Hoare Logic

Hoare Logic is used to reason about the correctness of programs. In the end, it reduces a program and its specification to a set of verifications conditions.

Slides by Wishnu Prasetya
URL : www.cs.uu.nl/~wishnu
Course URL : www.cs.uu.nl/docs/vakken/pc

Overview

- Hoare triples
- Basic statements // SEQ, IF, ASG
  - Composition rules for seq and if
  - Assignment
  - Weakest pre-condition
- Loops // WHILE
  - Invariants
  - Variants

Hoare triples

How do we prove our claims ?

- In Hoare logic we use inference rules.
- Usually of this form:

premise-1 , premise-2 , …
----
conclusion

- A proof is essentially just a series of invocations of inference rules, that produces our claim from known facts and assumptions.

Needed notions

- Inference rule:

$$\frac{\{\,P\,\}\ \ S\ \ \{\,Q\,\}\quad ,\quad Q \Rightarrow R}{\{\,P\,\}\ S\ \{\,R\,\}}$$

  is this sound?

- What does a specification mean ?
- Programs
- Predicates
- States

We'll explain this in term of abstract models.

State

- In the sequel we will consider a program P with two variables: x:int , y:int.
- The state of P is determined by the value of x,y. Use record to denote a state:

  { x=0 , y=9 }   // denote state where x=0 and y=9

- This notion of state is abstract! Actual state of P may consists of the value of CPU registers, stacks etc.
- Σ denotes the space of all possible states of P.

Expression

- An expression can be seen as a function $\Sigma \to$ val

x + 1 { x=0 , y = 9 }	yields	1
x + 1 { x=9 , y = 9 }	yields	10
etc		

- A (state) predicate is an expression that returns a boolean:

x>0 { x=0 , y = 9 }	yields	false
x>0 { x=9 , y = 9 }	yields	true
etc		

Viewing predicate as set

- So, a (state) predicate P is a function $\Sigma \to$ bool. It induces a set:

$$\chi_P = \{ s \mid s \models P \}$$ // the set of all states satisfying P

- P and its induced set is 'isomorphic' :

$$P(s) = s \in \chi_P$$

- Ehm … so for convenience lets just overload "P" to also denote χ_P. Which one is meant, depends on the context.

- Eg. when we say "P is an empty predicate".

Implication

- $P \Rightarrow Q$ // $P \Rightarrow Q$ is valid

  This means: $\forall s.\ s \models P \Rightarrow s \models Q$

  In terms of set this is equivalent to: $\chi_P \subseteq \chi_Q$

- And to confuse you ☺, the often used jargon:
  - P is <u>stronger</u> than Q
  - Q is <u>weaker</u> than P
  - Observe that in term of sets, stronger means smaller!

Non-termination

- What does this mean?

  s Pr s' stands for (Pr,s)→ s'

  $\{s' \mid s\ Pr\ s'\} = \emptyset$, for some state s

- Can be used to model: "Pr does not terminate when executed on s".

- However, in discussion about models, we usually assume that our programs terminate.

- Expressing non-termination leads to some additional complications → not really where we want to focus now.

Hoare triples

- Now we have enough to define abstractly what a specification means:

s Pr s' stands for (Pr,s)→ s'

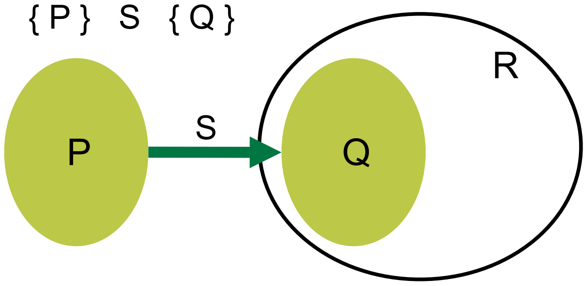
$$\{\ P\ \}\ \ Pr\ \ \{\ Q\ \}\ \ = \\ (\forall s.\ \ s \models P\ \Rightarrow\ (\forall s'.\ s\ Pr\ s' \Rightarrow s' \models Q))$$

- Since our model cannot express non-termination, we assume that Pr terminates.

- The interpretation of Hoare triple where termination is assumed is called "partial correctness" interpretation.
- Otherwise it is called total correctness.

Now we can explain …

{ P } S { Q }



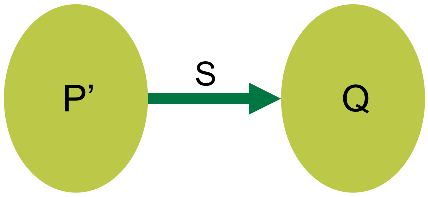
{ P } S { R }

Post-condition weakening Rule:

$$\frac{\{P\}\ \ S\ \ \{Q\}\ \ ,\ Q \Rightarrow R}{\{P\}\ S\ \{R\}}$$

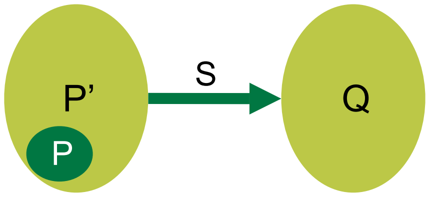
And the dual

{ P’ } S { Q }



And the dual

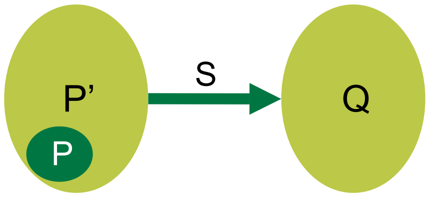
{ P’ } S { Q }



{ P } S { Q }

And the dual

$\{ P' \} \; S \; \{ Q \}$



$\{ P \} \; S \; \{ Q \}$

Pre-condition strengthening Rule:

$$\frac{P \Rightarrow P' \;,\; \{ P' \} \;\; S \;\; \{ Q \}}{\{ P \} \; S \; \{ Q \}}$$

Joining specifications

- Conjunction:

$$\frac{\{ P_1 \} \;\; S \;\; \{ Q_1 \} \quad , \quad \{ P_2 \} \; S \; \{ Q_2 \}}{\{ P_1 \wedge P_2 \} \;\; S \;\; \{ Q_1 \wedge Q_2 \}}$$

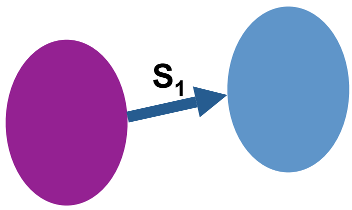
- Disjunction:

$$\frac{\{ P_1 \} \;\; S \;\; \{ Q_1 \} \quad , \quad \{ P_2 \} \; S \; \{ Q_2 \}}{\{ P_1 \vee P_2 \} \;\; S \;\; \{ Q_1 \vee Q_2 \}}$$

