# IA159 Formal Methods for Software Analysis

## Symbolic Execution and Applications

Jan Strejček

Faculty of Informatics
Masaryk University

# Focus and sources

focus

- symbolic execution
- automated whitebox fuzz testing

sources

- J. C. King: Symbolic Execution and Program Testing, Communications of ACM, 1976.
- P. Godefroid, M. Y. Levin, and D. Molnar: Automated whitebox fuzz testing, NDSS 2008.

# Motivation

```
int sum(int a, int b, int c) {
  int x = a + b;
  int y = b + c;
  int z = x + y - b;
  return z;
}
```

testing checks that the program behaves correctly on selected inputs

- `sum(1,1,1)` returns `3`
- `sum(1,2,3)` returns `6`
- ...

# Motivation

```c
int sum(int a, int b, int c) {
  int x = a + b;
  int y = b + c;
  int z = x + y - b;
  return z;
}
```

testing checks that the program behaves correctly on selected inputs

- `sum(1,1,1)` returns `3`
- `sum(1,2,3)` returns `6`
- ...

# Motivation

```c
int sum(int a, int b, int c) {
  int x = a + b;
  int y = b + c;
  int z = x + y - b;
  return z;
}
```

we can execute the program with symbols $\alpha_1, \alpha_2, \alpha_3$ representing arbitrary input values

- sum($\alpha_1, \alpha_2, \alpha_3$) returns

# Motivation

```c
int sum(int a, int b, int c) {
  int x = a + b;
  int y = b + c;
  int z = x + y - b;
  return z;
}
```

we can execute the program with symbols $\alpha_1, \alpha_2, \alpha_3$ representing arbitrary input values

- `sum(`$\alpha_1, \alpha_2, \alpha_3$`)` returns $\alpha_1 + \alpha_2 + \alpha_3$ (if `int` interpreted as $\mathbb{Z}$)

$\rightarrow$ symbolic execution

# Symbolic execution semantics in general

each programming language has an execution semantics describing

- which data objects the program manipulates
- how statements manipulate data objects
- how control flows through the statements of a program

in symbolic execution semantics

- real data objects are replaced by symbolic ones, which are typically expressions over symbols $\alpha_1, \alpha_2, \ldots$ representing arbitrary input values
- the semantics of statements is extended to accept symbolic input and produce symbolic output
- control flow is handled differently as some branching conditions can be evaluated to both *true* and *false* depending on the values of symbols

# Symbolic execution

## assumptions and notation

- consider a program that handles only integer ($\mathbb{Z}$) variables and it is built from assignments and branching statements
- a special assignment `x = *` (or `x = input()`) corresponds to reading input
- *Vars* = the set of variables in the considered program
- $Sym = \{\alpha, \beta, \ldots, \alpha_1, \alpha_2, \ldots\}$ = countable set of symbols representing arbirary input values
- *Exp*(*Sym*) = expressions over *Sym*, integers, and arithmetic operations
- for example, $2\alpha + \beta^3 - 7 \in Exp(Sym)$
- *Exp*(*Sym*) are symbolic data objects

# Symbolic execution

## assumptions and notation

- consider a program that handles only integer ($\mathbb{Z}$) variables and it is built from assignments and branching statements
- a special assignment `x = *` (or `x = input()`) corresponds to reading input
- *Vars* = the set of variables in the considered program
- *Sym* $= \{\alpha, \beta, \ldots, \alpha_1, \alpha_2, \ldots\}$ = countable set of symbols representing arbirary input values
- *Exp(Sym)* = expressions over *Sym*, integers, and arithmetic operations
- for example, $2\alpha + \beta^3 - 7 \in \mathit{Exp}(\mathit{Sym})$
- *Exp*(*Sym*) are symbolic data objects

symbolic execution computes symbolic states consisting of

1. current program location
2. symbolic memory
3. path condition

# Symbolic memory $m$

- $m\colon$ *Vars* $\to$ *Exp*(*Sym*)
- assigns expressions of *Exp*(*Sym*) to program variables
- a symbol $\alpha \in$ *Sym* is called fresh if it was not used before in the considered computation
- initial symbolic memory $m_0$ assigns to each variable $\mathtt{x} \in$ *Vars* a fresh symbol $m(\mathtt{x}) = \alpha \in$ *Sym*
- symbolic memory is modified by assignments
- for any program expression $\mathtt{exp}$ (Boolean or integer), by $m(\mathtt{exp})$ we denote the expression where each program variable $\mathtt{x} \in$ *Vars* is replaced by $m(\mathtt{x})$

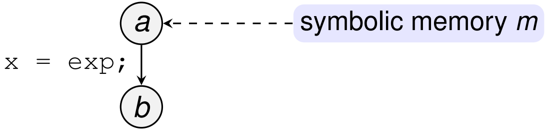
# Symbolic memory $m$

- $m\colon \mathit{Vars} \to \mathit{Exp}(\mathit{Sym})$
- assigns expressions of $\mathit{Exp}(\mathit{Sym})$ to program variables
- a symbol $\alpha \in \mathit{Sym}$ is called fresh if it was not used before in the considered computation
- initial symbolic memory $m_0$ assigns to each variable $\mathtt{x} \in \mathit{Vars}$ a fresh symbol $m(\mathtt{x}) = \alpha \in \mathit{Sym}$
- symbolic memory is modified by assignments
- for any program expression $\mathtt{exp}$ (Boolean or integer), by $m(\mathtt{exp})$ we denote the expression where each program variable $\mathtt{x} \in \mathit{Vars}$ is replaced by $m(\mathtt{x})$

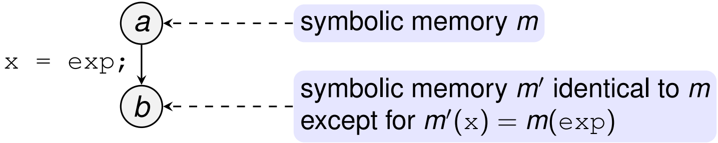
## example

- let $m(\mathtt{x}) = \alpha + 4$ and $m(\mathtt{y}) = 2\alpha + \beta$
- $m(\mathtt{5x - y + 8}) = 5(\alpha + 4) - (2\alpha + \beta) + 8 = 3\alpha - \beta + 28$
- $m(\mathtt{x\ !=\ y}) = (\alpha + 4 \neq 2\alpha + \beta) = (\alpha + \beta \neq 4)$

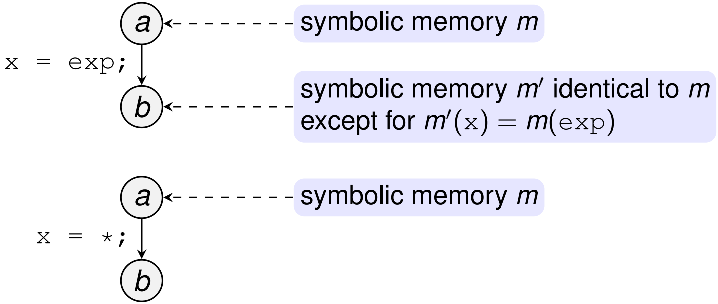
# Symbolic execution of assignments



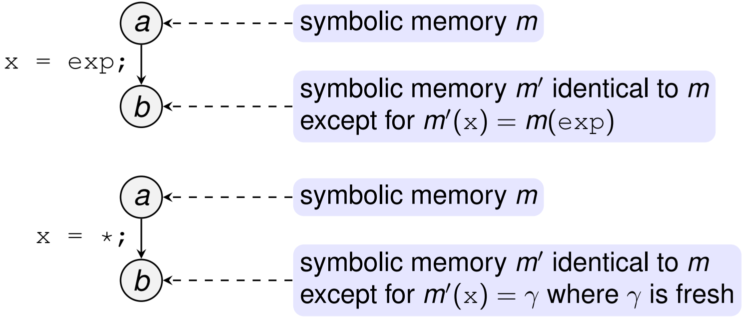
# Symbolic execution of assignments



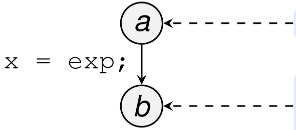
# Symbolic execution of assignments



# Symbolic execution of assignments

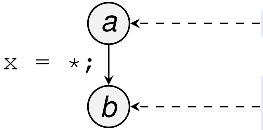


# Symbolic execution of assignments




symbolic memory $m$

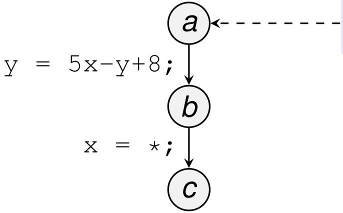
symbolic memory $m'$ identical to $m$ except for $m'(\mathtt{x}) = m(\mathtt{exp})$




