

## Module 3: Reasoning about Programs and Hardware

What's common to reasoning about these entities?

- Flow-chart programs
- Block-structured programs
- Recursive programs
- Instruction sequence processors
- Abstract data types
- ...

Induction and recursion!

## A very brief tour of some principles of induction

- One can relations inductively (e.g. via recursive programs)
- One can define sets inductively (e.g. via datatype constructors)
- One can then perform inductive reasoning over the “structure” of the recursion.

## Reasoning about flow-chart programs

### The “inductive assertions” method

- Here, we reason about things like “while loops” and “iterative SRT division loops”
- The idea is to separate correctness (of I/O mappings) from termination.
- For each loop, we establish an assertion (called loop invariant, LI) via induction.
  - After zero iterations (upon entry to the loop) LI is true.
  - Assuming that something “weaker than or equal to LI” (denoted  $wp(\text{loopbody}, LI)$ ) is true at the beginning of the n-th iteration, we prove that LI is true at the conclusion of the n-th iteration.
  - Thus by induction, for all iterations, the assertion is preserved.
- Termination is shown by arguing that each iteration brings down a “measure” and when the measure hits a min value, we exit.

## Reasoning about recursive programs

### Subgoal induction

- Again we separate correctness (of I/O mappings) from termination.
- We argue that all basis cases when recursion terminates result in the correct I/O behavior.
- For all recursive cases, by induction hypothesis, we assume that all “calls” made on the RHS behave correctly, and argue that the LHS recursive call also works correctly.
- There is no explicit LI, but its role is played by the correctness assertion for the whole program.

## Reasoning about instruction-set processors

### Establishing the simulation relation inductively

- We define correctness as follows: anything that the implementation does is according to specification.
  - “Liveness” (that the implementation does anything at all!) is often separately established.
  - Of course, one can seek bisimulation - one of the strongest reasonable agreements where the comparison goes both ways.
- To establish simulation, we induct over “time”
  - Initially the systems agree - e.g. have the same observables (e.g. register/memory set)
  - Whenever they have the same observables and are operating in a reasonable region of the state-space, all instructions leave them looking the same.

One can do structural induction  
to reason over datatypes

- We can inductively define sets; example:
  - `nil` is a list
  - If `L` is a list and `a` is an atom, so is `cons(a,L)`.
  - Nothing else is a list.
- Inductive definition of sets through inference rules:

Define type `List` as the minimal set defined  
by the following rules:

-----	L is a list
nil is a List	cons(a,L) is a list

## Free inductive definitions, and “ADT verification”

- They do not allow non-trivial semantic interpretations for the constructors.
  - So for instance, there is no rule such as  $\text{cons}(a, \text{cons}(a, \text{cons}(a, \text{nil}))) = \text{nil}$ .
- If such “non-free” rules are allowed, it is easy to introduce contradictions.

`length(nil) = 0`  
`length(cons(a,L)) = 1 + length(L)`  
With non-free definitions, length isn't well defined.

- If we allow only free inductive definitions, then recursively defined functions are well-defined (contradiction-free).
- Also such sets are well-founded. Then, any total predicate  $\phi$  can be proven over such sets using structural induction:

*if*  $(\forall a \in S) \{[(\forall b \in S \text{ s.t. } b \prec a) \phi(b)] \Rightarrow \phi(a)\}$   
*then*  $(\forall c \in S) \phi(c)$

## Organization

- Basics of Floyd's method (flow-chart program verification)
  - Flow-charts were despised in programming...
  - ...but high-level finite-state machines are like flow-charts!
- The Floyd/Hoare logic (for “while” programs)
- Subgoal induction (for recursive programs)
- How to induct over instruction sequences?
  - Abstract data types
  - “Commute diagrams”
  - Layering proofs via *completion functions*

## A taste of Floyd's method - XOR swap

This program swaps bit-vectors

```
begin
  A := A xor B ;
  B := A xor B ;
  A := A xor B ;
end
```

If the above is run in a state

{ A = a and B = b }

it will terminate in state

{ A = b and B = a }

## The “wp” rule

- *Weakest pre-condition* captures the largest number of states from which one could execute a command to end-up in a desired postcondition.
- The wp of  $x := E$  with respect to post-condition  $P$  is  $P [ E / x ]$ .
- $P [ E / x ] \ x := E \ P$
- A weakest precondition is one where if  $\{ Q \} \ x := E \ \{ P \}$  for some other precondition  $Q$ , then  $Q \Rightarrow P[E/x]$  .
  - This is the assertion  $P$ , except where  $x$  is mentioned, one must use  $E$ .
- This rule “works” because the assertion  $P$  is being made with respect to the *new* value of  $x$ .
- Therefore, to say something that relates to  $P$ , we must use  $E$  in place of  $x$  because it is  $E$ 's value in the old state that  $x$  assumes in the new state.