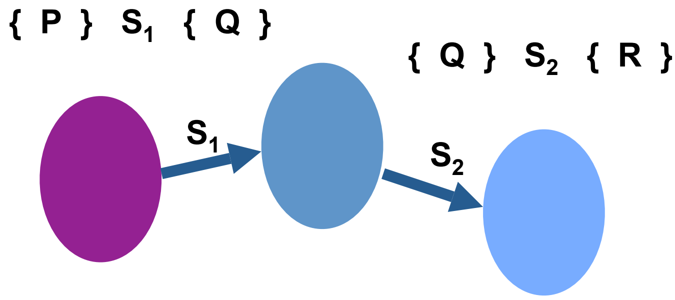
Reasoning about basic statements

Rule for SEQ composition

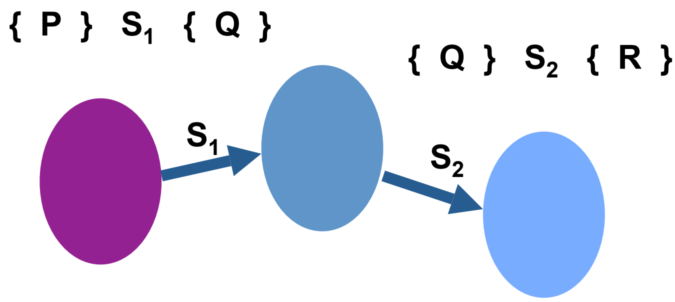
{ P } S_1 { Q }



Rule for SEQ composition

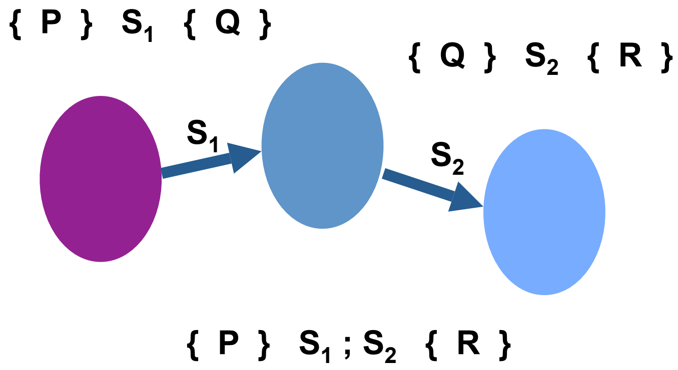


Rule for SEQ composition



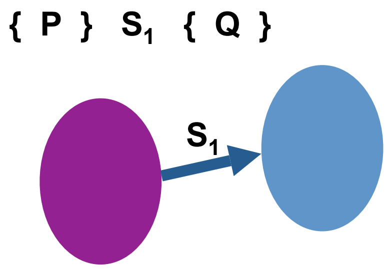
$\{ P \}\ S_1 ; S_2\ \{ R \}$

Rule for SEQ composition



$$\frac{\{ P \}\ S_1\ \{ Q \}\ ,\ \{ Q \}\ S_2\ \{ R \}}{\{ P \}\ S_1 ; S_2\ \{ R \}}$$

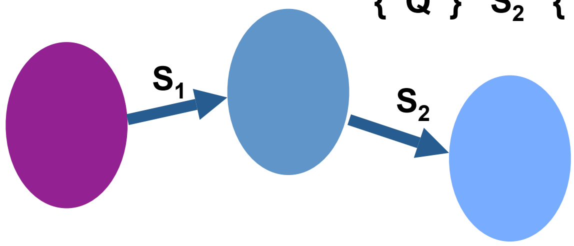
Rule for SEQ composition



Rule for SEQ composition

$\{ P \}\ S_1\ \{ Q \}$

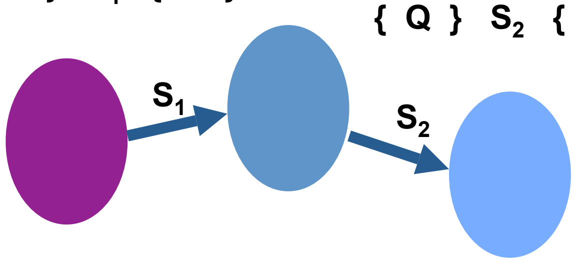
$\{ Q \}\ S_2\ \{ R \}$



Rule for SEQ composition

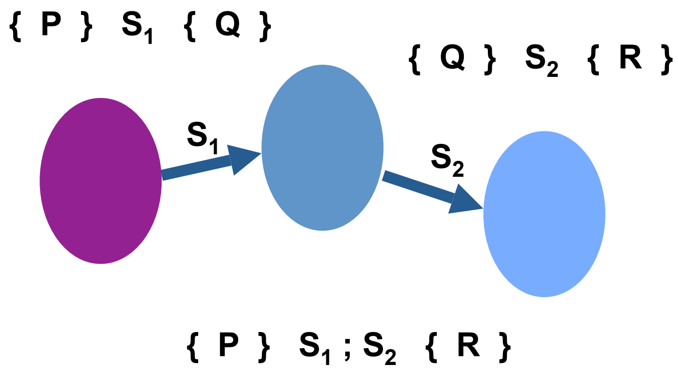
$\{ P \}\ S_1\ \{ Q \}$

$\{ Q \}\ S_2\ \{ R \}$



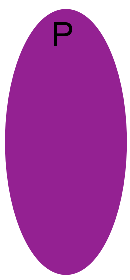
$\{ P \}\ S_1 ; S_2\ \{ R \}$

Rule for SEQ composition

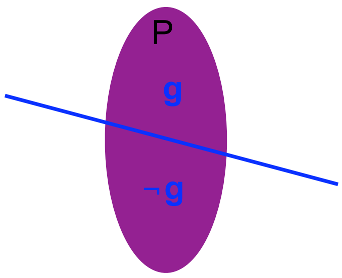


$$\frac{\{\ P\ \}\ S_1\ \{\ Q\ \}\quad ,\quad \{\ Q\ \}\ S_2\ \{\ R\ \}}{\{\ P\ \}\ S_1 ; S_2\ \{\ R\ \}}$$

Rule for IF

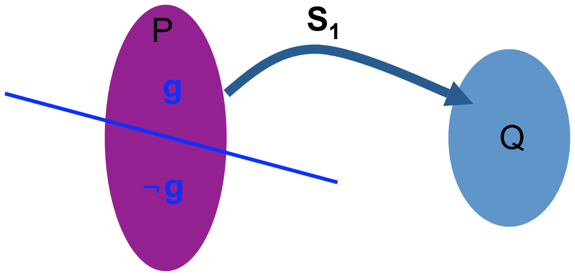


Rule for IF



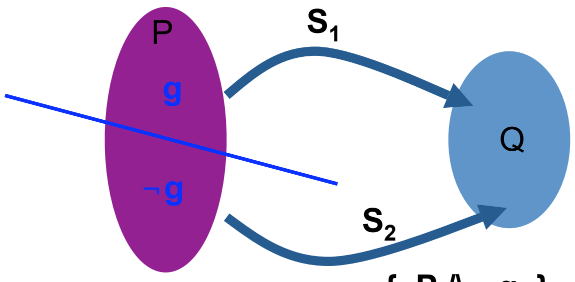
Rule for IF

$\{ P \wedge g \} \; S_1 \; \{ Q \}$



Rule for IF

{ $P \wedge g$ } S_1 { Q }

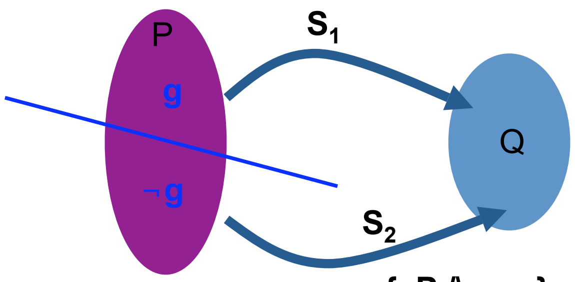


{ $P \wedge \neg g$ } S_2 { Q }

Rule for IF

{ $P \wedge g$ } S_1 { Q }

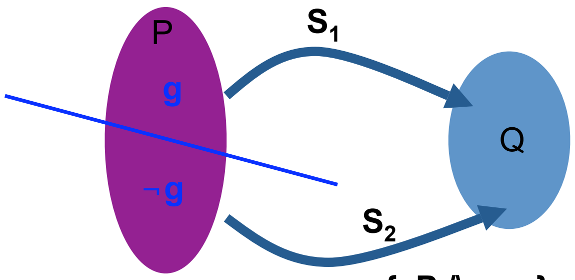
{ P } <u>if</u> g <u>then</u> S_1 <u>else</u> S_2 { Q }



