

Adobe® Symbolic Execution

James C. King
Principal Scientist
Enterprise Technology Lab (ETL)
Office of Technology (OoT)





References

“Symbolic Execution and Program Testing”

James C. King

Communications of the ACM, Vol.19, No. 7, July 1976

<http://technology.corp.adobe.com/users/jking/Presentations/ProgramCorrectness/KingPapers/SymbolicExecution1976.pdf>

“An Introduction to Proving the Correctness of Programs”

Sidney L. Hantler and James C. King

ACM Computing Surveys, Vol. 8, No. 3, September 1976

<http://technology.corp.adobe.com/users/jking/Presentations/ProgramCorrectness/KingPapers/Tutorial1976.pdf>

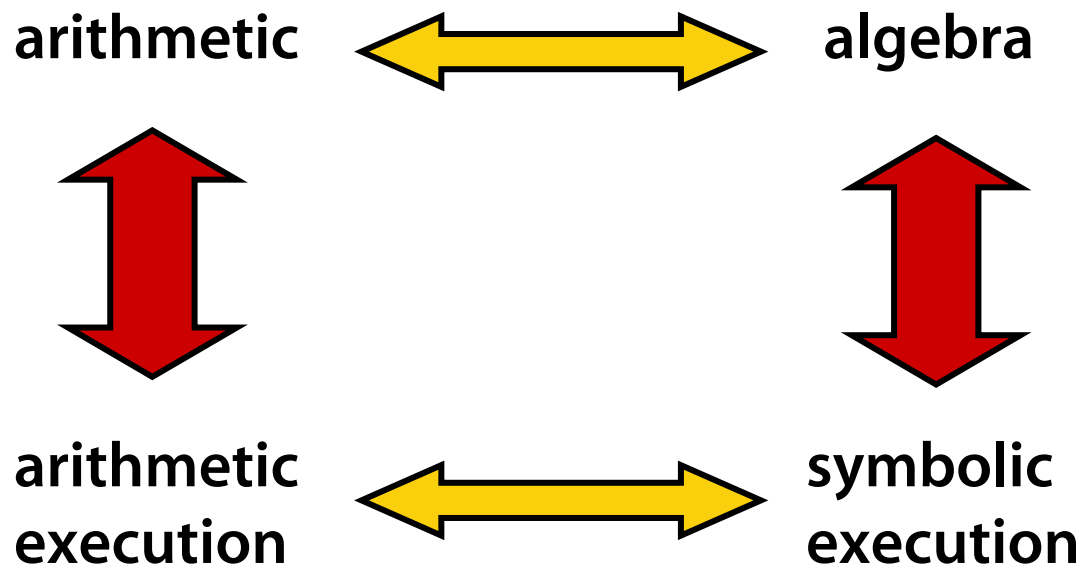


Outline

- **Very basic symbolic execution**
- **Finite symbolic execution trees**
- **Correctness assertions**
- **Assume and Prove statements**
- **Definition of executing the “if” statement**
- **Infinite symbolic execution trees**
- **Proofs of correctness**
- **Subroutines and functions**
- **Arrays**



