      a[i+1]; a[i]  
      a[i];  
      a[i] := a[i+1];  
      a[i+1];  
      a[Y]  
{ array(a, s) }
```

Comparison does not  
depend on s

⇒ Can ignore “if”

# Further Refinement via Static Analysis

```
{ array(a, s) }  
  if s > B then  
    for i in [L : s-R] do  
      a[i+1]; a[i]  
      a[i];  
      a[i] := a[i+1];  
      a[i+1];  
    a[Y]  
{ array(a, s) }
```

Subsumed by  $a[i+Z]$



# Further Refinement via Static Analysis

```
{ array(a, s) }  
  if s > B then  
    for i in [L : s-R] do  
      a[i+Z]  
      a[Y]  
{ array(a, s) }
```

Subsumed by a[i+Z]

# Scalability

## CT Combinators

Sequencing

$c_1; c_2$

CTs  $Q_1, Q_2$



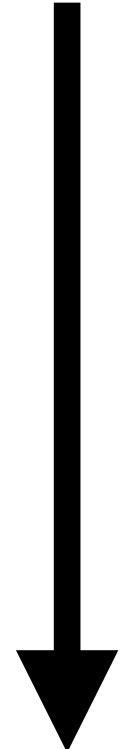
$$Q = Q_1 \cup Q_2$$

# Scalability

## CT Combinators

Sequencing

$c_1; c_2$

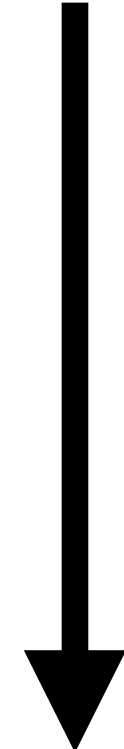


$$Q = Q_1 \cup Q_2$$

CTs  $Q_1, Q_2$

Branching

if  $e$  then  $c_1$  else  $c_2$



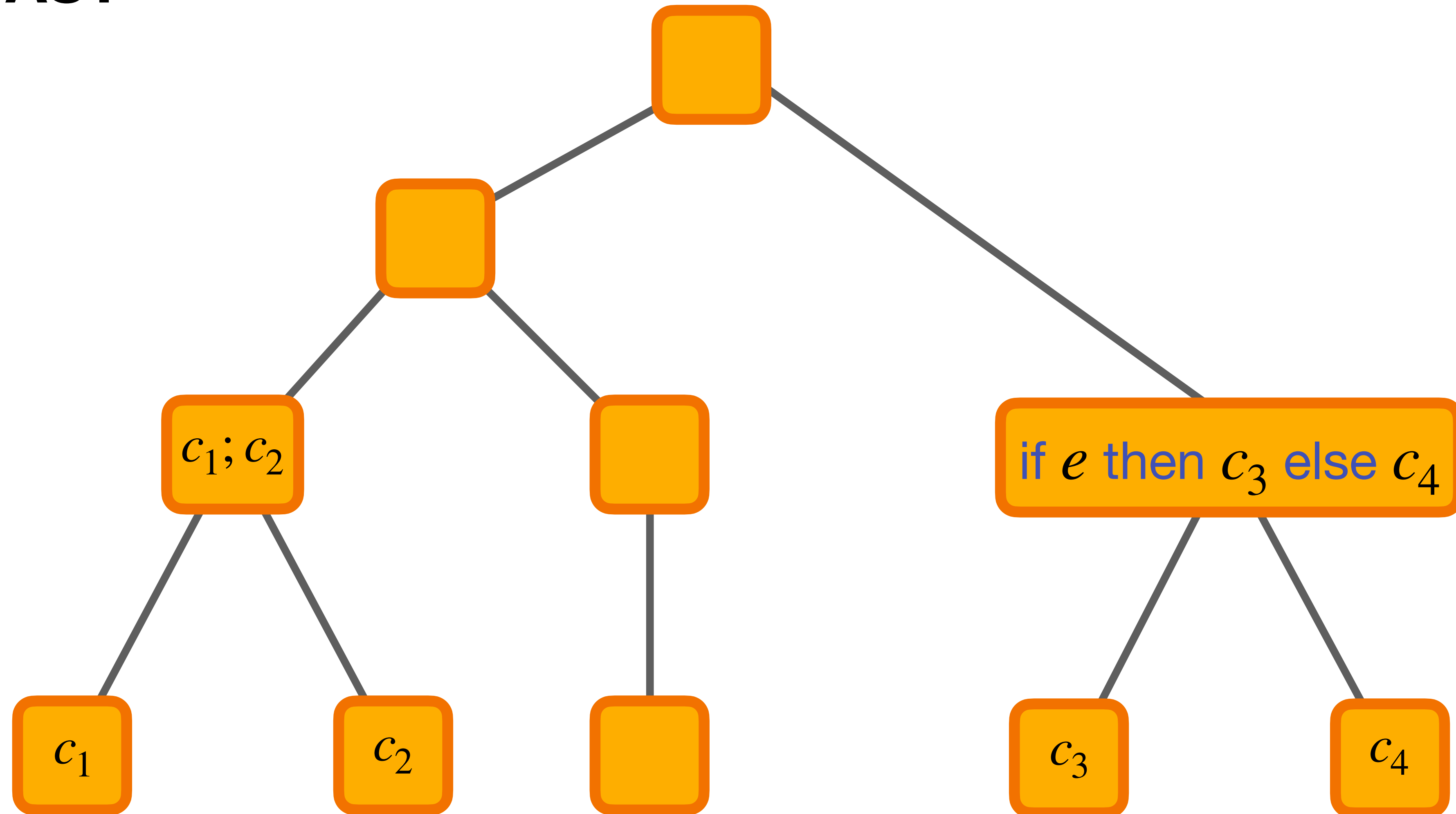
$$Q \sim (e \wedge K_1) \cup (\neg e \wedge K_2)$$

CTs as constraint sets

