Software Reliability

Lecture 3

Static Program Verification (ctd.)

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

Static program verification so far

Recap:

Static verification of call- and loop-free programs: **done**

- SSA conversion
- Predication to handle conditionals
- SMT-LIB formula generation
- Invoke SMT solver

Now let us see how to handle **procedure calls** and **loops**

We start by introducing two new statements: **havoc** and **assume**

Havoc

```
havoc v;
```

Sets **v** to an **arbitrary** value

Sometimes written as:

```
v = *;
```

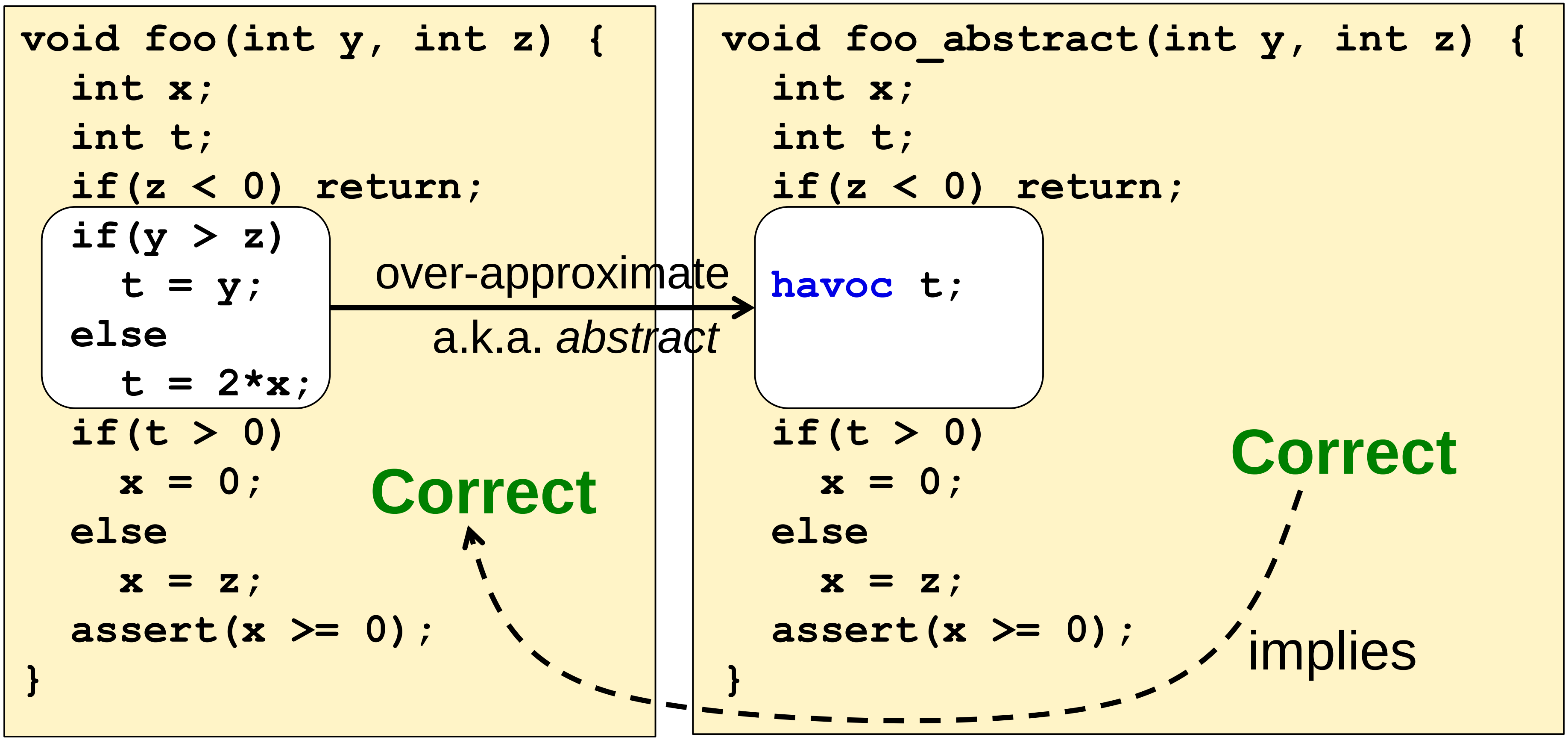
Not an executable statement – only makes sense in the context of program analysis

```
int x = 5;
havoc x;
assert(x == 5); // May fail
```

```
int x = 5;
havoc x;
assert(x != 5); // May also fail
```

What’s the point of havoc?

Allows us to **over-approximate** parts of programs



```
void foo(int y, int z) {
  int x;
  int t;
  if(z < 0) return;
  if(y > z)
    t = y;
  else
    t = 2*x;
  if(t > 0)
    x = 0;
  else
    x = z;
  assert(x >= 0);
}
```

over-approximate

a.k.a. *abstract*

```
void foo_abstract(int y, int z) {
  int x;
  int t;
  if(z < 0) return;

  havoc t;

  if(t > 0)
    x = 0;
  else
    x = z;
  assert(x >= 0);
}
```

Correct

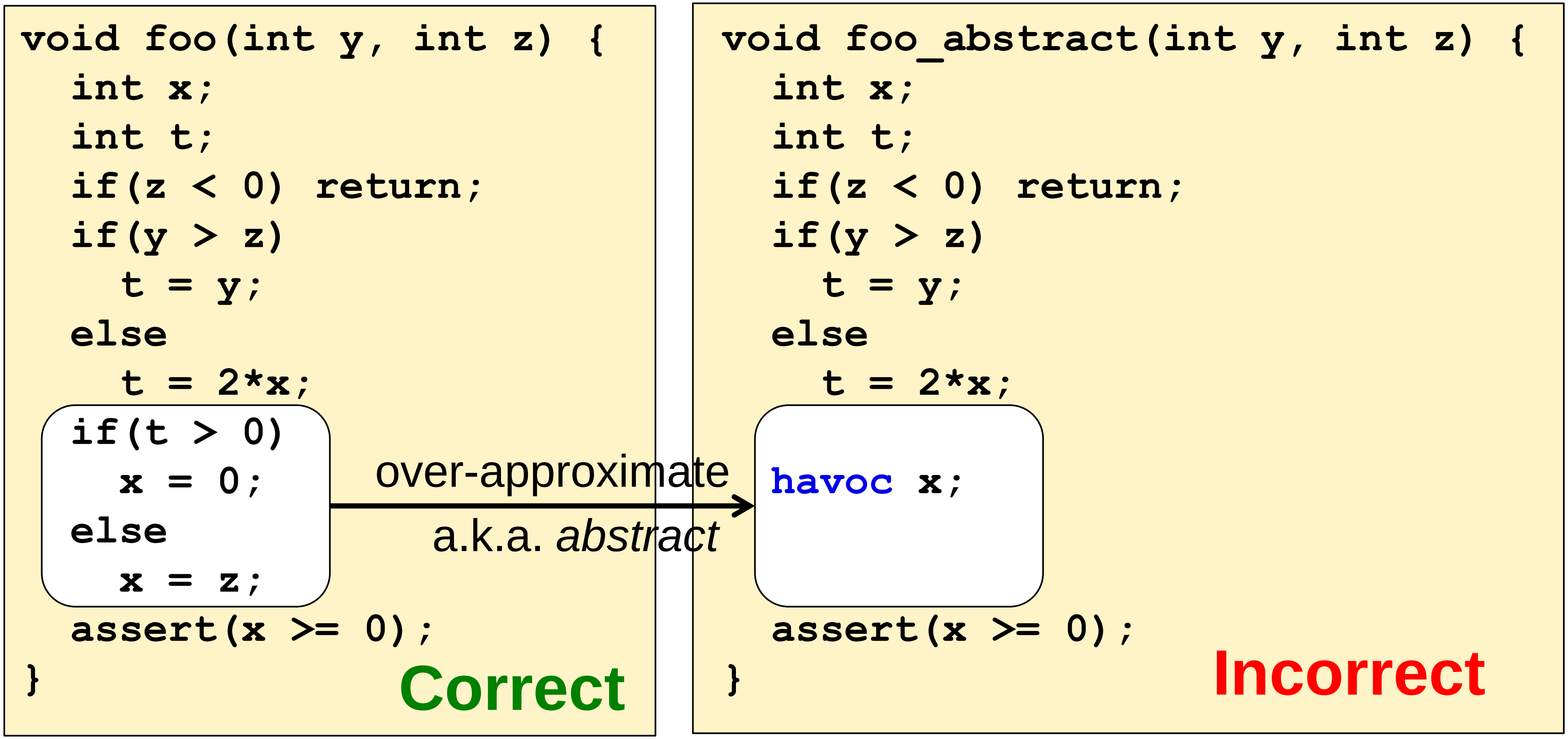
Correct

implies

foo_abstract captures all the behaviours of **foo**, and **more**

foo_abstract correct => **foo correct**

Over-approximation: if, not iff



```
void foo(int y, int z) {
  int x;
  int t;
  if(z < 0) return;
  if(y > z)
    t = y;
  else
    t = 2*x;
  if(t > 0)
    x = 0;
  else
    x = z;
  assert(x >= 0);
}
```

