Lecture 26: Program Verification

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

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

  where P and Q are *assertions* about the program state and $\mathcal{S}$ is a piece of program text.
- This statement means "If P is true just before statement $\mathcal{S}$ is executed and $\mathcal{S}$ terminates, then at that point Q will be true."
- It asserts the *weak correctness* of $\mathcal{S}$ with respect to *precondition* P and *postcondition* Q.
- Strong correctness is the same, but also requires that $\mathcal{S}$ terminate.

Weakest Liberal Preconditions

- In order for

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

to be true, it suffices to show that $P \implies \text{wlp}(\llbracket\mathcal{S}\rrbracket, Q)$. That is, P implies some logical assertion that depends on $\mathcal{S}$ and Q.

- wlp stands 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, wp($\llbracket\mathcal{S}\rrbracket$, Q), or *weakest precondition*, is a bit stronger than the wlp; it implies both the wlp and termination of $\mathcal{S}$.
- 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.

Program Verification Conditions

- Again, our goal is to be able to prove assertions such as

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

  where P is a logical assertion stating conditions we assume before we execute program text $\mathcal{S}$ and Q asserts conditions we *want* to be true afterwards.

- When we convert this to $P \Longrightarrow \textsf{wlp}(\llbracket\mathcal{S}\rrbracket, Q)$, we will have translated a program statement and some mathematical assertions into a single mathematical assertion that we (hopefully) can prove—a *verification condition.*

- We'll proceed in recursive fashion to define wlp for all types of statement in our programming language.

Predicate Transformations: pass

- We start with the most obvious:

$$\textbf{wlp}(\llbracket \text{pass} \rrbracket, Q) \equiv ?$$

- That is, the least restrictive condition that guarantees that Q is true after executing pass in Python is ?.

Predicate Transformations: pass

- We start with the most obvious:

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

- 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

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

as well.

Predicates: Sequencing

- Sequencing is also easy:

$$\mathsf{wlp}(\,[\![\mathcal{S}_1;\mathcal{S}_2]\!],\; Q) \equiv\, ?$$

- Here, we reason that in order for Q to be true after $\mathcal{S}_2$, we must establish that $\mathsf{wlp}(\,[\![\mathcal{S}_2]\!], Q)$ is true after executing $\mathcal{S}_1$.

Predicates: Sequencing

- Sequencing is also easy:

$$\mathsf{wlp}(\,[\![\mathcal{S}_1;\mathcal{S}_2]\!],\;Q) \equiv \mathsf{wlp}(\,[\![\mathcal{S}_1]\!],\mathsf{wlp}(\,[\![\mathcal{S}_2]\!],Q))$$

- Here, we reason that in order for Q to be true after $\mathcal{S}_2$, we must establish that $\mathsf{wlp}(\,[\![\mathcal{S}_2]\!],Q)$ is true after executing $\mathcal{S}_1$.
- So what we need is basically composition of wlp.
- Again, this works as well if we replace wlp with wp.

Predicate Transformations: Assignment

- 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,

$$\mathsf{wlp}(\llbracket \texttt{X} = E \rrbracket, Q) \equiv Q[E/\texttt{X}]$$

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

- For example,

$$\mathsf{wlp}(\llbracket \texttt{X} = \texttt{X} + 1 \rrbracket, X > 2) \equiv (X + 1) > 2.$$

 That is, for $X > 2$ to be true after the assignment, $X + 1$ had to be > 2 (or $X > 1$) before the assignment.

Predicate Transformations: If-Then-Else

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

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

- 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."

- Just plain **if-then** is the same, with the **else** clause being **pass**.

Example of If-Then-Else

$$
\begin{aligned}
&wlp(\llbracket \textbf{if } x > y \textbf{ then } z = x \textbf{ else } z = y \rrbracket, z \geq x \wedge z \geq y)\\
&\quad\equiv\ (x > y \Longrightarrow wlp(\llbracket z = x \rrbracket, z \geq x \wedge z \geq y))\\
&\quad\quad\ \wedge (x \leq y \Longrightarrow wlp(\llbracket z = y \rrbracket, z \geq x \wedge z \geq y))\\
&\quad\equiv\ (x > y \Longrightarrow x \geq x \wedge x \geq y)\\
&\quad\quad\ \wedge (x \leq y \Longrightarrow y \geq x \wedge y \geq y)\\
&\quad\equiv\ \text{true}
\end{aligned}
$$

- In other words, *any* assertion will imply this wlp; the **if** statement always works.

