

Software Reliability

Lecture 2

Static Program Verification

Alastair Donaldson

www.doc.ic.ac.uk/~afd

Pre- and post-conditions

Pre-condition: fact that must hold on method entry – a pre-condition is **required** by the method

Post-condition: fact that must hold on method return – a correctly implemented method **ensures** its post-condition

Pre- and post-condition for method: collectively called **specification** or **contract** for the method

Correctness

Correctness with respect to pre- and post-conditions and assertions:

A method (function/procedure) with pre-condition **P** and post-condition **Q** is **correct** if every execution starting in a state which satisfies **P**

- does not violate any assertions, and
- either:
 - does not terminate, or
 - terminates in a state which satisfies **Q**

This is really **partial correctness**: total correctness demands termination

We shall use **correct** to mean **partially correct**

Reminder of a couple of logic essentials

$P \Rightarrow Q$

- **P** implies **Q**
- If **P** holds then **Q** holds
- Many tools use notation **$P \implies Q$**

False implies everything!

- **false $\Rightarrow Q$** is always true
- **false $\Rightarrow (4 == 5)$** holds

True implied by everything

- **$P \Rightarrow \text{true}$** is always true

Logical formulae can denote sets

Suppose program has integer variables x and y

Formula $(x > y)$ can be thought of as denoting the set of all program states where x is bigger than y

More generally, formula R denotes set of all program states where R holds

Method pre-condition P : set of all program states from which the method can be safely executed

Method post-condition Q : set of states that includes all possible end states for the method

Logical formulae can denote sets

Which formula denotes **all** program states?

true

Which formula denotes the empty set?

false

What does \Rightarrow correspond to in set theory?

\subseteq

Aim of Static Program Verification

Given a set of procedures, each with a specification (pre- and post-condition), show that every procedure is correct

Correct means:

If pre-condition holds then

- **no assertions fail**
- **post-condition holds** on procedure return

In Hoare's notation we write:

$$\{\mathbf{P}\} \mathbf{C} \{\mathbf{Q}\}$$

for a procedure with pre-condition **P**, post-condition **Q** and body **C**

Simple C

We'll present static verification using a simple C-like language:

- Only type is (signed) int
- Only simple control flow (if, while)
- Only pure, immediate operators (no ++, +=, no short-circuit evaluation)
- etc.

Allows us to focus on verification techniques without getting bogged down in language details

Full-blown verifiers must (and to some extent do!) deal with complexities such as pointers and function pointers

Static program verification: top-level approach

Turn program \mathbf{P} into a *logical formula* φ such that:

- If φ is unsatisfiable, \mathbf{P} is correct
- If φ is satisfiable, \mathbf{P} may be incorrect

For loop-free programs, we will proceed as follows:

- 1) Turn \mathbf{P} into predicated static single assignment (SSA) form \mathbf{P}'
- 2) Build a formula φ encoding buggy paths through \mathbf{P}'
- 3) Use an **SMT solver** to analyse φ , to prove whether a buggy path exists

SSA form: example

In SSA form, every variable is assigned to once:

```
x = y + 1;  
x = x + 1;  
y = y + 1;  
assert x == y + 1;  
assert x > y;
```

is
expressed
as:

```
x1 = y0 + 1;  
x2 = x1 + 1;  
y1 = y0 + 1;  
assert x2 == y1 + 1;  
assert x2 > y1;
```

For code without conditionals and loops, this SSA renaming process is straightforward:

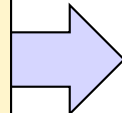
- increment the SSA id of a variable each time it is defined (assigned to)
- select the latest SSA id of a variable each time it is used

SSA renaming clearly preserves program correctness

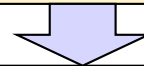
Checking correctness of an SSA program

Correctness conditions for SSA form program can be encoded as a set of constraints:

```
x = y + 1;  
x = x + 1;  
y = y + 1;  
assert x == y + 1;  
assert x > y;
```



