Lecture 10 – Symbolic Execution

SWS128: Software Analysis

Jihyeok Park



2026 Fall

Recall

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

Interval analysis at `return r`: $\texttt{r} \in [0, 2]$

Recall

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

Interval analysis at `return r`: $\texttt{r} \in [0, 2]$

- Can `f` return 2?
  Maybe.

Recall

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

Interval analysis at `return r`: $\texttt{r} \in [0, 2]$

- Can `f` return 2?
  Maybe.
- Which input makes it return 1?
  It cannot say.

Recall

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

Interval analysis at `return r`: $\texttt{r} \in [0, 2]$

- Can `f` return 2?
  Maybe.
- Which input makes it return 1?
  It cannot say.

- At every join, it **merged** the paths and forgot that `z` is `x + y`.

Recall

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

Interval analysis at `return r`: $\texttt{r} \in [0, 2]$

- Can `f` return 2?
  Maybe.
- Which input makes it return 1?
  It cannot say.

- At every join, it **merged** the paths and forgot that `z` is `x + y`.
- Today, **symbolic execution**:
  - run `f` with **unknown** inputs, one path at a time,
  - collect the conditions each path needs, and
  - ask a **solver** for an input that meets them all.

Recall

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

Interval analysis at `return r`: $\texttt{r} \in [0, 2]$

- Can `f` return 2?
  Maybe.
- Which input makes it return 1?
  It cannot say.

- At every join, it **merged** the paths and forgot that `z` is `x + y`.
- Today, **symbolic execution**:
  - run `f` with **unknown** inputs, one path at a time,
  - collect the conditions each path needs, and
  - ask a **solver** for an input that meets them all.

`f` returns	0	1	2
an input $(\texttt{x}, \texttt{y})$	$(10, 0)$	$(2, 9)$	**none**: no input can

Contents

1. Symbolic Execution
   - Symbols Instead of Values
   - The Execution Tree

2. Solving
   - SMT Solvers
   - In Python

3. Path Explosion
   - Branches and Loops
   - Coping

4. Limits

Contents

1. Symbolic Execution
   - Symbols Instead of Values
   - The Execution Tree

2. Solving
   - SMT Solvers
   - In Python

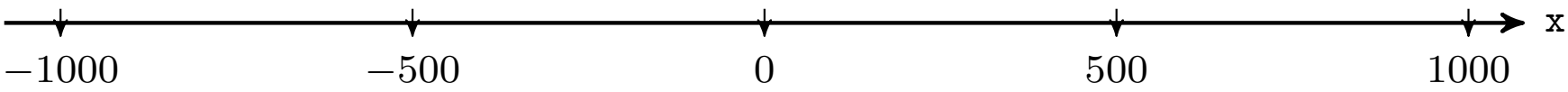
3. Path Explosion
   - Branches and Loops
   - Coping

4. Limits

Guessing an Input

```
def magic(x):
    r = 0
    if x * 7 + 3 == 5001:
        r = 1
    return r
```

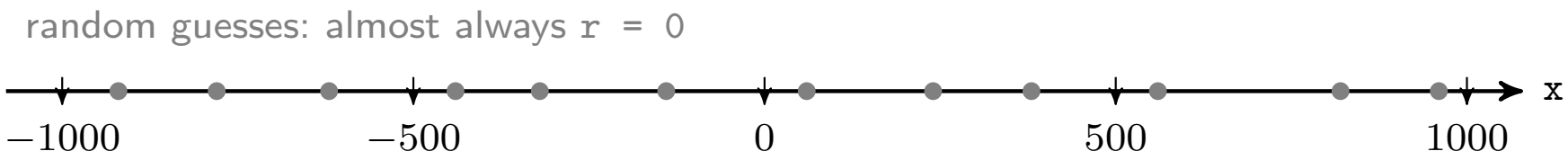
Which `x` reaches `r = 1`?



Guessing an Input

```
def magic(x):
    r = 0
    if x * 7 + 3 == 5001:
        r = 1
    return r
```

Which `x` reaches `r = 1`?

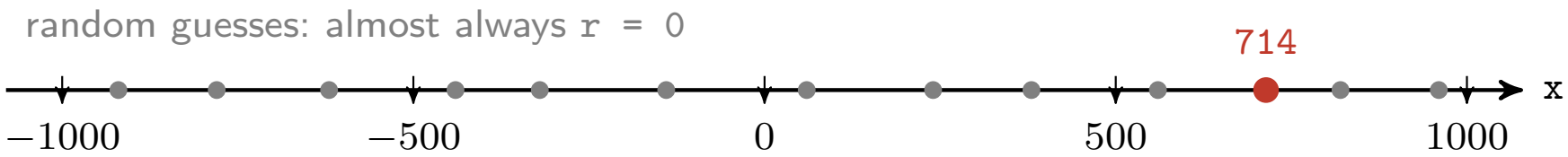


- **Guess**: exactly one `x` works. Random inputs almost never hit it.

Guessing an Input

```
def magic(x):
    r = 0
    if x * 7 + 3 == 5001:
        r = 1
    return r
```

Which `x` reaches `r = 1`?



- **Guess**: exactly one `x` works. Random inputs almost never hit it.
- **Solve**: the branch says $\alpha \times 7 + 3 = 5001$ for an unknown α. So $\alpha = 714$, at once.

Running with Symbols

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

edge	**concrete** $(2, 9)$	**symbolic** (α, β)
`entry`	$\mathtt{x} = 2$, $\mathtt{y} = 9$	$\mathtt{x} \mapsto \alpha$, $\mathtt{y} \mapsto \beta$

- Each input is a **symbol**: α for $\mathtt{x}$, β for $\mathtt{y}$. A variable holds an **expression** over them.

Running with Symbols

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

edge	concrete $(2, 9)$	symbolic (α, β)
`entry`	$\mathtt{x} = 2, \mathtt{y} = 9$	$\mathtt{x} \mapsto \alpha, \mathtt{y} \mapsto \beta$
`r = 0`	$\mathtt{r} = 0$	$\mathtt{r} \mapsto 0$

- Each input is a **symbol**: α for $\mathtt{x}$, β for $\mathtt{y}$. A variable holds an **expression** over them.

Running with Symbols

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

edge	concrete $(2, 9)$	symbolic (α, β)
`entry`	$\mathtt{x} = 2,\ \mathtt{y} = 9$	$\mathtt{x} \mapsto \alpha,\ \mathtt{y} \mapsto \beta$
`r = 0`	$\mathtt{r} = 0$	$\mathtt{r} \mapsto 0$
`z = x + y`	$\mathtt{z} = 11$	$\mathtt{z} \mapsto \alpha + \beta$

- Each input is a **symbol**: α for `x`, β for `y`. A variable holds an **expression** over them.

Running with Symbols

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

edge	concrete $(2, 9)$	symbolic (α, β)
`entry`	$\mathtt{x} = 2$, $\mathtt{y} = 9$	$\mathtt{x} \mapsto \alpha$, $\mathtt{y} \mapsto \beta$
`r = 0`	$\mathtt{r} = 0$	$\mathtt{r} \mapsto 0$
`z = x + y`	$\mathtt{z} = 11$	$\mathtt{z} \mapsto \alpha + \beta$
`10 < z`	`True`: go in	$\Phi = 10 < \alpha + \beta$

- Each input is a **symbol**: α for $\mathtt{x}$, β for $\mathtt{y}$. A variable holds an **expression** over them.
- A branch cannot be decided. Take it anyway, and remember its condition in the **path condition** Φ.

Running with Symbols

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

edge	concrete $(2, 9)$	symbolic (α, β)
`entry`	$\mathtt{x} = 2$, $\mathtt{y} = 9$	$\mathtt{x} \mapsto \alpha$, $\mathtt{y} \mapsto \beta$
`r = 0`	$\mathtt{r} = 0$	$\mathtt{r} \mapsto 0$
`z = x + y`	$\mathtt{z} = 11$	$\mathtt{z} \mapsto \alpha + \beta$
`10 < z`	`True`: go in	$\Phi = 10 < \alpha + \beta$
`x < 3`	`True`: go in	$\Phi = 10 < \alpha + \beta \wedge \alpha < 3$

- Each input is a **symbol**: α for $\mathtt{x}$, β for $\mathtt{y}$. A variable holds an **expression** over them.
- A branch cannot be decided. Take it anyway, and remember its condition in the **path condition** Φ.

