Lecture 10: Formal Verification

Formal Methods

Basics of Logic

- first order predicate logic

Program proofs:

- input/output assertions
- intermediate assertions
- proof rules

Practical formal methods

Motivation

Here is a specification

```
void merge(int a[ ], a_len, b[ ], b_len, *c)
/*requires a and b are sorted arrays of integers of length a_len and b_len
  respectively; c is an array that is at least as long as a_len+b_len.
  effects: c is a sorted array containing all the elements of a and b.  */
```

…and here is a program

```
int i = 0, j = 0, k = 0;
while (k < a_len+b_len) {
  if (a[i] < b[j]) {
    c[k] = a[i];
    i++; }
  else {
    c[k] = b[j];
    j++; };
  k++;
}
```

does the program meet the specification?

Notes on Logic

We will need a suitable logic

First Order Propositional Logic provides:

a set of *primitives* for building expressions:

variables, numeric constants, brackets

a set of logical *connectives*:

and (∧), or (∨), not (¬), implies (⇒), logical equality (≡)

the *quantifiers*:

∀ - "for all"

∃ - "there exists"

a set of *deduction rules*

Expressions in FOPL

expressions can be *true* or *false*

$(x>y \land y>z) \Rightarrow x>z$

$x=y \equiv y=x$

$\forall x,y,z\ ((x>y \land y>z) \Rightarrow x>z)$

➢ $x+1 < x-1$

➢ $\forall x\ (\exists y\ (y=x+z))$

➢ $x>3 \lor x<-6$

Free vs. bound variables

More notes on Logic

- a variable that is not quantified is *free*
- a variable that is quantified is *bound*
  - E.g. □x (□y (y=x+z))
  - x and y are bound
  - z is free

Closed formulae

- if all the variables in a formula are bound, the formula is *closed*
- a closed formula is either true or false
- the truth of a formula that is not closed cannot be determined
  - (it depends on the environment)
- we can close any formula by quantifying all free variables with □
  - if a formula is true for all values of its free variables then its closure is true.

Input/Output Assertions

Pre-conditions and Post-conditions

we could formalize:

- a requires clause as a pre-condition
- an effects clause as a post-condition

e.g. for a program with inputs i_1, i_2, i_3 and return value r, we could specify the program by:

$\{ Pre(i_1, i_2, i_3) \}$
Program
$\{ Post(r, i_1, i_2, i_3) \}$

where $Pre(i_1, i_2, i_3)$ is a logic statement that refers to i_1, i_2, i_3

The specification then says:

"if $Pre(i_1, i_2, i_3)$ is true before executing the program then $Post(r, i_1, i_2, i_3)$ should be true after it terminates"

E.g.

$\{ true \}$
Program
$\{ (r=i_1 / r=i_2) \square r >= i_1 \square r >= i_2 \}$

Strength of Preconditions

Strong preconditions

- a precondition limits the range of inputs for which the program must work
- a *strong* precondition places fewer constraints
  - the *strongest* possible precondition is {true} (same as an empty "requires" clause)
  - it is harder for a program to meet a spec that has a stronger precondition
- a *weak* precondition places more constraints
  - the weakest possible precondition is {false}
  - …which means that there are no conditions under which the program has to work
  - every program meets this spec!!!
- precondition A is stronger than B if: B implies A
  - read implies as "is not as true as" or "is true in fewer cases than"

```
{ □z (a=z*b and z>0) }
x := divide(a, b);
{ x*b=a }
```

```
{ a>=b }
x := divide(a, b);
{ □c (x*b+c=a and c>=0 and c<b) }
```

this precondition is stronger

- it doesn't require a to be a multiple of b
- (□z (a=z*b and z>0)) implies (a>=b)

Correctness Proofs

Program correctness

- if we write formal specifications we can **prove** that a program meets its specification
- "program correctness" only makes sense in relation to a specification

To prove a program is correct:

- We need to prove the post-condition is true after executing the program
  - (assuming the pre-condition was true beforehand)

E.g.

```
{ x>0 and y>0 }
z := x*y;
{ z>0 }
```

Step 1: for z>0 to be true after the assignment, x*y>0 must have been true before it

Step 2: for x*y>0 to be true before the assignment, the precondition must imply it.

Step 3: show that (x>0 and y>0) implies x*y>0 (after closure)

Weakest Pre-conditions

The general strategy is:

1) start with the post-condition
2) work backwards through the program line-by-line
3) find the weakest pre-condition (WP) that guarantees the post-condition
4) prove that the actual pre-condition implies WP
   - i.e. the actual pre-condition is weaker than the "weakest pre-condition", WP

For example

Pre	{ true }
S_1	x := 0;
S_2	y := 1
Post	{ x<y }

1) for Post to be true after S_2, then $x<1$ must be true before S_2
2) for $x<1$ to be true after S_1, then $0<1$ must be true before S_1
3) ($0<1$) is the weakest precondition for this program
4) So is (true implies $0<1$) true?

