Completeness Thresholds for Memory Safety: Unbounded Guarantees via Bounded Proofs

Tobias Reinhard, Justus Fasse, Bart Jacobs
KU Leuven

Programming Languages and Verification Seminar
Portland State University
November 30, 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?

Focus: Traversing Programs

- Target property: Memory safety
- Programs that:
  - Traverse a data structure
  - Preserve its memory layout
- But approach & many results are very general

Model Checking: Easy Off-by-1 Error

- Imperative language with pointer arithmetic
- Memory assumption $\texttt{array}(a, s)$:
  $a[0] \ \ldots \ a[s-1]$ allocated

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

Model Checking: Easy Off-by-1 Error

- Imperative language with pointer arithmetic
- Memory assumption $\texttt{array}(a, s)$:
  $a[0] \ \ldots \ a[s-1]$ 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:
$\texttt{array}(a, s)$

```
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:
$\texttt{array}(a, s)$

```
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:
$\texttt{array}(a, s)$

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

Which s can convince us?

Model Checking: No Off-by-N Error

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

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

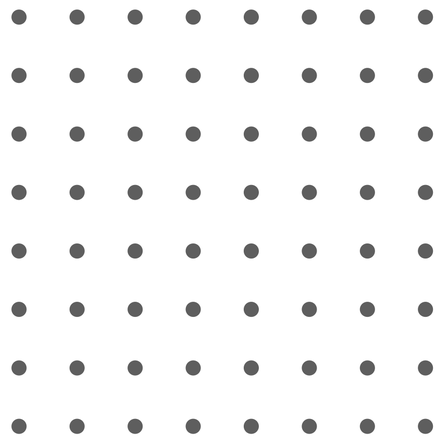
Which s can convince us?

- $s = 0$: No error
- $s = 1$: No error
- $s = 2$: No error
- $s = 3$: No error
- $\vdots$

$\Rightarrow$ Which size bound is large enough?

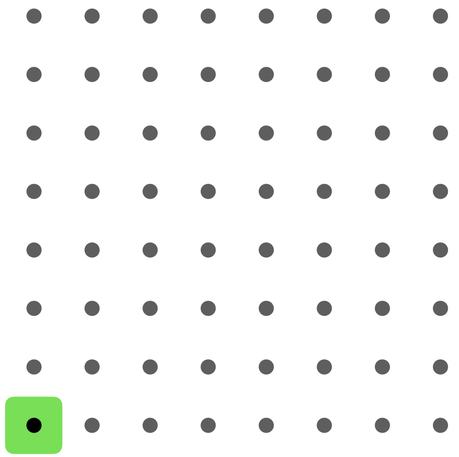
Model Checking Finite Systems

- 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

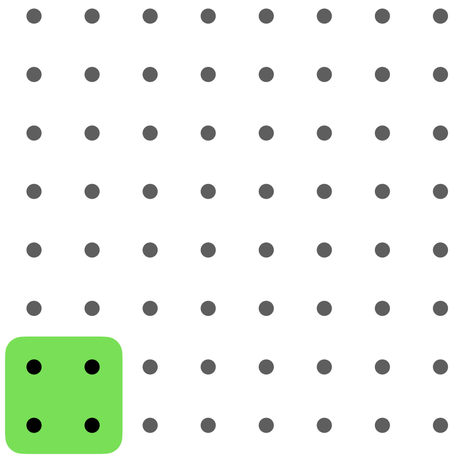
- Finite state transition system T
- Prove property Gp
  G $\approx$ globally $\approx$ p holds in every state
- Approach:
  Prove Gp for all paths up to length k
  $T \models_k \mathsf{G}p$



$T \models_0 \mathsf{G}p$

Model Checking Finite Systems

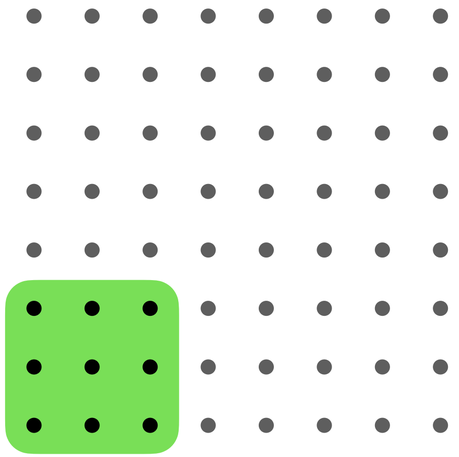
- Finite state transition system T
- Prove property Gp
  G $\approx$ globally $\approx$ p holds in every state
- Approach:
  Prove Gp for all paths up to length k
  $T \models_k \mathsf{G}p$



$T \models_1 \mathsf{G}p$

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
  $T \models_k \mathsf{G}p$



$T \models_2 \mathsf{G}p$

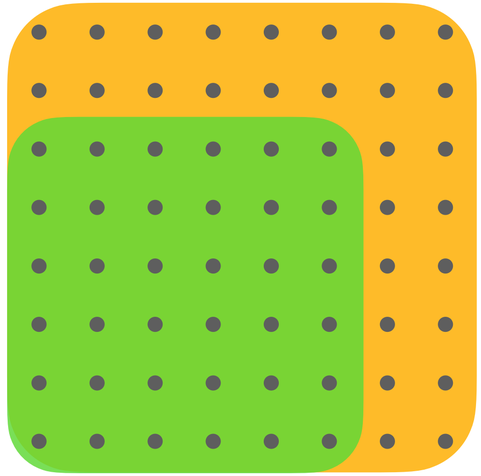
When should we stop?

Completeness Thresholds for Finite Systems

- k is completeness thresholds (CT) iff

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

- For specific ϕ:
  Can over-approximate CT via of key props of T



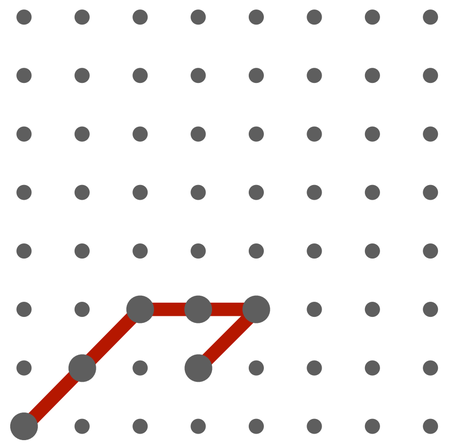
Completeness Thresholds for Finite Systems

- k is completeness thresholds (CT) iff

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

- For specific ϕ:
  Can over-approximate CT via of key props of T

- For $\phi = \mathsf{G}p$ we know
  $\mathrm{CT}(T, \mathsf{G}p) = \mathrm{diameter}(T)$
  (longest distance between any states)



$\mathrm{diameter}(T) = 5$

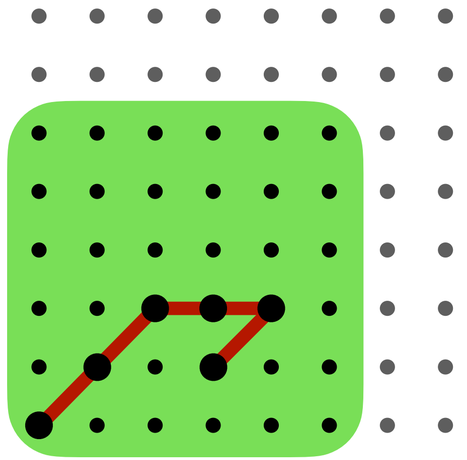
Completeness Thresholds for Finite Systems

- k is completeness thresholds (CT) iff

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

- For specific ϕ:
  Can over-approximate CT via of key props of T

- For $\phi = \mathsf{G}p$ we know
  $\mathrm{CT}(T, \mathsf{G}p) = \mathrm{diameter}(T)$
  (longest distance between any states)



$\mathrm{diameter}(T) = 5$

$T \models_5 \mathsf{G}p$

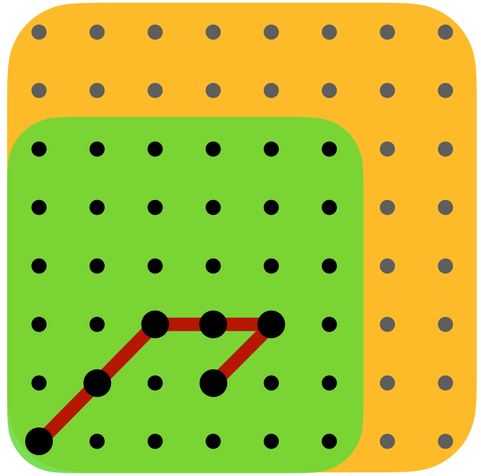
Completeness Thresholds for Finite Systems

- k is completeness thresholds (CT) iff

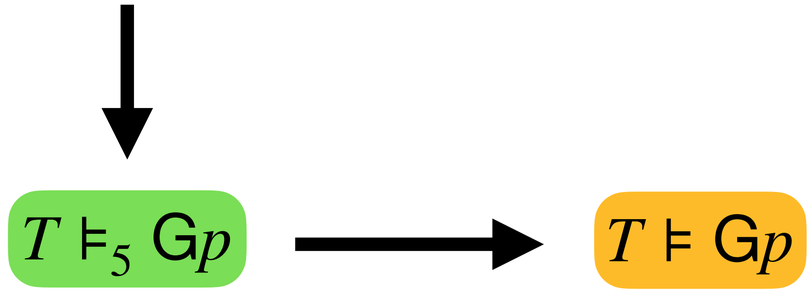
$$T \models_k \phi \;\Rightarrow\; T \models \phi$$

- For specific ϕ:
  Can over-approximate CT via of key props of T

- For $\phi = \mathsf{G}p$ we know
  $\mathrm{CT}(T, \mathsf{G}p) = \mathrm{diameter}(T)$
  (longest distance between any states)



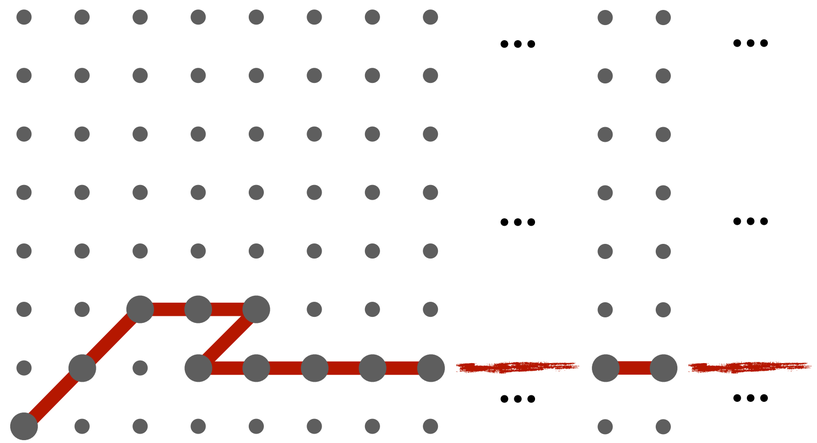
$\mathrm{diameter}(T) = 5$



CTs for Infinite Systems?

Problem

Key properties used to describe CTs may be ∞



diameter$(T) = \infty$

CTs for Infinite Systems?

Problem

Key properties used to describe CTs may be ∞

Our Approach

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

How Does the Array Size Affect Memory Safety?

Memory assumption:

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

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:	for i in [L : s-R] do
$\texttt{array}(a, s)$	!a[i+Z]

Range L, …, s-R empty?

Yes

$s^- < L + R$

No iteration

⇒ Memory safe

How Does the Array Size Affect Memory Safety?