Running with Symbols

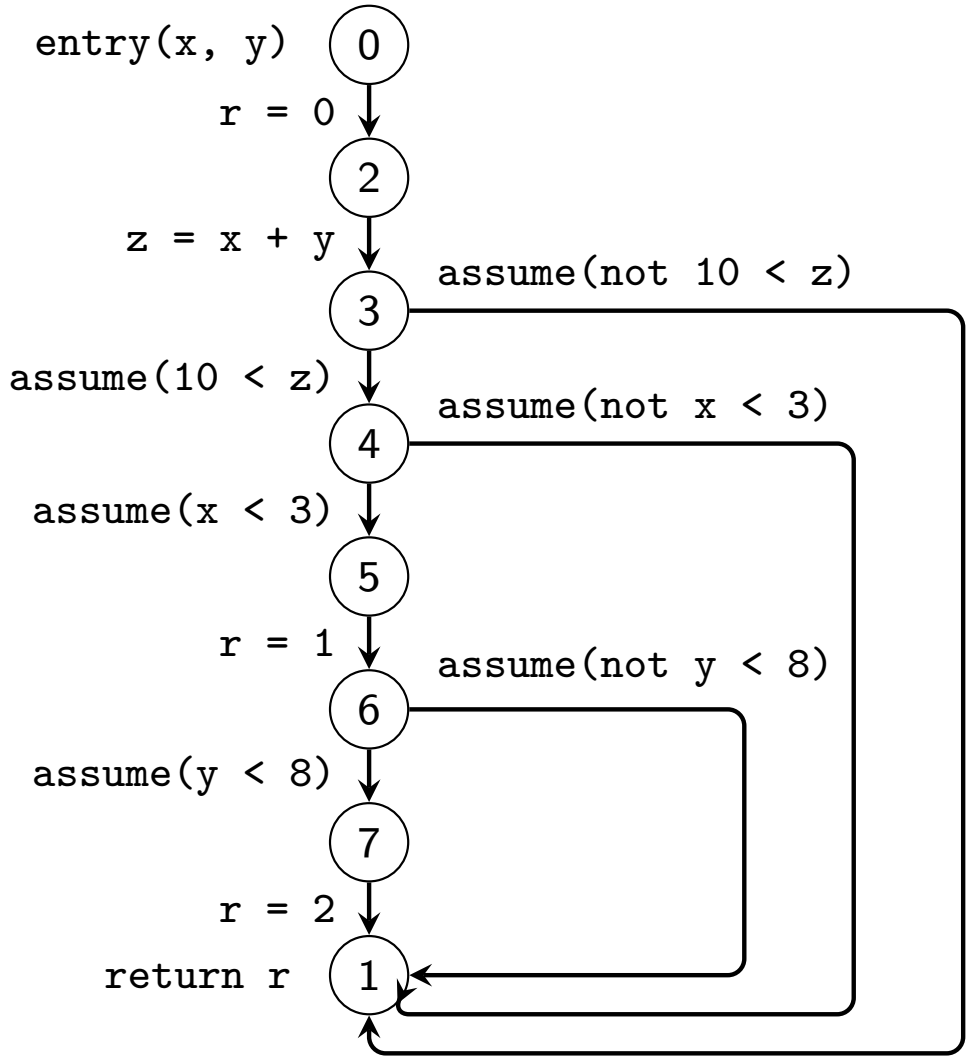
```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

edge	concrete $(2, 9)$	symbolic (α, β)
`entry`	$\mathtt{x} = 2, \mathtt{y} = 9$	$\mathtt{x} \mapsto \alpha, \mathtt{y} \mapsto \beta$
`r = 0`	$\mathtt{r} = 0$	$\mathtt{r} \mapsto 0$
`z = x + y`	$\mathtt{z} = 11$	$\mathtt{z} \mapsto \alpha + \beta$
`10 < z`	`True`: go in	$\Phi = 10 < \alpha + \beta$
`x < 3`	`True`: go in	$\Phi = 10 < \alpha + \beta \wedge \alpha < 3$
`r = 1`	$\mathtt{r} = 1$	$\mathtt{r} \mapsto 1$

- Each input is a **symbol**: α for $\mathtt{x}$, β for $\mathtt{y}$. A variable holds an **expression** over them.
- A branch cannot be decided. Take it anyway, and remember its condition in the **path condition** Φ.

Its CFG

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```



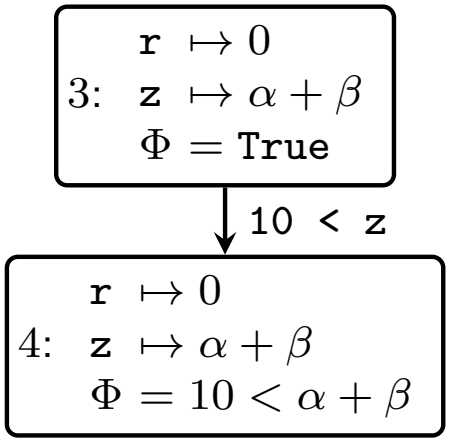
Three `if`s: points 3, 4, and 6 have two assume edges out.

The Execution Tree

$$
\begin{array}{ll}
 & \mathtt{r} \mapsto 0 \\
3: & \mathtt{z} \mapsto \alpha + \beta \\
 & \Phi = \mathtt{True}
\end{array}
$$

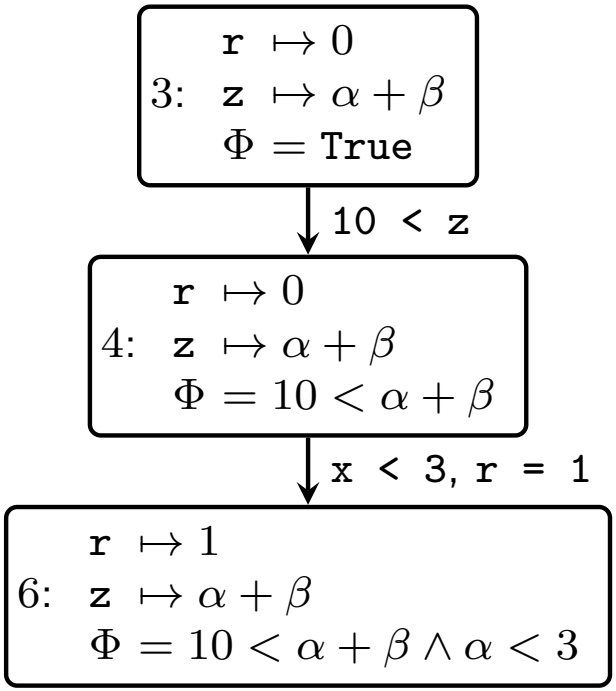
Start at point 3, after the two assignments. Nothing is known yet: $\Phi = \mathtt{True}$.

The Execution Tree



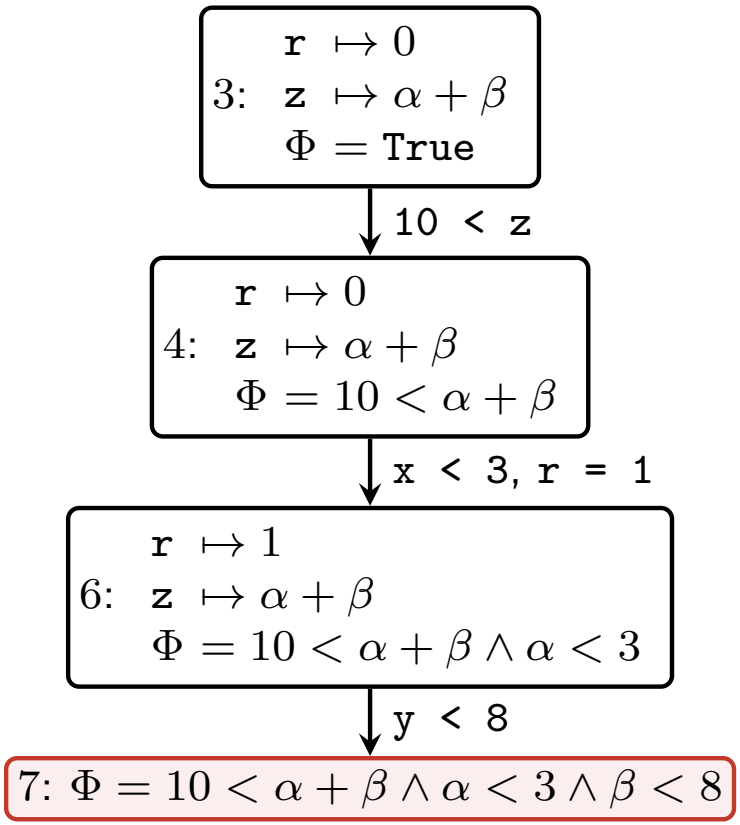
Point 3 forks. Take `10 < z` first and add it to Φ. Satisfiable? Yes: go on.

The Execution Tree



Point 4 forks. Take `x < 3`, then `r = 1`: now `r` $\mapsto 1$.

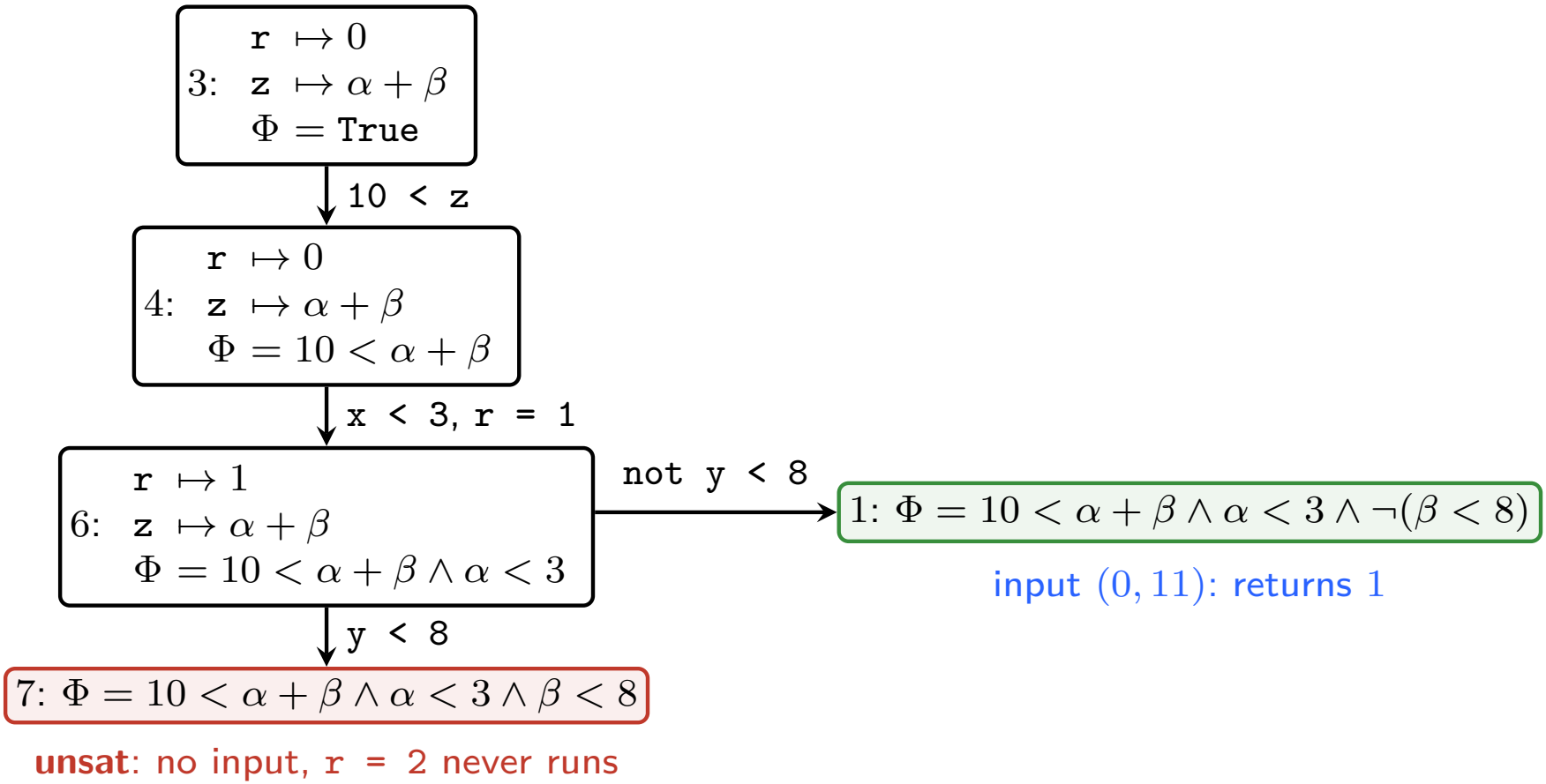
The Execution Tree



unsat: no input, `r = 2` never runs

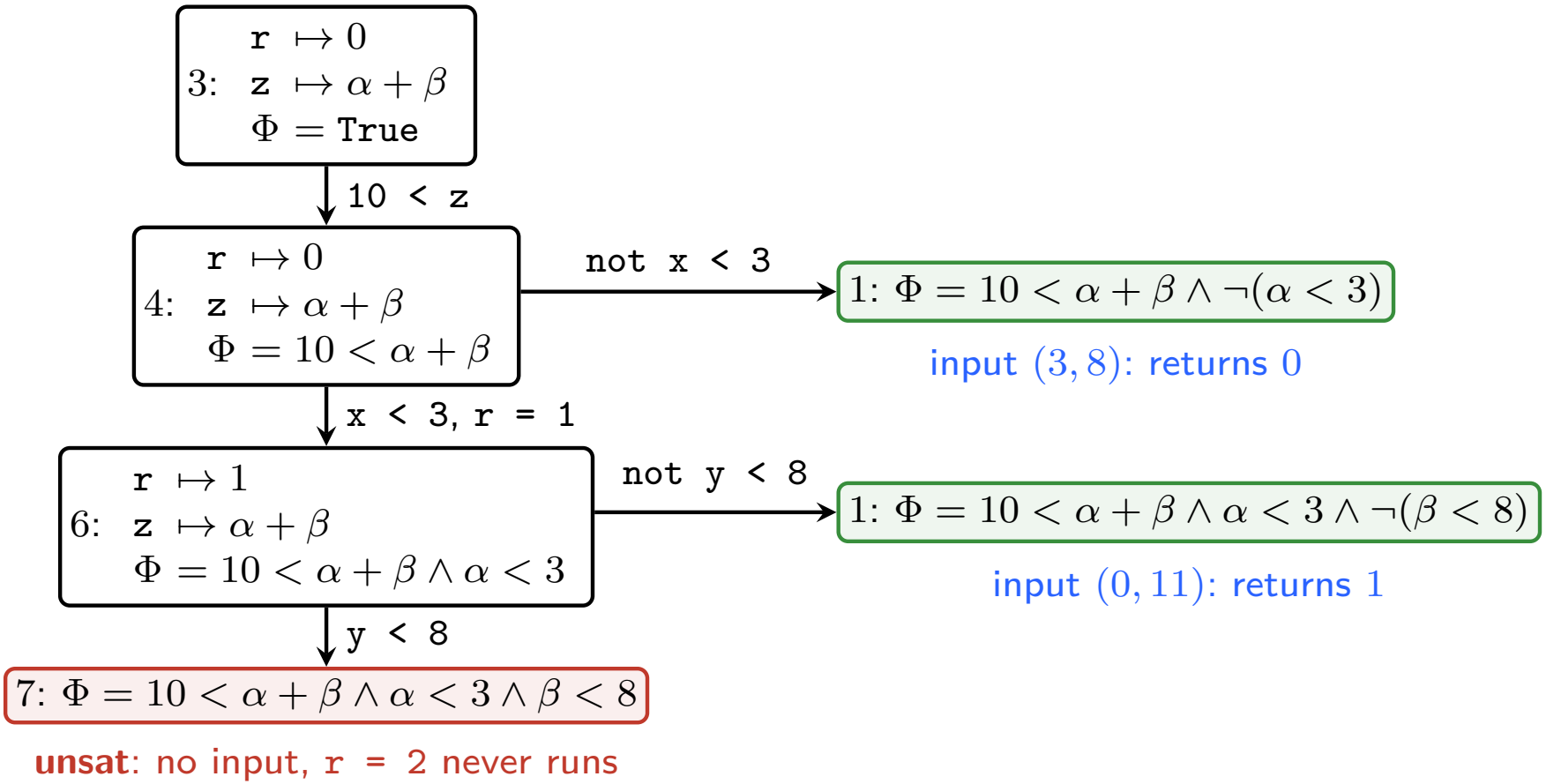
Point 6 forks. `y < 8` makes Φ **unsatisfiable**: $\alpha < 3$ and $10 < \alpha + \beta$ force $\beta \geq 9$. Drop it.

The Execution Tree



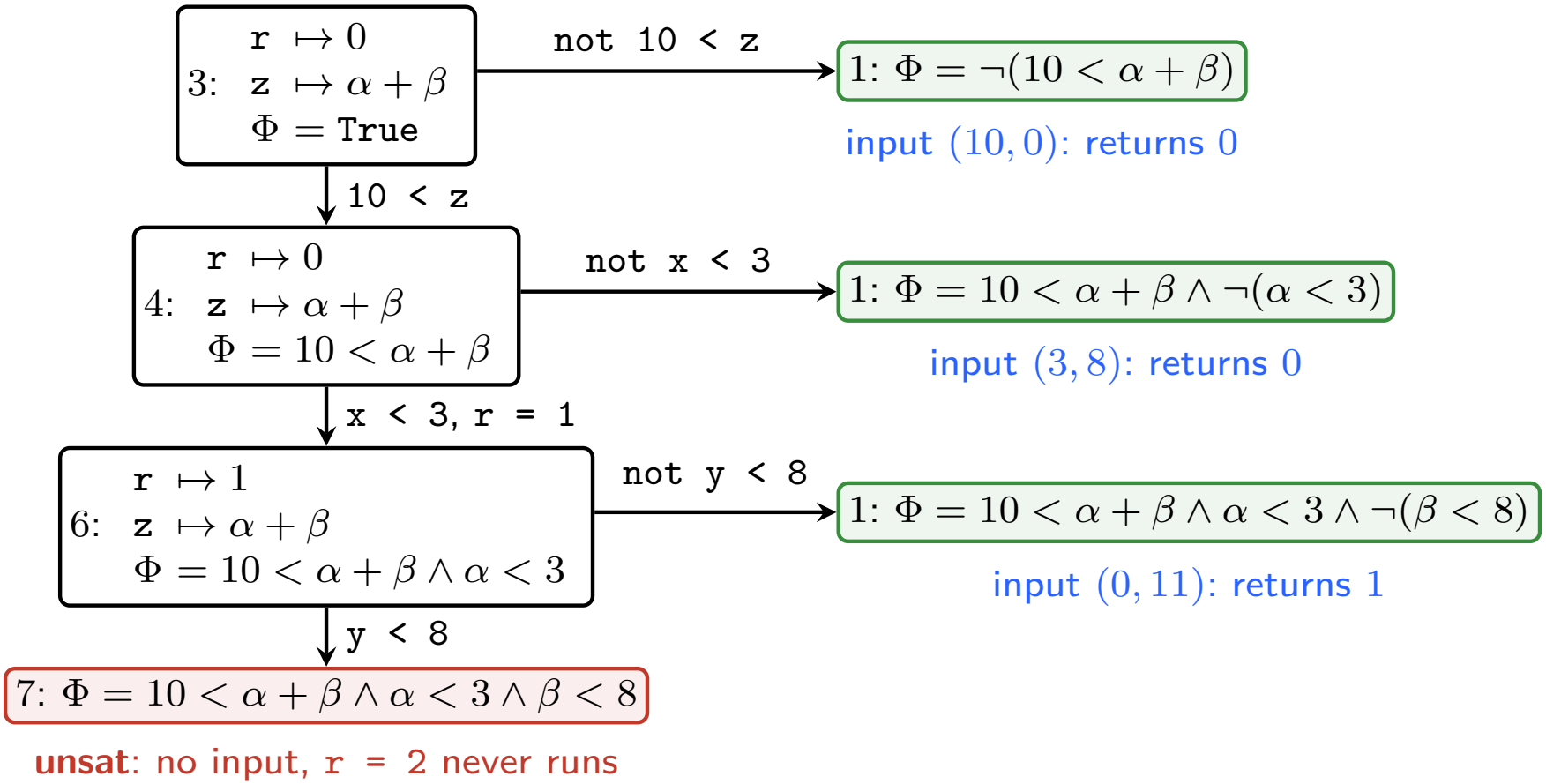
The other side, `not y < 8`, reaches the exit. The solver turns its Φ into an **input**.

The Execution Tree



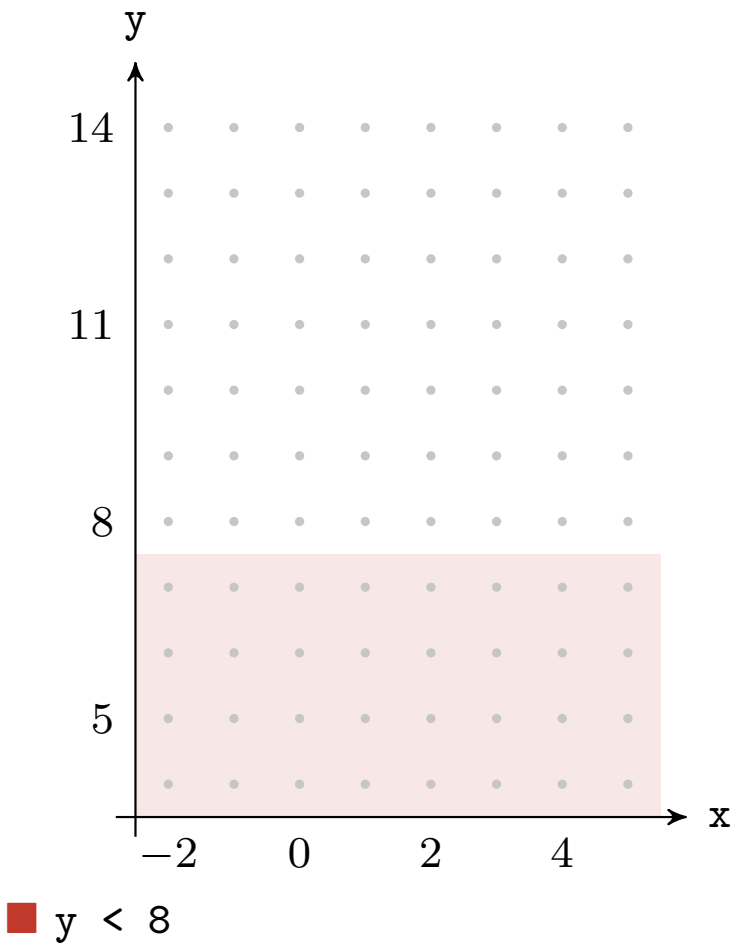