symbolic memory $m$

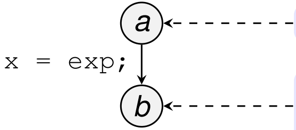
symbolic memory $m'$ identical to $m$ except for $m'(\mathtt{x}) = \gamma$ where $\gamma$ is fresh

example



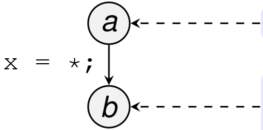

$m(\mathtt{x}) = \alpha + 4$
$m(\mathtt{y}) = 2\alpha + \beta$

# Symbolic execution of assignments




symbolic memory $m$

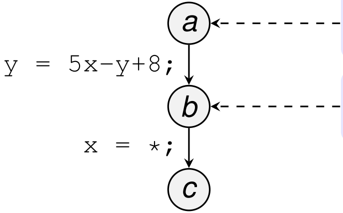
symbolic memory $m'$ identical to $m$ except for $m'(\mathtt{x}) = m(\mathtt{exp})$




symbolic memory $m$

symbolic memory $m'$ identical to $m$ except for $m'(\mathtt{x}) = \gamma$ where $\gamma$ is fresh

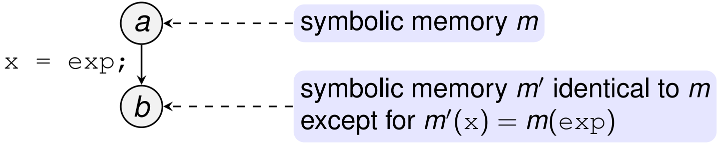
example

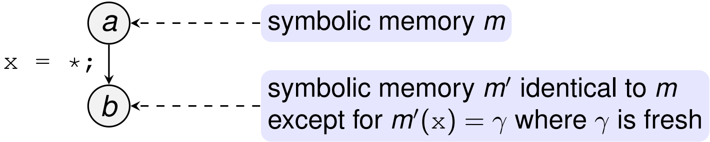



$m(\mathtt{x}) = \alpha + 4$
$m(\mathtt{y}) = 2\alpha + \beta$

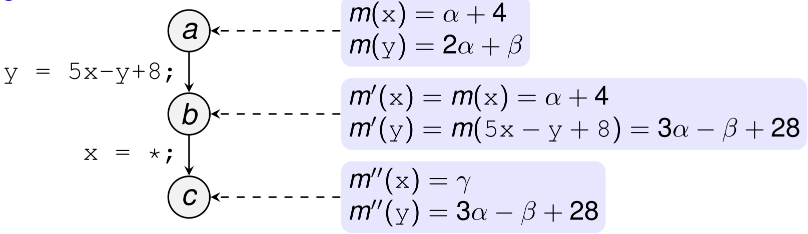
$m'(\mathtt{x}) = m(\mathtt{x}) = \alpha + 4$
$m'(\mathtt{y}) = m(\mathtt{5x - y + 8}) = 3\alpha - \beta + 28$

# Symbolic execution of assignments




example

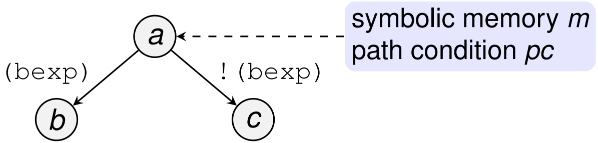


# Path condition *pc*

- a quantifier-free predicate formula over *Sym* corresponding to a program path
- *pc* is the necessary and sufficient condition on input values to navige the program execution along the current path
- if *pc* is not satisfiable the corresponding path is unfeasible
- *pc* is initially set to *true*
- *pc* is modified by evaluation of branching statements

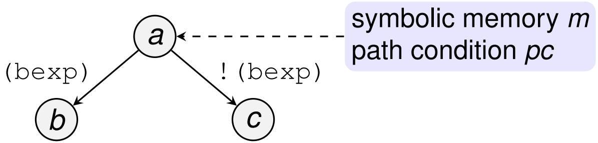
# Symbolic execution of branching statements

```
if (bexp) {...} else {...}
```



# Symbolic execution of branching statements

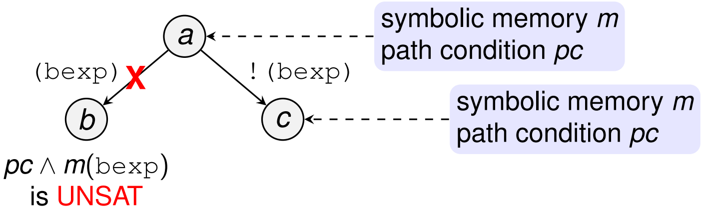
```
if (bexp) {...} else {...}
```




1. check feasability of the *true* branch:
   if $pc \wedge m(\texttt{bexp})$ is not satisfiable, continue to the *false* branch

# Symbolic execution of branching statements

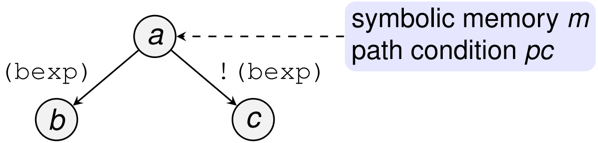
`if (bexp) {...} else {...}`




1. check feasability of the *true* branch:
   if $pc \wedge m(\texttt{bexp})$ is not satisfiable, continue to the *false* branch

# Symbolic execution of branching statements

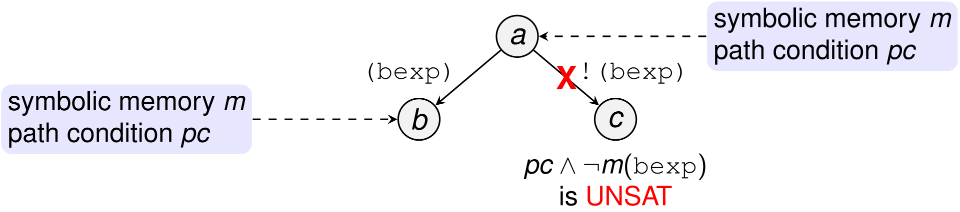
```
if (bexp) {...} else {...}
```




1. check feasability of the *true* branch:
   if $pc \wedge m(\texttt{bexp})$ is not satisfiable, continue to the *false* branch
2. check feasability of the *false* branch:
   if $pc \wedge \neg m(\texttt{bexp})$ is not satisfiable, continue to the *true* branch

# Symbolic execution of branching statements

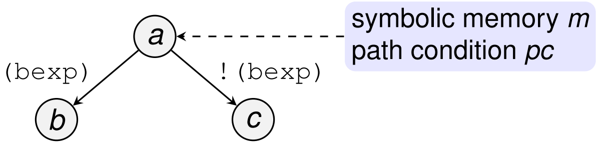
```
if (bexp) {...} else {...}
```




1. check feasability of the *true* branch:
   if $pc \wedge m(\texttt{bexp})$ is not satisfiable, continue to the *false* branch
2. check feasability of the *false* branch:
   if $pc \wedge \neg m(\texttt{bexp})$ is not satisfiable, continue to the *true* branch