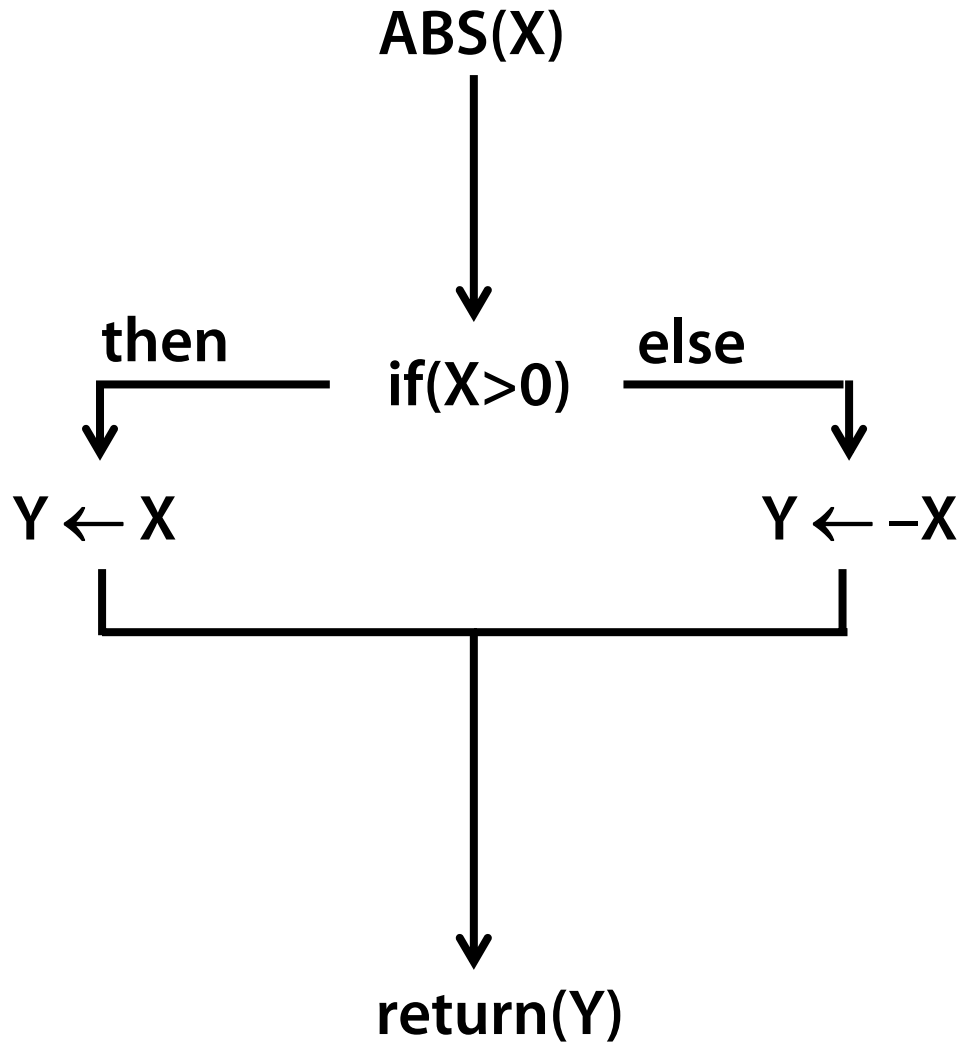
Simple Analog



Symbolic (algebraic) discoveries cover whole classes of arithmetic cases

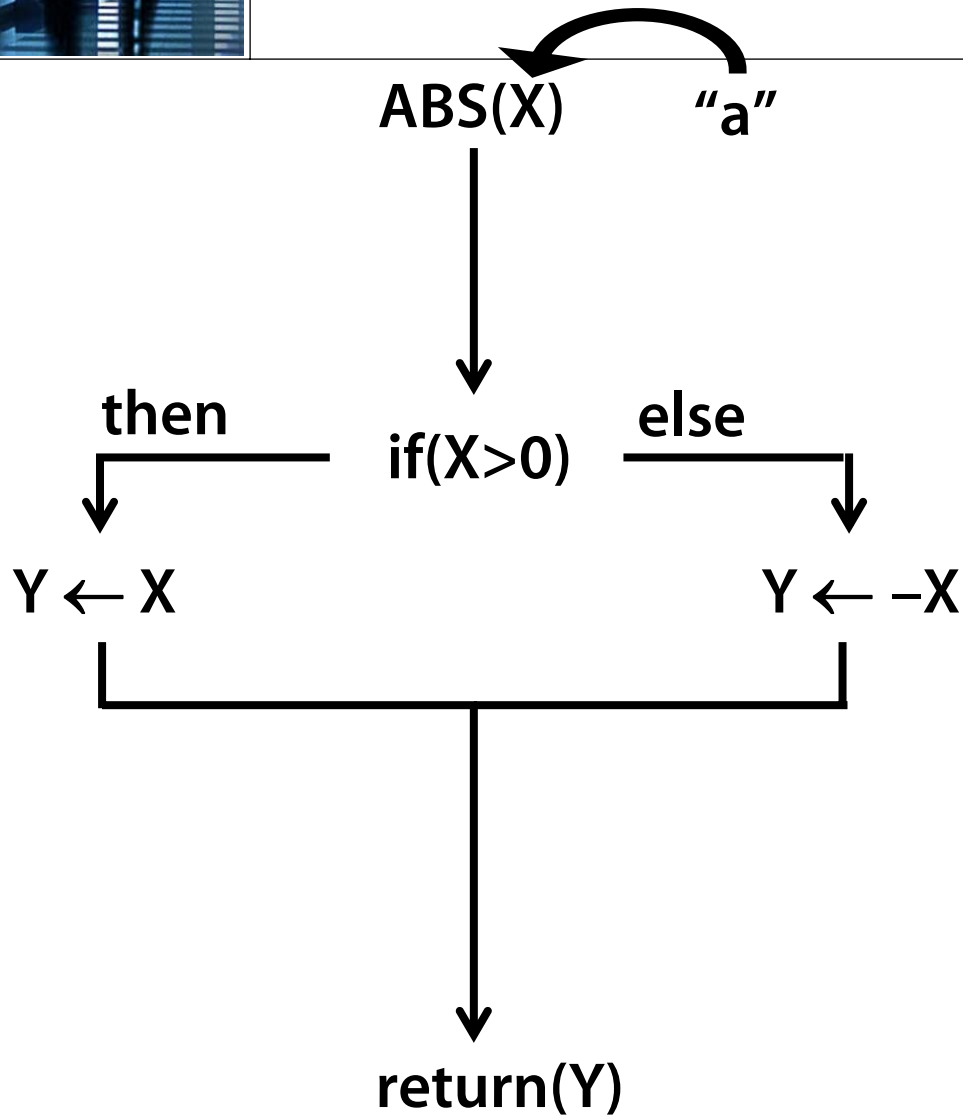


Absolute Value Function





Symbolic Execution of ABS Function



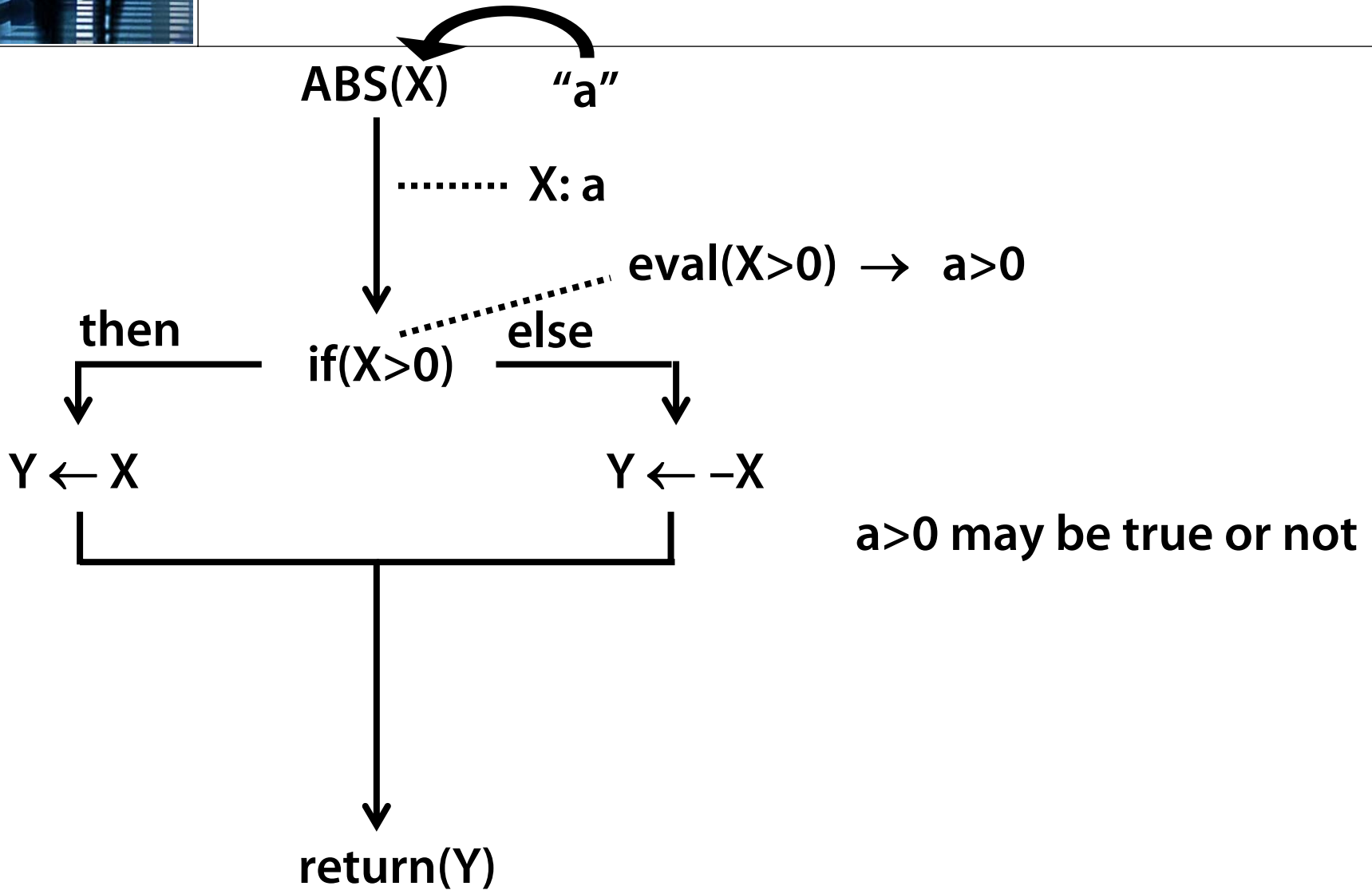
Invoke function
"symbolically"