Back to point 4: `not x < 3` reaches the exit with $\texttt{r} \mapsto 0$.

The Execution Tree

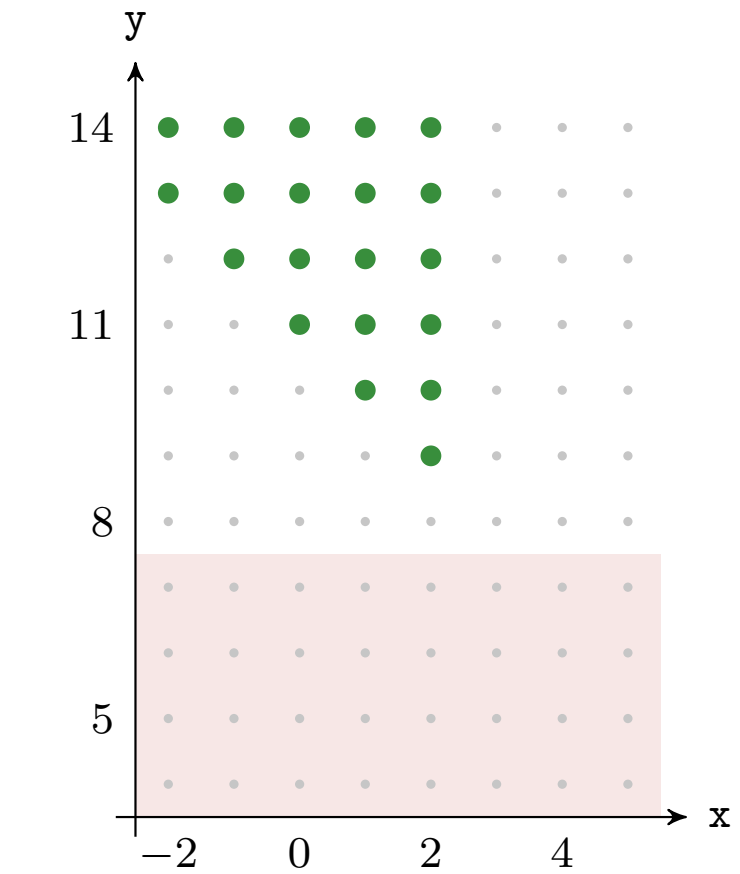


Back to point 3. Four paths, one infeasible: **three inputs**, one per path, and a **proof** that `r = 2` never runs.

No Input Reaches `r = 2`



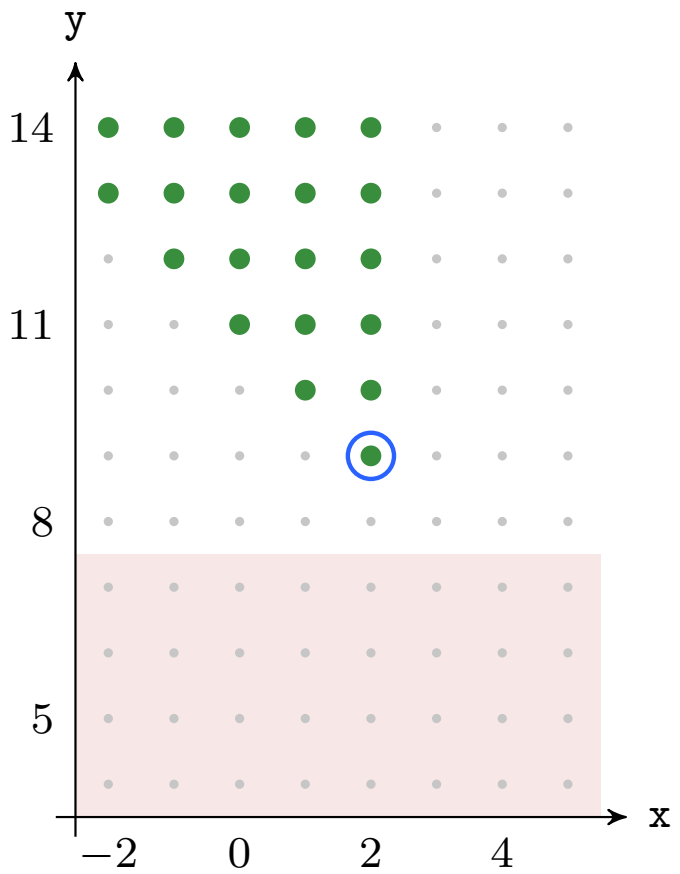
No Input Reaches `r = 2`



- `y < 8`
- `x < 3 and 10 < x + y`: reach point 6

- Inputs that reach point 6: `x` at most 2, so `y` at least 9.

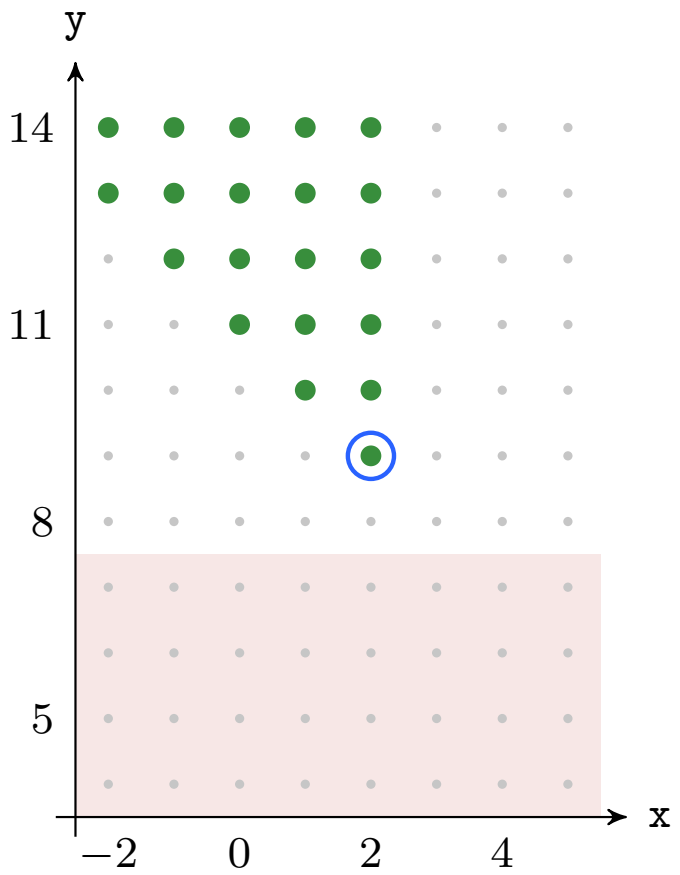
No Input Reaches `r = 2`



- `y < 8`
- `x < 3 and 10 < x + y`: reach point 6
- $(2, 9)$: the solver's model

- Inputs that reach point 6: `x` at most 2, so `y` at least 9.
- The solver picks one of them: $(2, 9)$ reaches `r = 1`.

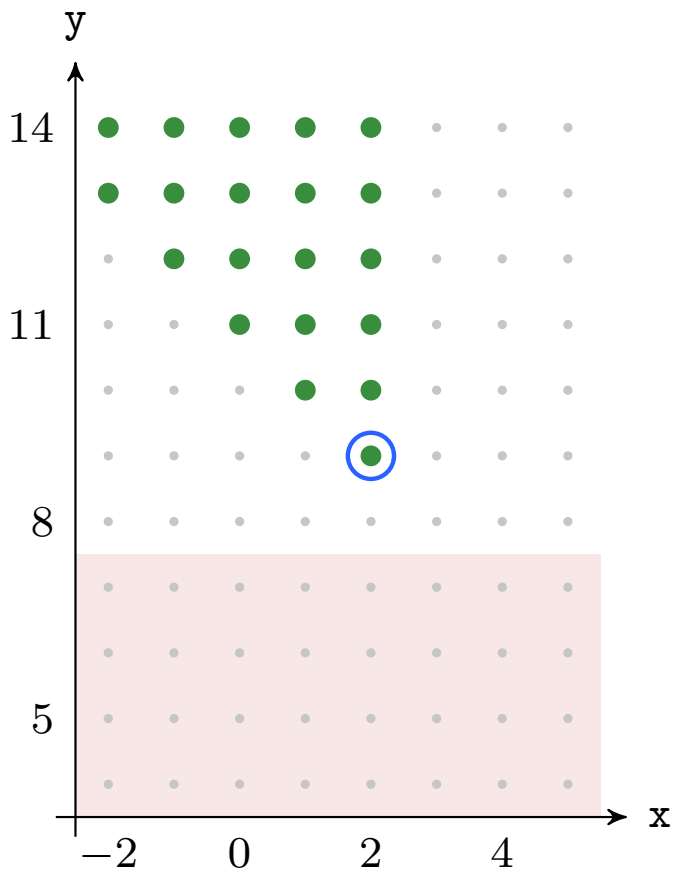
No Input Reaches `r = 2`



- `y < 8`
- `x < 3 and 10 < x + y`: reach point 6
- $(2, 9)$: the solver's model

- Inputs that reach point 6: `x` at most 2, so `y` at least 9.
- The solver picks one of them: $(2, 9)$ reaches `r = 1`.
- **None** lies in the red band. `r = 2` never runs, and `f` never returns 2.

No Input Reaches `r = 2`



- `y < 8`
- `x < 3 and 10 < x + y`: reach point 6
- $(2, 9)$: the solver's model

- Inputs that reach point 6: `x` at most 2, so `y` at least 9.
- The solver picks one of them: $(2, 9)$ reaches `r = 1`.
- **None** lies in the red band. `r = 2` never runs, and `f` never returns 2.
- Interval analysis kept $\texttt{x} \leq 2$ but lost `z = x + y`, so it said "maybe".

Contents

1. Symbolic Execution
   - Symbols Instead of Values
   - The Execution Tree

2. Solving
   - SMT Solvers
   - In Python

3. Path Explosion
   - Branches and Loops
   - Coping

4. Limits

An SMT Solver

- An **SMT solver** takes a formula over integers, booleans, . . . and answers **sat** with a **model**, or **unsat**.

An SMT Solver

PLRG

- An **SMT solver** takes a formula over integers, booleans, . . . and answers **sat** with a **model**, or **unsat**.