A Technical Caution

- I have been playing a bit fast and loose with notation here. In

  $$\text{wlp}(\llbracket \textbf{if } C \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}\rrbracket, Q)$$

  The expression C is in the programming language, whereas Q is in whatever mathematical *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.

- In particular, we will assume that C and other expressions have no side-effects, don't cause exceptions, and always terminate.

Predicate Transformations: While

- The predicate transformations we've seen so far can all be done completely mechanically by operations on the ASTs representing the ststements and assertions (for example).
- The same could be done for **while**, but would require adding to the assertion language for each **while** statement in the program. For various reasons, that is undesirable.
- So usually, finding the wlp for **while** statements requires a little help 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 and let I_w stand for the (alleged) loop invariant that the programmer provides:

  ```
  assert I_w;
  W: while C do S; assert I_w; od
  ```

  we get this simple rule:

  $$\textsf{wlp}([\![W]\!], Q) \equiv I_w$$

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

  $$(C \wedge I_w \implies \textsf{wlp}([\![S]\!], I_w)) \wedge (\neg C \wedge I_w \implies Q)$$

- This makes sense, because it means that

  (a) if I_w is true just before the loop, and

  (b) if whenever I_w and the loop condition are true, executing the loop body maintains I_w (hence the name "invariant"), and finally

  (c) if I_w is true and the loop condition C becomes false so that the loop exits, then Q must be true.

  then Q must be true (if and) when the loop exits.

Example

- Consider an annotated program for computing x^n:

```
{ n ≥ 0 ∧ x > 0   }
k = n; z = x; y = 1;
{ Invariant: y · z^k = x^n ∧ z > 0 ∧ k ≥ 0 }
while k > 0 do
    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) $y \cdot z^k = x^n \wedge z > 0 \wedge k \geq 0$.
- Applying the assignment rule three times tells us that the wlp of the whole program is

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

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

Example, Correctness at Termination

```
{ n ≥ 0 ∧ x > 0 }
k = n; z = x; y = 1;
{ Invariant: y · z^k = x^n ∧ z > 0 ∧ k ≥ 0 }
while k > 0 do
    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:

$$k \leq 0 \land y \cdot z^k = x^n \land z > 0 \land k \geq 0 \Longrightarrow y = x^n$$

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

Example: Invariant (I)

```
{ n ≥ 0 ∧ x > 0  }
k = n; z = x; y = 1;
{ Invariant: y · z^k = x^n ∧ z > 0 ∧ k ≥ 0 }
while k > 0 do
    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:

$$k > 0 \wedge y \cdot z^k = x^n \wedge z > 0 \wedge k \geq 0 \Longrightarrow \textbf{wlp}([\![S]\!], y \cdot z^k = x^n \wedge z > 0 \wedge k \geq 0)$$

where S is the body of the loop.

- This simplifies to

$$y \cdot z^k = x^n \wedge z > 0 \wedge k > 0 \Longrightarrow \textbf{wlp}([\![S]\!], y \cdot z^k = x^n \wedge z > 0 \wedge k \geq 0)$$

Example: Invariant (II)

```
{ n ≥ 0 ∧ x > 0  }
k = n; z = x; y = 1;
{ Invariant: y · z^k = x^n ∧ z > 0 ∧ k ≥ 0 }
while k > 0 do
    if odd(k) then y = y * z; fi
    z = z * z;
    k = k // 2;
od
{ y = x^n }
```

- From

$$y \cdot z^k = x^n \wedge z > 0 \wedge k > 0 \Longrightarrow \textsf{wlp}([\![S]\!], y \cdot z^k = x^n \wedge z > 0 \wedge k \geq 0)$$

we can apply the assignment rule twice to get

$$y \cdot z^k = x^n \wedge z > 0 \wedge k > 0 \Longrightarrow \textsf{wlp}([\![\textsf{if} \ldots \textsf{fi}]\!], y \cdot (z^2)^{\lfloor k/2 \rfloor} = x^n \wedge z^2 > 0 \wedge \lfloor k/2 \rfloor \geq 0)$$

or

$$y \cdot z^k = x^n \wedge z > 0 \wedge k > 0 \Longrightarrow \textsf{wlp}([\![\textsf{if} \ldots \textsf{fi}]\!], y \cdot z^{2 \lfloor k/2 \rfloor} = x^n \wedge z^2 > 0 \wedge \lfloor k/2 \rfloor \geq 0)$$

Example: Invariant (III)