# Symbolic execution of branching statements

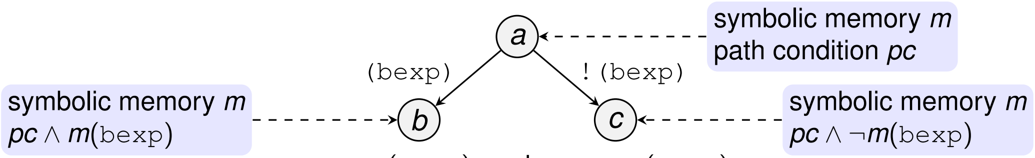
```
if (bexp) {...} else {...}
```




1. check feasability of the *true* branch:
   if $pc \wedge m(\texttt{bexp})$ is not satisfiable, continue to the *false* branch
2. check feasability of the *false* branch:
   if $pc \wedge \neg m(\texttt{bexp})$ is not satisfiable, continue to the *true* branch
3. if both are satisfiable, fork the symbolic execution
   set *pc* in *true* branch to $pc \wedge m(\texttt{bexp})$
   set *pc* in *false* branch to $pc \wedge \neg m(\texttt{bexp})$

# Symbolic execution of branching statements

```
if (bexp) {...} else {...}
```

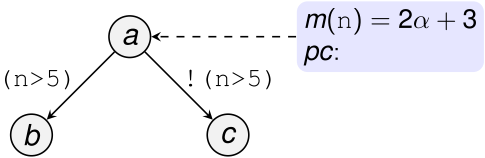



$pc \wedge m(\texttt{bexp})$ and $pc \wedge \neg m(\texttt{bexp})$
are SAT

1. check feasability of the *true* branch:
   if $pc \wedge m(\texttt{bexp})$ is not satisfiable, continue to the *false* branch
2. check feasability of the *false* branch:
   if $pc \wedge \neg m(\texttt{bexp})$ is not satisfiable, continue to the *true* branch
3. if both are satisfiable, fork the symbolic execution
   set *pc* in *true* branch to $pc \wedge m(\texttt{bexp})$
   set *pc* in *false* branch to $pc \wedge \neg m(\texttt{bexp})$

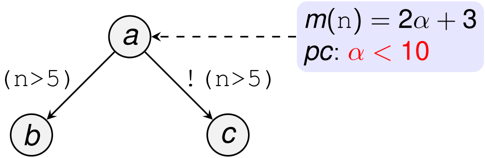
# Symbolic execution of branching statements: example

```
if (n>5) {...} else {...}
```



# Symbolic execution of branching statements: example

```
if (n>5) {...} else {...}
```

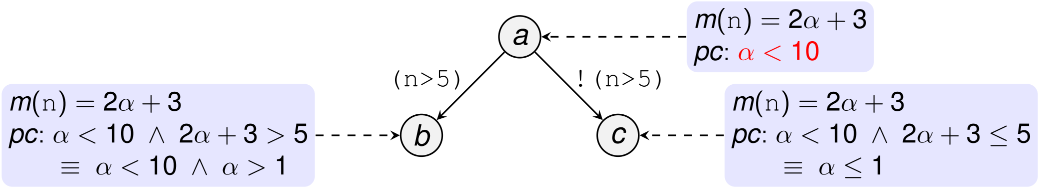



*pc* is $\alpha < 10$

- *true* branch is feasible as $\alpha < 10 \;\wedge\; m(\mathtt{n} > 5) \;\equiv\; \alpha < 10 \;\wedge\; 2\alpha + 3 > 5$ is satisfiable (e.g. by $\alpha = 3$)
- *false* branch is feasible as $\alpha < 10 \;\wedge\; \neg m(\mathtt{n} > 5) \;\equiv\; \alpha < 10 \;\wedge\; 2\alpha + 3 \leq 5$ is satisfiable (e.g. by $\alpha = 0$)
- fork execution and update *pc* on both branches

# Symbolic execution of branching statements: example

```
if (n>5) {...} else {...}
```

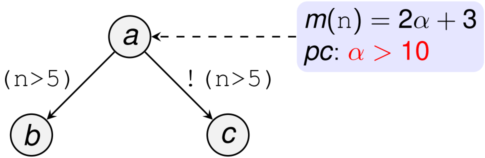



*pc* is $\alpha < 10$

- *true* branch is feasible as $\alpha < 10 \;\wedge\; m(\mathrm{n} > 5) \;\equiv\; \alpha < 10 \;\wedge\; 2\alpha + 3 > 5$ is satisfiable (e.g. by $\alpha = 3$)
- *false* branch is feasible as $\alpha < 10 \;\wedge\; \neg m(\mathrm{n} > 5) \;\equiv\; \alpha < 10 \;\wedge\; 2\alpha + 3 \leq 5$ is satisfiable (e.g. by $\alpha = 0$)
- fork execution and update *pc* on both branches

# Symbolic execution of branching statements: example

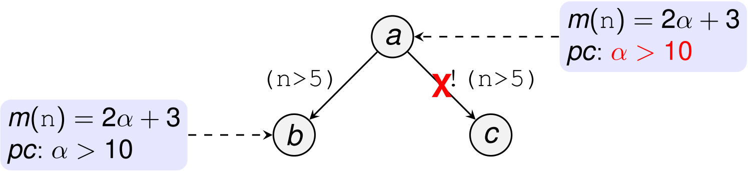
```
if (n>5) {...} else {...}
```




$pc$ is $\alpha > 10$

- *false* branch is unfeasible as
  $\alpha > 10 \;\wedge\; \neg m(\mathtt{n} > 5) \;\equiv\; \alpha > 10 \;\wedge\; 2\alpha + 3 \leq 5 \;\equiv\;$ *false*
- continue to *true* branch with the same *pc*

```
if (n>5) {...} else {...}
```

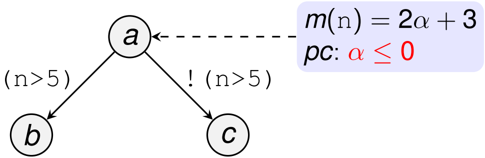



*pc* is $\alpha > 10$

- *false* branch is unfeasible as
  $\alpha > 10 \ \wedge \ \neg m(\mathrm{n} > 5) \ \equiv \ \alpha > 10 \ \wedge \ 2\alpha + 3 \leq 5 \ \equiv$ *false*
- continue to *true* branch with the same *pc*

# Symbolic execution of branching statements: example

```
if (n>5) {...} else {...}
```

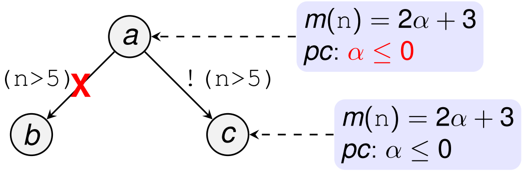



*pc* is $\alpha \leq 0$

- *true* branch is unfeasible as
  $\alpha \leq 0 \;\wedge\; m(\mathtt{n} > 5) \;\equiv\; \alpha \leq 0 \;\wedge\; 2\alpha + 3 > 5 \;\equiv\;$ *false*
- continue to *false* branch with the same *pc*

# Symbolic execution of branching statements: example

```
if (n>5) {...} else {...}
```




*pc* is $\alpha \leq 0$

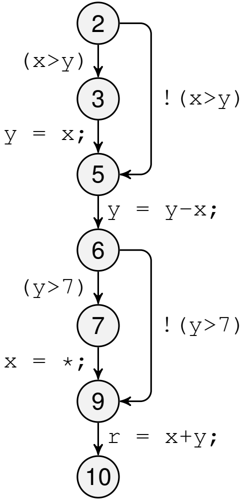
- *true* branch is unfeasible as
  $\alpha \leq 0 \ \wedge \ m(\mathrm{n} > 5) \ \equiv \ \alpha \leq 0 \ \wedge \ 2\alpha + 3 > 5 \ \equiv \ \mathit{false}$
- continue to *false* branch with the same *pc*