$R = ABS(a);$

Where "a" is an
unknown but specific
number

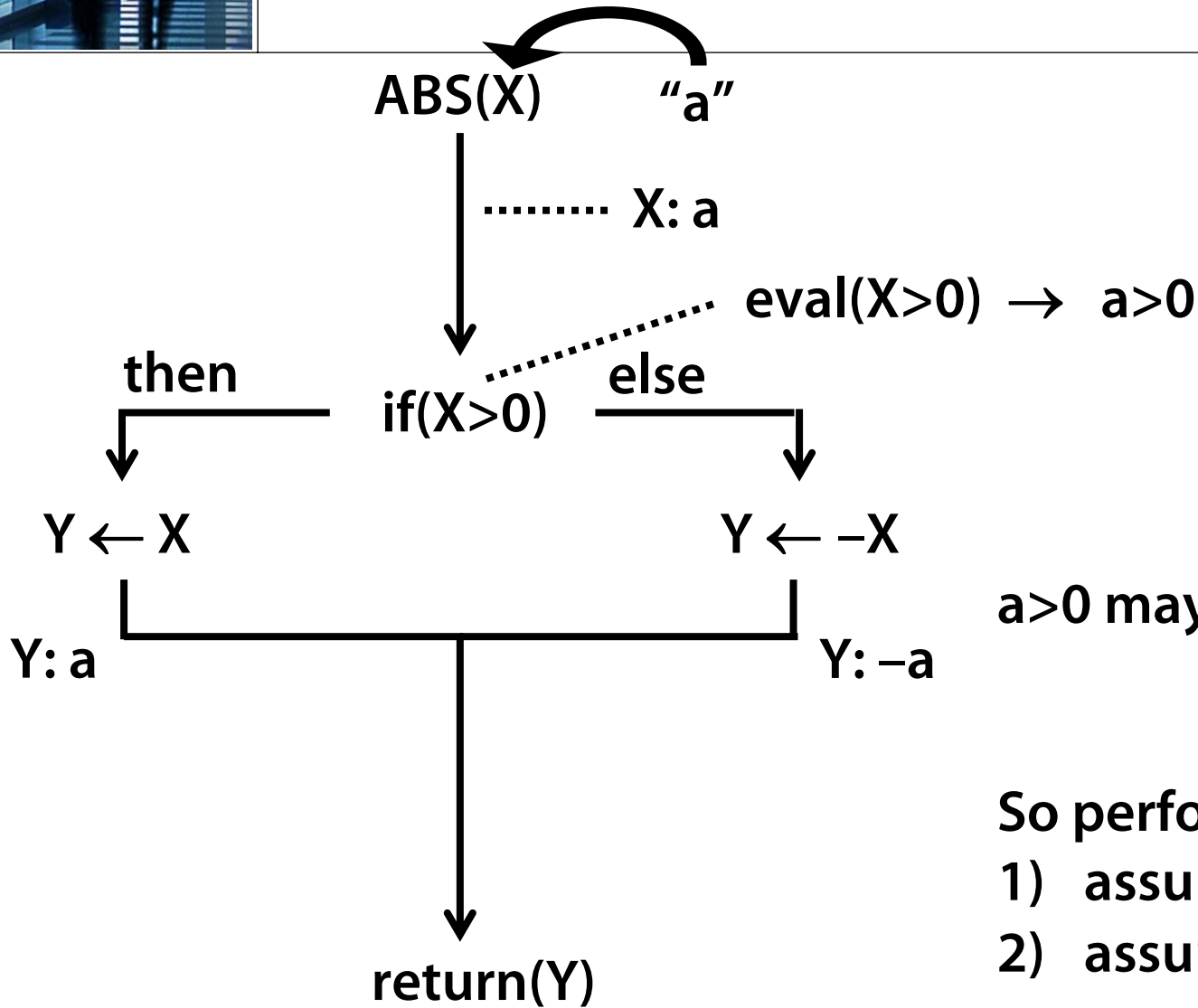


Symbolic Execution of "if" statement





Case Analysis



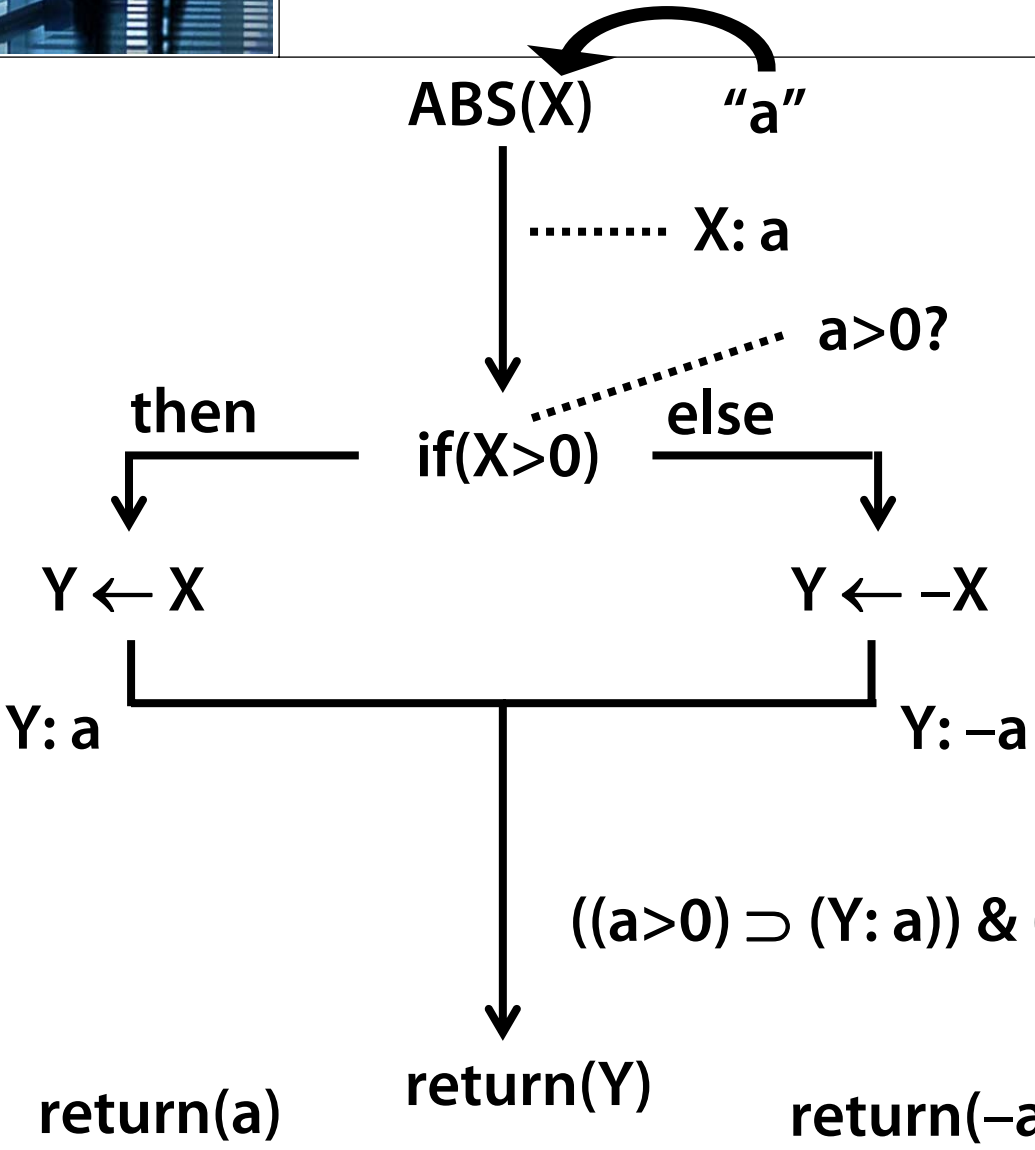
$a > 0$ may be true or not

So perform case analysis:

- 1) assume $a > 0$
- 2) assume $\neg (a > 0)$



Results From Case Analysis (2 cases)



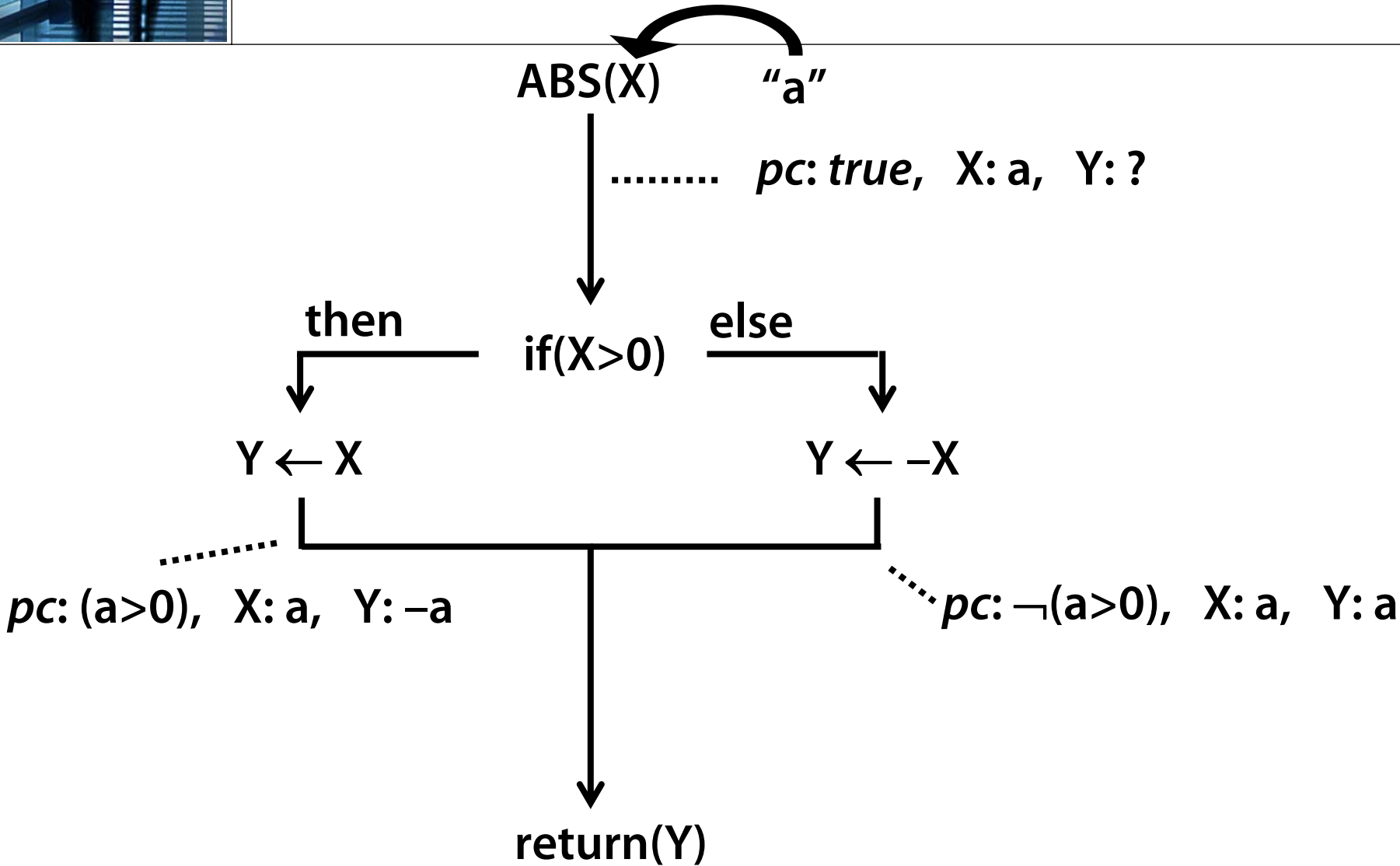
$a > 0$ may be true or not

So perform case analysis:

- 1) assume $a > 0$
- 2) assume $\neg(a > 0)$

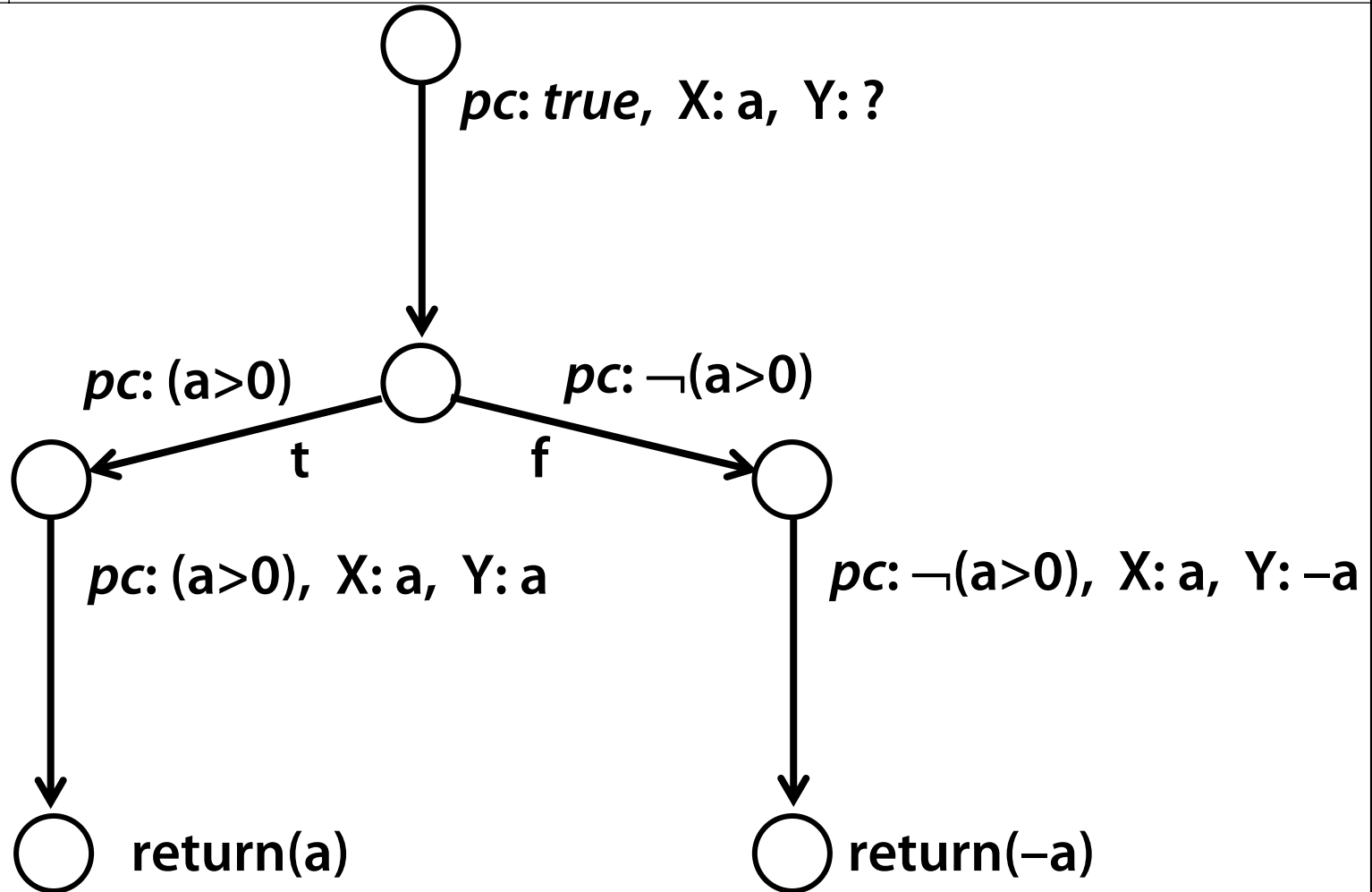


Path Condition (pc)