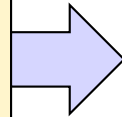
```
x1 = y0 + 1;  
x2 = x1 + 1;  
y1 = y0 + 1;  
assert x2 == y1 + 1;  
assert x2 > y1;
```



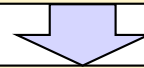
```
(x1 == y0 + 1) && (x2 == x1 + 1) && (y1 == y0 + 1)  
    &&  
    !((x2 == y1 + 1) && (x2 > y1))
```

Checking correctness of an SSA program

```
x = y + 1;  
x = x + 1;  
y = y + 1;  
assert x == y + 1;  
assert x > y;
```



```
x1 = y0 + 1;  
x2 = x1 + 1;  
y1 = y0 + 1;  
assert x2 == y1 + 1;  
assert x2 > y1;
```



```
(x1 == y0 + 1) && (x2 == x1 + 1) && (y1 == y0 + 1)  
    &&  
    !((x2 == y1 + 1) && (x2 > y1))
```

Constraints satisfiable \Leftrightarrow there exist values for x_1, x_2, y_0, y_1 that:

- satisfy the relationships between variables enforced by assignments
- cause at least one assertion to **fail**

P correct \Leftrightarrow constraints are **unsat**

Solving the formula

Automated verification tools rely on a:

theorem prover / constraint solver / SMT solver

to solve formulas

names used pretty
much interchangeably

Formula to be checked is called a **verification condition (VC)** or **proof obligation**

VC (or proof obligation) is **discharged** by a solver

Satisfiability Modulo Theories (SMT) in a slide

An SMT solver can decide whether a formula is satisfiable, where the formula is expressed using one or more **theories**

Common theories

- Integers (a.k.a. mathematical integers)
- Bit vectors (a.k.a. machine integers)
- Reals (and recent floating point support)
- Arrays

Standard input
language: SMT-LIB 2

Common logic + theory combinations

- QF_BV: quantifier-free formulae over bit-vectors
- QF_LIA: quantifier-free linear integer arithmetic formulae (boolean combinations of inequalities between linear polynomials over integer variables)

Successful solvers include Z3, CVC4, MathSAT, Boolector

Annual competition, **SMT-COMP**, drives progress!

Coding our formula in SMT-LIB 2

$(x_1 == y_0 + 1) \ \&\&$
 $(x_2 == x_1 + 1) \ \&\&$
 $(y_1 == y_0 + 1)$

$\&\&$

$! \ ((x_2 == y_1 + 1) \ \&\&$
 $(x_2 > y_1))$

```
(set-logic QF_LIA)
(declare-fun x1 () Int)
(declare-fun x2 () Int)
(declare-fun y0 () Int)
(declare-fun y1 () Int)
  (assert (= x1 (+ y0 1)))
  (assert (= x2 (+ x1 1)))
  (assert (= y1 (+ y0 1)))
  (assert (not (and
    (= x2 (+ y1 1))
    (> x2 y1)
  )))
(check-sat)
```

Note different meaning of assert: we are asserting facts to the solver

Result: unsat

Points from the example

Called **S-expressions**
(from Lisp)

`(set-logic QF_LIA)`

Specify which logic to use (quantifier-free linear integer arithmetic)

`(declare-fun x1 () Int)`

Declare a symbolic constant of type Int: a nullary (0-argument) function

`(assert (= x1 (+ y0 1)))`

Tell the solver a fact

`(check-sat)`

Expressions are written in prefix form (operator then operands)

Tell the solver to check satisfiability

Using bitvectors instead of mathematical integers

$(x_1 == y_0 + 1) \ \&\&$
 $(x_2 == x_1 + 1) \ \&\&$
 $(y_1 == y_0 + 1)$

$\&\&$

$!((x_2 == y_1 + 1) \ \&\&$
 $(x_2 > y_1))$

```
(set-logic QF_BV)
```

```
(declare-fun x1 () (_ BitVec 32))  
(declare-fun x2 () (_ BitVec 32))  
(declare-fun y0 () (_ BitVec 32))  
(declare-fun y1 () (_ BitVec 32))
```

```
(assert (= x1 (bvadd y0 (_ bv1 32))))  
(assert (= x2 (bvadd x1 (_ bv1 32))))  
(assert (= y1 (bvadd y0 (_ bv1 32))))
```

```
(assert (not (and  
  (= x2 (bvadd y1 (_ bv1 32)))  
  (bvsgt x2 y1)  
)))
```

```
(check-sat)
```

