Efficient Symbolic Execution for Software Testing

Johannes Kinder

Royal Holloway, University of London

Joint work with:

Stefan Bucur, George Candea, Volodymyr Kuznetsov @ EPFL

Symbolic Execution

- Automatically explore program paths
  - *Execute program on "symbolic" input values*
  - *"Fork" execution at each branch*
  - *Record branching conditions*
- Constraint solver
  - *Decides path feasibility*
  - *Generates test cases for paths and bugs*

Symbolic Execution

- (Very brief) history
  - *Test generation by SE in 70s* [King '75] [Boyer et al. '75]
  - *SAT / SMT solvers lead to boom in 2000s* [Godefroid et al. '05][Cadar et al. '06]
- Many successful tools
  - *KLEE, SAGE, PEX, SPF, CREST, Cloud9, S2E, …*
- Specific advantages
  - *No false positives, useful partial results*
  - *Reduces need for modeling*

Outline

- Symbolic Execution for Testing
- State Merging – Fighting Path Explosion
- Interpreted High-Level Code

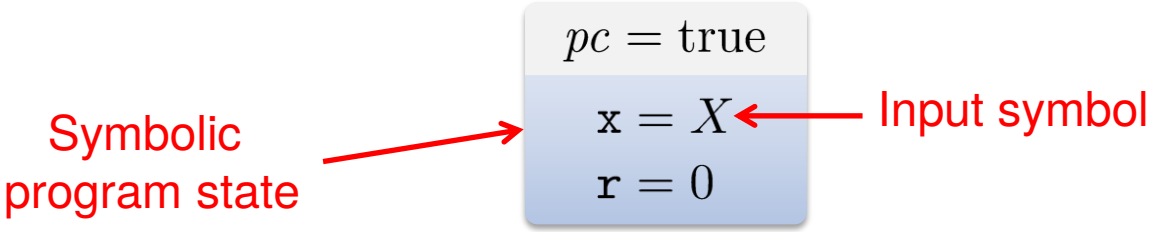
$pc = \text{true}$

$\mathtt{x} = X$

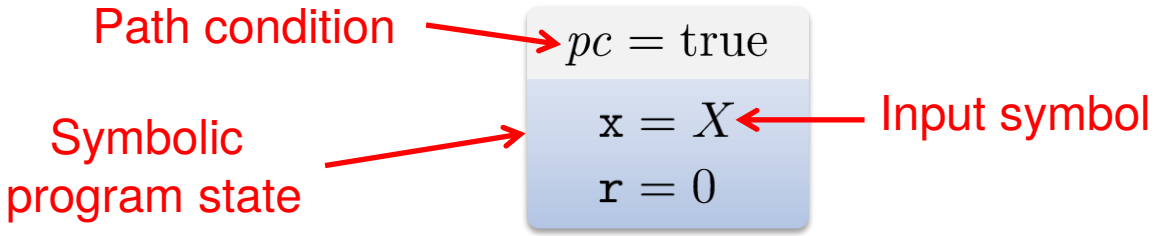
$\mathtt{r} = 0$

```
 1  int proc(int x) {
 2
 3   int r = 0
 5
 6   if (x > 8) {
 7     r = x - 7
 8   }
 9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```


```
 1  int proc(int x) {
 2
 3   int r = 0
 5
 6   if (x > 8) {
 7     r = x - 7
 8   }
 9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```

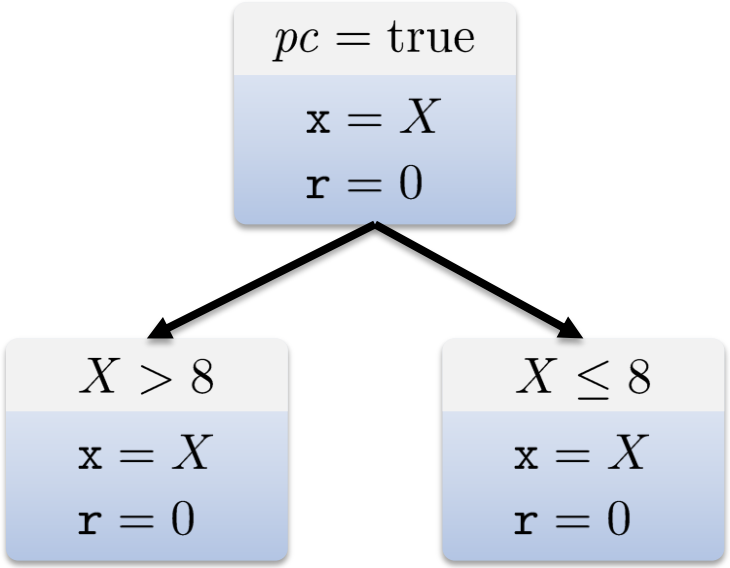



```
 1  int proc(int x) {
 2
 3   int r = 0
 5
 6   if (x > 8) {
 7     r = x - 7
 8   }
 9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```

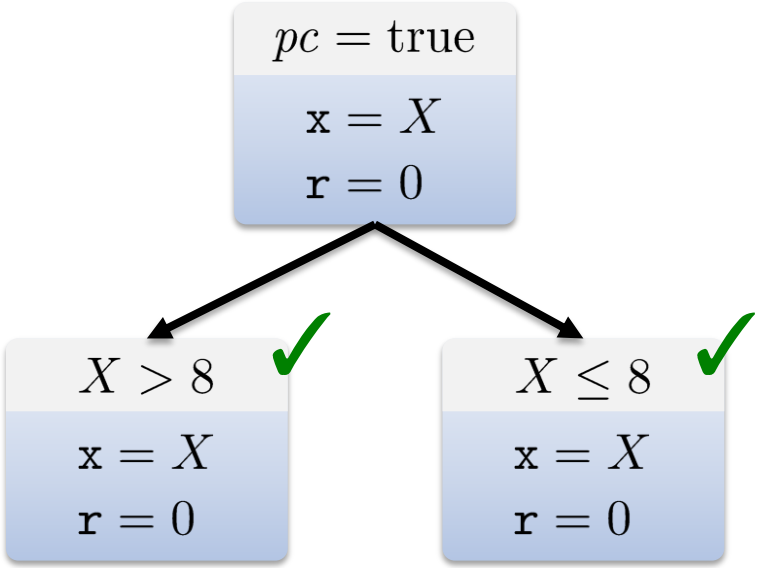



```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```

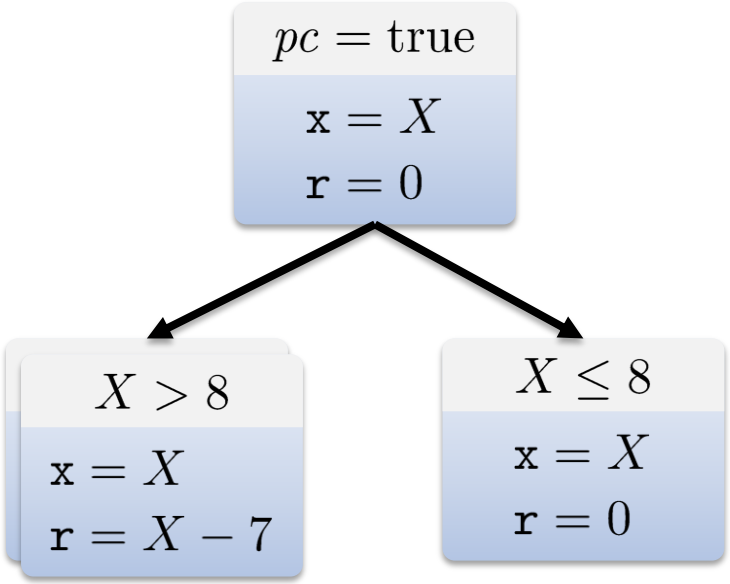
```
 1  int proc(int x) {
 2
 3   int r = 0
 5
 6   if (x > 8) {
 7     r = x - 7
 8   }
 9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



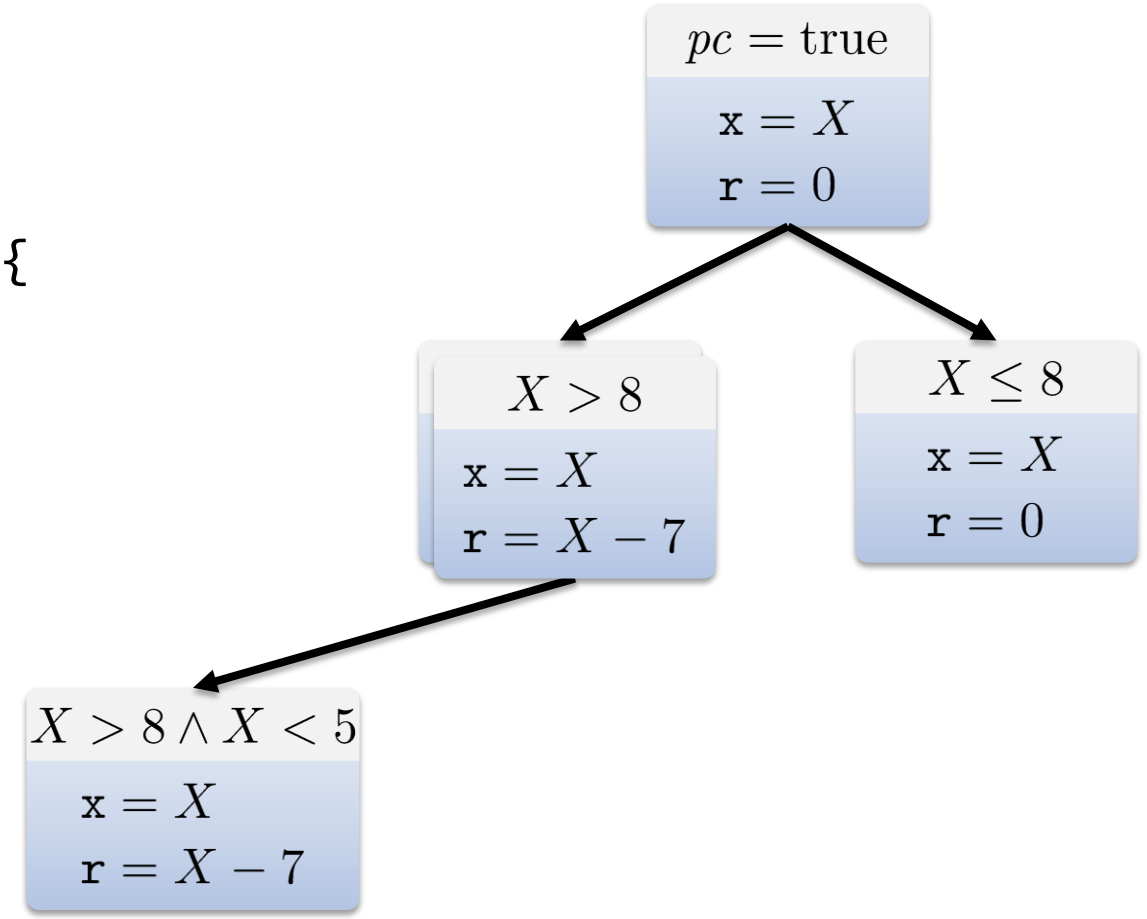
```
 1  int proc(int x) {
 2
 3   int r = 0
 5
 6   if (x > 8) {
 7     r = x - 7
 8   }
 9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



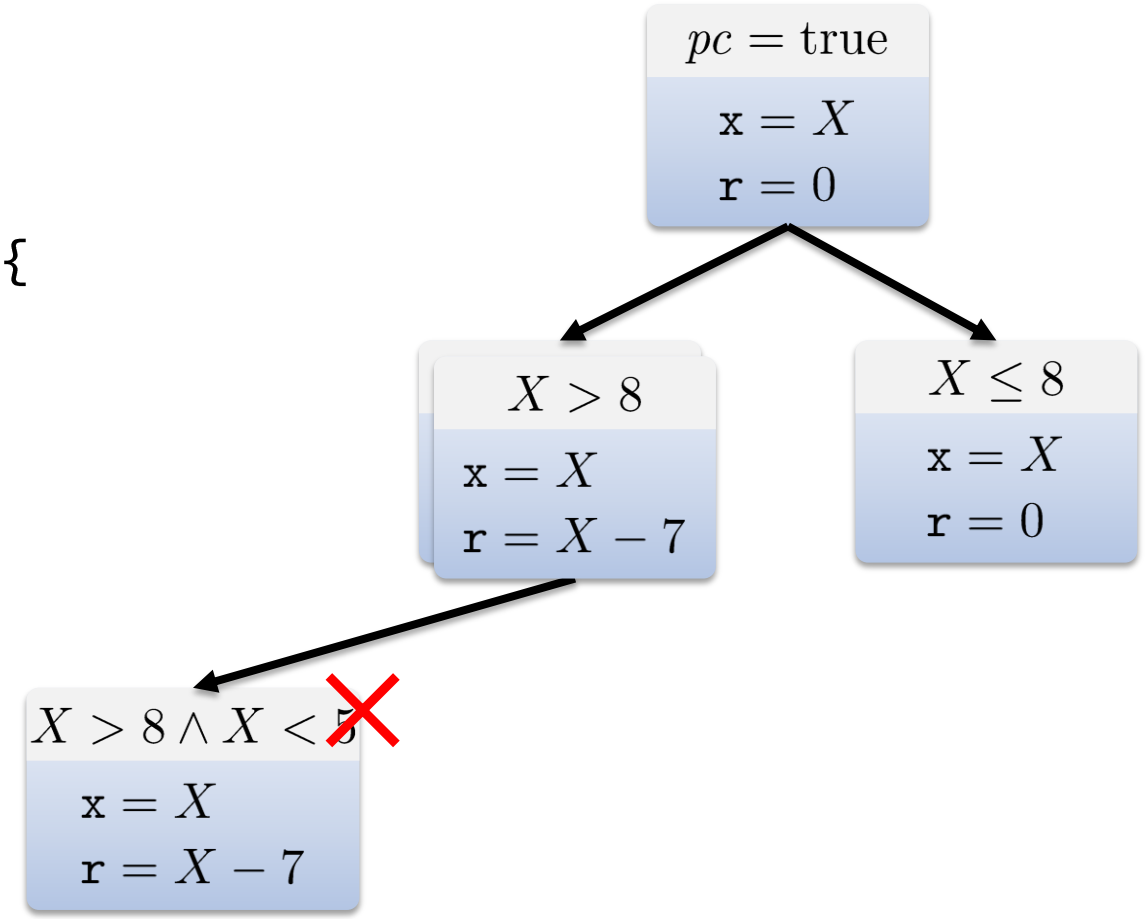
```
 1  int proc(int x) {
 2
 3   int r = 0
 5
 6   if (x > 8) {
 7     r = x - 7
 8   }
 9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



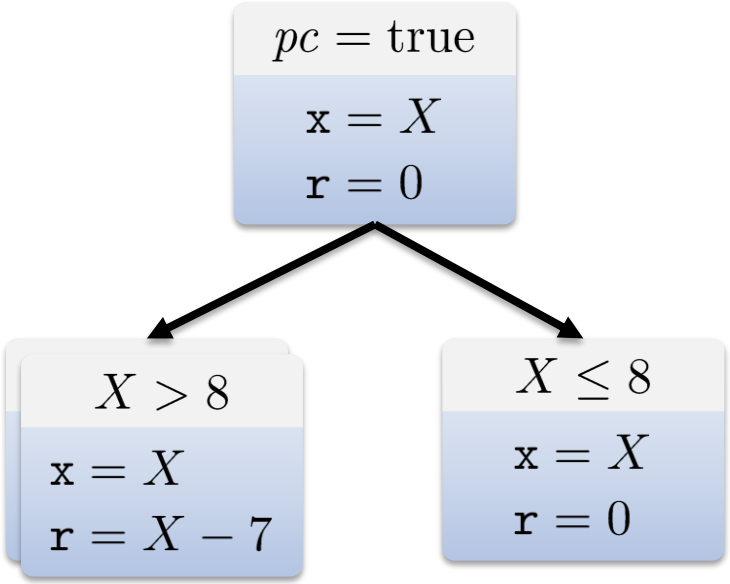
```
 1  int proc(int x) {
 2
 3   int r = 0
 5
 6   if (x > 8) {
 7     r = x - 7
 8   }
 9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



