272: Software Engineering
Fall 2018

Instructor: Tevfik Bultan

Lecture 7: Hoare Logic and Weakest Preconditions

A Simple Class and Its Contract in JML

```
class BankAccount {
  int: balance;
  //@ invariant balance >= 0;

  withdraw(int: i) {
    //@ requires balance >= i and i >= 0;
    balance = balance - i;
    //@ ensures balance == \old(balance) - i;
  }
  deposit(int: i) {
    //@ requires i >= 0;
    balance = balance + i;
    //@ ensures balance == \old(balance) + i;
  }

  boolean isEmpty() {
    return balance == 0;
    //@ ensures result == (balance == 0);
  }
}
```

Same Example, Contract Written as Assertions

```
class BankAccount {
  int: balance;
  withdraw(int: i) {
    int oldbalance = balance;
    assert(balance >= i and i >= 0);
    balance = balance - i;
    assert(balance == oldbalance - i);
  }
  deposit(int: i) {
    int oldbalance = balance;
    assert(i >= 0);
    balance = balance + i;
    assert(balance == oldbalance + i);
  }

  boolean isEmpty() {
    boolean result = (balance == 0);
    assert (result == (balance == 0));
    return result;
  }
}
```

Question: Where do we put the class invariant?
Answer: Add as a conjunction to pre and post-conditions

Dynamic Contract Monitoring

- We can do dynamic contract monitoring for such specifications using tools such as JContractor and Jass
- When the contract fails we know that there is an error in the implementation
  - We can identify who is responsible for the contract violation (i.e., the caller or the callee)
- Note that the contract monitoring is dynamic, i.e., it is done during the program execution
  - If we do not observe a contract violation for a set of executions, that does not mean that a contract violation will never happen.
- But some of the implementation code is so close to the pre and post-conditions specified in the contract, it looks like we should be able to ***prove*** that the implementation is correct with respect to the contract
  - Proving the implementation correct with respect to the contract means proving that there will never be a contract violation for any execution of the program!

Example

- Here is the question:
  - If we assume that the pre-condition holds, then does the implementation guarantee that the post-condition is satisfied?
  - I.e., if the pre-condition holds, then is it guaranteed that the assertion that checks the post-condition will not cause an assertion failure?

```
withdraw(int: i) {

  int oldbalance = balance;
  assert(balance >= i and i >= 0);

  balance = balance - i;

  assert(balance == oldbalance - i);

}
```

this is the implementation part

these are the assertions that come from the contract specification

Hoare Logic and Weakest Preconditions

- Hoare Logic and Weakest Preconditions are formalisms which can be used to answer such questions

- The material in the following slides is mostly from the following papers:
  - “An Axiomatic Basis for Computer Programming,” C. A. R. Hoare, Communications of the ACM, vol. 12, no. 10, pp. 576-583, 1969
  - “Guarded Commands, Nondeterminacy and Formal Derivation of Programs,” E. W. Dijkstra, Communications of the ACM, vol. 18, no. 8, pp. 453-457, 1975

Correctness

- How can we reason about the correctness of programs?
  - Use mathematics!
- We know what correctness means mathematically
  - For example:
    - $5 = 2 + 2$ is incorrect
    - $3 = 2 + 1$ is correct
    - $\forall x, \exists y, y = x + 1$ is correct for integers
    - $\exists x, \exists y, \exists z, x^4 = y^4 + z^4 \wedge x \neq 0$ is incorrect for integers
- So, what does correctness mean?
  - A mathematical statement about integers is correct if it can be inferred from the axioms defining integers
    - Showing this is called a proof
  - If we can show that the negation of a statement is correct, then we know that the statement is incorrect

What about Programs?

- Then the question becomes
  - Can we develop a mathematical framework for proving correctness of programs?
  - And the answer is yes.
  - But it is not very easy to do the proofs by hand.
  - And it is not possible to automate the proofs in general.

Reasoning About Programs

- Mathematical formalisms do not immediately translate to reasoning about programs
  - Integer arithmetic used in programs is different
    - Is $\forall x, \exists y, y = x + 1$ true for integer constants in a program?
    - No, because we will eventually get to MAXINT and get overflow

- We can still formalize mathematical rules about the programs
  - This is what the semantics of the programming language is supposed to do
  - Semantics of programming languages are complicated:
    - variables, assignments, arrays, pointers, procedures, parameter passing, object classes, inheritance, concurrency, etc.

Reasoning about program segments

- Reasoning about a program as a whole could be very complicated due to
  - procedure calls, parameter passing, recursion, dynamic memory allocation, etc.
- Let's focus on simple program segments
  - Sequences of assignments, loops etc. without procedure calls
- Note that, the example we had earlier suggests a form of modularization for checking correctness for procedures
  - To show the correctness of a procedure, show that when the precondition holds, the post-condition always holds after executing the procedure
  - Then we also have to show that whenever the procedure is called its precondition is established. We can check that by inserting assertions to the procedure call sites.

Assertions

- We can use logical assertions to state properties about variables of a program
  - Assertion $x > y$ (where x and y are integer variables) is true if the value of x is greater than value of y
  - Assertion $x+y=C$ is (x,y integer variables, C an integer constant) is true if addition of the values of variables x and y is equal to the constant C
  - $\forall i, 0 \leq i < A.length, A[i] = 0$ is true if all members of the integer array A have the value 0

Using Assertions To Specify Properties

- We can use assertions to reason about the correctness of program segments

- Hoare Logic formalizes this idea

- An Hoare triple is in the following form:
  - {P} S {Q}

  where P and Q are assertions, and S is a program segment

- {P} S {Q} means "if we assume that P holds before S starts executing, then Q holds at the end of the execution of S"
  - I.e., if we *assume* P before execution of S, Q is *guaranteed* after execution of S

Example Hoare triples

- Correct Hoare triples (i.e., we can prove them)

Assignment

Assertions are written within {}

  - $\{x=0\}\ x:=x+1\ \{x=1\}$
  - $\{x+y=5\}\ x:=x+5;\ y:=y-1\ \{x+y=9\}$
  - $\{x+y=C\}\ x:=x+5;\ y:=y-1\ \{x+y=C+4\}$ where C is a place holder for any integer constant, i.e., it is equivalent to
    - $\forall C,\ \{x+y=C\}\ x:=x+5;\ y:=y-1\ \{x+y=C+4\}$
  - $\{x>C\}\ x:=x+1\ \{x>C+1\}$
  - $\{x>C\}\ x:=x+1\ \{x>C\}$

- Incorrect Hoare triples
  - $\{x=1\}\ x:=x+1\ \{x=1\}$
  - $\{x+y=C\}\ x:=x+1;\ y:=y-1\ \{x+y=C+1\}$

What about our example?

Here is the Hoare triple for the procedure body of the withdraw method:

$\{balance \geq i \wedge i \geq 0 \wedge balance = oldbalance \wedge balance \geq 0\}$

$balance := balance - i$

$\{balance = oldbalance - i \wedge balance \geq 0\}$

Here is the Hoare triple for the procedure body of the deposit method:

$\{i \geq 0 \wedge balance = oldbalance \wedge balance \geq 0\}$

$balance := balance + i$

$\{balance = oldbalance + i \wedge balance \geq 0\}$

If we can PROVE the above Hoare triples, then that means that we proved the implementation of the withdraw and deposit methods

Partial vs. Total Correctness

- I use the notation
  - {P} S {Q}
- instead of the original notation in Hoare's paper
  - P {S} Q

- Some researchers differentiate the meaning of these notations
  - {P} S {Q} means total correctness:
    - If we assume that P holds before S starts executing, then S terminates and Q holds at the end of the execution of S
  - P {S} Q means partial correctness:
    - If we assume that P holds before S starts executing and if S terminates then Q holds at the end of the execution of S

Proving properties of program segments

- How can we prove that:
  - {x=0} x:=x+1 {x=1} is correct?

- We need an axiom which explains what assignment does

- First, we will need more notation

- We need to define the substitution operation
  - Let P[x←exp] denote the assertion obtained from P by replacing every appearance of x in P by the value of the expression exp

- Examples
  - $x=0[x \leftarrow 0] \equiv 0=0$
  - $x+y=z[x \leftarrow 0] \equiv 0+y=z \equiv y=z$

I am using "≡" to denote equivalence between assertions

Axiom of Assignment

- Here is the **axiom of assignment**:
  - {P[x←exp]} x:=exp {P}
    - where exp is a simple expression (no procedure calls in exp) that has no side effects (evaluating the expression does not change the state of the program)

- Now, let's try to prove
  - {x=0} x:=x+1 {x=1}
  - We have
  - {x=1[x←x+1]} x:=x+1 {x=1} (by axiom of assignment)

    ≡ {x+1=1} x:=x+1 {x=1} (by definition of the substitution operation)

    ≡ {x=0} x:=x+1 {x=1} (arithmetic manipulation, i.e., by some axiom of arithmetic)
    - This is the end of our proof, we showed that the Hoare triple {x=0} x:=x+1 {x=1} follows from the axiom of assignment

Axiom of Assignment

- Another example
  - $\{x \geq 0\}$ x:=x+1 $\{x \geq 1\}$
  - We have

    $\{x \geq 1[x \leftarrow x+1]\}$ x:=x+1 $\{x \geq 1\}$ (by axiom of assignment)

    $\equiv \{x+1 \geq 1\}$ x:=x+1 $\{x \geq 1\}$ (by definition of the substitution operation)

    $\equiv \{x \geq 0\}$ x:=x+1 $\{x \geq 1\}$ (arithmetic manipulation, i.e., by some axiom of arithmetic)

Justification for the Axiom of Assignment

- Axiom assignment: $\{P[x \leftarrow exp]\}\ x:=exp\ \{P\}$
- Let us write the assignment using equality and primed variables:

$x' = exp$

where x denotes the value of variable x before the assignment, and x' denotes the value of the variable x after the assignment

- Then we can consider the assignment and the property P as a conjunction if we replace every appearance of x in P with x'

$x' = exp \wedge P[x \leftarrow x']$

- Then we have:

$x' = exp \wedge P[x \leftarrow x'] \Rightarrow (P[x \leftarrow x'])[x' \leftarrow exp]$

- For example:

$x:=x+1\ \{x=1\}$ becomes: $x'=x+1 \wedge x'=1$

$x'=x+1 \wedge x'=1 \Rightarrow x+1 = 1 \equiv x=0$

Rules of Inference

- Once we prove a Hoare triple we may want to use it to prove other Hoare triples

- If we already proved {x=0} x:=x+1 {x=1}, then we should be able to conclude that {x=0} x:=x+1 {x>0} also holds

- Here is the general rule (**rule of consequence 1**)
  - If {P}S{Q} and Q⇒R then we can conclude {P}S{R}

- This rule means that once you prove a post-condition, you can always infer a weaker post-condition
- Example:
  - {x=0} x:=x+1 {x=1} and x=1⇒x>0
    - hence, we conclude {x=0} x:=x+1 {x>0}

Rules of Inference

- If we already proved $\{x \geq 0\}$ $x:=x+1$ $\{x \geq 1\}$, then we should be able to conclude $\{x \geq 5\}$ $x:=x+1$ $\{x \geq 1\}$

- Here is the general rule (**rule of consequence 2**)
  - If $\{P\}S\{Q\}$ and $R \Rightarrow P$ then we can conclude $\{R\}S\{Q\}$

- This rule means that once you prove a pre-condition assumption, you can always infer a stronger pre-condition assumption

- Example
  - $\{x \geq 0\}$ $x:=x+1$ $\{x \geq 1\}$ and $x \geq 5 \Rightarrow x \geq 0$
    - hence, we conclude $\{x \geq 5\}$ $x:=x+1$ $\{x \geq 1\}$

Back to Our Example

Proving the implementation of the withdraw method:

{balance = oldbalance – i $\wedge$ balance $\geq$ 0 $\wedge$ i $\geq$ 0[balance $\leftarrow$ balance – i] }

balance := balance – i

{balance = oldbalance – i $\wedge$ balance $\geq$ 0 $\wedge$ i $\geq$ 0} (by axiom of assignment)

$\equiv$ {balance – i=oldbalance – i $\wedge$ balance – i $\geq$ 0 $\wedge$ i $\geq$ 0}

balance := balance – i

{balance = oldbalance – i $\wedge$ balance $\geq$ 0 $\wedge$ i $\geq$ 0} (by definition of the substitution operation)

$\equiv$ {balance =oldbalance $\wedge$ balance $\geq$ i $\wedge$ i $\geq$ 0}

balance := balance – i

{balance = oldbalance – i $\wedge$ balance $\geq$ 0 $\wedge$ i $\geq$ 0} (arithmetic manipulation)

$\equiv$ {balance =oldbalance $\wedge$ balance $\geq$ i $\wedge$ i $\geq$ 0}

balance := balance – i

{balance = oldbalance – i $\wedge$ balance $\geq$ 0 } (rule of consequence 1)

$\equiv$ {balance =oldbalance $\wedge$ balance $\geq$ i $\wedge$ i $\geq$ 0 $\wedge$ balance $\geq$ 0}

balance := balance – i

{balance = oldbalance – i $\wedge$ balance $\geq$ 0 } (rule of consequence 2)

Rule of Sequential Composition

- Program segments can be formed by sequential composition
  - x:=x+5; y:=y–1 is sequential composition of two assignment statements x:=x+5 and y:=y-1
  - x:=x+5; y:=y–1; t:=0 is a sequential composition of the program segment x:=x+5; y:=y–1 and the assignment statement t:=0

- How do we reason about sequences of program statements?

- Here is the inference **rule of sequential composition**
  - If $\{P\}S_1\{Q\}$ and $\{Q\}S_2\{R\}$ then we can conclude that

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

Example: Swap

- Let's try to prove a swap operation based on what we learned
  - Here is the program segment for swap:
    t:=x; x:=y; y:=t

- Let's assume that $x=A \wedge y=B$ holds before we start executing the swap segment.
- If swap is working correctly we would like $x=B \wedge y=A$ to hold at the end of the swap (note that we did not restrict the values A and B in any way)

- Let's apply the axiom of assignment twice
  - $\{x=B \wedge y=A[y \leftarrow t]\}$ y:=t $\{x=B \wedge y=A\}$

    $\equiv \{x=B \wedge t=A\}$ y:=t $\{x=B \wedge y=A\}$
  - $\{x=B \wedge t=A[x \leftarrow y]\}$ x:=y $\{x=B \wedge t=A\}$

    $\equiv \{y=B \wedge t=A\}$ x:=y $\{x=B \wedge t=A\}$

Example: Swap

- Now since we have
  - $\{y=B \wedge t=A\}$ x:=y $\{x=B \wedge t=A\}$ and $\{x=B \wedge t=A\}$ y:=t $\{x=B \wedge y=A\}$,
  - using the rule of sequential composition we get:
  - $\{y=B \wedge t=A\}$ x:=y; y:=t $\{x=B \wedge y=A\}$

- Let's apply the axiom of assignment once more
  - $\{y=B \wedge t=A[t \leftarrow x]\}$ t:=x $\{y=B \wedge t=A\}$

    $\equiv \{y=B \wedge x=A\}$ t:=x $\{y=B \wedge t=A\}$

- Using the rule of sequential composition once more

$\{y=B \wedge x=A\}$ t:=x $\{y=B \wedge t=A\}$ and $\{y=B \wedge t=A\}$ x:=y; y:=t $\{x=B \wedge y=A\}$

$\Rightarrow \{y=B \wedge x=A\}$ t:=x; x:=y; y:=t $\{x=B \wedge y=A\}$

Inference rule for conditionals

- There are two inference rules for conditional statements, one for if-then and one for if-then-else statements

- For if-then-else statements the rule is (**rule of conditional 1)**
  - If $\{P \wedge B\}\ S_1\ \{Q\}$ and $\{P \wedge \neg B\}\ S_2\ \{Q\}$ hold then we conclude that
    $\{P\}$ if B then S_1 else S_2 $\{Q\}$

- For if-then statements the rule is (**rule of conditional 2)**
  - If $\{P \wedge B\}\ S\ \{Q\}$ and $P \wedge \neg B \Rightarrow Q$ hold then we conclude that
    $\{P\}$ if B then S $\{Q\}$

Example for conditionals

- Here is an example
  - if (x >y) max:=x else max:=y
  - We want to prove
  - {True} if (x>y) max:=x else max:=y {max≥x∧max≥y}

{max≥x∧max≥y[max←x]} max:=x {max≥x∧max≥y} (r.assign.)

≡ {x≥x∧x≥y} max:=x {max≥x∧max≥y} (definition of subs.)

≡ {True∧x≥y} max:=x {max≥x ∧ max≥y} (some axiom of arith.)

≡ {x≥y} max:=x {max≥x ∧ max≥y} (some axiom of logic)

≡ {x>y} max:=x {max≥x ∧ max≥y} (r. of cons. 2)

Example for conditionals

$\{max \geq x \wedge max \geq y[max \leftarrow y]\}$ $max:=y$ $\{max \geq x \wedge max \geq y\}$ (r.assign.)

$\equiv$ $\{y \geq x \wedge y \geq y\}$ $max:=y$ $\{max \geq x \wedge max \geq y\}$ (definition of subs.)

$\equiv$ $\{y \geq x \wedge True\}$ $max:=y$ $\{max \geq x \wedge max \geq y\}$ (some axiom of arith.)

$\equiv$ $\{y \geq x\}$ $max:=y$ $\{max \geq x \wedge max \geq y\}$ (some axiom of logic)

$\equiv$ $\{\neg x > y\}$ $max:=y$ $\{max \geq x \wedge max \geq y\}$ (some axiom of logic)

So we proved that $\{x>y\}$ $max:=x$ $\{max \geq x \wedge max \geq y\}$ and

$\{\neg x>y\}$ $max:=y$ $\{max \geq x \wedge max \geq y\}$ then we can use the rule of conditional 1 and conclude that:

$\{True\}$ if $(x>y)$ $max:=x$ else $max:=y$ $\{max \geq x \wedge max \geq y\}$

What about the loops?

- Here is the inference rule (**rule of iteration**) for while loops
  - If $\{P \wedge B\}$ S $\{P\}$ then we can conclude that

    $\{P\}$ while B do S $\{\neg B \wedge P\}$

- This is what the inference rule for while loop is saying:
  - If you can show that every iteration of the loop preserves the property P,
  - and you know that the property holds before you start executing the loop,
  - then you can conclude that the property holds at the termination of the loop.
  - Also the loop condition will not hold at the termination of the loop (otherwise the loop would not terminate).

Loop invariants

- Given a loop
  - while B do S
  - Any assertion P which satisfies $\{P \wedge B\}$ S $\{P\}$ is called a ***loop invariant***

- A loop invariant is an assertion such that, every iteration of the loop body preserves it
  - We write this as a Hoare triple as $\{P \wedge B\}$ S $\{P\}$

- Note that rule of iteration given in the previous slide is for partial correctness
  - It does not guarantee that the loop will terminate

Example

- Here is an example loop
  while (y <= r) do (r:=r–y; q:=q+1)

- Let's pick P as $r+y\times q=A$ where A is an integer value

$\{r+y\times(q+1)=A\}$ q:=q+1 $\{r+y\times q=A\}$ (by axiom of assignment)

$\{r-y+y\times(q+1)=A\}$ r:=r–y $\{r+y\times(q+1)=A\}$ (by axiom of assignment)

$\{r+y\times q=A\}$ r:=r-y; q:=q+1 $\{r+y\times q=A\}$ (by sequential composition rule)

$\{r+y\times q=A \wedge (y\leq r)\}$ r:=r–y; q:=q+1 $\{r+y\times q=A\}$ (by rule of consequence 2)

$\{r+y\times q=A\}$ while (y <= r) do (r:=r–y; q:=q+1) $\{\neg\ (y\leq r) \wedge r+y\times q=A\}$ (by rule of iteration)

Using the rule of iteration

- To prove that a property Q holds after the loop while B do S terminates, we can use the following strategy
  - Find a strong enough loop invariant P such that:

    $(\neg B \wedge P) \Rightarrow Q$
  - Show that P is a loop invariant: $\{P \wedge B\}$ S $\{P\}$
  - If we can show that P is a loop invariant, we get

    $\{P\}$ while B do S $\{\neg B \wedge P\}$
  - Since we had $(\neg B \wedge P) \Rightarrow Q$, using the rule of consequence 1, we get

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

Example

- Consider the following program segment:
  sum:=0; i:=1; while (i <=10) do (sum:=sum+i; i:=i+1)

- We want to prove that $Q \equiv sum=\Sigma_{0\leq\ k\ \leq\ 10}\ k$
  holds at the loop termination, i.e., we want to prove the Hoare triple:

  {true} sum:=0; i:=0; while (i <=10) do (sum:=sum+i; i:=i+1) {Q}

- We need to find a strong enough loop invariant P
- Let's choose P as follows:

  $P \equiv i \leq 11 \wedge sum=\Sigma_{0\leq\ k<i}\ k$

Example

- To use the rule of iteration we need to show $\{P \wedge B\}\ S\ \{P\}$ where
  - $P \equiv i \leq 11 \wedge sum=\Sigma_{0\leq k<i} k$
  - S: sum:=sum+i; i:=i+1
  - $B \equiv i \leq 10$

- Using the rule of assignment we get:

$\{i \leq 11 \wedge sum=\Sigma_{0\leq k<i} k[i \leftarrow i+1]\}$ i:=i+1 $\{i \leq 11 \wedge sum=\Sigma_{0\leq k<i} k\}$

$\equiv$ $\{i+1 \leq 11 \wedge sum=\Sigma_{0\leq k<i+1} k\}$ i:=i+1 $\{i \leq 11 \wedge sum=\Sigma_{0\leq k<i} k\}$

$\equiv$ $\{i \leq 10 \wedge sum=\Sigma_{0\leq k<i+1} k\}$ i:=i+1 $\{i \leq 11 \wedge sum=\Sigma_{0\leq k<i} k\}$

Example

Using the rule of assignment one more time:

$\{i \leq 10 \wedge sum=\Sigma_{0\leq k<i+1}k[sum \leftarrow sum+i]\}$ sum:=sum+i $\{i \leq 10 \wedge sum=\Sigma_{0\leq k<i+1}k\}$

$\equiv \{i \leq 10 \wedge sum+i=\Sigma_{0\leq k<i+1}k\}$ sum:=sum+i $\{i \leq 10 \wedge sum=\Sigma_{0\leq k<i+1}k\}$

$\equiv \{i \leq 10 \wedge sum=\Sigma_{0\leq k<i}k\}$ sum:=sum+i $\{i \leq 10 \wedge sum=\Sigma_{0\leq k<i+1}k\}$

Using the rule of sequential composition we get:

$\{i \leq 10 \wedge sum=\Sigma_{0\leq k<i}k\}$ sum:=sum+i; i:=i+1 $\{i \leq 11 \wedge sum=\Sigma_{0\leq k<i}k\}$

Example

- Note that

$P \wedge B \equiv (i \leq 11 \wedge sum=\Sigma_{0\leq k<i}k) \wedge (i \leq 10) \equiv i \leq 10 \wedge sum=\Sigma_{0\leq k<i}k$

$P \wedge \neg B \equiv (i \leq 11 \wedge sum=\Sigma_{0\leq k<i}k) \wedge \neg(i \leq 10)$

$\equiv i \leq 11 \wedge i > 10 \wedge sum=\Sigma_{0\leq k<i}k \equiv i = 11 \wedge sum=\Sigma_{0\leq k<i}k$

$\equiv sum=\Sigma_{0\leq k<11}k$

- Using the rule of iteration we get:

$\{i \leq 11 \wedge sum=\Sigma_{0\leq k<i}k\}$ while (i <=10) do (sum:=sum+i; i:=i+1) $\{sum=\Sigma_{0\leq k<11}k\}$

Example

- To finish the proof, apply rule of assignment

$\{i \leq 11 \wedge sum=\Sigma_{0\leq k<i} k[i \leftarrow 1]\}$ i := 1$\{i \leq 11 \wedge sum=\Sigma_{0\leq k<i} k\}$

$\equiv$ $\{1 \leq 11 \wedge sum=\Sigma_{0\leq k<1} k\}$ i := 1$\{i \leq 11 \wedge sum=\Sigma_{0\leq k<i} k\}$

$\equiv$ {sum=0} i := 1$\{i \leq 11 \wedge sum=\Sigma_{0\leq k<i} k\}$

Another rule of assignment application

{sum=0 [sum$\leftarrow$0]} sum := 0 {sum=0}

{0=0} sum := 0 {sum=0}

{true} sum := 0 {sum=0}

Example

- Finally, combining the previous results with rule of sequential composition we get:

$$\{true\}\ sum:=0;\ i:=0;\ while\ (i <= 10)\ do\ (sum:=sum+i;\ i:=i+1)\ \{sum=\Sigma_{0\leq k\leq 10}\ k\}$$

Difficulties in Proving Programs Correct

- Finding a loop invariant that is strong enough to prove the property that we are interested in can be difficult

- Also, note that we did not prove that the loop will terminate
  - To prove total correctness we also have to prove that the loop terminates

- Things get more complicated when there are procedures and recursion

Difficulties in Proving Programs Correct

- Hoare Logic is a formalism for reasoning about correctness about programs

- Developing proof of correctness using this formalism is another issue

- In general proving correctness about programs is uncomputable
  - For example determining that a program terminates is uncomputable

- This means that there is no automatic way of generating these proofs

- Still Hoare's formalism is useful for reasoning about programs

Weakest Preconditions

- Dijkstra added another tool to Hoare's formalism called ***weakest precondition***.
  - It is another useful tool in reasoning about programs

- Given an assertion Q and a program segment S weakest precondition of S with respect to Q written wp(S,Q) is defined as:
  - the weakest condition such that if S starts executing in a state which satisfies that condition, when it terminates it is guaranteed that Q will hold.

- Note that the Hoare triple {P}S{Q} is correct if and only if $P \Rightarrow wp(S,Q)$
  - this is why it is called the weakest precondition, every other assertion P where we can show {P}S{Q} implies (i.e., is stronger than) wp(S,Q)

Weakest Preconditions

- Dijkstra calls wp(S,Q) a predicate transformer
  - wp(S,Q) takes a predicate (assertion, same thing) Q and a program segment S, and transforms it to another predicate that corresponds to the weakest precondition of S with respect to Q

- For example, for simple assignments x:=exp (where exp is a simple expression with no procedure calls and no side effects) we already know the predicate transformer:
  - $wp(x{:=}exp, Q) = Q[x \leftarrow exp]$
    - where exp is a simple expression (no procedure calls in exp) that has no side effects (evaluating the expression does not change the state of the program)

Some rules about weakest preconditions

- Some rules about weakest preconditions
  - If $P \Rightarrow Q$ then $wp(S,P) \Rightarrow wp(S,Q)$
  - $wp(S,P) \wedge wp(S,Q) \equiv wp(S, P \wedge Q)$
  - $wp(S,P) \vee wp(S,Q) \equiv wp(S, P \vee Q)$
  - $wp(S_1 ; S_2 , P) \equiv wp(S_1, wp(S_2, P))$
  - $wp(\text{if } B \text{ then } S_1 \text{ else } S_2 , P) \equiv (B \Rightarrow wp(S_1, P)) \wedge (\neg B \Rightarrow wp(S_2, P))$
  - $wp(\text{if } B \text{ then } S_1, P) \equiv (B \Rightarrow wp(S_1, P)) \wedge (\neg B \Rightarrow P)$

Examples

- $wp(x:=x+1,\ x\geq 1)$

$\equiv$ $x\geq 1[x\leftarrow x+1]$

$\equiv$ $x+1\geq 1$

$\equiv$ $x\geq 0$

- $wp(x:=x+1;x:=x+2,x<10)$

$\equiv$ $wp(x:=x+1,\ wp(x:=x+2,\ x<10))$

$\equiv$ $wp(x:=x+1,\ x<10[x\leftarrow x+2])$

$\equiv$ $wp(x:=x+1,\ x+2<10)$

$\equiv$ $wp(x:=x+1,\ x<8)$

$\equiv$ $x<8[x\leftarrow x+1]$

$\equiv$ $x+1<8$

$\equiv$ $x<7$

Examples

- $wp(\text{if } (x > y)\ max{:=}x \text{ else } max{:=}y,\ max \geq x \wedge max \geq y)$

$\equiv$ $(x>y \Rightarrow wp(max{:=}x,\ max \geq x \wedge max \geq y)) \wedge (\neg(x>y) \Rightarrow wp(max{:=}y,\ max \geq x \wedge max \geq y))$

$\equiv$ $(x>y \Rightarrow max \geq x \wedge max \geq y[max \leftarrow x]) \wedge (x \leq y \Rightarrow max \geq x \wedge max \geq y[max \leftarrow y])$

$\equiv$ $(x>y \Rightarrow x \geq x \wedge x \geq y) \wedge (x \leq y \Rightarrow y \geq x \wedge y \geq y)$

$\equiv$ $(x>y \Rightarrow x \geq y) \wedge (x \leq y \Rightarrow y \geq x)$

$\equiv$ true

Loops

- Loops are more complicated
- We want to compute wp(while B do S, P)
- We will need the following definitions:
  - Let $H_0(P) \equiv \neg B \wedge P$
  - Let (for $k > 0$) $H_k(P) \equiv$ wp(if B then S, $H_{k-1}(P)$) $\vee$ $H_0(P)$
- *Intuition*: $H_k(P)$ is the weakest precondition for the case that the loop body is executed less than or equal to k times

Loops

- One can show that the weakest precondition is the (infinite) disjunction of the iterates $H_0(P)$, $H_1(P)$, $H_2(P)$, ... :
  - wp(while B do S, P) $\equiv H_0(P) \vee H_1(P) \vee H_2(P) \ldots$
  - Equivalently (by replacing the infinite disjunction with existential quantification, we get):
    - wp(while B do S, P) $\equiv \exists m, m \geq 0, H_m(P)$
  - Intuition: The weakest precondition states that there exists an m where the loop will iterate at most m times, and the weakest precondition of the loop is the weakest precondition that corresponds to iterating the loop m times or less

Loops

- One can show that, if there is an n where $H_n(P) \equiv H_{n-1}(P)$ then
  - $H_0(P) \vee H_1(P) \vee H_2(P) \ldots \equiv H_n(P)$
- Hence, if we can find an n where $H_n(P) \equiv H_{n-1}(P)$ then
  - wp(while B do S, P) $\equiv H_n(P)$
  - However, there may not be an n where $H_n(P) \equiv H_{n-1}(P)$

Loops: Example

- Assume that we want to compute the following weakest precondition
  - wp(while (i<=10) do i:=i+1, i=11)

$H_0(i=11) \equiv i>10 \wedge i=11 \equiv i=11$

$H_1(i=11) \equiv wp(\text{if}(i<=10) \text{ then } i:=i+1, i=11) \vee i=11$

$\equiv i=10 \vee i=11$

$H_2(i=11) \equiv i=9 \vee i=10 \vee i=11$

$H_3(i=11) \equiv i=8 \vee i=9 \vee i=10 \vee i=11$

...

We can see that, $H_k(i=11) \equiv \bigvee_{0\leq j \leq k} i = 11- j$

Note that, for each k, $H_k(i=11) \not\equiv H_{k-1}(i=11)$

Loops: Example

Remember, we said that the weakest precondition can be written as an infinite disjunction of the iterates:

$$wp(\text{while } (i<=10) \text{ do } i:=i+1,\ i=11) \equiv H_0(i=11) \vee H_1(i=11) \vee H_2(i=11) \ldots$$

and that the infinite disjunction is equivalent to

$$wp(\text{while } (i<=10) \text{ do } i:=i+1,\ i=11) \equiv \exists m,\ m\geq 0,\ H_m(i=11)$$

$$\equiv\ \exists m,\ m\geq 0,\ \bigvee_{0\leq j\leq m} i = 11-j\ \equiv \exists m,\ m\geq 0,\ 11 - m \leq i \leq 11$$

$$\equiv \exists m,\ m\geq 0 \wedge 11 - m \leq i \wedge i \leq 11 \equiv\ i \leq 11$$

Loops: Fixpoint

- Note that $i \leq 11$ is a ***fixpoint*** of the iterative definition for the weakest precondition in this example.

  The iterative definition was:

  $H_k\ (i=11) \equiv wp(\text{while } (i<=10) \text{ do } i:=i+1, H_{k-1}(i=11)) \vee H_0(i=11)$

  where $H_0(i=11) \equiv i>10 \wedge i=11 \equiv i=11$

- What does fixpoint mean?
  - It means that, if we set $H_{k-1}(i=11) \equiv i \leq 11$ we will get $H_k \equiv H_{k-1}$

- Let's try:

  $H_k \equiv wp(\text{if}(i<=10) \text{ then } i:=i+1, i \leq 11) \vee i = 11$

  $\equiv i \leq 10 \vee i = 11 \equiv i \leq 11$

  We see that, $H_k \equiv H_{k-1} \equiv i \leq 11$

Loops: Least Fixpoint

- Actually, $i \leq 11$ is the ***least fixpoint*** of the iterative definition for the weakest precondition in this example.

- What does it mean that $i \leq 11$ is the least fixpoint of the iterative definition?
  - It means that for any other predicate P which is the fixpoint of the iteration $i \leq 11 \Rightarrow P$

- For example, $i \leq 12$ is also a fixpoint of the iterative definition for the weakest precondition in this example, however it is not the least fixpoint since $i \leq 12 \not\Rightarrow i \leq 11$
  - Note that, "true" is also a fixpoint of the iterative definition for the weakest precondition in this example

- Weakest precondition if the least fixpoint of the iterative definition

Loops: A Non-terminating Example

- We can check termination using weakest preconditions
  – To check termination set the post-condition to "true"

- Let's look at the following loop: while (i <= i) do i:=i+1
  – Let's compute, wp(while (i <= i) do i:=i+1, true)

  $H_0(i=11) \equiv i > i \wedge \text{true} \equiv \text{false}$

  $H_1(i=11) \equiv \text{wp(if(i<=i) then i:=i+1, false)} \vee \text{false}$

  $\equiv \text{false}$

  – Hence, wp(while (i <= i) do i:=i+1, true) $\equiv$ false
    - i.e., the loop does not terminate

- Remember that halting problem is undecidable
  – We cannot automatically compute weakest preconditions