# Symbolic execution tree

- nodes are states of symbolic execution, i.e., triples $(l, m, pc)$ of program location $l$, symbolic memory $m$, and path condition $pc$
- root node $(l_0, m_0, true)$ consists of initial program location $l_0$, initial symbolic memory $m_0$ assigning fresh symbols, and initial path condition $true$
- successors are computed by symbolic execution of the assignment or branching statement corresponding to the current location
- only locations with branching statement can have more successors

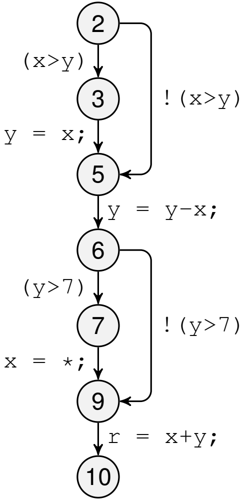
# Symbolic execution tree: example 1

```
1  int foo(int x, int y) {
2    if (x>y) {
3      y = x;
4    }
5    y = y-x;
6    if (y>7) {
7      x = *;
8    }
9    return x+y;
10 }
```

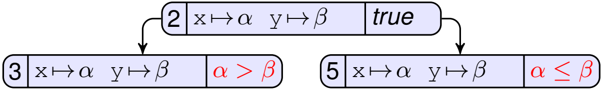


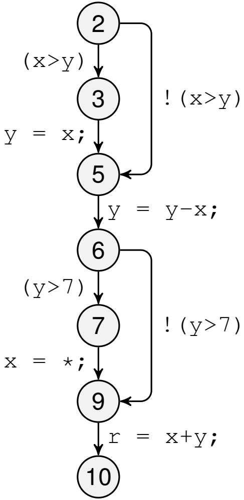
# Symbolic execution tree: example 1

| | | |
|---|---|---|
| 2 | $x \mapsto \alpha \quad y \mapsto \beta$ | *true* |

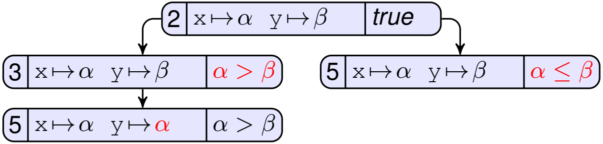


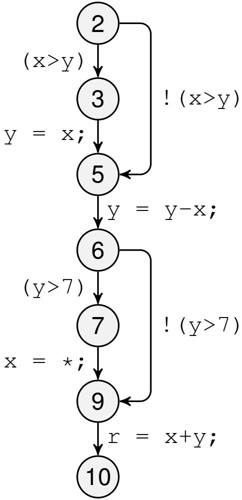