Memory assumption:	for i in [L : s-R] do
$\texttt{array}(a, s)$	!a[i+Z]

Range L, …, s-R empty?

Yes

$s^- < L + R$

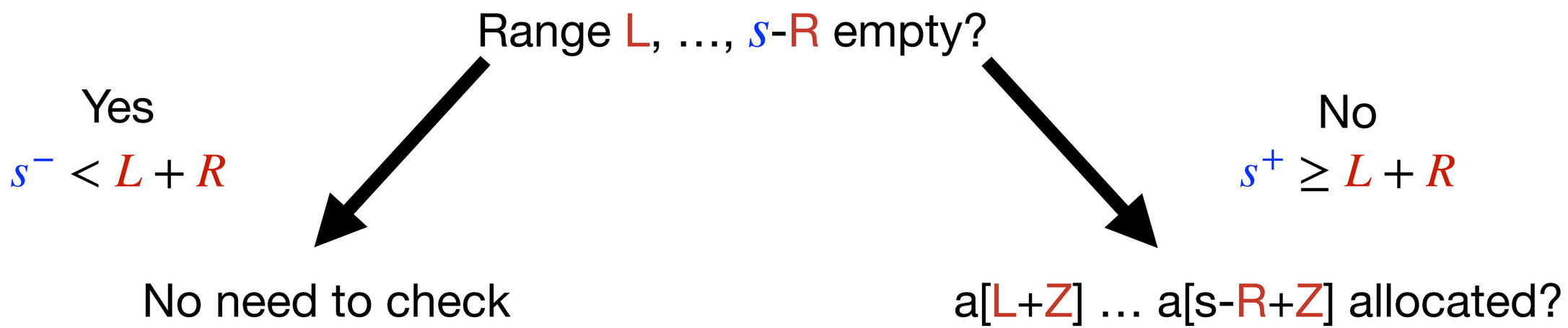
No need to check

How Does the Array Size Affect Memory Safety?

Memory assumption:

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

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



How Does the Array Size Affect Memory Safety?

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

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

Range L, …, s-R empty?

Yes
$s^- < L + R$

No need to check

No
$s^+ \geq L + R$

a[L+Z] … a[s-R+Z] allocated?

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

How Does the Array Size Affect Memory Safety?

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

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

Range L, …, s-R empty?

Yes
$s^- < L + R$

No need to check

No
$s^+ \geq L + R$

a[L+Z] … a[s-R+Z] allocated?

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

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

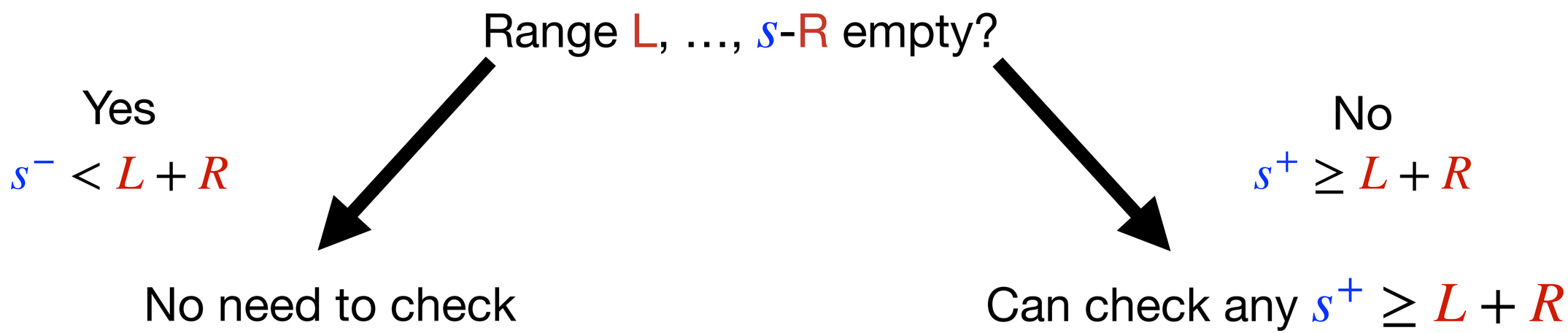
No $s^+ \Rightarrow$ Can check any $s^+ \geq L + R$

How Does the Array Size Affect Memory Safety?

Memory assumption:

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

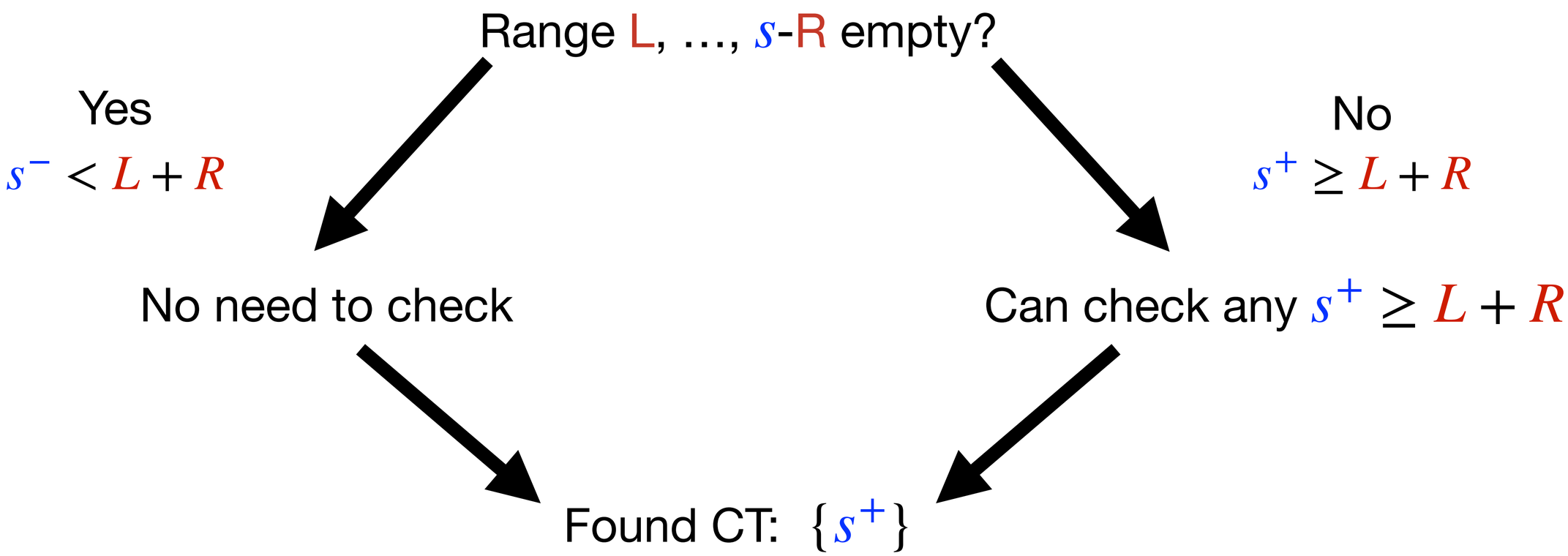
```
for i in [L : s-R] do
    !a[i+Z]
```



How Does the Array Size Affect Memory Safety?

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

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



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

$$\models \forall x \in Q \,.\, Spec(c) \quad \Rightarrow \quad \models \forall x \in X \,.\, Spec(c)$$

- For us: CT are subdomains, not depths

Verification Conditions

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

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

- 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 $vc(x)$

$$\left(\vDash vc(x) \quad \Leftrightarrow \quad \vDash vc' \quad \text{with } x \notin \text{free}(vc') \right)$$

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

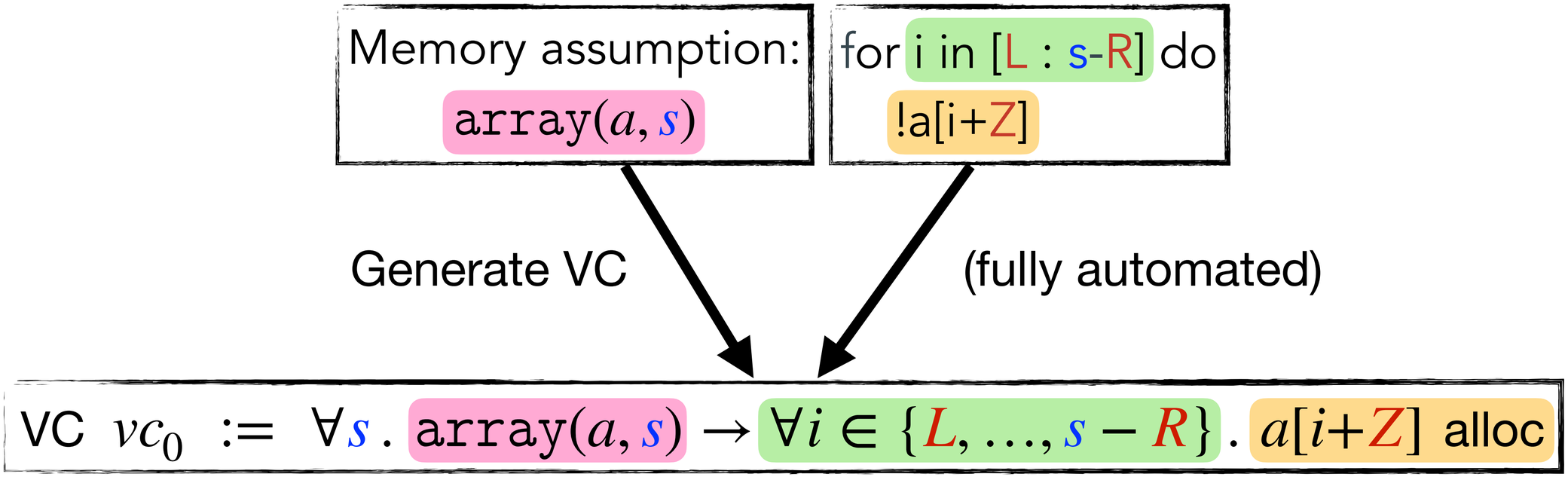
Proving CT in Action

Memory assumption:

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

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

Proving CT in Action



Proving CT in Action

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

Range L, …, s-R empty?

Proving CT in Action

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

Range L, …, s-R empty?

Yes

$s^- < L + R$

Simplify VC!

$$vc_0 \equiv \forall s^-.\ \ldots \to \forall i \in \varnothing.\ \ldots$$
$$\equiv \text{True}$$

Proving CT in Action

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

Range L, …, s-R empty?

Yes

$s^- < L + R$

Simplify VC!

No need to check

Proving CT in Action

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

Range L, …, 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

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

Range L, …, s-R empty?

Yes

$s^- < L + R$

No need to check

Simplify VC!

No

$s^+ \geq L + R$

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

$$\equiv \forall i\,.\,(L \leq i \rightarrow 0 \leq i{+}Z)$$

$$\wedge\ (i \leq -R) \rightarrow i{+}Z < 0)$$

$\Rightarrow$ Validity does not depend on size

Proving CT in Action

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

Range L, …, s-R empty?

