

Lecture 39: Program Verification

Announcement: Come to class on Friday to fill out the course survey with the help of HKN, and get extra credit.

Extending Static Semantics

- Project 2 considered selected static properties of programs, both of which assisted in translating the program.
 - Scope analysis figured out what identifiers meant.
 - Type analysis figured out what representations to use for certain data.
- But type analysis served the additional function of discovering certain inconsistencies in a program before execution.
- These are not the only error-finding analyses possible before program execution.
- The subject of *program verification* considers the internal consistency of more general static properties of programs.
- The study of formal program verification began in the 1960s.

Basic Goal

- The idea is to detect errors in programs before execution and thus to increase our confidence in our programs' correctness.
- Here, "error" is potentially much broader than it was in Project 2, and includes such things as failing to conform to a specification of what the program is intended to do.
- Today, we'll take an introductory look at one technique for this purpose, known as *axiomatic semantics*.
- Here, we are interested in statements of the form

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

where P and Q are *assertions* about the program statement and S is a piece of program text.

- This statement means "If P is true just before statement S is executed and S terminates, then at that point Q will be true."
- It asserts the *weak correctness* of S with respect to *precondition* P and *postcondition* Q .
- Strong correctness is the same, but also requires that S terminate.

Weakest Liberal Preconditions

- In order for

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

to be true, it suffices to show that $P \implies ?(S, Q)$. That is, P implies some logical assertion that depends on S and Q .

- The usual name for '?' is **wlp**, for *weakest liberal precondition*.
- Here, the term "weakest" means "least restrictive" or "most general", and "liberal" refers to the fact that this precondition need not guarantee termination of S .
- Another notation, $\text{wp}(S, Q)$, or *weakest precondition*, is a bit stronger than the wlp; it implies both the wlp and termination of S .
- We call wlp and wp *predicate transformers*, because they transform the logical expression Q into another logical expression.
- By defining wlp or wp for all statements in a language, we effectively define the dynamic semantics of the language.

Examples of Predicate Transformations (I)

- We start with the most obvious:

$$\text{wlp}(\text{pass}, Q) \equiv Q$$

- That is, the least restrictive condition that guarantees that Q is true after executing `pass` in Python is Q itself.
- Since `pass` always terminates, in this case

$$\text{wp}(\text{pass}, Q) \equiv Q$$

as well.

- Sequencing is also easy:

$$\text{wlp}(S_1; S_2, Q) \equiv \text{wlp}(S_1, \text{wlp}(S_2, Q))$$

or basically composition of `wlp`.

Examples of Predicate Transformations (II)

- If-then-else results in essentially a case analysis:

$$\begin{aligned} & \text{wlp}(\text{if } C \text{ then } S_1 \text{ else } S_2 \text{ fi}, Q) \\ & \equiv \\ & (C \implies \text{wlp}(S_1, Q)) \wedge (\neg C \implies \text{wlp}(S_2, Q)) \end{aligned}$$

- Or

"The weakest liberal precondition insuring that Q is true after **if** C **then** S_1 **else** S_2 **fi** is that C being true must ensure that Q will be true after S_1 and that C being false must ensure that Q is true after executing S_2 ."

- I am playing a bit fast and loose with notation here. The expression C is in the programming language, whereas Q is in whatever *assertion language* we are using to talk about programs written in that language.
- For the purposes of this lecture, we'll ignore the problems that can arise here.
- Similarly, assume C and other expressions have no side-effects.

Examples of Predicate Transformations (III)

- Assignment starts to get interesting.
- After executing $X = E$, of course, X will have value E had before the assignment.
- So for Q to be true after the assignment, it must have been true before as well, if we substitute the value of E for X .
- Formally,

$$\text{wlp}(X = E, Q) \equiv Q[E/X]$$

where the notation $A[\alpha/\beta]$ means “the logical expression A with all (free) instances of β replaced by α .”

- For example,

$$\text{wlp}(X = X + 1, X > 2) \equiv (X + 1) > 2.$$

Examples of Predicate Transformations (IV)

- The predicate transformations we've seen so far can all be done completely mechanically by operations on the ASTs representing the statements and assertions (for example).
- The same could be done for **while**, but would require extending the logical language used for assertions for every **while** statement in the program. For various reasons, that is undesirable.
- So usually, finding the wlp for **while** statements requires a little inventing from the programmer, in the form of a *loop invariant*.
- A loop invariant is an assertion at the beginning of the loop.
- The invariant assertion is intended to be true whenever the program is just about to (re)check the conditional test of the loop.