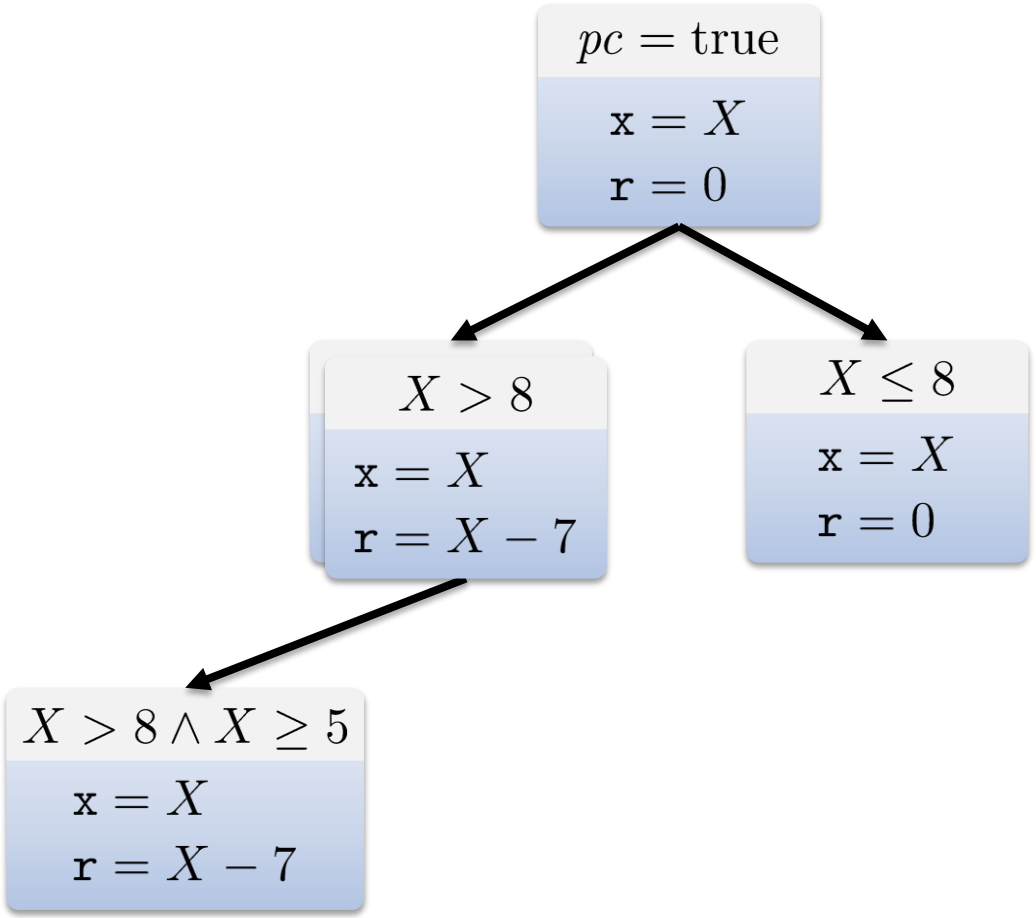
```
 1  int proc(int x) {
 2
 3   int r = 0
 5
 6   if (x > 8) {
 7     r = x - 7
 8   }
 9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



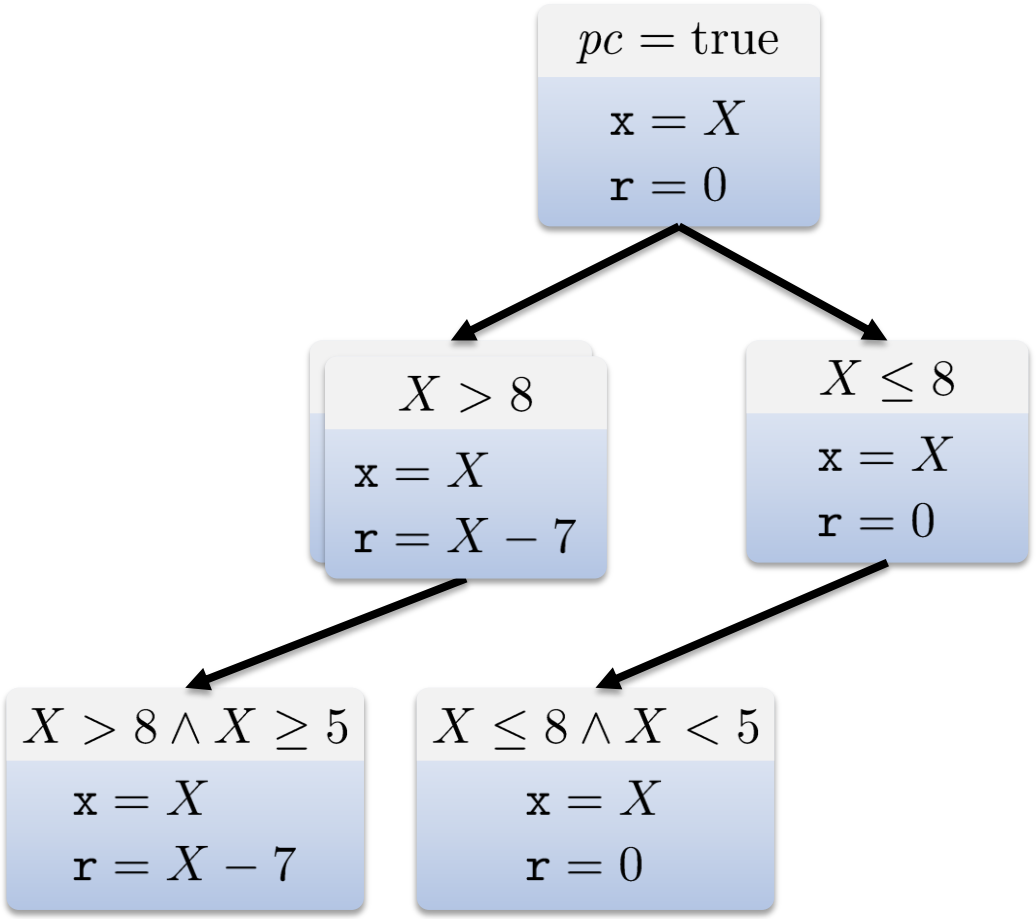
```
 1  int proc(int x) {
 2
 3   int r = 0
 5
 6   if (x > 8) {
 7     r = x - 7
 8   }
 9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



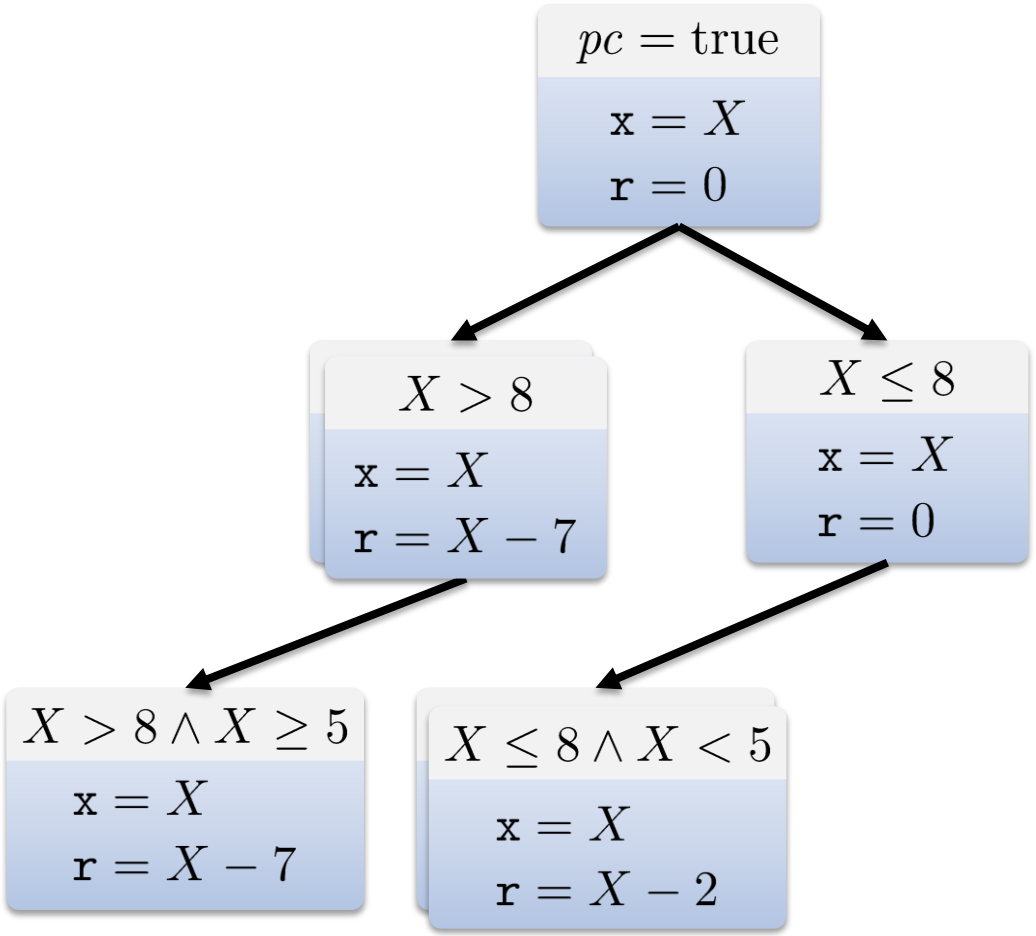
```
 1  int proc(int x) {
 2
 3   int r = 0
 5
 6   if (x > 8) {
 7     r = x - 7
 8   }
 9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



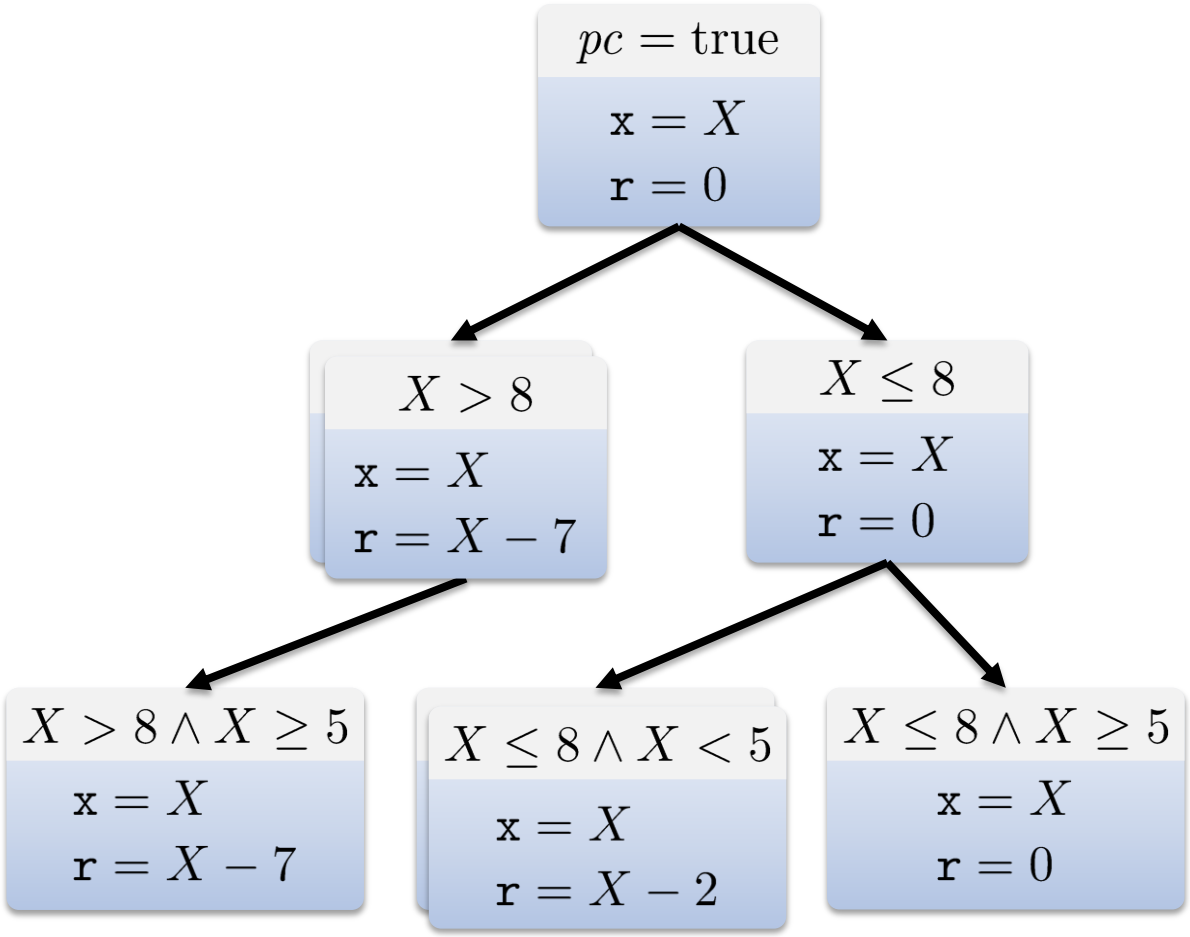
```
 1  int proc(int x) {
 2
 3   int r = 0
 5
 6   if (x > 8) {
 7     r = x - 7
 8   }
 9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