Correct

over-approximate

a.k.a. *abstract*

```
void foo_abstract(int y, int z) {
  int x;
  int t;
  if(z < 0) return;
  if(y > z)
    t = y;
  else
    t = 2*x;

  havoc x;

  assert(x >= 0);
}
```

Incorrect

foo_abstract captures all the behaviours of **foo**, and **more**

We may have foo_abstract incorrect but foo **correct**

Modifies sets again

Recall:

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

- **modset**(*v* **=** *E*) = { *v* }
- **modset**(`assert` *E*) = { }
- **modset**(*S* `;` *T*) = **modset**(*S*) ∪ **modset**(*T*)
- **modset**(`if (`*E*`) {`*S*`} else {`*T*`}`) = **modset**(*S*) ∪ **modset**(*T*)

Let us add:

- **modset**(`havoc` *v*) = { *v* }

And while we are at it:

- **modset**(`while (`*E*`) {`*S*`}`) = **modset**(*S*)

Modifies sets again

Put simply: statement *S* **possibly modifies** *v* if *v* = *e* or `havoc` *v* occurs in *S*

if and **while** are *compound* statements

```
x = 5;
```

possibly modifies { **x** }

```
if(e) x = 4;
else y = 5;
```

possibly modifies { **x**, **y** }

```
while(x < 100) {
  if((x % y) == 0) {
    havoc z;
    y = y - 1;
  }
  x = x + 1;
}
```

possibly modifies { **x**, **y**, **z** }

Over-approximation using havoc

modset(*S*): variables **possibly modified** by *S*

Suppose:

- Statement *S* appears in program **P**
- *S* contains no assertions
- **modset**(*S*) = { v_1, v_2, …, v_n }

Program **P'** – identical to **P**, but *S* replaced with:

`havoc` v_1; **`havoc`** v_2; …; **`havoc`** v_n;

havoc: an **extreme** form of over-approximation – we will see less extreme forms

P' **over-approximates** **P**

If **P'** is **correct** then **P** is **correct**

If **P'** is **incorrect** then **P** **may or may not be incorrect**

Assume

```
assume e;
```

No-op if e is **true**
Blocks execution if e is **false**

Can be thought of as equivalent to:

```
while(!e) { }
```

Like **havoc**, only really makes sense in the context of program analysis

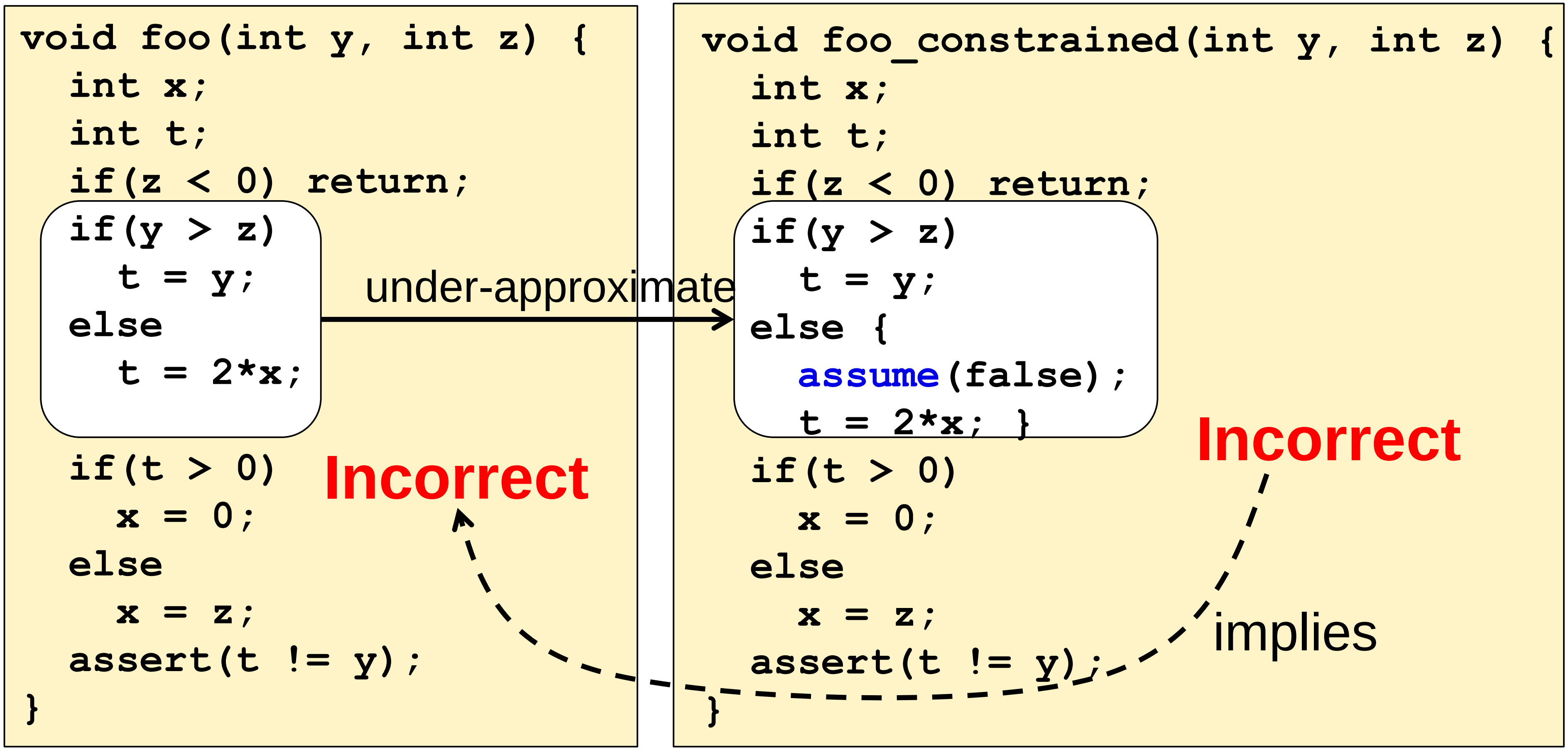
```
int x = 5;
assume(x == 4);
assert(x == 0); // CORRECT!
```

```
int x = 5;
assume(x > 0);
assert(x == 0); // INCORRECT!
```

Which **assume** statement **unconditionally** blocks execution?

What’s the point of assume?

Allows us to **under-approximate** parts of programs



```
void foo(int y, int z) {
  int x;
  int t;
  if(z < 0) return;
  if(y > z)
    t = y;
  else
    t = 2*x;

  if(t > 0)
    x = 0;
  else
    x = z;
  assert(t != y);
}
```

```
void foo_constrained(int y, int z) {
  int x;
  int t;
  if(z < 0) return;
  if(y > z)
    t = y;
  else {
    assume(false);
    t = 2*x; }
  if(t > 0)
    x = 0;
  else
    x = z;
  assert(t != y);
}
```

foo_constrained captures a **subset** of **foo**’s behaviours

foo_constrained incorrect => **foo incorrect**

Under-approximation: if, not iff

```
void foo(int y, int z) {
  int x;
  int t;
  if(z < 0) return;
  if(y > z)
    t = y;
  else
    t = 2*x;

  if(t > 0)
    x = 0;
  else
    x = z;
  assert(t != y);
}
```

Incorrect

under-approximate

```
void foo_constrained(int y, int z) {
  int x;
  int t;
  if(z < 0) return;
  assume(false);
  if(y > z)
    t = y;
  else
    t = 2*x;
  if(t > 0)
    x = 0;
  else
    x = z;
  assert(t != y);
}
```

Correct

foo_constrained captures a **subset** of **foo**'s behaviours

We may have foo_constrained correct but foo incorrect

Under-approximation using assume

Suppose program **P** contains statement *S*

Program **P'** – identical to **P**, except *S* replaced with:

> **`assume`** *e*; *S*

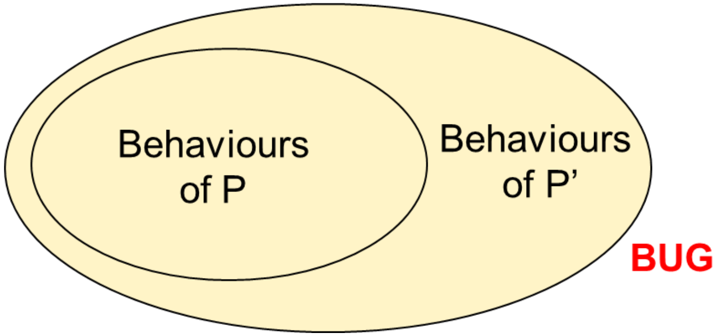
P' under-approximates P

- If **P'** is **incorrect** then **P** is **incorrect**
- If **P'** is **correct**, **P may or may not be correct**

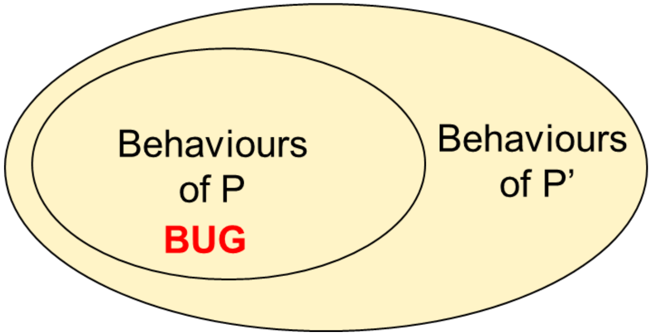
P’ over-approximates P

Three possible scenarios:

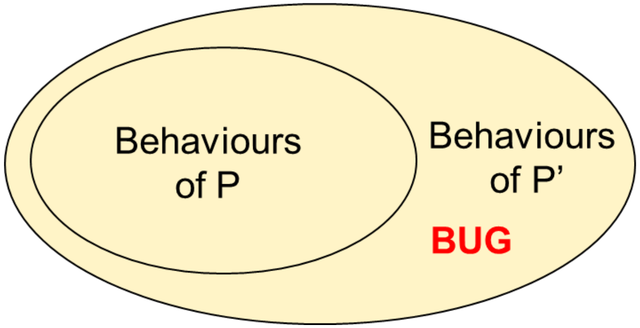
P and P’ both correct



P and P’ both incorrect



P’ incorrect bug P correct

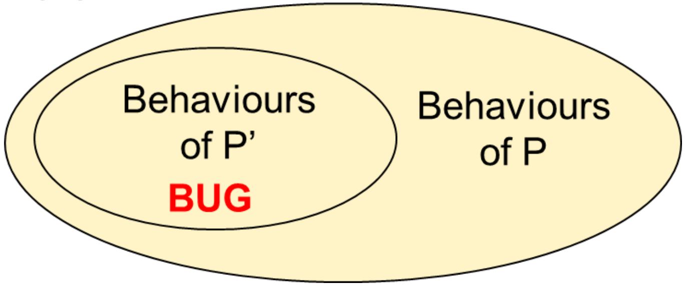


If we analyse P’ **only**, we cannot directly distinguish between these

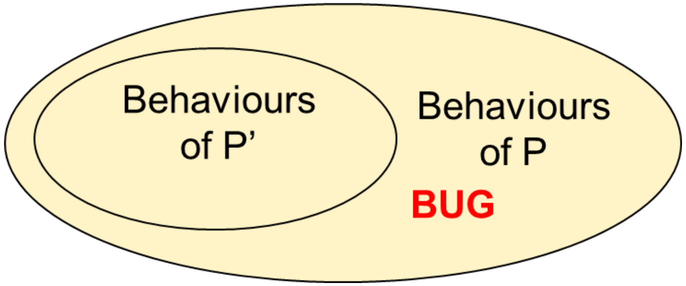
P’ under-approximates P

Three possible scenarios:

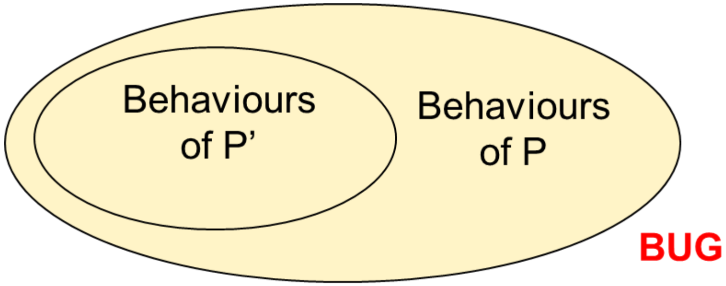
P and P’ both incorrect



P’ correct but P incorrect



P and P’ both correct



If we analyse P’ **only**, we cannot distinguish between these

Usefulness of over- and under-approximations

Both can be used to **simplify analysis**

Over-approximation:

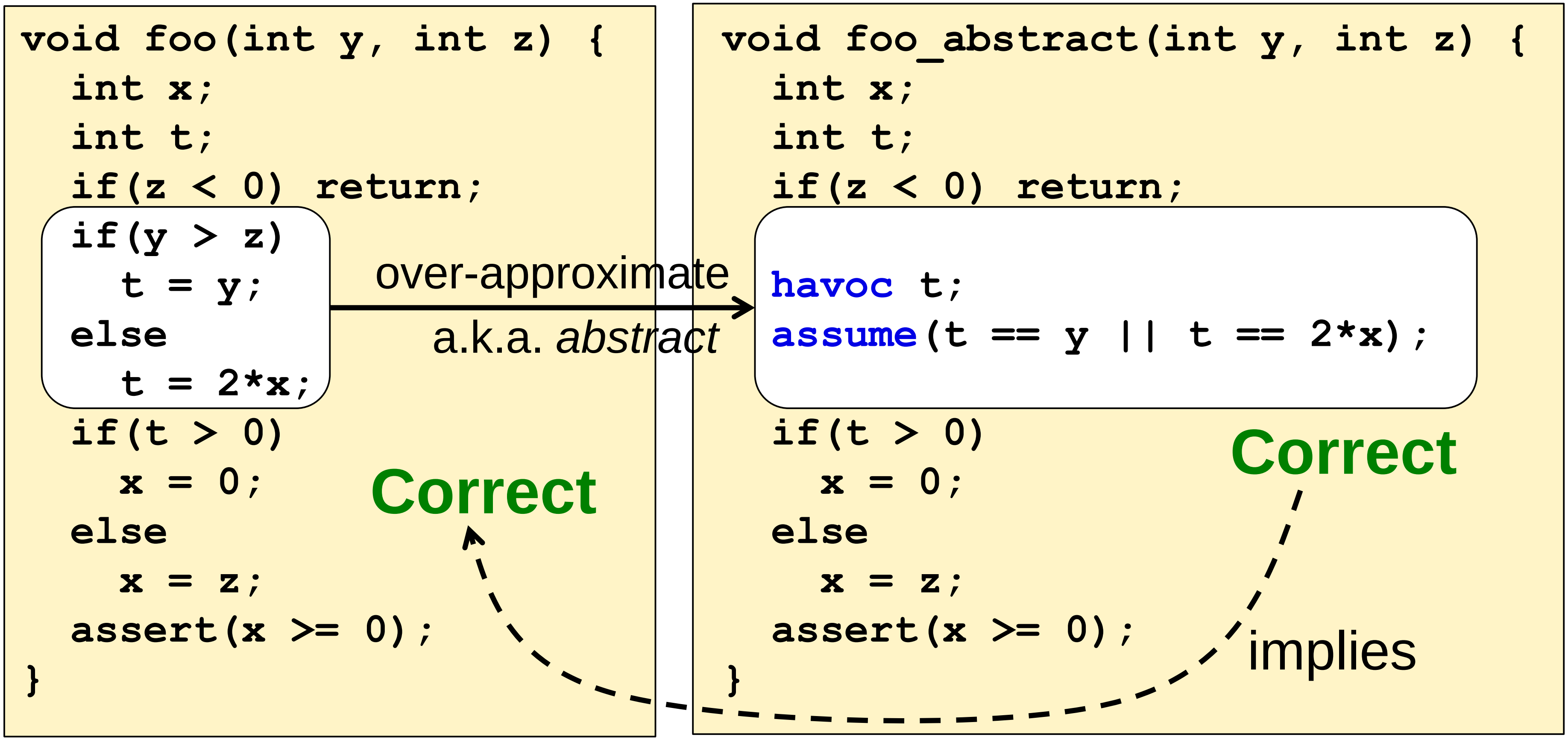
- may enable a proof of correctness
- **adds** program behaviours – **may add bugs**, leading to **false positives**

Under-approximation:

- may enable detection of bugs
- **removes** program behaviours – **may remove genuine bugs**, leading to **false negatives**

Over-approximation using havoc and assume

Havoc is extreme – we can use **assume** to make it less so



```
void foo(int y, int z) {
  int x;
  int t;
  if(z < 0) return;
  if(y > z)
    t = y;
  else
    t = 2*x;
  if(t > 0)
    x = 0;
  else
    x = z;
  assert(x >= 0);
}
```

over-approximate

a.k.a. *abstract*

```
void foo_abstract(int y, int z) {
  int x;
  int t;
  if(z < 0) return;
  havoc t;
  assume(t == y || t == 2*x);
  if(t > 0)
    x = 0;
  else
    x = z;
  assert(x >= 0);
}
```

Correct

implies

Correct

Captures the **possible** effects of the conditional

Loses relationship between value of **t** and condition **y > z**

Handling calls: inlining

//@ requires/ensures/modifies: JML notation

```
int a, b, c;

//@ requires x > 0;
void foo(int x) {
  a = 2;
  b = bar(x);
  assert(b >= 0);
}

//@ requires a > 0;
//@ requires y > 0;

//@ modifies c;

//@ ensures \result > 0;

int bar(int y) {
  c = 0;
  return a + y;
}
```

To verify **foo**, we could inline **bar**:

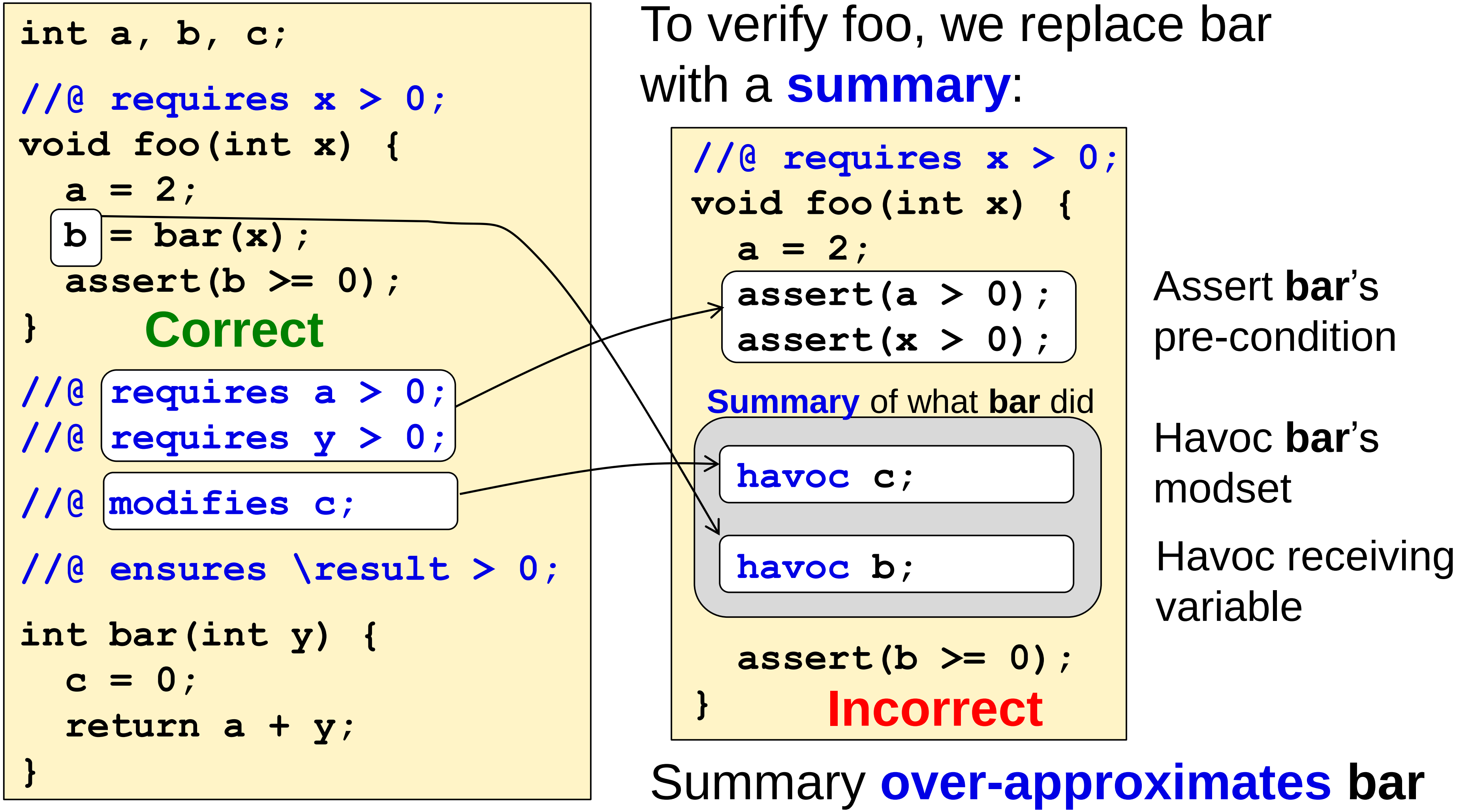
```
//@ requires x > 0;
void foo(int x) {
  a = 2;

  int y = x;

  assert(a > 0);
  assert(y > 0);

  c = 0;
  b = a + y;

  assert(b > 0);

  assert(b >= 0);
}
```

Check **bar**'s pre-condition

Check **bar**'s post-condition

Inlining: does not **scale**, cannot handle **recursion**

Handling calls: summarisation



```
int a, b, c;

//@ requires x > 0;
void foo(int x) {
  a = 2;
  b = bar(x);
  assert(b >= 0);
}

//@ requires a > 0;
//@ requires y > 0;

//@ modifies c;

//@ ensures \result > 0;

int bar(int y) {
  c = 0;
  return a + y;
}
```

Correct

To verify foo, we replace bar with a **summary**:

```
//@ requires x > 0;
void foo(int x) {
  a = 2;
  assert(a > 0);
  assert(x > 0);

  havoc c;

  havoc b;

  assert(b >= 0);
}
```

Summary of what **bar** did

Incorrect

Assert **bar**'s pre-condition

Havoc **bar**'s modset

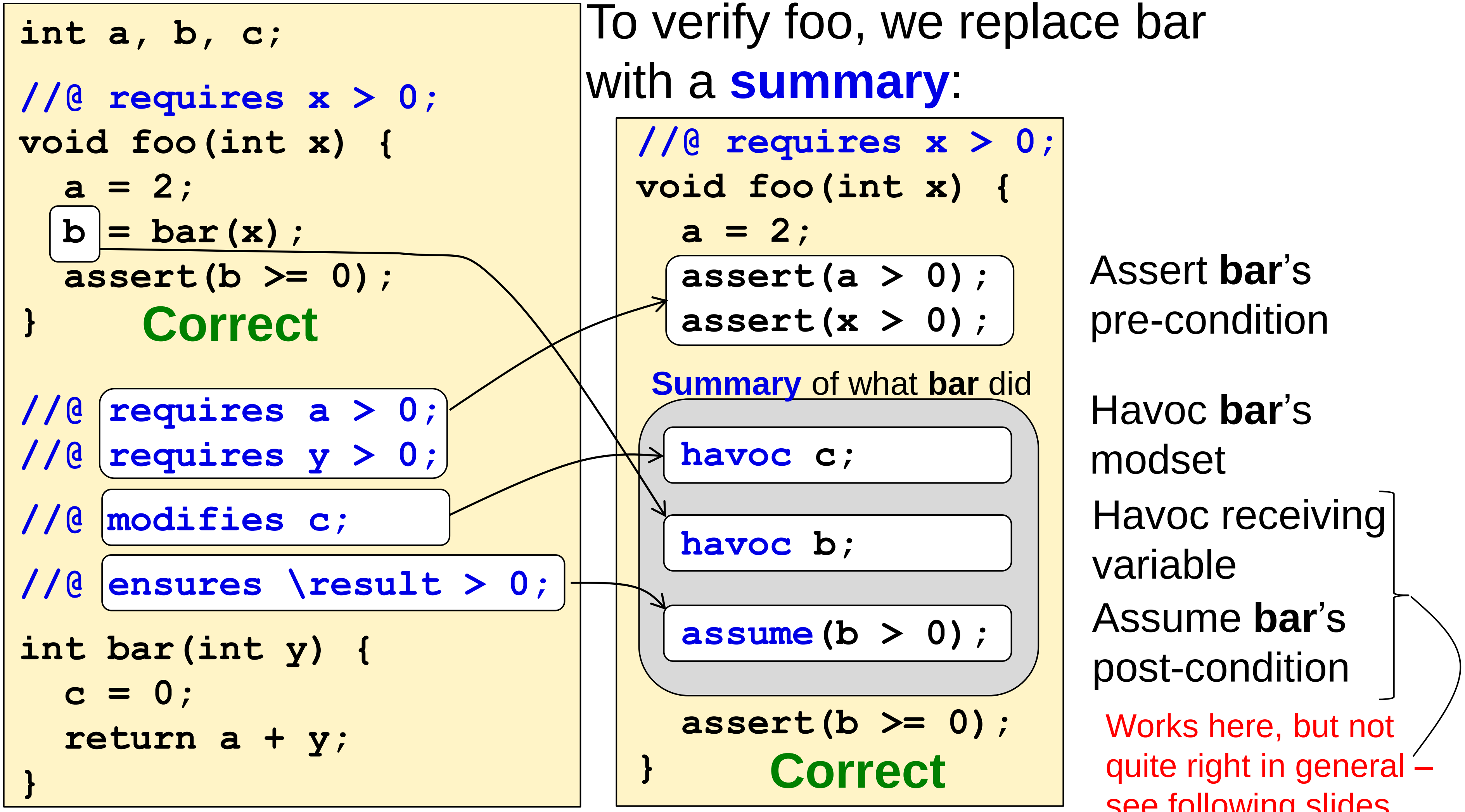
Havoc receiving variable

Summary **over-approximates** **bar**

…but the over-approximation is **too coarse** – **false positive**

Improving summary using bar’s post-condition

To verify foo, we replace bar with a **summary**:



```
int a, b, c;

//@ requires x > 0;
void foo(int x) {
  a = 2;
  b = bar(x);
  assert(b >= 0);
}
```

Correct

```
//@ requires a > 0;
//@ requires y > 0;

//@ modifies c;

//@ ensures \result > 0;

int bar(int y) {
  c = 0;
  return a + y;
}
```

```
//@ requires x > 0;
void foo(int x) {
  a = 2;
  assert(a > 0);
  assert(x > 0);
```

Summary of what **bar** did

```
  havoc c;

  havoc b;

  assume(b > 0);

  assert(b >= 0);
}
```

Correct

Assert **bar**’s pre-condition

Havoc **bar**’s modset

Havoc receiving variable

Assume **bar**’s post-condition

Works here, but not quite right in general – see following slides

Exploiting **bar**’s post-condition eliminates the false positive here

Summarisation in general: Ally's first attempt

Call statement:

v = **bar**(**e**$_1$, … , **e**$_n$);

Specification (a.k.a. **contract**) for **bar**:

