SLAM
Abstraction

Roderick Bloem

Hoare Logic (A Few Things We'll Use)

Hoare triple **{P} S {Q}**

P: **precondition**
Q: **postcondition**
S: **program**

Meaning: if P holds before execution and S finishes, then Q holds afterwards.
Note: we prove **partial correctness**. If S runs forever, {P}S{Q} holds.

Example:

$\{x \geq 0\}$ y:= sqrt(x) $\{y * y = x\}$
x := x + 1 $\{x = 2\}$
$\{x > 9\}$ x := x + 1
x := x + 1 $\{x > 10\}$

In the following, all variables are integer.

Hoare Logic

Hoare triple **{P} S {Q},**

P: **precondition**
S: **program**
Q: **postcondition**

Meaning: if P holds before execution and S finishes, then Q holds afterwards.
Note: we prove **partial correctness**. If S runs forever, {P}S{Q} holds.

Examples:

1. $\{x \geq 0\}$ y:= sqrt(x) $\{y * y = x\}$
2. $\{\mathrm{TRUE}\}$ while(1) do skip od $\{\mathrm{x} + \mathrm{y} = 42\}$
3. {z = 1} z := y + 1 {y = 2}
4. {z > 9} z := y + 1 {y > 10}
5. {z > 100} z := y + 1 {y > 10}

Example 1 and 2: **weakest** precondition (all circumstances under which the program fulfills postcondition)
In the following, all variables are integer.

Hoare Logic: Rules

Axioms to find the weakest precondition

- Assignment: x := e
- Consecution: S1; S2
- if-statement: if b then S1 else S2
- Loops: while b do S od

- Plus
  - extra “glue” rules to make things work
  - Function calls, mallocs, pointers, etc

Axiom of Assignment

Example:

x := y {x = 4}
z := y + 1 {y = 4}
y := 2 * x {x = 8}
y := 2 * x {x < 8}

This rule gives the *weakest precondition,* i.e., {P[x→e]} holds before S **if and only if** P holds afterwards

Axiom of Assignment

$$\frac{}{\{P[x \rightarrow e]\}\ x{:=}e\ \{P\}}$$

P[x → e] means that x is replaced by e in P

Example:

{y = 4} x := y {x = 4}

{x+1 = 4} x := x + 1 {x = 4}

{x = 4} x := 2 * x {x = 8}

{x < 4} x := 2 * x {x < 8}

This rule gives the *weakest precondition,* i.e., {P[x→e]} holds before S **if and only if** P holds afterwards

Axiom of Skip

$$\frac{}{\{P\}\ \text{skip}\ \{P\}}$$

(skip is an abbreviation for x:=x)

Sequencing Rule (Consecution)

Example:

(1) {x = 3}	x := x + 1 {x = 4}	(ass.)
(2) {x = 4}	x := x * 2 {x = 8}	(ass.)
(3)	x := x + 1; x := x * 2	

***horizontal line**: if everything above the line is true, then everything below the line is true.*

Sequencing Rule (Consecution)

$$\frac{\{P\}\ S1\ \{Q\} \qquad \{Q\}\ S2\ \{R\}}{\{P\}\ S1;\ S2\ \{R\}}$$

Example:

