Weakest-Precondition Reasoning

- 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?
- \([\text{WP} \wedge \text{test&action}] \rightarrow \text{Assertion}\)

What do we mean by “weakest”?

- If \(A \Rightarrow B\) but not \((B \Rightarrow A)\), then \(B\) is “weaker” than \(A\), and \(A\) is “stronger” than \(B\).
- The weakest possible predicate is the one that is identically true since \(A \Rightarrow \text{true}\) no matter what \(A\) is. Similarly, the strongest possible predicate is \(\text{false}\).

WP for an Assignment Statement

- Consider the assignment statement \(x = F(\xi)\);
  where \(\xi\) 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 \(\text{wp}(x = F(\xi); \ P) = P(F(\xi))\)
- The general rule is:
  \(\{P(F(\xi))\} \ x = F(\xi); \ {P(x)}\)
- Examples, where we show the assertions within braces, and the wp to be found as ??:
  \(\{??\} \ x = 5; \ {y < x}\)
The general rule is:

\[ \{ P(F(\xi)) \} \ x = F(\xi); \ \{ P(x) \} \]

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 \( F(\xi) \) as 5, in the rule to get WP: \( y > 5 \)

Examples:

\[ \{ y > x+y \} \ x = x+y; \ \{ y > x \} \]
i.e. \( w.p(x = x+y; \ y > x) = y > x+y \)

\[ \{ 2*y < 5 \} \ y = 2*y; \ \{ y < 5 \} \]
i.e. \( w.p(y = 2*y; \ y < 5) = 2*y < 5 \)

\[ \{ \text{even}(2*y) \} \ y = 2*y; \ \{ \text{even}(y) \} \]
i.e. \( w.p(y = 2*y; \ \text{even}(y)) = \text{even}(2*y) \)

It is common to include logic simplification in the WP expression.

Example: If working in the domain of integers, then \( \text{even}(2*y) \)
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:
  
  \[ \text{wp}(S; T, P) = \text{wp}(S; \text{wp}(T, P)) \]

- In other words, predicate transformers compose in a manner similar to functions.

Composition Rule Example

- Consider
  
  \{??\} x = z+1; y = x+y; \{y > 5\}
- \text{wp}(y = x+y; y > 5) = x+y > 5
- \text{wp}(x = z+1; x+y > 5) = z+1+y > 5
- So ?? is
  
  \[ z+1+y > 5 \]
Consider the sequence and post-condition
\[ s1 = s1 + s2; \]
\[ s2 = s2 + s3; \]
\[ s3 = s3 + 6; \]
\[ i = i+1; \]
\[ \{s1=i^3, s2=(i+1)^2, i, s3=6^3(i+1)\} \]

Working backward
\[ wp(i=i+1; ...) = \{s1=(i+1)^3, s2=(i+2)^3-(i+1)^3, s3=6^3(i+2)\} \]

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
  - \( P \Rightarrow wp(\text{loop-body}, P) \)
- 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
  \[ \{s1+s2=(i+1)^3, s2+s3=(i+2)^3-(i+1)^3, s3=6^3(i+1)\} \]
  is implied by the original post-condition of the body:
  \[ \{s1=i^3, s2=(i+1)^2, i, s3=6^3(i+1)\} \]
- Assume the latter post-condition. Then
  \[ s1+s2 = i^3(i+1)^3 \]
  \[ s2+s3 = (i+1)^2(i+2)^3-(i+1)^3 \]
  \[ s3=6^3(i+1)+6^3(i+2) \]
  This equality is not so obvious.
Showing the Non-Obvious Equality

- Show \((i+1)^3 + 6\cdot(i+1) = (i+2)^3 - (i+1)^3\)
- LHS = \((i^3 + 3i^2 + 3i + 1)^3 + 6i + 6 = 3i^2 + 9i + 7\)
- RHS = \((i^3 + 6i^2 + 12i + 8) - (i^3 + 3i^2 + 3i + 1) = 3i^2 + 9i + 7\)