{ $P \wedge \neg g$ } S_2 { Q }

Rule for IF

$\{ P \wedge g \}\ S_1\ \{ Q \}$



$\{ P \}$ <u>if</u> g <u>then</u> S_1 <u>else</u> S_2 $\{ Q \}$

$\{ P \wedge \neg g \}\ S_2\ \{ Q \}$

$$\frac{\{ P \wedge g \}\ S_1\ \{ Q \} \quad , \quad \{ P \wedge \neg g \}\ S_2\ \{ Q \}}{\{ P \}\ \underline{\text{if}}\ g\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \{ Q \}}$$

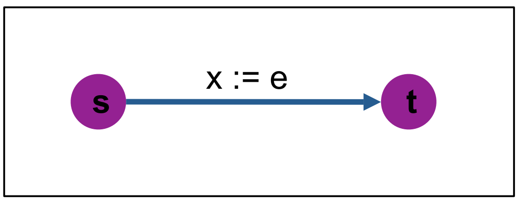
Rule for Assignment

- Let see ….
-

$$\frac{??}{\{ P \}\ x:=e\ \{ Q \}}$$

- Find a pre-condition W, such that, for any begin state s, and end state t:

$s \models W \quad \Leftrightarrow \quad t \models Q$



- Then we can equivalently prove $P \Rightarrow W$

Assignment, examples

- { 10 = y }   x:=10   {   x=y   }

- { x+a = y }   x:=x+a   {   x=y   }

- So, W can be obtained by Q[e/x]

Assignment

- Theorem:

    Q holds after x:=e iff Q[e/x] holds before the assignment.

- Express this indirectly by:

    { P } x:=e { Q }   = $P \Rightarrow Q[e/x]$

- Corollary:

    { Q[e/x] }   x:=e   { Q }   always valid.

How does a proof proceed now ?

How does a proof proceed now ?

- { $x \neq y$ }   tmp:= x ; x:=y ; y:=tmp   { $x \neq y$ }

How does a proof proceed now ?

- { x≠y } tmp:= x ; x:=y ; y:=tmp { x≠y }

- Rule for SEQ requires you to come up with intermediate assertions:

  { x≠y } tmp:= x { ? } ; x:=y { ? } ; y:=tmp { x≠y }

- What to fill ??
  - Use the "Q[e/x]" suggested by the ASG theorem.
  - Work in reverse direction.
  - "Weakest pre-condition"

Weakest Pre-condition (wp)

- "wp" is a meta function:

  wp : Stmt X Pred → Pred

- wp(S,Q) gives the weakest (largest) pre-cond such that executing S in any state in any state in this pre-cond results in states in Q.
  - Partial correctness → termination assumed
  - Total correctness → termination demanded

Weakest pre-condition

- Let W = wp(S,Q)
- Two properties of W
  - Reachability: from any s|=W, if s S s' then s' |= Q
  - Maximality: s S s' and s' |= Q implies s|=W

s S s' stands for (S,s)→ s'

Defining wp

- In terms of our abstract model:

  wp(S,Q) = { s | forall s'. s S s' implies s' |= Q }

- Abstract characterization:

  $\boxed{\{\ P\ \}\ S\ \{\ Q\ \} \quad = \quad P \Rightarrow wp(S,Q)}$

- Nice, but this is not a constructive definition (does not tell us how to actually construct "W")

Some examples

- All these pre-conditions are the weakest:
- { y=10 }   x:=10   { y=x }
- { Q }   skip   { Q }
- { Q[e/x] }   x:=e   { Q }

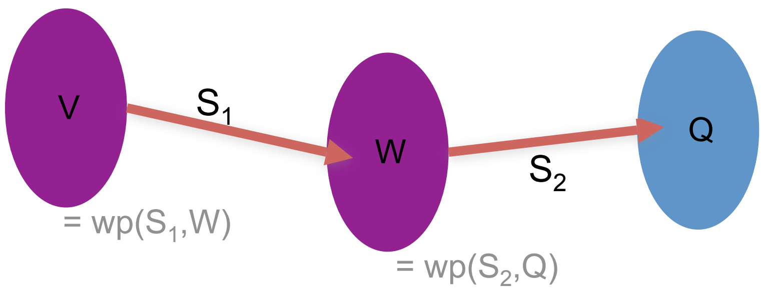
Some examples

- All these pre-conditions are the weakest:
- { y=10 }   x:=10   { y=x }
- { Q }   skip   { Q }
- { Q[e/x] }   x:=e   { Q }

wp	skip Q	=	Q
wp	(x:=e) Q	=	Q[e/x]

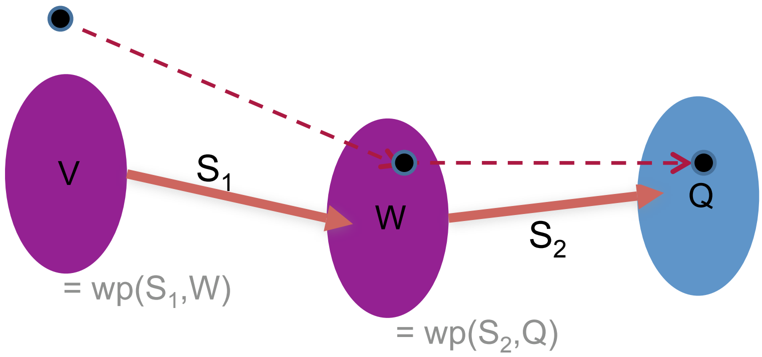
wp of SEQ

$$wp\ ((S_1\ ;\ S_2)\ ,Q) \quad = \quad wp(S_1\ ,\ (wp(S_2,Q)))$$



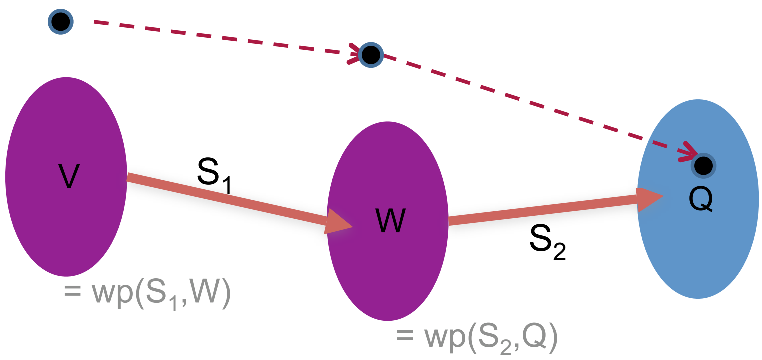
wp of SEQ

$$wp\ ((S_1\ ;\ S_2)\ ,Q) \quad = \quad wp(S_1\ ,\ (wp(S_2,Q)))$$



wp of SEQ

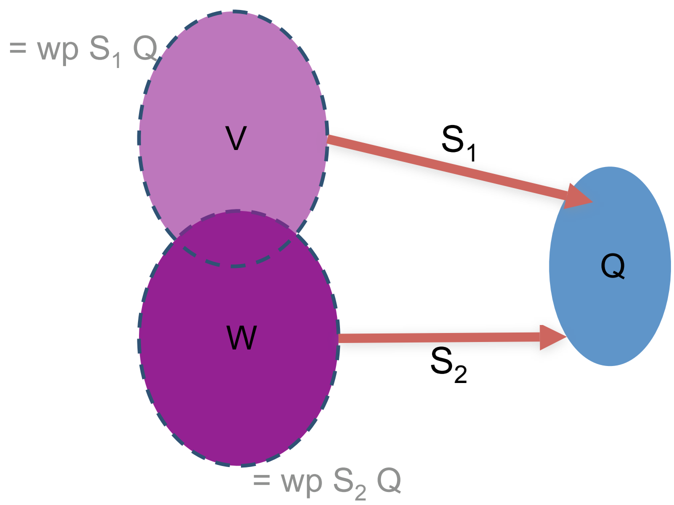
$$wp\ ((S_1\ ;\ S_2)\ ,Q) \quad = \quad wp(S_1\ ,\ (wp(S_2,Q)))$$