//@ requires **P**;
//@ modifies **g**$_1$, … , **g**$_m$;
//@ ensures **Q**;
int **bar**(**int p**$_1$, … , int **p**$_n$);

Replace call with **pre-condition assertion** and **summary**:

assert(**P**[**e**$_1$/**p**$_1$, … , **e**$_n$/**p**$_n$]);	Assert pre-condition
havoc g$_1$; … ; **havoc g**$_m$;	Havoc **modset**
havoc v;	Havoc receiving variable
assume(**Q**[**v / \result**]);	Assume post-condition

This is not quite right

Problem with first attempt

Consider:

```
int c;

void foo(int x) {
  c = bar();
  assert(false);
}
```

Incorrect

```
//@ ensures \result != c;
int bar() {
  return c + 1;
}
```

Replacing bar with summary (according to previous slide) gives:

```
void foo(int x) {
  havoc c; // havoc receiving variable
  assume( (\result != c)[c / \result] ); // assume post-condition
  assert(false);
}
```

Simplifies to:

```
void foo(int x) {
  havoc c; // havoc receiving variable
  assume(c != c); // !!!
  assert(false);
}
```

Correct

So this is **not** a sound over-approximation

Correct summarisation

Fix: Introduce fresh temporary variable, **bar_ret**, to capture post-condition of **bar**

Call statement:

```
v = bar($e_1$, … , $e_n$);
```

Specification for **bar**:

```
//@ requires P;
//@ modifies $g_1$, … , $g_m$;
//@ ensures Q;
int bar(int $p_1$, … , int $p_n$);
```

Replace call with **pre-condition assertion** and **summary**:

```
assert(P[$e_1$/$p_1$, … , $e_n$/$p_n$]);
havoc $g_1$; … ; havoc $g_m$;
havoc bar_ret;
assume(Q[bar_ret / \result]);

v = bar_ret;
```

Assert pre-condition

Havoc **modset**

Havoc return temp

Assume post-condition **for return temp**

Copy into receiving variable

Soundness of summarisation

foo calls **bar**

When verifying **foo** we replace **bar** by its summary

Is this sound?

Yes, as long as we also verify **bar**

bar's summary **over-approximates** **bar** if **bar** is **correct**

If **bar** is **not** correct then it may be **unsound** to assume **bar**'s post-condition

Result: modular verification only succeeds if we manage to verify **all** procedures

Note: coarse summarisation without assuming post-condition is always OK (though not often useful)

Exercise

Can you work out how to adapt summarisation to support post-conditions that use **\old** to refer to the values held by global variables on procedure entry?

Modsets in real-world languages

Sound analysis depends on being able to compute **modifies** sets for procedures

Why is this **easy** in Simple C?

Why is it **not at all easy** in Java or C?

Big picture

Given program with procedures and procedure calls, but no loops:

- Use **summarisation** to over-approximate calls
- Every procedure is now loop- and call-free
- Apply techniques of last lecture for verification

Problem: how do we treat our new friends, **havoc** and **assume**, during **SSA conversion**?

SSA conversion for havoc

This is trivial

- “`havoc` *v*” means: “forget everything about *v*”
- How do we do that during SSA conversion?
- Just give *v* a fresh SSA id

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

```
y1 = x0 + 1;
// increment x's id
assert y1 == x1 + 1;
```

SSA conversion for assume

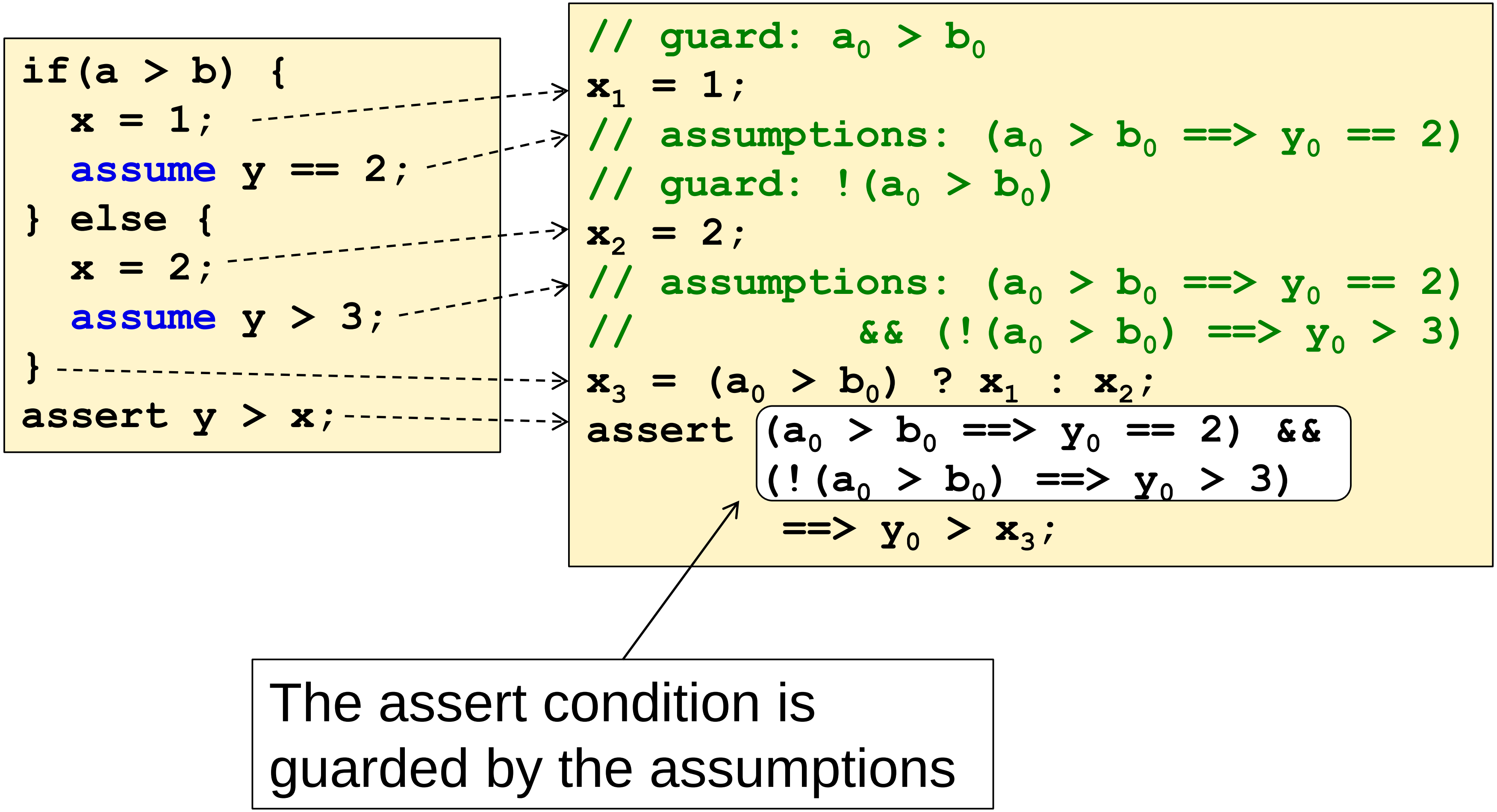
Recall: **“`assume`** *e*” blocks execution unless *e* holds

Our goal in verification is to check assertions so:

- If an **assume** fails…
- …no **assertion** should be checked afterwards

We can account for this by tracking a set of **global assumptions**, and guarding assertions by these assumptions

SSA conversion for assume: example



```
if(a > b) {
  x = 1;
  assume y == 2;
} else {
  x = 2;
  assume y > 3;
}
assert y > x;
```

```
// guard: a_0 > b_0
x_1 = 1;
// assumptions: (a_0 > b_0 ==> y_0 == 2)
// guard: !(a_0 > b_0)
x_2 = 2;
// assumptions: (a_0 > b_0 ==> y_0 == 2)
//              && (!(a_0 > b_0) ==> y_0 > 3)
x_3 = (a_0 > b_0) ? x_1 : x_2;
assert (a_0 > b_0 ==> y_0 == 2) &&
       (!(a_0 > b_0) ==> y_0 > 3)
        ==> y_0 > x_3;
```

The assert condition is guarded by the assumptions

Updated SSA conversion algorithm

We had:

toSSA(*S*, *Pred*, *M*)

We now have:

toSSA(*S*, *Pred*, *Assumptions*, *M*)

Assumptions is **passed by reference**: assumptions grow as we translate **assume** statements

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

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

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

Updated SSA conversion algorithm

```
toSSA(v = E, Pred, Assumptions, M) {
   newId := fresh(v);
   emit("$v_{newId}$ = apply(E, M) ; ");
   M(v) := newId;
}
toSSA(assert E, Pred, Assumptions, M) {
   emit("assert (Assumptions && Pred) ==> apply(E, M) ; ");
}
toSSA(S ; T, Pred, Assumptions, M) {
   toSSA(S, Pred, Assumptions, M);
   toSSA(T, Pred, Assumptions, M);
}
```

Only check the assertion if the current assumptions hold

SSA conversion algorithm

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

Updated SSA conversion algorithm

New cases for `havoc` and `assume`:

```
toSSA(havoc v, Pred, Assumptions, M) {
  M(v) := fresh(v);
}
toSSA(assume E, Pred, Assumptions, M) {
  Assumptions := Assumptions && (Pred ==> apply(E, M));
}
```

