Synthesizing Imperative Programs from Examples Guided by Static Analysis

Sunbeom So and Hakjoo Oh
Korea University

30 August 2017
SAS 2017 @ New York, USA

SIMPL : Synthesizer for Imperative Language

- Input

  (1) examples

  (2) resource components

  (3) partial program

- Output

  - complete program

Running Example 1

- Reverse a given natural number.

1. Examples

$1 \Rightarrow 1$
$12 \Rightarrow 21$
$123 \Rightarrow 321$

2. Resources

Integers : {0,1,10}
Variables : {n, r, x}

3. Partial program

```
reverse (n)
  r := 0;
  while ([ ]) {
    [ ]
  }
  return r;
```

2.5 s

Complete program

```
reverse (n)
  r := 0;
  while (n > 0) {
    x := n % 10;
    r := r * 10;
    r := r + x;
    n := n / 10;
  }
  return r;
```

Running Example 1

- Reverse a given natural number.

1. Examples

$1 \Rightarrow 1$
$12 \Rightarrow 21$
$123 \Rightarrow 321$

2. Resources

Integers : {0,1,10}

Variables : {n, r, x}

3. Partial program

```
reverse (n)
  r := 0;
  while (      ) {

  }
  return r;
```

2.5 s

Complete program

```
reverse (n)
  r := 0;
  while (n > 0) {
    x := n % 10;
    r := r * 10;
    r := r + x;
    n := n / 10;
  }
  return r;
```

Running Example 1

- Reverse a given natural number.

1. Examples

1 ⇒ 1
12 ⇒ 21
123 ⇒ 321

2. Resources

Integers : {0,1,10}
Variables : {n, r, x}

3. Partial program

```
reverse (n)
  r := 0;
  while ([ ]) {
    [ ]
  }
  return r;
```

2.5 s

Complete program

```
reverse (n)
  r := 0;
  while (n > 0) {
    x := n % 10;
    r := r * 10;
    r := r + x;
    n := n / 10;
  }
  return r;
```

Running Example 1

- Reverse a given natural number.

1. Examples

1 ⇒ 1
12 ⇒ 21
123 ⇒ 321

2. Resources

Integers : {0,1,10}

Variables : {n, r, x}

3. Partial program

```
reverse (n)
  r := 0;
  while ( [ ] ) {
    [ ]
  }
  return r;
```

2.5 s

Complete program

```
reverse (n)
  r := 0;
  while (n > 0) {
    x := n % 10;
    r := r * 10;
    r := r + x;
    n := n / 10;
  }
  return r;
```

Running Example 1

- Reverse a given natural number.

1. Examples

1 ⇒ 1
12 ⇒ 21
123 ⇒ 321

2. Resources

Integers : {0,1,10}

Variables : {n, r, x}

3. Partial program

```
reverse (n)
  r := 0;
  while ([ ]) {
    [ ]
  }
  return r;
```

2.5 s

Complete program

```
reverse (n)
  r := 0;
  while (n > 0) {
    x := n % 10;
    r := r * 10;
    r := r + x;
    n := n / 10;
  }
  return r;
```

Running Example 1

- Reverse a given natural number.

1. Examples

$1 \Rightarrow 1$
$12 \Rightarrow 21$
$123 \Rightarrow 321$

2. Resources

Integers : {0,1,10}
Variables : {n, r, x}

3. Partial program

```
reverse (n)
  r := 0;
  while ( [ ] ) {
    [ ]
  }
  return r;
```

2.5 s

Complete program

```
reverse (n)
  r := 0;
  while ( n > 0 ) {
    x := n % 10;
    r := r * 10;
    r := r + x;
    n := n / 10;
  }
  return r;
```

Running Example 2

- Count the numbers of occurrences each digit.

1. Examples

11, <0,0> ⇒ <0,2>
220, <0,0,0> ⇒ <1,0,2>

2. Resources

Integers : {0,1,10}

Int-Vars : {i, n, t}

Arr-Vars : {a}

3. Partial program

```
count (n, a)
 while ( [    ] ) {
   [          ]
 }
 return a;
```

0.2 s

Complete program

```
count (n, a)
 while (n > 0) {
   t := n % 10;
   a[t] := a[t] + 1;
   n := n / 10;
 }
 return a;
```

Running Example 2

- Count the numbers of occurrences each digit.