Result: sat

SMT type for n -bit
bitvector:

```
(_ BitVec  $n$ )
```

SMT syntax for m
as n -bit bitvector:

```
(_ bv $m$   $n$ )
```

Getting a value from the solver

If solver says **sat**, we can ask the solver for values for individual variables. E.g., if we ask:

```
(get-value (y0))
```

the solver says:

```
((y0 #x7fffffffef))
```

Think why the program is incorrect for this value of y_0

Try Z3

Z3 is packaged with the given files for Part 1 of the coursework

To experiment with SMT-LIB 2, do:

```
z3 -smt2 -file query.txt
```

Our story so far, for programs without conditionals

Turn program into **SSA form**. Program then consists of a mixture of:

- Assignments: $v_1 = d_1, v_2 = d_2, \dots, v_m = d_m$
- Assertions: `assert e_1` , `assert e_2` , ..., `assert e_n`

Program is correct if and only if this formula is **unsatisfiable**:

$$(v_1 == d_1 \ \&\& \ v_2 == d_2 \ \&\& \ \dots \ \&\& \ v_m == d_m) \\ \&\& \\ \!(e_1 \ \&\& \ e_2 \ \&\& \ \dots \ \&\& \ e_n)$$

We can use an **SMT solver** to check this

Next: handling conditionals

SSA form for conditionals: example 1

```
x = y;
if(x > z) {
  x = x + 1;
  y = y + 1;
} else {
  x = x + y;
}
```

→

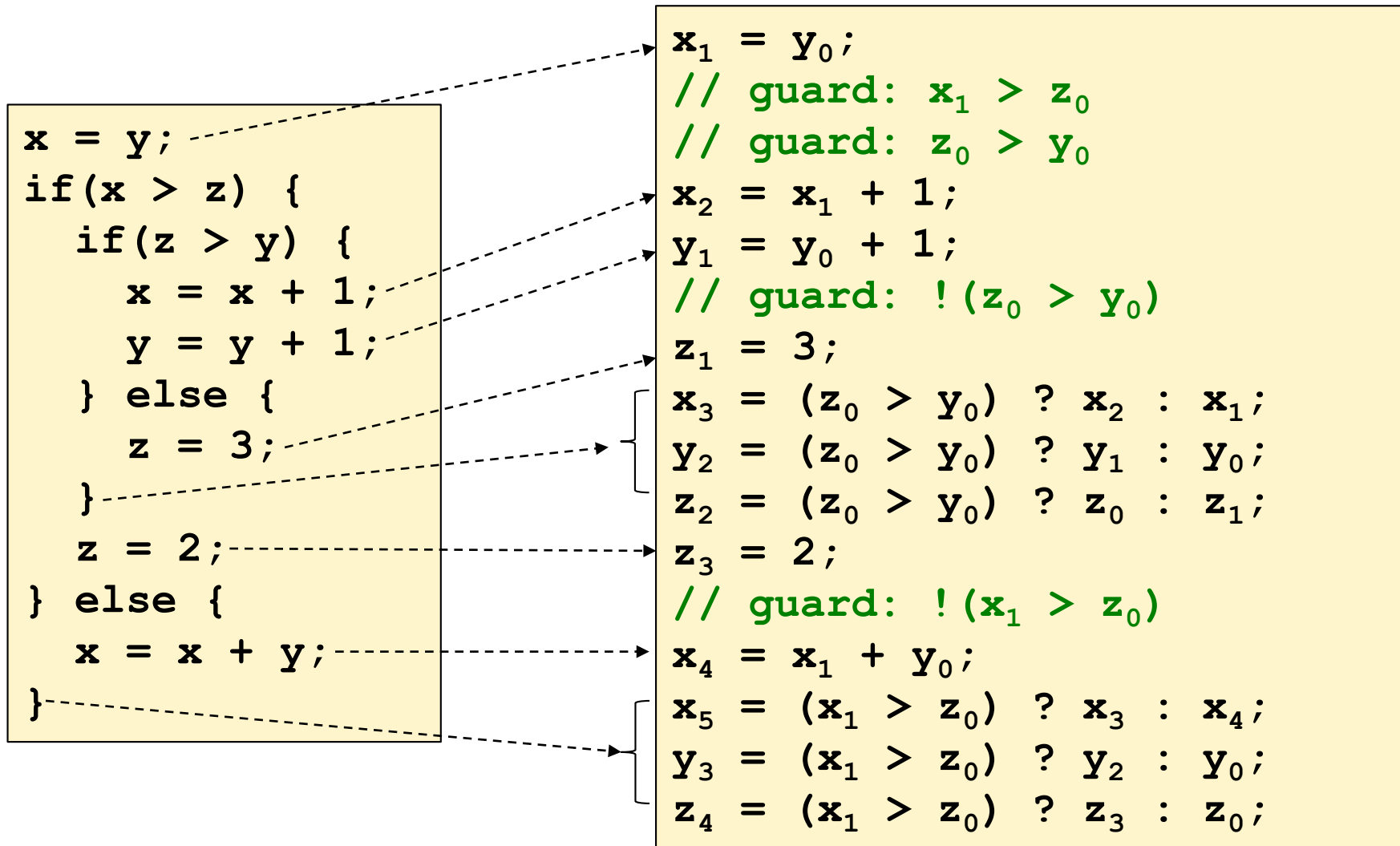
```
x1 = y0;
// guard: x1 > z0
x2 = x1 + 1;
y1 = y0 + 1;
// guard: !(x1 > z0)
x3 = x1 + y0; // reads values of x and
                  // before conditional
x4 = (x1 > z0) ? x2 : x3;
y2 = (x1 > z0) ? y1 : y0;
```