# Symbolic execution tree: example 1



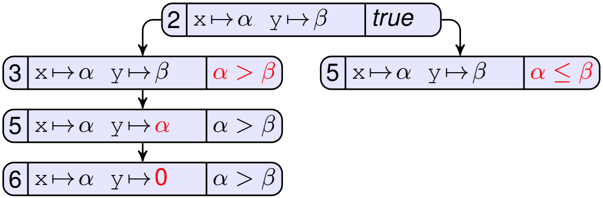



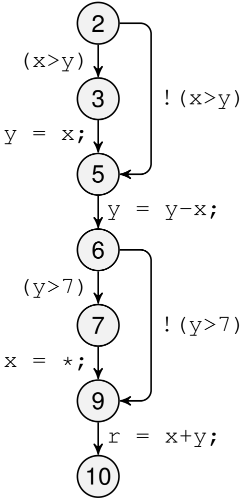
# Symbolic execution tree: example 1



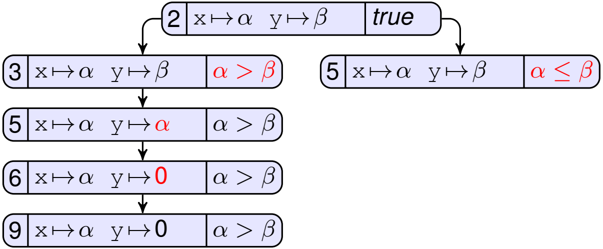



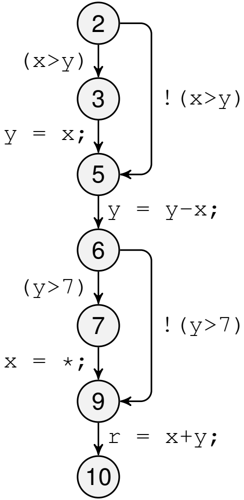
# Symbolic execution tree: example 1



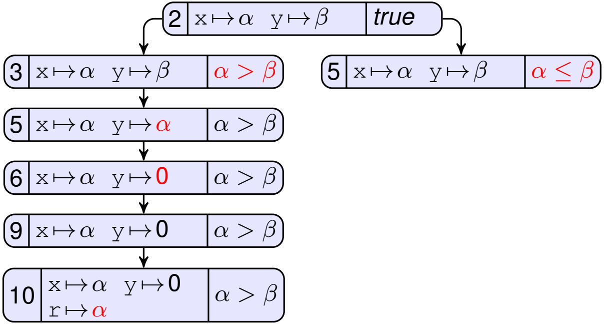



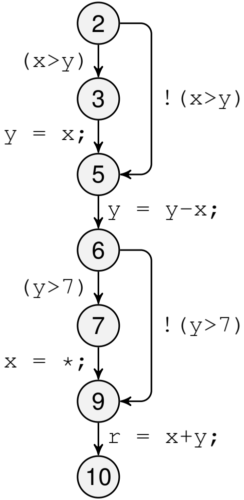
# Symbolic execution tree: example 1



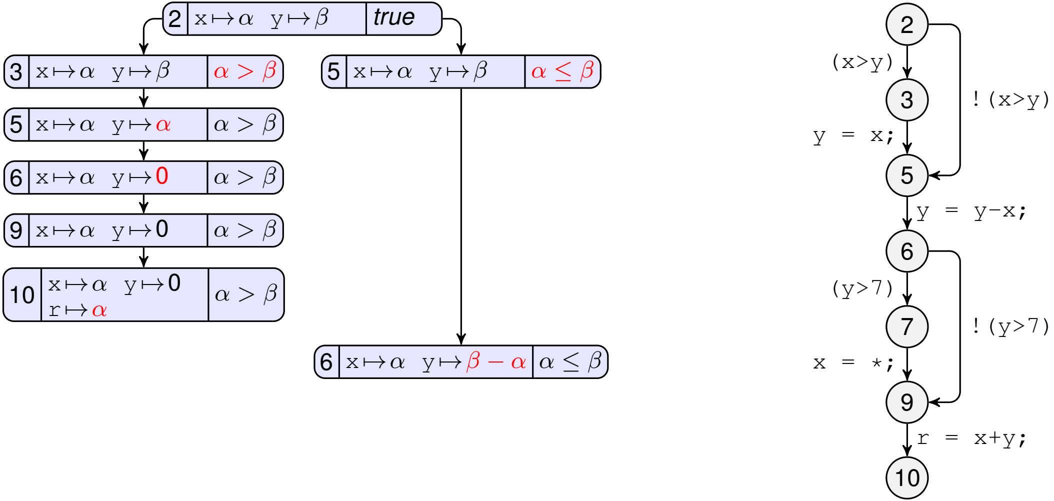



# Symbolic execution tree: example 1

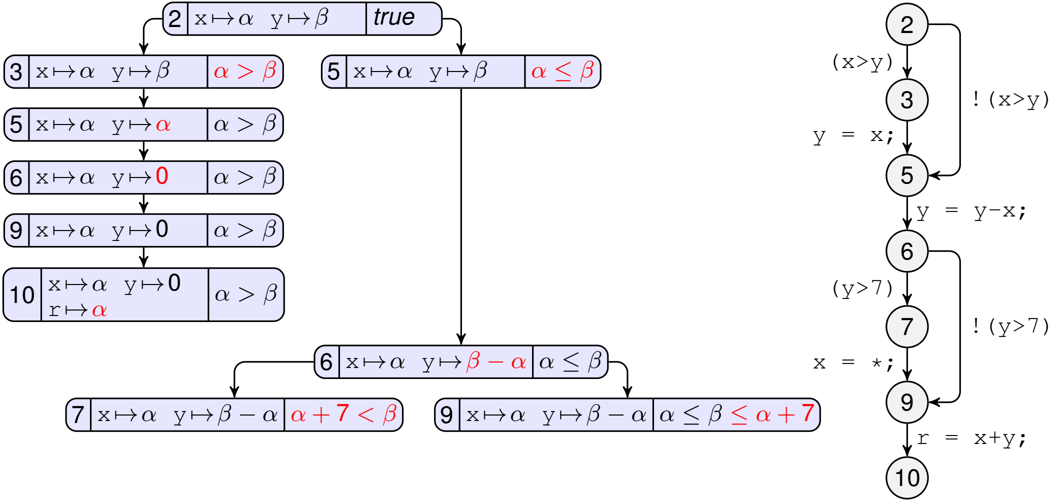


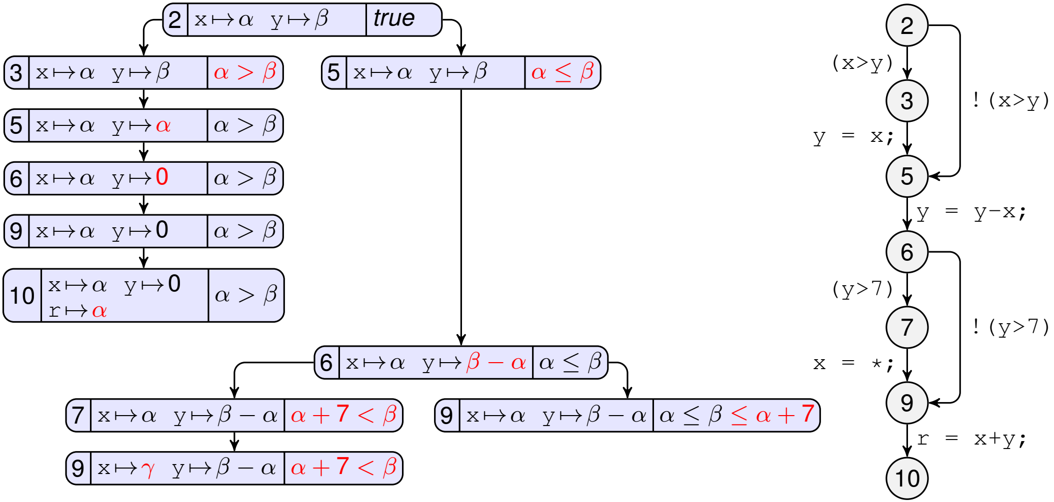
# Symbolic execution tree: example 1



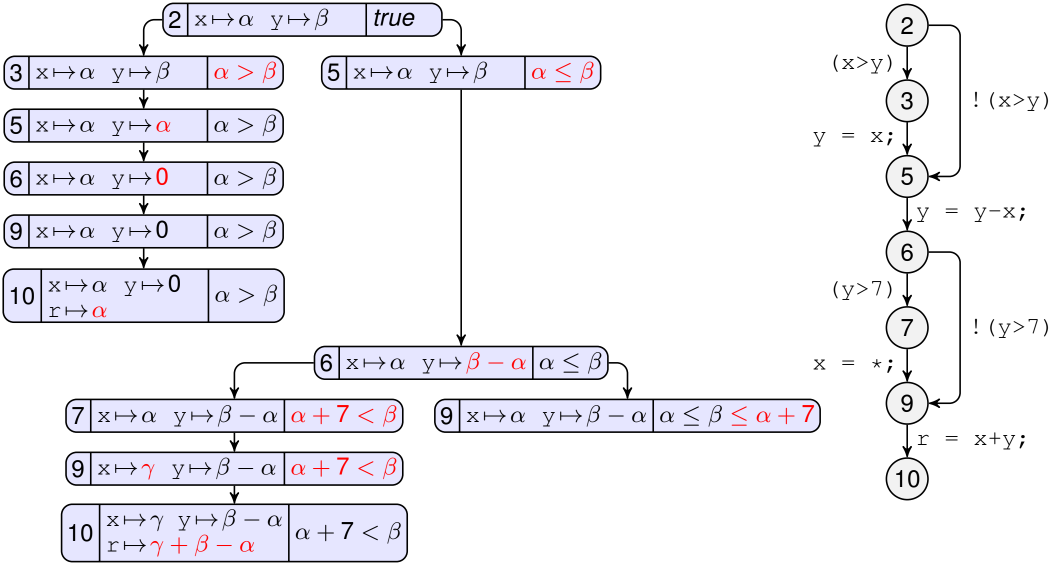
# Symbolic execution tree: example 1



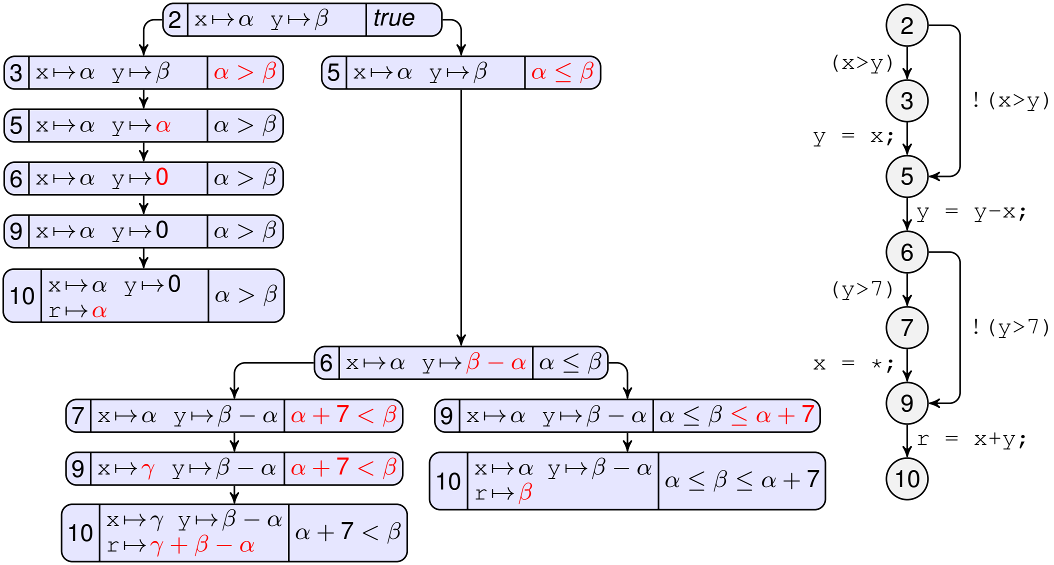
# Symbolic execution tree: example 1



# Symbolic execution tree: example 1

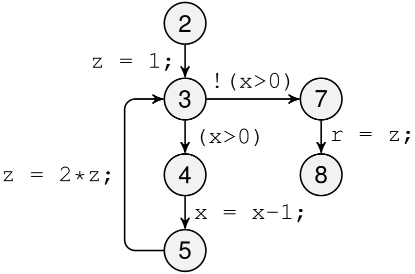


# Symbolic execution tree: example 1



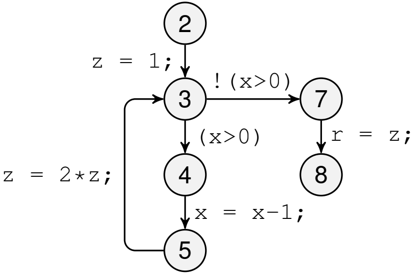
# Symbolic execution tree: example 2

```
int power(int x) {
  int z = 1;
  while (x>0) {
    x = x-1;
    z = 2*z;
  }
  return z;
}
```