```
>>> import z3
>>> x, y = z3.Int("x"), z3.Int("y")
>>> s = z3.Solver()
>>> s.add(10 < x + y, x < 3)    Φ at point 6
>>> s.check(), s.model()
(sat, [x = 2, y = 9])
>>> s.add(y < 8)    and y < 8: point 7
>>> s.check()
unsat
```

An SMT Solver

PLRG

- An **SMT solver** takes a formula over integers, booleans, . . . and answers **sat** with a **model**, or **unsat**.

```
>>> import z3
>>> x, y = z3.Int("x"), z3.Int("y")
>>> s = z3.Solver()
>>> s.add(10 < x + y, x < 3)    Φ at point 6
>>> s.check(), s.model()
(sat, [x = 2, y = 9])
>>> s.add(y < 8)    and y < 8: point 7
>>> s.check()
unsat
```

- **Z3**: `pip install z3-solver`. The only package this course needs.
- `z3.Int("x")` is the symbol α. Integers do not wrap, like Python's.

A Symbolic State

```
@dataclass(frozen=True)
class SymState:
    node: NodeId
    env: tuple                          # ((name, z3 expr), ...) -- hashable
    pc: tuple                           # conjuncts of the path condition
    ...                                 # get, set, at, assume

def start(g: CFG) -> SymState:
    """One fresh symbol per parameter: the input is unknown."""
    return SymState(g.entry, tuple((p, z3.Int(p)) for p in g.params), ())
```

the state at point 6 of the execution tree	
node	6
env	$\mathtt{x} \mapsto \alpha,\ \ \mathtt{y} \mapsto \beta,\ \ \mathtt{r} \mapsto 1,\ \ \mathtt{z} \mapsto \alpha + \beta$
pc	$(10 < \alpha + \beta,\ \ \alpha < 3)$: their conjunction is Φ

A Symbolic State

```python
@dataclass(frozen=True)
class SymState:
    node: NodeId
    env: tuple                          # ((name, z3 expr), ...) -- hashable
    pc: tuple                           # conjuncts of the path condition
    ...                                 # get, set, at, assume

def start(g: CFG) -> SymState:
    """One fresh symbol per parameter: the input is unknown."""
    return SymState(g.entry, tuple((p, z3.Int(p)) for p in g.params), ())
```

the state at point 6 of the execution tree	
node	6
env	$\mathtt{x} \mapsto \alpha,\ \ \mathtt{y} \mapsto \beta,\ \ \mathtt{r} \mapsto 1,\ \ \mathtt{z} \mapsto \alpha + \beta$
pc	$(10 < \alpha + \beta,\ \ \alpha < 3)$: their conjunction is Φ

A Symbolic State