Recall that *Assumptions* is passed **by reference**

Add the predicated assume condition as a new assumption

Translating a loop-free procedure

```
int foo(…)
  requires $R_1$, …, requires $R_m$,
  ensures $E_1$, …, ensures $E_n$ {
      $S$;
      return $e$;
}
```

Suppose that specifications for all procedures called inside *S* are available

Translating a loop-free procedure

First, rewrite:

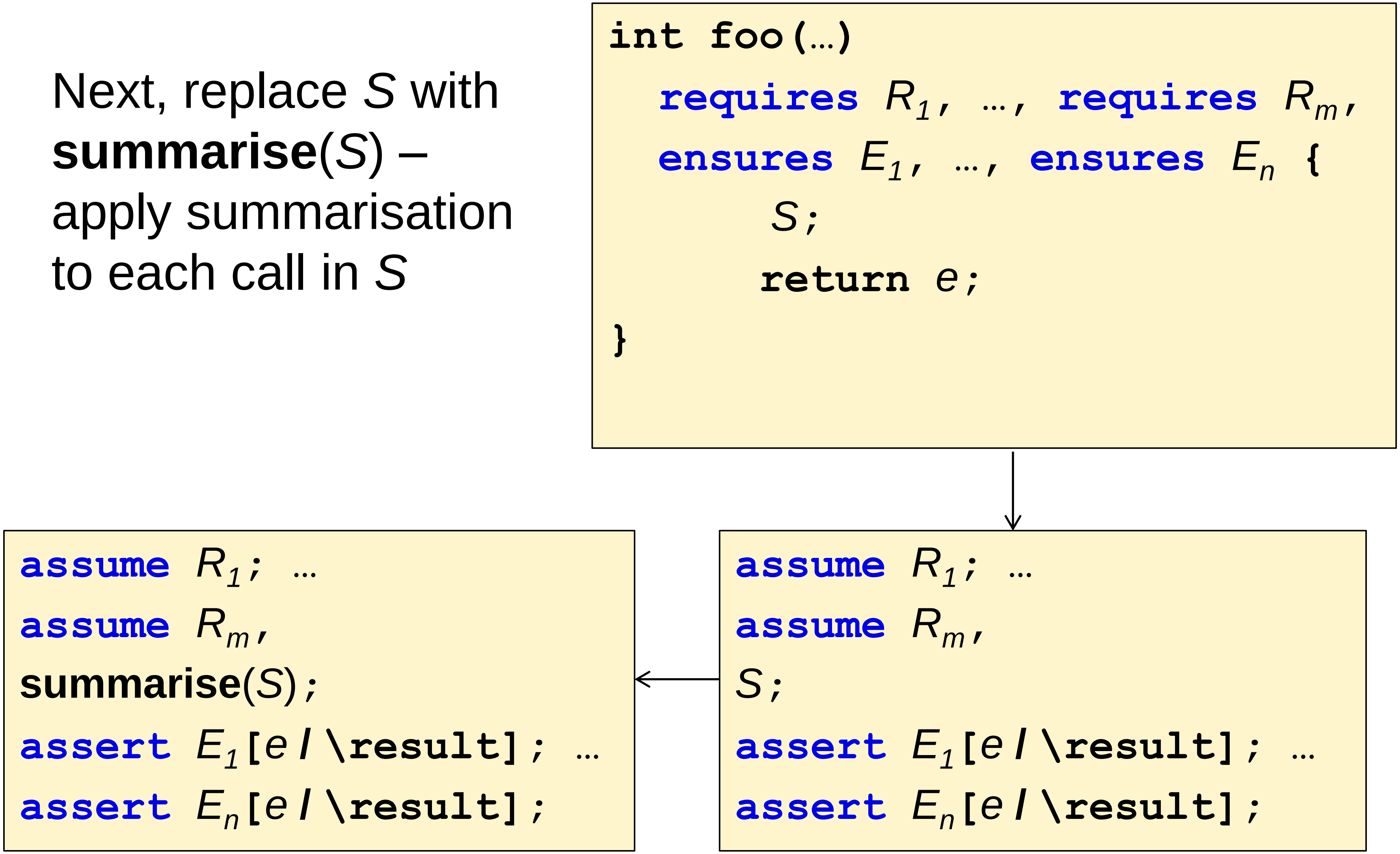
- preconditions as **assumes**
- postconditions as **asserts**

```
int foo(…)
  requires R1, …, requires Rm,
  ensures E1, …, ensures En {
        S;
        return e;
}
```

```
assume R1; …
assume Rm,
S;
assert E1[e / \result]; …
assert En[e / \result];
```

Translating a loop-free procedure

Next, replace S with **summarise**(S) – apply summarisation to each call in S



`int foo(…)`
  `requires` R_1, …, `requires` R_m,
  `ensures` E_1, …, `ensures` E_n `{`
    S;
    `return` e;
`}`

`assume` R_1; …
`assume` R_m,
S;
`assert` $E_1[e\,/\,$`\result`$]$; …
`assert` $E_n[e\,/\,$`\result`$]$;

`assume` R_1; …
`assume` R_m,
summarise(S);
`assert` $E_1[e\,/\,$`\result`$]$; …
`assert` $E_n[e\,/\,$`\result`$]$;

Translating a loop-free procedure

We now have a loop-free, call-free program:

```
assume $R_1$; …
assume $R_m$,
summarise($S$);
assert $E_1$[$e$ / \result]; …
assert $E_n$[$e$ / \result];
```

Apply **SSA conversion**, turn the program into a formula, and ask an SMT solver whether the formula is **satisfiable**

Formula **unsat** => original program is **correct**

Why => and not <=>?

Next: **loops**

Simple procedure with loop

```
void foo(int x) {
  int i;
  i = 0;
  while(i < x) {
    i = i + 1;
  }
  assert(i == x);
}
```

Obviously (to a human) correct

?

Not for all values of **x**

Simple procedure with loop

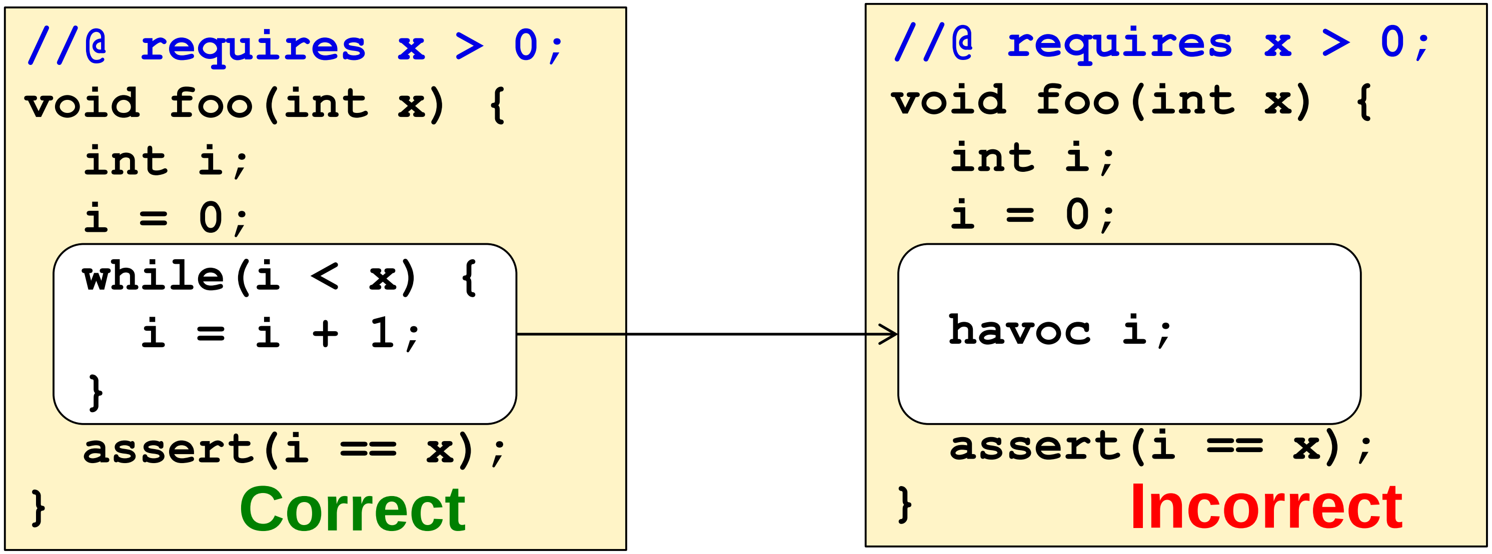
```
void foo(int x)
  requires x > 0 {
  int i;
  i = 0;
  while(i < x) {
    i = i + 1;
  }
  assert(i == x);
}
```

Now it is correct.

How do we verify it?