1. Examples

11, <0,0> ⇒ <0,2>
220, <0,0,0> ⇒ <1,0,2>

2. Resources

Integers : {0,1,10}
Int-Vars : {i, n, t}
Arr-Vars : {a}

3. Partial program

```
count (n, a)
 while ( ) {

 }
 return a;
```

0.2 s

Complete program

```
count (n, a)
 while (n > 0) {
   t := n % 10;
   a[t] := a[t] + 1;
   n := n / 10;
 }
 return a;
```

Running Example 2

- Count the numbers of occurrences each digit.

1. Examples

11, <0,0> ⇒ <0,2>
220, <0,0,0> ⇒ <1,0,2>

2. Resources

Integers : {0,1,10}
Int-Vars : {i, n, t}
Arr-Vars : {a}

3. Partial program

```
count (n, a)
 while ( ☐ ) {
   ☐
 }
 return a;
```

0.2 s

Complete program

```
count (n, a)
 while (n > 0) {
   t := n % 10;
   a[t] := a[t] + 1;
   n := n / 10;
 }
 return a;
```

Running Example 2

- Count the numbers of occurrences each digit.

1. Examples

11, <0,0> ⇒ <0,2>
220, <0,0,0> ⇒ <1,0,2>

2. Resources

Integers : {0,1,10}
Int-Vars : {i, n, t}
Arr-Vars : {a}

3. Partial program

```
count (n, a)
 while ([ ]) {
   [ ]
 }
 return a;
```

0.2 s

Complete program

```
count (n, a)
 while (n > 0) {
   t := n % 10;
   a[t] := a[t] + 1;
   n := n / 10;
 }
 return a;
```

Running Example 2

- Count the numbers of occurrences each digit.

1. Examples

11, <0,0> ⇒ <0,2>
220, <0,0,0> ⇒ <1,0,2>

2. Resources

Integers : {0,1,10}

Int-Vars : {i, n, t}

Arr-Vars : {a}

3. Partial program

```
count (n, a)
 while ( [ ] ) {
   [ ]
 }
 return a;
```

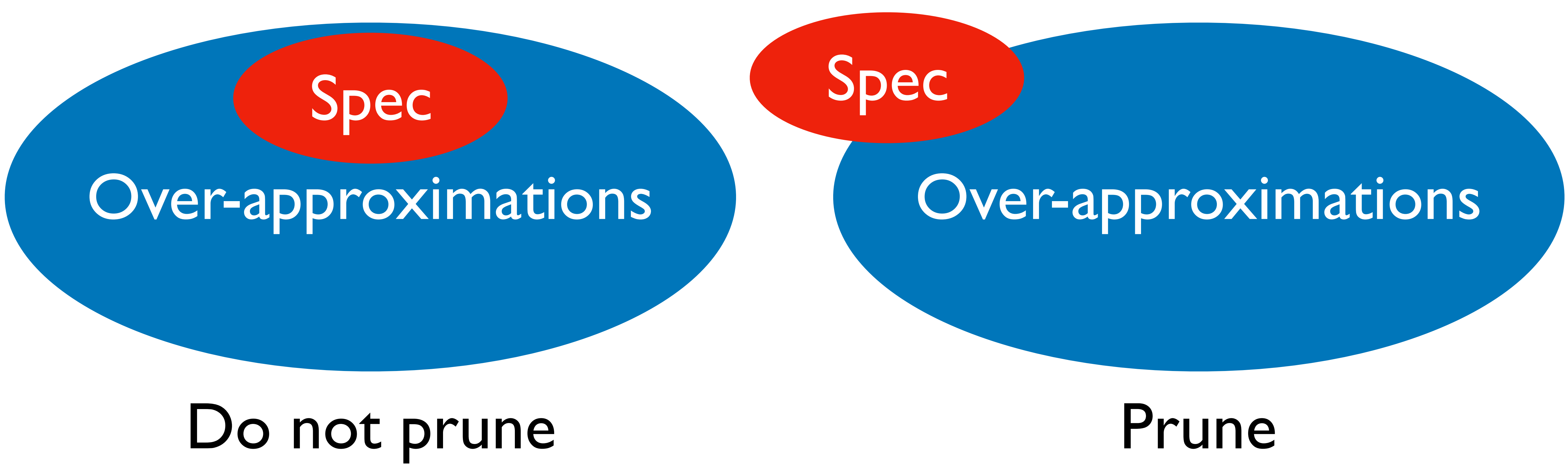
0.2 s

Complete program

```
count (n, a)
 while (n > 0) {
   t := n % 10;
   a[t] := a[t] + 1;
   n := n / 10;
 }
 return a;
```

Key : Static Analysis

- Over-approximate behaviors of a candidate.
  - by performing abstract interpretation
- If the result does not contain spec, prune the state.