```
@dataclass(frozen=True)
class SymState:
    node: NodeId
    env: tuple                          # ((name, z3 expr), ...) -- hashable
    pc: tuple                           # conjuncts of the path condition
    ...                                 # get, set, at, assume

def start(g: CFG) -> SymState:
    """One fresh symbol per parameter: the input is unknown."""
    return SymState(g.entry, tuple((p, z3.Int(p)) for p in g.params), ())
```

the state at point 6 of the execution tree	
node	6
env	$\mathtt{x} \mapsto \alpha,\ \ \mathtt{y} \mapsto \beta,\ \ \mathtt{r} \mapsto 1,\ \ \mathtt{z} \mapsto \alpha + \beta$
pc	$(10 < \alpha + \beta,\ \ \alpha < 3)$: their conjunction is Φ

A Symbolic State

```python
@dataclass(frozen=True)
class SymState:
    node: NodeId
    env: tuple                          # ((name, z3 expr), ...) -- hashable
    pc: tuple                           # conjuncts of the path condition
    ...                                 # get, set, at, assume

def start(g: CFG) -> SymState:
    """One fresh symbol per parameter: the input is unknown."""
    return SymState(g.entry, tuple((p, z3.Int(p)) for p in g.params), ())
```

the state at point 6 of the execution tree	
node	6
env	$\mathtt{x} \mapsto \alpha,\ \ \mathtt{y} \mapsto \beta,\ \ \mathtt{r} \mapsto 1,\ \ \mathtt{z} \mapsto \alpha + \beta$
pc	$(10 < \alpha + \beta,\ \ \alpha < 3)$: their conjunction is Φ

- `start`: every parameter gets its own symbol, and Φ is empty.

From Expr to Z3

```
def to_z3(e: Expr, s: SymState):
    ...
```

The interpreter's `eval_expr`, but it builds a **term** instead of a value. With `s.env` $= (\texttt{x} \mapsto \alpha,\ \texttt{y} \mapsto \beta,\ \texttt{z} \mapsto \alpha + \beta)$:

`e`	`to_z3(e, s)`	**why**
`42`	42	a literal: a constant
`x`	α	a variable: the term in `s`

From Expr to Z3

```
def to_z3(e: Expr, s: SymState):
    ...
```

The interpreter's `eval_expr`, but it builds a **term** instead of a value. With `s.env` $= (\texttt{x} \mapsto \alpha,\ \texttt{y} \mapsto \beta,\ \texttt{z} \mapsto \alpha + \beta)$:

`e`	`to_z3(e, s)`	**why**
`42`	42	a literal: a constant
`x`	α	a variable: the term in `s`
`x + 1`	$\alpha + 1$	an operator: on the terms
`z * 2`	$(\alpha + \beta) \times 2$	

From Expr to Z3

```
def to_z3(e: Expr, s: SymState):
    ...
```

The interpreter's `eval_expr`, but it builds a **term** instead of a value. With `s.env` $= (\texttt{x} \mapsto \alpha,\ \texttt{y} \mapsto \beta,\ \texttt{z} \mapsto \alpha + \beta)$:

`e`	`to_z3(e, s)`	**why**
`42`	42	a literal: a constant
`x`	α	a variable: the term in `s`
`x + 1`	$\alpha + 1$	an operator: on the terms
`z * 2`	$(\alpha + \beta) \times 2$	
`not x < 3`	$\neg(\alpha < 3)$	a condition: a boolean term
`x < 3 and not y < 8`	$\alpha < 3 \wedge \neg(\beta < 8)$	

From Expr to Z3

```
def to_z3(e: Expr, s: SymState):
    ...
```

The interpreter's `eval_expr`, but it builds a **term** instead of a value. With `s.env` $= (\texttt{x} \mapsto \alpha,\ \texttt{y} \mapsto \beta,\ \texttt{z} \mapsto \alpha + \beta)$:

`e`	`to_z3(e, s)`	**why**
`42`	42	a literal: a constant
`x`	α	a variable: the term in `s`
`x + 1`	$\alpha + 1$	an operator: on the terms
`z * 2`	$(\alpha + \beta) \times 2$	
`not x < 3`	$\neg(\alpha < 3)$	a condition: a boolean term
`x < 3 and not y < 8`	$\alpha < 3 \wedge \neg(\beta < 8)$	
`x + "abc"`	`NotSupported`	a string: only ints and bools
`w`	`NameError`	`w` is not defined: a bug

Same Expression, Different States

```
def g(x, y):
    x = x + y
    y = x * 2
    x = 7
    return x
```

after	x ↦	y ↦	to_z3(x + y, s)
entry	α	β	$\alpha + \beta$

- One expression, `x + y`, and four different terms: `to_z3` reads whatever `s.env` holds **now**.

Same Expression, Different States

```
def g(x, y):
    x = x + y
    y = x * 2
    x = 7
    return x
```

after	`x` $\mapsto$	`y` $\mapsto$	`to_z3(x + y, s)`
`entry`	α	β	$\alpha + \beta$
`x = x + y`	$\alpha + \beta$	β	$(\alpha + \beta) + \beta$

- One expression, `x + y`, and four different terms: `to_z3` reads whatever `s.env` holds **now**.

Same Expression, Different States

```
def g(x, y):
    x = x + y
    y = x * 2
    x = 7
    return x
```

after	x ↦	y ↦	to_z3(x + y, s)
entry	α	β	$\alpha + \beta$
x = x + y	$\alpha + \beta$	β	$(\alpha + \beta) + \beta$
y = x * 2	$\alpha + \beta$	$(\alpha + \beta) \times 2$	$(\alpha + \beta) + (\alpha + \beta) \times 2$

- One expression, `x + y`, and four different terms: `to_z3` reads whatever `s.env` holds **now**.
- `y = x * 2`: `y` no longer holds β. Its term is built from `x`'s.

Same Expression, Different States

```
def g(x, y):
    x = x + y
    y = x * 2
    x = 7
    return x
```

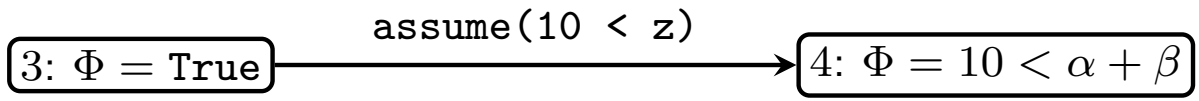
after	x ↦	y ↦	to_z3(x + y, s)
entry	α	β	$\alpha + \beta$
x = x + y	$\alpha + \beta$	β	$(\alpha + \beta) + \beta$
y = x * 2	$\alpha + \beta$	$(\alpha + \beta) \times 2$	$(\alpha + \beta) + (\alpha + \beta) \times 2$
x = 7	7	$(\alpha + \beta) \times 2$	$7 + (\alpha + \beta) \times 2$

- One expression, `x + y`, and four different terms: `to_z3` reads whatever `s.env` holds **now**.
- `y = x * 2`: `y` no longer holds β. Its term is built from `x`'s.
- `x = 7`: the symbol α is **gone** from `x`. No input can change `x` any more.

One Edge

```
def transfer(st: SymState, e: Edge) -> SymState | None:
    """The symbolic state after taking edge e, or None if that is infeasible."""
    ...
```

`assume(c)`, satisfiable:
add c to Φ

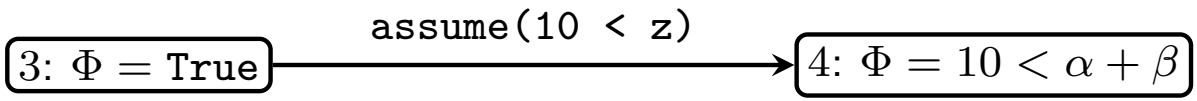


Every arrow of the execution tree is one call of `transfer`.

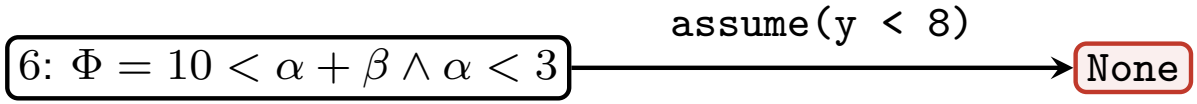
One Edge

```
def transfer(st: SymState, e: Edge) -> SymState | None:
    """The symbolic state after taking edge e, or None if that is infeasible."""
    ...
```

`assume(c)`, satisfiable:
add c to Φ



`assume(c)`, unsatisfiable:
`None`, no successor

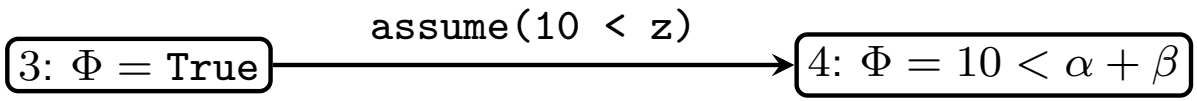


Every arrow of the execution tree is one call of `transfer`.

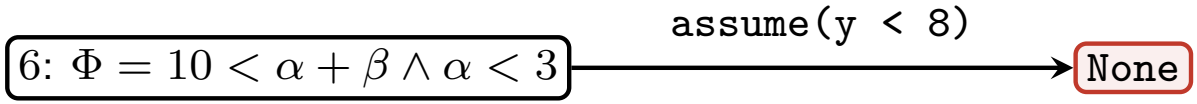
One Edge