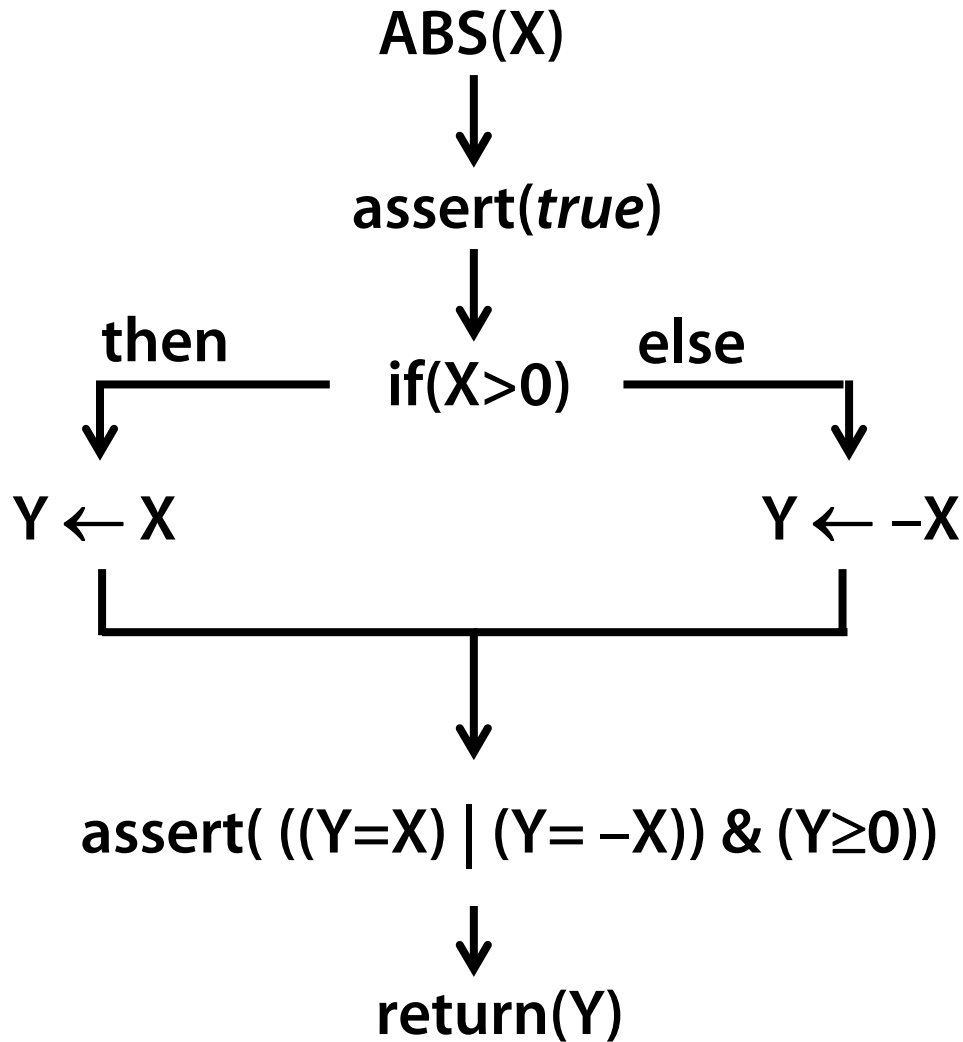


Symbolic Execution Tree



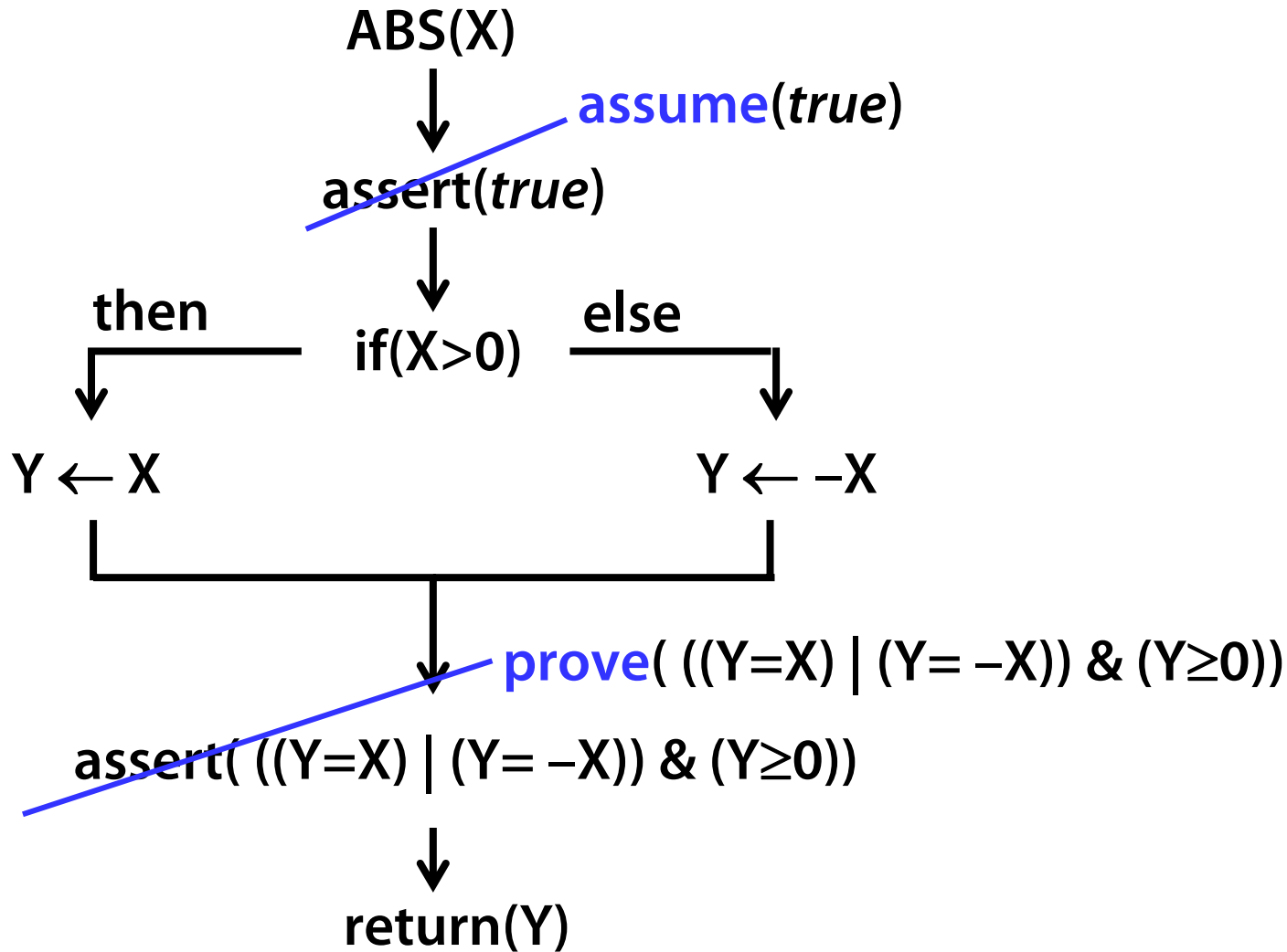


ABS Function with Correctness Assertions





Assertions in Context





Definitions for assume and prove

assume(P): $pc \leftarrow \text{eval}(pc) \ \& \ \text{eval}(P)$

pc maintains those conditions on input symbols that determines this path – “*the path condition*”

prove(P): $\text{eval}(pc) \supset \text{eval}(P)$ (reports: a) always true, b) always false, c) neither)

check the truth of P , given the conditions that hold on *this Path* (defined by *the pc*)

The evaluated results are always only in terms of the inputs or constants introduced in the P 's. No program variables!



Execution of “if” using “prove” and “assume”



