Concurrent Programming Lecture 2

28th August 2003

Reasoning about sequential programs

This is a very important lecture not only for this class, but for your life as a programmer. If you have not seen this material before, I guarantee that fully understanding it can make you a much better programmer. You may get away without using these concepts in sequential programming, but in concurrent programming they are vital to managing the complexity. [I'm feeling a bit like a preacher or a late-night television huckster as a write these notes!]

There are no new programming concepts here, but there is a different way of looking at programs and thinking about programs.

Logics - symbols, terms, formulas, axioms, inference rules - proof, theorems

- symbols - a,A,x,y,1,==,+,$\sum$...
- terms - x+y, f(x), ...
- formulas - x==y, $x \leq y$, P(x), $\forall x \bullet P(x)$, $P \vee Q$

Note we will be "free" with our notation and terminology; a logician would not approve! Consequently, we will need to be careful at times not to make mistakes. I may need to pull a new rule out of the hat from time to time – but this isn't a course in formal logic.

"Weakening" and "Strengthening"

Weaker predicate is true of *more* states. P is weaker than Q means $Q \Rightarrow P$

Stronger predicate is true of fewer states: P is stronger than Q means $P \Rightarrow Q$

Note that *true* is the weakest predicate ; $Q \Rightarrow true$ for any Q

false is the strongest predicate: $false \Rightarrow Q$ for any Q

Programming Logic (PL)

State

In talking about imperative programs the notion of *state* is crucial. The state of a program is a *map* or *function* from variables to values. (In more complicated situations such as in languages with pointers or references, state may include a map from ids to variables and from variables to values) Our notion of state includes the program counter in sequential programs and each of the threads' program counters in concurrent programs.

Examples of states.

Additional PL Symbols

= (assignment), while, if, else, ; (sequencing)

PL Formulas

{P} S {Q} where P and Q are formulas of first order logic and S is a programming language statement. P and Q each express some property of the program state before and after execution of S. The formulas are read: "if S is executed in a state where P is true, Q will be true in the resulting state". P and Q are thus predicates about states.

Example PL formula: {*true*} x=1 {x==1}

If x=1 is executed in a state where *true* is true (in other words any state at all) x==1 in the resulting state.

Predicates on states

– first-order logic will do for us: x==1, $x \leq y$, $\forall x.\ x<y \Rightarrow P(x,y)$, *true, false*

A *free* occurrence is one that is not *bound* by a quantifier, summation, integral, etc. $\forall x\ \exists y\ \sum_{i=1}^{n}(x, y,$ and i are bound variables, but n is free)

Free occurrences of variables in the first order formulas are interpreted according to the state. The x==1 means "x has value 1 in the current state"; $x \leq y$ means "the value of x in the current state is less than the value of y in the current state"; for all values x, less than the value of y in the current state, P(x,y) is true. Notice that x is bound, not free, in this last formula.

Problem 2.20: a[1..m], b[1..n]; What variables are free in each of the following?

All elements of a are less than all elements of b: $\forall i,j \bullet a[i] < b[j]$

Either a or b contains a single zero but not both: (exercise)

It is not the case that both a and b contain zeros: $(\forall i\ 0<i\leq n \bullet a[i]>0) \vee (\forall i\ 0<i\leq m \bullet b[i]>0)$

Assume m==n; The values in b are the same as those in a except in reverse order: $\forall i\ 0<i\leq n \bullet a[i]==b[n+1-i]$

Every element of a is an element of b: $\forall i\ 0<i\leq n \bullet \exists j\ 0<j\leq m \bullet a[i]=b[j]$

Some element of a is larger than some element of b and vice versa: $\exists i,j \bullet a[i]>b[j] \wedge \exists i,j \bullet b[i]>a[j]$

PL Assignment Axiom

$\{P_{x\leftarrow e}\}x = e\ \{P\}$

$P_{x\leftarrow e}$ means P with *e* textually substituted for all free occurrences of *x.*

Example: {y+1==1} x=y+1 {x==1}is a theorem of PL using the assignment axiom

Note that this is usually written as the equivalent

{y==0} x=y+1 {x==1}

Composition rule

if {P} S1 {Q}and {Q} S2 {R} are both theorems, then {P} S1;S2 {R} is a theorem

also show natural-deduction style as in book.