Method:

- turn **then** and **else** branches into SSA *separately*
- use different IDs for new variables
- **resolve branches** after conditional: updated variables take values depending on the conditional guard

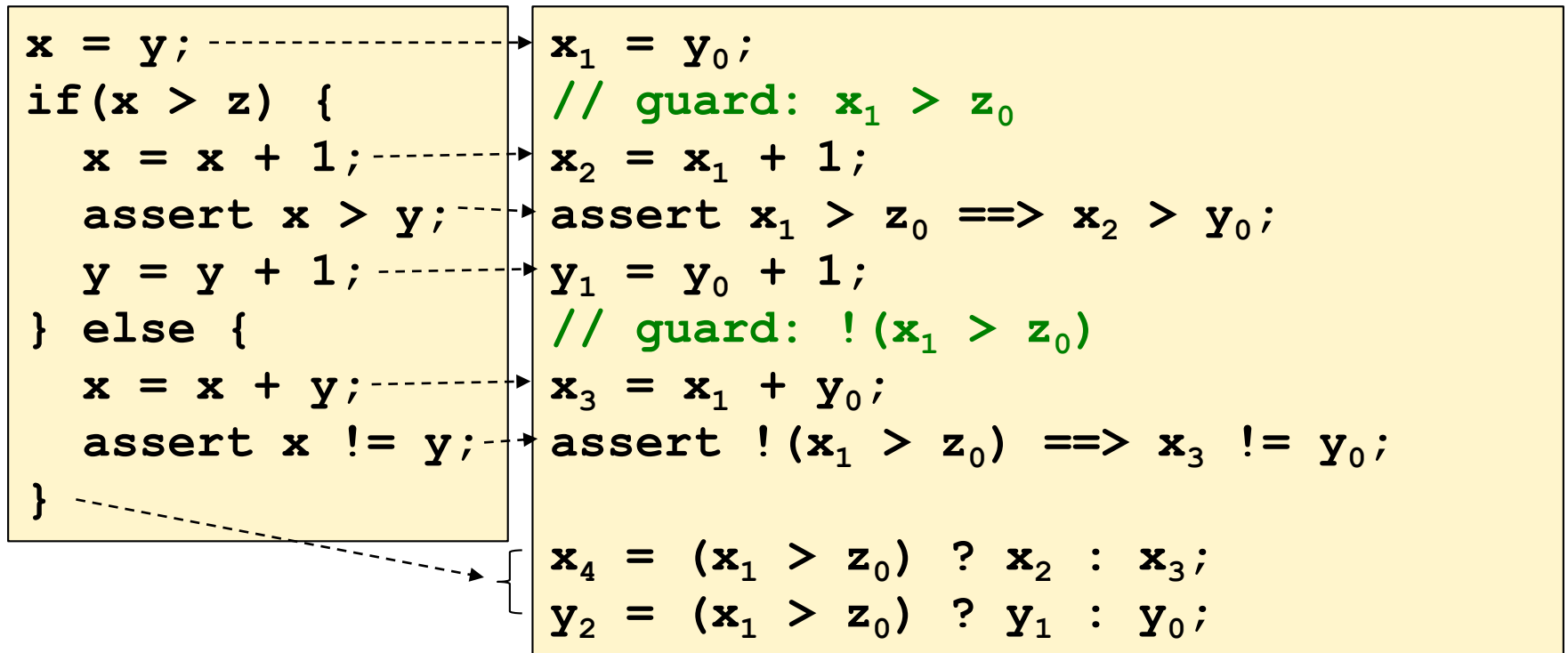
SSA form for conditionals: example 2

Nested conditionals: need to resolve branches multiple times



SSA form for conditionals: example 3

Assert statements must be **predicated** by guards of all enclosing conditional branches



Next: informed by these examples, we'll see an algorithm for SSA conversion

SSA conversion algorithm: notation

Let M be a mapping from variables to SSA ids

Let $M(v)$ denote the SSA id to which v is mapped

For an expression E , let **apply**(E, M) be the expression identical to E , but with each variable v replaced with $v_{M(v)}$

Example: suppose $M = \{ \mathbf{x} \mapsto 2, \mathbf{y} \mapsto 3, \mathbf{z} \mapsto 4 \}$

Then:

$$M(\mathbf{x}) = 2, M(\mathbf{y}) = 3, M(\mathbf{z}) = 4$$

$$\mathbf{apply}(\mathbf{x+y} / (\mathbf{x+z}), M) = \mathbf{x}_2 + \mathbf{y}_3 / (\mathbf{x}_2 + \mathbf{z}_4)$$

We write:

$$M(v) := id;$$

to update the mapping for v to id

SSA conversion algorithm: notation

Procedure **fresh**(v) returns an SSA id for a variable. The same id is never returned for the same variable twice

If M is an SSA mapping, M .**clone**() returns a duplicate of M

modset(S) returns variables that are possibly modified by statement S :

- **modset**($v = E$) = $\{ v \}$
- **modset**(**assert** E) = $\{ \}$
- **modset**($S; T$) = **modset**(S) \cup **modset**(T)
- **modset**(**if** (E) { S } **else** { T }) = **modset**(S) \cup **modset**(T)

SSA conversion algorithm

We will describe the algorithm as a recursive procedure:

toSSA(*Stmt*, *Pred*, *M*)

where:

- *Stmt* is a program statement
- *Pred* is a Boolean predicate
- *M* is an SSA mapping, and is **passed by reference**

Top-level statement *S* is converted by executing:

toSSA(*S*, true, *init*)

where *init* maps each variable to SSA id 0.