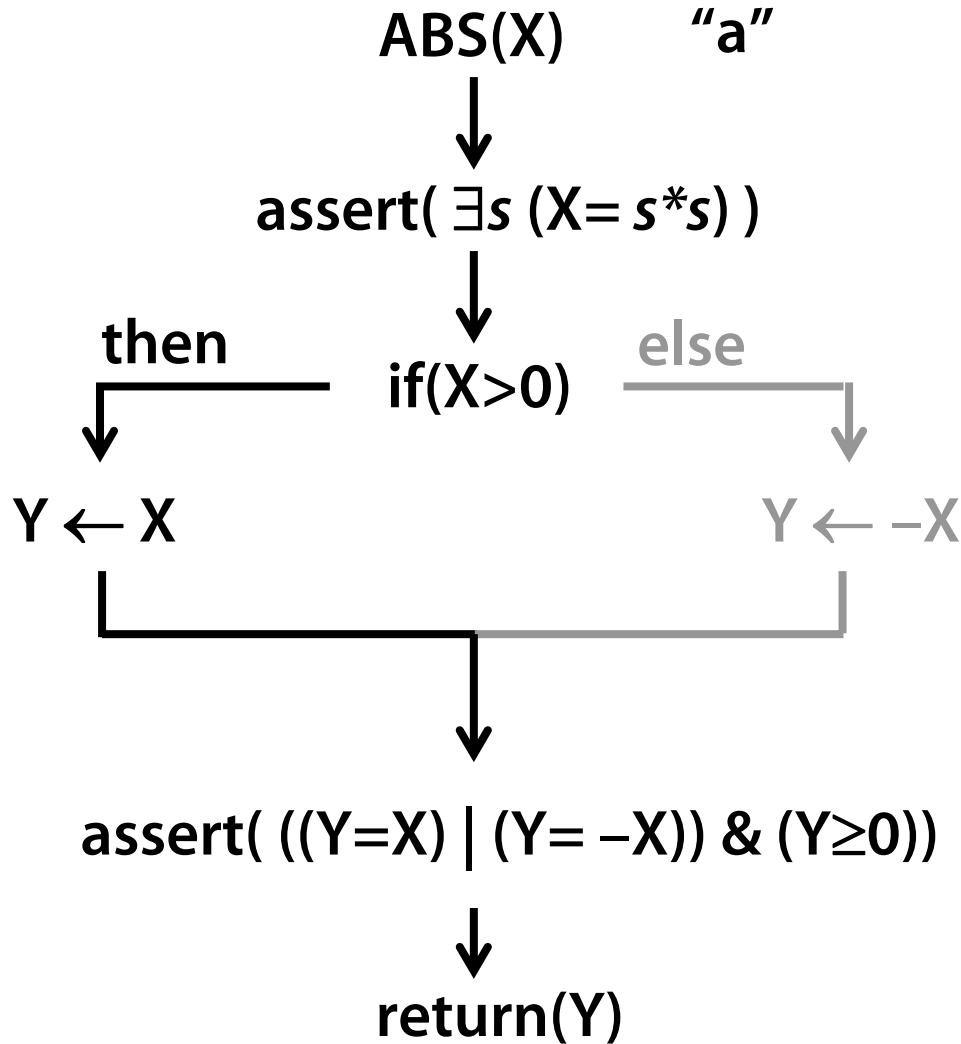
Execution of: $\text{if}(P)$ then ... else ...

1. if $\text{prove}(P)$ is a) always true
then just proceed to then clause
2. if $\text{prove}(P)$ is b) always false
then just proceed to else clause
3. If $\text{Prove}(P)$ is c) neither, then do case analysis:
 - 3.1. $\text{assume}(P)$, take *then* path
 - 3.2. $\text{assume}(\neg P)$, take *else* path

Remember: $\text{assume}(P)$ does $pc \leftarrow \text{eval}(pc) \ \& \ \text{eval}(P)$
 $\text{prove}(P)$ does $\text{eval}(pc) \supset \text{eval}(P)$



Symbolic Execution With Restricted Input Assertion



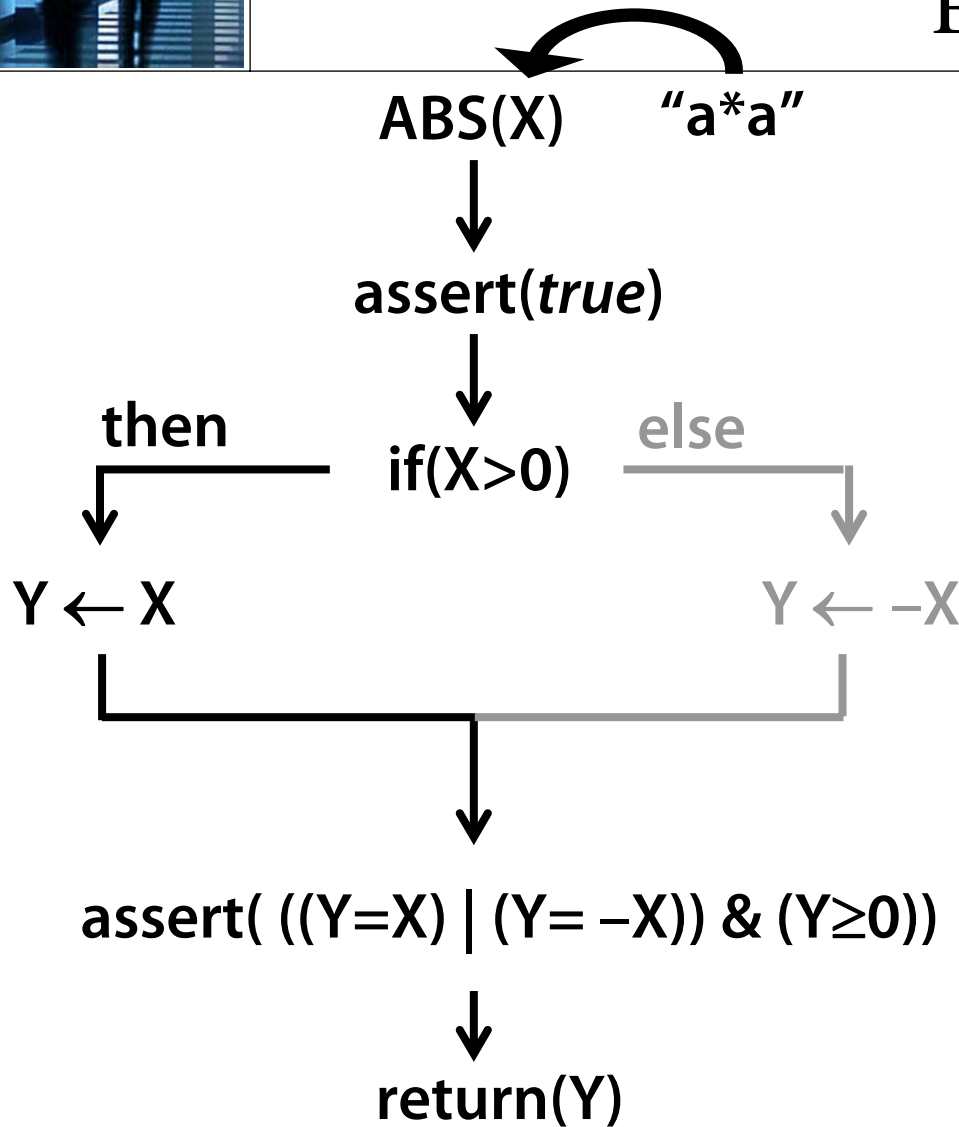
$\vdash \exists s (a = s*s) \supset a > 0$

So no case analysis
is needed.

returns (a)



Symbolic Execution With Expression as Input



$\vdash (a*a) > 0$

So no case analysis is needed.

returns $(a*a)$



Complete Program Verification



ABS(X)

"a"

..... $pc: true, X: a, Y: ?$

Assert(true)

..... $pc: true, X: a, Y: ?$

then

if(X>0)

else

Y ← X

Y ← -X

.....
 $pc: (a>0), X: a, Y: a$

.....
 $pc: \neg(a>0), X: a, Y: -a$

Assert(((Y=X) | (Y= -X)) & (Y≥0))

$(a>0) \supset$

.....
 $((a=a) | (a= -a)) \& (a\geq 0)$

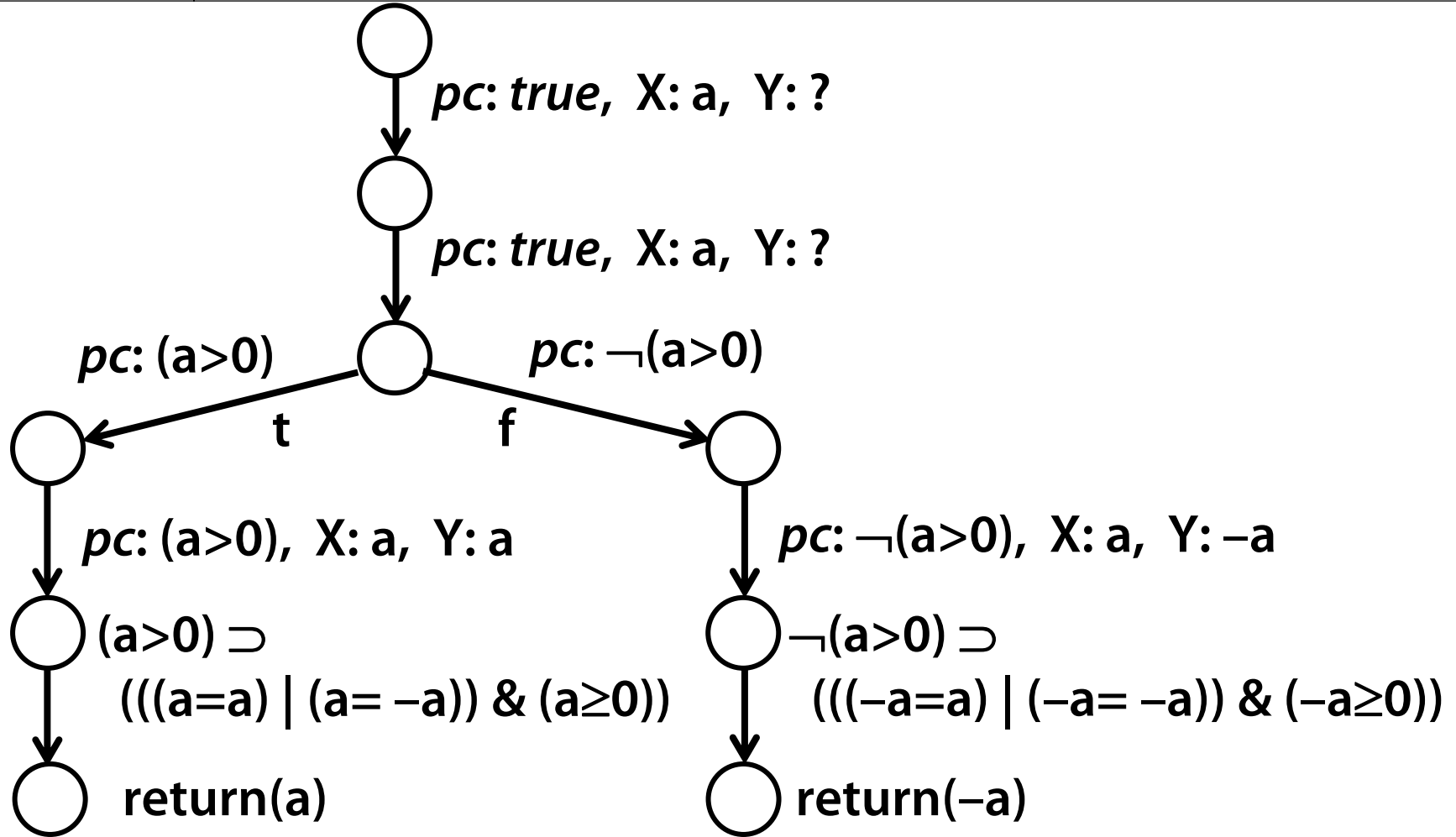
.....
 $\neg(a>0) \supset$

.....
 $((-a=a) | (-a= -a)) \& (-a\geq 0)$

Return(Y)