Yes

$s^- < L + R$

No need to check

No

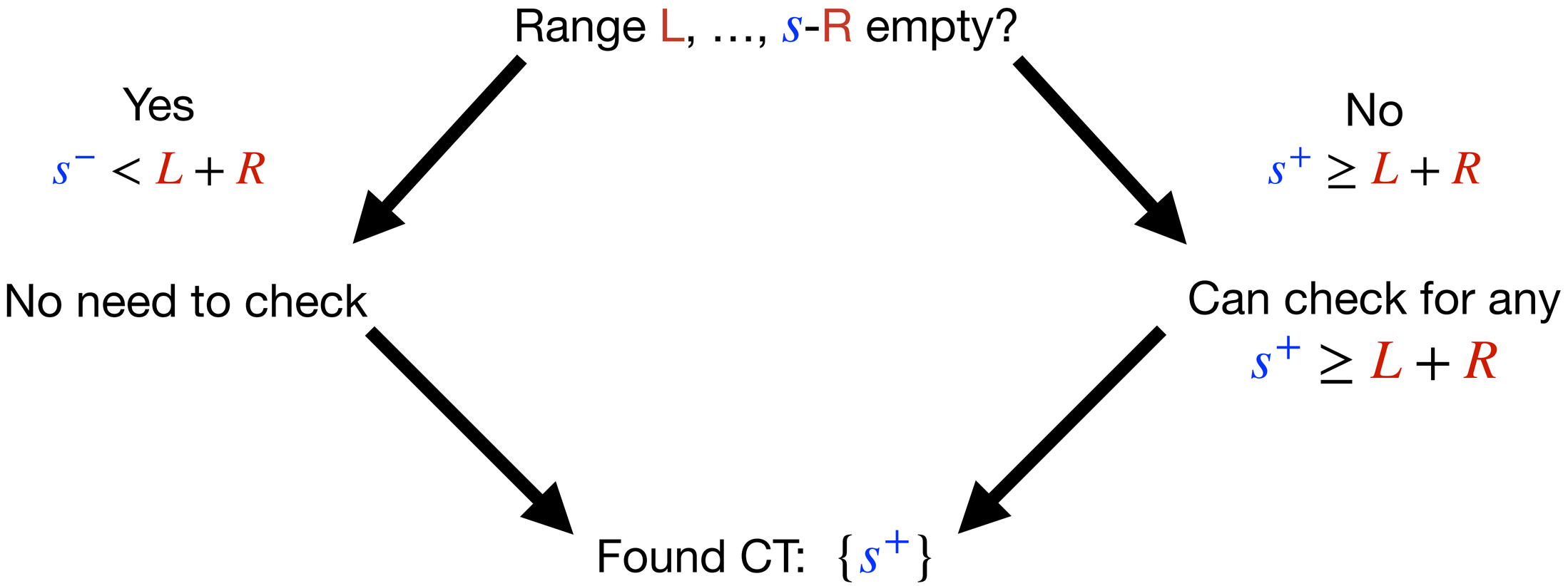
$s^+ \geq L + R$

Can check for any

$s^+ \geq L + R$

Proving CT in Action

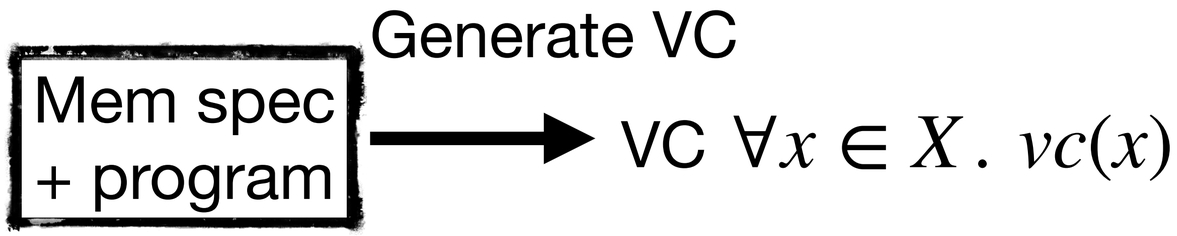
$$\text{VC } vc_0 := \forall s\,.\,\texttt{array}(a, s) \rightarrow \forall i \in \{L, \ldots, s - R\}\,.\,a[i{+}Z] \text{ alloc}$$