$$Q_i \sim K_i$$

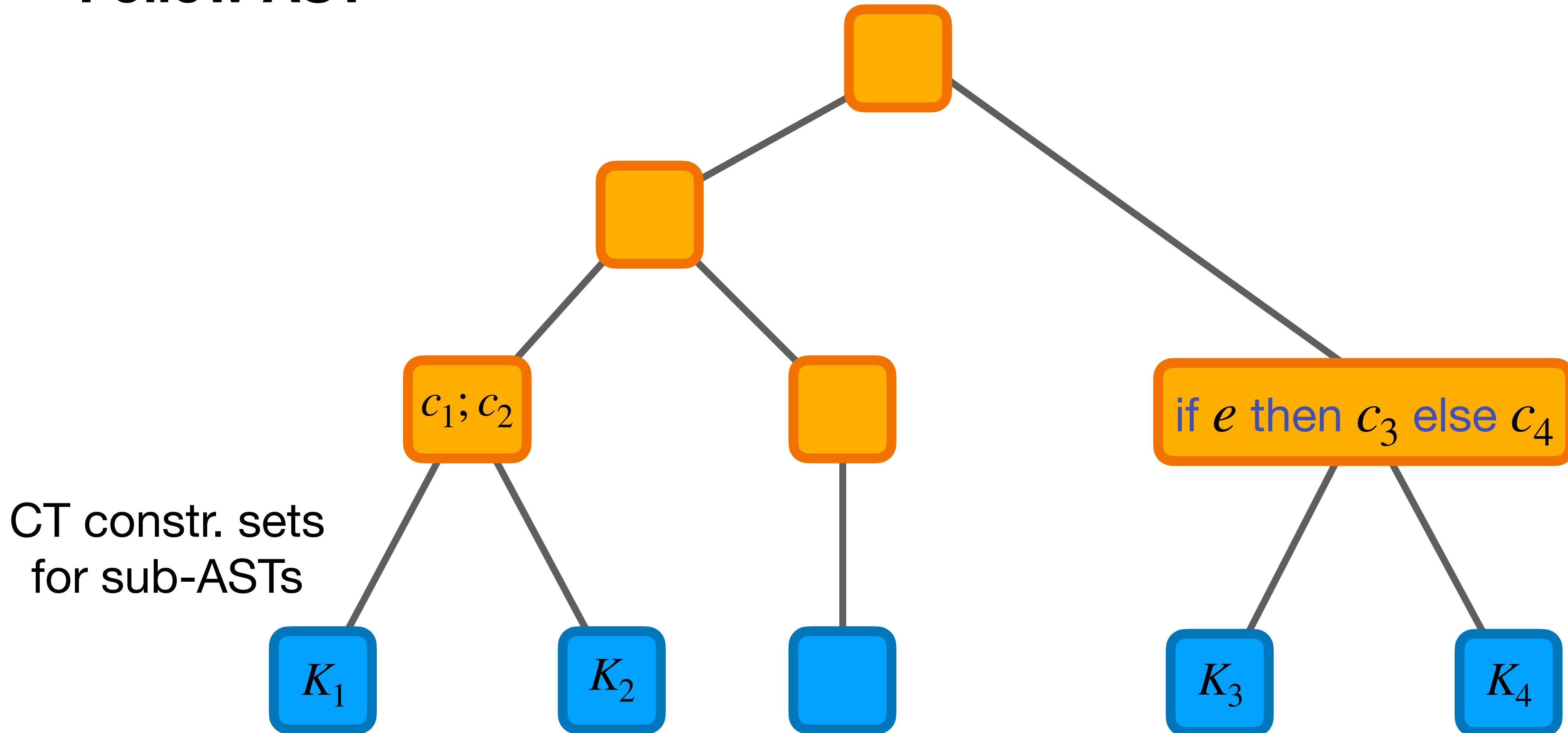
# Scalability

Follow AST



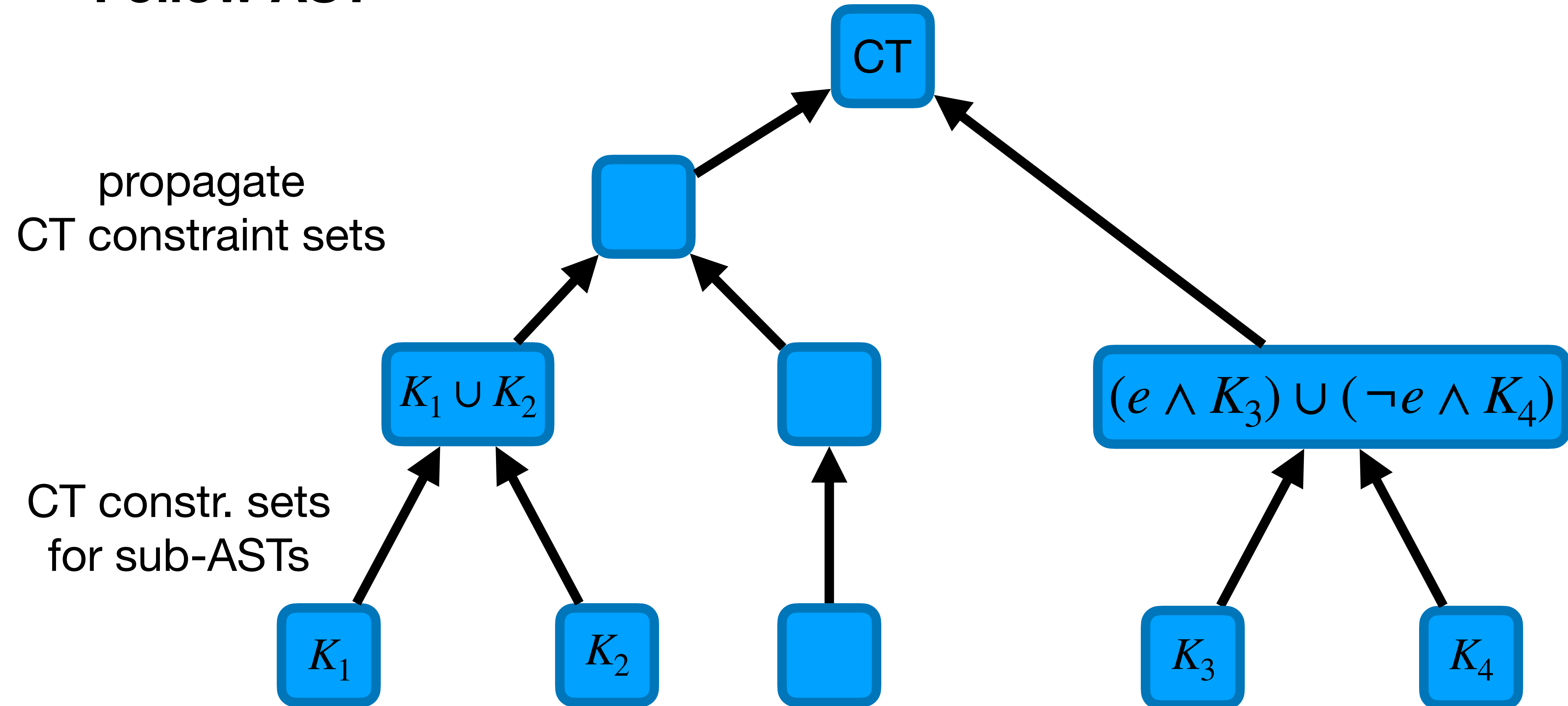
# Scalability

Follow AST



# Scalability

## Follow AST

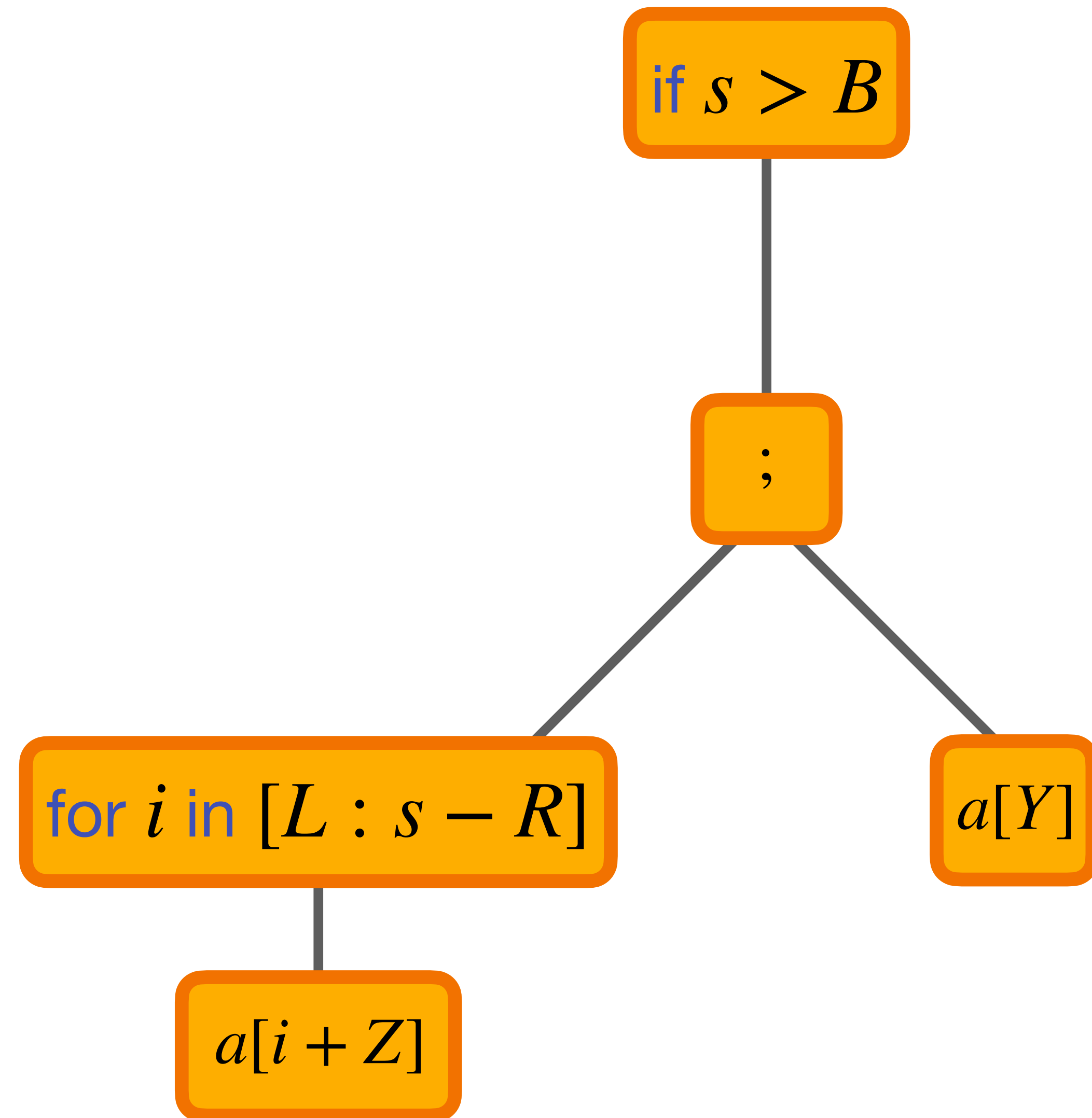


# CT Propagation In Action

```
{ array(a, s) }  
  if s > B then  
    for i in [L : s-R] do  
      a[i+Z]  
      a[Y]  
{ array(a, s) }
```

# CT Propagation In Action

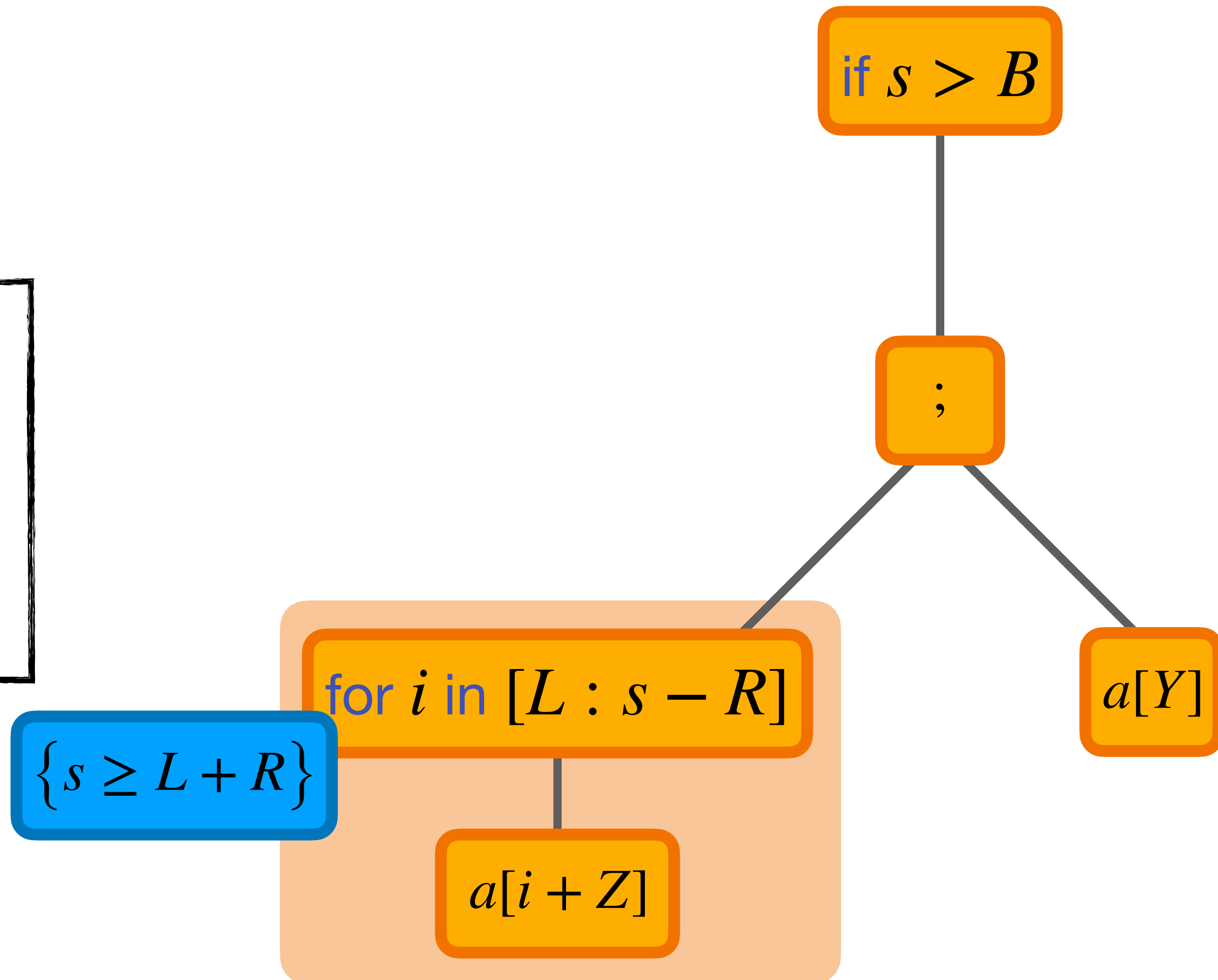
```
{ array(a, s) }  
  if s > B then  
    for i in [L : s-R] do  