## Why do things backwards??

The forward rule for assignment is

```
{ P }  
x := E  
{ exists oldx . P [ oldx/x ] and (x = E [ oldx/x ]) }
```

- This is kind of ugly and not goal-directed. Hence we prefer backward rules.
- As the name suggests, `oldx` stands for the old value of `x` that got destroyed upon assignment.

## XOR swap proof

- Annotate the program with the pre- and post conditions

```
{ A = a and B = b }  
  begin  
    A := A xor B ;  
    B := A xor B ;  
    A := A xor B ;  
  end  
{ A = b and B = a }
```

- Calculate the *weakest precondition* of each statement with respect to its postcondition. Ensure that the final inferred wp is implied by the precondition.

```
{ A = a and B = b }
```

```
  || Implication  
  \/  

```

```
(5) { ((A xor B) xor ((A xor B) xor B)) = b  
      and  
      ((A xor B) xor B) = a }
```

```

begin
(4)  { ((A xor B) xor ((A xor B) xor B)) = b
      and
      ((A xor B) xor B) = a }

      A := A xor B ;

(3)  { (A xor (A xor B)) = b and (A xor B) = a }
      B := A xor B ;

(2)  { (A xor B) = b and B = a }
      A := A xor B ;

(1)  { A = b and B = a }
      end
      { A = b and B = a }

```

## Assertions, sets of states, invariants

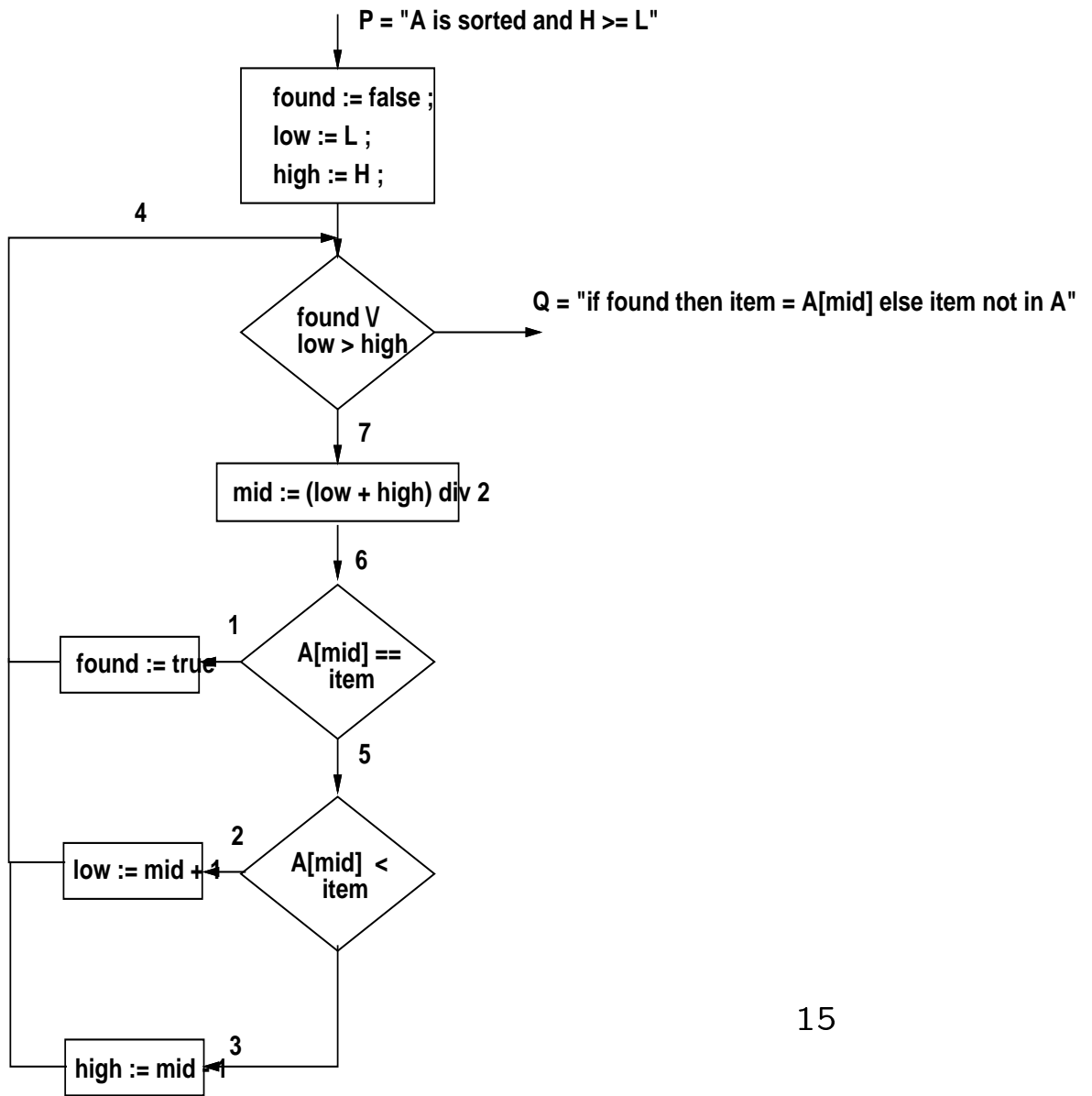
- Correctness assertions capture a set of program states
- TRUE: all states (all variable settings)
- FALSE: no state (no variable setting at all)
- $x=0$ : every program state in which  $x$  is set to 0 (all other variables taking all other possible values)
- Let  $[[ P ]]$  denote the set of states denoted by assertion  $P$ ; then
  - $P \Rightarrow Q$  IFF  $[[ P ]]$   $\subseteq$   $[[ Q ]]$
- Program-point Invariants: Assertions found true whenever execution reaches that program point.