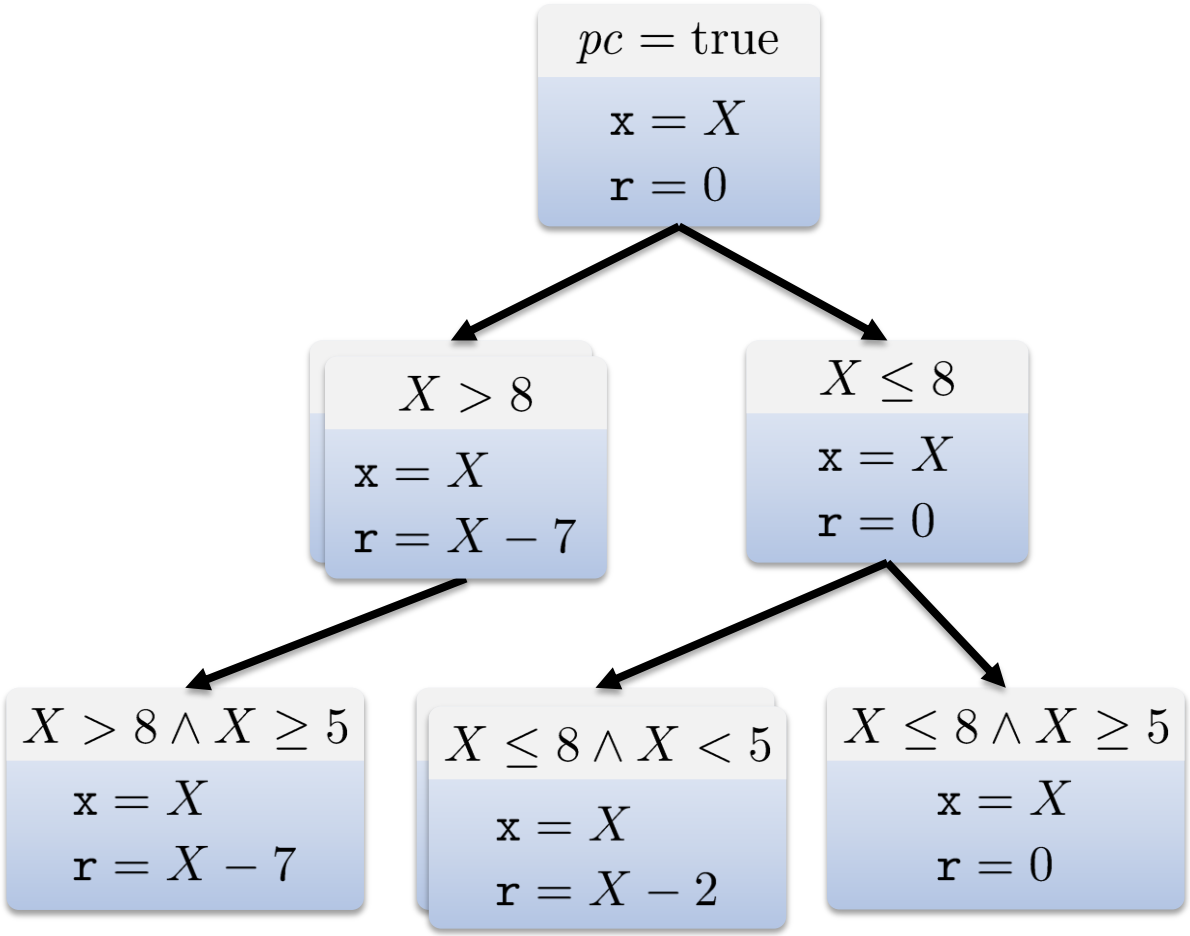
```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



```
 1  int proc(int x) {
 2
 3   int r = 0
 5
 6   if (x > 8) {
 7     r = x - 7
 8   }
 9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



Satisfying assignments:

`X = 9` `X = 4` `X = 7`

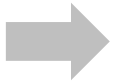
Test cases:

`proc(9)` `proc(4)` `proc(7)`

Finding Bugs

- Symbolic execution enumerates paths
  - *Runs into bugs that trigger whenever path executes*
  - *Assertions, buffer overflows, division by zero, etc., require specific conditions*
- Error conditions
  - *Treat assertions as conditions*

```
assert x != NULL
```



```
if (x == NULL)
    abort();
```

  - *Creates explicit error paths*

Finding Bugs

- Instrument program with properties
  - *Translate any safety property to reachability*
- Division by zero

```
y = 100 / x
```

```
assert x != 0
y = 100 / x
```

Finding Bugs

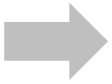
- Instrument program with properties
  - *Translate any safety property to reachability*
- Division by zero

```
y = 100 / x
```

```
assert x != 0
y = 100 / x
```

- Buffer overflows

```
a[x] = 10
```



```
assert x >= 0 && x < len(a)
```

Finding Bugs

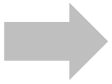
- Instrument program with properties
  - *Translate any safety property to reachability*
- Division by zero

```
y = 100 / x
```

```
assert x != 0
y = 100 / x
```

- Buffer overflows

```
a[x] = 10
```



```
assert x >= 0 && x < len(a)
```

- Implementation is usually implicit

Symbolic Execution Algorithms

- Static symbolic execution
  - *Simulate execution on program source code*
  - *Computes strongest postconditions from entry point*
- Dynamic symbolic execution (DSE)
  - *Run / interpret the program with concrete state*
  - *Symbolic state computed in parallel (“concolic”)*
  - *Solver generates new concrete state*
- DSE-Flavors
  - *EXE-style [Cadar et al. ‘06] vs. DART [Godefroid et al. ‘05]*

EXE

$$pc = \text{true}$$

$$S(\mathtt{x}) = X \quad C(\mathtt{x}) = 1 \quad C(\mathtt{r}) = 0$$

```
 1  int proc(int x) {
 2
 3   int r = 0
 5
 6   if (x > 8) {
 7     r = x - 7
 8   }
 9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```

EXE

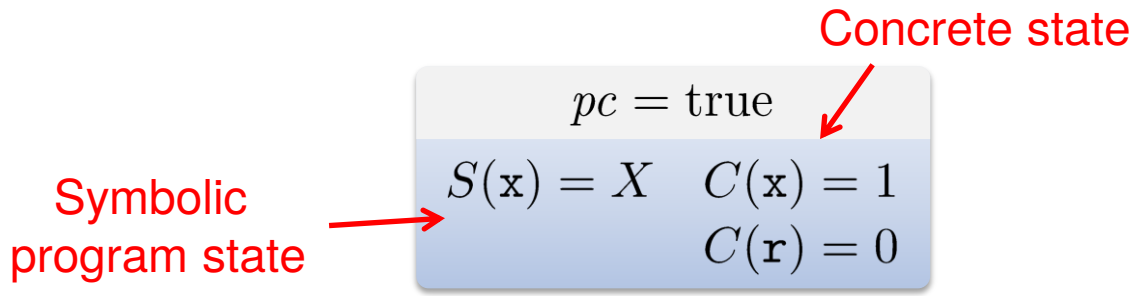
```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```

Symbolic program state

$$pc = \text{true}$$

$$S(\mathtt{x}) = X \quad C(\mathtt{x}) = 1$$
$$C(\mathtt{r}) = 0$$

EXE




```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15  }
```

EXE

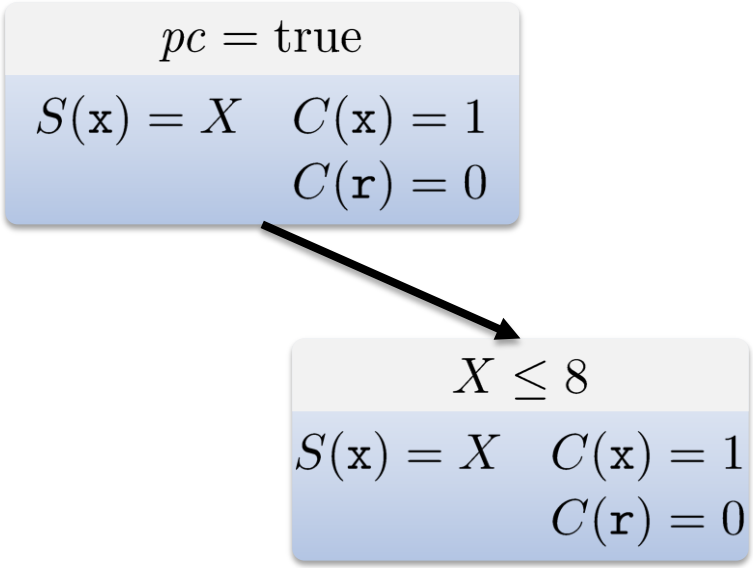
$$pc = \text{true}$$

$$S(\mathtt{x}) = X \quad C(\mathtt{x}) = 1 \quad C(\mathtt{r}) = 0$$

```
 1  int proc(int x) {
 2
 3   int r = 0
 5
 6   if (x > 8) {
 7     r = x - 7
 8   }
 9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```

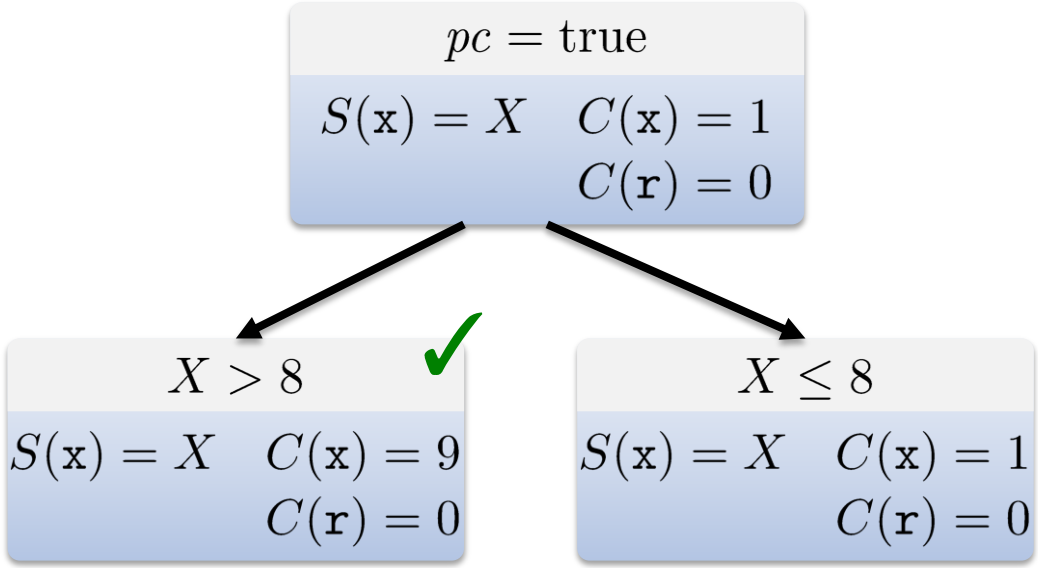
EXE

```
 1  int proc(int x) {
 2
 3   int r = 0
 5
 6   if (x > 8) {
 7     r = x - 7
 8   }
 9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



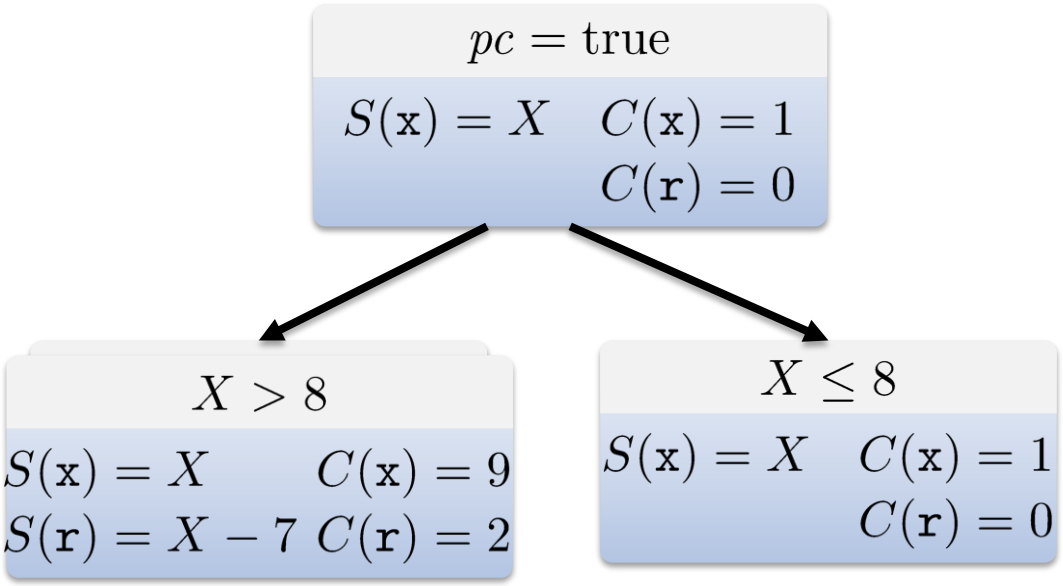
EXE

```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



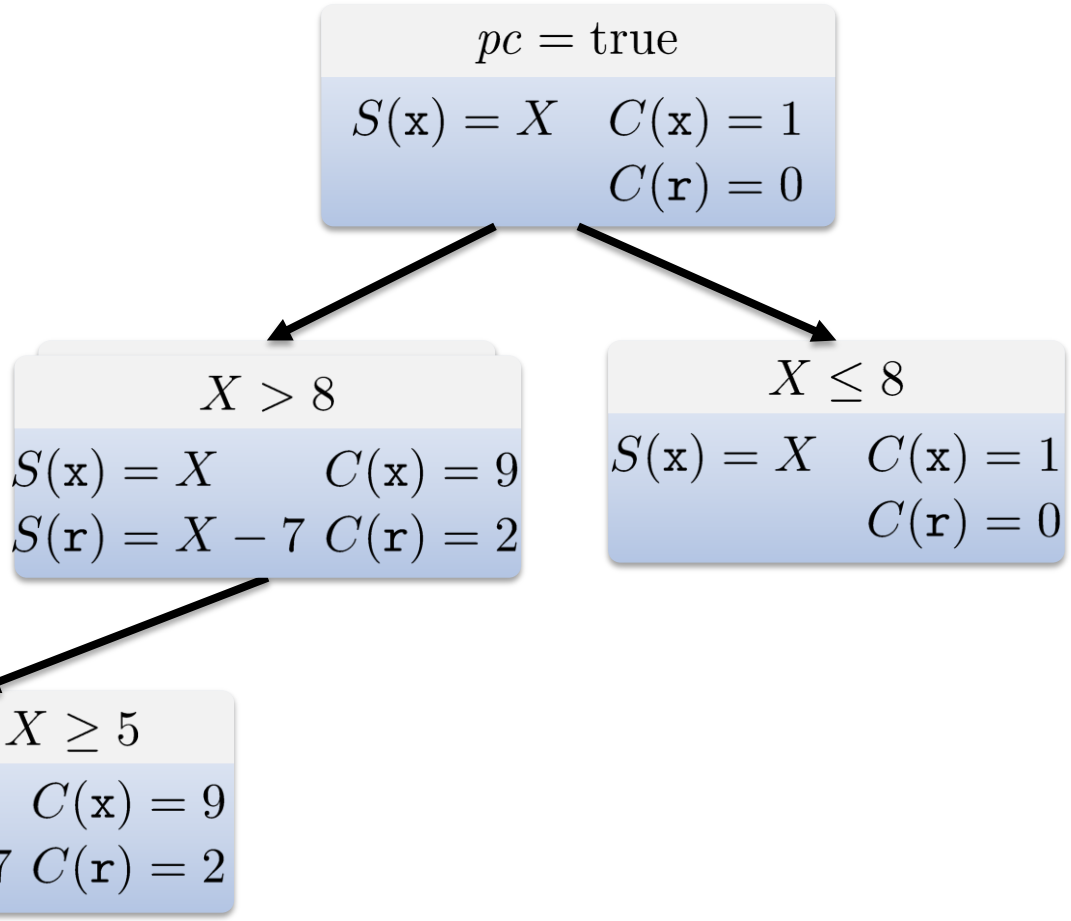
EXE