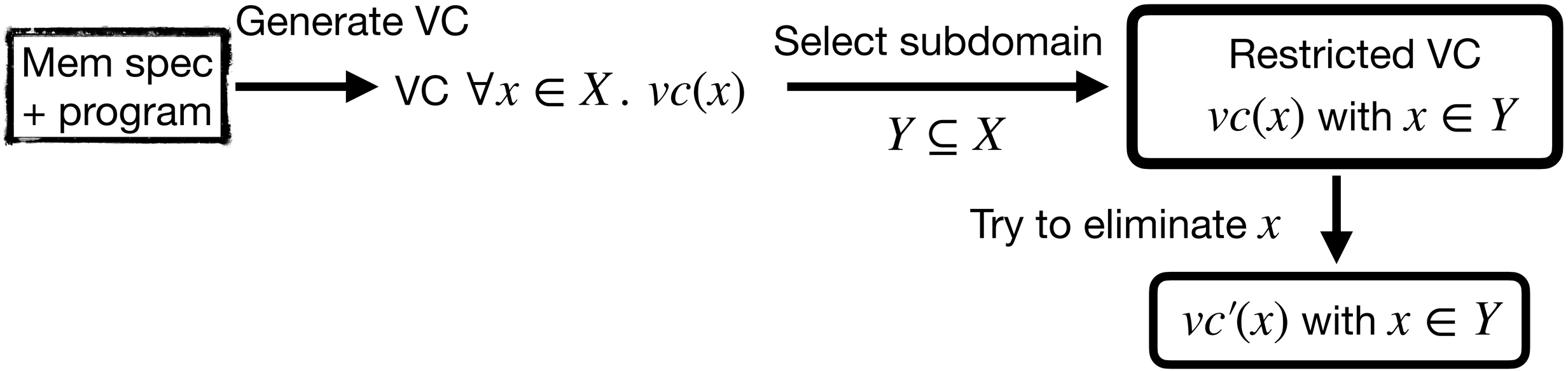
Workflow: How to Find CTs

Mem spec
+ program

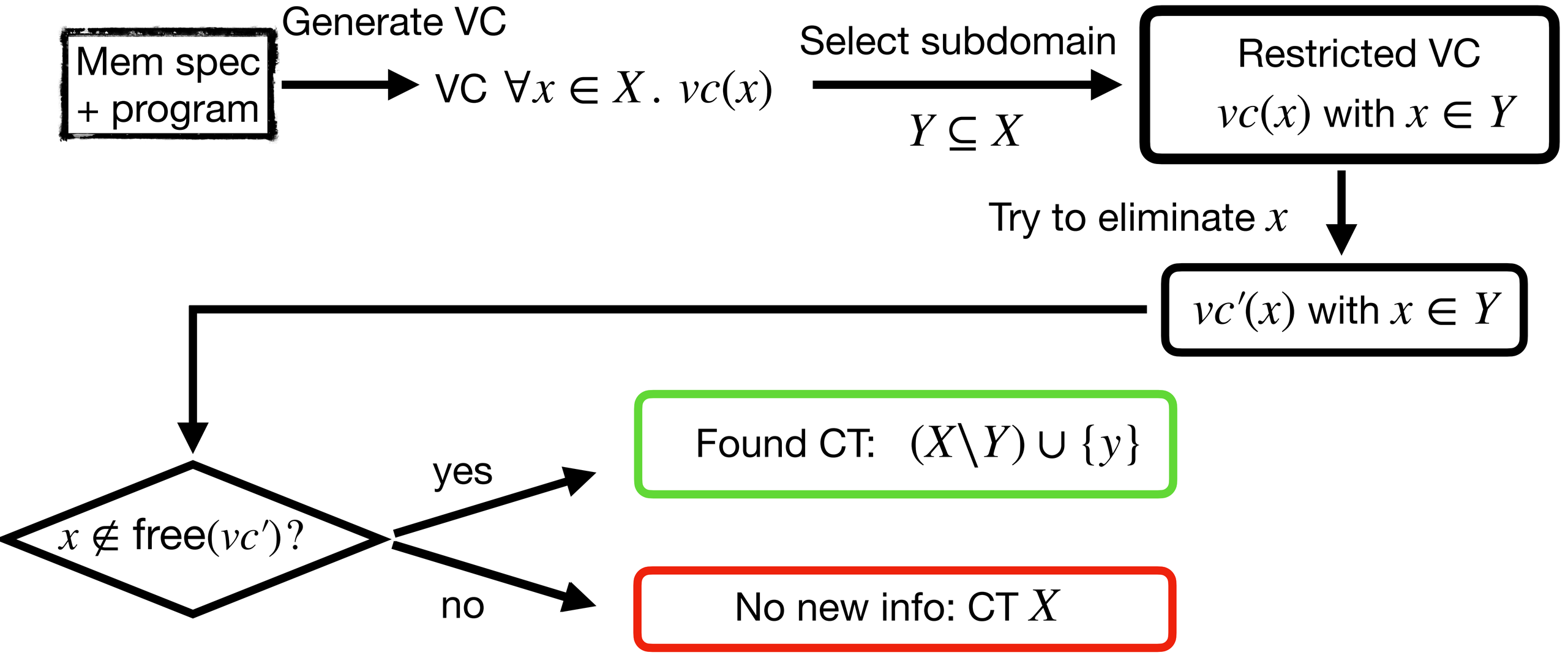
Workflow: How to Find CTs



Workflow: How to Find CTs

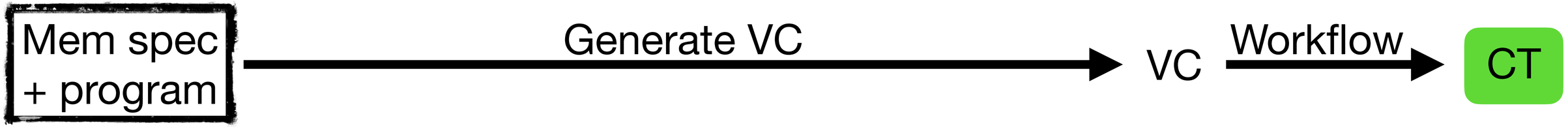


Workflow: How to Find CTs



Scalability

Program Slicing

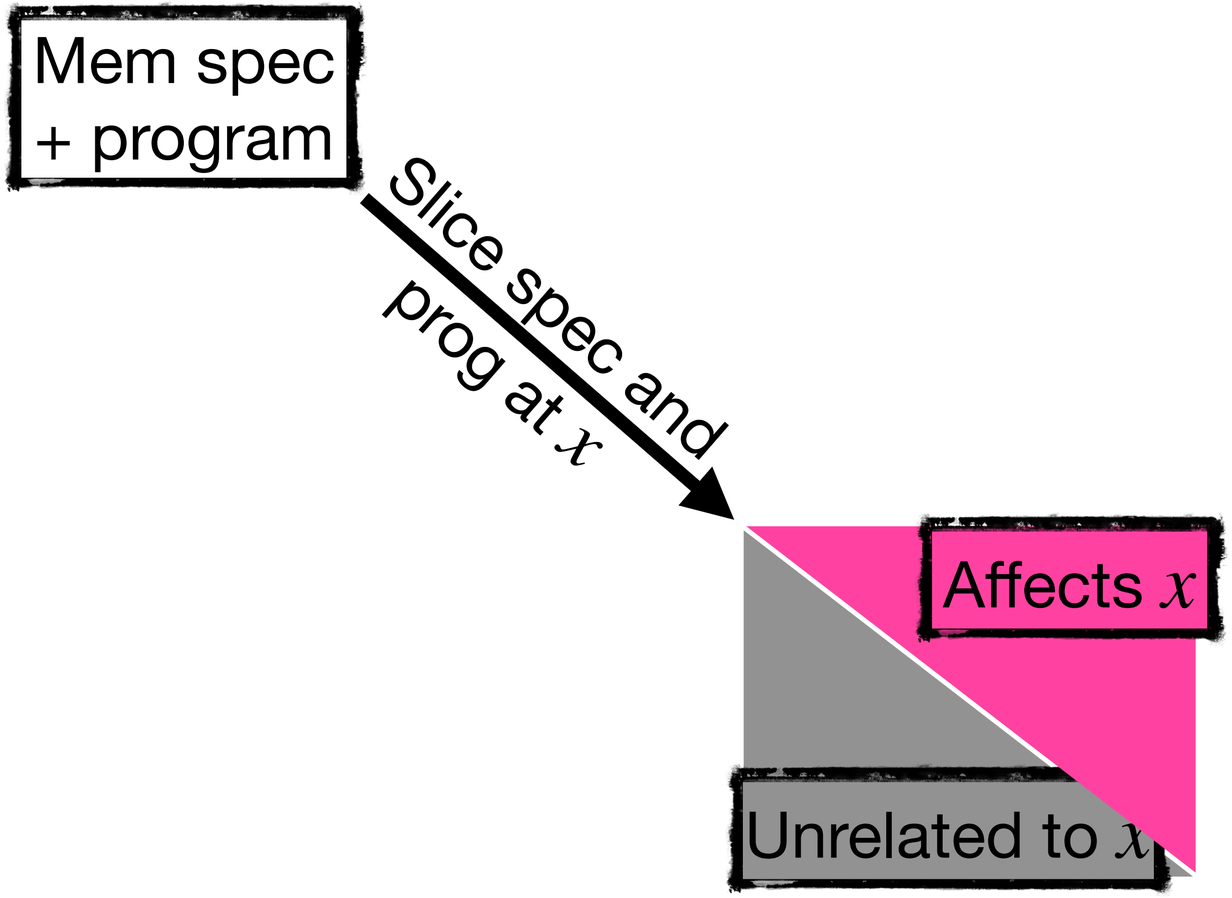


Scalability

Program Slicing

Scalability

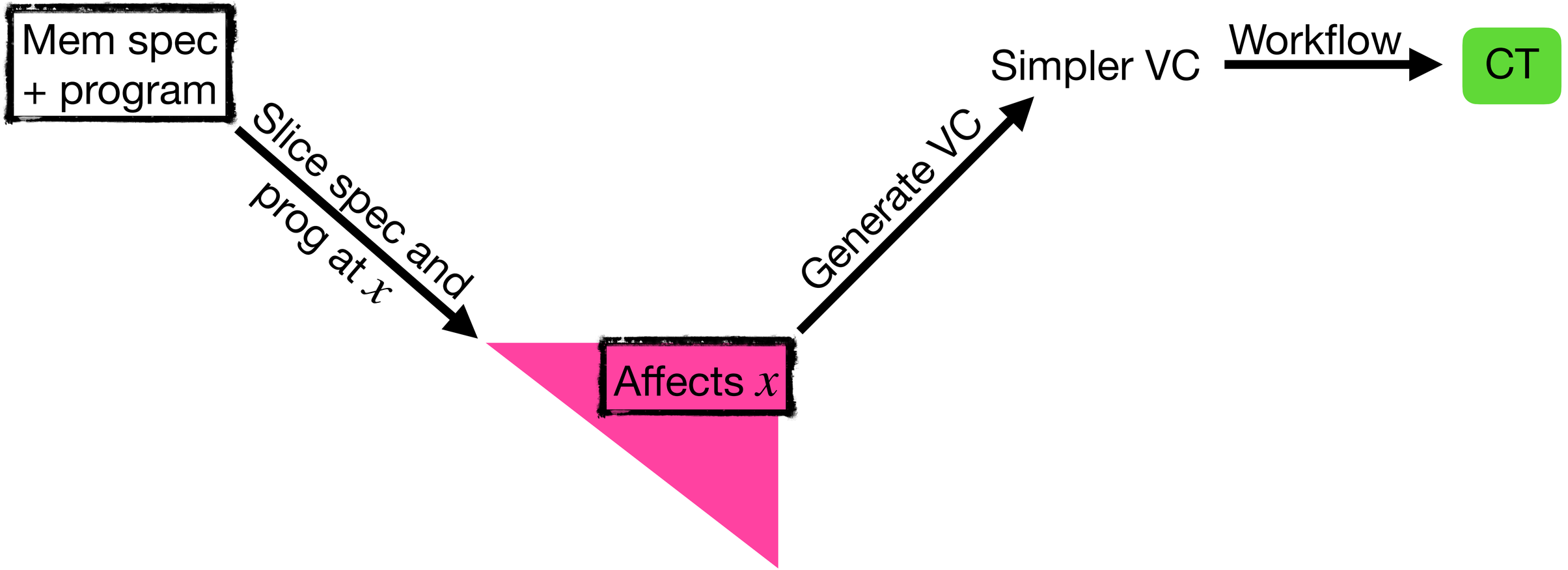
Program Slicing



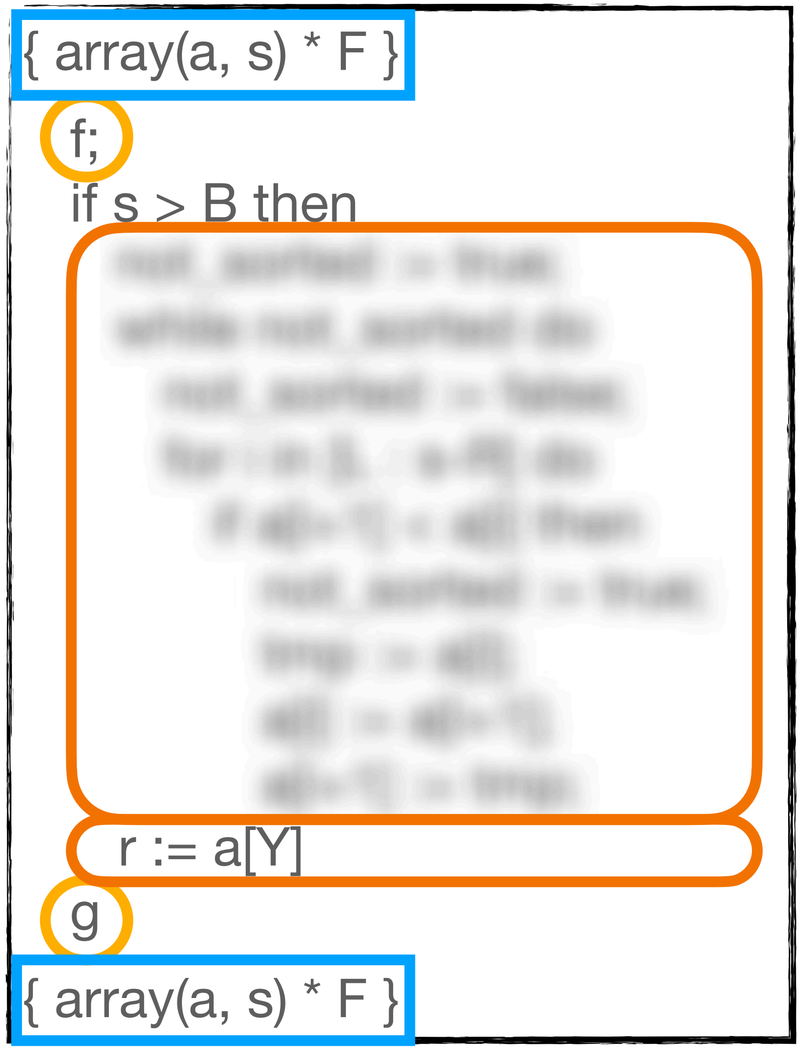
Simpler VC → Workflow → CT

Scalability

Program Slicing



Program Slicing In Action



Precondition

Complex code not mentioning a, s

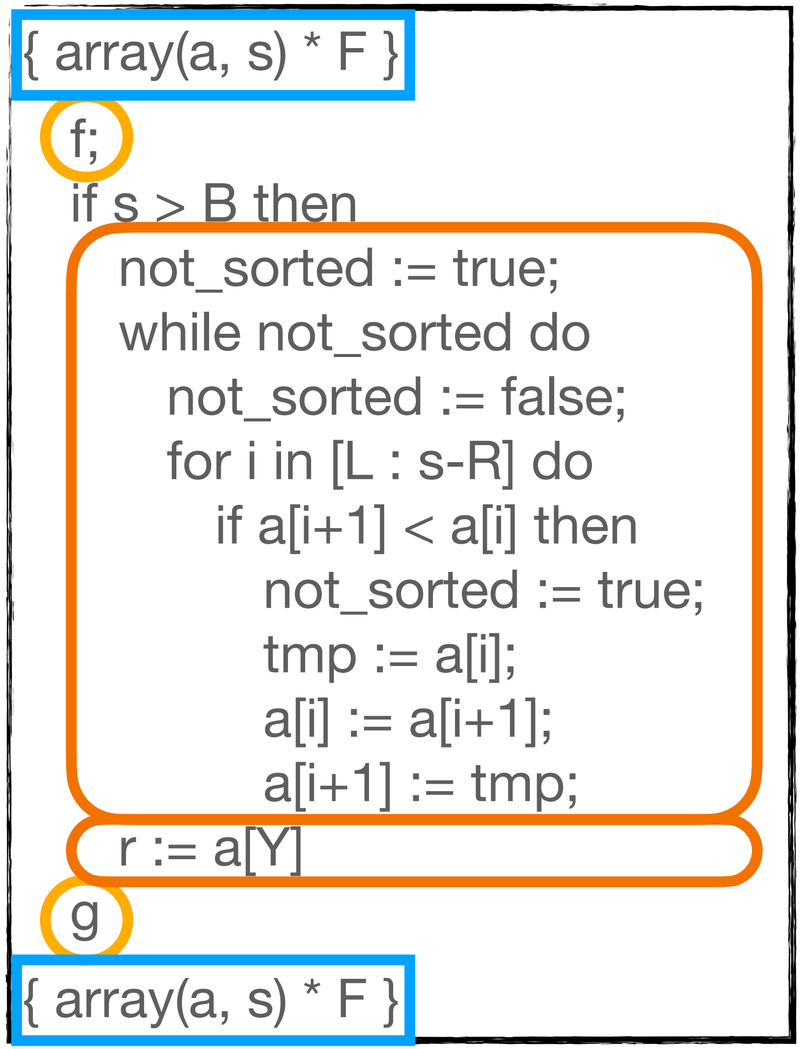
bubble_sort(a)

Select element

Complex code not mentioning a, s

Postcondition

Program Slicing In Action



Precondition

Complex code not mentioning a, s

bubble_sort(a)