wp of IF

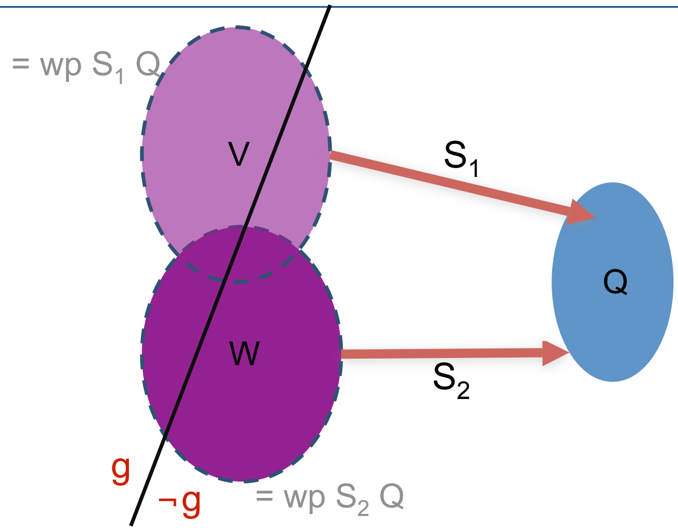
wp((<u>if</u> g <u>then</u> S_1 <u>else</u> S_2),Q) =

$g \wedge wp(S_1,Q) \vee \neg g \wedge wp\ (S_2,Q)$



wp of IF

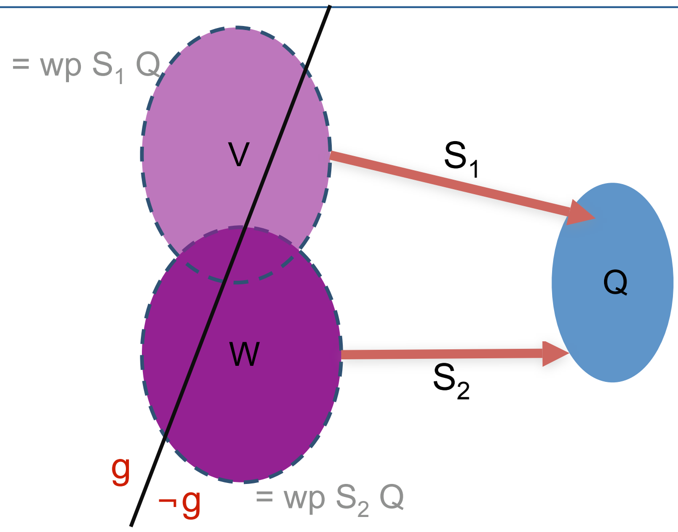
wp((<u>if</u> g <u>then</u> S_1 <u>else</u> S_2),Q) =
g $\wedge$ wp(S_1,Q) $\vee \neg$g $\wedge$ wp (S_2,Q)



wp of IF

wp((<u>if</u> g <u>then</u> S_1 <u>else</u> S_2),Q) =
g $\wedge$ wp(S_1,Q) $\vee \neg$g $\wedge$ wp (S_2,Q)

Other formulation :

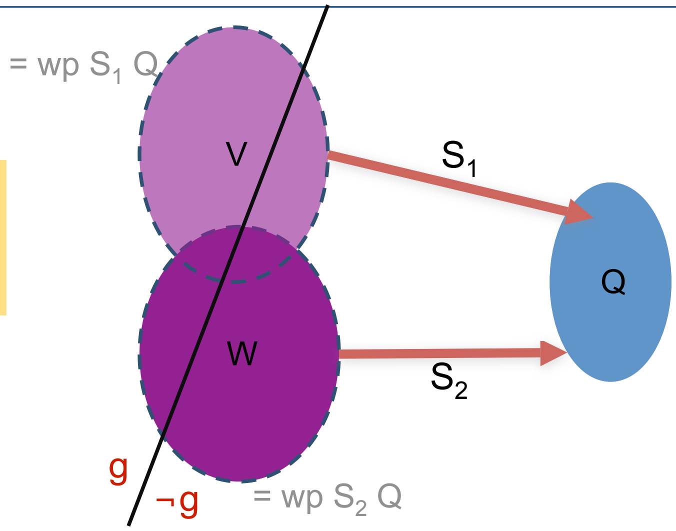


wp of IF

wp((<u>if</u> g <u>then</u> S_1 <u>else</u> S_2),Q) =
g $\wedge$ wp(S_1,Q) $\vee \neg$g $\wedge$ wp (S_2,Q)

Other formulation :

(g $\Rightarrow$ wp (S_1 ,Q))
$\wedge$
($\neg$g $\Rightarrow$ wp (S_2 ,Q))



Proof: homework ☺

How does a proof proceed now ?

How does a proof proceed now ?

- { x≠y }   tmp:= x ; x:=y ; y:=tmp   { x≠y }

How does a proof proceed now ?

- { x≠y }   tmp:= x ; x:=y ; y:=tmp   { x≠y }

- Calculate:

  W = wp((tmp:= x ; x:=y ; y:=tmp) , x≠y)

How does a proof proceed now ?

- { x≠y }   tmp:= x ; x:=y ; y:=tmp   { x≠y }

- Calculate:

  W = wp((tmp:= x ; x:=y ; y:=tmp) , x≠y)

How does a proof proceed now ?

- { x≠y }   tmp:= x ; x:=y ; y:=tmp   { x≠y }

- Calculate:

  W = wp((tmp:= x ; x:=y ; y:=tmp) , x≠y)

- Then prove: x≠y ⇒ W

How does a proof proceed now ?

- { $x \neq y$ } tmp:= x ; x:=y ; y:=tmp { $x \neq y$ }

- Calculate:

  W = wp((tmp:= x ; x:=y ; y:=tmp) , $x \neq y$)

- Then prove: $x \neq y \Rightarrow W$

- We <u>calculate</u> the intermediate assertions, rather than figuring them out by hand!

Proof via wp

- Wp calculation is *fully syntax driven*. *(But no while yet!)*
  - No human intelligence needed.
  - Can be automated.
- Works, as long as we can calculate "wp" → not always possible.
- Recall this abstract def:

  { P } S { Q } = $P \Rightarrow$ wp(S,Q)

  It follows: if $P \Rightarrow W$ <u>not valid</u>, then so does the original spec.

Proof via wp

- Wp calculation is *fully syntax driven*. *(But no while yet!)*
  - No human intelligence needed.
  - Can be automated.
- Works, as long as we can calculate "wp" → not always possible.
- Recall this abstract def:

W

$$\{\ P\ \}\ \ S\ \ \{\ Q\ \}\quad = \quad P \Rightarrow wp(S,Q)$$

It follows: if $P \Rightarrow W$ <u>not valid</u>, then so does the original spec.