```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



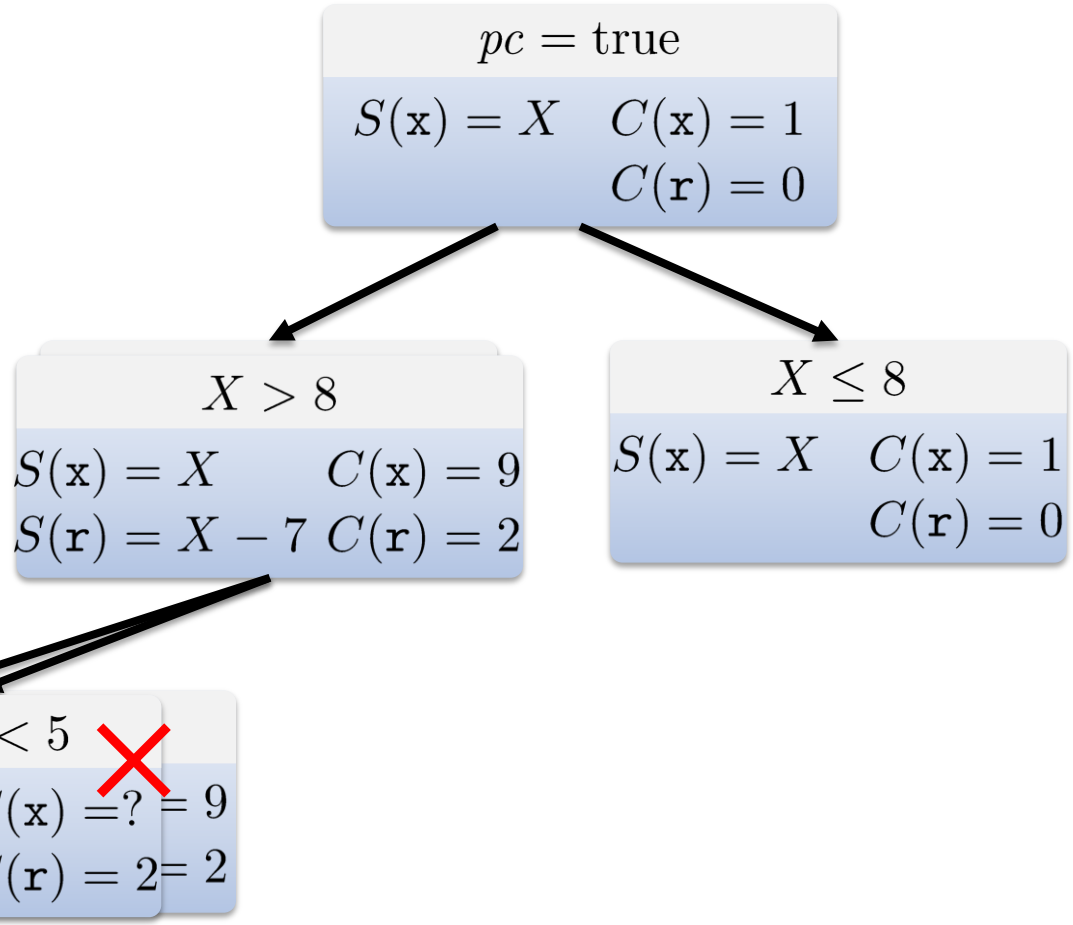
EXE

```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



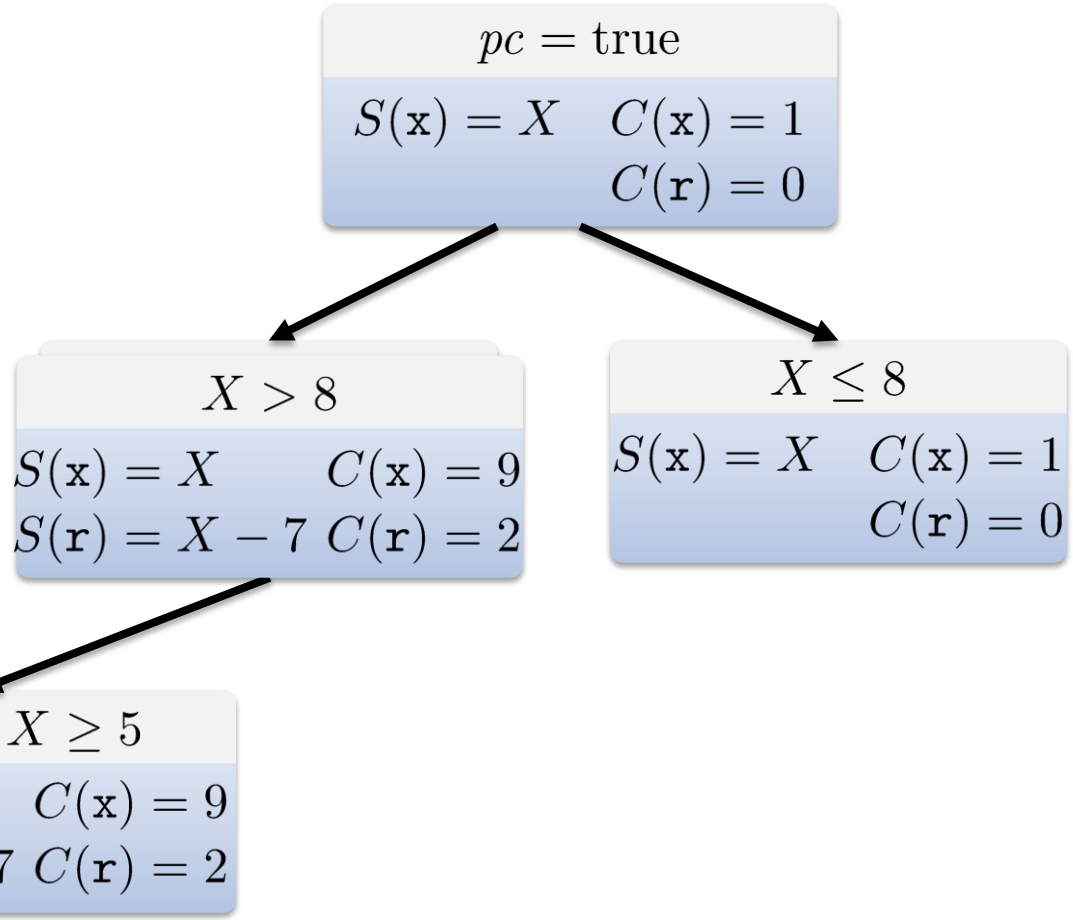
EXE

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5)
11     r = x -
12   }
13
14   return r;
15 }
```



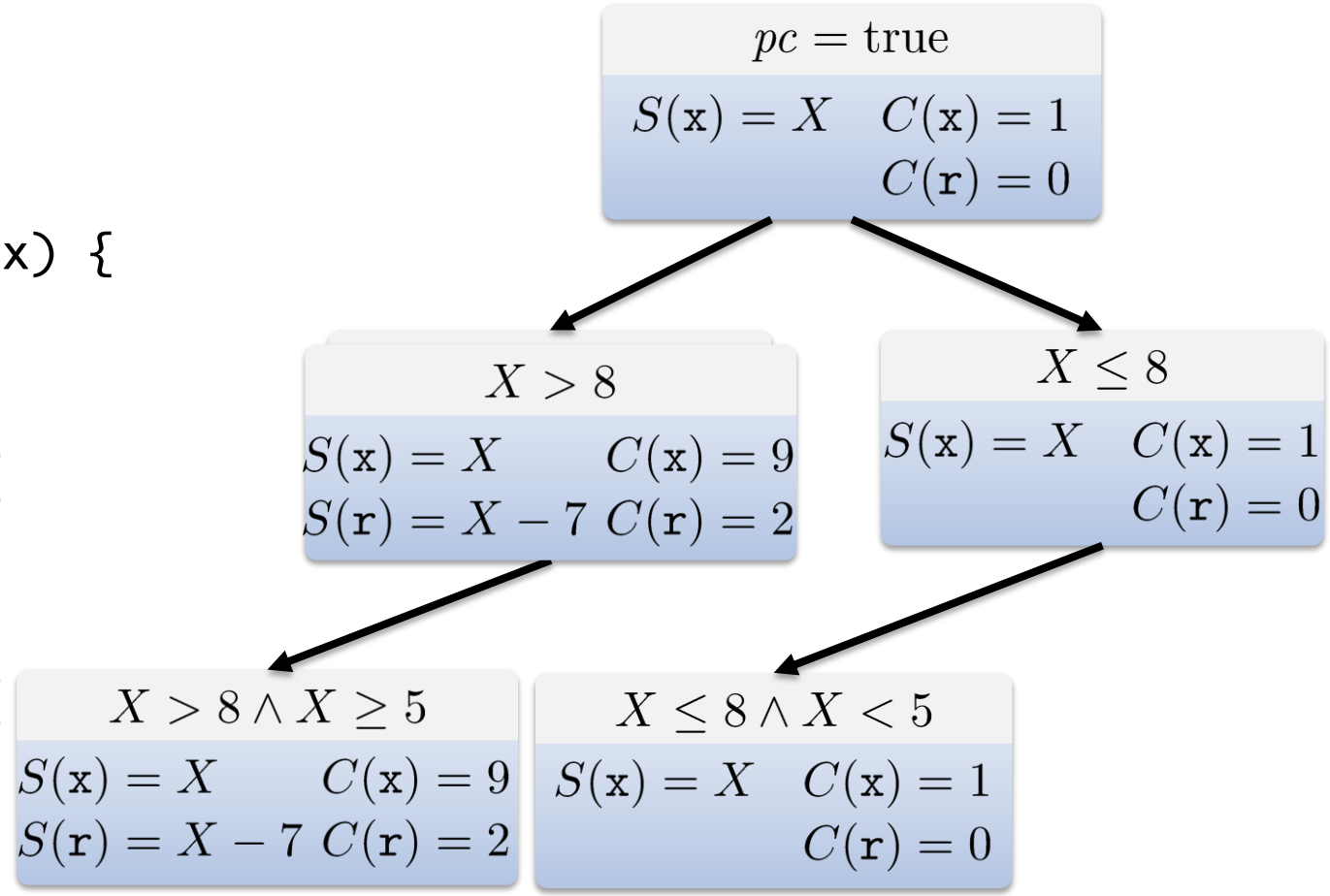
EXE

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



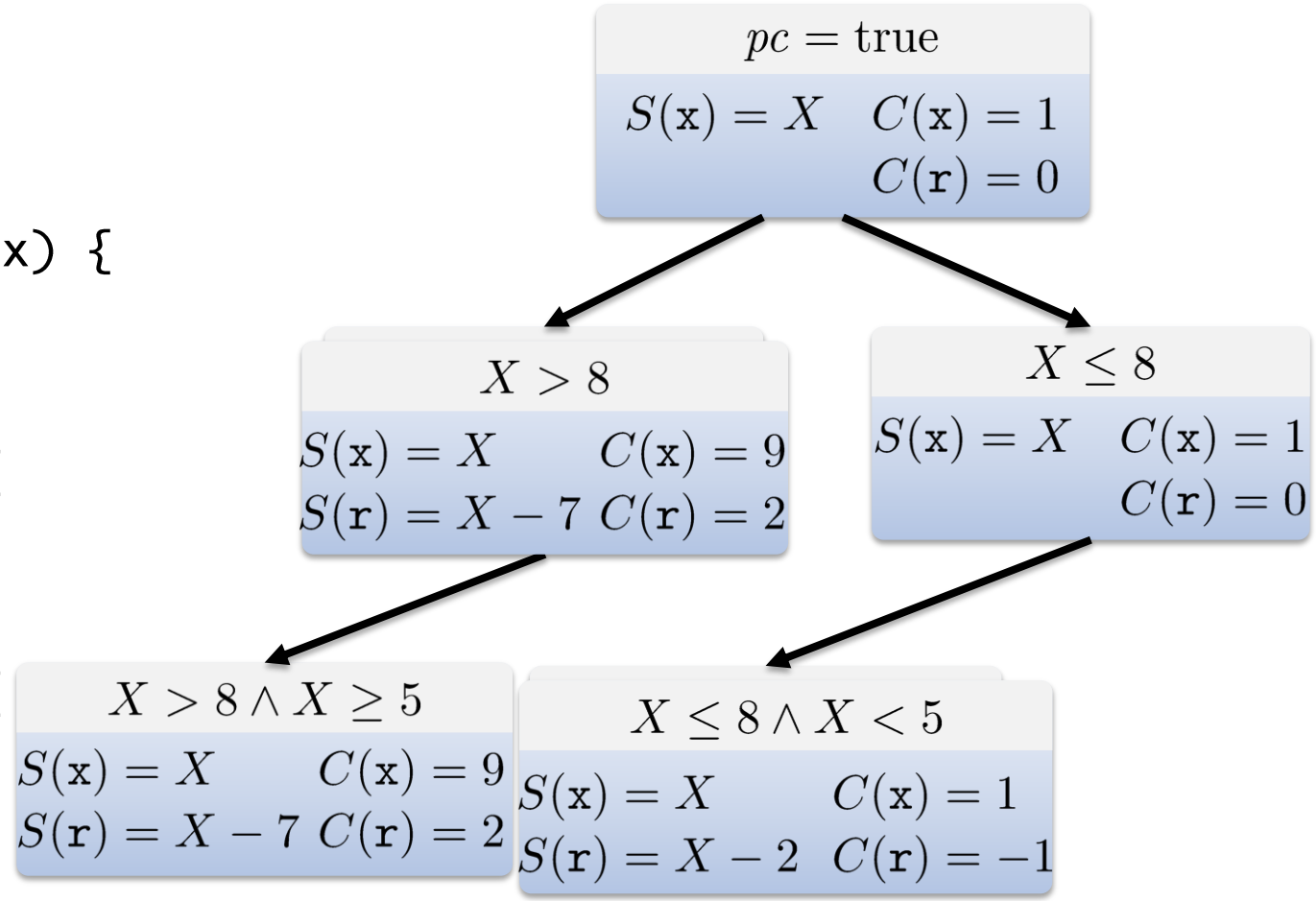
EXE

```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



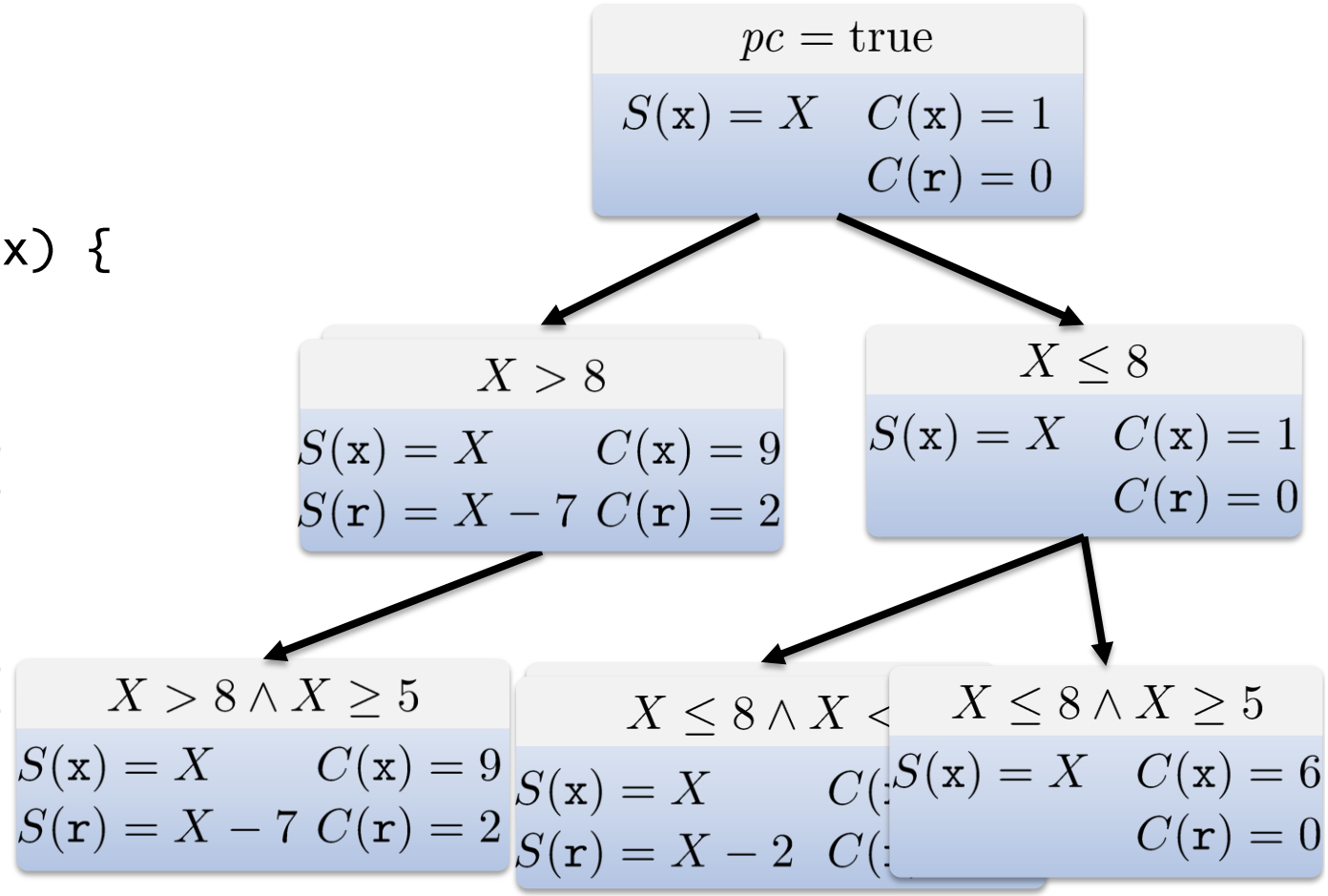
EXE

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



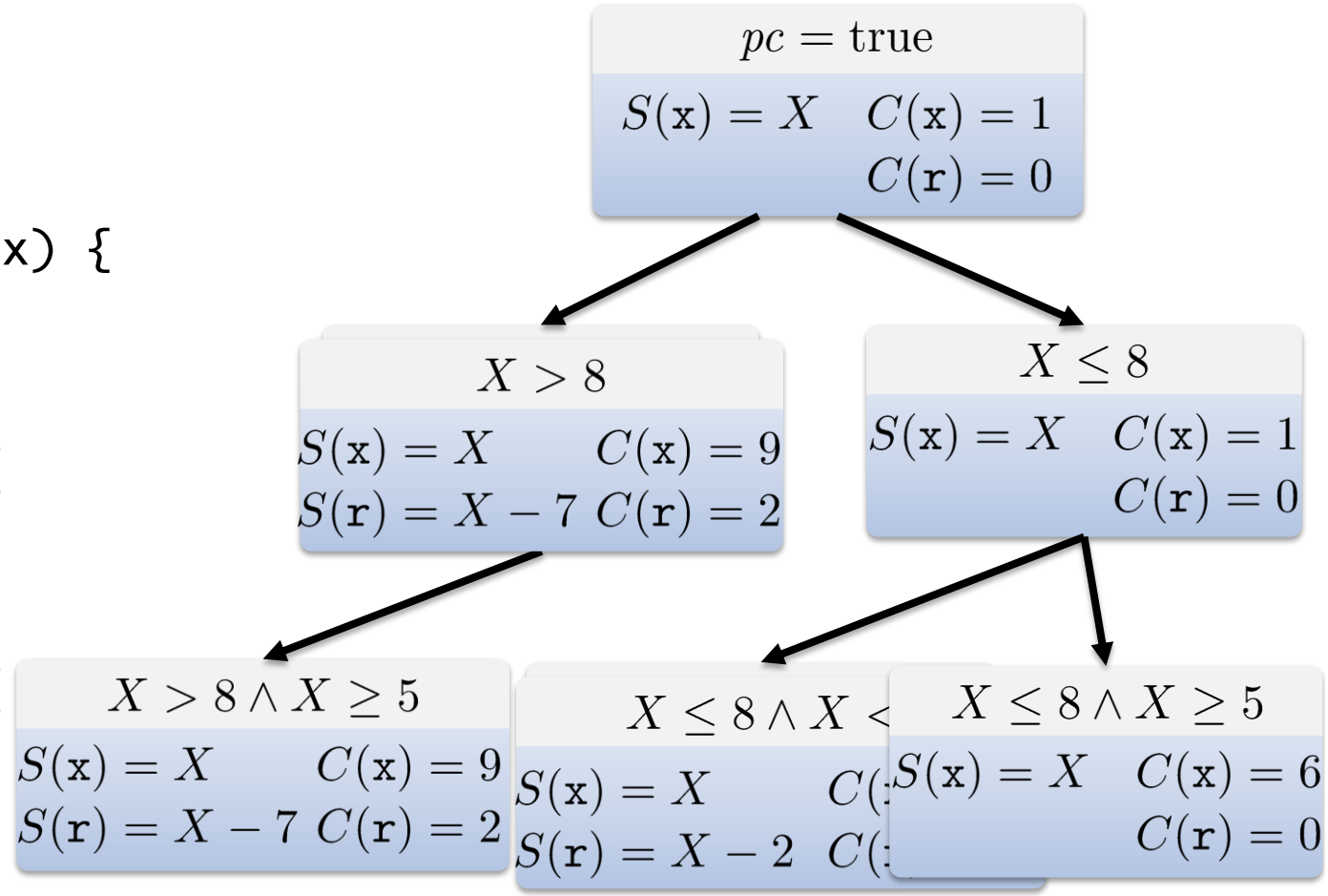
EXE