Effectiveness

- Without our pruning:

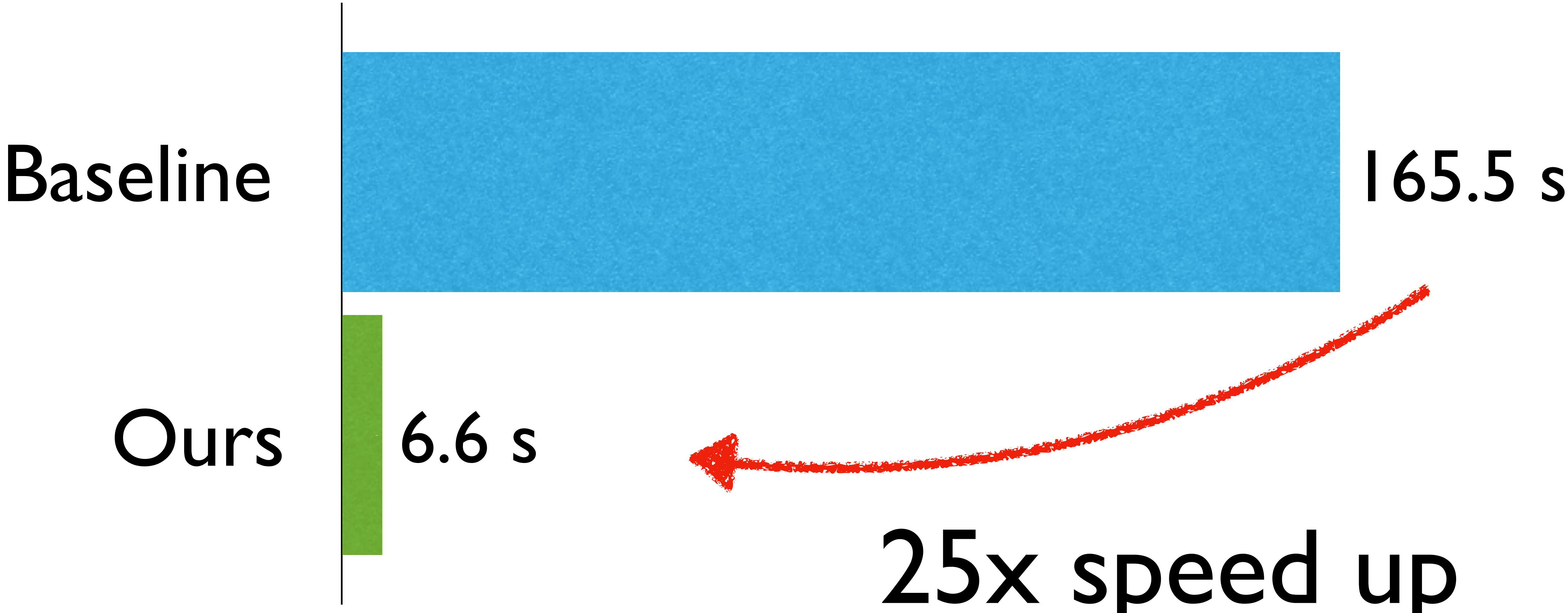
```
reverse (n)
  r := 0;
  while (n > 0) {
    x := n % 10;
    r := r * 10;
    r := r + x;
    n := n / 10;
  }
  return r;
```

~~2.5 s~~ 367.3 s

```
count (n)
  while (n > 0) {
    t := n % 10;
    a[t] := a[t] + 1;
    n := n / 10;
  }
  return r;
```

~~0.2 s~~ 1094.1 s

Effectiveness



25x speed up
(the pruning added)

Technical Outline

- Baseline algorithm
  - enumerative search
  - state normalization
- Static analysis guided pruning

Baseline Algorithm

Basic Method : Enumeration

Language

$$
\begin{aligned}
\oplus &::= + \mid - \mid * \mid / \mid \%, \qquad \prec \;::=\; = \mid > \mid < \\
l &::= x \mid x[y] \\
a &::= n \mid l \mid l_1 \oplus l_2 \mid l \oplus n \mid \Diamond \\
b &::= \mathit{true} \mid \mathit{false} \mid l_1 \prec l_2 \mid l \prec n \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b \mid \triangle \\
c &::= l := a \mid \mathit{skip} \mid c_1 ; c_2 \mid \mathit{if}\ b\ c_1\ c_2 \mid \mathit{while}\ b\ c \mid \Box
\end{aligned}
$$