      a[i+Z]  
      a[Y]  
{ array(a, s) }
```





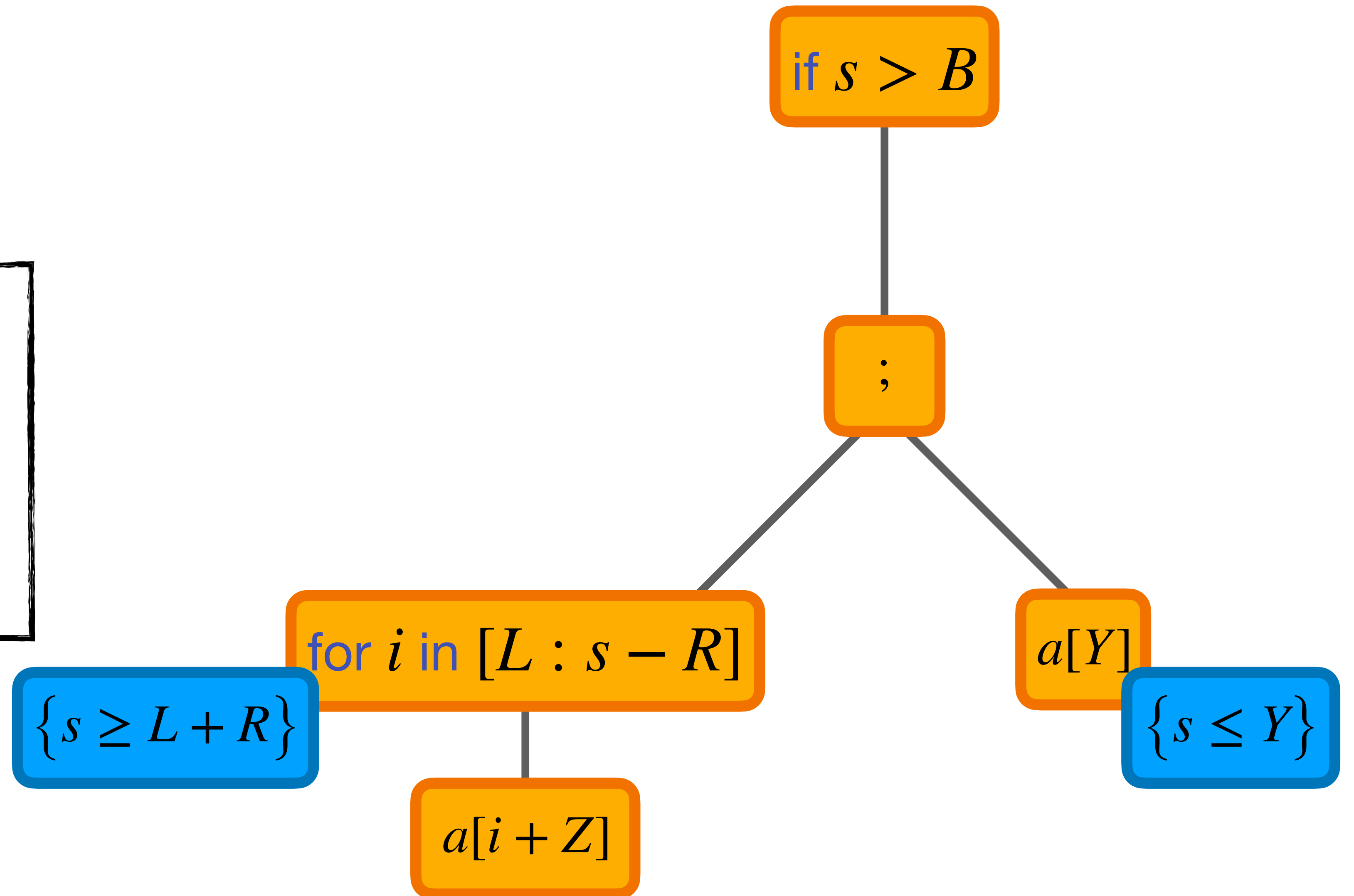
# CT Propagation In Action

```
{ array(a, s) }  
  if s > B then  
    for i in [L : s-R] do  
      a[i+Z]  
      a[Y]  
{ array(a, s) }
```



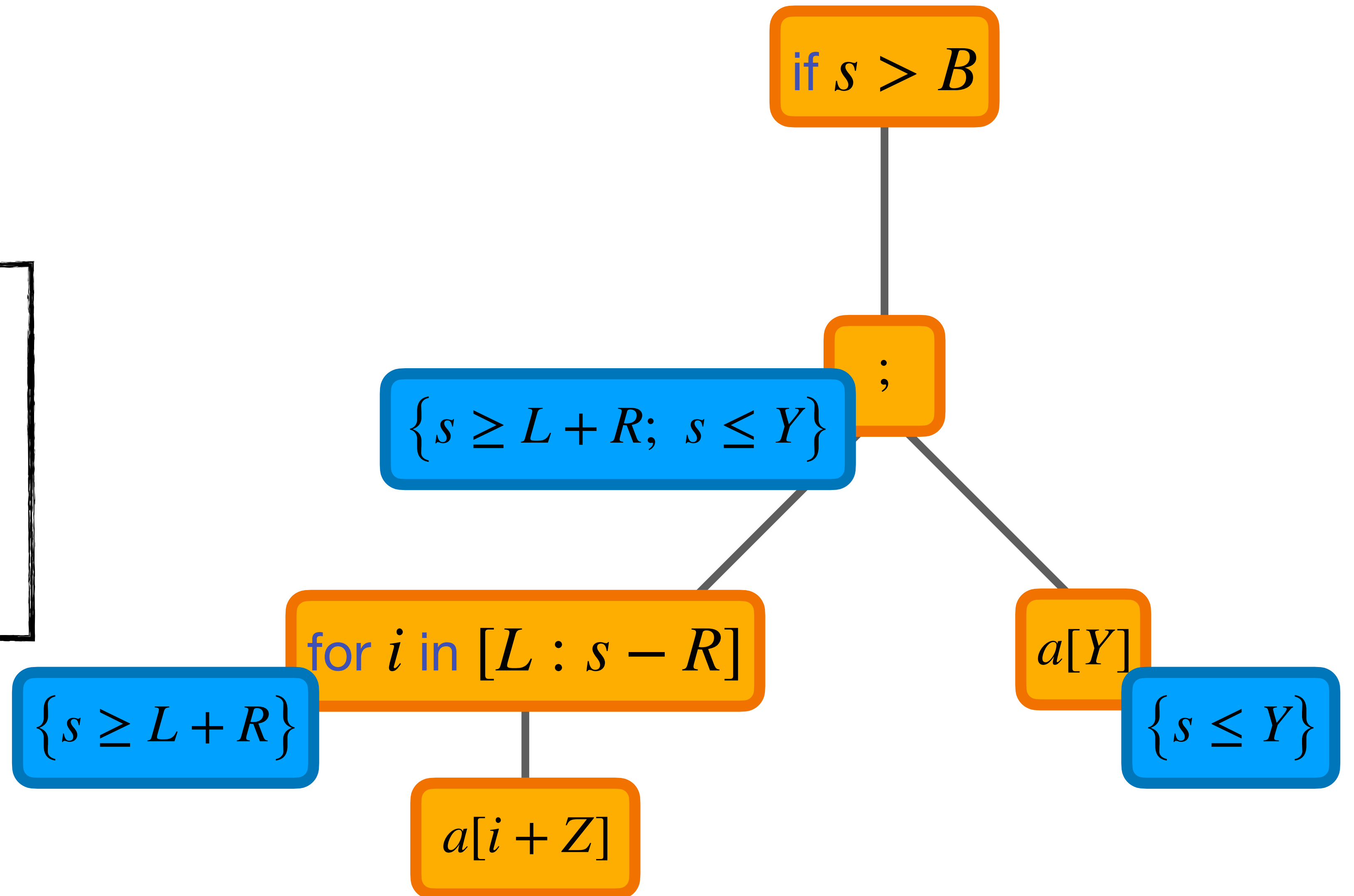
# CT Propagation In Action

```
{ array(a, s) }  
  if s > B then  
    for i in [L : s-R] do  
      a[i+Z]  
      a[Y]  
{ array(a, s) }
```



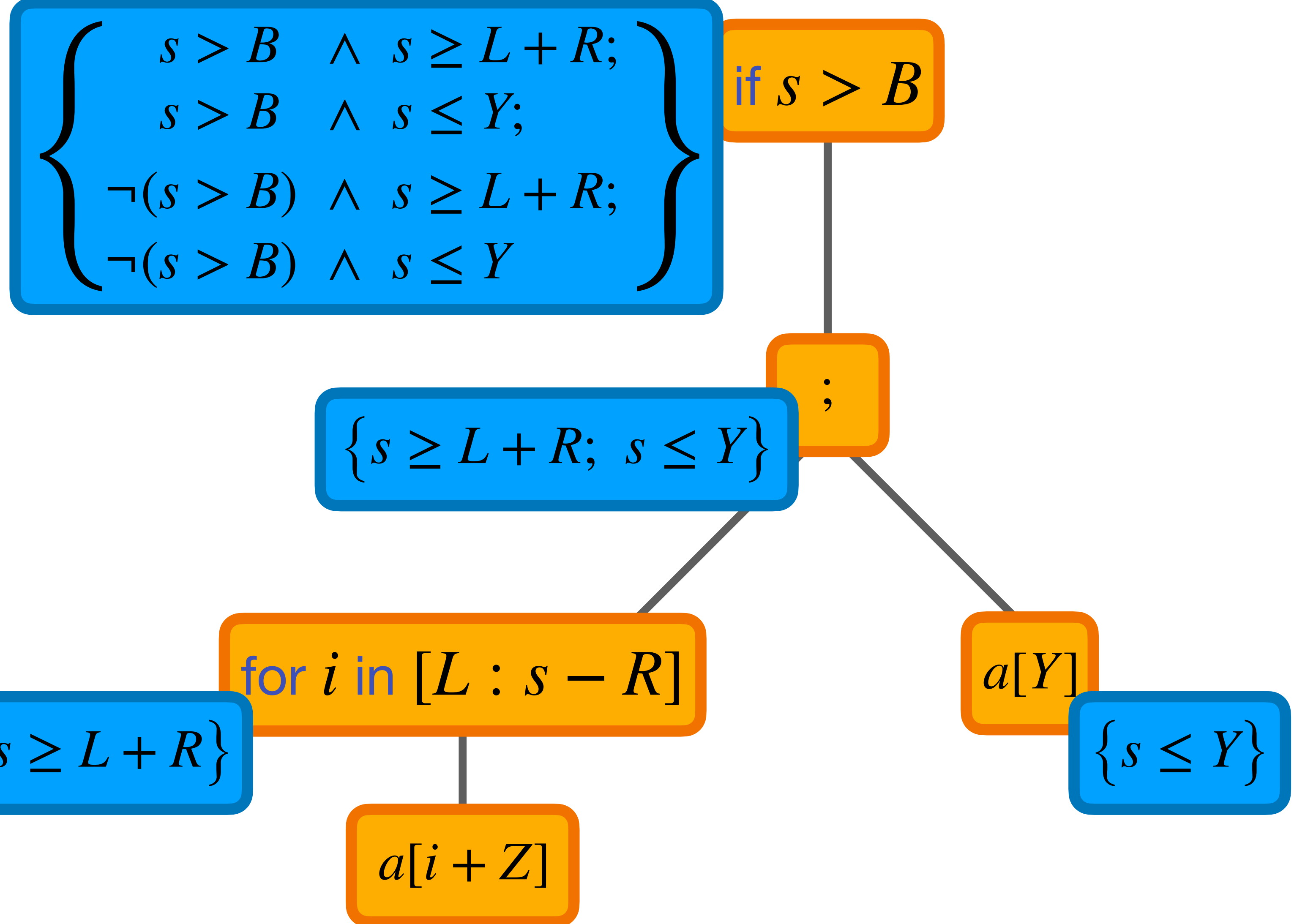
# CT Propagation In Action

```