WP for a Test

- \{??\} if(\(Q(\xi)\)) S else T \{P(\xi)\}
- \(wp(if(Q(\xi)) S else T, P)(\xi) = Q(\xi) \rightarrow wp(S, P)(\xi) \land \neg Q(\xi) \rightarrow wp(T, P)(\xi)\)

WP for a Test - Example

- \{??\}
  if(\(x > y\)) \(x = x - y\); else \(y = y - x\);
  \(\{gcd(x, y) = z\}\)
- \(wp is \ wp's \ of \ the \ assignment \ statements\)
  \( (x > y) \rightarrow gcd(x-y, y) = z \)
  \(\neg (x > y) \rightarrow gcd(x, y-x) = z \)

WP for a Test - Example (2)

- \{??\}
  if(\(x > y\)) \(z = x\); else \(z = y\);
  \(\{z = max(x, y)\}\)
- \(wp is \ wp's \ of \ the \ assignment \ statements\)
  \( (x > y) \rightarrow max(x, y) = x \)
  \(\neg (x > y) \rightarrow max(x, y) = y \)
  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
  \[ \neg Q(\xi) \rightarrow P(\xi) \]
  \[ \land Q(\xi) \rightarrow wp(S, P)(\xi) \]
- since \( wp(\xi = \xi, P) = P \)

WP for a Test without else

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

Alternate WP for a Test

- \( wp(if(Q(\xi) \land S \text{ else } T, P)(\xi) = \)
  \[ (Q(\xi) \land wp(S, P)(\xi)) \lor \]
  \[ (\neg Q(\xi) \land wp(T, P)(\xi)) \]
- 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 \( Q \land A \lor (\neg Q \land B) \) is equivalent to \( Q \rightarrow A \land (\neg Q \rightarrow B) \)

Alternate WP for a Test

- \( (Q \land A) \lor (\neg Q \land B) =? (Q \rightarrow A) \land (\neg Q \rightarrow B) \)
- 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
  
  \[
  \neg (x > 0) \rightarrow x == 0 ^
  (x > 0) \rightarrow [\neg (x > 0) \rightarrow x == 0 ^
  (x > 0) \rightarrow [\neg (x > 0) \rightarrow x == 0 ^
  (x > 0) \rightarrow [\neg (x > 0) \rightarrow x == 0 ^
  (x > 0) \rightarrow [\neg (x > 0) \rightarrow x == 0 ^
  ...
  \]
  
  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) \land H_1(P) \land H_2(P) \land H_3(P) \land \ldots
- where
  - H_0(P) = \neg Q \rightarrow P
  - H_{k+1}(P) = Q \rightarrow wp(S, H_k(P))

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) \land H_1(x==0) \land H_2(x==0) \ldots
- 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))

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

- $\text{wp}(\text{while}(B \ S, Q))$
- $(\exists k \geq 0) H_k(Q)$
- where
  - $H_0(Q) = \neg B \land Q$
  - $H_{k+1}(Q) = (B \land \text{wp}(S, H_k(Q))) \lor (\neg B \land Q)$

Example: Alternate way to approach WP for a loop

- $\{??\} \text{while}(x > 0) x = x - 1; \{x==0\}$
  - can be expressed as the predicate $H(P)$
    - $H_0(x==0) \lor H_1(x==0) \lor H_2(x==0) \lor \ldots$
  - where
    - $H_0(x==0) = \neg x > 0 \land x==0$
    - $H_{k+1}(x==0) = (x > 0 \land \text{wp}(x=x-1; \ H_k(x==0))) \lor (\neg x > 0 \land x==0)$
    - $x == 0 \lor x==1 \lor x==2 \lor \ldots$