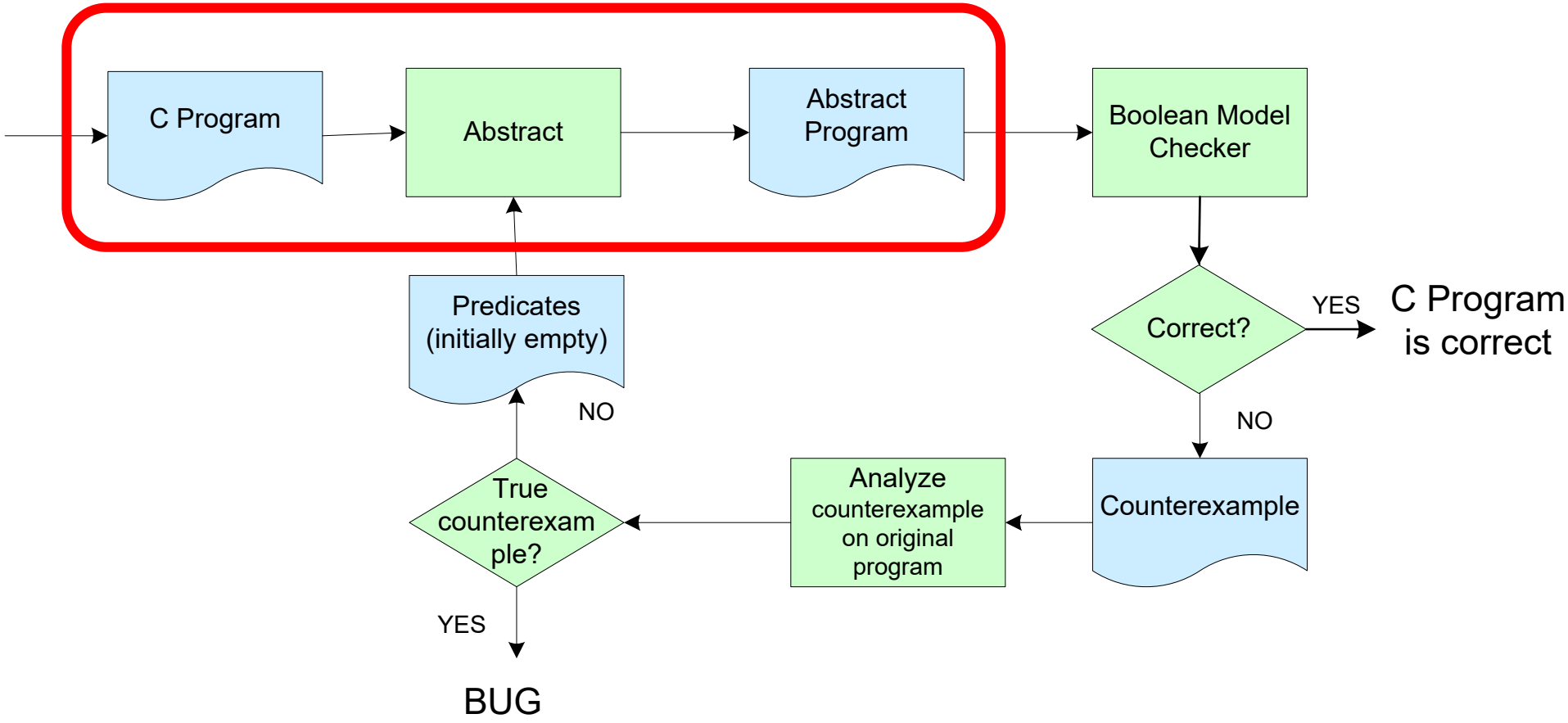
(1) {x+1 = 4} x := x + 1 {x = 4} (ass.)

(2) {x = 4} x := x * 2 {x = 8} (ass.)

(3) {x = 3} x := x + 1; x := x * 2 {x = 8} (consecution, 1,2)

***horizontal line**: if everything above the line is true, then everything below the line is true.*

The Approach



Abstraction

- Represent complex program by simple program
  - original program is **concrete**, simple one is **abstract**
- Construction: if abstraction correct, then original correct
  - But: abstract program may fail even if the original is correct
  - We will look at *refinement* later
- Whenever we can not make a decision with certainty, we allow all possibilities

Predicate Abstraction

- Replace variables by predicates. E.g., instead of `x` have the predicates
  - `b, meaning {x>0},`
  - `c: {x<0},`
  - `d: {x==0}`
- or replace `x` and `y` by
  - `e: {x==y}, or by`
  - `f: {x<y}, or by`
  - `g: {2x - y < 0},`

Predicate Abstraction

Example

`b: {x is odd}`

- `assert(x!=38)` becomes `assert(b)`
- `assert(b)` is **stricter**:
  - if `assert(x!=38)` fails then `assert(b)` also fails
  - But not vice-versa

What about

- If-then-else?
- Assignments?