Select element

Complex code not mentioning a, s

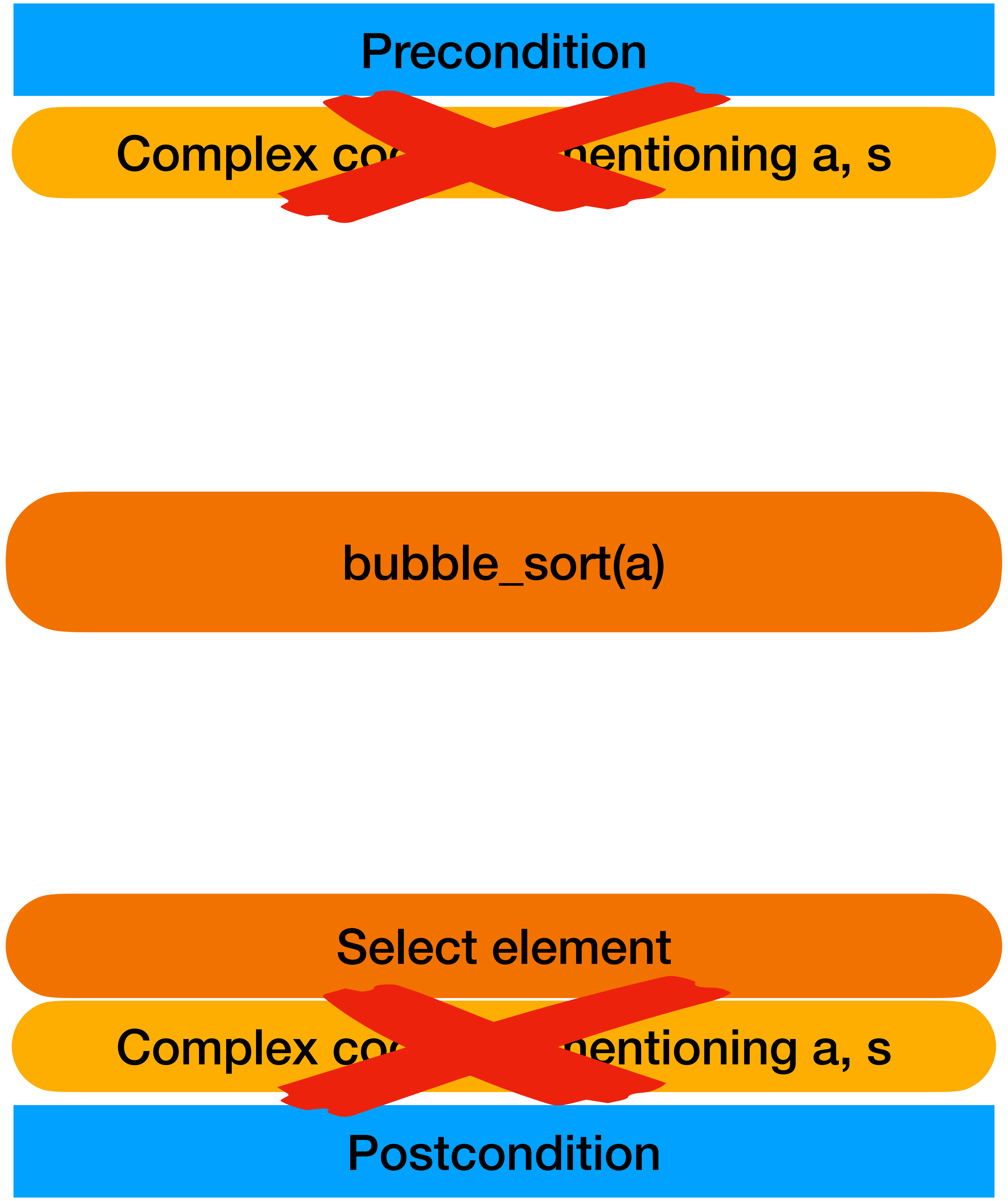
Postcondition

Program Slicing In Action

```
{ array(a, s) * }

  if s > B then
    not ... true;
    while ... ed do
      not ... alse;
      for i in [L : s-R] do
        if a[i+1] < a[i] then
          not ... true;
          t p := a[i];
          a[i] := a[i+1];
          a[i+1] := t p;
   := a[Y]

{ array(a, s) * }
```



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

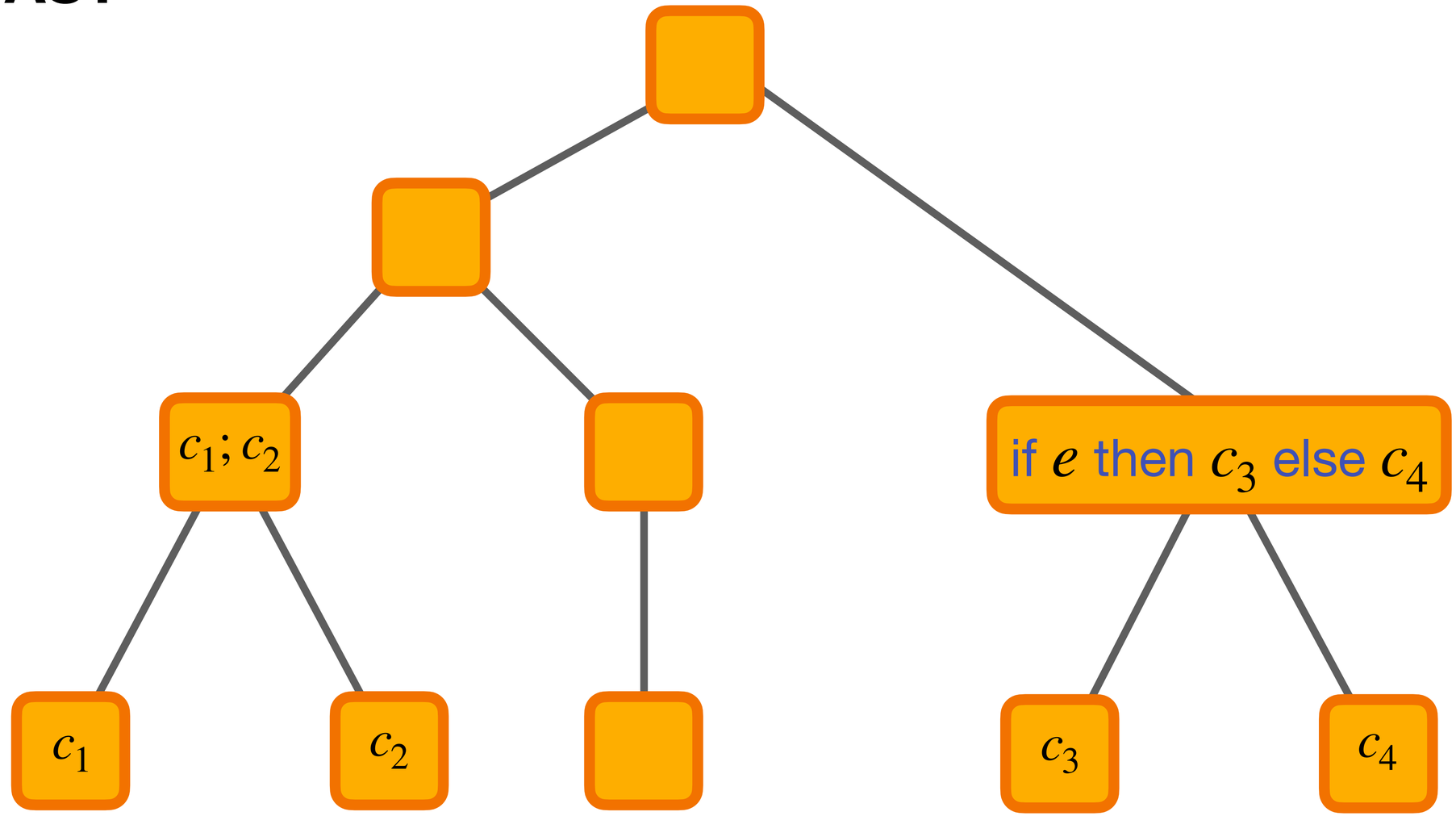
$Q \sim (e \wedge K_1) \cup (\neg e \wedge K_2)$