Rule for While Loops

- If we let the label W stand for the **while** statement

$\text{while } C \text{ do } S \text{ od}$

and let I_w stand for the (alleged) loop invariant the programmer provides for this loop, we get the simple rule:

$$\text{wlp}(W, Q) \equiv I_w$$

assuming we can prove that I_w really is a loop invariant: that is,

$$(C \wedge I_w \implies \text{wlp}(S, I_w)) \wedge (\neg C \wedge I_w \implies Q)$$

- This makes sense, because it means that if (a) I_w is true as a precondition of the loop, and if (b) whenever I_w and the loop condition are true, executing the loop body maintains I_w (hence the name “invariant”), and finally if (c) the loop condition C becomes false so that the loop exits, then Q must be true.

Example

- Consider an annotated program for computing x^n :

```
{  $n \geq 0 \wedge x > 0$  }  
k = n; z = x; y = 1;  
while k > 0 do  
  { Invariant:  $y \cdot z^k = x^n \wedge z > 0 \wedge k \geq 0$  }  
  if odd(k) then y = y * z; fi  
  z = z * z;  
  k = k // 2;  
od  
{  $y = x^n$  }
```

- So the wlp of the loop is (proposed to be) $y \cdot z^k = x^n \wedge z > 0 \wedge k \geq 0$.
- And therefore, the wlp of the whole program is

$$1 \cdot x^n = x^n \wedge x > 0 \wedge n \geq 0$$

(apply the assignment rule three times).

- This is obviously implied by $n \geq 0 \wedge x > 0$. So far, so good.

Example, Continued

```
{  $n \geq 0 \wedge x > 0$  }  
k = n; z = x; y = 1;  
while k > 0 do  
  { Invariant:  $y \cdot z^k = x^n \wedge z > 0 \wedge k \geq 0$  }  
  if odd(k) then y = y * z; fi  
  z = z * z;  
  k = k // 2;  
od  
{  $y = x^n$  }
```

- Now we need to show that the loop invariant really does imply Q (in this case, $y = x^n$) when the loop ends. In other words:

$$k \leq 0 \wedge y \cdot z^k = x^n \wedge z > 0 \wedge k \geq 0 \implies y = x^n$$

But since the left side of the implication means that k must be 0, this too is obvious.

Example, Continued

```
{  $n \geq 0 \wedge x > 0$  }  
k = n; z = x; y = 1;  
while k > 0 do  
  { Invariant:  $y \cdot z^k = x^n \wedge z > 0 \wedge k \geq 0$  }  
  if odd(k) then y = y * z; fi  
  z = z * z;  
  k = k // 2;  
od  
{  $y = x^n$  }
```

- This leaves just the invariance of the alleged invariant to show:

$$k > 0 \wedge y \cdot z^k = x^n \wedge z > 0 \wedge k \geq 0 \implies \text{wlp}(S, y \cdot z^k = x^n \wedge z > 0 \wedge k \geq 0)$$

where S is the body of the loop.

- I'll leave it to you to apply the conditional, pass, and assignment rules (with a little math) to show this is true.

Termination

- We actually have the tools to find the “strong” version of wlp (also implying termination):

$$\text{wp}(S, Q) \equiv \text{wlp}(S, Q) \wedge \neg \text{wlp}(S, \text{false})$$

- (Huh? Why does this work?)
- More usual technique is to use *variant expressions* in the important places (like loops):

```
while C do
  {  $e = e_0$  }
  S
  {  $e < e_0$  }
```

where e is an expression whose value is in a *well-founded set* (such as the non-negative integers), where all descending sequences of values must have finite length.

Limitations

- Even this small example involves a lot of tedious detail.
- Machine assistance helps “reduce” the problem to logic, but for general programs the resulting assertions are at best challenging for current theorem-proving techniques.
- Furthermore, it is tedious and error-prone to come up with formal specifications (pre- and post-conditions and invariants) for even moderately sized programs.
- Nevertheless, some applications (like safety-critical software) warrant such efforts.
- But for general programs, the verification enterprise fell out of favor in the 1980s.

Rebirth

- However, by limiting our objectives, there are numerous uses for the machinery described here.
- For example, there are certain *program properties* that are useful to verify:
 - Is this array index always in bounds here?
 - Is this pointer always non-null here?
 - Does this concurrent program ever deadlock?

- Thus a compiler could (in effect) insert assertions in front of certain statements:

```
{ i ≥ 0 ∧ i < A.length }  
A[i] = E;
```

And then verify a piece of the program to show the assertions are always true.

- Not only shows the program does not cause exceptions, but allows the compiler to avoid generating code to check the value of *i*.