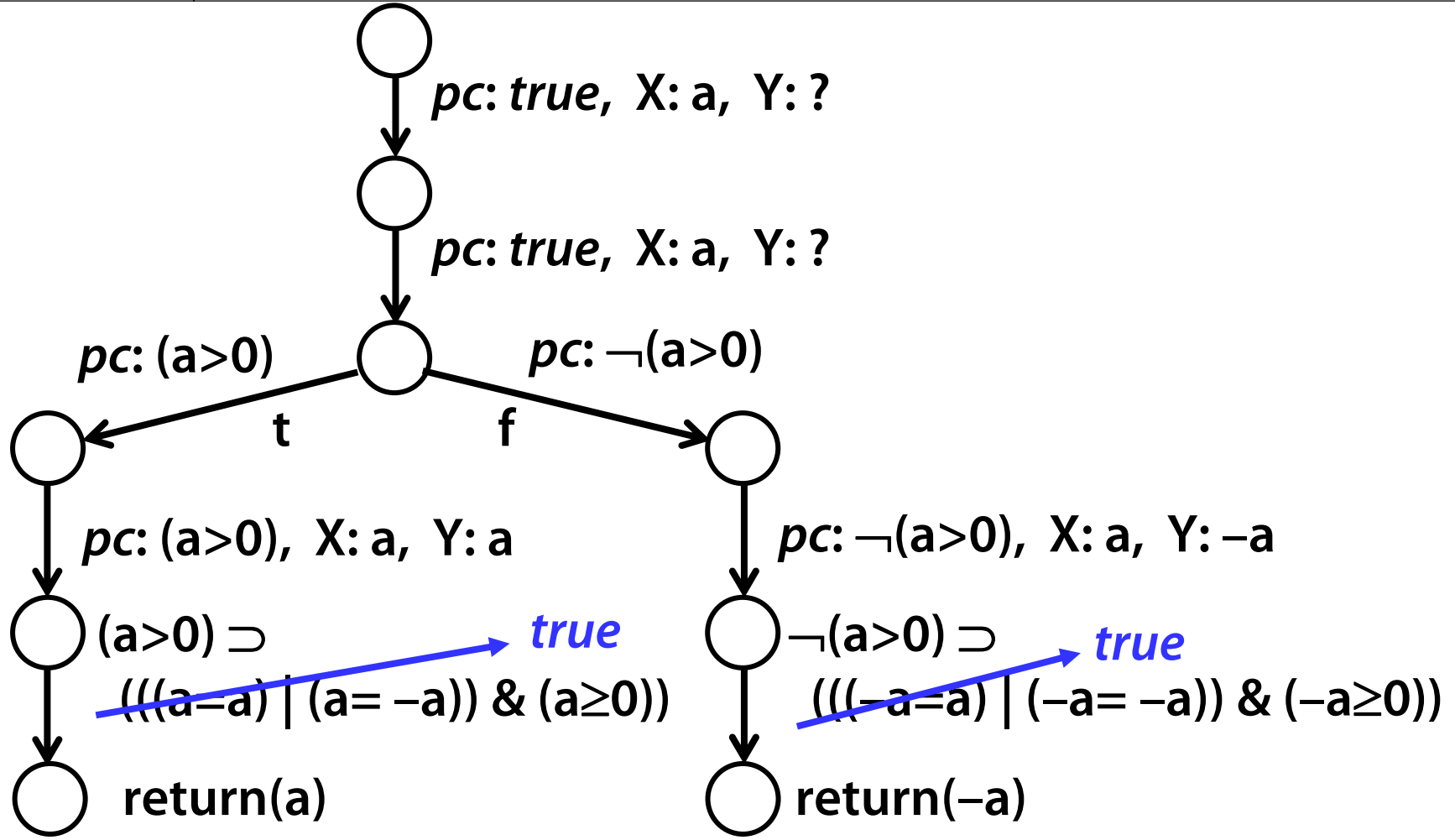


Symbolic Execution Tree for Verification





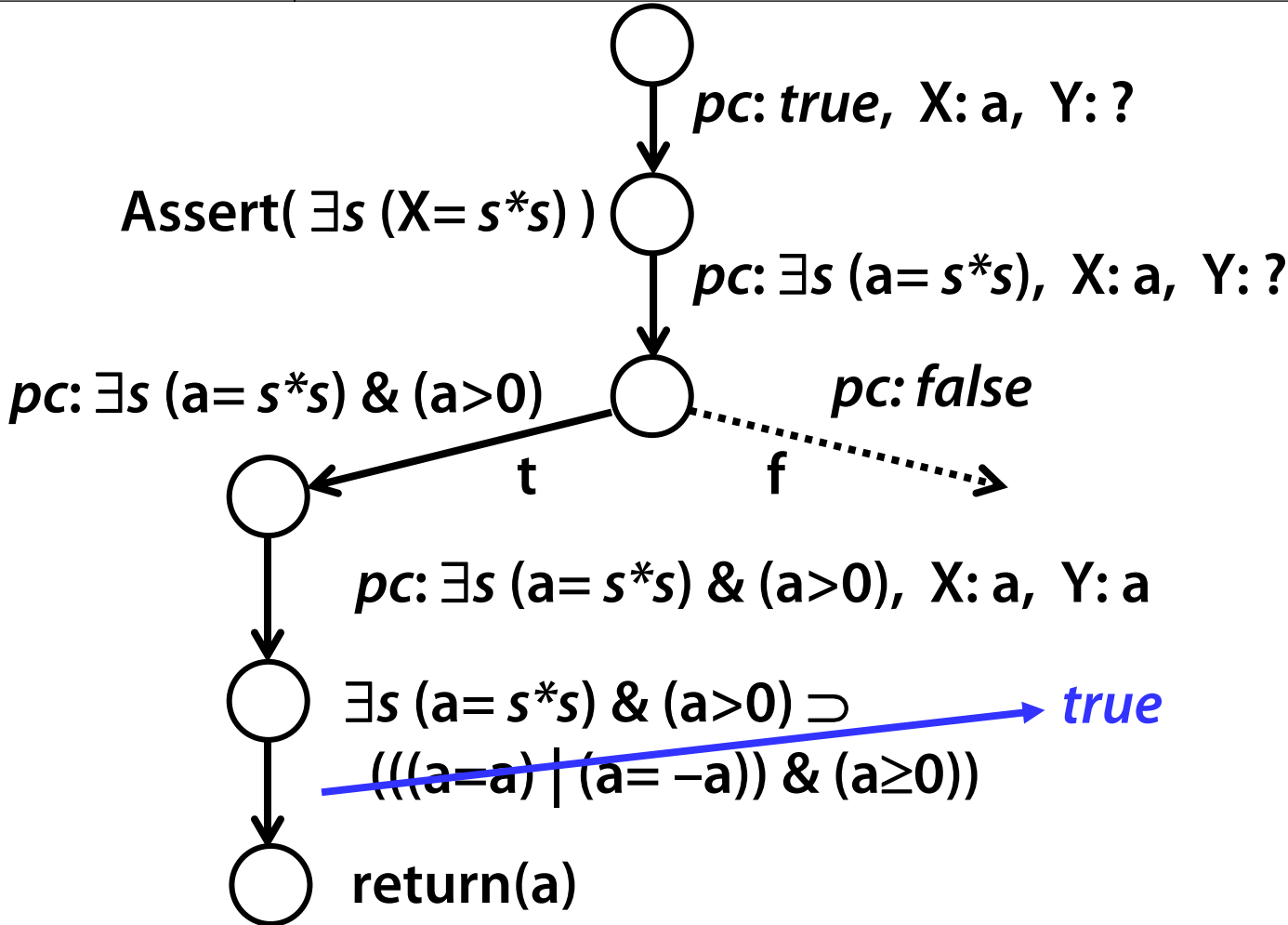
Verified Symbolic Execution Tree



Program with those Assertions is "verified"



Specialized Verification



Program with those Assertions is "verified"



Infinite Symbolic Execution Trees

Programs with Loops



A GCD Example (with loop)

```
int GCD(int M, int N)
{ int A, B;
  assert(M>0 & N>0);
  A ← M;
  B ← N;
  while (A ≠ B)
  { if (A>B) { A ← A - B; }
    else { B ← B - A; }
  }
  assert ( A = (M, N) );
  return (A);
}
```