```
{ n ≥ 0 ∧ x > 0  }
k = n; z = x; y = 1;
{ Invariant: y · z^k = x^n ∧ z > 0 ∧ k ≥ 0 }
while k > 0 do
    if odd(k) then y = y * z; fi
    z = z * z;
    k = k // 2;
od
{ y = x^n }
```

- Finally, the conditional:

$$y \cdot z^k = x^n \wedge z > 0 \wedge k > 0 \Longrightarrow \textbf{wlp}(\llbracket \textbf{if} \ldots \textbf{fi} \rrbracket, y \cdot z^{2\lfloor k/2 \rfloor} = x^n \wedge z^2 > 0 \wedge \lfloor k/2 \rfloor \geq 0)$$

becomes

$$\begin{aligned}
& y \cdot z^k = x^n \wedge z > 0 \wedge k > 0 \Longrightarrow \\
& \quad \textbf{odd}(k) \Longrightarrow \textbf{wlp}(\llbracket y = \ldots \rrbracket, y \cdot z^{2\lfloor k/2 \rfloor} = x^n \wedge z^2 > 0 \wedge \lfloor k/2 \rfloor \geq 0) \\
& \quad \wedge \neg\textbf{odd}(k) \Longrightarrow \textbf{wlp}(\llbracket \textbf{pass} \rrbracket, y \cdot z^{2\lfloor k/2 \rfloor} = x^n \wedge z^2 > 0 \wedge \lfloor k/2 \rfloor \geq 0) \\
& y \cdot z^k = x^n \wedge z > 0 \wedge k > 0 \Longrightarrow \\
& \quad \textbf{odd}(k) \Longrightarrow y \cdot z \cdot z^{2\lfloor k/2 \rfloor} = x^n \wedge z^2 > 0 \wedge \lfloor k/2 \rfloor \geq 0 \\
& \quad \wedge \neg\textbf{odd}(k) \Longrightarrow y \cdot z^{2\lfloor k/2 \rfloor} = x^n \wedge z^2 > 0 \wedge \lfloor k/2 \rfloor \geq 0
\end{aligned}$$

Example: Invariant (IV)

```
{ n ≥ 0 ∧ x > 0  }
k = n; z = x; y = 1;
{ Invariant: y·z^k = x^n ∧ z > 0 ∧ k ≥ 0 }
while k > 0 do
    if odd(k) then y = y * z; fi
    z = z * z;
    k = k // 2;
od
{ y = x^n }
```

- And we are left to check:

$$
\begin{array}{l}
y \cdot z^k = x^n \wedge z > 0 \wedge k > 0 \Longrightarrow \\
\qquad \textsf{odd}(k) \Longrightarrow y \cdot z \cdot z^{2\lfloor k/2 \rfloor} = x^n \wedge z^2 > 0 \wedge \lfloor k/2 \rfloor \geq 0 \\
\qquad \wedge \neg\textsf{odd}(k) \Longrightarrow y \cdot z^{2\lfloor k/2 \rfloor} = x^n \wedge z^2 > 0 \wedge \lfloor k/2 \rfloor \geq 0 \\
y \cdot z^k = x^n \wedge z > 0 \wedge k > 0 \Longrightarrow \\
\qquad \textsf{odd}(k) \Longrightarrow y \cdot z^k = x^n \\
\qquad \wedge \neg\textsf{odd}(k) \Longrightarrow y \cdot z^k = x^n \\
y \cdot z^k = x^n \wedge z > 0 \wedge k > 0 \Longrightarrow y \cdot z^k = x^n
\end{array}
$$

which is obvious. (Whew!)

Termination

- For all the rules except for the **while** statement, we could have used wp rather than wlp, but the **while** rule obviously requires that we be "liberal."
- We actually have the tools to find the "strong" version of wlp (also implying termination):

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

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