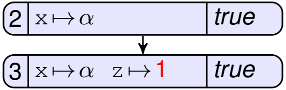


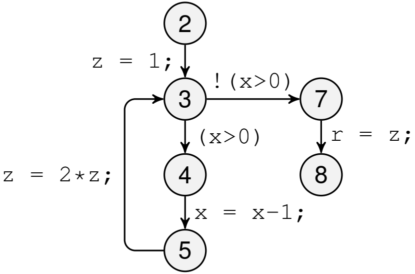
# Symbolic execution tree: example 2

| | | |
|---|---|---|
| 2 | $\mathrm{x} \mapsto \alpha$ | *true* |

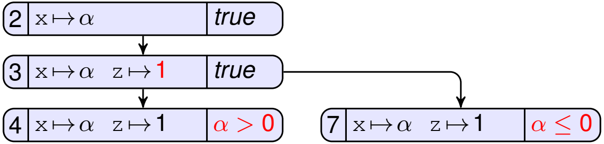


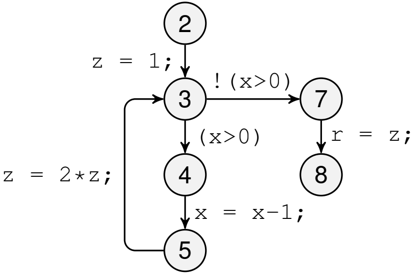
# Symbolic execution tree: example 2



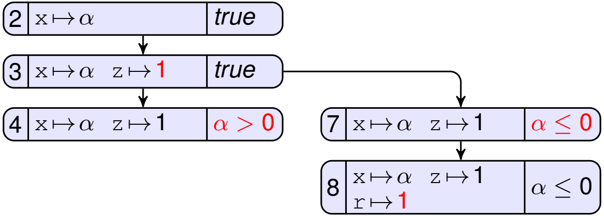



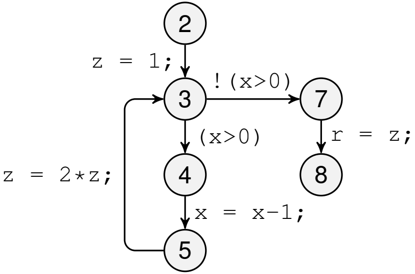
# Symbolic execution tree: example 2



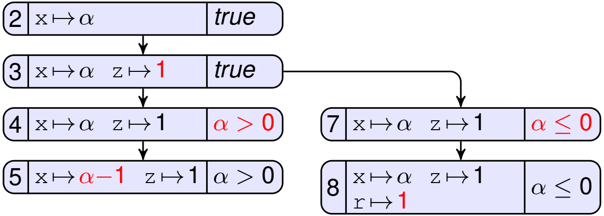



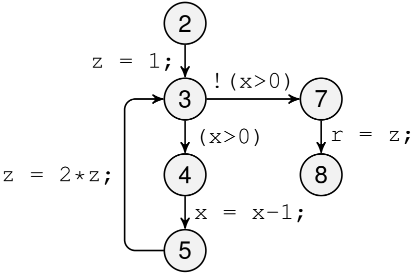
# Symbolic execution tree: example 2



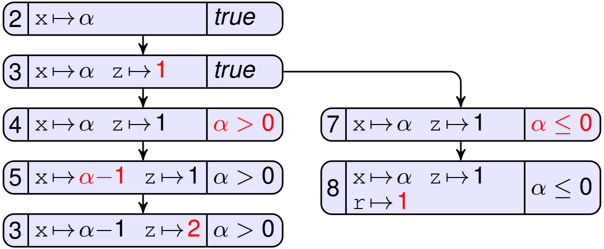



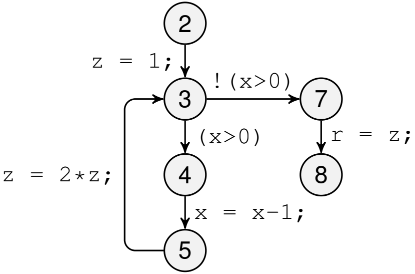
# Symbolic execution tree: example 2



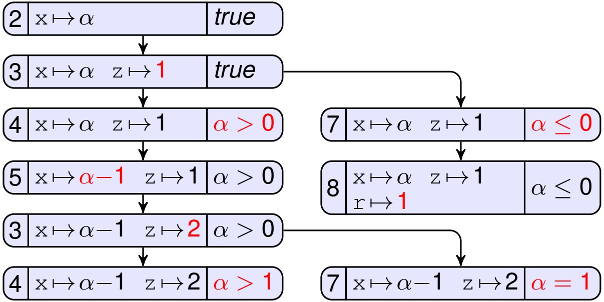



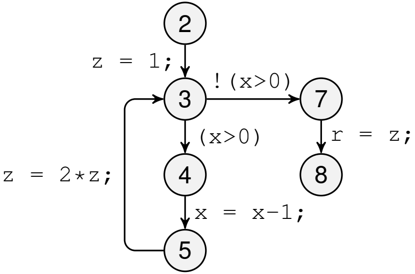
# Symbolic execution tree: example 2



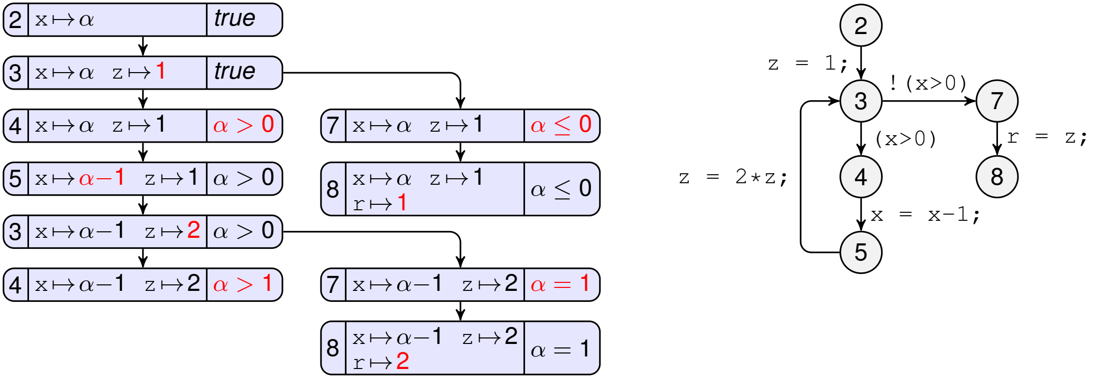



# Symbolic execution tree: example 2

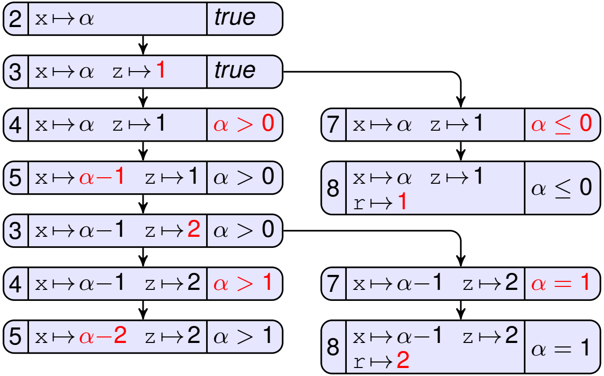


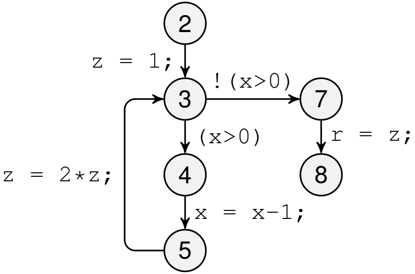