Example

```
bool find(a,n,x) {

    int i = 0 ;
    bool found = false ;

    while (¬found /\ i<n)  {

        found :=  a[i]=x ;
        i++

    }
    return found ;
}
```

Example

```
bool find(a,n,x) {

    int i = 0 ;
    bool found = false ;

    while (¬found /\ i<n)  {

        found :=  a[i]=x ;
        i++

    }
    return found ;
}
```

found = (∃k : 0≤k<n : a[k]=x)

Example

```
bool find(a,n,x) {

    int i = 0 ;
    bool found = false ;

    while (¬found /\ i<n)  {

        found :=  a[i]=x ;
        i++

    }
    return found ;
}
```

found = (∃k : 0≤k<i : a[k]=x)

found = (∃k : 0≤k<n : a[k]=x)

Example

```
bool find(a,n,x) {

    int i = 0 ;
    bool found = false ;

    while (¬found /\ i<n)  {

        found :=  a[i]=x ;
        i++

    }
    return found ;
}
```

found = (∃k : 0≤k<i : a[k]=x)

found = (∃k : 0≤k<n : a[k]=x)

Example

```
bool find(a,n,x) {

    int i = 0 ;
    bool found = false ;

    while (¬found /\ i<n)  {

        found :=  a[i]=x ;
        i++

    }
    return found ;
}
```

found = (∃k : 0≤k<i : a[k]=x)

found = (∃k : 0≤k<n : a[k]=x)

Example

```
bool find(a,n,x) {

    int i = 0 ;
    bool found = false ;

    while (¬found /\ i<n)  {
                                       found = (∃k : 0≤k<i :  a[k]=x)
        found :=  a[i]=x ;
        i++
                                       found = (∃k : 0≤k<i :  a[k]=x)
    }
                                       found = (∃k : 0≤k<n :  a[k]=x)
    return found ;
}
```


Example

{ ¬found /\ ... /\ (found = (∃k : 0≤k<i : a[k]=x)) }

found := a[i]=x ;

i:=i+1

{ found = (∃k : 0≤k<i : a[k]=x) }

Example

{ ¬found /\ ... /\ (found = (∃k : 0≤k<i : a[k]=x)) }

found := a[i]=x ;

i:=i+1

{ found = (∃k : 0≤k<i : a[k]=x) }

Example

{ ¬found /\ ... /\ (found = (∃k : 0≤k<i : a[k]=x)) }

found := a[i]=x ;

i:=i+1

found = (∃k : 0≤k<i+1 : a[k]=x)

{ found = (∃k : 0≤k<i : a[k]=x) }

Example

{ ¬found /\ ... /\ (found = (∃k : 0≤k<i : a[k]=x)) }

found := a[i]=x ;

i:=i+1

found = (∃k : 0≤k<i+1 : a[k]=x)

{ found = (∃k : 0≤k<i : a[k]=x) }

wp (x:=e) Q = Q[e/x]

Example

{ ¬found /\ ... /\ (found = (∃k : 0≤k<i : a[k]=x)) }

found := a[i]=x ;

i:=i+1

found = (∃k : 0≤k<i+1 : a[k]=x)

{ found = (∃k : 0≤k<i : a[k]=x) }

wp (x:=e) Q = Q[e/x]

Example

$\{\ \neg found\ \wedge\ \ldots\ \wedge\ (found\ =\ (\exists k : 0 \le k < i :\ a[k]=x))\ \}$

found := a[i]=x ;

$(a[i]=x)\ =\ (\exists k : 0 \le k < i+1 :\ a[k]=x)$

i:=i+1

$found\ =\ (\exists k : 0 \le k < i+1 :\ a[k]=x)$

$\{\ found\ =\ (\exists k : 0 \le k < i :\ a[k]=x)\ \}$

$wp\ (x:=e)\ Q\ =\ Q[e/x]$

Example

$\{\ \underbrace{\neg found\ \wedge\ \ldots\ \wedge\ (found\ =\ (\exists k : 0 \le k < i :\ a[k]=x))}\ \}$

found := a[i]=x ;

$\Downarrow$

$(a[i]=x)\ =\ (\exists k : 0 \le k < i+1 :\ a[k]=x)$

i:=i+1

$found\ =\ (\exists k : 0 \le k < i+1 :\ a[k]=x)$

$\{\ found\ =\ (\exists k : 0 \le k < i :\ a[k]=x)\ \}$

$wp\ (x:=e)\ Q\ =\ Q[e/x]$

Example

$0 \le i$

$$\{\ \underbrace{\neg found\ \wedge\ \dots\ \wedge\ (found\ =\ (\exists k : 0 \le k < i :\ a[k]=x))}\ \}$$

found := a[i]=x ;

$\Downarrow$

$\boxed{(a[i]=x)\ =\ (\exists k : 0 \le k < i+1 :\ a[k]=x)}$

i:=i+1

$\boxed{found\ =\ (\exists k : 0 \le k < i+1 :\ a[k]=x)}$

$$\{\ found\ =\ (\exists k : 0 \le k < i :\ a[k]=x)\ \}$$

wp (x:=e) Q = Q[e/x]

Reasoning about loops

How to prove this ?

- { P } <u>while</u> g <u>do</u> S { Q }

- Calculate wp first ?
  - We don't have to
  - But wp has nice property → wp completely captures the statement:

    $\{ P \} \;\; T \; \{ Q \} \quad = \quad P \Rightarrow \; wp \;\; T \;\; Q$

wp of a loop ….

- Recall :
  - wp(S,Q) = { s | forall s'. s S s' implies s'|=Q }
  - $\{ P \} \; S \; \{ Q \} \quad = \quad P \Rightarrow \; wp(S,Q)$

- But none of these definitions are actually useful to <u>construct</u> the weakest pre-condition.

- In the case of a loop, a constructive definition is not obvious. → pending.

How to prove this ?

- { P } <u>while</u> g <u>do</u> S { Q }

- Plan-B: try to come up with an inference rule:

```
          condition about   g
          condition about   S
---------------------------------------
 { P }  while g do S  { Q }
```

- The rule only need to be "sufficient".

Idea

Idea

- { P } while g do S { Q }

Idea

- { P } while g do S { Q }

37

Idea

- { P } <u>while</u> g <u>do</u> S { Q }

- Try to come up with a predicate I that holds <u>after</u> each iteration :

$iter_1$:	// g // ; S	{ I }	
$iter_2$:	// g // ; S	{ I }	
…			
$iter_n$:	// g // ; S	{ I }	// last iteration!
exit :	// ¬g //		

Idea

- { P } <u>while</u> g <u>do</u> S { Q }

- Try to come up with a predicate I that holds <u>after</u> each iteration :

$iter_1$:	// g // ; S	{ I }	
$iter_2$:	// g // ; S	{ I }	
…			
$iter_n$:	// g // ; S	{ I }	// last iteration!
exit :	// ¬g //		

Idea

- { P } <u>while</u> g <u>do</u> S { Q }

- Try to come up with a predicate I that holds <u>after</u> each iteration :

  iter$_1$: // g // ; S { I }
  iter$_2$: // g // ; S { I }
  …
  iter$_n$: // g // ; S { I } // last iteration!
  exit : // ¬g //

- I ∧ ¬g holds as the loop exit!

Idea

- { P } <u>while</u> g <u>do</u> S { Q }

- Try to come up with a predicate I that holds <u>after</u> each iteration :

  iter$_1$: // g // ; S { I }
  iter$_2$: // g // ; S { I }
  …
  iter$_n$: // g // ; S { I } // last iteration!
  exit : // ¬g //

So, to get postcond Q, sufficient to prove:

$I \land \neg g \Rightarrow Q$

- I ∧ ¬g holds as the loop exit!

Idea

- { P } while g do S { Q }

Still need to capture this.

- Try to come up with a predicate I that holds after each iteration :

$iter_1$:	// g // ; S	{ I }
$iter_2$:	// g // ; S	{ I }
…		
$iter_n$:	// g // ; S	{ I }
exit :	// ¬g //	

// last iteration!

- I /\ ¬g holds as the loop exit!