```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



EXE

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



Satisfying assignments:

X = 9 X = 1 X = 6

Test cases:

proc(9) proc(1) proc(6)

DART

$$pc = \text{true}$$

$$S(\texttt{x}) = X \quad C(\texttt{x}) = 1$$
$$C(\texttt{r}) = 0$$

```
 1  int proc(int x) {
 2
 3   int r = 0
 5
 6   if (x > 8) {
 7     r = x - 7
 8   }
 9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```

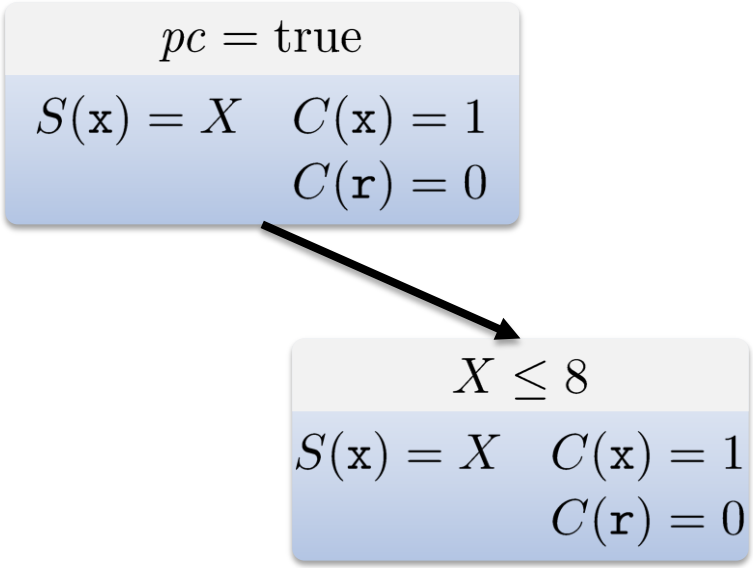
Path condition:

Test cases:

`proc(1)`

DART

```
 1  int proc(int x) {
 2
 3   int r = 0
 5
 6   if (x > 8) {
 7     r = x - 7
 8   }
 9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



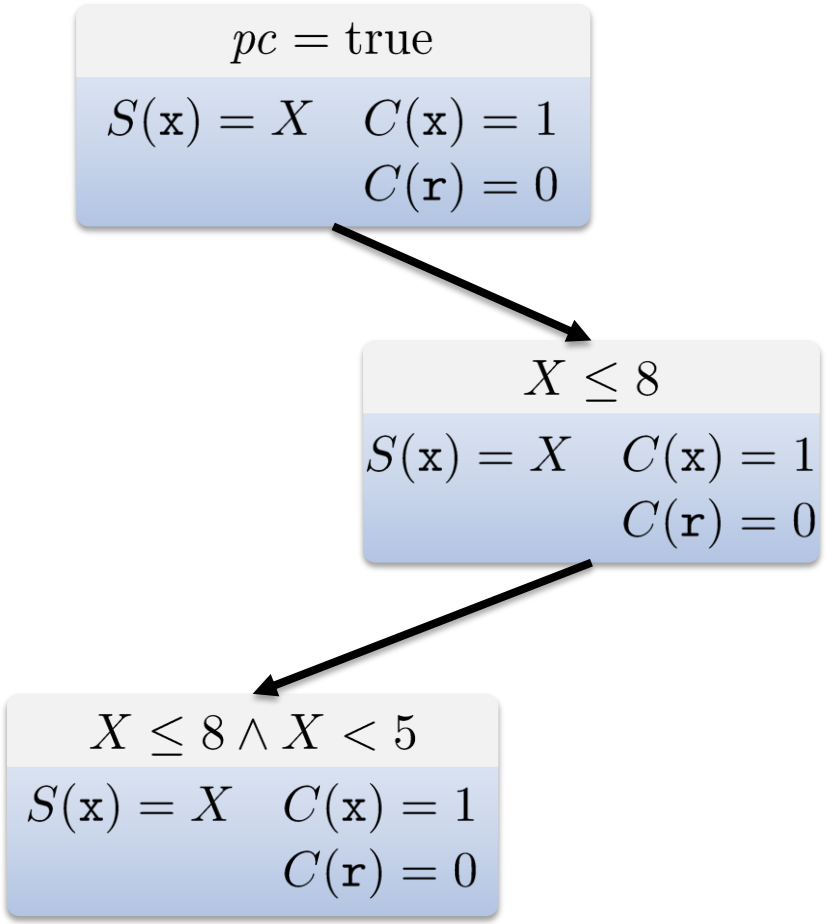
Path condition:

Test cases:

`proc(1)`

DART

```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



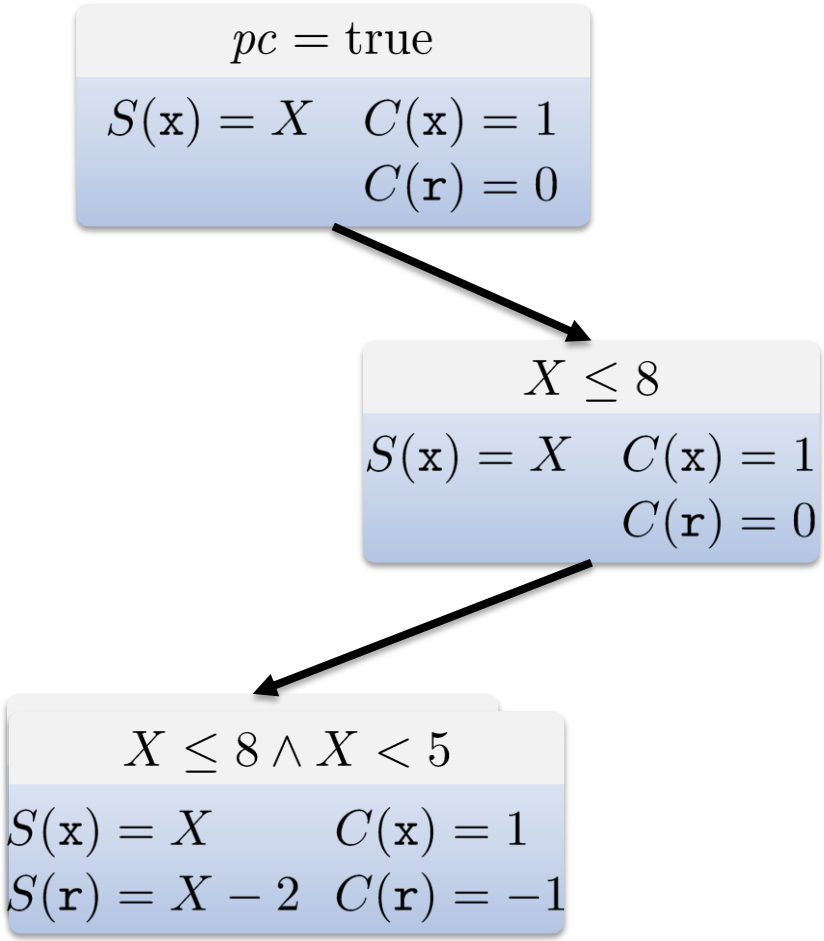
Path condition:

Test cases:

`proc(1)`

DART

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



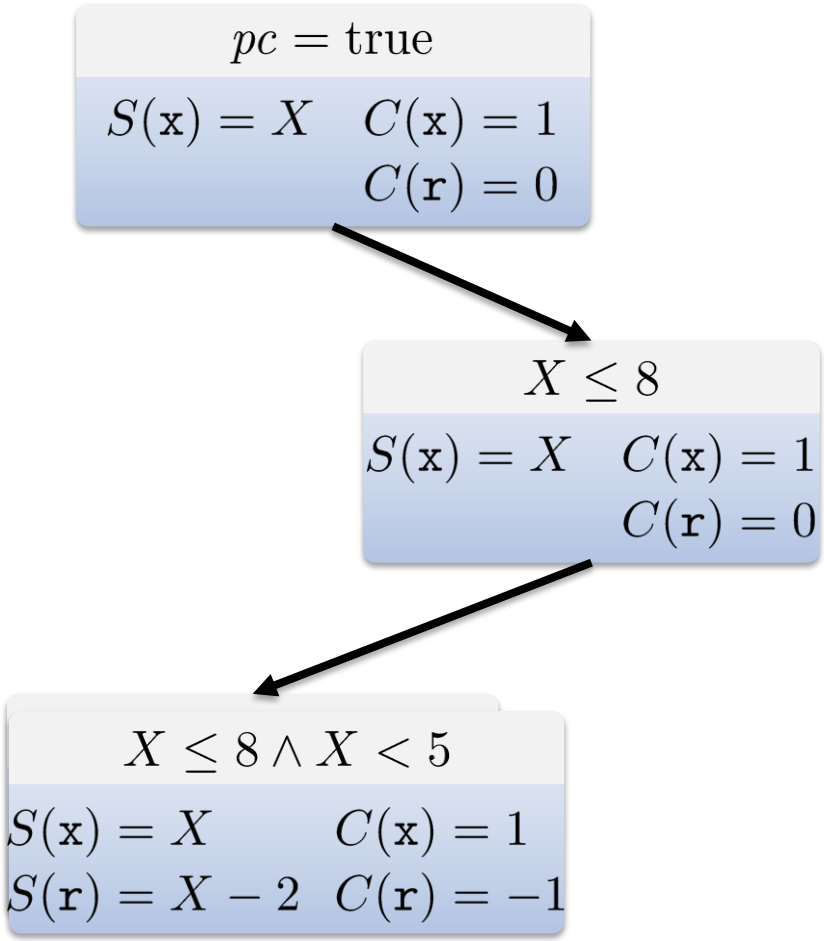
Path condition:

Test cases:

`proc(1)`

DART

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



Path condition:

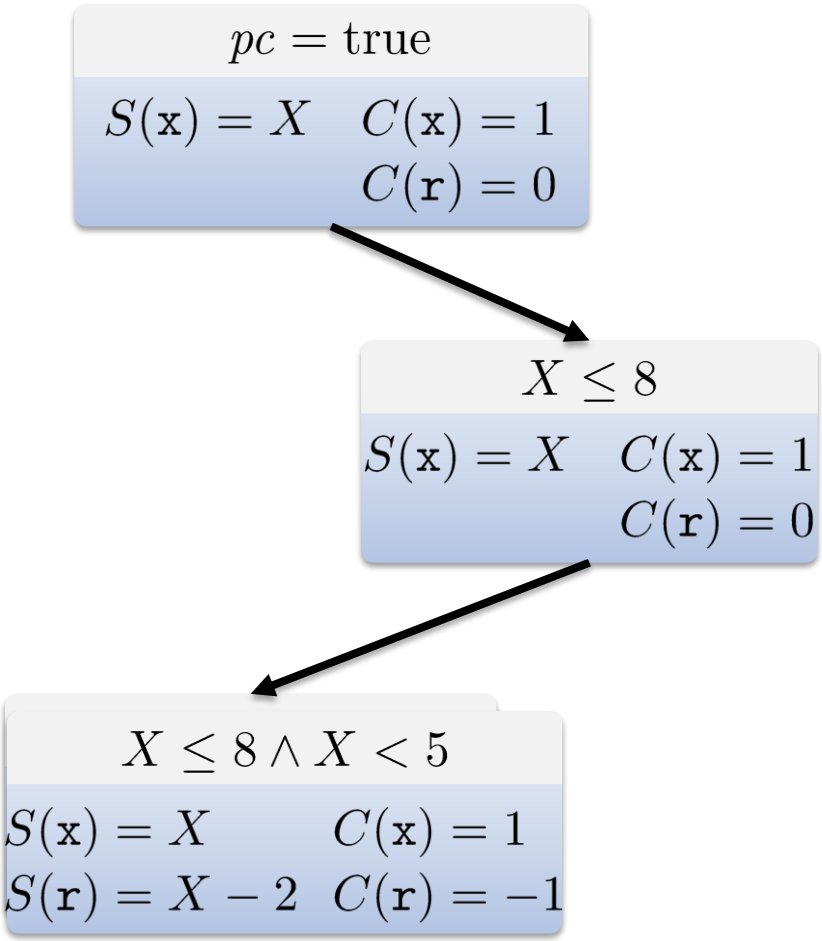
$$X \leq 8 \wedge \neg(X < 5)$$

Test cases:

`proc(1)`

DART