- Start from an initial partial program
- Enumerate every possible candidate program according to the CFG of the language.
  - by instantiating holes ($\Diamond$, $\triangle$, $\Box$)

Basic Method : Enumeration

Resources

Integer : {1}
Variable : {x}

$\square$; r := 1; r := $\Diamond$

current state

Basic Method : Enumeration

Resources

Integer : {1}
Variable : {x}

□; r := 1; r := ◇

current state

x:=◇; r:=1; r:=◇
skip; r:=1; r:=◇
□; □; r:=1; r:=◇
if △ □ □; r:=1; r:=◇
while △ □; r:=1; r:=◇
□; r:=1; r:=1
□; r:=1; r:=x
□; r:=1; r:=x+x
□; r:=1; r:=x-x
□; r:=1; r:=x*x
□; r:=1; r:=x/x
□; r:=1; r:=x%x
□; r:=1; r:=x+1
□; r:=1; r:=x-1
□; r:=1; r:=x*1
□; r:=1; r:=x/1
□; r:=1; r:=x%1

Basic Method : Enumeration

Resources

Integer : {1}
Variable : {x}

current state

□; r := 1; r := ◇

x:=◇; r:=1; r:=◇
skip; r:=1; r:=◇
□; □; r:=1; r:=◇
if △ □ □; r:=1; r:=◇
while △ □; r:=1; r:=◇
□; r:=1; r:=1
□; r:=1; r:=x
□; r:=1; r:=x+x
□; r:=1; r:=x-x
□; r:=1; r:=x*x
□; r:=1; r:=x/x
□; r:=1; r:=x%x
□; r:=1; r:=x+1
□; r:=1; r:=x-1
□; r:=1; r:=x*1
□; r:=1; r:=x/1
□; r:=1; r:=x%1