```
while C do
    { e = e0  }
    S
    { e < e0  }
```

where e (the *variant*) 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.
- Consider, for example, that our rules ignored the possibility of integer overflow (i.e., treated computer integer arithmetic as if it were on the mathematical integers.)
- 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:

  $\{\ i \geq 0 \wedge i < A.\mathtt{length}\ \}$
  ```
  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`.

Predicate Transformations: Functions

- The predicate transformations we've seen so far can all be done completely mechanically by operations on the ASTs representing the ststements and assertions (for example).
- With functions, good methodology suggests that the *implementer* should be documenting the desired semantics of the procedure.
- To keep it simple, let's consider procedures without parameters and that don't return values, and are executed for their side-effects alone.
- So we imagine a procedure declaration such as

```
def f():
    """Precondition:  P_f
    Postcondition:  Q_f"""
    body of f
```

- Meaning "if at the point of call, the precondition P_f is true, then on exit, the postcondition Q_f will be true (assuming termination.)"

Verifying Function Definitions

- It's pretty clear how to show that the definition itself makes sense—the *verification condition* for the definition of `f`:

$$P_f \Longrightarrow \mathsf{wlp}(\llbracket\mathsf{body\ of\ f}\rrbracket, Q_f).$$

Functions Calls

- You might think the rule for function calls is now simply:

$$\textbf{wlp}(\llbracket f() \rrbracket, Q) \equiv P_f \wedge (Q_f \Longrightarrow Q)$$

- That is, "the weakest conditions under which calling `f` will cause Q to be true is that `f`'s precondition is satisfied and its postcondition implies Q."
- But this doesn't work. Why not?

Function Calls (II)

- Consider

```
def g():
    """Precondition: x < 0.  Postcondition: x > 0"""
    global x
    x = -x
```

- Then the rule **wlp**$(\llbracket f() \rrbracket, Q) \equiv P_f \wedge (Q_f \Longrightarrow Q)$ would give us

$$wlp(\llbracket g() \rrbracket, x < 0) \equiv x < 0 \wedge (x > 0 \Longrightarrow x < 0) \equiv x < 0$$

- But this tells us that

$$\{x < 0\}\ g()\ \{x < 0\}$$

which is clearly wrong.

- The problem is that we are trying to use x for both the value of x before *and* after the call simultaneously.

Function Calls (III)

- For our purposes here, let's pretend that there is a single variable, x, that could be changed by the call.
- Then what we want to say is

$$\mathsf{wlp}(\llbracket f() \rrbracket, Q) \equiv P_f \wedge (\forall x.\, Q_f \Longrightarrow Q)$$

- That is, "for Q to be true after the call `f()`, the precondition of `f` must be true and Q must follow from the postcondition of `f` for any possible final value of x that `f()` might produce."
- So the example from before becomes

$$wlp(\llbracket g() \rrbracket, x < 0) \equiv x < 0 \wedge (\forall x.\, x > 0 \Longrightarrow x < 0) \equiv \textsf{false}$$

- That is, the call does ***not*** produce the desired result when `f` is called with an x that satisfies the precondition, which is the correct.

Initial Values

- (This slide was not in the online lecture. It's just for those curious.)
- It is awkward to write a Q_f that expresses something like "the value of `x` at exit is one greater than it was at entrance."
- So we add a bit of notation, and write this postcondition as $x = x' + 1$. That is x' in a postcondition denotes the value x had at the beginning.
- We then need to adjust for the use of primed variables. If $x_1, \ldots, x_n$ are the variables modified in the body, then the condition for correctness of `f` becomes

$$\forall x'_1, \ldots, x'_n . (P_f \wedge x_1 = x'_1 \wedge \ldots \wedge x_n = x'_n \Longrightarrow wlp([\![\textsf{body of f}]\!], Q_f)).$$

- (Using the $\forall$ takes care of introducing new primed variables for every call.)