Proof rules

Proof rules

tell us how to find weakest preconditions for different programs

we need a proof rule for each programming language construct

Proof rule for *assignment*

e.g. for

```
{ Pre }
x := e;
{ Post }
```

the weakest precondition is Post with all *free occurrences* of x replaced by e

Proof rule for *sequence*

e.g. for

```
{ Pre }
S1; S2
{ Post }
```

if WP_2 is the weakest precondition for S_2, then the weakest precondition for the whole program is the same as the weakest precondition for
{ Pre } S_1 { WP_2 }

Hoare Notation

We can express proof rules more concisely

e.g. using Hoare notation:

$$\frac{\text{claim}_1,\ \text{claim}_2,\ \ldots}{\text{conclusion}}$$

this means "if claim$_1$ and claim$_2$ have both been proved, then conclusion must be true"

E.g. for sequence:

$$\frac{\{Pre\}S_1\{Q\},\ \ \{Q\}S_2\{Post\}}{\{Pre\}S_1;\ S_2\{Post\}}$$

E.g. for if statements:

$$\frac{\{Pre \text{ and } c\}S_1\{Post\},\ \ \{Pre \text{ and not}(c)\}S_2\{Post\}}{\{Pre\}\text{if } (c) \text{ then } S_1 \text{ else } S_2\{Post\}}$$

find the weakest precondition for S_1 and the weakest precondition for S_2.

Then show ((Pre and c) implies WP S_1) and ((Pre and not(c)) implies WP S_2)

Proving an IF statement

E.g.

```
{ true }
if (x>y) then
  max := x;
else
  max := y;
{ (max=x or max=y) and max>=x and max>=y) }
```

1) the first branch:

```
{ true and x>y }
max := x;
{ Post }
```

to find the weakest precondition,
substitute x for max:

WP_1 = {(x=x or x=y) and (x>=x) and (x>=y)}
= {(true or x=y) and true and (x>=y)}
= {(true) and (x>=y)}
= {x>=y}

which is okay because
(Pre and c) implies WP_1,
{true and x>y} implies {x>=y}

2) the second branch:

```
{ true and not(x>y) }
max := y;
{ Post }
```

to find the weakest precondition,
substitute y for max:

WP2 = {(y=x or y=y) and (y>=x) and (y>=y)}
= {(y=x or true) and (y>=x) and true}
= {(true) and (y>=x)}
= {y>=x}

which is okay because
(Pre and not(c)) implies WP2,
{true and not(x>y)} implies {y>=x}

Practicalities

Program proofs are not (currently) widely used:

- they can be tedious to construct
- they tend to be longer than the programs they refer to
- they could contain mistakes too!
- they require mathematical expertise
- they do not ensure against hardware errors, compiler errors, etc.
- they only prove functional correctness (i.e. not termination, efficiency,…)

Practical formal methods:

- Just use for small parts of the program
  - e.g. isolate the safety-critical parts
- Use to reason about changes to a program
  - e.g. prove that changing a statement preserves correctness
- Automate some of the proof
  - use proof checkers and theorem provers
- Use formal reasoning for other things
  - test properties of the specification to see if we got the spec right
  - ie. use for validation, rather than verification

Other approaches

Model-checking

- A model checker takes a state-machine model and a temporal logic property and tells you whether the property holds in the model
    - temporal logic adds modal operators to propositional logic:
    - e.g. □x - x is true now and always (in the future)
    - e.g. □x - x is true eventually (in the future)
- The model may be:
    - of the program itself (each statement is a 'state')
    - an abstraction of the program
    - a model of the specification
    - a model of the domain
- Model checking works by searching all the paths through the state space
    - …with lots of techniques for reducing the size of the search
- Model checking does not guarantee correctness…
    - it only tells you about the properties you ask about
    - it may not be able to search the entire state space (too big!)
- …but is (generally) more practical than proofs of correctness.

References

van Vliet, H. “Software Engineering: Principles and Practice (2nd Edition)” Wiley, 1999.

Section 15.4 gives a very brief introduction to program proofs, and includes some pointers to more readings. The rest of chapter 15 covers some other uses of formal analysis for specifications. In particular, section 15.5 is a nice summary of the arguments in favour of formal methods.

Easterbrook, S. M., Lutz, R., Covington, R., Kelly, J., Ampo, Y. & Hamilton, D. “Experiences Using Lightweight Formal Methods for Requirements Modeling”. IEEE Transactions on Software Engineering, vol 24, no 1, pp1-11, 1998

Provides an overview of experience with practical formal methods for requirements validation. Is available from my web page (http://www.cs.toronto.edu/~sme/papers/)

F. Schneider, S. M. Easterbrook, J. R. Callahan and G. J. Holzmann, "Validating Requirements for Fault Tolerant Systems using Model Checking" Third IEEE Conference on Requirements Engineering, Colorado Springs, CO, April 6-10, 1998.

Presents a case study of the use of model checking for validating requirements. Is available from my web page (http://www.cs.toronto.edu/~sme/papers/)