```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



Path condition:

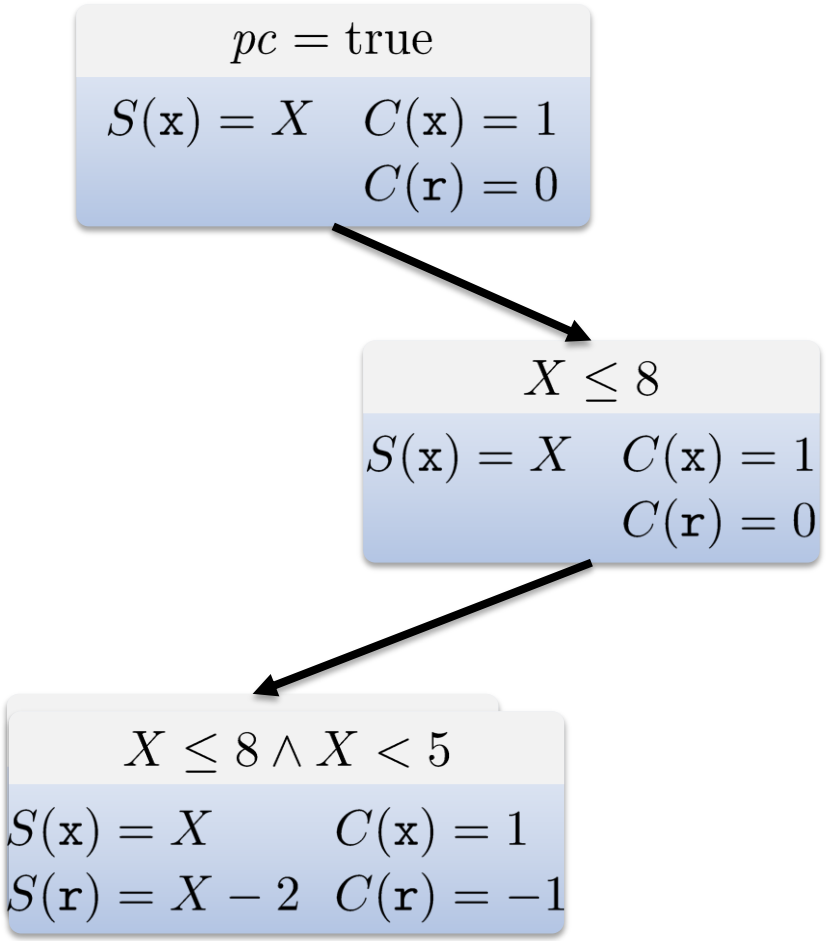
$$X \leq 8 \wedge \neg(X < 5)$$ ✓

Test cases:

`proc(1)`

DART

```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



Path condition:

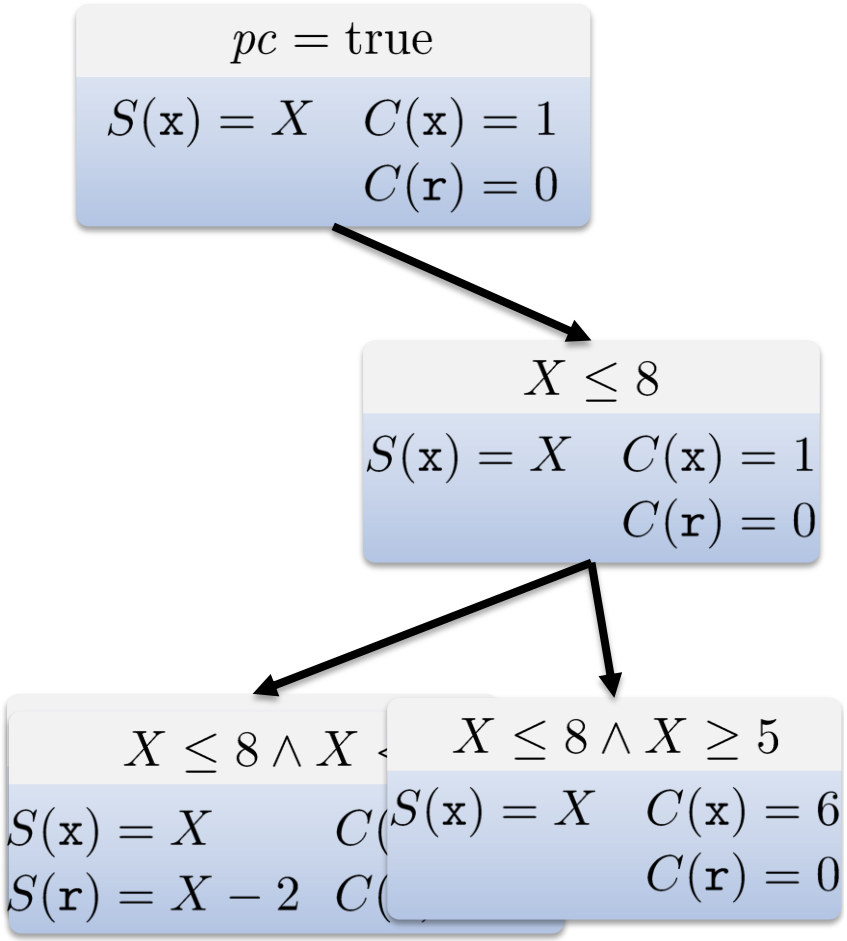
$$X \leq 8 \wedge \neg(X < 5)$$ ✓

Test cases:

proc(1) proc(6)

DART

```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



Path condition:

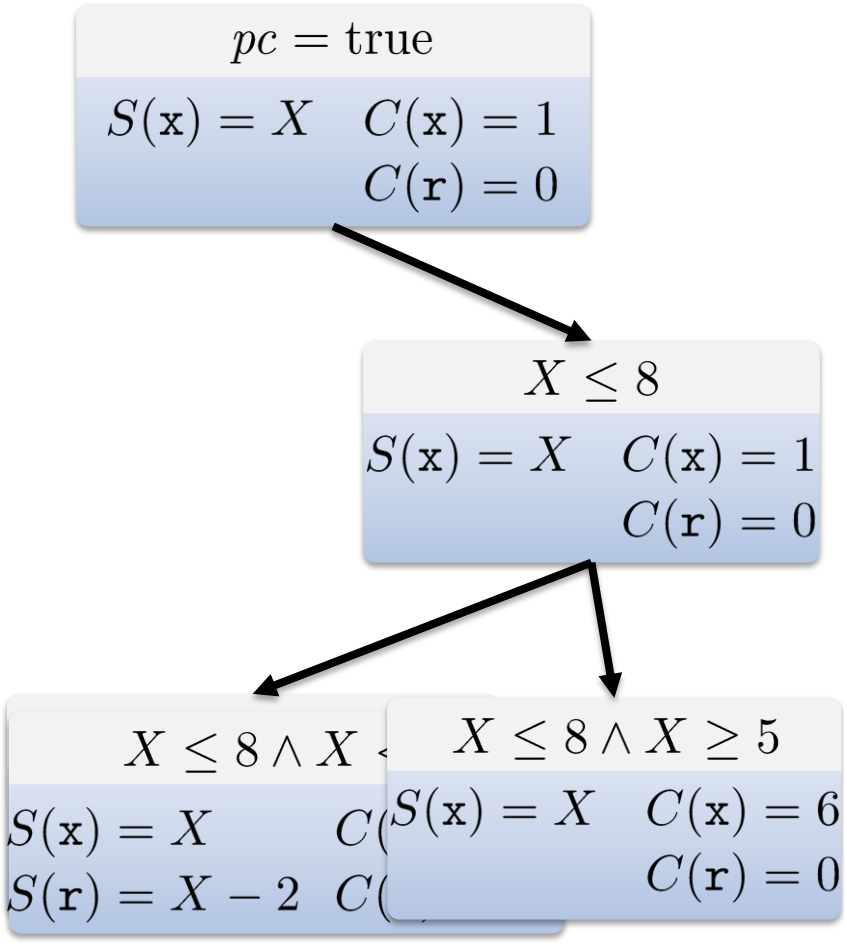
$$X \leq 8 \wedge \neg(X < 5)$$

Test cases:

`proc(1)` `proc(6)`

DART

```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



Path condition:

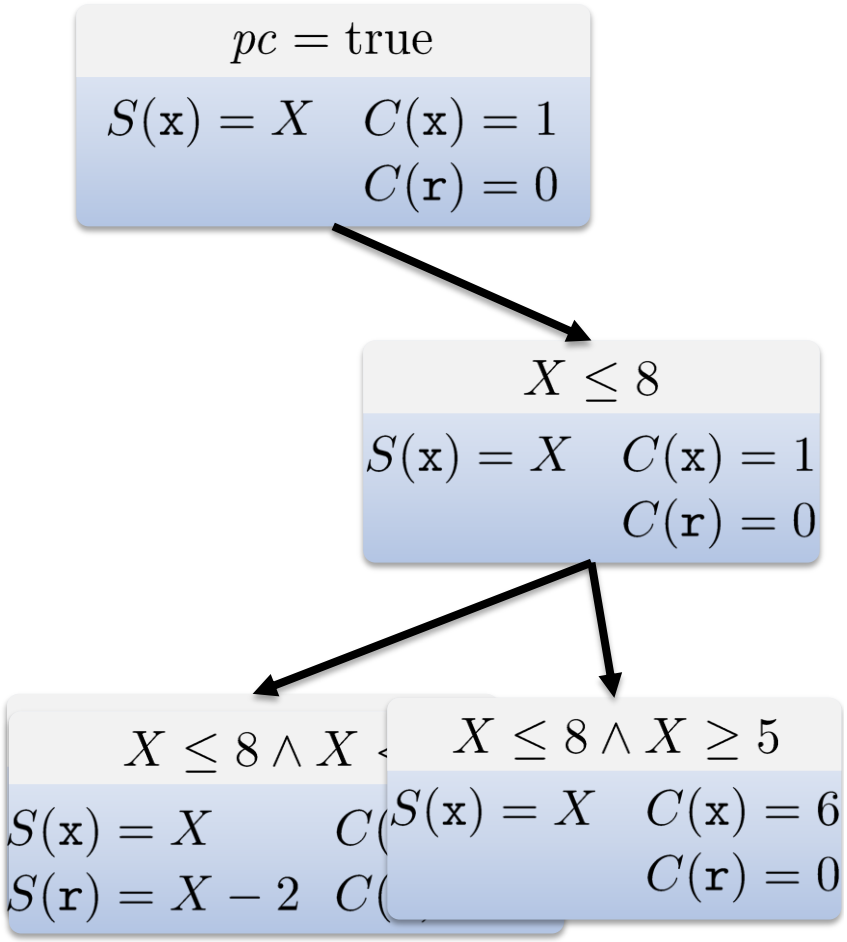
$\neg(X \leq 8)$

Test cases:

`proc(1)` `proc(6)`

DART

```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



Path condition:

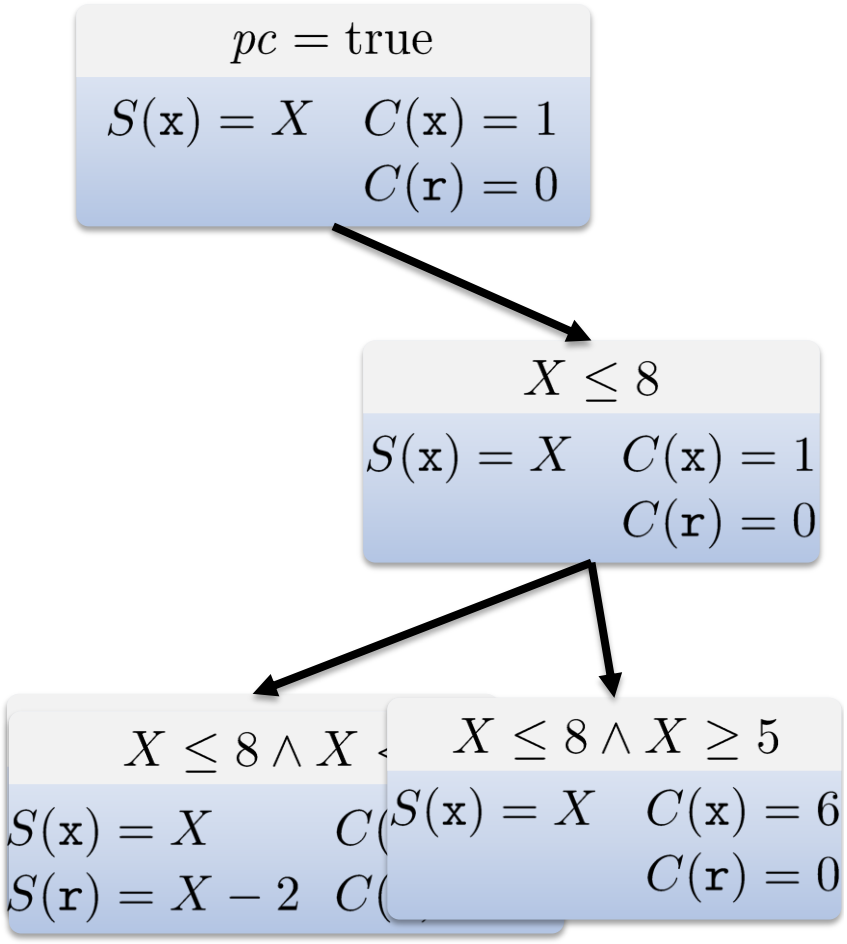
$\neg(X \leq 8)$ ✓

Test cases:

proc(1) proc(6)