CTs as contraint sets

$Q_i \sim K_i$

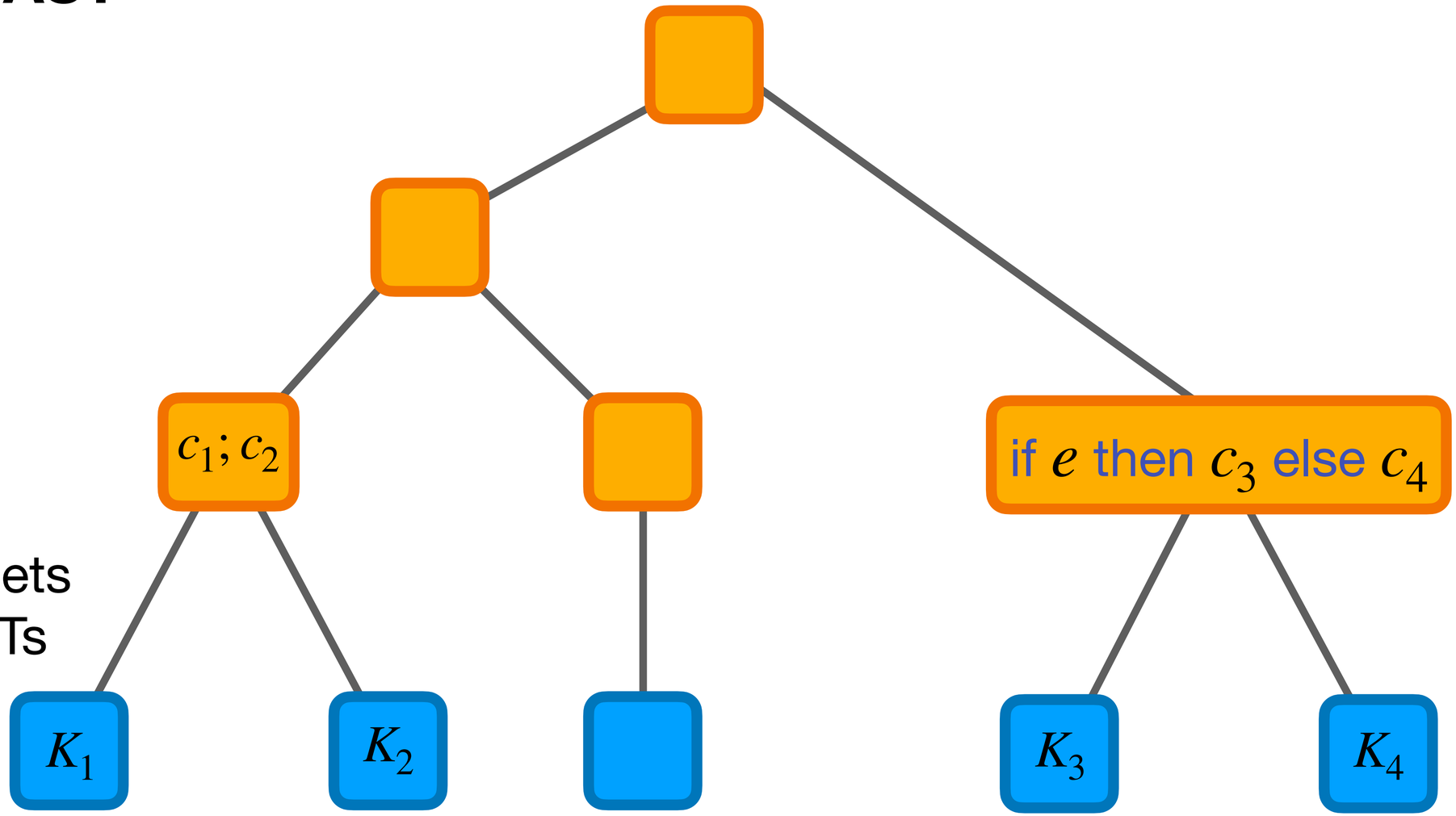
Scalability

Follow AST



Scalability

Follow AST



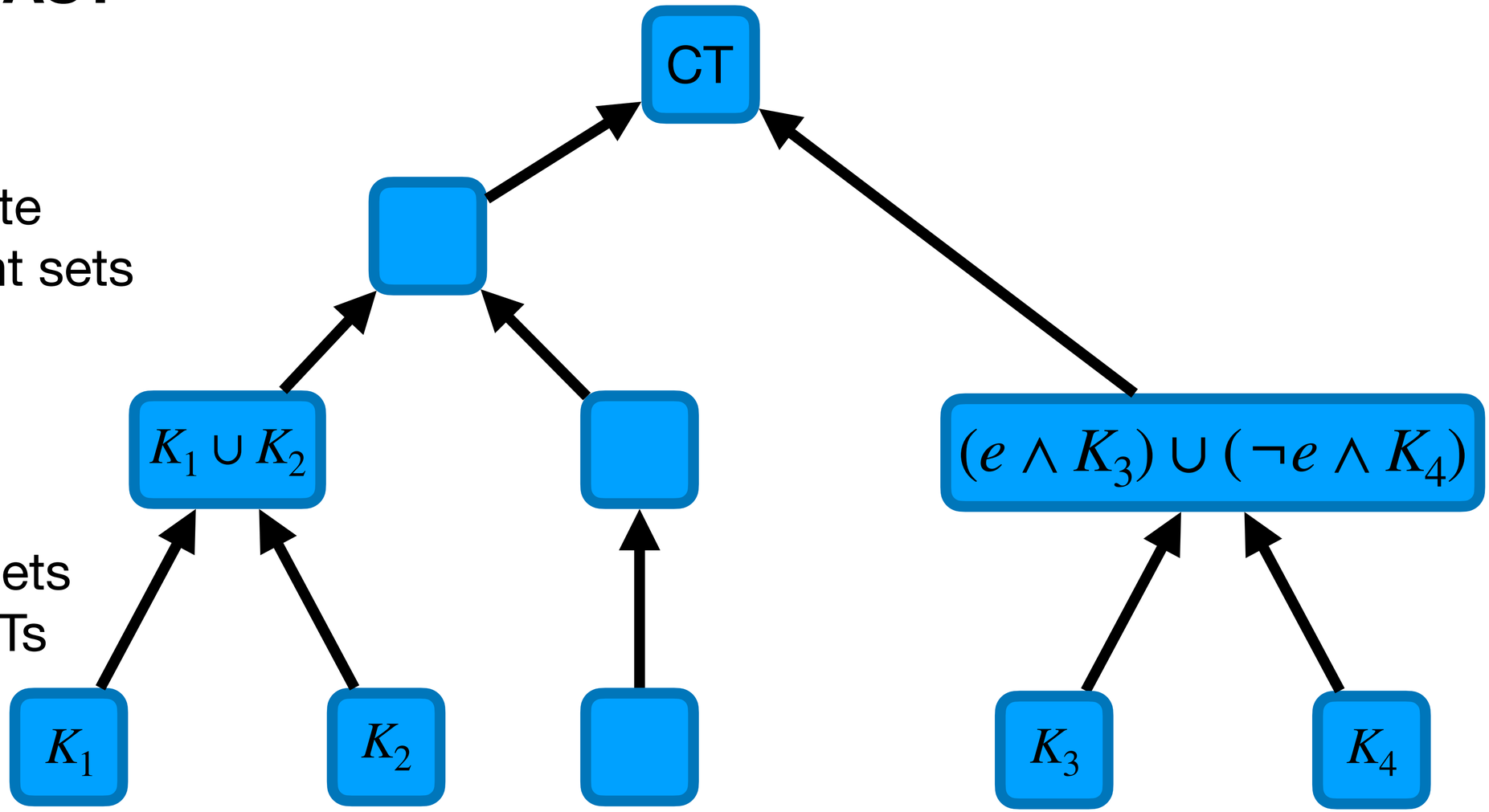
CT constr. sets for sub-ASTs

Scalability

Follow AST

propagate
CT constraint sets

CT constr. sets
for sub-ASTs

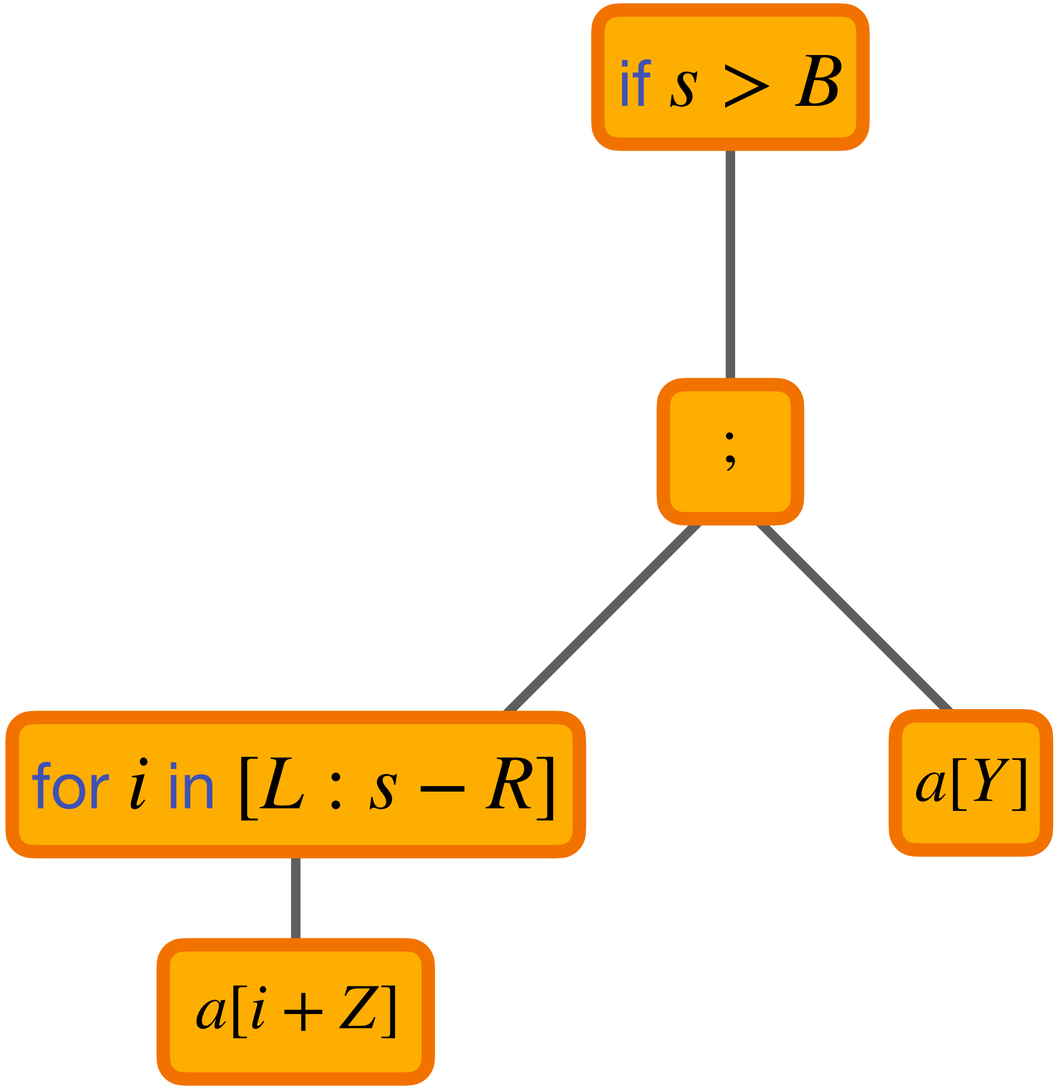


CT Propagation In Action

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

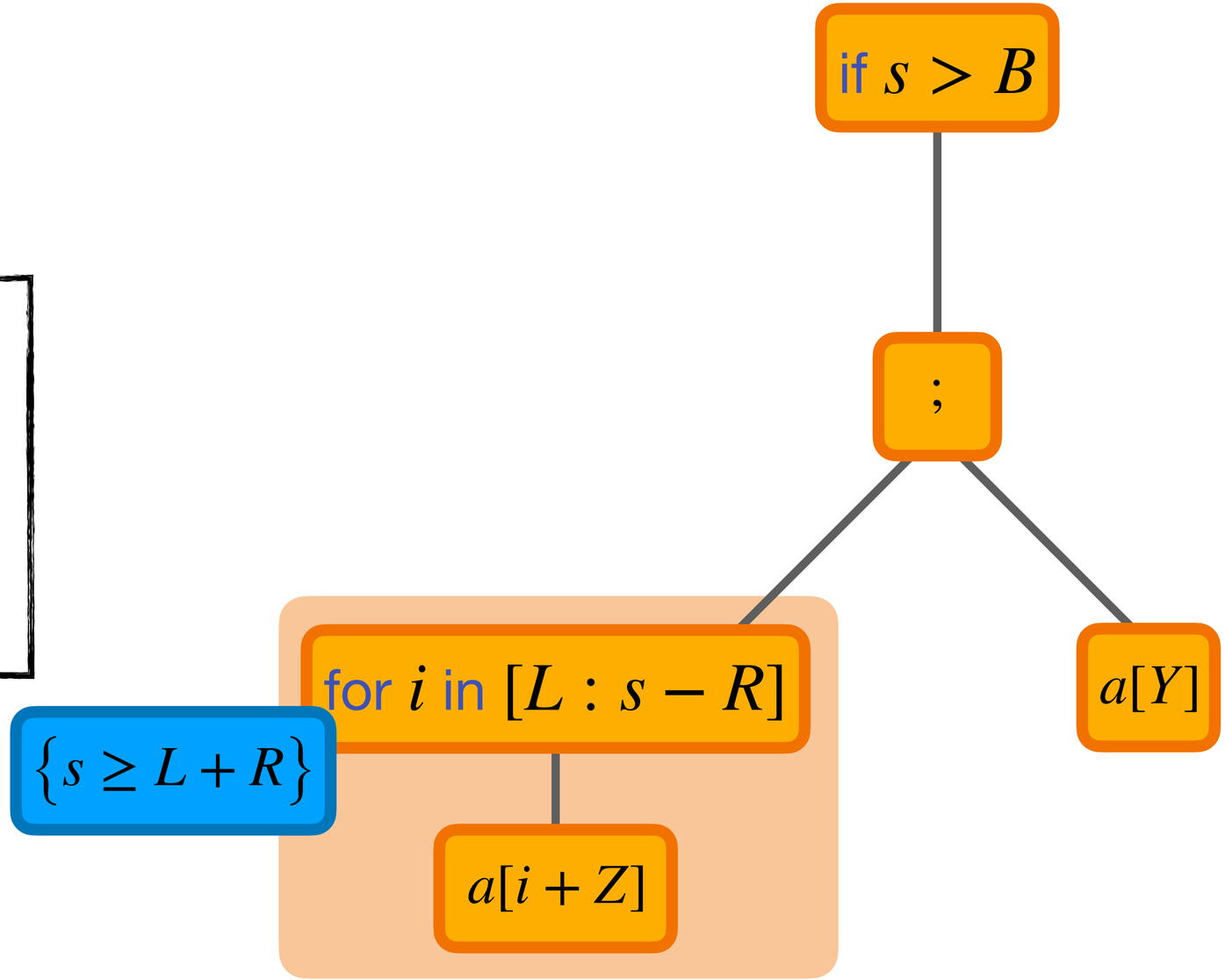
CT Propagation In Action

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



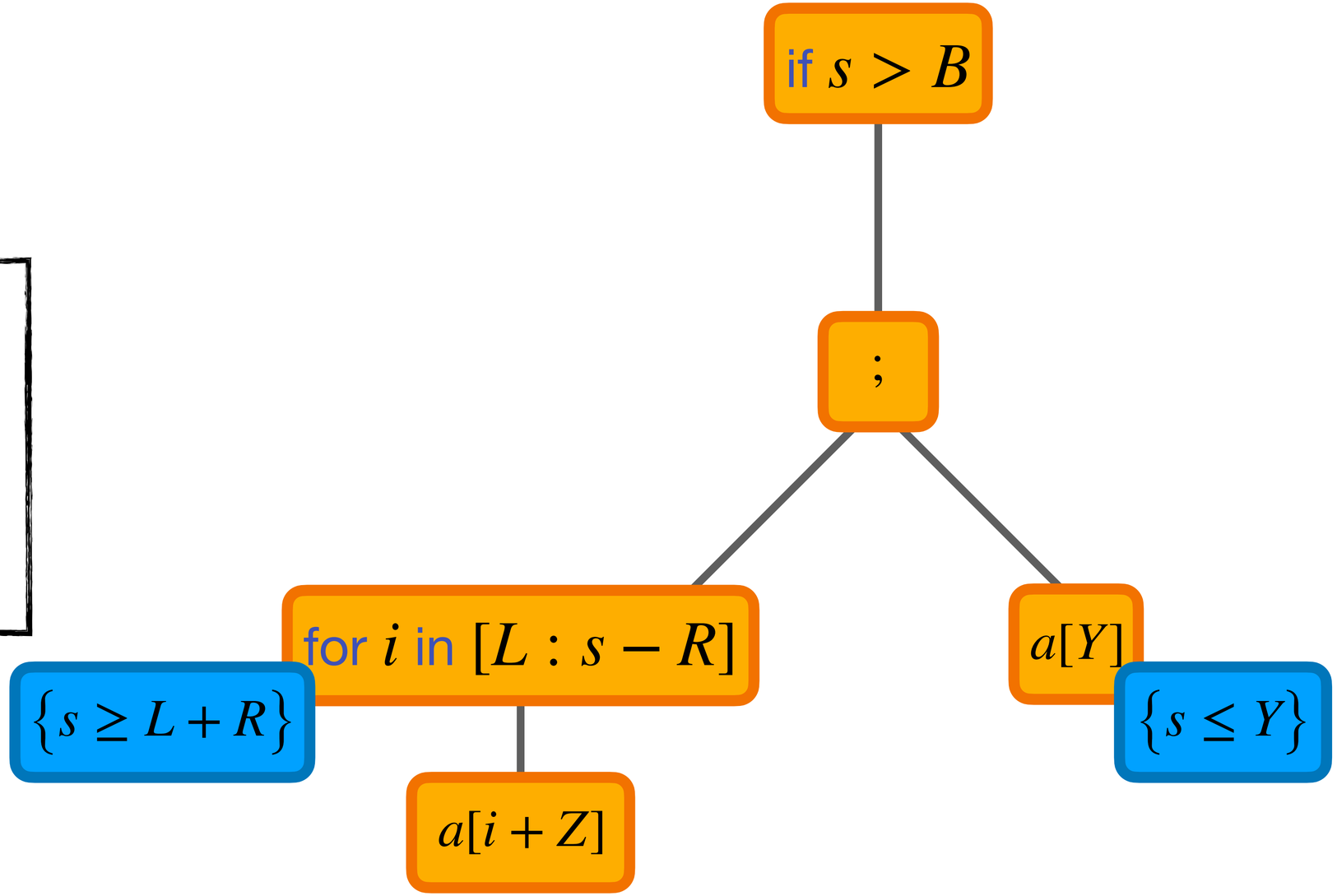
CT Propagation In Action

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



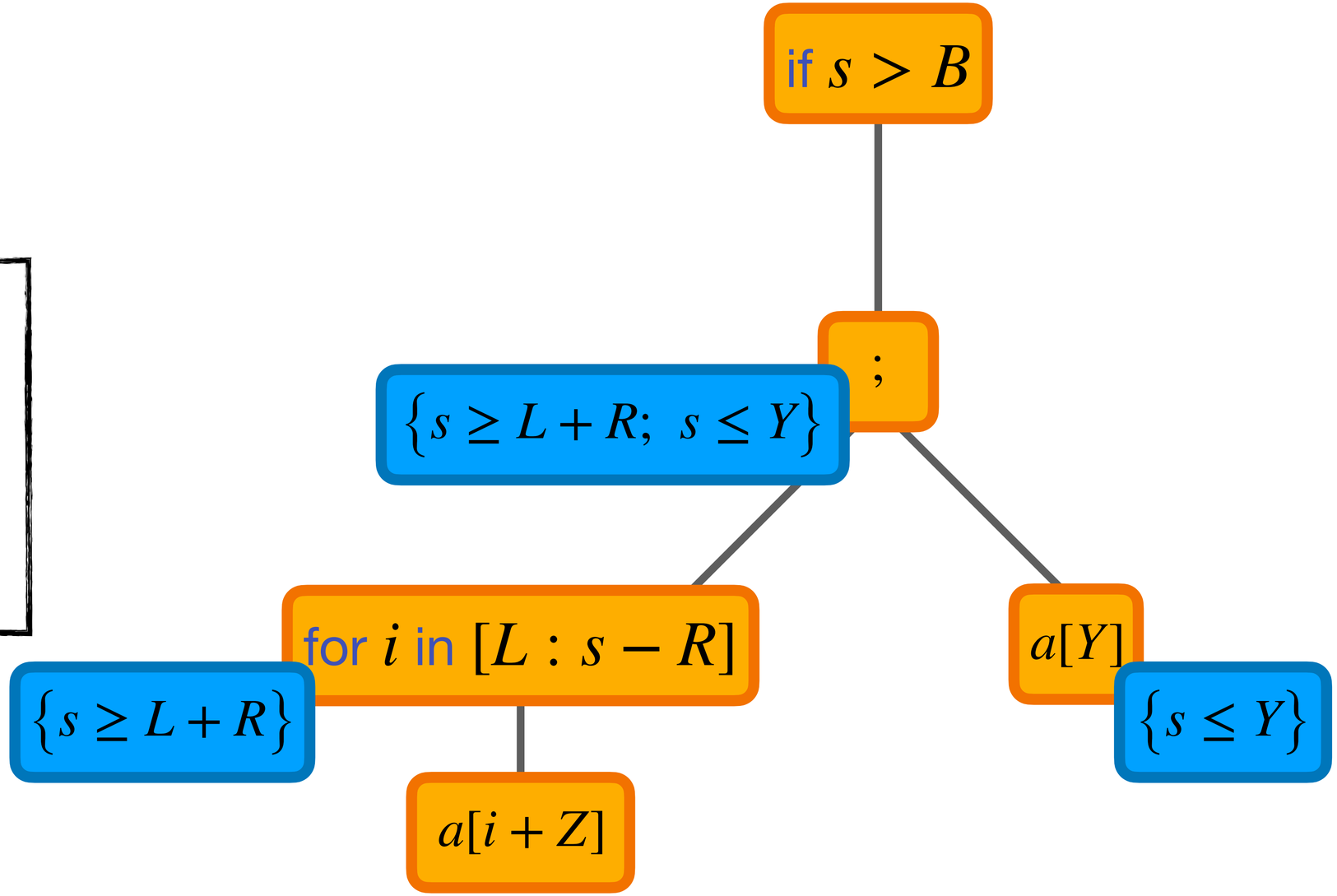
CT Propagation In Action

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