So, to get postcond Q, sufficient to prove:

$I \wedge \neg g \Rightarrow Q$

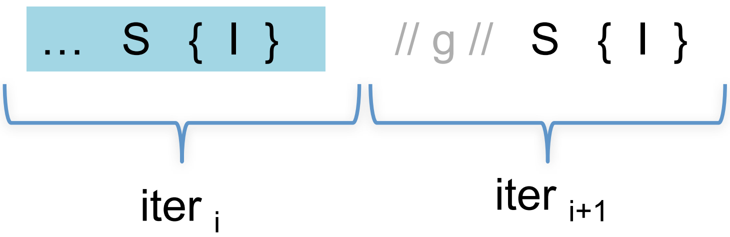
Idea

- while g do S
- I is to holds after each iteration

$$\underbrace{//\ g\ //\quad S\quad \{\ I\ \}}_{iter_{\ i+1}}$$

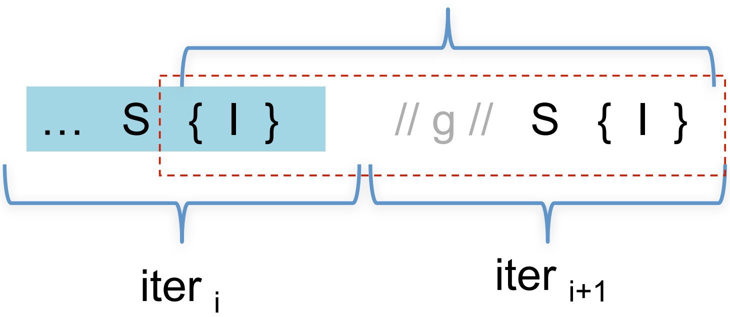
Idea

- while g do S
- I is to holds after each iteration



Idea

- while g do S
- I is to holds after each iteration



Idea

- <u>while</u> g <u>do</u> S
- I is to holds after each iteration

Sufficient to prove: { I /\ g } S { I }

$$\underbrace{\ldots\ \ S\ \ \{\ I\ \}}_{\text{iter}_{i}}\qquad \underbrace{/\!/\ g\ /\!/\ \ S\ \ \{\ I\ \}}_{\text{iter}_{i+1}}$$

Except for the first iteration !

Idea

- { P } <u>while</u> g <u>do</u> S
- For the first iteration :

$$\underbrace{/\!/\ g\ /\!/\ \ S\ \ \{\ I\ \}}_{\text{Iter}_{1}}$$

Idea

- { P } <u>while</u> g <u>do</u> S
- For the first iteration :

$$\boxed{\{\ I\ \}}\ \underbrace{//\ g\ //\quad S\quad \{\ I\ \}}_{\text{Iter}_1}$$

Idea

- { P } <u>while</u> g <u>do</u> S
- For the first iteration :

Recall the condition: { I /\ g } S { I }

$$\overbrace{\boxed{\{\ I\ \}}\ \underbrace{//\ g\ //\quad S\quad \{\ I\ \}}_{\text{Iter}_1}}$$

Idea

- { P } while g do S
- For the first iteration :

Recall the condition: $\{ I \land g \}\ S\ \{ I \}$

{ P } { I } // g // S { I }

$\underbrace{\text{// g // S \{ I \}}}_{\text{Iter}_1}$

We know this from the given pre-cond

Idea

- { P } while g do S
- For the first iteration :

Recall the condition: $\{ I \land g \}\ S\ \{ I \}$

{ P } { I } // g // S { I }

$\underbrace{\text{// g // S \{ I \}}}_{\text{Iter}_1}$

We know this from the given pre-cond

Idea

- { P } while g do S

- For the first iteration :

Additionally we need : $P \Rightarrow I$

Recall the condition: $\{ I \wedge g \}\ S\ \{ I \}$

$$\{ P \} \quad \{ I \} \quad \underbrace{//\ g\ //\quad S \quad \{ I \}}_{Iter_1}$$

We know this from
the given pre-cond

To Summarize

- Capture this in an inference rule:

$P \Rightarrow I$ // setting up I
$\{ g \wedge I \}\ S\ \{ I \}$ // invariance
$I \wedge \neg g \Rightarrow Q$ // exit cond

{ P } while g do S { Q }

- This rule is only good for partial correctness though.
- I satisfying the second premise above is called invariant.

Examples

- Prove:

  { i=0 } while i<n do i++ { i=n }

- Prove:

  { i=0 /\ s=0 }

  while i<n do { s = s +a[i] ; i++ }

  { s = SUM(a[0..n)) }

Note

- Recall :

  wp ((while g do S),Q) =
  { s | forall s'. s (while g do S) s' implies s' |=
  Q }

- Theoretically, we can still construct this set if the state space is finite. The construction is exactly as the def. above says.
- You need a way to tell when the loop does not terminate:
  - Maintain a history H of states after each iteration.
  - Non-termination if the state t after i-th iteration is in H from the previous iteration.
- Though then you can just as well 'execute' the program to verify it (testing), for which you don't need Hoare logic.

Tackling while termination: invariant and variant

To prove

$\{P\}$ **while** B **do** S **end** $\{Q\}$

find ~~invariant J~~ and well-founded ~~variant function vf~~ such that:

- ~~invariant holds initially~~: $P \Rightarrow J$
- ~~invariant is maintained~~: $\{J \wedge B\}\ S\ \{J\}$
- ~~invariant is sufficient~~: $J \wedge \neg B \Rightarrow Q$
- variant function is bounded:

  $J \wedge B \Rightarrow 0 \leq vf$
- variant function decreases:

  $\{J \wedge B \wedge vf=VF\}\ S\ \{vf<VF\}$

Proving termination

- { P } <u>while</u> g <u>do</u> S { Q }
- Idea: come up with an integer expression m, satisfying :
  - At the start of every iteration $m \geq 0$
  - Each iteration decreases m
- These imply that the loop will terminates.

Capturing the termination conditions

- At the start of every iteration $m \geq 0$:
  - $g \Rightarrow m \geq 0$
  - If you have an invariant: $I \wedge g \Rightarrow m \geq 0$
- Each iteration decreases m :

$$\{ \ I \wedge g \ \} \quad C:=m; S \quad \{ \ m<C \ \}$$

To Summarize

To Summarize

- $P \Rightarrow I$ // setting up I
  $\{\ g \wedge I\ \}\ \ S\ \ \{\ I\ \}$ // invariance
  $I \wedge \neg g \Rightarrow Q$ // exit cond
  $\{\ I \wedge g\ \}\ \ C:=m;\ S\ \ \{\ m<C\ \}$ // m decreasing
  $I \wedge g \Rightarrow m \geq 0$ // m bounded below

  $\{\ P\ \}\ \ \underline{\text{while}}\ g\ \underline{\text{do}}\ \ S\ \ \{\ Q\ \}$

To Summarize

- $P \Rightarrow I$ // setting up I
  $\{\ g \wedge I\ \}\ \ S\ \ \{\ I\ \}$ // invariance
  $I \wedge \neg g \Rightarrow Q$ // exit cond
  $\{\ I \wedge g\ \}\ \ C:=m;\ S\ \ \{\ m<C\ \}$ // m decreasing
  $I \wedge g \Rightarrow m \geq 0$ // m bounded below

  $\{\ P\ \}\ \ \underline{\text{while}}\ g\ \underline{\text{do}}\ \ S\ \ \{\ Q\ \}$

- Since we also have this pre-cond strengthening rule:

$$\frac{P \Rightarrow I\ ,\ \{\ I\ \}\ \text{while } g \text{ do } S\ \{\ Q\ \}}{\{\ P\ \}\ \text{while } g \text{ do } S\ \{\ Q\ \}}$$

A Bit History and Other Things

History

- Hoare logic, due to CAR Hoare 1969.
- Robert Floyd, 1967 → for *Flow Chart*. "Unstructured" program.
- Weakest preconditon → Edsger Dijkstra, 1975.
- Early 90s: the rise of theorem provers. Hoare logic is mechanized. e.g. "A Mechanized Hoare Logic of State Transitions" by Gordon.
- Renewed interests in Hoare Logic for automated verification: Leino et al, 1999, "*Checking Java programs via guarded commands*"
  Tool: ESC/Java.
- Byte code verification. Unstructured → going back to Floyd. Ehm... what did Dijkstra said again about GOTO??