# Symbolic execution tree: example 2

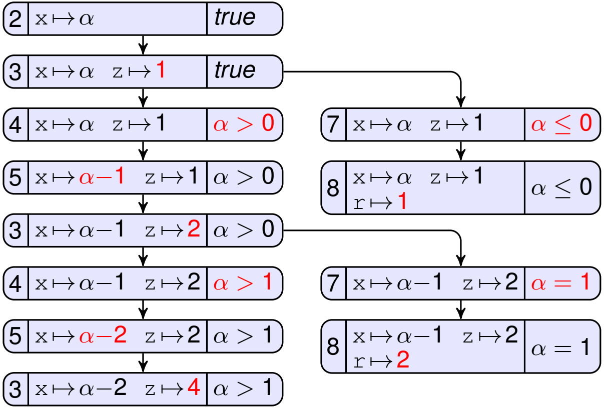


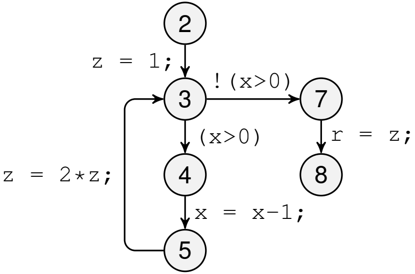
# Symbolic execution tree: example 2



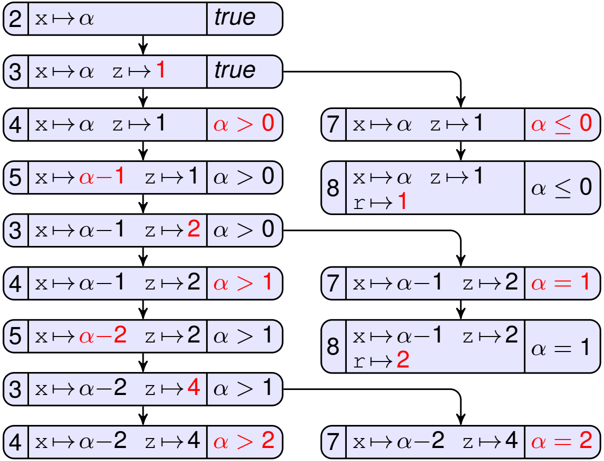



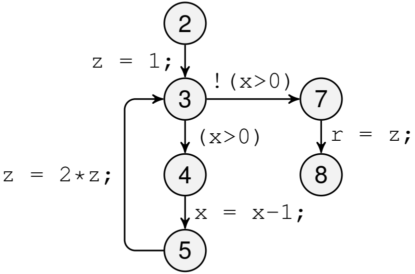
# Symbolic execution tree: example 2



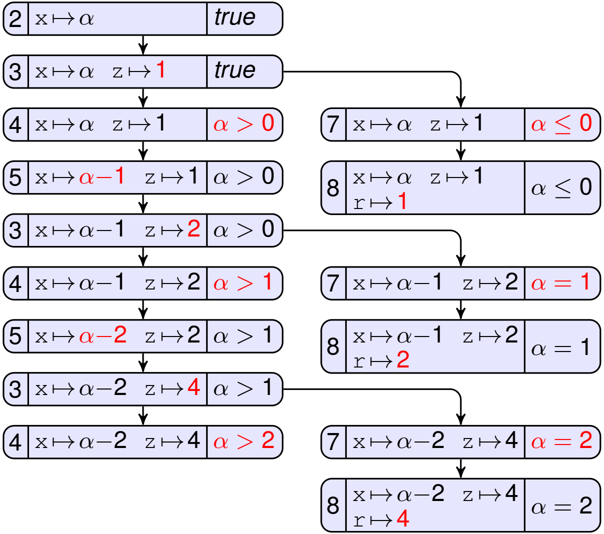



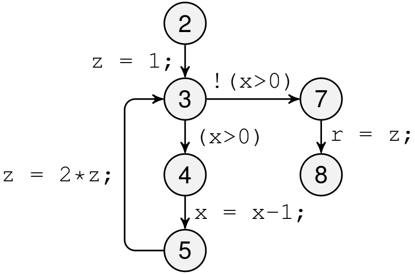
# Symbolic execution tree: example 2

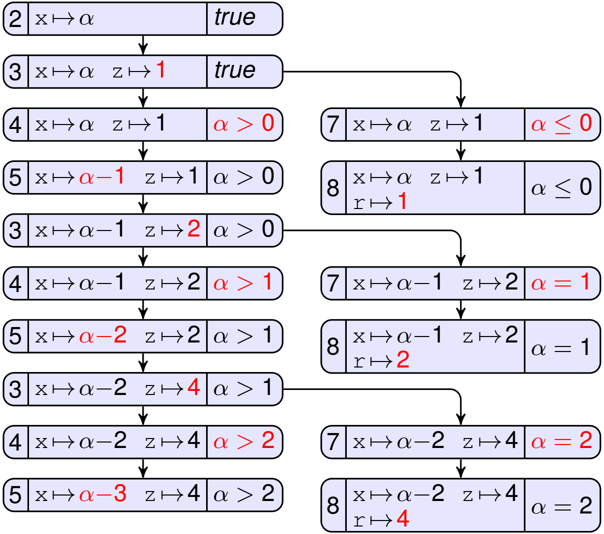


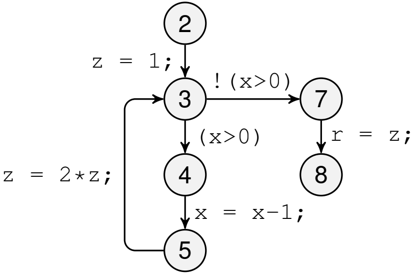



# Symbolic execution tree: example 2

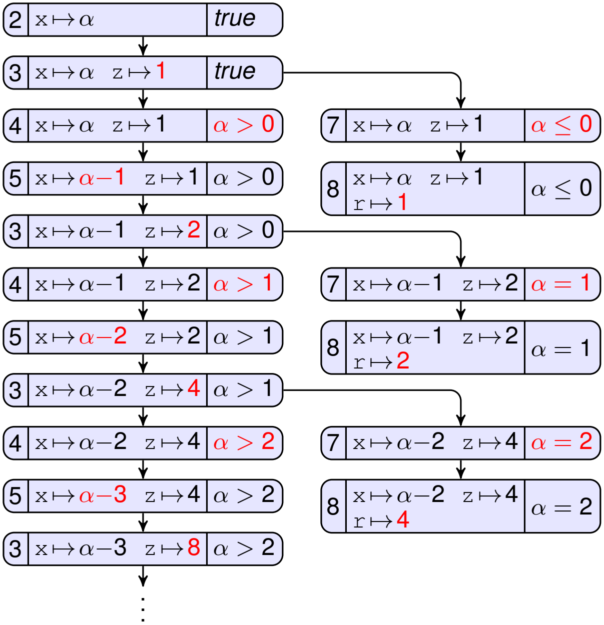


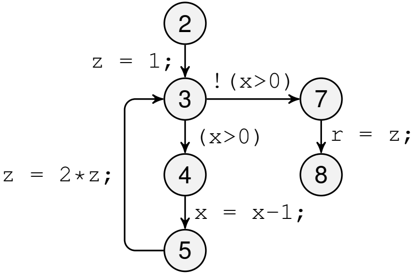




2 x ↦ α true
3 x ↦ α z ↦ 1 true
4 x ↦ α z ↦ 1 α > 0
7 x ↦ α z ↦ 1 α ≤ 0
5 x ↦ α−1 z ↦ 1 α > 0
8 x ↦ α z ↦ 1 r ↦ 1 α ≤ 0
3 x ↦ α−1 z ↦ 2 α > 0
4 x ↦ α−1 z ↦ 2 α > 1
7 x ↦ α−1 z ↦ 2 α = 1
5 x ↦ α−2 z ↦ 2 α > 1
8 x ↦ α−1 z ↦ 2 r ↦ 2 α = 1
3 x ↦ α−2 z ↦ 4 α > 1
4 x ↦ α−2 z ↦ 4 α > 2
7 x ↦ α−2 z ↦ 4 α = 2
5 x ↦ α−3 z ↦ 4 α > 2
8 x ↦ α−2 z ↦ 4 r ↦ 4 α = 2