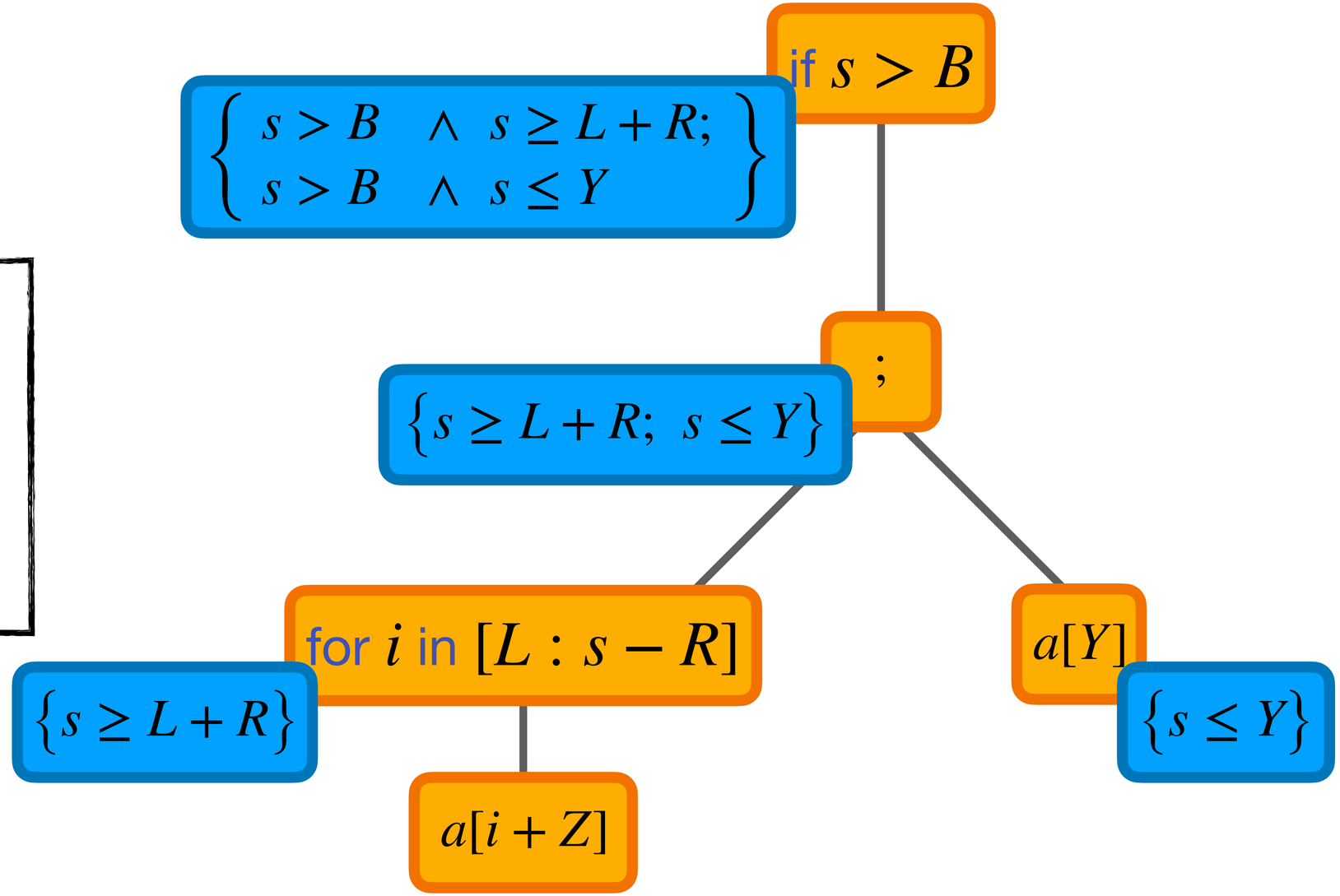
CT Propagation In Action

```
{ 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) }
```



Generalisation

	Memory Safety	Arbitrary Correctness Property ϕ
Theory		
• What are CTs?	✓	✓
• Describe behaviour	✓	✓
• Extraction approach	✓	✓
Practice		
• Obtain CT	✓	
• Scale	✓	

Generalisation

	Memory Safety	Arbitrary Correctness Property ϕ
Theory		
• What are CTs?	✓	✓
• Describe behaviour	✓	✓
• Extraction approach	✓	✓
Practice		ϕ-specific relationship: program↔VC
• Obtain CT	✓	?
• Scale	✓	

Generalisation

	Memory Safety	Arbitrary Correctness Property ϕ
Theory		
• What are CTs?	✓	✓
• Describe behaviour	✓	✓
• Extraction approach	✓	✓
Practice		Requires "local" ϕ
• Obtain CT	✓	?
• Scale	✓	?