Construct abstract programs **one statement at a time**

- Abstracting statement does not require full program analysis

Abstraction of Conditional

`*`: nondeterministic value

Original Program

```
if(x == 5) then
  S1
else
  S2
fi
```

Abstract Program `b = {x odd}`

Abstraction of Conditional

`*`: nondeterministic value

b	
{x odd}	{x = 5}
T	
F	

Original Program

```
if(x == 5) then
  S1
else
  S2
fi
```

Abstract Program `b = {x odd}`

Abstraction of Conditional

We use * to denote a nondeterministic value

b	
{x odd}	{x = 5}
T	* (e.g., x=3, x=5)
F	F

Original Program

```
if(x == 5) then
   S1
else
   S2
fi
```

Abstract Program (b = {x odd})

```
if(b?*:F) then
   S1
else
   S2
fi
```

Note:
- b=false is the same as x even, which implies x!=5.
- b=true means that x is odd, which means x may or may not be 5

Abstraction Example

Let's say we have one predicate:

`b = {x ≤ y}`

How do we abstract

`x := y?`

`y := y+1?`

Computing Abstraction

$b = \{x \leq y\}$

Use Hoare's weakest precondition

$x := y$

$x := y$ is abstracted to

Computing Abstraction

`b = {x ≤ y}`

Use Hoare's weakest precondition

`{y ≤ y}`

`x := y`

`{x ≤ y}`

Thus, `y≤y` before the statement iff `x≤y` after

`x := y` is abstracted to `b = true`

Computing Abstraction

Now for `y := y + 1`.
`b = {x ≤ y}`

Thus, `x ≤ y + 1` before iff `x ≤ y` after.
In which cases can we guarantee `x ≤ y+1`?

b	b′
{x ≤ y}	{x ≤ y+1}
T	
F	

abstraction:

Computing Abstraction

Now for `y := y + 1`.

```
{x ≤ y + 1}
y := y + 1
{x ≤ y}
```

Thus, `x ≤ y + 1` before iff `x ≤ y` after.
In which cases can we guarantee `x ≤ y+1`?

b	b′
{x ≤ y}	{x ≤ y+1}
T	T
F	* (e.g. (x,y)=(9,3) or (3,3))

Not enough information to decide whether `x≤y+1` before, so we approximate.
abstraction: `b = b ? T : *;`

Conservative Abstraction

Let us abstract `x` by **`b:{x < 0}.`**

We may lose some information Example:

```
x = -2;
x = x + 1;
assert(x<0);
```

is abstracted statement-by statement-to

The abstraction is *conservative*: bugs are preserved (but new bugs may occur).

Conservative Abstraction

Let us abstract `x` by **`b:{x < 0}.`**

We may loose some information Example:

```
x = -2;
x = x + 1;
assert(x<0);
```

is abstracted statement-by statement-to

```
b = true;
b = b ? * : false;
assert(b);
```

The abstraction is *conservative*: bugs are preserved (but new bugs may occur).

Multiple Predicates

Two predicates: `b={x ≤ y}` and `c={x=y+1}`

preconditions:

b	c		b'	c'
x≤y	x=y+1		x≤y+1	x=y+2
T	T			
T	F			
F	T			
F	F			

Multiple Predicates

Two predicates: `b={x ≤ y}` and `c={x=y+1}`

preconditions:

```
{x ≤ y + 1}
y := y + 1
{x ≤ y}

{x = y + 2}
y := y + 1
{x = y+1}
```

`y:=y+1` is abstracted to

```
(b, c) :=
  (b||c ? T : F,
   b||c ? F : *)
```

b	c		b'	c'
x≤y	x=y+1		x≤y+1	x=y+2
T	T	X	T	F
T	F	a ≤ b	T	F
F	T	a=b+1	T	F
F	F	a>b+1	F	*

In general, simultaneous assignments are needed for abstract statements

Multiple Predicates

- Two predicates: `b={x ≤ y}` and `c={x=y+1}`
- $x \leq y$ and $x = y + 1$ can never happen, so `b` and `c` should not both be true.
- To beginning of abstract program, add `assume(!b || !c)`
  - Invariant then holds by induction throughout program

Multiple Predicates

To beginning of program, add assumption forbidding impossible combinations

Example:

- Predicates: `a={x=0}, b={x>y}, c={y=0}`
- $a = b = c = 1$ cannot happen
- Possible abstraction of `x:=x+1` in table

Program	abstraction
`x := 0`	`a,b:=T,c?F:*`
`x := x + 1`	`see table`
`assert(x !=0)`	`assert(!a)`