Basic Method : Enumeration

Resources

Integer : {1}
Variable : {x}

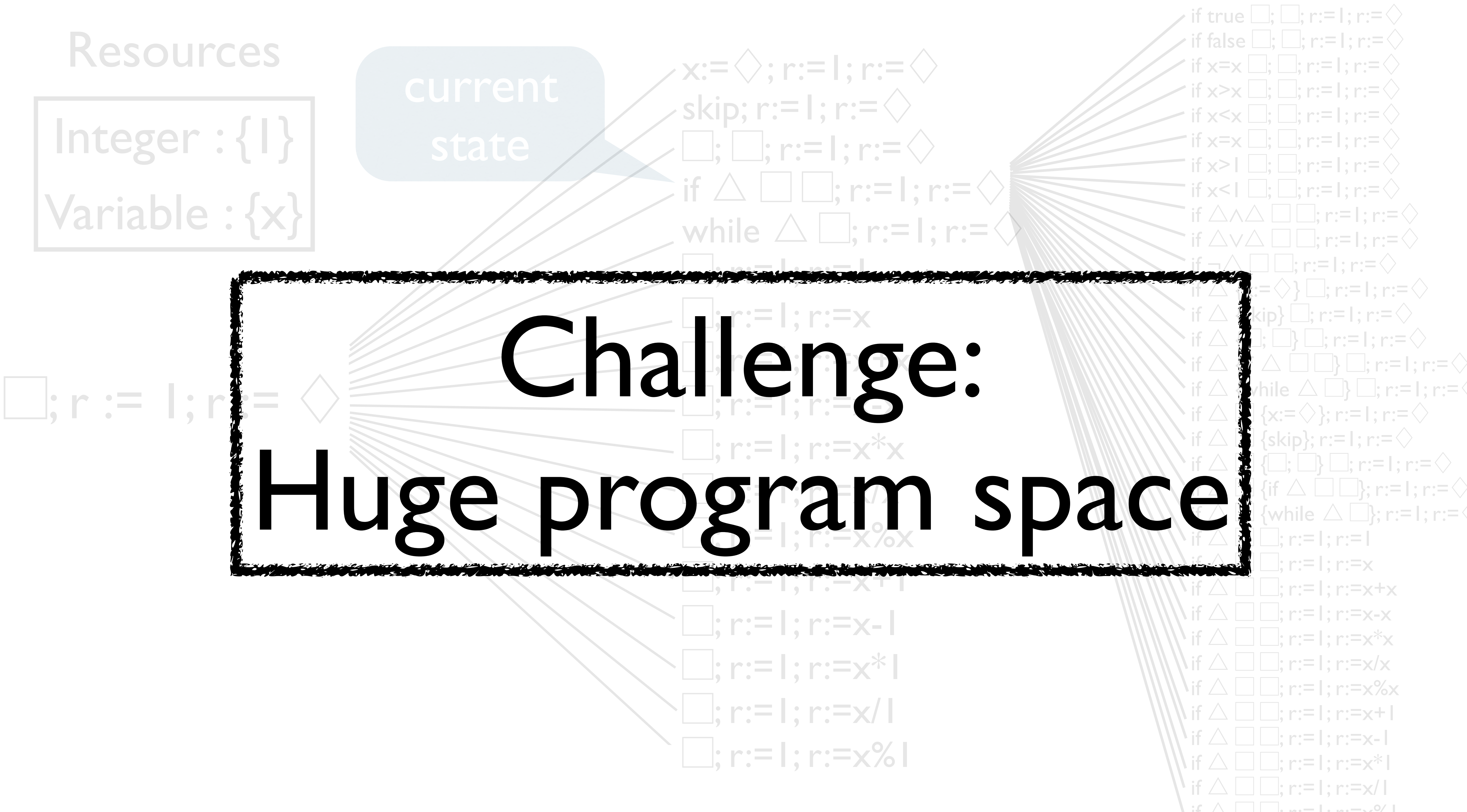
current state

□; r := 1; r := ◇

x:=◇; r:=1; r:=◇
skip; r:=1; r:=◇
□; □; r:=1; r:=◇
if △ □ □; r:=1; r:=◇
while △ □; r:=1; r:=◇
□; r:=1; r:=1
□; r:=1; r:=x
□; r:=1; r:=x+x
□; r:=1; r:=x-x
□; r:=1; r:=x*x
□; r:=1; r:=x/x
□; r:=1; r:=x%x
□; r:=1; r:=x+1
□; r:=1; r:=x-1
□; r:=1; r:=x*1
□; r:=1; r:=x/1
□; r:=1; r:=x%1

if true □; □; r:=1; r:=◇
if false □; □; r:=1; r:=◇
if x=x □; □; r:=1; r:=◇
if x>x □; □; r:=1; r:=◇
if x<x □; □; r:=1; r:=◇
if x=x □; □; r:=1; r:=◇
if x>1 □; □; r:=1; r:=◇
if x<1 □; □; r:=1; r:=◇
if △∧△ □ □; r:=1; r:=◇
if △∨△ □ □; r:=1; r:=◇
if ¬△ □ □; r:=1; r:=◇
if △ {x:=◇} □; r:=1; r:=◇
if △ {skip} □; r:=1; r:=◇
if △ {□; □} □; r:=1; r:=◇
if △ {if △ □ □} □; r:=1; r:=◇
if △ {while △ □} □; r:=1; r:=◇
if △ □ {x:=◇}; r:=1; r:=◇
if △ □ {skip}; r:=1; r:=◇
if △ □ {□; □} □; r:=1; r:=◇
if △ □ {if △ □ □}; r:=1; r:=◇
if △ □ {while △ □}; r:=1; r:=◇
if △ □ □; r:=1; r:=1
if △ □ □; r:=1; r:=x
if △ □ □; r:=1; r:=x+x
if △ □ □; r:=1; r:=x-x
if △ □ □; r:=1; r:=x*x
if △ □ □; r:=1; r:=x/x
if △ □ □; r:=1; r:=x%x
if △ □ □; r:=1; r:=x+1
if △ □ □; r:=1; r:=x-1
if △ □ □; r:=1; r:=x*1
if △ □ □; r:=1; r:=x/1
if △ □ □; r:=1; r:=x%1

Basic Method : Enumeration



Challenge:
Huge program space

State Normalization

- Avoid exploring semantically redundant ones
  - Code optimization techniques
  - Variable reordering : e.g.,

$$x := b + a \quad \Rightarrow \quad x := a + b$$

Code Optimizations

- Constant propagation
- Copy propagation
- Deadcode elimination
- Expression simplification