Outlook: Plans & Challenges

Plans

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

Outlook: Plans & Challenges

Plans

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

Challenge: Automation

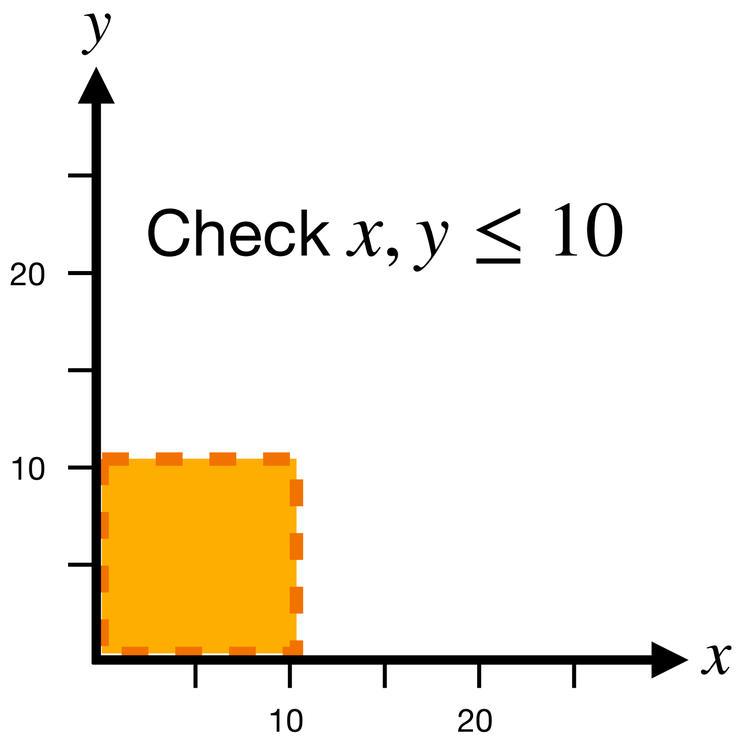
- Pattern recognition
- automatic VC rewriting

Outlook: Increase Trust in BMC

- Turn bounded into unbounded proof

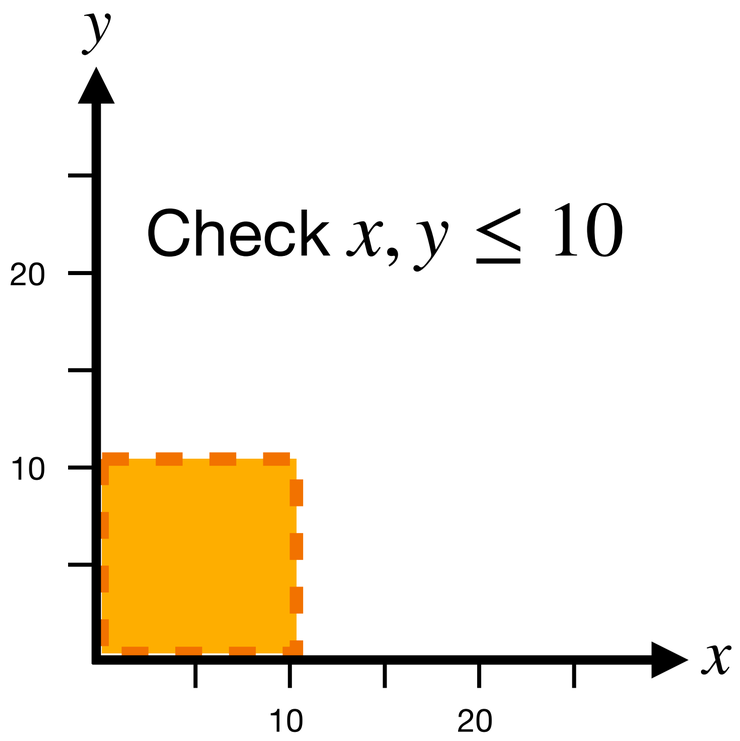
Outlook: Increase Trust in BMC

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

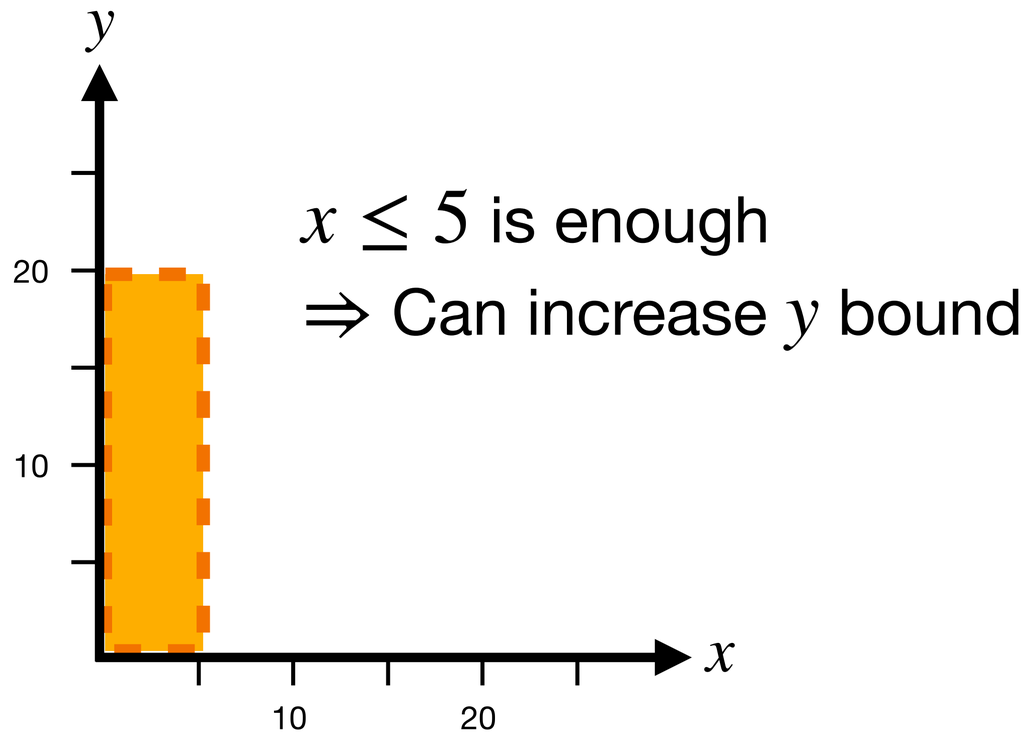


Outlook: Increase Trust in BMC

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



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



$x \leq 5$ is enough
$\Rightarrow$ Can increase y bound

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

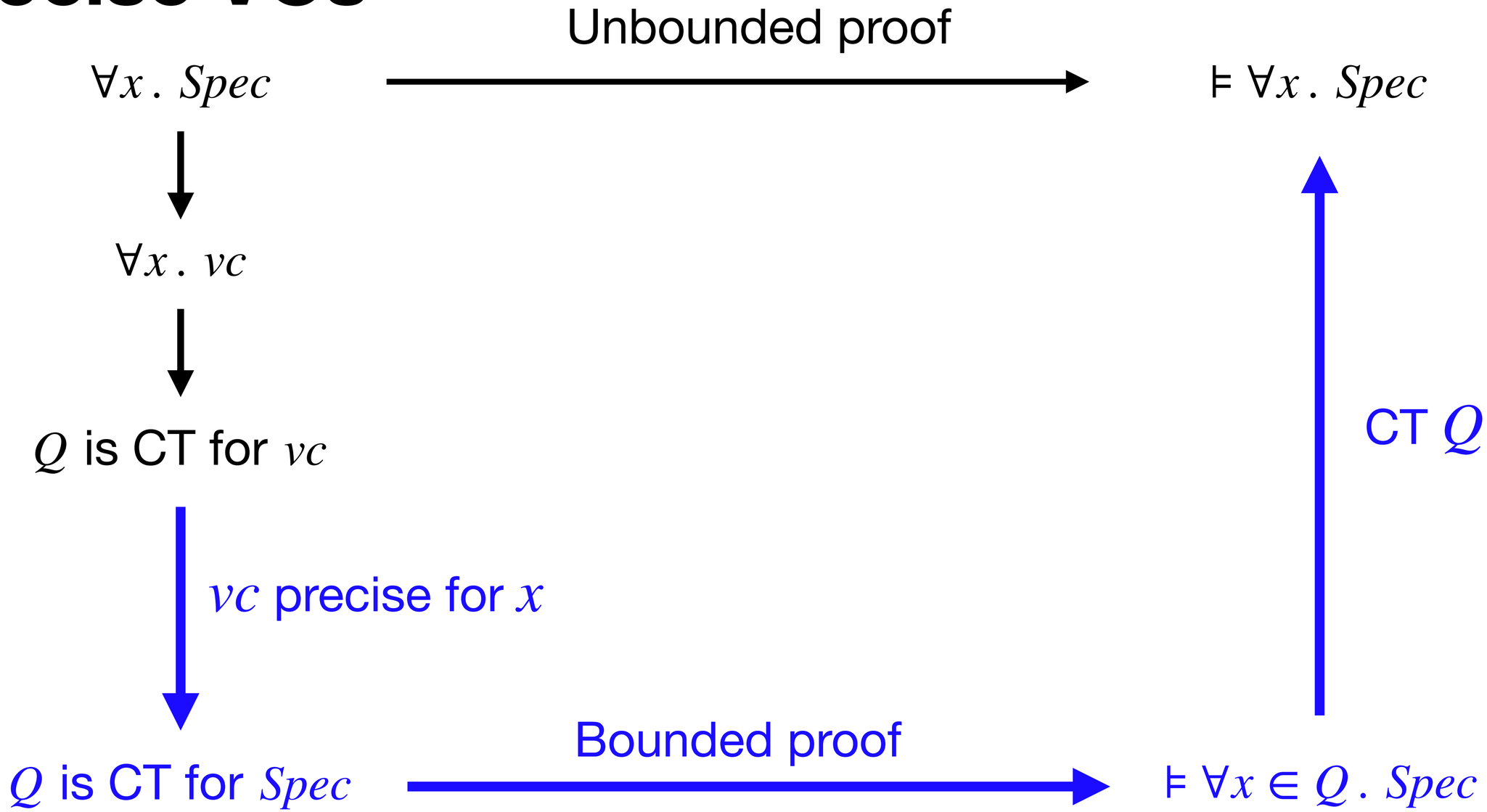
- VC vc is *precise* for x in $Spec$ iff

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

  Intuition: vc does not over-approximate wrt. x

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

Precise VCs


Unbounded proof
$\forall x\,.\,Spec$
$\models \forall x\,.\,Spec$
$\forall x\,.\,vc$
Q is CT for vc
CT Q
vc precise for x
Bounded proof
Q is CT for $Spec$
$\models \forall x \in Q\,.\,Spec$