History

- Hoare: "*An axiomatic basis for computer programming*", 1969.
- Charles Antony Richard Hoare, born 1934 in Sri Lanka
- 1980 : winner of Turing Award
- Other achievement:
  - CSP (Communicating Sequential Processes)
  - Implementor ALGOL 60
  - Quicksort
  - 2000 : *sir* Charles ☺

History

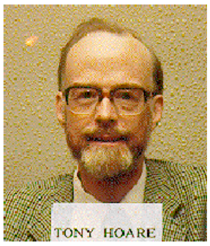
- Edsger Wybe Dijkstra, 1930 in Rotterdam.
- Prof. in TU Eindhoven, later in Texas, Austin.
- 1972 : winner Turing Award
- Achievement
  - Shortest path algorithm
  - Self-stabilization
  - Semaphore
  - *Structured Programming*, with Hoare.
  - "*A Case against the GO TO Statement*"
  - Program derivation
- Died in 2002, Nuenen.

Edsger W. Dijkstra
(cf. shortest path, semaphore)

ALGOL-60

- ALGOL-60: "ALGOrithmic Language" (1958-1968) by very many people IFIP(International Federation for Information Processing) , including John Backus, Peter Naur, Alan Perlis, Friedrich L. Bauer, John McCarthy, Niklaus Wirth, C. A. R. Hoare, Edsger W. Dijkstra
- Join effort by Academia and Industry
- Join effort by Europe and USA
- ALGOL-60 the most influential imperative language ever
- First language with syntax formally defined (BNF)
- First language with structured control structures
  - If then else
  - While (several forms)
  - But still goto
- First language with … (see next)
- Did not include I/O considered too hardware dependent
- ALGOL-60 revised several times in early 60's, as understanding of programming languages improved
- ALGOL-68 a major revision
  - by 1968 concerns on data abstraction become prominent, and ALGOL-68 addressed them
  - Considered too Big and Complex by many of the people that worked on the original ALGOL-60 (*C. A. R. Hoare*' Turing Lecture, cf. ADA later)

C. A. R. Hoare
(cf. axiomatic semantics, quicksort, CSP)

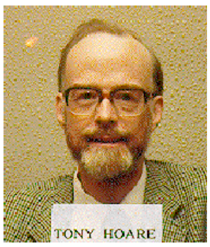


Edsger W. Dijkstra
(cf. shortest path, semaphore)

ALGOL-60

- First language with syntax formally defined (BNF)
  (after such a success with syntax, there was a great hope to being able to formally define semantics in an similarly easy and accessible way: this goal failed so far)
- First language with structured control structures
  - If then else
  - While (several forms)
  - But still goto
- First language with procedure activations based on the STACK (cf. recursion)
- First language with well defended parameters passing mechanisms
  - Call by value
  - Call by name (sort of call by reference)
  - Call by value result (later versions)
  - Call by reference (later versions)
- First language with explicit typing of variables
- First language with blocks (static scope)
- Data structure primitives: integers, reals, booleans, arrays of any dimension; (no records at first),
- Later version had also references and records (originally introduced in COBOL), and user defined types

C. A. R. Hoare
(cf. axiomatic semantics, quicksort, CSP)



Unstructured programs

Unstructured programs

- “Structured” program: the control flow follows the program’s syntax.

Unstructured programs

- "Structured" program: the control flow follows the program's syntax.

- Unstructured program:

  if y=0 then **goto** exit ;
  x := x/y ;
  exit: S_2

Unstructured programs

- "Structured" program: the control flow follows the program's syntax.

- Unstructured program:

  if y=0 then **goto** exit ;
  x := x/y ;
  exit: S_2

- The "standard" Hoare logic rule for sequential composition breaks out!

Unstructured programs

- "Structured" program: the control flow follows the program's syntax.

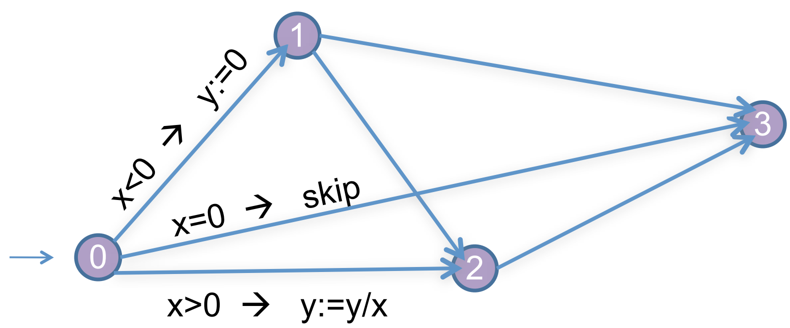
- Unstructured program:

```
          if  y=0  then   goto exit ;
          x := x/y ;
exit:     S2
```

- The "standard" Hoare logic rule for sequential composition breaks out!
- Same problem with exception, and "return" in the middle.

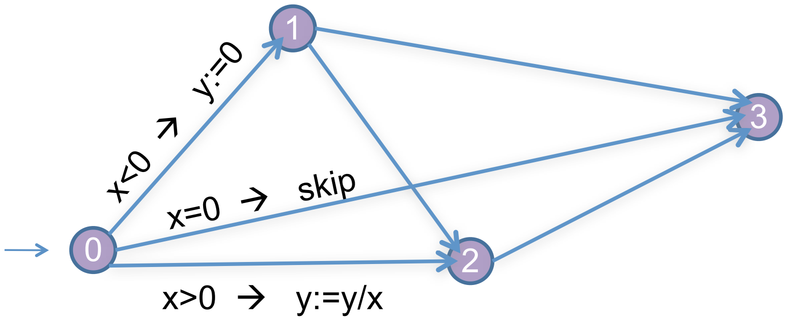
Adjusting Hoare Logic for Unstructured Programs

Program S :



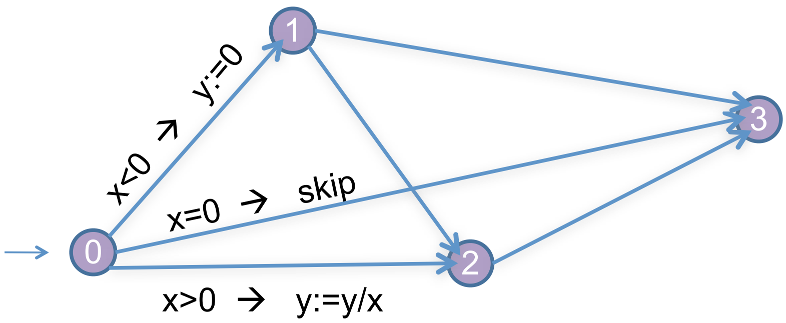
Adjusting Hoare Logic for Unstructured Programs

Program S : *represented by a graph of guarded assignments; here acyclic.*



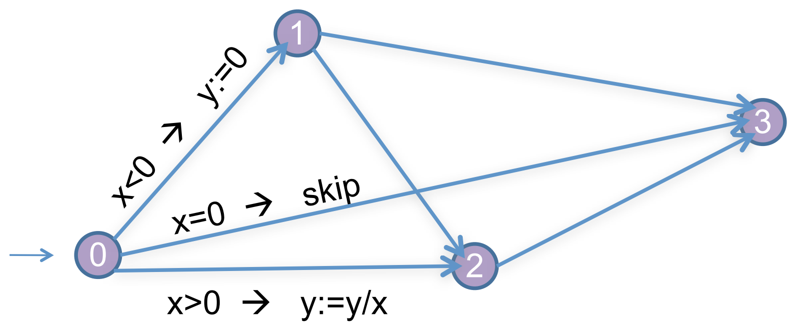
Adjusting Hoare Logic for Unstructured Programs