Code Optimizations

current state

$\boxed{r := 0;\ r := x * 0;\ \square}$

Code Optimizations

current state

r := 0; r := x * 0; □

Code Optimizations

current state

r := 0; r := x * 0; □

expression simplification

r := 0; r := 0; □

Code Optimizations

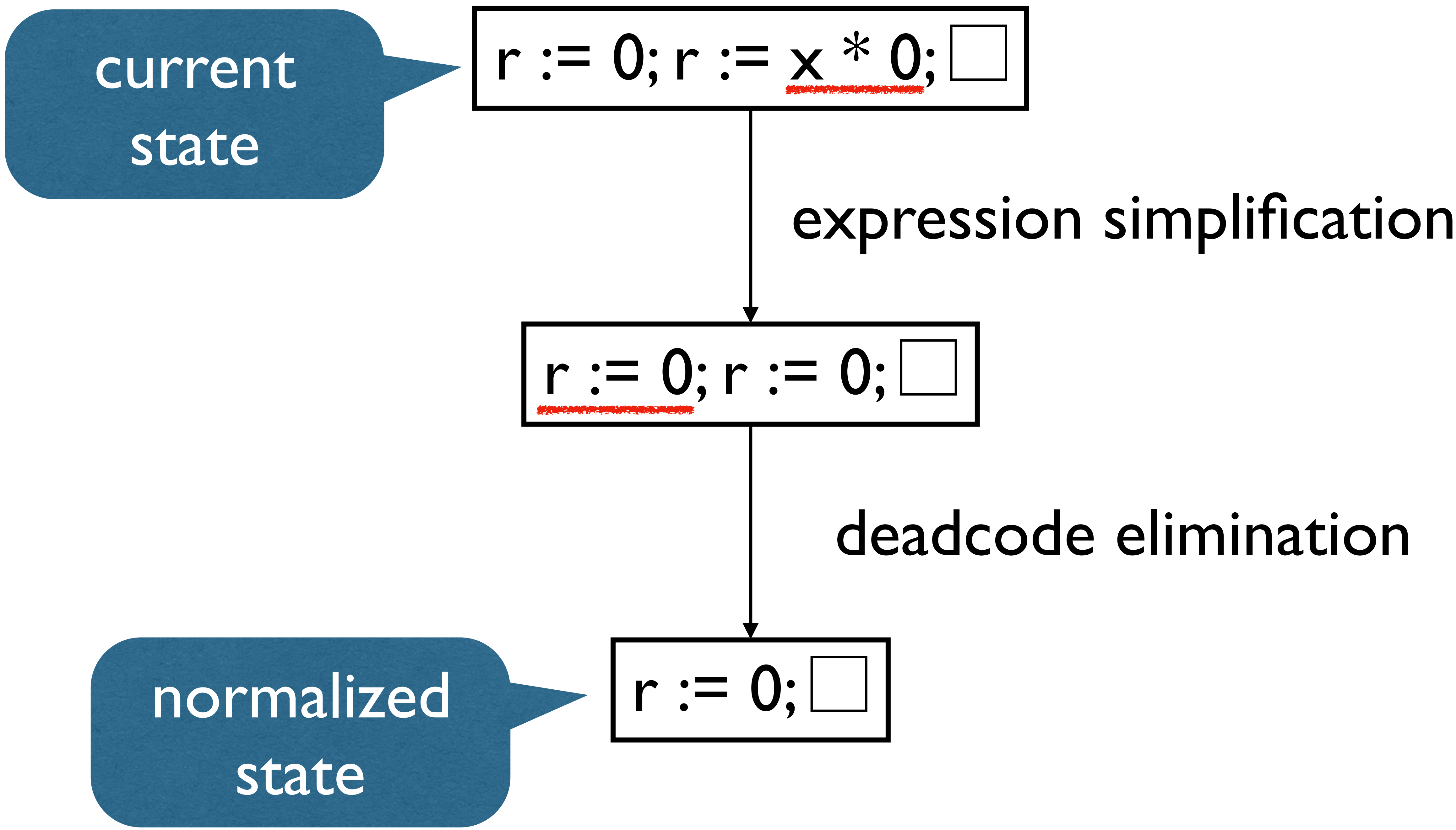
current state

r := 0; r := x * 0; □

expression simplification

r := 0; r := 0; □

Code Optimizations



Still, Very Slow

- State normalization significantly speeds up the enumerative search.
  - 3 failed benchmarks (> 1hour) $\Rightarrow$ success
- However, often took > 100 seconds.