Over-approximating loops: first attempt

First idea: replace loop with statements that havoc the loop body's **modset**



```
//@ requires x > 0;
void foo(int x) {
  int i;
  i = 0;
  while(i < x) {
    i = i + 1;
  }
  assert(i == x);
}
```

Correct

```
//@ requires x > 0;
void foo(int x) {
  int i;
  i = 0;

  havoc i;

  assert(i == x);
}
```

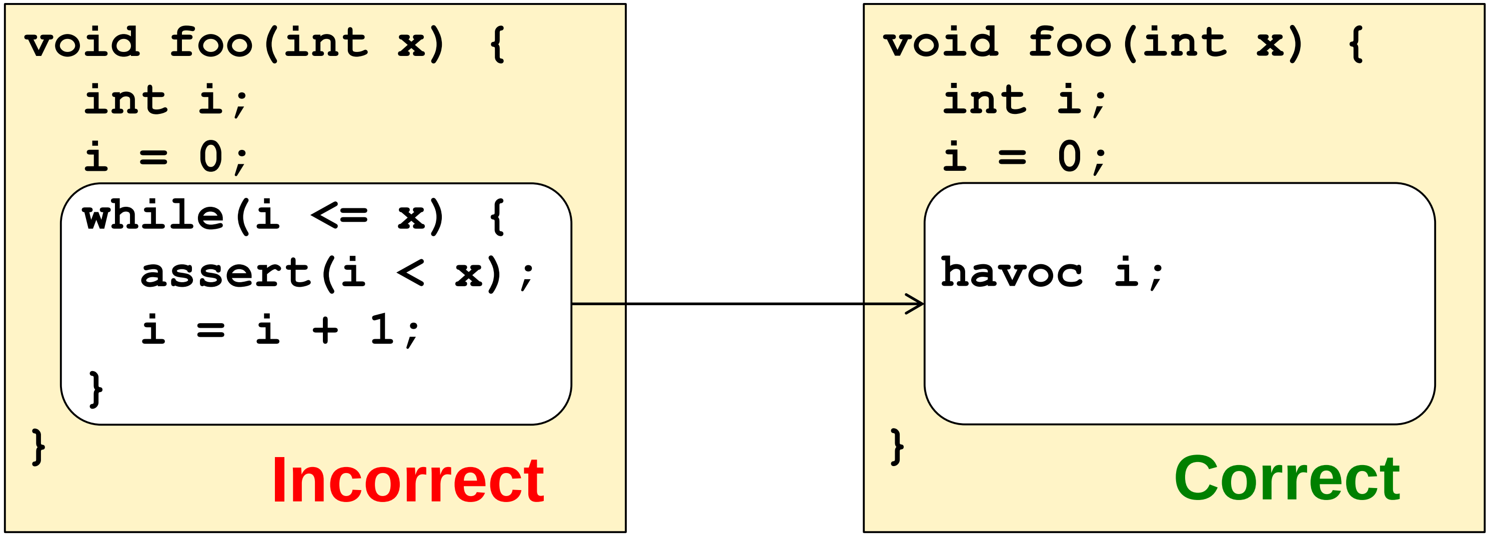
Incorrect

Problems:

1. only sound if loop body does not contain assertions
2. leads to a very coarse over-approximation (too coarse here)

Over-approximating loops: first attempt

Illustration of problem (1)



```
void foo(int x) {
  int i;
  i = 0;
  while(i <= x) {
    assert(i < x);
    i = i + 1;
  }
}
```

Incorrect

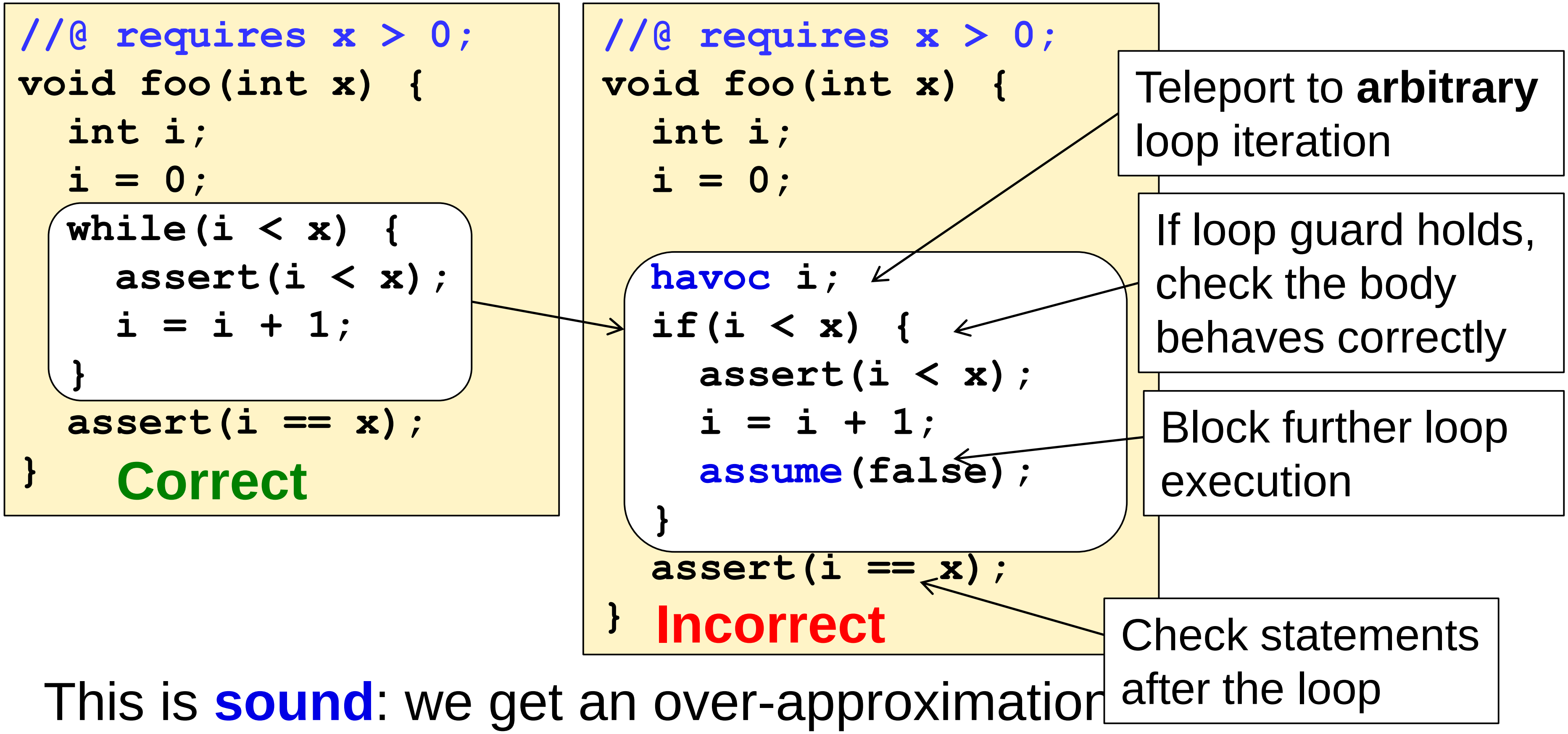
```
void foo(int x) {
  int i;
  i = 0;

  havoc i;


}
```

Correct

Havocking does not yield an over-approximation – we miss bugs that occur in the loop body

Over-approximating loops: second attempt

Check an **arbitrary** loop iteration



```
//@ requires x > 0;
void foo(int x) {
  int i;
  i = 0;
  while(i < x) {
    assert(i < x);
    i = i + 1;
  }
  assert(i == x);
}
```

Correct

```
//@ requires x > 0;
void foo(int x) {
  int i;
  i = 0;

  havoc i;
  if(i < x) {
    assert(i < x);
    i = i + 1;
    assume(false);
  }
  assert(i == x);
}
```

Incorrect

Teleport to **arbitrary** loop iteration

If loop guard holds, check the body behaves correctly

Block further loop execution

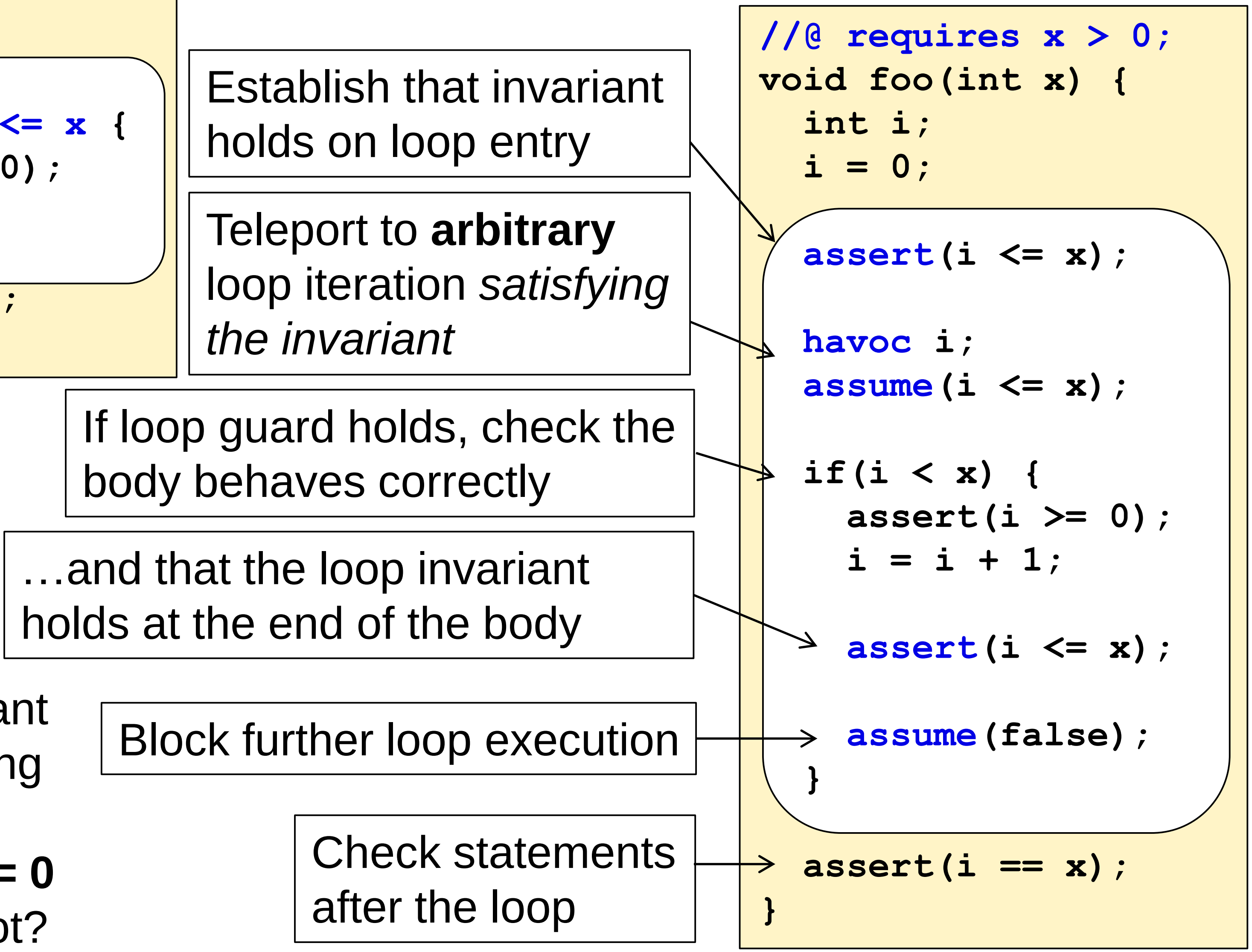
Check statements after the loop

This is **sound**: we get an over-approximation

However, **havocking** is very **coarse**

Over-approximating loops: third attempt - use **invariant**

```
//@ requires x > 0;
void foo(int x) {
  int i;
  i = 0;
  while(i < x)
    invariant i <= x {
    assert(i >= 0);
    i = i + 1;
  }
  assert(i == x);
}
```



Establish that invariant holds on loop entry

Teleport to **arbitrary** loop iteration *satisfying the invariant*

If loop guard holds, check the body behaves correctly

…and that the loop invariant holds at the end of the body

Block further loop execution

Check statements after the loop

```
//@ requires x > 0;
void foo(int x) {
  int i;
  i = 0;

  assert(i <= x);

  havoc i;
  assume(i <= x);

  if(i < x) {
    assert(i >= 0);
    i = i + 1;

    assert(i <= x);

    assume(false);
  }

  assert(i == x);
}
```

Problem: invariant **i <= x** is not strong enough to prove the assertion **i >= 0**

Exercise: why not?

A stronger invariant allows verification to succeed

```
//@ requires x > 0;
void foo(int x) {
  int i;
  i = 0;
  while(i < x)
    invariant i <= x,
    invariant i >= 0 {
    assert(i >= 0);
    i = i + 1;
  }
  assert(i == x);
}
```

Strengthening the invariant to include also **i >= 0** allows verification to succeed

```
//@ requires x > 0;
void foo(int x) {
  int i;
  i = 0;

  assert(i <= x);
  assert(i >= 0);
  havoc i;
  assume(i <= x);
  assume(i >= 0);
  if(i < x) {
    assert(i >= 0);
    i = i + 1;

    assert(i <= x);
    assert(i >= 0);
    assume(false);
  }

  assert(i == x);
}
```

Thanks to a former student for pointing out that **i <= x** alone is not strong enough!

The power of a loop invariant

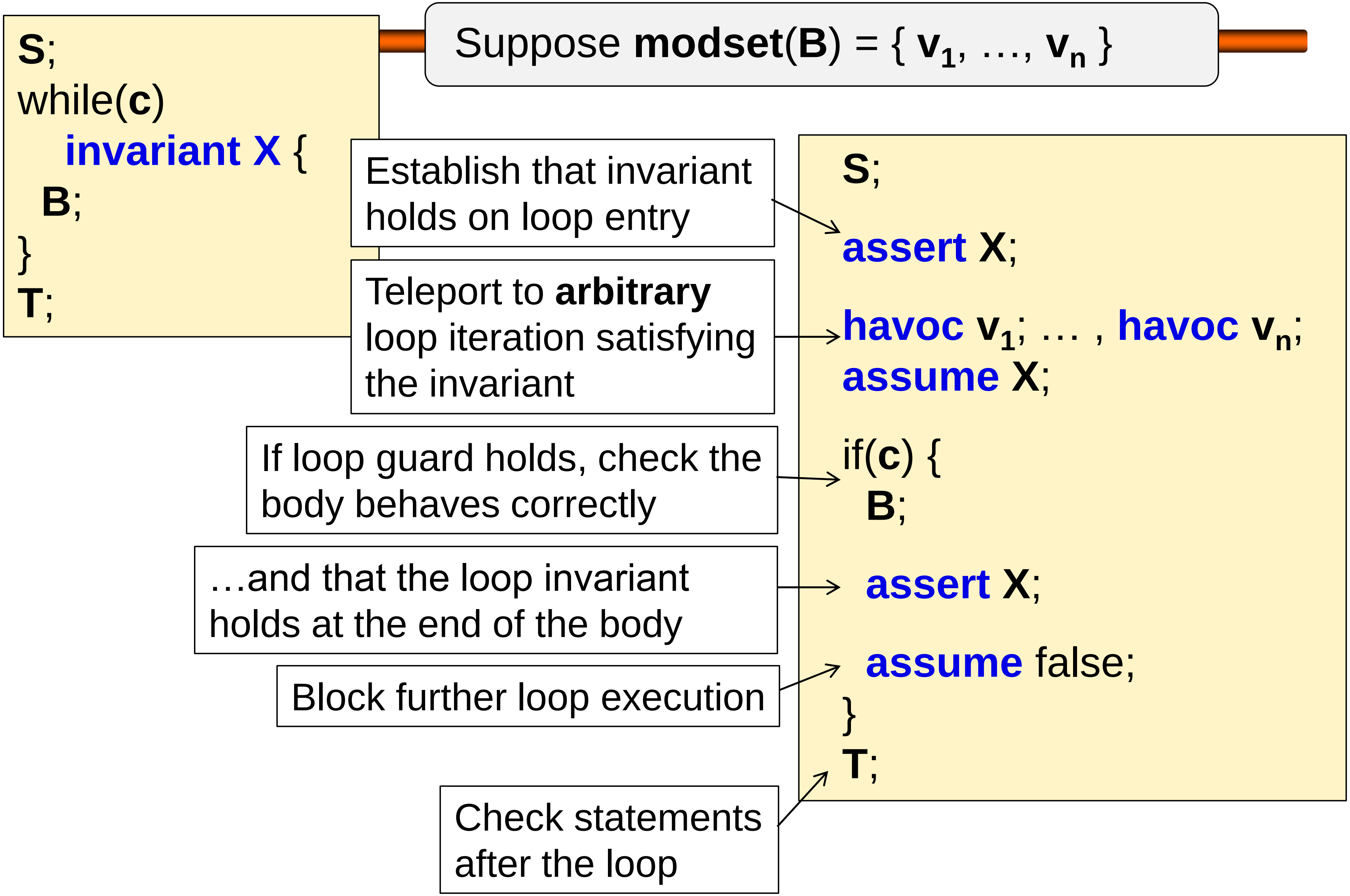
When checking code immediately after a loop we can **assume that the loop invariant holds**

Can provide a much better summary of the loop than obtained by only havocking the modset

We make this sound using **induction**:

- Prove invariant holds on **loop entry** (**base case**)
- Assume invariant holds for **arbitrary iteration** (**induction hypothesis**)
- Prove invariant holds at **end of iteration** (**step case**)

Over-approximating loops: general case



```
S;
while(c)
   invariant X {
  B;
}
T;
```

Suppose **modset**(**B**) = { $\mathbf{v_1}$, …, $\mathbf{v_n}$ }

Establish that invariant holds on loop entry

Teleport to **arbitrary** loop iteration satisfying the invariant

If loop guard holds, check the body behaves correctly

…and that the loop invariant holds at the end of the body

Block further loop execution

Check statements after the loop

```
S;

assert X;

havoc $v_1$; … , havoc $v_n$;
assume X;

if(c) {
  B;

  assert X;

  assume false;
}
T;
```

Static verification: complete story

Given set of procedures, specification for each procedure and invariant for every loop:

Replace each procedure call with summary according to callee’s specification

Replace each loop with summary according to loop’s invariant

Every procedure is now loop- and call-free

Apply **assume**- and **havoc**-aware SSA conversion

Build verification condition formula

Give it to SMT solver