```
def transfer(st: SymState, e: Edge) -> SymState | None:
    """The symbolic state after taking edge e, or None if that is infeasible."""
    ...
```

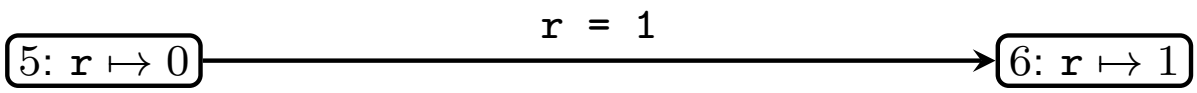
`assume(c)`, satisfiable:
add c to Φ



`assume(c)`, unsatisfiable:
`None`, no successor



`x = e`:
`x` holds the **term** for `e`



Every arrow of the execution tree is one call of `transfer`.

One Edge

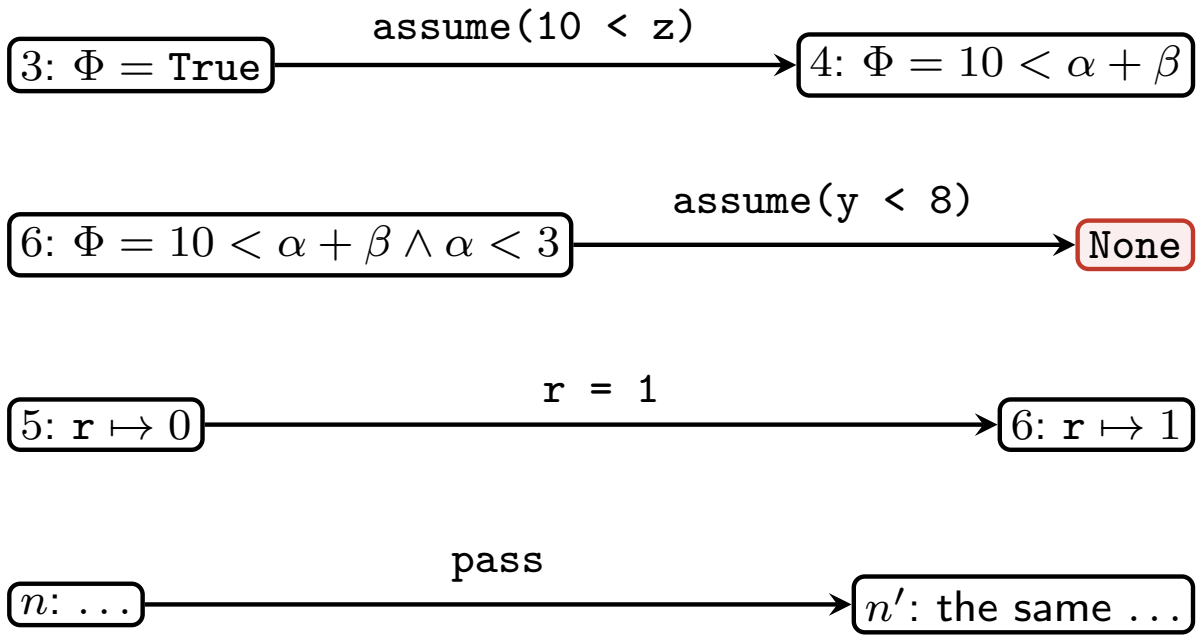
```
def transfer(st: SymState, e: Edge) -> SymState | None:
    """The symbolic state after taking edge e, or None if that is infeasible."""
    ...
```

`assume(c)`, satisfiable:
add c to Φ

`assume(c)`, unsatisfiable:
`None`, no successor

`x = e`:
`x` holds the **term** for `e`

`pass`:
only move to the next point



Every arrow of the execution tree is one call of `transfer`.

The Search

```
def walk(g: CFG, max_depth: int = 60):
    """Every feasible symbolic state, depth-first."""
    stack = [(start(g), 0)]
    while stack:
        st, d = stack.pop()
        yield st, d
        if d >= max_depth:
            continue
        for e in reversed(g.succ[st.node]):
            nxt = transfer(st, e)
            if nxt is not None:
                stack.append((nxt, d + 1))
```

The execution tree, built one state at a time.

The Search

```
def walk(g: CFG, max_depth: int = 60):
    """Every feasible symbolic state, depth-first."""
    stack = [(start(g), 0)]
    while stack:
        st, d = stack.pop()
        yield st, d
        if d >= max_depth:
            continue
        for e in reversed(g.succ[st.node]):
            nxt = transfer(st, e)
            if nxt is not None:
                stack.append((nxt, d + 1))
```

Start with one state: the entry, at depth 0.

The Search

```
def walk(g: CFG, max_depth: int = 60):
    """Every feasible symbolic state, depth-first."""
    stack = [(start(g), 0)]
    while stack:
        st, d = stack.pop()
        yield st, d
        if d >= max_depth:
            continue
        for e in reversed(g.succ[st.node]):
            nxt = transfer(st, e)
            if nxt is not None:
                stack.append((nxt, d + 1))
```

Pop the newest state: **depth-first**, as in the tree we drew. Hand it to the caller.

The Search

```
def walk(g: CFG, max_depth: int = 60):
    """Every feasible symbolic state, depth-first."""
    stack = [(start(g), 0)]
    while stack:
        st, d = stack.pop()
        yield st, d
        if d >= max_depth:
            continue
        for e in reversed(g.succ[st.node]):
            nxt = transfer(st, e)
            if nxt is not None:
                stack.append((nxt, d + 1))
```

Push every feasible successor: two at a fork. `transfer` has already dropped an unsat one.

The Search

```
def walk(g: CFG, max_depth: int = 60):
    """Every feasible symbolic state, depth-first."""
    stack = [(start(g), 0)]
    while stack:
        st, d = stack.pop()
        yield st, d
        if d >= max_depth:
            continue
        for e in reversed(g.succ[st.node]):
            nxt = transfer(st, e)
            if nxt is not None:
                stack.append((nxt, d + 1))
```

Stop a path after `max_depth` edges. Without it, a loop on a symbolic bound never ends.

Asking for an Input

```python
def reach_node(g: CFG, target: NodeId,
               max_depth: int = 60) -> dict[str, int] | None:
    """An input that reaches `target`."""
    ...

def returns(g: CFG, value: int, max_depth: int = 60) -> dict[str, int] | None:
    """An input that makes the program return `value`."""
    ...
```

An input is a `dict` from each parameter to an `int`: `{'x': 2, 'y': 9}` means `f(2, 9)`. On the running example:

call	input	why
`reach_node(g, 6)`	`{'x': 2, 'y': 9}`	the first state at point 6

`reach_node`: solve Φ of the first state at the target.

Asking for an Input

```python
def reach_node(g: CFG, target: NodeId,
               max_depth: int = 60) -> dict[str, int] | None:
    """An input that reaches `target`."""
    ...

def returns(g: CFG, value: int, max_depth: int = 60) -> dict[str, int] | None:
    """An input that makes the program return `value`."""
    ...
```

An input is a `dict` from each parameter to an `int`: `{'x': 2, 'y': 9}` means `f(2, 9)`. On the running example:

call	input	why
`reach_node(g, 6)`	`{'x': 2, 'y': 9}`	the first state at point 6
`returns(g, 1)`	`{'x': 0, 'y': 11}`	the only exit with $\mathtt{r} \mapsto 1$

`returns`: at each exit, solve Φ and "returned term $=$ `value`".

Asking for an Input

```python
def reach_node(g: CFG, target: NodeId,
               max_depth: int = 60) -> dict[str, int] | None:
    """An input that reaches `target`."""
    ...

def returns(g: CFG, value: int, max_depth: int = 60) -> dict[str, int] | None:
    """An input that makes the program return `value`."""
    ...
```

An input is a `dict` from each parameter to an `int`: `{'x': 2, 'y': 9}` means `f(2, 9)`. On the running example:

call	input	why
`reach_node(g, 6)`	`{'x': 2, 'y': 9}`	the first state at point 6
`returns(g, 1)`	`{'x': 0, 'y': 11}`	the only exit with `r` $\mapsto 1$
`returns(g, 0)`	`{'x': 3, 'y': 8}`	the first of two exits with `r` $\mapsto 0$

`returns`: at each exit, solve Φ and "returned term $=$ `value`".

Asking for an Input

```python
def reach_node(g: CFG, target: NodeId,
               max_depth: int = 60) -> dict[str, int] | None:
    """An input that reaches `target`."""
    ...

def returns(g: CFG, value: int, max_depth: int = 60) -> dict[str, int] | None:
    """An input that makes the program return `value`."""
    ...
```

An input is a `dict` from each parameter to an `int`: `{'x': 2, 'y': 9}` means `f(2, 9)`. On the running example:

call	input	why
`reach_node(g, 6)`	`{'x': 2, 'y': 9}`	the first state at point 6
`returns(g, 1)`	`{'x': 0, 'y': 11}`	the only exit with `r` $\mapsto 1$
`returns(g, 0)`	`{'x': 3, 'y': 8}`	the first of two exits with `r` $\mapsto 0$
`returns(g, 2)`	`None`	no exit with `r` $\mapsto 2$

`returns`: at each exit, solve Φ and "returned term $=$ `value`".

Running It

```
>>> from symex import reach
>>> src = """
... def f(x, y):
...     r = 0
...     z = x + y
...     if 10 < z:
...         if x < 3:
...             r = 1
...             if y < 8:
...                 r = 2
...     return r
... """
>>> reach(src, "r = 1")
{'x': 2, 'y': 9}
>>> print(reach(src, "r = 2"))
None
>>> reach(open("examples/magic.py").read(), "r = 1")
{'x': 714}
```

Running It

```
>>> from symex import reach
>>> src = """
... def f(x, y):
...     r = 0
...     z = x + y
...     if 10 < z:
...         if x < 3:
...             r = 1
...             if y < 8:
...                 r = 2
...     return r
... """
>>> reach(src, "r = 1")
{'x': 2, 'y': 9}
>>> print(reach(src, "r = 2"))
None
>>> reach(open("examples/magic.py").read(), "r = 1")
{'x': 714}
```

- Not a warning but an **input**. Run `f(2, 9)`: it does reach `r = 1`.
- `None` for `r = 2`: every path to it is unsatisfiable.

Contents

1. Symbolic Execution
   - Symbols Instead of Values
   - The Execution Tree

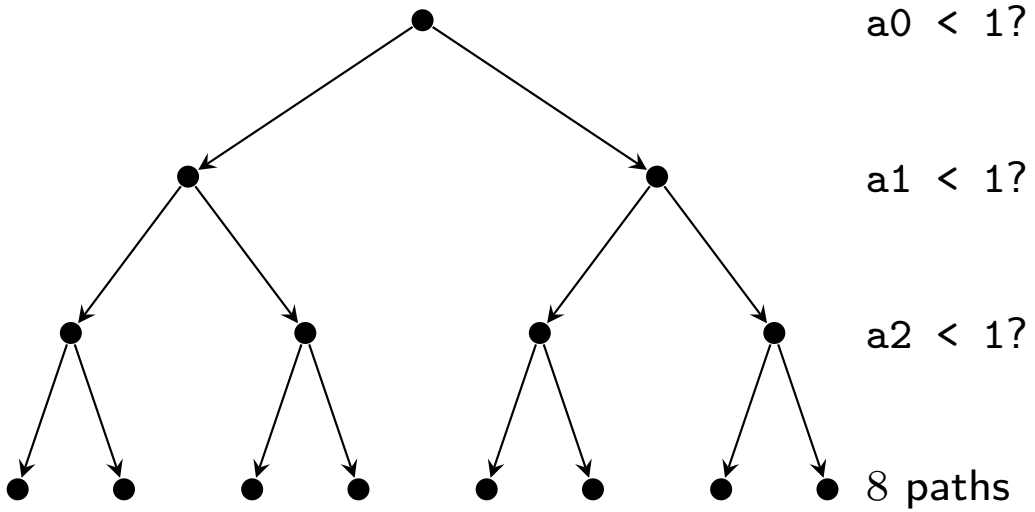
2. Solving
   - SMT Solvers
   - In Python

3. Path Explosion
   - Branches and Loops
   - Coping

4. Limits

Every Branch Doubles