Program S : *represented by a graph of guarded assignments; here acyclic.*



Adjusting Hoare Logic for Unstructured Programs

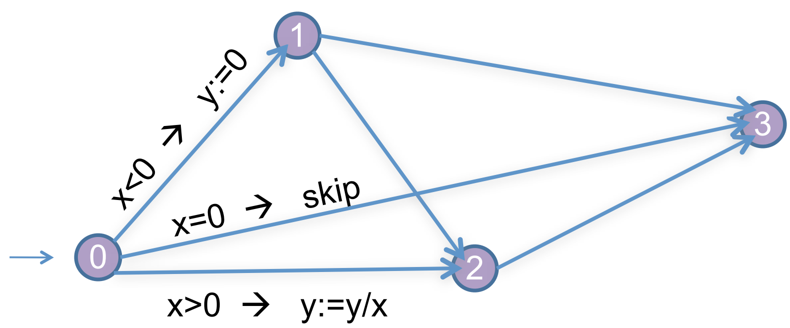
Program S :



1. Node represents “control location”
2. Edge is an assignment that moves the control of S, from one location to another.
3. An assignment can only execute if its guard is true.

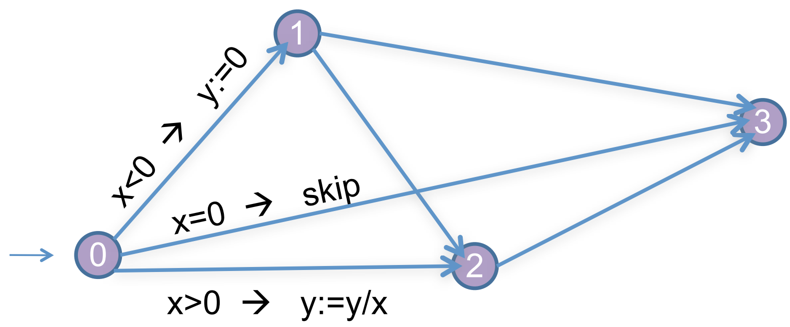
Adjusting Hoare Logic for Unstructured Programs

Prove { P } S { Q }



Adjusting Hoare Logic for Unstructured Programs

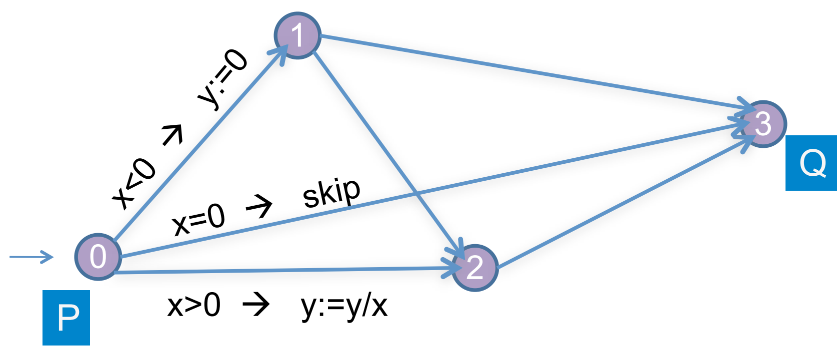
Prove { P } S { Q }



1. Decorate nodes with assertions.
2. Prove for each edge, the corresponding Hoare triple.

Adjusting Hoare Logic for Unstructured Programs

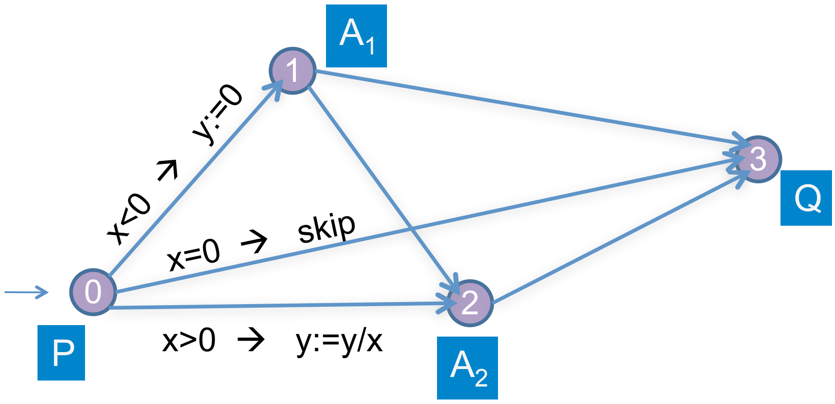
Prove { P } S { Q }



1. Decorate nodes with assertions.
2. Prove for each edge, the corresponding Hoare triple.

Adjusting Hoare Logic for Unstructured Programs

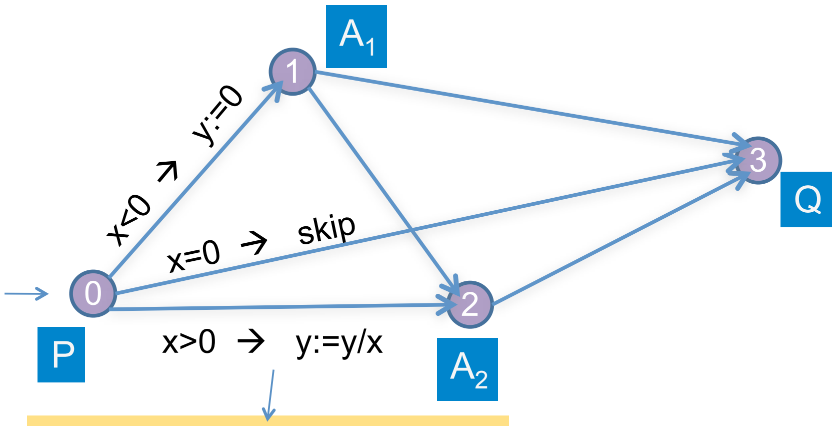
Prove { P } S { Q }



1. Decorate nodes with assertions.
2. Prove for each edge, the corresponding Hoare triple.

Adjusting Hoare Logic for Unstructured Programs

Prove { P } S { Q }



{ P /\ x>0 } y:=y/x { A2 }

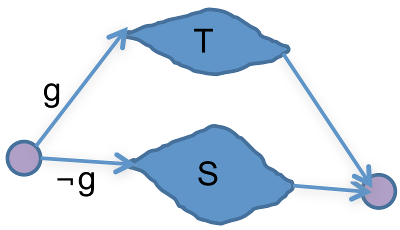
1. Decorate nodes with assertions.
2. Prove for each edge, the corresponding Hoare triple.

Handling exception and return-in-the-middle

- Map the program to a graph of control structure, then simply apply the logic for unstructured program.
- Example:

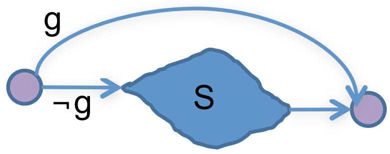
  try { if g then throw ; S }

  handle T ;



- Example:

  if g then return ;
  S ;
  return ;



Beyond pre/post conditions

- Class invariant
- When specifying the order of certain actions within a program is important:
  - E.g. CSP
- When sequences of observable states through out the execution have to satisfy certain property:
  - E.g. Temporal logic
- When the environment cannot be fully trusted:
  - E.g. Logic of belief