Axioms Defining GCD

**Notation: (a, b) means
the greatest common divisor of a and b**

$$(a, a) = a, \text{ if } a > 0$$

$$(a, b) = (b, a)$$

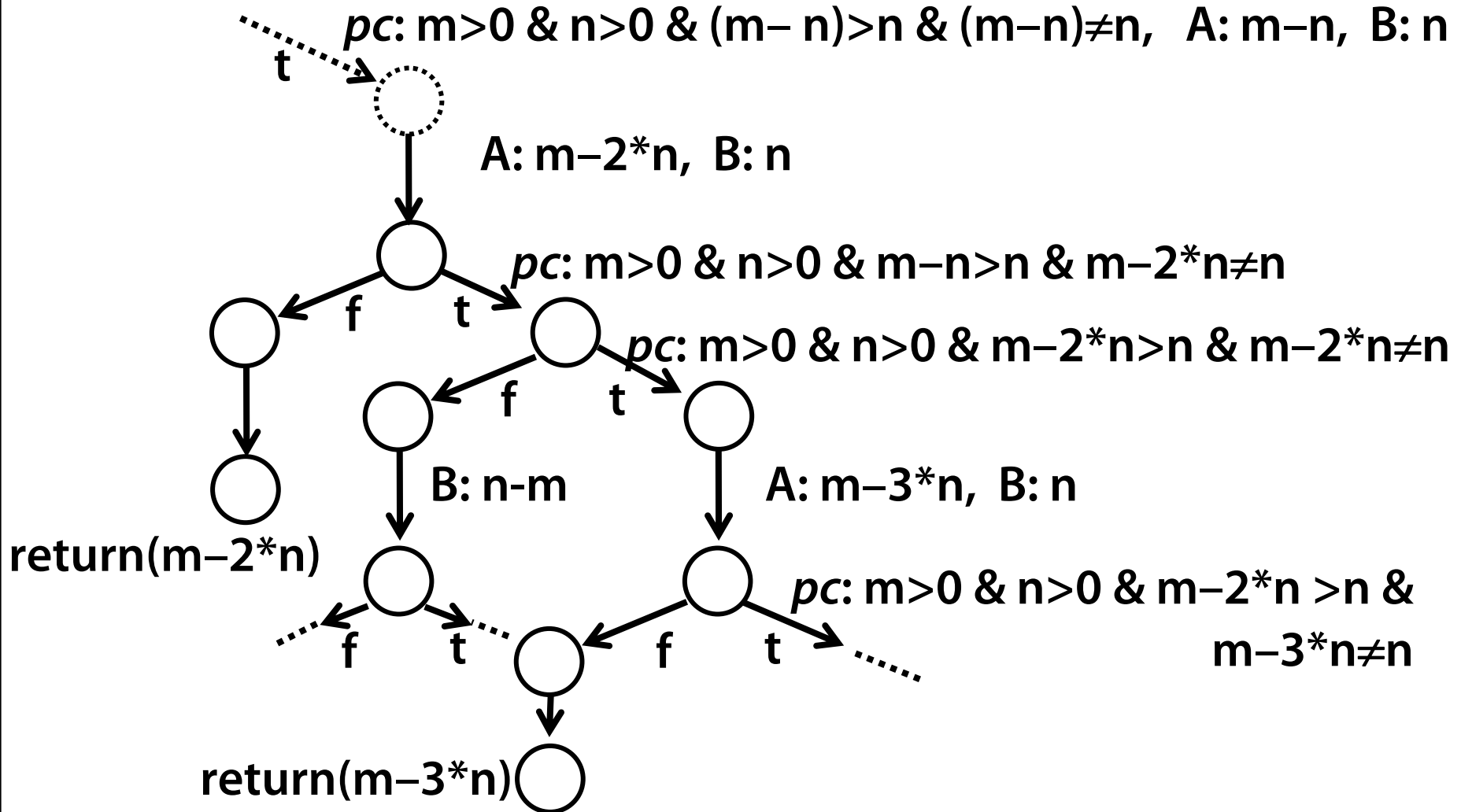
$$(a, b) = (a+b, b)$$

Claim: these three axioms define GCD

Corollary: $(a, b) = (a-b, b)$

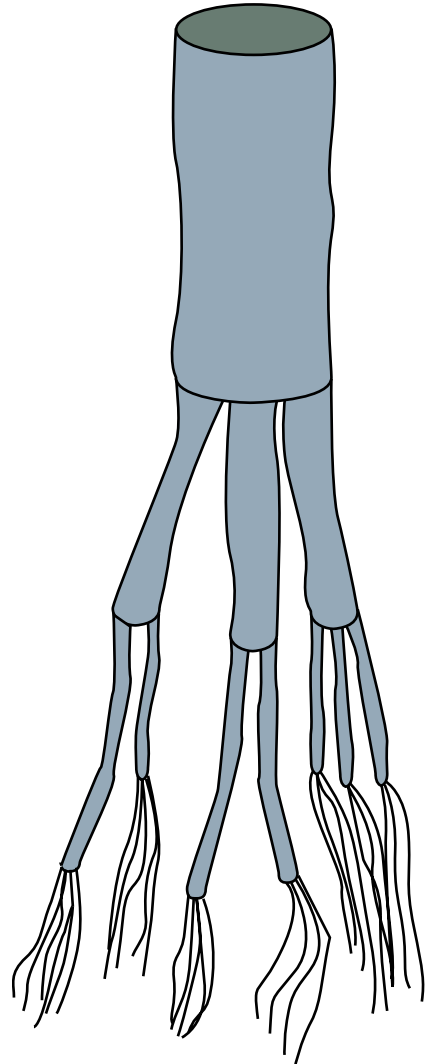


Symbolic Execution Tree (infinite)





Cable Harness





Proving Correctness of Programs

Using Inductive Assertions and Induction over finite subtree proofs



Proof Over Infinite Tree

- **Need induction**
- **“cut” every loop in program with an assert()**
- **Loop causes repeated pattern in symbolic execution tree**



GCD with Inductive Assert

```
int GCD(int M, int N)
```

```
{ int A, B;
```

```
cut1 ..... assert(M>0 & N>0);
```

```
  A ← M;
```

```
  B ← N;
```

```
  while (A ≠ B)
```

```
cut2 ..... { assert( (A, B) = (M,N) & A ≠ B);
```

```
              if (A>B) { A ← A - B; }
```

```
              else { B ← B - A; }
```

```
            }
```

```
return ..... assert ( A = (M, N) );
```

```
  return (A);
```

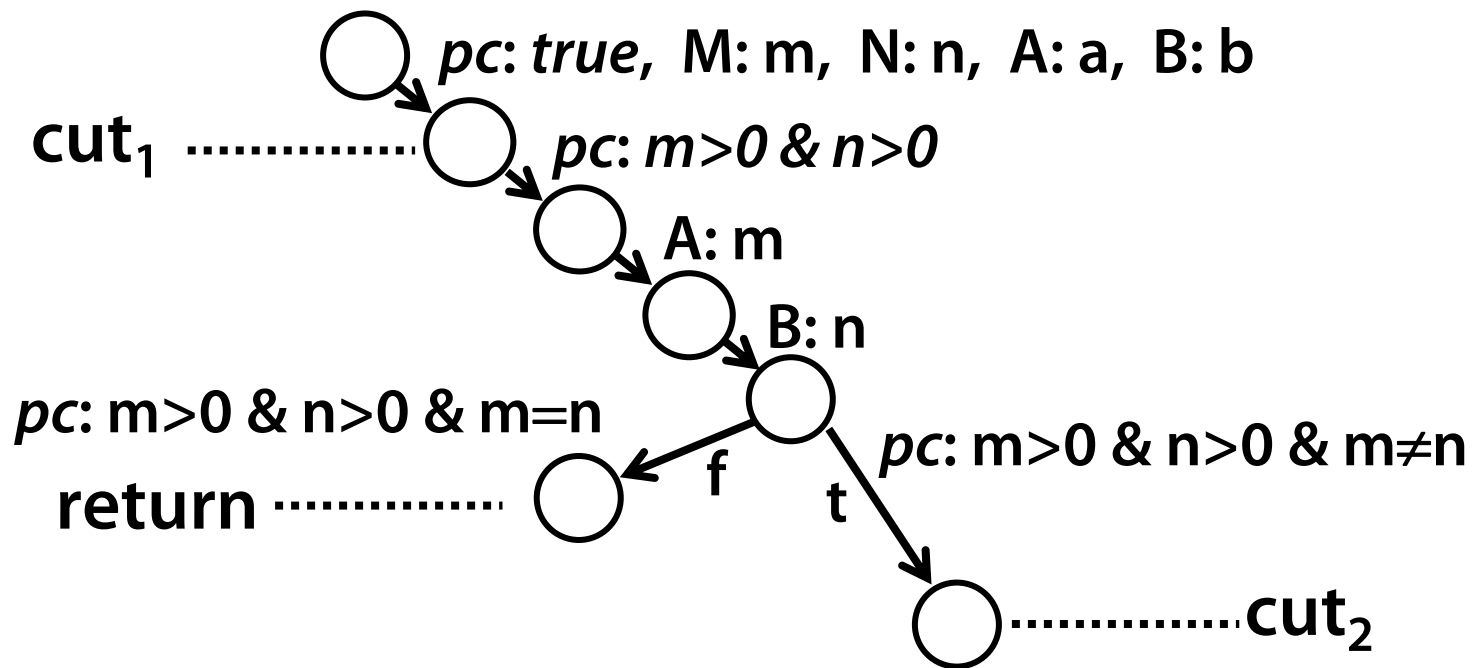
```
}
```



Cut (Symbolic Execution) Trees

From cut_1

- Initialize pc to *true* and all variables to new values
- Execute until another cut encountered