```
def f(a0, a1, a2):
    if a0 < 1:
        a0 = 1
    if a1 < 1:
        a1 = 1
    if a2 < 1:
        a2 = 1
    return a0
```

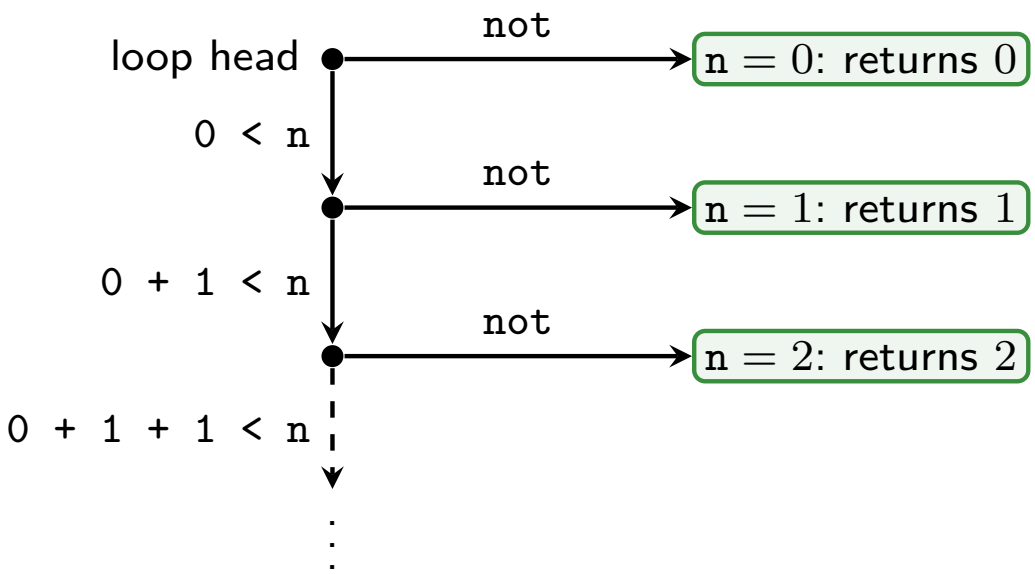


ifs in a row	1	2	3	4	$\cdots$	30
paths	2	4	8	16	$\cdots$	$\approx 10^9$

Every path is a separate state and a separate solver call: **path explosion**.

Loops

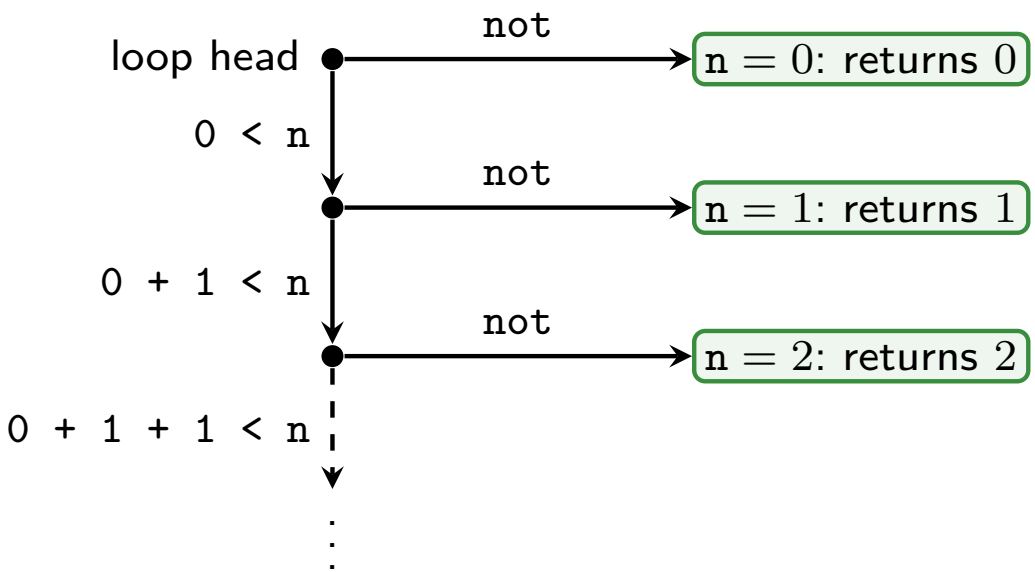
```
def loop(n):
    i = 0
    while i < n:
        i = i + 1
    return i
```



- A symbolic bound `n`: the loop head forks **every** time. **Infinitely many** paths, and `i` grows to `0 + 1 + 1`.

Loops