{ array(a, s) }  
  if s > B then  
    for i in [L : s-R] do  
      a[i+Z]  
      a[Y]  
{ array(a, s) }
```



# CT Propagation In Action

```
{ array(a, s) }  
  if s > B then  
    for i in [L : s-R] do  
      a[i+Z]  
      a[Y]  
{ array(a, s) }
```



# Outlook: Plans & Challenges

## Plans

- Demo scalability: Complex programs & data (e.g. lists, trees)
- Evaluate CT's impact on runtime:  
⇒ Case study: FreeRTOS' TCP stack
- Generalise CTs to arbitrary correctness properties

# Outlook: Plans & Challenges

## Plans

- Demo scalability: Complex programs & data (e.g. lists, trees)
- Evaluate CT's impact on runtime:  
⇒ Case study: FreeRTOS' TCP stack
- Generalise CTs to arbitrary correctness properties



# Outlook: Plans & Challenges

## Plans

- Demo scalability: Complex programs & data (e.g. lists, trees)
- Evaluate CT's impact on runtime:  
⇒ Case study: FreeRTOS' TCP stack
- Generalise CTs to arbitrary correctness properties



## Challenge: Automation

- Pattern recognition
- automatic VC rewriting

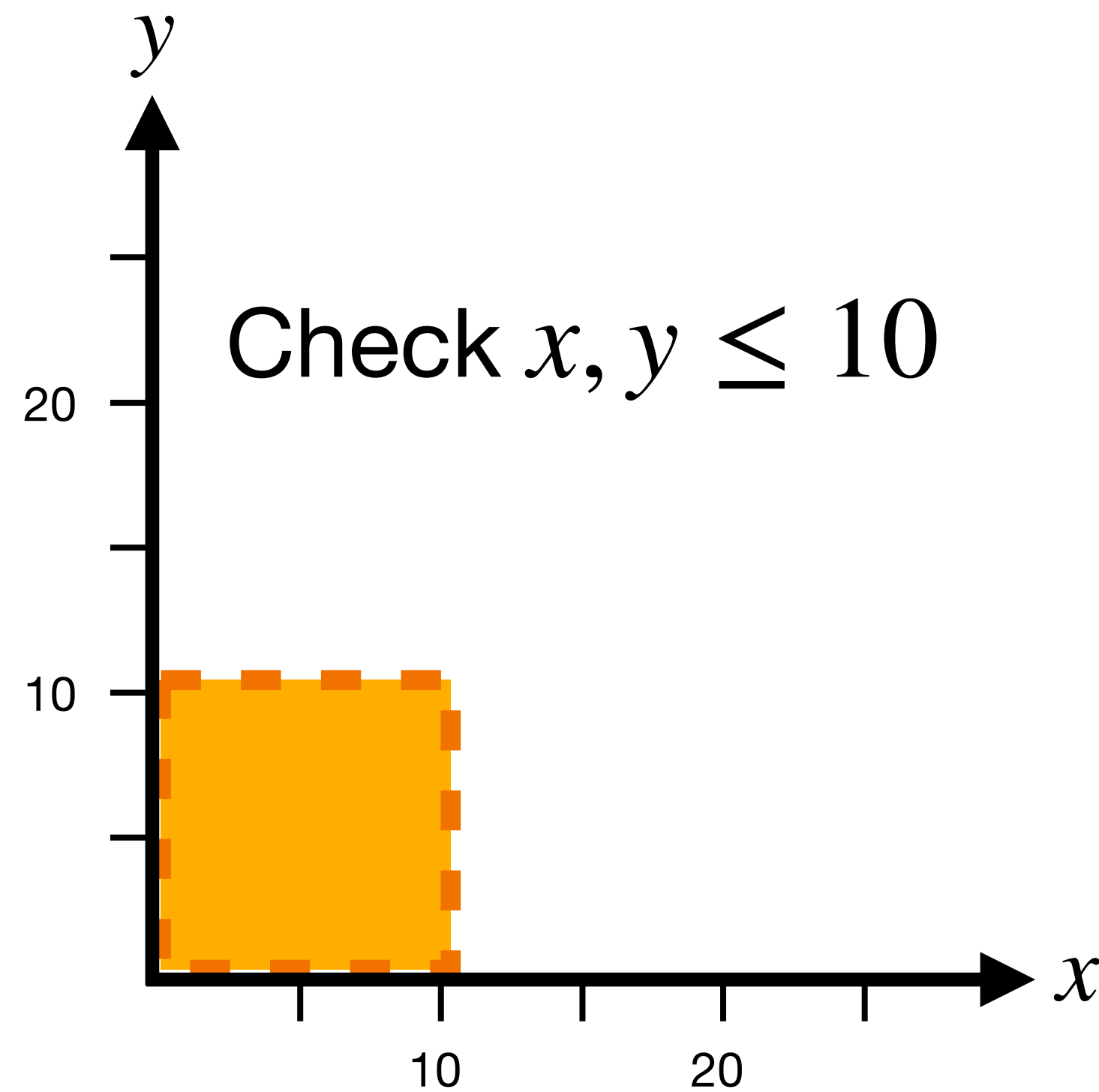
# Outlook: Increase Trust in BMC

- Turn bounded into unbounded proof



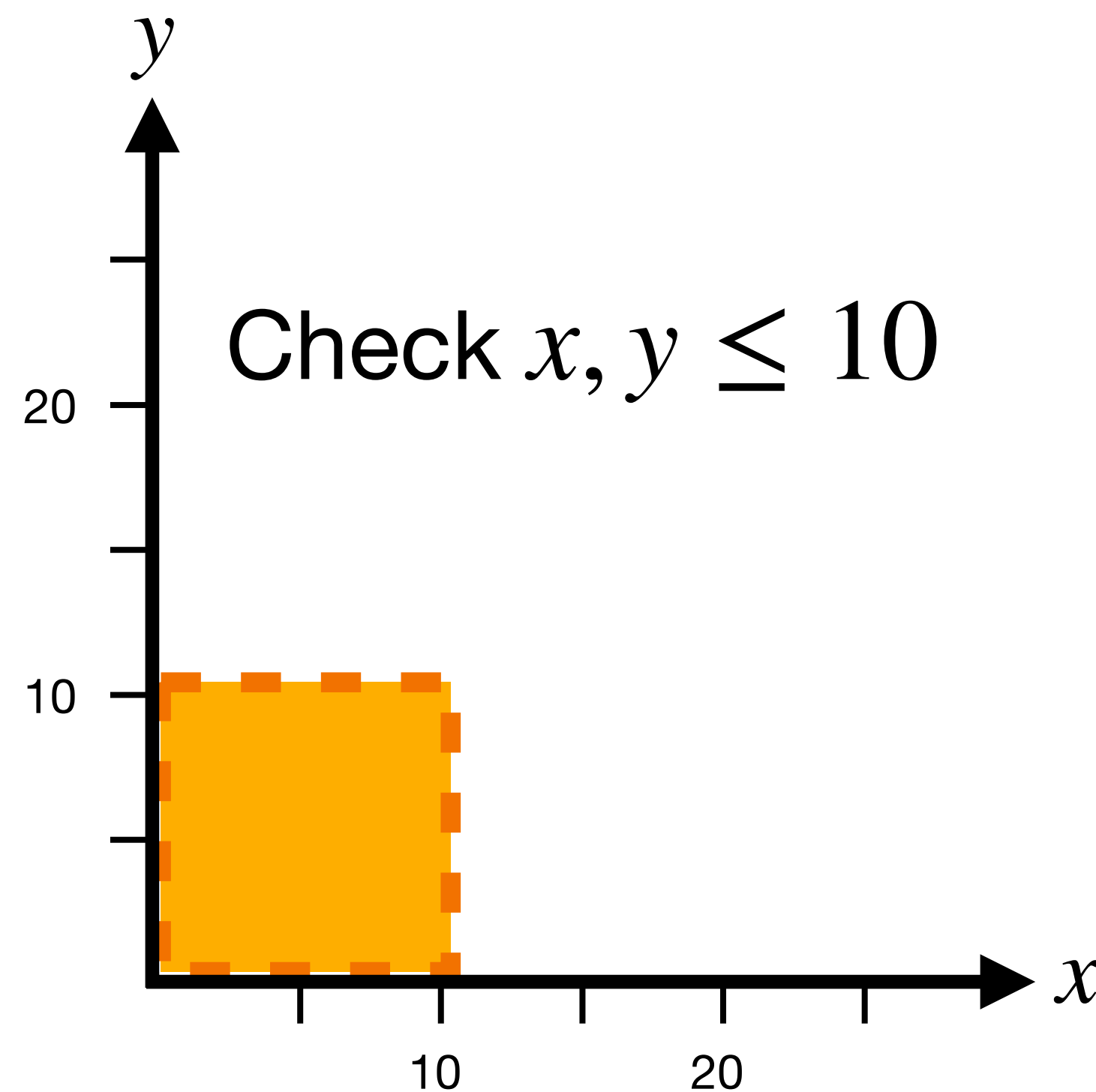
# Outlook: Increase Trust in BMC

- Turn bounded into unbounded proof
- Shift resources to critical bounds

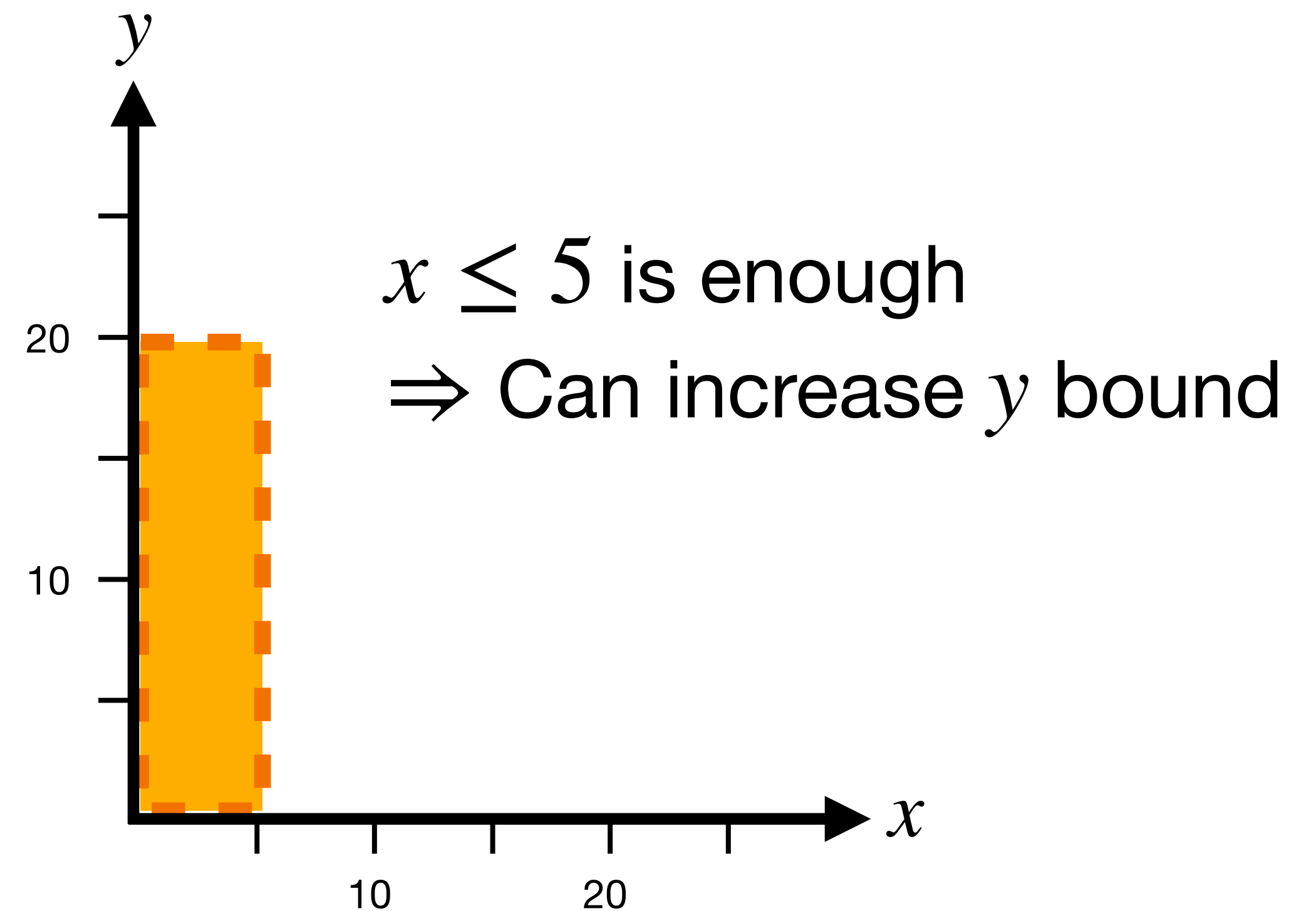
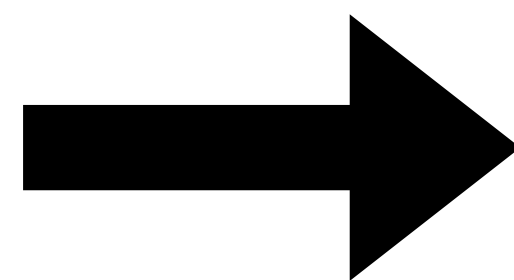


# Outlook: Increase Trust in BMC

- Turn bounded into unbounded proof
- Shift resources to critical bounds



CT for  $x$ :  
 $\{0, \dots, 5\}$



# Conclusion

- First generalisation of CTs to infinite state systems
- Connection between bounded & unbounded proofs in program verification
- Foundational research but potential for integration into BMC



# Backup Slides

# Precise VCs

- VC  $vc$  is *precise* for  $x$  in  $Spec$  iff

$$\forall v. \left( \models Spec[x \mapsto v] \Rightarrow \models vc[x \mapsto v] \right)$$

Intuition:  $vc$  does not over-approximate wrt.  $x$

- $Q$  is CT  $vc \wedge vc$  is precise  $\Rightarrow Q$  is CT  $Spec$

# Precise VCs