Execution of abstract program:

```
        (a,b,c) = (T,T,T)
a := T
        (a,b,c) = (T,T,T)
see table
        (a,b,c) = (T,T,T)
assert(!a)
        fails
```

a	b	c		a'	b'	c'
x=0	x>y	y=0		x=-1	x-1>y	y=0
F	F	F	$0 \neq x \leq y \neq 0$	*	*	F
F	F	T	$0 \neq x \leq y = 0$	*	*	T
F	T	F	$0 \neq x > y \neq 0$	*	*	F
F	T	T	$x > y = 0$	*	*	T
T	F	F	$0 = x < y$	*	*	F
T	F	T	$x = y = 0$	*	T	T
T	T	F	$0 = x > y$	*	*	F
T	T	T	impossible	T	T	T

Multiple Predicates

To beginning of program, add assumption forbidding impossible combinations

Example:

- Predicates: `a={x=0}, b={x>y}, c={y=0}`
- $a = b = c = 1$ cannot happen
- Possible abstraction of `x:=x+1` in table

Program	abstraction
	`assume(!(a && b && c))`
`x := 0`	`a,b:=T,c?F:*`
`x := x + 1`	`see table`
`assert(x !=0)`	`assert(!a)`

Execution of abstract program:

```
(a,b,c) = (T,T,T)
a := T
(a,b,c) = (T,T,T)
see tabl
(a,b,c) = (T,T,T)
assert(a)
fails
```

a	b	c		a'	b'	c'
x=0	x>y	y=0		x=-1	x-1>y	y=0
F	F	F	$0 \neq x \leq y \neq 0$	*	*	F
F	F	T	$0 \neq x \leq y = 0$	*	*	T
F	T	F	$0 \neq x > y \neq 0$	*	*	F
F	T	T	x>y=0	*	*	T
T	F	F	0=x<y	*	*	F
T	F	T	x=y=0	*	T	T
T	T	F	0=x>y	*	*	F
T	T	T	impossible	T	T	T

Another Example

```
done = 0;
while(done == 0){
  if(x != 0)
    x--;
  else
    done++;
}
assert(x == 0);
```

How do you argue that the program is correct?

Which predicates do you need to prove that?

Function Calls

```
int y;

f(){
   h(y);
}

void h(int z){
   y = z;
}
```

```
bool b; // y>0

f(){

}

void h(bool c){ // c: z =0

}
```

Function Calls

```
int y;

f(){
   h(y);
}

void h(int z){
   y = z;
}
```

```
bool b; // y>0

f(){
   h(b ? F : *)
}

void h(bool c){ // c: z =0
   b = c ? F : *;
}
```

Abstraction

- Tricky: find the proper abstraction!
  - You use the counterexamples, but how?
  - You can do it by hand
  - You can try to do it automatically
- Automatically finding the proper abstraction cannot always work. Why not?

Precisely: assignment

Original: `x := e`
Predicates p1,…,pn.

Suppose we have

```
{qi}
x := e;
{pi}
```

Let ai be the disjunction of assignments to p1…pn that imply qi.

let bi be the disjunction of assignments to p1…pn that imply ¬qi.

`x := e` is replaced by

```
simultaneous
  p1 = a1 ? T : b1 ? F : *
  …
  pn = an ? T : bn ? F : *
end simultaneous
```

example

Assignment: b := b+1
Predicates: `p1 = {a ≤ b}` and `p2 = {a=b+1}`

```
{a ≤ b + 1}     {a = b + 2}
b := b + 1      b := b + 1
{a ≤ b}         {a = b + 1}
```

Look at the table: row TT, TF, and FT have a T in column a≤b and TT and FF have an F in that column. Therefore:

```
p1 ∨ p2              implies a ≤ b + 1
(p1∧p2)∨(¬p1∧¬p2)  implies a > b + 1
(note: false implies anything)
```

For the 2nd predicate:

```
p1 ∧ p2 implies a = b+2
p1 ∨ ¬p2 implies a ≠ b+2
```

`b:=b+1` is abstracted to

```
simultaneous
 {a≤b}   := p1||p2 ? T : p1==p2 ? F : *
 {a=b+1} := p1&&p2 ? T : p1!=p2 ? F : *
end
```

p1	p2			
a≤b	a=b+1		a≤b+1	a=b+2
T	T	×	T/F	T/F
T	F	a≤b	T	F
F	T	a=b+1	T	F
F	F	a>b+1	F	*

(Cf. same example on an earlier slide)