```
def loop(n):
    i = 0
    while i < n:
        i = i + 1
    return i
```

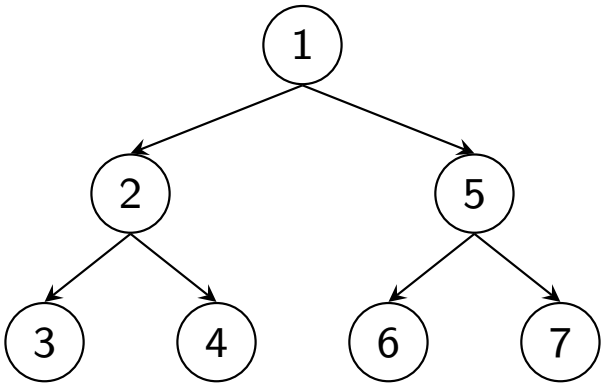


- A symbolic bound `n`: the loop head forks **every** time. **Infinitely many** paths, and `i` grows to `0 + 1 + 1`.
- So `walk` stops a path after `max_depth` edges:

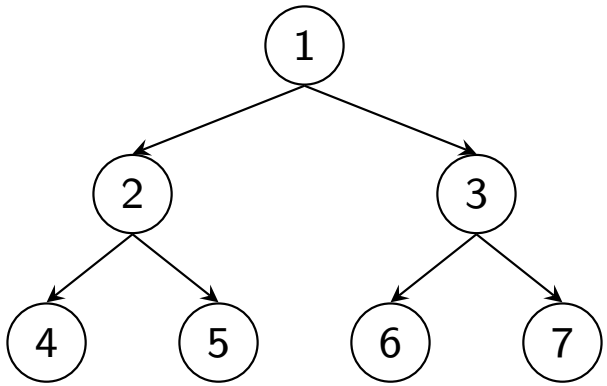
`max_depth`	5	10	20	30	···
`len(paths(g, max_depth))`	3	6	11	16	no end

Which Path First

DFS (`walk`)



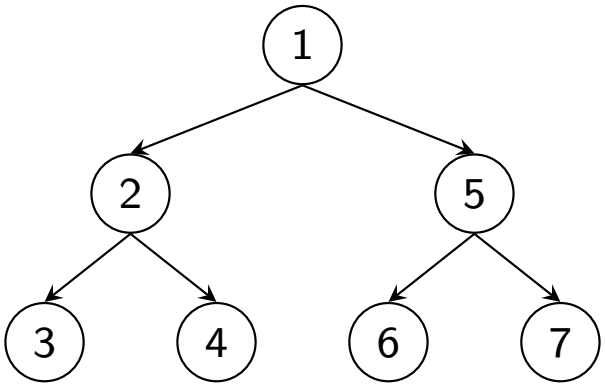
BFS



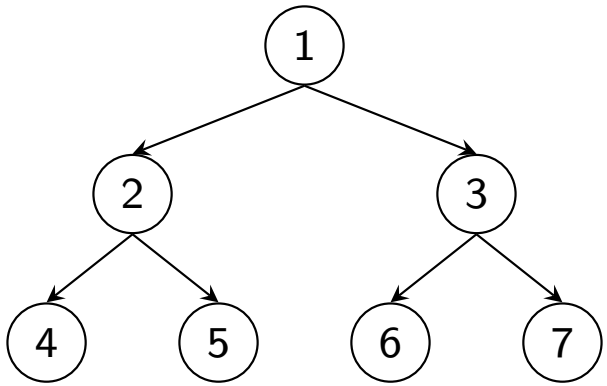
- **DFS**: little memory, finishes paths early. In a loop, it goes down forever.

Which Path First

DFS (`walk`)



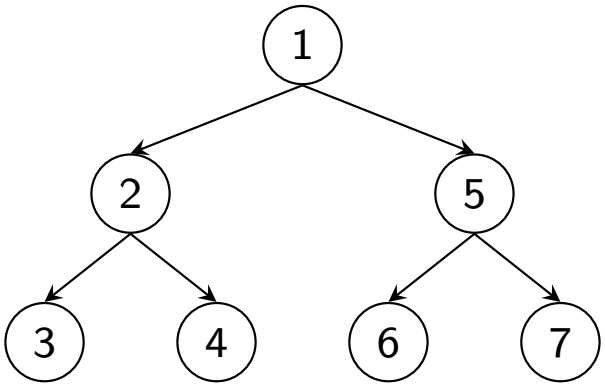
BFS



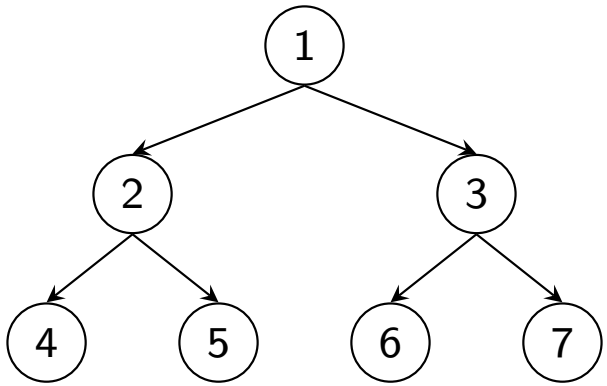
- **DFS**: little memory, finishes paths early. In a loop, it goes down forever.
- **BFS**: shallow code first. The frontier grows with the tree.

Which Path First

DFS (`walk`)



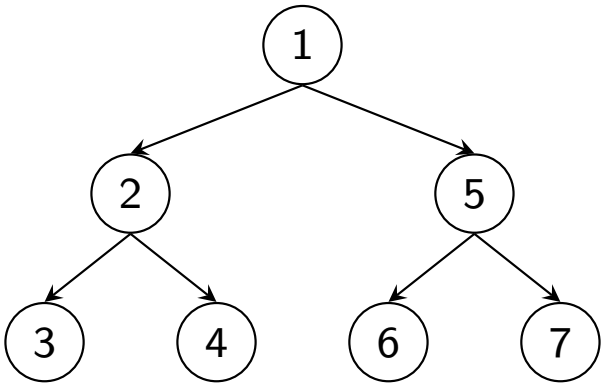
BFS



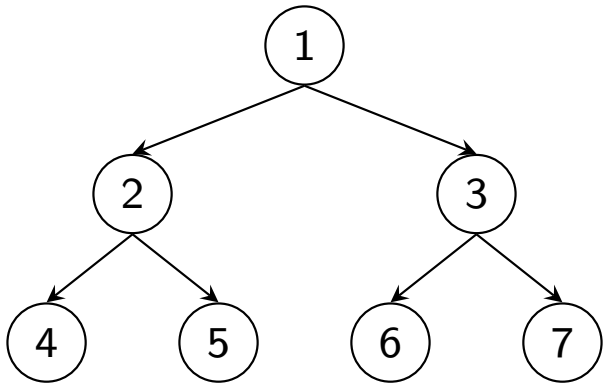
- **DFS**: little memory, finishes paths early. In a loop, it goes down forever.
- **BFS**: shallow code first. The frontier grows with the tree.
- **Coverage-guided**: pick the state that reaches code no path has reached yet (KLEE).

Which Path First

DFS (`walk`)



BFS



- **DFS**: little memory, finishes paths early. In a loop, it goes down forever.
- **BFS**: shallow code first. The frontier grows with the tree.
- **Coverage-guided**: pick the state that reaches code no path has reached yet (KLEE).
- A **heuristic**: no order wins on every program.

Merging States

Two of the paths reach the exit, point 1, with different $\mathtt{r}$:

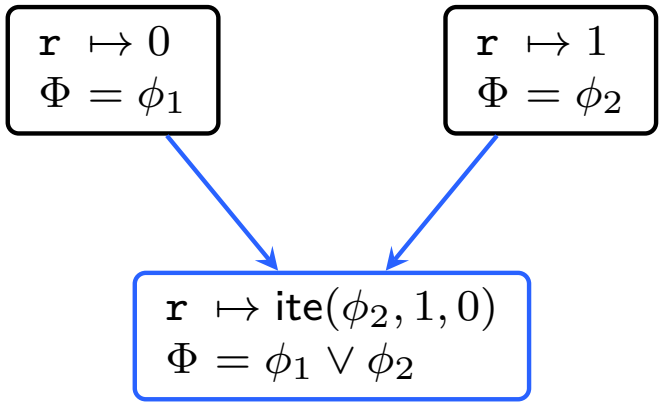
$\mathtt{r} \mapsto 0$
$\Phi = \phi_1$

$\mathtt{r} \mapsto 1$
$\Phi = \phi_2$

$$\phi_1 = 10 < \alpha + \beta \wedge \neg(\alpha < 3) \qquad \phi_2 = 10 < \alpha + \beta \wedge \alpha < 3 \wedge \neg(\beta < 8)$$

Merging States

Two of the paths reach the exit, point 1, with different $\mathtt{r}$:



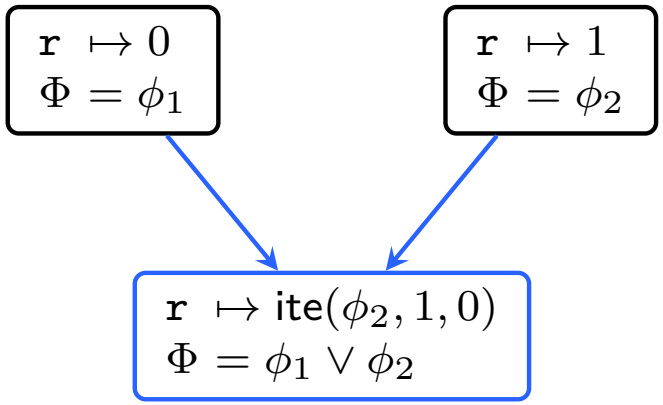
merge: one state, nothing lost

$$\phi_1 = 10 < \alpha + \beta \wedge \neg(\alpha < 3) \qquad \phi_2 = 10 < \alpha + \beta \wedge \alpha < 3 \wedge \neg(\beta < 8)$$

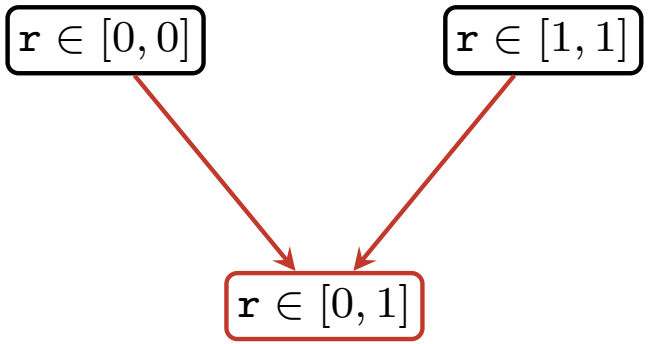
- **Merge**: the `if` moves **into** the formula. ite is if-then-else.

Merging States

Two of the paths reach the exit, point 1, with different $\mathtt{r}$:



merge: one state, nothing lost



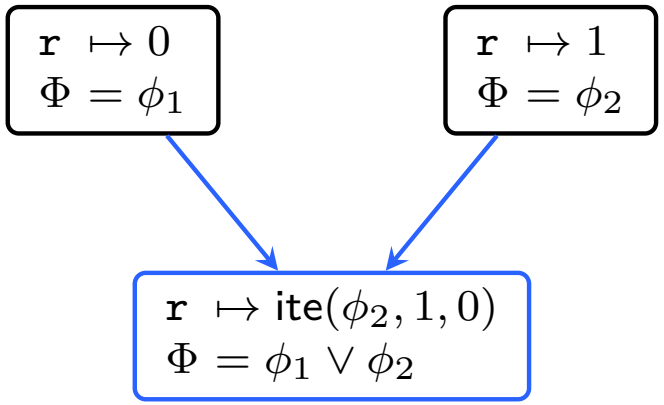
join (intervals): one state, lossy

$$\phi_1 = 10 < \alpha + \beta \wedge \neg(\alpha < 3) \qquad \phi_2 = 10 < \alpha + \beta \wedge \alpha < 3 \wedge \neg(\beta < 8)$$

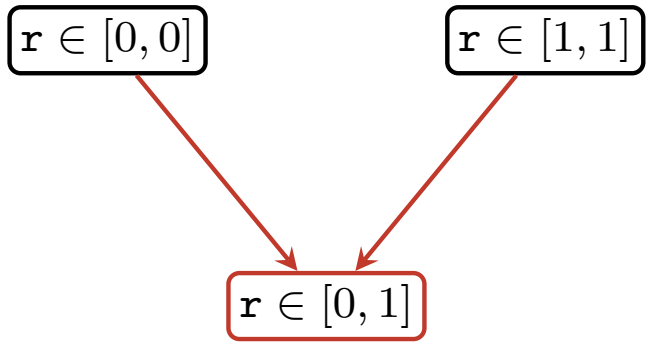
- **Merge**: the `if` moves **into** the formula. ite is if-then-else.
- Interval analysis **joins** the same two states: $\mathtt{r}$ is only somewhere in $[0, 1]$.

Merging States

Two of the paths reach the exit, point 1, with different `r`:



merge: one state, nothing lost



join (intervals): one state, lossy

$$\phi_1 = 10 < \alpha + \beta \wedge \neg(\alpha < 3) \qquad \phi_2 = 10 < \alpha + \beta \wedge \alpha < 3 \wedge \neg(\beta < 8)$$

- **Merge**: the `if` moves **into** the formula. ite is if-then-else.
- Interval analysis **joins** the same two states: `r` is only somewhere in $[0, 1]$.
- Fewer states, harder formulas. Real tools merge **some** paths, between "never" (today) and "always".

Contents

1. Symbolic Execution
   - Symbols Instead of Values
   - The Execution Tree

2. Solving
   - SMT Solvers
   - In Python

3. Path Explosion
   - Branches and Loops
   - Coping

4. Limits

What the Solver Cannot See

```
>>> reach(open("examples/query.py").read(), "sql = sql + tmp")
Traceback (most recent call last):
  ...
symex.NotSupported: Str(s='SELECT ...')
```

- The taint example has **strings**. Our `to_z3` refuses them; real tools use z3's string theory.

What the Solver Cannot See

```
>>> reach(open("examples/query.py").read(), "sql = sql + tmp")
Traceback (most recent call last):
  ...
symex.NotSupported: Str(s='SELECT ...')
```

- The taint example has **strings**. Our `to_z3` refuses them; real tools use z3's string theory.

```
row = db.query("SELECT role FROM users WHERE id = " + uid)
if row["role"] == "admin":          # what does the solver know? nothing
```

- Code we do not have — libraries, files, the network — must be **modelled** by hand.

What the Solver Cannot See

```
>>> reach(open("examples/query.py").read(), "sql = sql + tmp")
Traceback (most recent call last):
  ...
symex.NotSupported: Str(s='SELECT ...')
```

- The taint example has **strings**. Our `to_z3` refuses them; real tools use z3's string theory.

```
row = db.query("SELECT role FROM users WHERE id = " + uid)
if row["role"] == "admin":          # what does the solver know? nothing
```

- Code we do not have — libraries, files, the network — must be **modelled** by hand.
- The other way out: **run** the program for real, and keep symbols only where we can. Next lecture.

In Practice

tool	**analyzes**	**known for**
KLEE	C (LLVM)	tests for GNU Coreutils; bugs missed for 15 years
angr	binaries	CTFs; DARPA Cyber Grand Challenge
SAGE	x86 binaries	a third of the file-fuzzing bugs found in Windows 7
Z3, cvc5	formulas	the solvers under all of them

In Practice

tool	**analyzes**	**known for**
KLEE	C (LLVM)	tests for GNU Coreutils; bugs missed for 15 years
angr	binaries	CTFs; DARPA Cyber Grand Challenge
SAGE	x86 binaries	a third of the file-fuzzing bugs found in Windows 7
Z3, cvc5	formulas	the solvers under all of them

- **Unsound**: it misses paths past the depth bound and code it cannot model.

In Practice

tool	**analyzes**	**known for**
KLEE	C (LLVM)	tests for GNU Coreutils; bugs missed for 15 years
angr	binaries	CTFs; DARPA Cyber Grand Challenge
SAGE	x86 binaries	a third of the file-fuzzing bugs found in Windows 7
Z3, cvc5	formulas	the solvers under all of them

- **Unsound**: it misses paths past the depth bound and code it cannot model.
- **Complete**: every input it reports is **real**. Run it, and you reach the line.

In Practice

tool	**analyzes**	**known for**
KLEE	C (LLVM)	tests for GNU Coreutils; bugs missed for 15 years
angr	binaries	CTFs; DARPA Cyber Grand Challenge
SAGE	x86 binaries	a third of the file-fuzzing bugs found in Windows 7
Z3, cvc5	formulas	the solvers under all of them

- **Unsound**: it misses paths past the depth bound and code it cannot model.
- **Complete**: every input it reports is **real**. Run it, and you reach the line.
- The opposite corner from every analysis before this one.

Homework 3

- Fill in `symex.py`: `to_z3`, `transfer`, `reach_node`, and `returns`.
- Everything you need, and the tests that grade you:

  github.com/ku-plrg-classroom/py-symex
- The tests run the program on your input, so any correct input passes.
- Needs `pip install z3-solver`.
- Due December 7 on the **LMS**. Submit `symex.py` only.

Summary

PLRG

1. Symbolic Execution
   - Symbols Instead of Values
   - The Execution Tree
2. Solving
   - SMT Solvers
   - In Python
3. Path Explosion
   - Branches and Loops
   - Coping
4. Limits

Next Lecture

- Dynamic Analysis

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr