Weakest-Precondition Reasoning

- Reference: E.W. Dijkstra, A Discipline of Programming, Prentice-Hall, 1976.
- Starting with a post-assertion, what is the *weakest* pre-condition that makes the assertion true?
- In other words, what *must* be true before to make the assertion true after?
- [WP ^ [test&action]] $\rightarrow$ Assertion

What do we mean by “weakest”?

- If A => B but not (B => A), then B is “weaker” than A, and A is “stronger” than B.
- The weakest possible predicate is the one that is identically

  true

  since A => true no matter what A is. Similarly, the strongest possible predicate is

  false

WP for an Assignment Statement

- Consider the assignment statement
  x = F(ξ);
  where ξ denotes the “state vector” (vector of all program variables).
- This statement is “all action”; the test is vacuously true.
- If we want P(x) to be true after, what must be true before?

WP for an Assignment Statement

- The answer is wp(x = F(ξ);, P) = P(F(ξ))
- The general rule is:
  - {P(F(ξ))} x = F(ξ); {P(x)}
- Examples, where we show the assertions within braces, and the wp to be found as ??:
  - {??} x = 5; {y > x}

WP for an Assignment Statement

- The general rule is:
  - {P(F(ξ))} x = F(ξ); {P(x)}
- Example:
  - {y > 5} x = 5; {y > x}
  - i.e. if we want y > x to be true after the assignment x = 5; then we need, at a minimum, y > 5 before the assignment.
  - Identify P(x) as y > x, and F (ξ) as 5, in the rule to get WP: y > 5

WP for an Assignment Statement

- Examples:
  - {??} x = x+y; {y > x}
  - {??} y = 2*y; {y < 5}
  - {??} y = 2*y; {even(y)}

WP for an Assignment Statement

- Examples:
  - {y > x+y} x = x+y; {y > x}
    i.e. wp(x = x+y;, y > x) = y > x+y
  - {2*y < 5} y = 2*y; {y < 5}
    i.e. wp(y = 2*y;, y < 5) = 2*y < 5
  - {even(2*y)} y = 2*y; {even(y)}
    i.e. wp(y = 2*y;, even(y)) = even(2*y)

WP Convention

- It is common to include logic simplification in the WP expression.
- Example: If working in the domain of integers, then
  even(2*y)
  would simplify to
  true
  while
  $i + 1 \leq n$
  would simplify to
  $i < n$

Predicate Transformers

- A program statement can thus be viewed as a "predicate transformer", transforming a post-condition into a weakest pre-condition.

Predicates and State Sets

- A program statement can thus be viewed as a "predicate transformer", transforming a post-condition into a weakest pre-condition.
- But a predicate is just a **set of states**, so that WP transforms the set of states after a statement to a set before.

WP Composition Rule

- Suppose we have two statements in a sequence:
  S; T
- The wp for composite obeys the following equation:
  wp(S; T, P) = wp(S; wp(T, P))
- In other words, predicate transformers compose in a manner similar to functions.

Composition Rule Example

- Consider
  {??} x = z+1; y = x+y; {y > 5}
- wp(y = x+y;, y > 5) = x+y > 5
- wp(x = z+1;, x+y > 5) = z+1+y > 5
- So ?? is
  z+1+y > 5

Composition Rule Example

- Consider the sequence and post-condition
  {??}
  s1 = s1 + s2;
  s2 = s2 + s3;
  s3 = s3 + 6;
  i = i+1;
  {s1=i^3 ÿ s2=(i+1)^3-i^3 ÿ s3 = 6*(i+1)}

- Working backward
  wp(i=i+1; …) =
  {s1=(i+1)^3 ÿ s2=(i+2)^3-(i+1)^3 ÿ s3 = 6*(i+2)}

Composition Rule Example (2)

Working backword from

{s1=(i+1)^3 ÿ s2=(i+2)^3-(i+1)^3 ÿ s3 = 6*(i+2)}

- wp(s3 = s3 + 6; …) =
  {s1=(i+1)^3 ÿ s2=(i+2)^3-(i+1)^3 ÿ (s3+6) = 6*(i+2)}

- wp(s2 = s2 + s3; …) =
  {s1=(i+1)^{3}ÿ(s2+s3)=(i+2)^3-(i+1)^{3}ÿ (s3+6) =6*(i+2)}

- wp(s1 = s1+s2; …) =
  {(s1+s2)=(i+1)^{3}ÿ(s2+s3)=(i+2)^3-(i+1)^{3}ÿ (s3+6) =6*(i+2)}

Using the Composition Rule to Prove a Loop Invariant

- An assertion P is a loop invariant provided that:
  - P is true at the start of the loop
  - P => wp(loop-body, P)

- The second condition above is the same as the verification condition for the loop body.

Example: Using the Composition Rule to Prove a Loop Invariant

- We claim that
  {(s1+s2)=(i+1)^{3}ÿ(s2+s3)=(i+2)^3-(i+1)^{3}ÿ (s3+6) =6*(i+2)}
- is **implied by** the original post-condition of the body:
  {s1=i^3 ÿ s2=(i+1)^3-i^3 ÿ s3 = 6*(i+1)}
- Assume the latter post-condition. Then
  - s1+s2 = i^3+(i+1)^3-i^3
    = (i+1)^3
  - s2+s3 = (i+1)^3-i^3+ 6*(i+1)
    = (i+2)^3-(i+1)^3
  - s3+6 = 6*(i+1)+6
    = 6*(i+2)

} This equality is not so obvious.

Showing the Non-Obvious Equality

- Show $(i+1)^3 - i^3 + 6*(i+1) = (i+2)^3 - (i+1)^3$
- LHS= $(i^3 + 3i^2 + 3i + 1) - i^3 + 6i + 6$

  $= 3i^2 + 9i + 7$
- RHS= $(i^3 + 6i^2 + 12i + 8) - (i^3 + 3i^2 + 3i + 1)$

  $= 3i^2 + 9i + 7$

WP for a Test

- {??} if(Q(ξ)) S else T {P(ξ)}
- wp(if(Q(ξ)) S else T, P)(ξ) =

  $Q(\xi) \rightarrow wp(S, P)(\xi)$ ^
  $\neg Q(\xi) \rightarrow wp(T, P)(\xi)$

WP for a Test - Example

- {??}
  if(x > y) x = x-y; else y = y-x;
  {gcd(x, y) = z}
- wp is

  $(x > y) \rightarrow \overbrace{gcd(x-y, y) = z}^{\text{wp's of the assignment statements}}$ ÿ
  $\neg(x > y) \rightarrow gcd(x, y-x) = z$

WP for a Test - Example (2)

- {??}
  if(x > y) z = x; else z = y;
  {z = max(x, y)}
- wp is

  $(x > y) \rightarrow \overbrace{max(x, y) = x}^{\text{wp's of the assignment statements}}$ ÿ
  $\neg(x > y) \rightarrow max(x, y) = y$
  which simplifies to *true*.

When the *else* part is missing

- If the else part is missing, then T is effectively a “no-op” or “skip”:
  $\xi = \xi;$
- The wp is then
  $\neg Q(\xi) \rightarrow P(\xi)$
  ^ $Q(\xi) \rightarrow$ wp(S, P)(ξ)
- since wp($\xi = \xi$; P) = P

WP for a Test without else

- {??}
  if(x > y) y = x;
  {y = max(x, y)}
- wp is
  $\neg(x > y) \rightarrow \overbrace{y = \max(x, y)}^{\text{wp's of the assignment statements}}$
  ÿ $(x > y) \rightarrow x = \max(x, x)$
  which simplifies to *true*.

Alternate WP for a Test

- wp(if(Q(ξ)) S else T, P)(ξ) =

  (Q(ξ) ^ wp(S, P)(ξ)) $\vee$
  ($\neg$Q(ξ) ^ wp(T, P)(ξ))
- To see that this is equivalent to the previous version, let wp(S, P) be A and wp(T, P) be B. Then we are asking whether (Q ^ A) $\vee$ ($\neg$Q ^ B) is equivalent to (Q $\rightarrow$ A) ^ ($\neg$Q $\rightarrow$ B)

Alternate WP for a Test

- (Q ^ A) $\vee$ ($\neg$Q ^ B) =? (Q $\rightarrow$ A) ^ ($\neg$Q $\rightarrow$ B)
- For Q = true, this becomes A =? A.
- For Q = false, this becomes B =? B
- Therefore the two forms are equivalent.

WP for a Loop

- {??} while(Q) S {P}
- Consider this to be *unrolled* to a cascade of *if*'s (without *else*'s)
- if(Q) {S; if(Q) {S; if(Q) {S; … }}}
- So WP is

  $\neg Q(\xi) \rightarrow P(\xi)$ ^
  $Q(\xi) \rightarrow (wp(S, \neg Q(\xi) \rightarrow P(\xi)$ ^
  $Q(\xi) \rightarrow (wp(S, …))))$
- but this may be difficult to capture in closed form.

Example: WP for a Loop

- {??} while(x > 0) x = x-1; {x == 0}
- WP is

  $\neg(x > 0) \rightarrow \quad x == 0$ ^
  $(x > 0) \rightarrow [\neg(x\text{-}1 > 0) \rightarrow x\text{-}1 == 0$ ^
  $(x\text{-}1 > 0) \rightarrow [\neg(x\text{-}2 > 0) \rightarrow x\text{-}2 == 0$ ^
  $(x\text{-}2 > 0) \rightarrow [… \ ]]]$

which simplifies to

- {??} while(x > 0) x = x-1; {x == 0}
- WP is

  $x \leq 0 \rightarrow x == 0$ ^
  $(x > 0)$ ^ $x \leq 1 \rightarrow x == 1$ ^
  $(x > 1)$ ^ $x \leq 2 \rightarrow x == 2$ ^
  $(x > 2)$ ^ $x \leq 3 \rightarrow x == 3$ ^
  …

which further simplifies to

- {??} while(x > 0) x = x-1; {x == 0}
- WP is
  $x \geq 0$
- In other words, the loop will terminate with x == 0 provided that $x \geq 0$ initially.

Recurrence for Loop WP

- {??} while(Q) S; {P}
  can be expressed as the predicate H(P) = $H_0(P) \wedge H_1(P) \wedge H_2(P) \wedge H_3(P) \wedge \ldots$
- where
  - $H_0(P) = \neg Q \rightarrow P$
  - $H_{k+1}(P) = Q \rightarrow wp(S, H_k(P))$

Recurrence for Loop WP

- {??} while(Q) S; {P}
- In particular, the WP H is the weakest predicate satisfying the recurrence:
  - $H \rightarrow (\neg Q \rightarrow P)$
  - $H \rightarrow (Q \rightarrow wp(S, H))$
- In this sense, H is like a loop invariant, but derived from *post-conditions*.

Example: Recurrence for Loop WP

- {??} while(x > 0) x = x-1; {x==0}
  can be expressed as the predicate H(P) = H_0(x==0) ÿ H_1(x==0) ÿ H_2(x==0) ÿ …
- where
  - $H_0(x{=}{=}0) = \neg\, x > 0 \rightarrow x{=}{=}0$
  - $H_{k+1}(x{=}{=}0) = x > 0 \rightarrow wp(x{=}x{-}1;, H_k(x{=}{=}0))$

Example: Recurrence for Loop WP

- {??} while(x > 0) x = x-1; {x==0}
- Check that H = $x \geq 0$ satisfies the recurrence:
  - $x \geq 0 \rightarrow (\neg\, x > 0 \rightarrow x{=}{=}0)$

  which is valid, and
  - $x \geq 0 \rightarrow (x > 0 \rightarrow wp(x{=}x{-}1;, x \geq 0))$
- But wp(x=x-1;, $x \geq 0$) *is* $x \geq 1$, so we check
  - $x \geq 0 \rightarrow (x > 0 \rightarrow x \geq 1)$, which is true (for integers)

Another way to approach WP for a loop

- wp(while(B) S, Q)
- $(\exists k \geq 0)\ H_k(Q)$
- where
  - $H_0(Q)$ = ¬ B ÿ Q
  - $H_{k+1}(Q)$ = (B ÿ wp(S, $H_k(Q)$)) ∨ (¬B ÿ Q)

Example: Alternate way to approach WP for a loop

- {??} while(x > 0) x = x-1; {x==0}
  can be expressed as the predicate H(P)
  = H_0(x==0) ∨ H_1(x==0) ∨ H_2(x==0) ∨ …
- where
  - H_0(x==0) = ¬ x > 0 ÿ x==0
  - H_{k+1}(x==0) = (x > 0 ÿ wp(x=x-1;, H_k(x==0)))
    ∨ (¬ x > 0 ÿ x==0)
- x == 0 ∨ x==1 ∨ x==2 ∨ ...

More on WP for a Loops

- Note that WP for a loop captures *total* correctness.
- Since it is generally difficult to derive WP in a closed form, we may be content with finding a pre-condition that satisfies the recurrence but is not the weakest.
- Such a condition ***implies*** the weakest.

Practical Example of a Loop WP

- Consider the java code:

```
    for( j = 0; j < a.length; j++ )
      {
  if( a[j] == v )
    {
    break;
    }
   }
 assert: a[j] == v
```

- What is the weakest pre-condition?

Practical Example of a Loop WP

- The weakest pre-condition is:

  ((∃j) (0 ≤ j < a.length) ÿ a[j] == v)

Further Standard Properties of wp

- wp(S, false) = false
- wp(S, true) = condition under which S terminates
- wp(skip, Q) = Q
- wp(abort, Q) = false
- If Q → R then wp(S, Q) → wp(S, R)
- wp(S, Q ÿ R) = wp(S, Q) ÿ wp(S, R)
- wp(S, Q) ∨ wp(S, R) → wp(S, Q ∨ R) (equality holds for *deterministic* S)

Structural Induction

- The **Structural Induction Principle** can also be used for proving correctness.
- It generalizes conventional mathematical induction, in that it is on the formation of **information structures**, such as lists (of which numbers are a special case).
- It has the advantage of proving **total** correctness in one single technique.
- It is useful for functional and logic programs in particular.
- It can also be used for proving properties of information structures themselves.

Structural Induction Proof (1)

- Consider the following rex program:
  - shunt([ ], M) => M;
  - shunt([A | L], M) => shunt(L, [A | M]);
- We want to show:
  - (L)(M) shunt(L, M) returns the M appended to the reverse of L, i.e.
  - (L)(M) shunt(L, M) == append(reverse(L), M)

Structural Induction Proof (2)

- Show by induction “on L”
  (L)(M) shunt(L, M) == append(reverse(L), M)
- Basis: Show it true for L = the empty list:
  - TBS: (M) shunt([ ], M) == append(reverse([ ]), M)
  - From the program, shunt([ ], M) => M.
  - But M == append([ ], M) == append(reverse([ ]), M). QED.

Structural Induction Proof (3)

- We are showing:
  - (L)(M) shunt(L, M) == append(reverse(L), M)
- Induction step: Assume for an arbitrary list L:
  - (M) shunt(L, M) == append(reverse(L), M)
- Show it is true for list [A | L], i.e. show:
  - (M) shunt([A | L], M) ==
    append(reverse([A | L]), M)

Structural Induction Proof (4)

- Show (M) shunt([A | L], M) ==
  append(reverse([A | L]), M).
  - From the program, shunt([A | L], M) returns the result of shunt(L, [A | M]).
  - By the inductive hypothesis, this equals append(reverse(L), [A | M]), so we need to show
    append(reverse(L), [A | M]) ==
    append(reverse([A | L]), M).

Structural Induction Proof (5)

- To show: append(reverse(L), [A | M]) ==
  append(reverse([A | L]), M).
  - append(reverse(L), [A | M]) =
    from definition of append
  - append(reverse(L), append([A], M)) =
    associativity of append
  - append(append(reverse(L), [A]), M) =
    from definition of reverse
  - append(reverse([A | L]), M)

Try to Prove this by Structural Induction

- (a)(b)(c) app(app(a, b), c) ==
  app(a, app(b, c))
- Using the definition of app:

  app([], b) => b;
  app([x|a], b) => [x | app(a, b)];

Structural Induction

- (a)(b)(c) app(app(a, b), c) ==
  app(a, app(b, c))
- Induct on a:
- Basis: app(app([], b), c) ==
  app([], app(b, c))
- By direct evaluation, this reduces to
  app(b, c) == app(b, c),
  which reduces to true.

Structural Induction

- (a)(b)(c) app(app(a, b), c) ==
  app(a, app(b, c))
- Induction Hypothesis:
  (b)(c) app(app(a, b), c) == app(a, app(b, c))
- Induction Conclusion:
  (b)(c) app(app(cons(x, a), b), c)
  == app(cons(x, a), app(b, c))

Structural Induction

- TBS:app(app(cons(x, a), b), c)
  == app(cons(x, a), app(b, c))
- By two symbolic evaluations, based on the definition of app this equation reduces to:
- app(cons(x, app(a, b)), c) == cons(x, app(a, app(b, c)))

Structural Induction

- TBS: app(cons(x, app(a, b)), c) == cons(x, app(a, app(b, c)))
- By one more symbolic evaluations, this reduces to:
  cons(x, app(app(a, b), c)) ==
  cons(x, app(a, app(b, c)))
- Using the induction hypothesis, this is an identity.

"Mathematical Induction" is a special case of Structural Induction

- Mathematical induction says: "To prove a property P for all natural numbers, it suffices to prove:
  - P(0)
  - (n)(P(n)→ P(n+1))"
- This is structural induction where number n+1 is thought to be "constructed" from n by the +1 operator.

"Strong form of Mathematical Induction"

- To prove a property P for all natural numbers, it suffices to prove:
  - (n)(((m<n)P(m)) → P(n))
- The strong form allows use of a stronger induction hypothesis, which may simplify a proof.
- The strong form can be derived from the ordinary form.

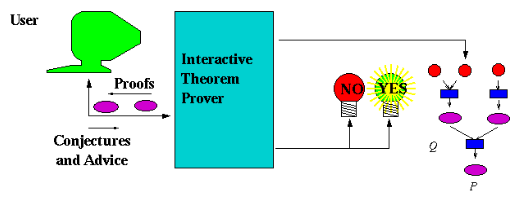
Notes

- Those items to which we appealed as "definitions" on the previous slide could themselves be proved as lemmas using structural induction.
- Automated tools such as ACL2 can be used to do this form of proof on a computer.

Overview of ACL2

- ACL2 = “Applicative Common Lisp 2”
- ACL2 is an interactive theorem prover based on Lisp and structural induction
- History of ACL2:
  - Boyer-Moore Theorem Prover (Edinborough, PARC, UT Austin)
  - Nqthm (Computational Logic Incorporated)
  - ACL2 (UT Austin)

Theorem Prover Use-Case Diagram



ACL2 includes

- Normal Lisp execution
- Symbolic execution
- Automated theorem proving
- Formalism for admitting axioms to the system

Sample Function Definition in ACL2

```
ACL2 !>
(defun app (x y)
  (cond ((endp x) y)
        (t (cons (car x)
              (app (cdr x) y)))))
```

endp checks for the list being empty

Sample Evaluations

```
ACL2 !>(app nil '(x y z))
(X Y Z)


ACL2 !>(app '(1 2 3) '(4 5 6 7))
(1 2 3 4 5 6 7)


ACL2 !>(app '(a b c d e f g) '(x y z))
(A B C D E F G X Y Z)


ACL2 !>(app (app '(1 2) '(3 4)) '(5 6))
(1 2 3 4 5 6)
```

Sample Theorem

This theorem asserts that function app is associative:

```
ACL2!>
      (defthm associativity-of-app
    (equal (app (app a b) c)
         (app a (app b c))))
```

This is just what we proved earlier by Structural Induction

- (equal (app (app a b) c)
      (app a (app b c))))
- In other words,
  (a)(b)(c) app(app(a, b), c) ==
      app(a, app(b, c))

ACL2 Theorem Prover Output

```
  (defthm associativity-of-app
(equal (app (app a b) c)
    (app a (app b c))))
```

Name the formula above *1.

Perhaps we can prove *1 by induction. Three induction schemes are suggested by this conjecture. Subsumption reduces that number to two. However, one of these is flawed and so we are left with one viable candidate.

(continued)

ACL2 Theorem Prover Output

```
We will induct according to a scheme suggested by (APP A B).  If we
let  (:P A B C) denote *1 above then the induction scheme we'll use
is
(AND
  (IMPLIES (AND (NOT (ENDP A))
               (:P (CDR A) B C))
           (:P A B C))
  (IMPLIES (ENDP A) (:P A B C))).
This induction is justified by the same argument used to admit APP,
namely, the measure (ACL2-COUNT A) is decreasing according to the relation
E0-ORD-< (which is known to be well-founded on the domain recognized
by E0-ORDINALP).  When applied to the goal at hand the above induction
scheme produces the following two nontautological subgoals.
```

Simplification of the Induction Step

```
Subgoal *1/2
(IMPLIES (AND (NOT (ENDP A))
              (EQUAL (APP (APP (CDR A) B) C)
                     (APP (CDR A) (APP B C))))
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).

By the simple :definition ENDP we reduce the conjecture to
```

Simplification of the Induction Step

```
Subgoal *1/2'
(IMPLIES (AND (CONSP A)
              (EQUAL (APP (APP (CDR A) B) C)
                     (APP (CDR A) (APP B C))))
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).

But simplification reduces this to T, using the :definition APP, the
:rewrite rules CDR-CONS and CAR-CONS and primitive type
 reasoning.
```

Simplification of the Basis

```
Subgoal *1/1
(IMPLIES (ENDP A)
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/1'
(IMPLIES (NOT (CONSP A))
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).

But simplification reduces this to T, using the :definition APP and
primitive type reasoning.
```

Proof of a Theorem

- Once the theorem is proved, it is saved in the system to be used as a **rewrite rule**.
- The system will henceforth rewrite
  (app (app x y) z)
  as
  (app x (app y z))
- This is not necessarily a good thing.

Controlling Rewrites

- The problem with universal application of a rewrite rule is that it can divert from the main problem.
- For example, resubmitting the previous theorem would cause an infinite loop in the form of repeated application of the rule.
- This can be avoided, as shown next.

Avoiding automatic rule application

```
(defthm associativity-of-app
  (equal (app (app a b) c)
         (app a (app b c)))

  :rule-classes nil)
```

Example Use of the Associativity Theorem

```
(defthm trivial-consequence
     (equal (app (app (app (app x1 x2) (app x3 x4)) (app x5 x6)) x7)
          (app x1 (app (app x2 x3) (app (app x4 x5) (app x6 x7))))))
```

```
ACL2 Warning [Subsume] in ( DEFTHM TRIVIAL-CONSEQUENCE ...):
The previously
added rule ASSOCIATIVITY-OF-APP subsumes the newly proposed :REWRITE
rule TRIVIAL-CONSEQUENCE, in the sense that the old rule rewrites a
more general target.  Because the new rule will be tried first, it
may nonetheless find application.
```

Example Use of the Associativity Theorem

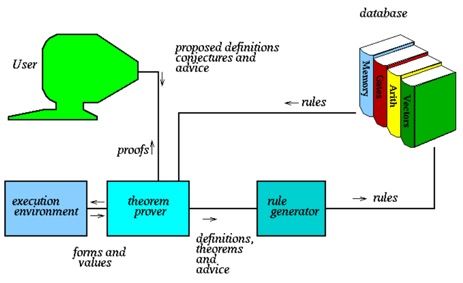
By the simple :rewrite rule ASSOCIATIVITY-OF-APP we reduce the conjecture to

```
Goal'
(EQUAL (APP X1
           (APP X2
               (APP X3 (APP X4 (APP X5 (APP X6 X7))))))
      (APP X1
           (APP X2
               (APP X3 (APP X4 (APP X5 (APP X6 X7))))))).
```

But we reduce the conjecture to T, by primitive type reasoning.

Q.E.D.

ACL2 System Architecture


database
User
proposed definitions conjectures and advice
Memory
Gates
Arith
Vectors
← rules
proofs
execution environment
theorem prover
rule generator
→ rules
forms and values
definitions, theorems and advice