Need more aggressive pruning

Observation

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
    r := n + 1;
    n := ◇;
  }
  return r;
```

Output : 1

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
        □;
    r := x * 10;
    n := n / 10;
  }
  return r;
```

Output : 1

Observation

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
    r := n + 1;
    n := ◇;
  }
  return r;
```

$r \geq 2$

Output : 1

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
        □;
    r := x * 10;
    n := n / 10;
  }
  return r;
```

Output : 1

Observation

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
    r := n + 1;
    n := ◇;
  }
  return r;
```

$r \geq 2$

Output : 1

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
        □;
    r := x * 10;
    n := n / 10;
  }
  return r;
```

multiple of 10

Output : 1

Observation

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
    r := n + 1;
    n := ◇;
  }
  return r;
```

$r \geq 2$

Output : 1

Interval

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
        □;
    r := x * 10;
    n := n / 10;
  }
  return r;
```

multiple of 10

Output : 1

Symbolic expression

Static Analysis Guided Pruning

Pruning with Static Analysis

1. Run static analysis on a candidate.

Pruning with Static Analysis

1. Run static analysis on a candidate.

2. Generate a constraint on the relationship between the analysis result and the output.

   - The over-approx must contain the output.

Pruning with Static Analysis

1. Run static analysis on a candidate.
2. Generate a constraint on the relationship between the analysis result and the output.
   - The over-approx must contain the output.
3. If the constraint is unsatisfiable for some examples, prune out the candidate.

1. Run Static Analysis

Interval

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
    r := n + 1;
    n := ◇;
  }
  return r;
```

Output : 1

Symbolic expression

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
         □;
    r := x * 10;
    n := n / 10;
  }
  return r;
```

Output : 1

1. Run Static Analysis

Interval

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
    r := n + 1;
    n := ◇;
  }
  return r;
```

n	
r	[2, 2]

Output : 1

Symbolic expression

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
        □;
    r := x * 10;
    n := n / 10;
  }
  return r;
```

n	
r	
x	

Output : 1

1. Run Static Analysis

Interval

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
    r := n + 1;
    n := ◇;
  }
  return r;
```

Assign top

n	[-∞,+∞]
r	[2, 2]

Output : 1

Symbolic expression

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
        □;
    r := x * 10;
    n := n / 10;
  }
  return r;
```

n	
r	
x	

Output : 1

1. Run Static Analysis

Interval

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
    r := n + 1;
    n := ◇;
  }
  return r;
```

Assign top

n	[-∞,+∞]
r	[2, 2]

Output : 1

Symbolic expression

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
         □;
    r := x * 10;
    n := n / 10;
  }
  return r;
```

Assign symbol

n	β_n
r	β_r
x	β_x

Output : 1

1. Run Static Analysis

Interval

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
    r := n + 1;
    n := ◇;
  }
  return r;
```

Assign top

n	$[-\infty, +\infty]$
r	[2, 2]

return r; → $[2, +\infty]$

Output : 1

Symbolic expression

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
          □;
    r := x * 10;
    n := n / 10;
  }
  return r;
```

Assign symbol

n	β_n
r	β_r
x	β_x

return r; → $\beta_x * 10$

Output : 1

2. Constraint Generation

Interval

Input : I

```
program (n)
  r := 0;
  while ( n > 0 ) {
    r := n + 1;
    n := ◇;
  }
  return r;
```

Assign top

n	[-∞,+∞]
r	[2, 2]

[2, +∞]

Output : I

Symbolic expression

Input : I

```
program (n)
  r := 0;
  while ( n > 0 ) {
    □;
    r := x * 10;
    n := n / 10;
  }
  return r;
```

Assign symbol

n	β_n
r	β_r
x	β_x

$\beta_x * 10$

Output : I

2. Constraint Generation

Interval

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
    r := n + 1;
    n := ◇;
  }
  return r;
```

Assign top

n	$[-\infty, +\infty]$
r	[2, 2]

$[2, +\infty]$

Output : 1 ⟶ [1, 1]

Symbolic expression

Input : 1