Code is generated by procedure **emit**(*s*), where *s* is a string

SSA conversion algorithm

```
toSSA( $v = E$ ,  $Pred$ ,  $M$ ) {  
     $newId := \mathbf{fresh}(v)$ ;  
    emit(" $v_{newId} = \mathbf{apply}(E, M)$  ; ");  
     $M(v) := newId$ ;  
}  
  
toSSA(assert  $E$ ,  $Pred$ ,  $M$ ) {  
    emit("assert  $Pred ==> \mathbf{apply}(E, M)$  ; ");  
}  
  
toSSA( $S$ ;  $T$ ,  $Pred$ ,  $M$ ) {  
    toSSA( $S$ ,  $Pred$ ,  $M$ ); // recall that  $M$  is passed  
    toSSA( $T$ ,  $Pred$ ,  $M$ ); // by reference  
}
```

SSA conversion algorithm

```
toSSA( if (E) { S } else { T } , Pred, M) {  
    NewPred := apply(E, M);  
    M' := M.clone();  
    M'' := M.clone();  
    toSSA(S, Pred && NewPred, M');  
    toSSA(T, Pred && ! (NewPred) , M''); // omit if else  
                                           // branch is empty  
    for(v : modset(S) ∪ modset(T)) {  
        M(v) := fresh(v);  
        emit(" vM(v) = NewPred ? vM'(v) : vM''(v) ");  
    }  
}
```

A simple example

```
int getXOrZero(int x)
  requires x != 5,
  ensures \result >= 0,
  ensures \result != 5
{
  int z;
  if(x < 0) {
    z = 0;
  } else {
    assert(z != -1);
    z = x;
  }
  return z;
}
```

Try turning this program
into SSA form using
toSSA

For purposes of
verification, equivalent to:

```
// Initially, values of
// x, y z are arbitrary
if(x != 5) {
  if(x < 0) {
    z = 0;
  } else {
    assert(z != -1);
    z = x;
  }
  assert z >= 0,
  assert z != 5;
}
```

Expected result

Assuming that `fresh(v)` has the effect of incrementing SSA ids, we end up with:

```
z1 = 0;  
assert(x0 != 5 && !(x0 < 0) ==> z0 != -1);  
z2 = x0;  
z3 = x0 < 0 ? z1 : z2;  
assert(x0 != 5 ==> z3 >= 0);  
assert(x0 != 5 ==> z3 != 5);  
z4 = x0 != 5 ? z3 : z0;
```

We can turn this into a formula

```
(z1 == 0) && (z2 == x0) &&  
(z3 == x0 < 0 ? z1 : z2) && (z4 == x0 != 5 ? z3 : z0)  
  &&  
!( (x0 != 5 && !(x0 < 0) ==> z0 != -1) &&  
  (x0 != 5 ==> z3 >= 0) &&  
  (x0 != 5 ==> z3 != 5) )
```

Expected result

In SMT, with bitvectors, the formula translates to:

```
(set-logic QF_BV)

(declare-fun z0 () (_ BitVec 32))
(declare-fun z1 () (_ BitVec 32))
(declare-fun z2 () (_ BitVec 32))
(declare-fun z3 () (_ BitVec 32))
(declare-fun z4 () (_ BitVec 32))
(declare-fun x0 () (_ BitVec 32))

(assert (= z1 (_ bv0 32)))
(assert (= z2 x0))
(assert (= z3 (ite (bvslt x0 (_ bv0 32)) z1 z2)))
(assert (= z4 (ite (not (= x0 (_ bv5 32))) z3 z0)))

(assert (not (and
  (=> (and (not (= x0 (_ bv5 32)))
    (not (bvslt x0 (_ bv0 32)))) (not (= z0 (bvneg (_ bv1 32)))))
  (=> (not (= x0 (_ bv5 32))) (bvsgt z3 (_ bv0 32)))
  (=> (not (= x0 (_ bv5 32))) (not (= z3 (_ bv5 32))))
)))

(check-sat)
```

Use `(get-value (x0 z0))`
to find inputs that make the
program fail

Summary

To verify a loop-free, call-free piece of code:

- Transform to static single assignment (SSA) form
- In SSA form each variable is assigned once
- Conditionals are handled during SSA conversion using predication
- From SSA form we can turn the program into a set of constraints
- Constraints are **unsat** \Leftrightarrow program is **correct**
- Satisfiability can be checked by an SMT solver
- Constraints are described using SMT-LIB2 format
- Z3 is a state-of-the-art SMT solver