## The “if then else” rule

In a similar vein, the rules for an if-then-else as well as for a “single-armed” if statement are as follows.

$$\frac{\begin{array}{l} |- \{ P \wedge B \} C1 \{ Q \} , \quad |- \{ P \wedge \text{not}(B) \} C2 \{ Q \} \\ \hline \end{array}}{|- \{ P \} \quad \text{if } B \text{ then } C1 \text{ else } C2 \{ Q \}}$$
$$\frac{\begin{array}{l} |- \{ P \wedge B \} C \{ Q \} , \quad |- P \wedge \text{not}(B) \Rightarrow Q \\ \hline \end{array}}{|- \{ P \} \quad \text{if } B \text{ then } C \{ Q \}}$$

Let's now consider an example: The Binary Search program (see Jon Bentley's book “Programming Pearls.”)

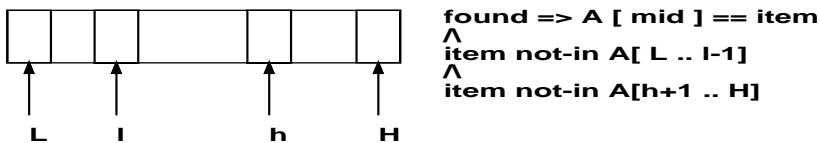


## Steps in Floyd's method

1. Annotate each entry-point and exit-point with pre- and post conditions.
2. Cut each loop with a cut-point and attach an **invariant** there. The invariant must be true **whenever** the execution reaches the loop cut-point.
3. Argue about each linear execution path that results from the above using the assignment *wp* (weakest precondition) rule and the if-then-else *wp* rules.

What does a loop invariant try to capture?  
How do we obtain one?

- A loop invariant captures the relationships among the program variables in a very general manner. E.g., see if the following suffices. The loop invariant also has to be “sufficiently informative” to conclude the “final answer.”



- Generally it takes a few tries to obtain a loop invariant (because some aspect of some path originating and leading back to the cut-point may have been forgotten).
- Automatic invariant generation methods have considerably improved in recent years - and can often provide the “seed” of the “ultimate invariant.”