2
z = 1;
3
!(x>0)
7
(x>0)
r = z;
z = 2*z;
4
8
x = x-1;
5

# Symbolic execution tree: example 2



# Properties of symbolic execution

- there is a bijection between paths in the symbolic execution tree (starting in its root) and feasible execution paths of the program
- the path condition gives the necessary and sufficient condition on input values to drive the execution along the corresponding path

# Properties of symbolic execution

- there is a bijection between paths in the symbolic execution tree (starting in its root) and feasible execution paths of the program
- the path condition gives the necessary and sufficient condition on input values to drive the execution along the corresponding path
- in each symbolic execution, the path condition is satisfiable
  - initially, *pc* is set to *true*
  - *pc* is changed only when both branches of a branching statements are feasible
  - *pc* is extended with a conjunct corresponding to the corresponding branch and the new conjunction is satisfiable as the branch is feasible
- path conditions $pc_1, pc_2$ corresponding to two distinct leaves of the symbolic execution tree are mutually exclusiove, i.e., $pc_1 \wedge pc_2 \equiv \mathit{false}$
- if the symbolic executon tree is finite, then the disjunction of all path conditions in its leaves is equivalent to *true*

# Applications in verification

Programs can be enriched with `assume(`$\varphi$`)` and `assert(`$\varphi$`)` statements. When symbolic execution passes through

- `assume(`$\varphi$`)`, it executes $pc \leftarrow pc \wedge \varphi$.
- `assert(`$\varphi$`)` and $pc \implies \varphi$ is not valid, it reports an error.

With these constructs, symbolic execution can be used with a modification of Floyd's proof method to prove program correctness.

This application is straightforward for any program whose symbolic execution tree is finite.

# Practical issues

- deciding validity or satisfiability of formulas can be expensive or even impossible (e.g. for our simple language with unbounded data types)
- in practice, symbolic execution uses expressions and formulas over bitvector theory (operations and relations correspond to CPU instructions, e.g. artihmetic operations with overflows, bitwise operations, etc.), where validity and satisfiability are decidable (but expensive)

# Practical issues

## variable storage referencing problem

- when $i$ is dependent on input, then $A[i]$ can point to various locations in memory
- unsatisfactory solution:
  handle $A[i]$ as $ITE(i = 1, A[1], ITE(i = 2, A[2], \ldots))$

# Practical issues

## variable storage referencing problem

- when $i$ is dependent on input, then $A[i]$ can point to various locations in memory
- unsatisfactory solution:
  handle $A[i]$ as $ITE(i = 1, A[1], ITE(i = 2, A[2], \ldots))$

## other memory related problems

- reading/writing via pointers
- comparison of addresses (inner program nondeterminism)
- allocation of memory blocks of symbolic size

# Practical issues

## variable storage referencing problem

- when $i$ is dependent on input, then $A[i]$ can point to various locations in memory
- unsatisfactory solution:
  handle $A[i]$ as $ITE(i = 1, A[1], ITE(i = 2, A[2], \ldots))$

## other memory related problems

- reading/writing via pointers
- comparison of addresses (inner program nondeterminism)
- allocation of memory blocks of symbolic size

solution: fully symbolic memory model

- performance issues

# Practical issues

- path explosion problem
  - the number of branches in the symbolic execution tree can be extremely high or even infinite
  - typical for program cycles with the number of iterations depending on the input (symbolic execution forks again and again)
  - construction of full symbolic execution tree is often infeasible
- issues with complex arithmetic operations (e.g. in hashing, encryption or decryption), calls to the operating system and libraries
- practical solutions
  - concretization
  - concolic execution

# Concolic execution

- concolic = concrete + symbolic
- program is executed on a real input and on symbolic input simultaneously
- symbolic execution does not fork, it always follows the concrete execution and computes *pc*
- if a symbolic value is not available, we can switch to a concrete one

# Real applications

- typical applications
  - bug finding
  - test generation
  - analysis of abstract error traces
- often combined with other techniques
- used in many tools including Klee, PEX, SAGE, SLAM, Ultimate Automizer, Symbiotic

# Automated whitebox fuzz testing

# Automated whitebox fuzz testing

- an example of modern and sophisticated testing method
- implemented in SAGE (Scalable, Automated, Guided Execution)
- discovered 30+ new bugs in large-shipped (and thus intensively tested) file-reading Windows applications including image processors, media players, file decoders
- combines fuzz testing and symbolic execution

# Key ideas

- symbolic execution is expensive compared to running tests
- thus we want to generate as many new inputs from one symbolic execution as possible
- input for the next symbolic execution is selected by some scoring function applied to all generated inputs
- in particular, the input that explored the most (so-far uncovered) pieces of code is chosen for the next symbolic execution

# The main algorithm

```
procedure GenerateInputs(inputSeed)
    inputSeed.bound ← 0
    workList ← {inputSeed}
    Run&Check(program, inputSeed)
    while workList ≠ ∅ do
        input ← PickFirstItem(workList)
        childInputs ← ExpandExecution(input)
        foreach newInput ∈ childInputs do
            Run&Check(program, newInput)
            Score(newInput)
            workList ← workList ∪ {newInput}
```

- `Score`(newInput) counts the newly covered blocks
- workList is ordered by the score of inputs
- `PickFirstItem`(workList) returns the input with the highest score

# Application of symbolic execution

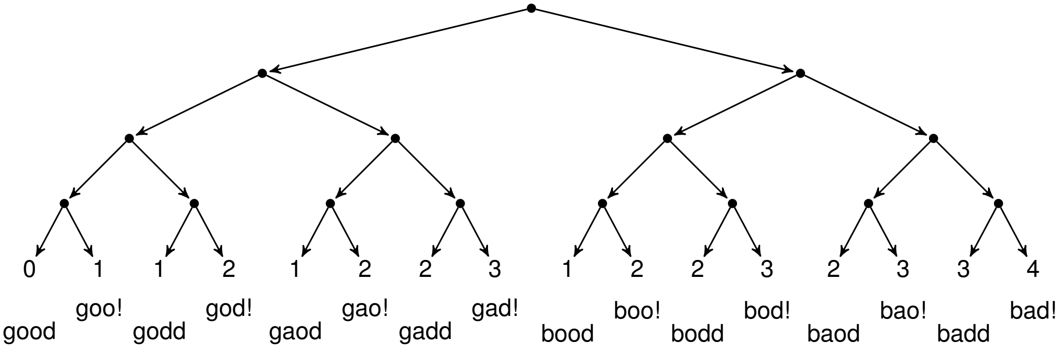
**procedure** ExpandExecution(input)
  childInputs $\leftarrow \emptyset$
  PC $\leftarrow$ SymbolicExecution(program, input)
  **for** $j \leftarrow$ input.bound **to** $|PC| - 1$ **do**
    **if** $\bigwedge_{i=0}^{j-1} PC[i] \ \wedge \ \neg PC[j]$ has solution $M$ **then**
      newInput $\leftarrow$ Combine(input, $M$)
      newInput.bound $\leftarrow j$
      childInputs $\leftarrow$ childInputs $\cup$ {newInput}
  **return** childInputs

- Combine(input, $M$)
  - creates a new input from the original input and $M$
  - Combine("abcde", input[3] $\mapsto$ "F") returns "abcFe"
- path conditions are represented as arrays PC of conjuncts

# Example

```c
void top(char input[4]) {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 3) abort(); // error
}
```

```c
void top(char input[4]) {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 3) abort(); // error
}
```



# Notes

- the algorithm can be parallelized: only workList and the overall block coverage need to be shared
- SAGE recovers easily from divergencies (situations when an execution deviates from the assumed execution path) induced e.g. by inner program nondeterminism
- SAGE runs 24/7 on large clusters, available for Microsoft developers