{y==0} x=y+1 {x==1}is a theorem

{x==1} y=x+1 {y==2}is a theorem

therefore {y==0} x=y+1; y=x+1 {y==2}is a theorem

IF rule

if $\{P \wedge B\}$ S {Q} and $(P \wedge \neg B \Rightarrow Q)$ are both theorems (note the former is a theorem of PL and the latter is a theorem of the underlying logic) then {P} IF B S {Q} is a theorem

IF-ELSE rule

if $\{P \wedge B\}$ S1 {Q} and $\{P \wedge \neg B\}$ S2 {Q} are both theorems then {P} IF B S1 ELSE S2 {Q} is a theorem

WHILE rule

if {I∧B} S {I} is a theorem then {I} WHILE (B) S {I∧¬B} is a theorem

In the WHILE rule, I is called the *invariant* of the loop. Invariants will play a major role in our approach to concurrent programming.

Note the similarity to mathematical induction: if I is true initially (base case) and is preserved by each iteration of the loop (induction case), it is true when the loop terminates (if it terminates)

Consequence rule

if P'⇒P, {P} S {Q}, Q⇒Q' are all theorems, then so is P' {S} Q'

Partial correctness

partially correct, partial function, total function, total correctness, totally correct.

Safety properties; Liveness properties

Safety properties are usually described as statements that something "bad" does not happen. They are usually expressed as a predicate about the state at some point in the program with the interpretation "if execution reaches this point, the predicate will hold of the state." The bad thing is reaching that point with the predicate not holding.

Liveness properties are assertions that something "good" *will* happen. An example for sequential programs is the property of termination. Termination cannot be expressed as a property of the state alone – it is a property of the state and the future execution of the program from that state.

Missing rules

(recursive) procedure/function rules, I/O rules

Completeness and Soundness

A logic is complete is every true formula is a theorem, derivable from the axioms according to the rules of the logic.

A logic is sound if every theorem, derivable from the axioms according to the rules is true.

Our PL is sound and relatively complete. That is it does not introduce any additional incompleteness beyond that which it inherits from first order logic.

From PL formulas to asserted programs and back

A proof is a sequence of formulas each derived from previously derived formulas using the rules of the logic. Writing the formulas and fragments of program text repeatedly soon becomes tedious. Asserted programs offer a less-tedious alternative presentation of program and proof. Example - finding the minimum value in an array a[1..n]:

The theorem to be proved: {true} S {∀j in [1..n] min ≤a[j]}

```
(9!){1≤n}
(8) {a[1]≤a[1] ∧ 1≤n}
(7) {∀j in [1..1] a[1]≤a[j] ∧ 1≤n}
(5) i = 1
(6) {∀j in [1..i] a[i]≤a[j] ∧ i≤n}
(5) min = a[i]
(3) {∀j in [1..i] min≤a[j] ∧ i≤n} -- Invariant
(2) while (i < n )
(10)   {∀j in [1..i] min≤a[j] ∧ i<n}
(11)   i = i+1
(12)   {∀j in [1..i-1] min≤a[j] ∧ i-1<n}
(13)   if (a[i] < min)
(15)      {∀j in [1..i] a[i]≤a[j] ∧ i-1<n}
(12)      min = a[i]
(14)   {∀j in [1..i] min≤a[j] ∧ i≤n} -- Invariant
(4) {∀j in [1..i] min≤a[j] ∧ i≤n ∧ i≥n}
(1) {∀j in [1..n] min≤a[j]}
```

One way to think about an assertion P at a particular location L in a program is: in *any* execution of this program, if the program counter is at L, P is true of the state. P.S. I cheated in this example: notice that I have not expressed a requirement that min be an element of the array and proved that the requirement is met.

Preparation for next class

Read the rest of Chapter 2 paying particular attention to sections 2.4, 2.6.3 and 2.7.

Assume that integer array `a[1..n]` has been initialized. Write a sequential program that computes the sum of the elements. Annotate the program with a proof outline, similar to that above, using the techniques of Section 2.6. The theorem to be proved is {}your-program $\{x = \sum_{i=1}^{n} a_i\}$.

Write up solutions to problems 2.5a, 2.20b, 2.23 to hand in at the beginning of class. Be prepared to describe your solution to the class.