## Examples of invariants

What is a loop invariant of these programs?

```
/* Prog1 */  
cnt = 4'b0000  
while (1)  
    cnt = ( {(cnt[1] ^ cnt[0]),3'b000} | (cnt>>1) );
```

```
/* Prog2 */  
cnt = 4'b0001  
while (1)  
    cnt = ( {(cnt[1] ^ cnt[0]),3'b000} | (cnt>>1) );
```

## A “paper and pencil” proof (semi-formal)

The first step is to show that the purported loop invariant  $LI = (f \Rightarrow A[m]=i) \wedge (i \text{ notin } A[L..l-1, h+1..H])$  is indeed one. To prove this, examine the following paths:

### 1. Path 4,1,6,7,4:

- At 1:  
 $A[m]=i \wedge (i \text{ notin } A[L..l-1, h+1..H])$
- At 6:  
 $A[m]=i \Rightarrow (A[m]=i \wedge (i \text{ notin } A[L..l-1, h+1..H]))$   
which simplifies to  
 $A[m]=i \Rightarrow \wedge (i \text{ notin } A[L..l-1, h+1..H])$
- At 7:  
 $A[(l+h)/2]=i \Rightarrow (i \text{ notin } A[L..l-1, h+1..H])$
- At 4:  
 $(f \wedge l \leq h) \Rightarrow$

$(A[(l+h)/2]=i \Rightarrow (i \notin A[L..l-1, h+1..H]))$

- To show that

LI  $\Rightarrow$

$( ( f \wedge l \leq h ) \Rightarrow$

$(A[(l+h)/2]=i \Rightarrow (i \notin A[L..l-1, h+1..H]))$

Do a case analysis on  $f$ .

– Case  $f=false$ :

$(i \notin A[L..l-1, h+1..H]) \Rightarrow$

$(l \leq h) \Rightarrow$

$(A[(l+h)/2]=i \Rightarrow (i \notin A[L..l-1, h+1..H]))$

which is true

– Case  $f=true$ :

LI  $\Rightarrow$

$false \Rightarrow$

$(A[(l+h)/2]=i \Rightarrow (i \notin A[L..l-1, h+1..H]))$

which is also true.

2. Path 4,2,5,6,7,4:

- At 2:  
 $(f \Rightarrow A[m]=i) \wedge (i \notin A[L..m, h+1..H])$
- At 5:  
 $A[m] < i \Rightarrow$   
 $(f \Rightarrow A[m]=i) \wedge (i \notin A[L..m, h+1..H])$
- At 6: same as above because  $A[m] < i \Rightarrow$   
means  $A[m] \neq i$ .
- At 7:  
 $A[(1+h)/2] < i \Rightarrow$   
 $(f \Rightarrow A[(1+h)/2]=i) \wedge (i \notin A[L..(1+h)/2,$   
 $h+1..H])$
- At 4:  
 $f \wedge l \leq h$   
 $A[(1+h)/2] < i \Rightarrow$

$(f \Rightarrow A[(1+h)/2]=i) \wedge (i \notin A[L..(1+h)/2, h+1..H])$

To show that LI  $\Rightarrow$  above.

Do a case analysis on  $f$ .

– Case  $f=false$ :

$(i \notin A[L..l-1, h+1..H]) \Rightarrow$

$(l \leq h) \Rightarrow$

$A[(1+h)/2] < i \Rightarrow$

$(i \notin A[L..(1+h)/2, h+1..H])$

The fact that  $i \notin A[L..(1+h)/2]$  follows from

$A[(1+h)/2] < i \Rightarrow$  and that  $A$  is sorted.

The fact that  $i \notin A[h+1..H]$  follows from the antecedent.

3. In the same fashion, establish that LI is an invariant on path 4,3,5,6,7,4.

4. LI is true upon initial entry because  $\text{wp}(\text{LI}, (f := \text{false}; l := L; h := H))$  reduces to  $(i \text{ notin } A[L..L-1, H+1..H])$ , which is true because the ranges are empty.
5. The output assertion  $Q$  is established upon exit because  $\text{wp}(Q, f \setminus / l > h)$  is

$f \setminus / l > h) \Rightarrow$   
 $(f \rightarrow A[m]=i, i \text{ notin } A).$

LI must imply the above. *i.e.*,

$(f \Rightarrow A[m]=i) \wedge (i \text{ notin } A[L..l-1, h+1..H]) \Rightarrow$   
 $f \setminus / l > h) \Rightarrow$   
 $(f \rightarrow A[m]=i, i \text{ notin } A).$

Case analysis on  $f$ :

- Case  $f$  false:  
 $(i \text{ notin } A[L..l-1, h+1..H]) \Rightarrow$   
 $l > h) \Rightarrow$   
 $i \text{ notin } A$

which is true because if  $l > h$ , then  $l-1 \geq h$ , and hence the whole range is covered by the antecedent.