```
program (n)
  r := 0;
  while ( n > 0 ) {
    □;
    r := x * 10;
    n := n / 10;
  }
  return r;
```

Assign symbol

n	β_n
r	β_r
x	β_x

$\beta_x * 10$

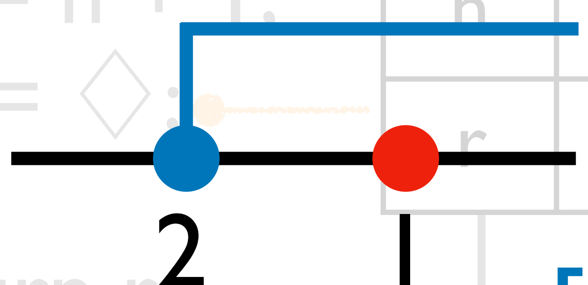
Output : 1 ⟶ [1, 1]

2. Constraint Generation

Interval

Constraint:

$2 \leq 1 \;\land\; 1 \leq +\infty$



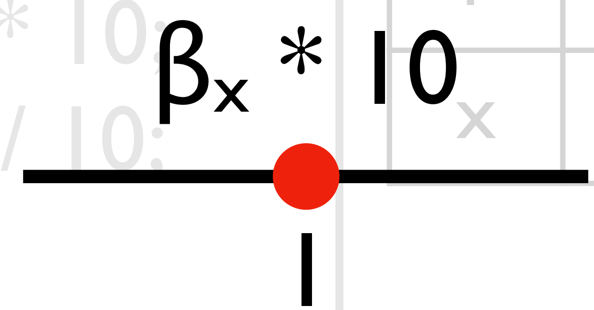
$[2, +\infty]$

Output : 1 ⟶ [1, 1]

Symbolic expression

Constraint:

$1 \leq \beta_x * 10 \leq 1$



$\beta_x * 10$

Output : 1 ⟶ [1, 1]

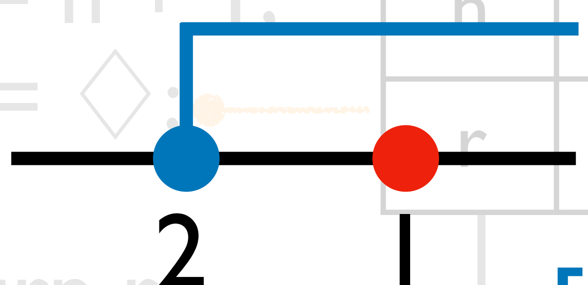
3. Checking Satisfiability

Interval

Unsatisfiable

Constraint:

$2 \leq 1 \;\wedge\; 1 \leq +\infty$



$[2, +\infty]$

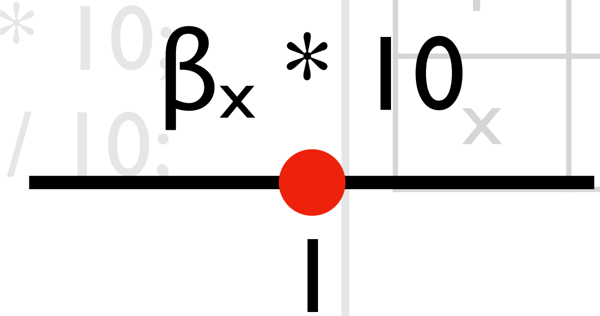
Output : 1 $\longrightarrow$ [1, 1]

Symbolic expression

Unsatisfiable

Constraint:

$1 \leq \beta_x * 10 \leq 1$



$\beta_x * 10$

Output : 1 $\longrightarrow$ [1, 1]

Safeness

- We prune out a state only when any further search of the state is *guaranteed to fail*.

Theorem 1 (Safety). $\forall s \in S.\ \mathsf{prune}(s) \implies \mathsf{fail}(s)$.

Evaluation

- Collected 30 benchmarks from online forums on introductory programming problems
- Various tasks for manipulating integers and arrays of integers
- Timeout : 3,600 seconds

Evaluation

Timeout : 3 benchmarks

	Enum		
Average (seconds)	>616.8		

- Enum : Naive enumerative search

Evaluation

Timeout : 3 benchmarks

No timeout

	Enum	Base	
Average (seconds)	>616.8	165.5	

x 3.7↑

- Enum : Naive enumerative search
- Base : Enum + state normalization

Evaluation

	Enum	Base	Ours
Average (seconds)	>616.8	165.5	6.6

Timeout : 3 benchmarks

No timeout

x 3.7↑

x 25

- Enum : Naive enumerative search
- Base : Enum + state normalization
- Ours : Base + static analysis guided pruning

Conclusion

- Static analysis is useful in program synthesis, too.
  - Speed up the baseline algorithm x25.

Conclusion

- Static analysis is useful in program synthesis, too.
  - Speed up the baseline algorithm x25.

Thank you