DART

```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



Path condition:

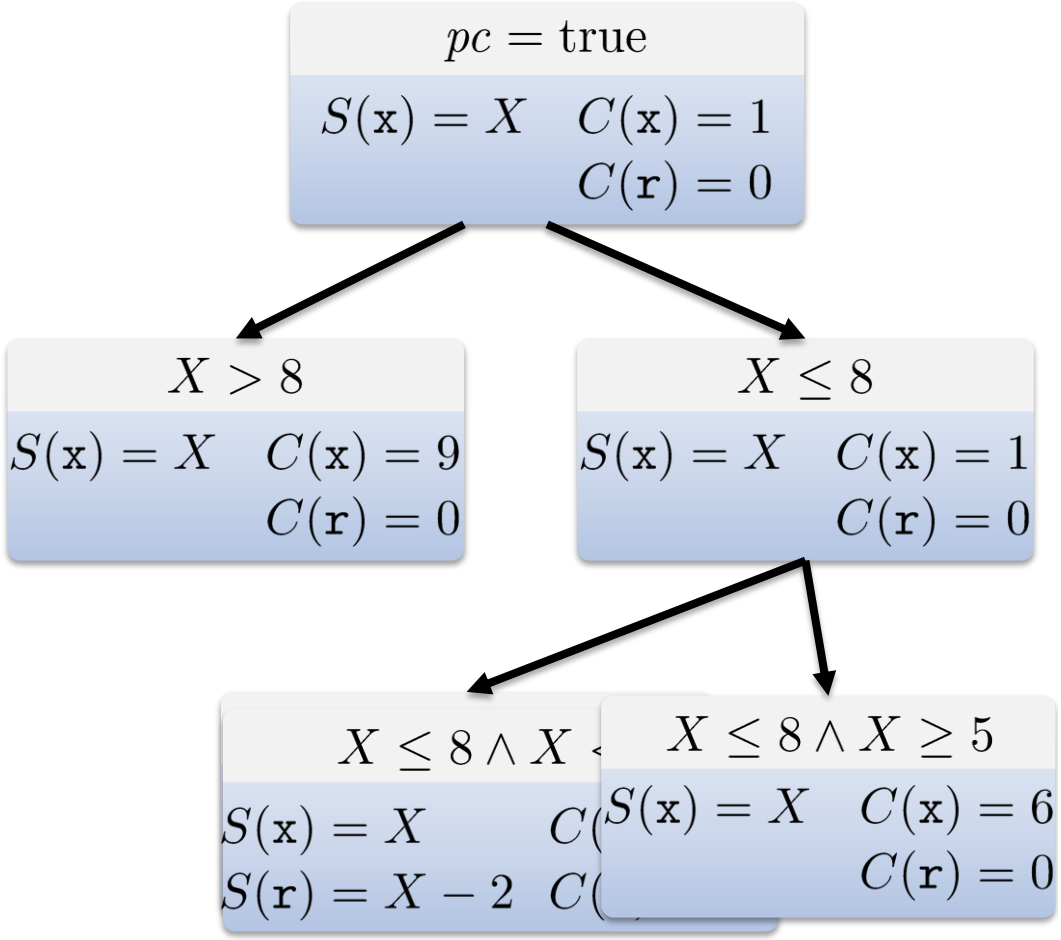
$\neg(X \leq 8)$ ✓

Test cases:

`proc(9)` `proc(1)` `proc(6)`

DART

```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



Path condition:

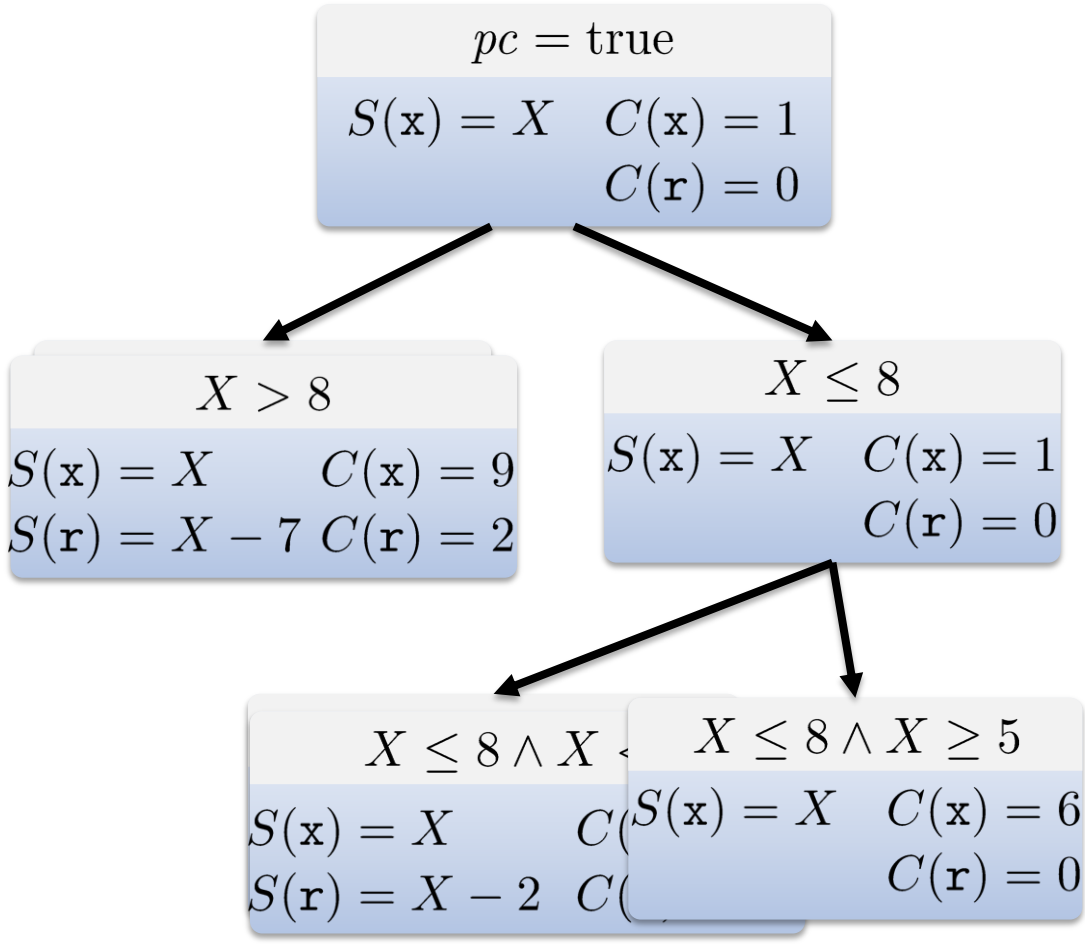
$$\neg(X \leq 8)$$

Test cases:

`proc(9)` `proc(1)` `proc(6)`

DART

```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



Path condition:

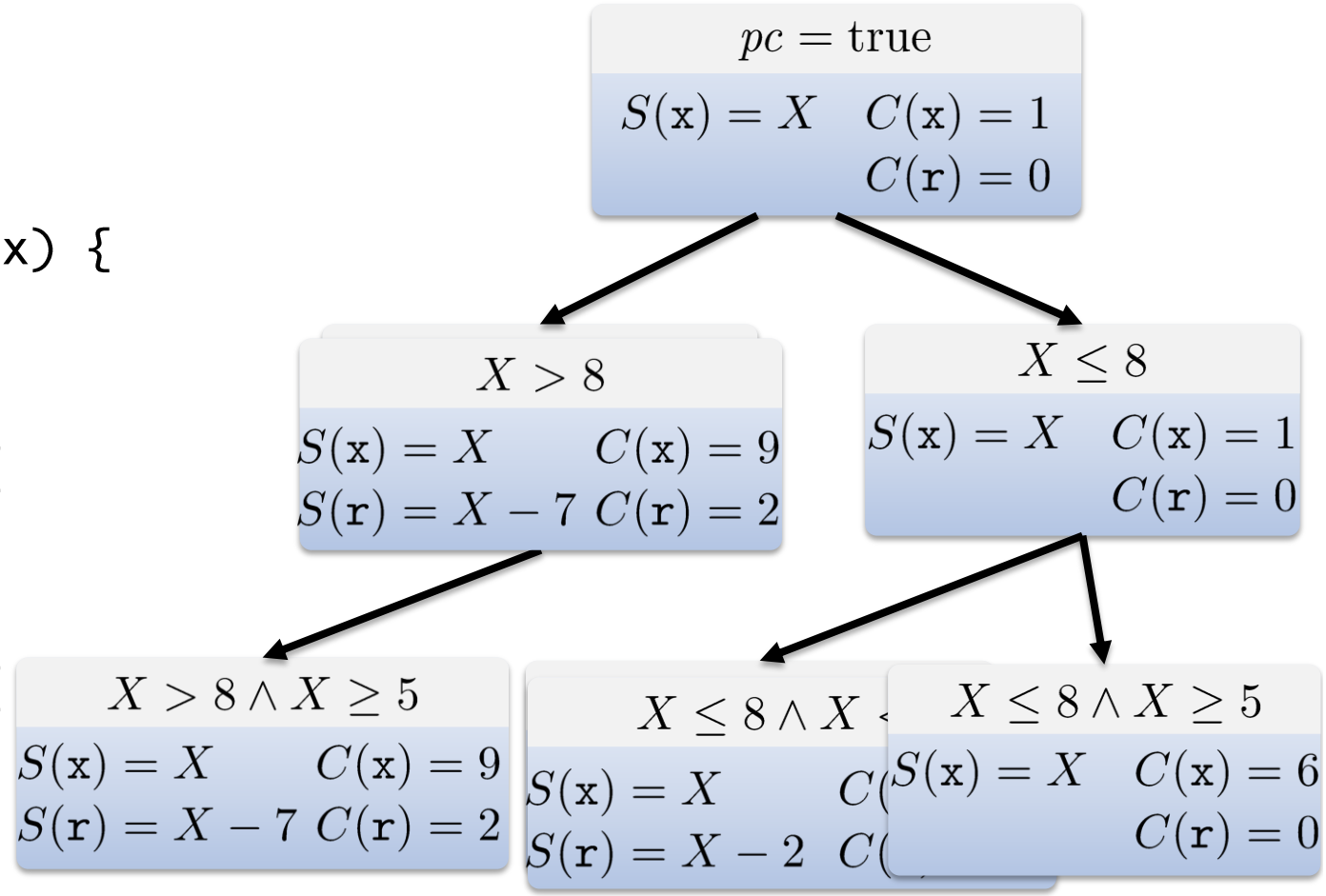
$$\neg(X \leq 8)$$

Test cases:

`proc(9)` `proc(1)` `proc(6)`

DART

```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



Path condition:

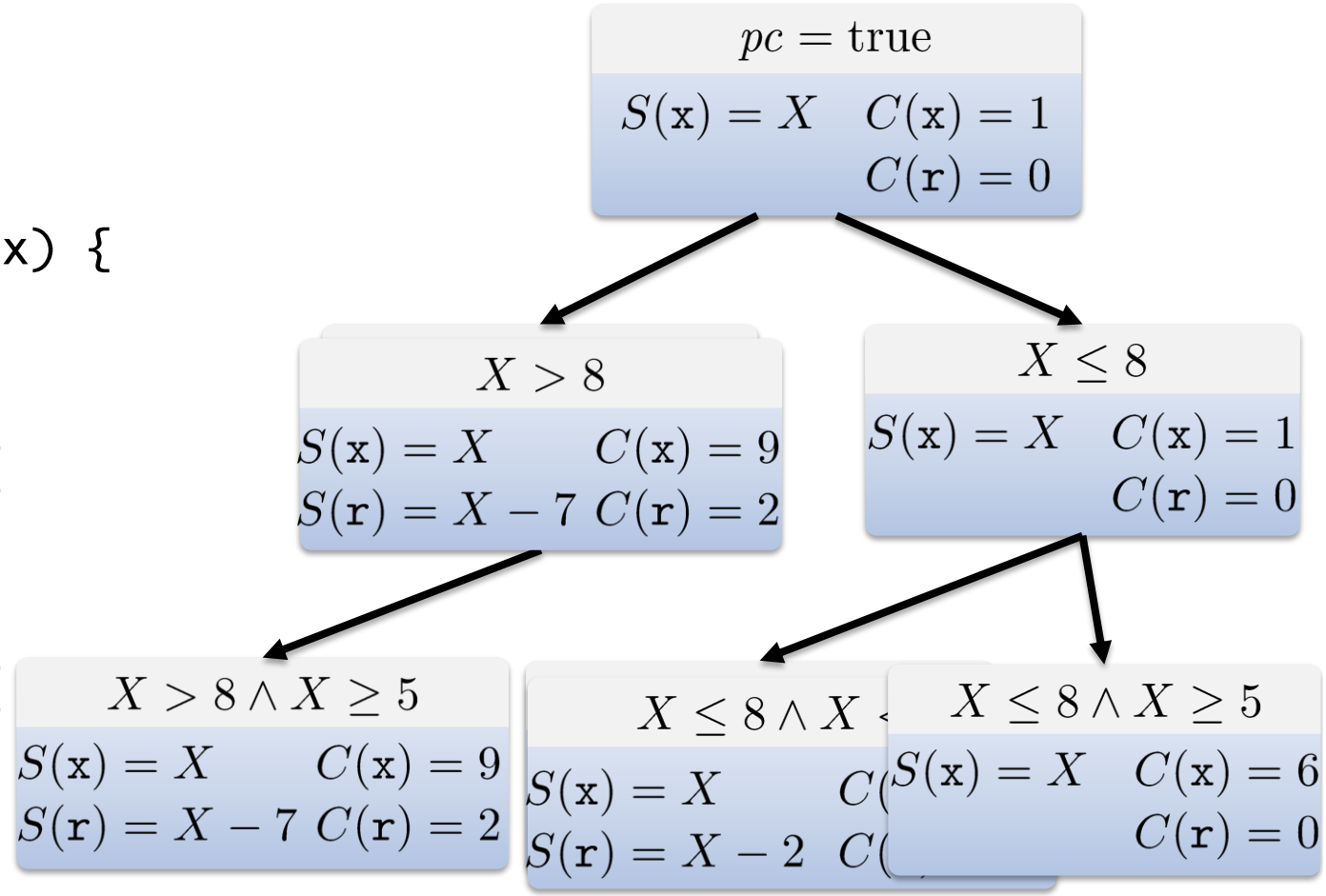
$$\neg(X \leq 8)$$

Test cases:

`proc(9)` `proc(1)` `proc(6)`

DART

```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



Path condition:

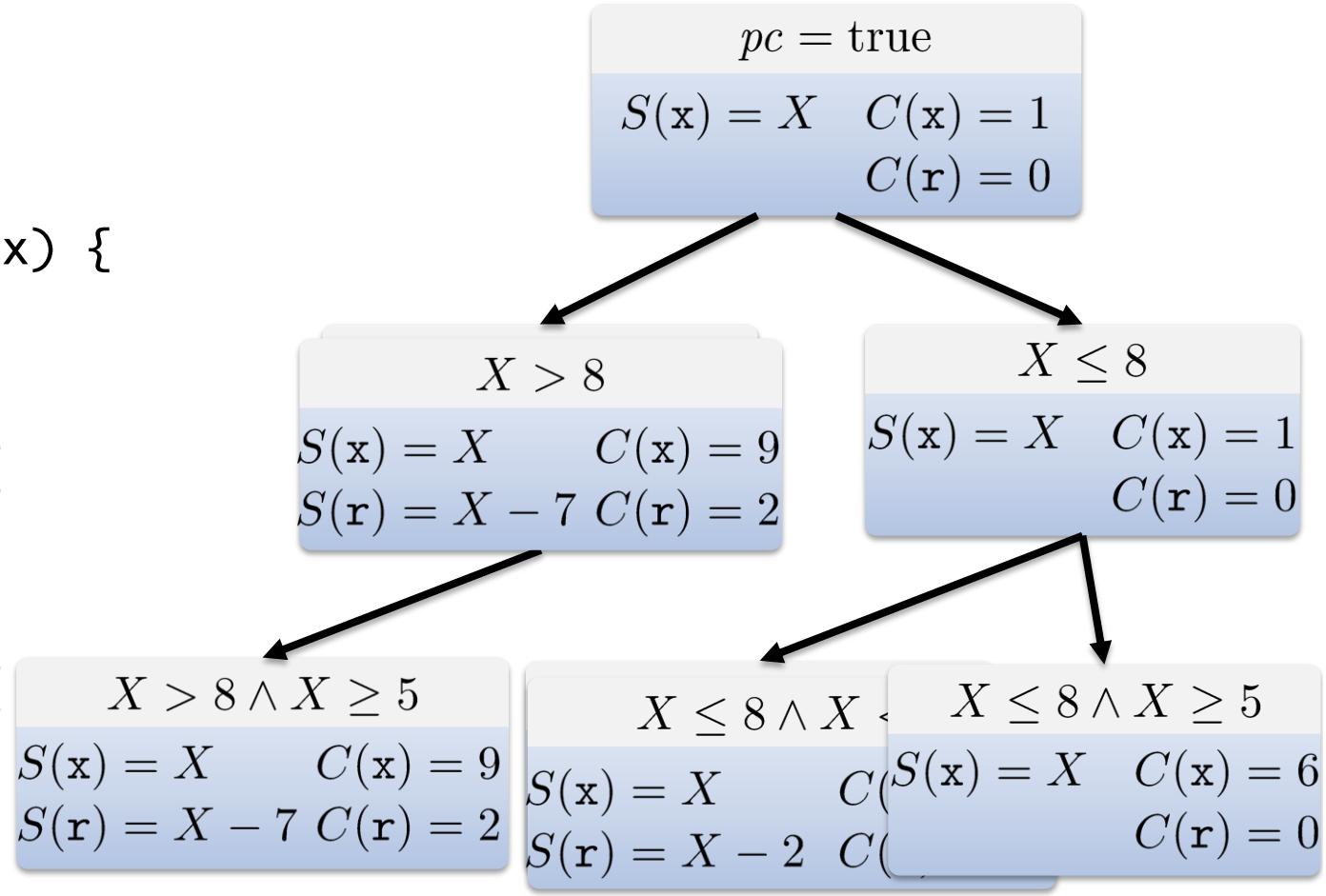
$$X > 8 \wedge \neg(X \geq 5)$$

Test cases:

`proc(9)` `proc(1)` `proc(6)`

DART

```
 1  int proc(int x) {
 2
 3    int r = 0
 5
 6    if (x > 8) {
 7      r = x - 7
 8    }
 9
10    if (x < 5) {
11      r = x - 2
12    }
13
14    return r;
15 }
```



Path condition:

$$X > 8 \wedge \neg(X \geq 5)$$

Test cases:

`proc(9)` `proc(1)` `proc(6)`

DART vs. EXE

- Complete execution from 1st step
- Deep exploration
  - *One query per run*
- Offline SE possible
  - *Follow recorded trace*
- Fine-grained control of execution
- Shallow exploration
  - *Many queries early on*
- Online SE
  - *SE and interpretation in lockstep*

Concretization

- Parts of input space can be kept concrete
  - *Reduces complexity*
  - *Focuses search*
- Expressions can be concretized at runtime
  - *Avoid expressions outside of SMT solver theories (non-linear etc.)*
- Sound but incomplete

Concretization (Example)

$$true$$

$$C(X) = 5$$

$$S(\texttt{m}) = X + 2 \qquad C(\texttt{m}) = 7$$

$$S(\texttt{size}) = Y \qquad C(\texttt{size}) = 256$$

```
if (m*m > size) {
    …
```

Concretization (Example)

$$true$$

$$C(X) = 5$$

$$S(\texttt{m}) = X + 2 \qquad C(\texttt{m}) = 7$$

$$S(\texttt{size}) = Y \qquad C(\texttt{size}) = 256$$

```
if (m*m > size) {
    …
```

$$(X + 2)(X + 2) > Y$$

Concretization (Example)

$true$	
	$C(X) = 5$
$S(\texttt{m}) = X + 2$	$C(\texttt{m}) = 7$
$S(\texttt{size}) = Y$	$C(\texttt{size}) = 256$

```
if (m*m > size) {
    …
```

$$(X + 2)(X + 2) > Y$$

$$(5 + 2)(5 + 2) > Y$$

Concretization (Example)

$$true$$

$$C(X) = 5$$

$$S(\texttt{m}) = X + 2 \qquad C(\texttt{m}) = 7$$

$$S(\texttt{size}) = Y \qquad C(\texttt{size}) = 256$$

```
if (m*m > size) {
    …
```

$$(X + 2)(X + 2) > Y$$

$$49 > Y$$

Concretization (Example)

$$true$$

$$C(X) = 5$$
$$S(\texttt{m}) = X + 2 \qquad C(\texttt{m}) = 7$$
$$S(\texttt{size}) = Y \qquad C(\texttt{size}) = 256$$

```
if (m*m > size) {
    …
```

$$(X + 2)(X + 2) > Y$$

$$49 > Y$$

$$49 > Y$$

$$C(X) = 5$$
$$S(\texttt{m}) = X + 2 \qquad C(\texttt{m}) = 7$$
$$S(\texttt{size}) = Y \qquad C(\texttt{size}) = 48$$

Concretization (Example)

$$true$$

$$\begin{aligned} C(X) &= 5 \\ S(\texttt{m}) = X + 2 \qquad C(\texttt{m}) &= 7 \\ S(\texttt{size}) = Y \qquad C(\texttt{size}) &= 256 \end{aligned}$$

```
if (m*m > size) {
  …
  if (m < 5) {
    …
```

$$49 > Y$$

$$\begin{aligned} C(X) &= 5 \\ S(\texttt{m}) = X + 2 \qquad C(\texttt{m}) &= 7 \\ S(\texttt{size}) = Y \qquad C(\texttt{size}) &= 48 \end{aligned}$$

Concretization (Example)

$$true$$

$$C(X) = 5$$
$$S(\texttt{m}) = X + 2 \qquad C(\texttt{m}) = 7$$
$$S(\texttt{size}) = Y \qquad C(\texttt{size}) = 256$$

```
if (m*m > size) {
  …
  if (m < 5) {
    …
```

$$X + 2 < 5$$

$$49 > Y$$

$$C(X) = 5$$
$$S(\texttt{m}) = X + 2 \qquad C(\texttt{m}) = 7$$
$$S(\texttt{size}) = Y \qquad C(\texttt{size}) = 48$$

Concretization (Example)

$$true$$

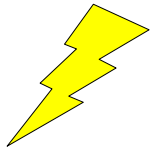
$$\begin{aligned} C(X) &= 5 \\ S(\texttt{m}) = X + 2 \quad & C(\texttt{m}) = 7 \\ S(\texttt{size}) = Y \quad & C(\texttt{size}) = 256 \end{aligned}$$

$$49 > Y$$

$$\begin{aligned} C(X) &= 5 \\ S(\texttt{m}) = X + 2 \quad & C(\texttt{m}) = 7 \\ S(\texttt{size}) = Y \quad & C(\texttt{size}) = 48 \end{aligned}$$

```
if (m*m > size) {
  …
  if (m < 5) {
    …
```

$$X + 2 < 5$$



Solution diverges from expected path! (e.g., X = 2)

Concretization (Example)

$$true$$

$$C(X) = 5$$

$$S(\texttt{m}) = X + 2 \qquad C(\texttt{m}) = 7$$

$$S(\texttt{size}) = Y \qquad C(\texttt{size}) = 256$$

```
if (m*m > size) {
  …
  if (m < 5) {
    …
```

Concretization constraint

$$49 > Y \wedge X = 5$$

$$C(X) = 5$$

$$S(\texttt{m}) = X + 2 \qquad C(\texttt{m}) = 7$$

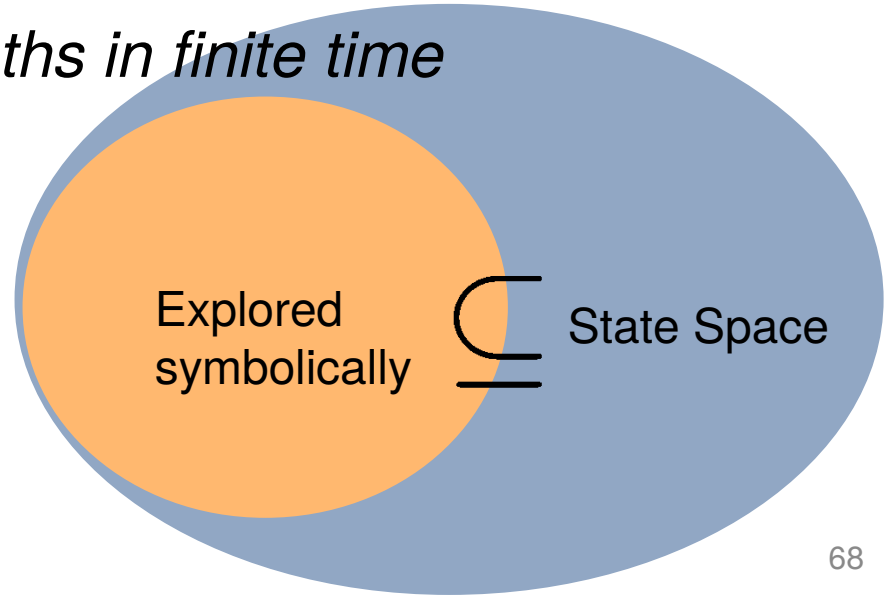
$$S(\texttt{size}) = Y \qquad C(\texttt{size}) = 48$$

Soundness & Completeness

- Conceptually, each path is exact
  - *Strongest postcondition in predicate transformer semantics*
  - *No over-approximation, no under-approximation*
- Globally, SE under-approximates
  - *Explores only subset of paths in finite time*
  - *“Eventual” completeness*

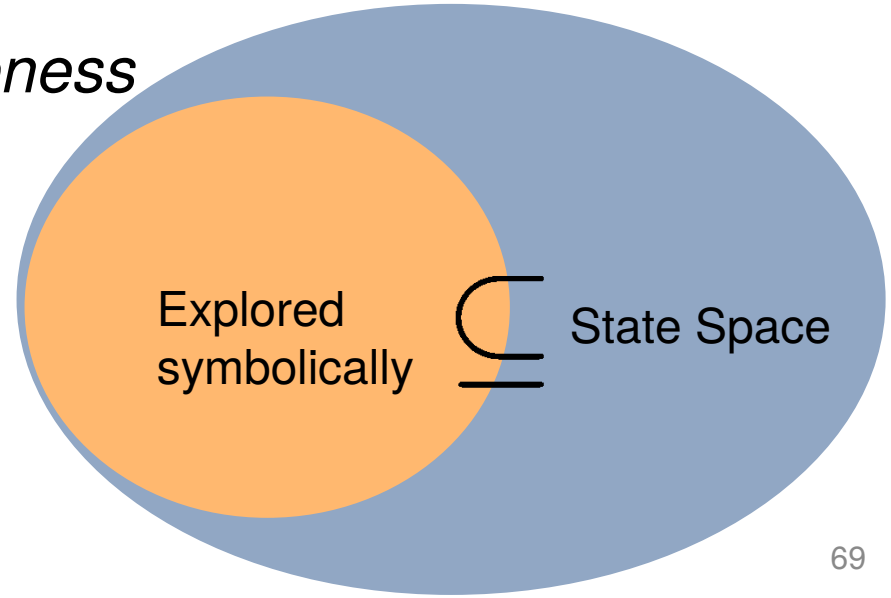
Soundness & Completeness

- Conceptually, each path is exact
  - *Strongest postcondition in predicate transformer semantics*
  - *No over-approximation, no under-approximation*
- Globally, SE under-approximates
  - *Explores only subset of paths in finite time*
  - *"Eventual" completeness*



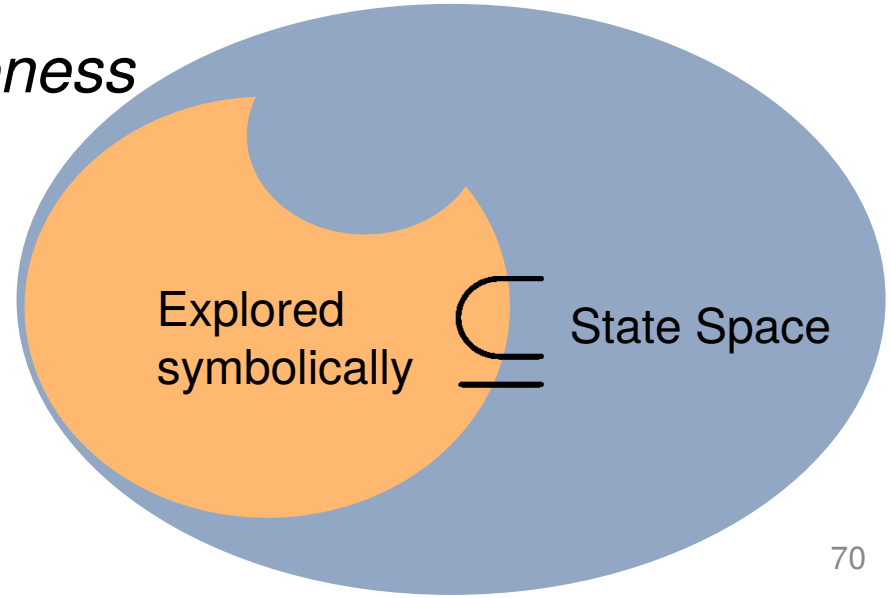
Soundness & Completeness

- Symbolic Execution = Underapproximates
  - *Soundness = does not include infeasible behavior*
  - *Completeness = explores all behavior*
- Concretization restricts state covered by path
  - *Remains sound*
  - *Loses (eventual) completeness*



Soundness & Completeness

- Symbolic Execution = Underapproximates
  - *Soundness = does not include infeasible behavior*
  - *Completeness = explores all behavior*
- Concretization restricts state covered by path
  - *Remains sound*
  - *Loses (eventual) completeness*



Concretization

- Key strength of dynamic symbolic execution
- Enables external calls
  - *Concretize call arguments*
  - *Callee executes concretely*
- Concretization constraints can be omitted
  - *Sacrifices soundness (original DART)*
  - *Deal with divergences by random restarts*