- Case  $f$  true:  
 $(A[m]=i) \wedge (i \text{ notin } A[L..l-1, h+1..H]) \Rightarrow$   
 $\text{true} \Rightarrow$   
 $(A[m]=i)$

which is true.

6. Hence the binary search routine is *partially correct* (means if it halts, it delivers the correct answer).

About “total correctness” (partially correct  
and it terminates)

Total correctness = partial correctness + termination.

For example, no one knew the status of the following total correctness assertion about the “Fermat program” till Andrew Wiles proved that the total correctness assertion is indeed true.

[true] *if*  $(\exists x, y, z : nat . \exists n : nat . (n > 2) \wedge x^n + y^n = z^n)$   
*then loop else halt* [true]

On the other hand, no one knows whether the following total correctness assertion is true or not (unless you want to hole-up in your attic room for 8 years and prove it, as Wiles did..)

[true]

```
while (x > 1) do  
  if odd(x) then x := 3x+1 else x := x/2 endif
```

[true]

On the other hand, the following total correctness assertion is known to be true:

[false]

```
while (x > 1) do  
  if odd(x) then x := 3x+1 else x := x/2 endif
```

[anything]

whereas, it is known that the following total correctness assertion is false

[true]

```
while (x > 1) do
  if odd(x) then x := 3x+1 else x := x/2 endif
```

[false]

while it wasn't known until Wiles's proof whether the following was true or not!

[true] *if*  $(\exists x, y, z : nat . \exists n : nat . (n > 2) \wedge x^n + y^n = z^n)$   
*then loop else halt* [false]

For Binary search:

1. To argue that the binary search routine terminates, notice that  $n-1$  diminishes in each iteration, and so the loop exit condition eventually holds. This is usually termed the *well-founded measure* for the loop.
2. Hence the binary search routine is *totally correct* (partial correctness plus termination equals total correctness).

## The Hoare-calculus rules for block-structured programs

$$\vdash \{ P \wedge B \} C \{ P \}$$

---

$$\vdash \{ P \} \text{ while } B \text{ do } C \text{ end } \{ P \wedge \sim B \}$$

```
>(verify '(SPEC
  T
  (BLOCK
    (ASSIGN R X)
    (ASSIGN Q 0)
    (ASSERT (( R = X) and (Q = 0)))
    (WHILE (Y <= R)
      (ASSERT (X = (R + (Y * Q))))
      (BLOCK (ASSIGN R (R - Y))
              (ASSIGN Q (Q + 1))))
    ((R < Y) and (X = (R + (Y * Q))))))
```

```
Checking syntax of annotated program .....
OK.
```

Trying to prove ((T IMPLIES ((X = X) AND (O = O))))

.....

Assignment Axiom

T1: |- {T} (ASSIGN R X) {((R = X) AND (O = O))}

Trying to prove (((X = (R + (Y \* Q))) AND (Y <= R)) IMPLIES  
(X = ((R - Y) + (Y \* (Q + 1)))))

.....

Assignment Axiom

T2: |- {((X = (R + (Y \* Q))) AND (Y <= R))} (ASSIGN R (R - Y))  
{(X = (R + (Y \* (Q + 1))))}

By Block rule

T3: T2 |- {((X = (R + (Y \* Q))) AND (Y <= R))}  
(BLOCK  
(ASSIGN R (R - Y))  
(ASSIGN Q (Q + 1)))  
{(X = (R + (Y \* Q)))}

Trying to prove (((R = X) AND (Q = O)) IMPLIES (X = (R + (Y \* Q))))  
(((X = (R + (Y \* Q))) AND (NOT (Y <= R))) IMPLIES  
(R < Y) AND (X = (R + (Y \* Q))))

.....

By While rule

```

T4:  T3 |- {((R = X) AND (Q = 0))} (WHILE (Y <= R)
                                     (ASSERT (X = (R + (Y * Q))))
                                     (BLOCK (ASSIGN R (R - Y))
                                             (ASSIGN Q (Q + 1))))
      {((R < Y) AND (X = (R + (Y * Q))))}

```

By Block rule

```

T5:  T1 T4 |- {T} (BLOCK (ASSIGN R X)
                        (ASSIGN Q 0)
                        (ASSERT ((R = X) AND (Q = 0)))
                        (WHILE (Y <= R) (ASSERT (X = (R + (Y * Q))))
                                (BLOCK (ASSIGN R (R - Y))
                                        (ASSIGN Q (Q + 1))))
                        {((R < Y) AND (X = (R + (Y * Q))))}

```

all proved.

T