Lecture #41: Topics in Static Analysis: Program Verification

- Previously looked at *static analysis*, finding properties of programs that don't depend on the specific input data.
- So far, have seen:
  - Static type checking (are these types consistent?)
  - Type inference (what must the type of this be?)
  - Analyses for optimization (what assignment statements might last have set x's value?)
- These have all used simple and fast algorithms.
- Today's example of more ambitious analysis: *program verification* tries to determine if a program does what it is specified to do.

Specifications

- Starting in the 1960's, researchers started asking what it meant to "prove" a program.
- First need a statement of what a program does.
- Obvious approach: notate a program with *assertions:*

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

  where $\mathcal{P}$ and $\mathcal{Q}$ are logical assertions and S is some program text.
- $\mathcal{P}$ is a *precondition*, and $\mathcal{Q}$ is a *postcondition*.
- Above means

  If $\mathcal{P}$ is true, S is executed, and S terminates normally, then $\mathcal{Q}$ will be true.
- Simple Example:

  $\{\ k > 0 \wedge x \leq y\ \}$ `x = x-k` $\{\ x < y\ \}$

Specifying a Language

- To prove "program assertions" like this, must first come up with axioms for the dynamic semantics of the language.
- One (older, but moderately intuitive) style due to C.A.R. Hoare.
- Start with something easy: For any predicate P,

  $\{P\}$ pass $\{P\}$

More Obvious Stuff

- Logically entailed assertions may replace other assertions.

$$\frac{P \Rightarrow R,\ \ \{R\}\mathtt{S}\{Q\}}{\{P\}\mathtt{S}\{Q\}}$$

- The line means "to prove the thing below, show the things on top."

Sequences

- To concatenate two statements:

$$\frac{\{P\}\texttt{S1}\{R\}, \{R\}\texttt{S2}\{Q\}}{\{P\}\texttt{S1; S2}\ \{Q\}}$$

If Statements

- Problem: want to demonstrate that

```
{  P  }
if C:
    S1
else:
    S2
{  Q  }
```

- Assume that conditional expression `C` has no side effects.
- So break into two cases:

$$\{\ P \wedge C\ \}\ \texttt{S1}\ \{\ Q\ \}$$
$$\{\ P \wedge \textbf{not}\ C\ \}\ \texttt{S2}\ \{\ Q\ \}$$

  and prove both.
- What would case without **else** look like? Change S2 rule to $P \wedge \textbf{not}\ C \Rightarrow Q$.
- What would case with **elifs** look like? Add rules such as $\{P \wedge \textbf{not}\ C1 \wedge \textbf{not}\ C2 \cdots \wedge Cn\}$ Sn $\{Q\}$

Assignment Statements

- A bit tricky. We'll consider scalar variables only.
- First, some terminology. If P is a logical assertion, define $P[x \rightarrow E]$ mean "P with all free instances of x replaced by E.
- For example $(x > y)[x \leftarrow 3]$ is $3 > y$.
- Now we can write a "*backward rule*" for assignment:

$$\frac{\{P \Rightarrow Q[x \leftarrow E]\}}{\{P\}\ \texttt{x = E}\ \{Q\}}$$

- Example: to show

$$\{\ i > 0 \wedge x^n = yx^i\ \}\ \texttt{y = y*x}\ \{\ i > 0 \wedge x^n = yx^{i-1}\ \}$$

Pitfalls

- Consider our first example:

$$\{\ k > 0 \wedge x \leq y\ \}\ \texttt{x = x-k}\ \{\ x < y\ \}$$

  Problem: It's not valid! Suppose that (mathematically) $x - k < -2^{31}$.
- The construct `x-k` is a "false friend"—it only *looks* like the mathematical expression $x - k$, but means something slightly different.
- Could sprinkle all our specifications with additional clauses checking for this, but things quickly become unwieldy.
- So generally, we punt in some way:
  - say that we have proved "modulo overflow"
  - or prove the property separately.
- Another example: a postcondition on a sorting routine that says simply "the array A is sorted." So is this OK?

```
def sort (A): for i in range (0, len (A)): A[i] = i
```

  In general, how do we know our specification is sufficient?