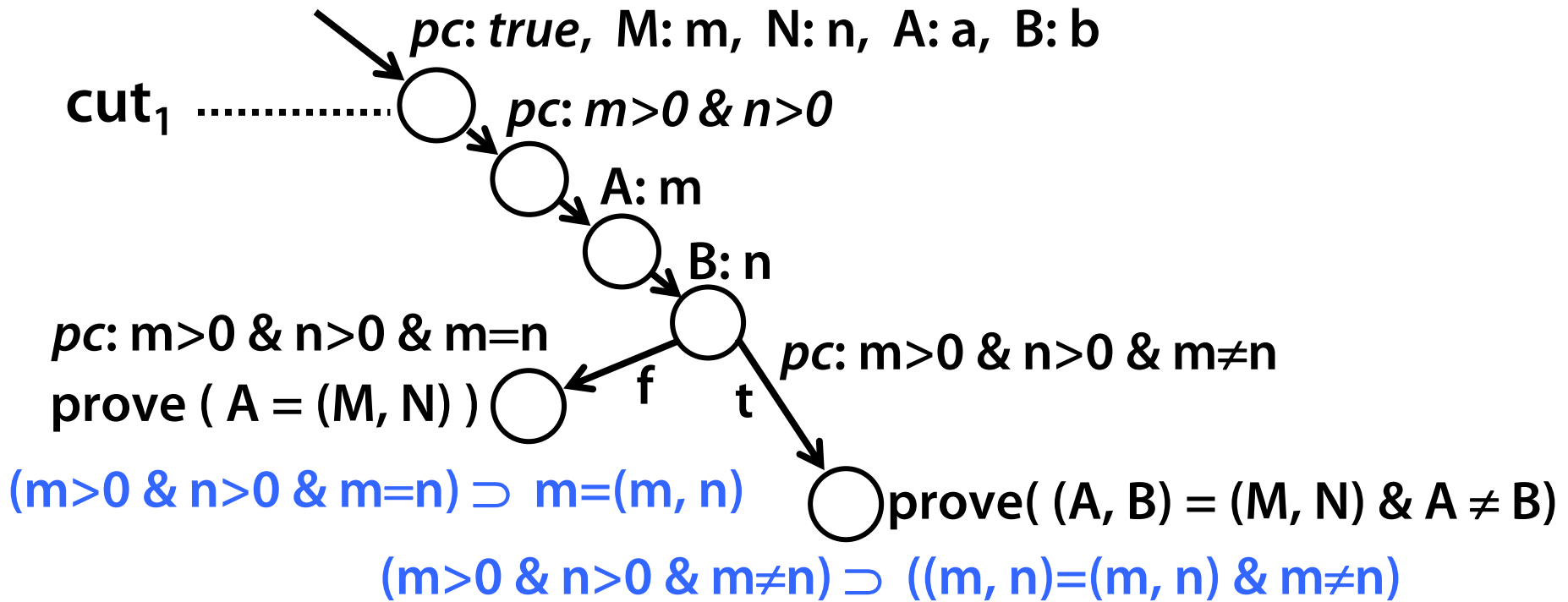




Cut (Symbolic Execution) Trees

From cut_1

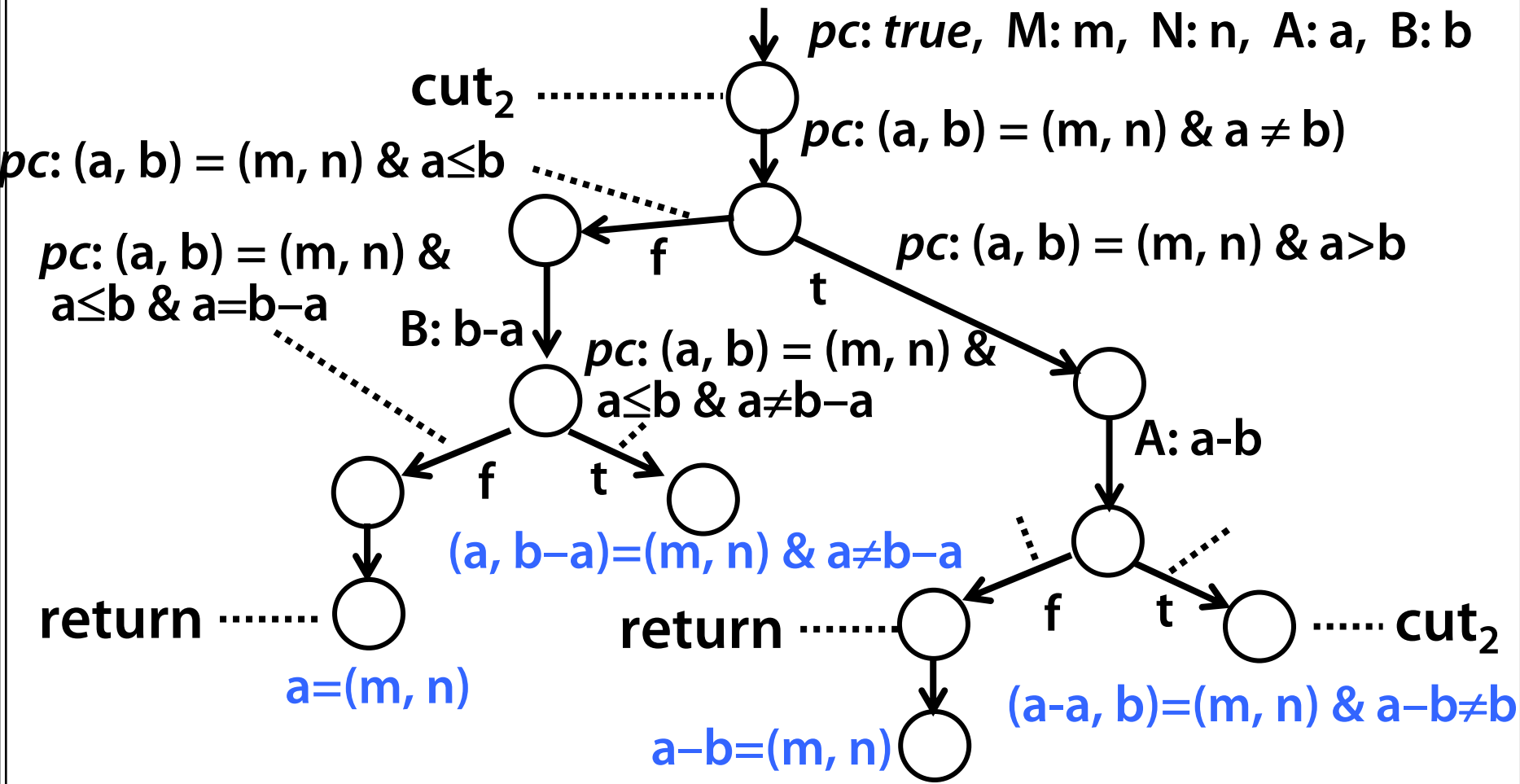
- Treat asserts encountered as “prove”



Two “verification conditions” are true



Cut (Symbolic Execution) Trees: From cut_2



Four "verification conditions" are true



Inductive Proof

- Two initial cut_1 paths verified
- Four cut_2 paths verified

- All executions are combinations of those paths
- Each path was proved under arbitrary conditions
 - pc initialized to *true*
 - All variables set to new unique symbols

Handling Subroutines and Functions

Abbreviated Procedures





Function Calls

```
int GCD(int M, int N)
{ int A, B, D;
  assert(M>0 & N>0);
  A ← M;
  B ← N;
  while (A ≠ B)
  { assert( (A, B) = (M,N) & A ≠ B);
    D ← ABS(A-B);
    if (A>B) { A ← D; }
    else { B ← D; }
  }
  assert ( A = (M, N) );
  return (A);
}
```



Use Assertions Rather Than Code



- **Like a lemma for a proof**
- **If function or subroutine has a loop**
 - Would need the inductive assertion to hold for caller
 - Different for each usage

- **Verified function**
 - Specification (asserts) and Code are consistent
 - Strongly specified characterizes what the routine does



Normal Procedure

```
int R ABS(const int X)
{ int Y;
  assert(true);
  if (X>0)
  { Y ← X;} else
  { Y ← -X;}
  assert( ((Y=X) | (Y= -X)) &
    (Y≥0) );
  return(Y);
}
```

Abbreviated Procedure

```
int R ABS(const int X)
{ int Y;
  prove(true);
  Y ← newsymbol;

  assume( ((Y=X) | (Y= -X)) &
    (Y≥0) );
  return(Y);
}
```



Execution into Abbreviate Procedure

$pc: (a, b) = (m, n) \ \& \ a \neq b, \ A: a, \ B: b, \ M: m, \ N: n$

Call ABS:

$prove(true)$

$(a, b) = (m, n) \ \& \ a > b \supset true$

$pc: (a, b) = (m, n) \ \& \ a \neq b, \ A: a, \ B: b, \ M: m, \ N: n, \ X: a - b$

$Y \leftarrow y; \ (newsymbol)$

$assume((y=a-b \mid y= b-a) \ \& \ y \geq 0)$

$\leftarrow pc: (a, b) = (m, n) \ \& \ a \neq b \ \& \ (y=a-b \mid y= b-a) \ \& \ y \geq 0, \ Y: y$
 $return(y);$

$\leftarrow pc: (a, b) = (m, n) \ \& \ a \neq b \ \& \ (y=a-b \mid y= b-a) \ \& \ y \geq 0, \ D: y$

$if \ pc: (a, b) = (m, n) \ \& \ a > b \ \& \ (y=a-b \mid y= b-a) \ \& \ y \geq 0, \ D: y$

$\leftarrow pc: (a, b) = (m, n) \ \& \ a > b \ \& \ (y=a-b \mid y= b-a) \ \& \ y \geq 0, \ D: y, \ A: y$

Arrays





Arrays

pc: true, A: (a₀, a₁, a₂, ...,), S: s, I: i, J: j

A[I] ← S/3;

A: if $\xi=i$ then $s/3$ else a_ξ

*B ← 2*A[J+1];*

B: 2(if $j+1=i$ then $s/3$ else a_j)*



What Makes It Work

- **Basic symbolic manipulation**
- **assume() and prove()**
 - Used for ifs, testing and proofs
- **Symbolic execution trees**
- **Inductive assertions to handle infinite trees**
- **Abbreviated procedures for lemma ability**
 - *newsymbol*
- **Issues with symbolic references, e.g., arrays**



Adobe

Tools for the New Work™