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:
  ```java
  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:
  
  \[(\exists j) \ (0 \leq j < a\text{.length}) \ \text{a}[j] == v)\]

**Further Standard Properties of wp**

- \(wp(S, \text{false}) = \text{false}\)
- \(wp(S, \text{true}) = \text{condition under which S terminates}\)
- \(wp(\text{skip}, Q) = Q\)
- \(wp(\text{abort}, Q) = \text{false}\)
- If \(Q \rightarrow R\) then \(wp(S, Q) \rightarrow wp(S, R)\)
- \(wp(S, Q \land R) = wp(S, Q) \land wp(S, R)\)
  (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:
  - \(\text{shunt}([], M) \Rightarrow M;\)
  - \(\text{shunt}([A \mid L], M) \Rightarrow \text{shunt}(L, [A \mid M]);\)

- We want to show:
  - \(\text{shunt}(L, M) \text{ returns the M appended to the reverse of L, i.e.}\)
  - \(\text{shunt}(L, M) = \text{append}(\text{reverse}(L), M)\)
### Structural Induction Proof (2)

- **Show by induction “on L”**
  - \((L)(M)\) \(\text{shunt}(L, M) == \text{append}(\text{reverse}(L), M)\)

- **Basis:** Show it true for \(L = \) the empty list:
  - **TBS:** \((L)(M)\) \(\text{shunt}([\ ], M) == \text{append}(\text{reverse}([\ ]), M)\)
  - From the program, \(\text{shunt}([\ ], M) \Rightarrow M\).
  - But \(M == \text{append}([\ ], M) == \text{append}(\text{reverse}([\ ]), M)\). QED.

### Structural Induction Proof (3)

- We are showing:
  - \((L)(M)\) \(\text{shunt}(L, M) == \text{append}(\text{reverse}(L), M)\)

- **Induction step:** Assume for an arbitrary list \(L:\)
  - \((L)(M)\) \(\text{shunt}(L, M) == \text{append}(\text{reverse}(L), M)\)

- Show it is true for list \([A | L]\), i.e. show:
  - \((L)(M)\) \(\text{shunt}([A | L], M) == \text{append}(\text{reverse}([A | L]), M)\)

### Structural Induction Proof (4)

- **Show** \((L)(M)\) \(\text{shunt}([A | L], M) == \text{append}(\text{reverse}([A | L]), M)\).

- From the program, \(\text{shunt}([A | L], M)\) returns the result of \(\text{shunt}(L, [A | M])\).

- By the inductive hypothesis, this equals \(\text{append}(\text{reverse}(L), [A | M])\), so we need to show
  - \(\text{append}(\text{reverse}(L), [A | M]) == \text{append}(\text{reverse}([A | L]), M)\).

### Structural Induction Proof (5)

- **To show:** \(\text{append}(\text{reverse}(L), [A | M]) == \text{append}(\text{reverse}([A | L]), M)\).

- \(\text{append}(\text{reverse}(L), [A | M]) == \text{append}(\text{reverse}(L), [A | M])\) from **definition of append**

- \(\text{append}(\text{reverse}(L), [A | M]) == \text{append}(\text{reverse}(L), [A | M])\) from **definition of append**

- \(\text{append}(\text{reverse}(L), [A | M]) == \text{append}(\text{reverse}(L), [A | M])\) from **definition of append**
Try to Prove this by Structural Induction

- \((a)(b)(c)\) \(\text{app}(\text{app}(a, b), c) == \text{app}(a, \text{app}(b, c))\)

- Using the definition of \(\text{app}\):

  \[
  \text{app}([], b) \Rightarrow b; \quad \text{app}([x|a], b) \Rightarrow [x | \text{app}(a, b)];
  \]

Structural Induction

- \((a)(b)(c)\) \(\text{app}(\text{app}(a, b), c) == \text{app}(a, \text{app}(b, c))\)

- Induct on \(a\):
  
  **Basis:** \(\text{app}(\text{app}([], b), c) == \text{app}([], \text{app}(b, c))\)

  By direct evaluation, this reduces to \(\text{app}(b, c) == \text{app}(b, c)\), which reduces to true.

- **Induction Hypothesis:**

  \((b)(c)\) \(\text{app}(\text{app}(a, b), c) == \text{app}(a, \text{app}(b, c))\)

- **Induction Conclusion:**

  \((b)(c)\) \(\text{app}(\text{app}(\text{cons}(x, a), b), c) == \text{app}(\text{cons}(x, a), \text{app}(b, c))\)

- **TBS:** \(\text{app}(\text{app}(\text{cons}(x, a), b), c) == \text{app}(\text{cons}(x, a), \text{app}(b, c))\)

  By two symbolic evaluations, based on the definition of \(\text{app}\) this equation reduces to:
  
  \(\text{app}(\text{cons}(x, \text{app}(a, b)), c) == \text{cons}(x, \text{app}(a, \text{app}(b, c)))\)
### Structural Induction

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

### “Mathematical Induction”

- Mathematical induction says: “To prove a property \( P \) for all natural numbers, it suffices to prove:
  - \( P(0) \)
  - \( (n) (P(n) \rightarrow P(n+1)) \)
- 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:
  - \( (n) (\forall m < n) P(m) \rightarrow P(n) \)
- 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)

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 \(\vdash\) (app nil '(x y z))

\((X \ Y \ Z)\)

ACL2 \(\vdash\) (app '(1 2 3) '(4 5 6 7))

\((1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7)\)

ACL2 \(\vdash\) (app '(a b c d e f g) '(x y z))

\((A \ B \ C \ D \ E \ F \ G \ X \ Y \ Z)\)

ACL2 \(\vdash\) (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 \(\vdash\)

\(\text{(defthm associativity-of-app)}\)

\(\text{(equal (app (app a b) c)}\)

\(\text{(app a (app b c))))}\)

---

### This is just what we proved earlier by Structural Induction

- (equal (app (app a b) c)
  (app a (app b c)))

- In other words,
  \((a)(b)(c) \text{app(app(a, b), c) == app(a, app(b, c))}\)

---

### ACL2 Theorem Prover Output

ACL2 \(\vdash\)

\(\text{(defthm associativity-of-app)}\)

\(\text{(equal (app (app a b) c)}\)

\(\text{(app a (app b c))))}\)

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)
We will induct according to a scheme suggested by \((\text{APP } A \ B)\). If we let \((P\ A\ B\ C)\) denote *1 above then the induction scheme we'll use is:
\[
(\text{AND}
\begin{align*}
(\text{IMPLIES } & (\text{AND } (\text{NOT } \text{ENDP} \ A)) \\
(\text{AND} & (P\ CDR\ A) \ B\ C)) \\
(\text{AND} & (P\ A\ B\ C))
\end{align*}
\]

This induction is justified by the same argument used to admit \APP, namely, the measure \((\text{ACL2} - \text{COUNT} \ A)\) is decreasing according to the relation \(E0\text{-ORD}<\) (which is known to be well-founded on the domain recognized by \E0\text{-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
\[
(\text{IMPLIES } (\text{AND } (\text{NOT } \text{ENDP} \ A)) \\
(\text{EQUAL } \text{APP} (\text{APP} (\text{CDR} \ A) \ B) \ C) \\
(\text{APP } (\text{CDR} \ A) \ (\text{APP} B \ C))) \\
(\text{EQUAL } \text{APP} (\text{APP} A \ B) \ C) \\
(\text{APP } A \ (\text{APP} B \ C))).
\]

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

**Simplification of the Basis**

Subgoal *1/1
\[
(\text{IMPLIES } \text{ENDP} \ A) \\
(\text{EQUAL } \text{APP} (\text{APP} A \ B) \ C) \\
(\text{APP } A \ (\text{APP} B \ C))).
\]

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