Concurrent Programming Lecture 3

3rd September 2002

Terminology

partial function/total function; partially correct/totally correct

Atomic Actions

Fine grain atomic action

We assume that all machine instructions are executed atomically: observers (including instructions in other threads) see the state before or after the instruction, not some intermediate state and the result of the instruction depends only on the state at the start of the instruction.

Coarse grain atomic action: an atomic action comprising multiple machine instructions; atomicity is achieved by some (as yet unspecified) combination of machine instructions that ensures another thread does not execute during the time the instructions are running.

At-most-once property

A high level language assignment statement x=e satisfies the at-most-once property if either a) e contains at most one reference to a variable changed in a different thread and x is not read by another thread or b) e contains no reference to a variable changed in a different thread. An assignment (or expression) satisfying the at-most-once property will appear to be executed/evaluated atomically.

A note on good practice: relying on the at-most-once property to achieve an atomic action is generally considered bad programming practice. Why? – discuss – Fragile: changes to other parts of the program can easily cause the property to no longer hold: either reading the value of the assignment target in another location, modifying the expression to reference a second variable modified elsewhere, or changing another thread to modify an additional variable can "break" the program. When is it safe? When, in case a) x is a thread-local variable by declaration, and e references only on non-thread-local variable b) e

references no non-thread-local variables. The temptation is to use this property for "efficiency" – resist!

Await statement

Instead, when it is important for a statement or sequence of statements to be executed atomically, we will wrap it in an await statement <await B S> . In an await statement B is true when execution of S begins and no internal state of S is visible to other processes. With the await statement we can make anything into an atomic action (perhaps at high cost!) What kind of cost: reduced concurrency (if we try to make large sections of code atomic, high execution overhead due to the frequent evaluation of B, high implementation complexity to schedule the evaluation of B at appropriate times.

Eventually, therefore, we want to develop programming notations that are more robust than primitive atomic actions + at-most-once but which are less costly than await statements.

For now though we'll look at programming with await statements and see what we can do in Programming Logic to accomodate them.

Programming Logic inference rules for concurrency

Await rule

if {P∧B} S {Q} is a theorem then {P} <await B S> {Q} is a theorem. This looks like the IF rule. What's different? First there is no hypothesis (P∧¬B)⇒Q. Our intention is that await does not execute until B becomes true. In a sequential program this wouldn't make sense, but in a concurrent program another thread can make B become true. We'll see another difference in the next rule.

Co rule

if {Pi} Si {Qi} are theorems for i in [1..n], and the proofs of those theorems are interference free, then {P1∧P2...Pn} co S1 // S2 //...// Sn oc {Q1∧Q2∧...Qn} is a theorem.

Non-interference

When we write proofs and programs in *proof outline form* as we did last time notice, that each statement, s, sits between its *precondition,* pre(s), and its *post-condition,* post(s), which are assertions. An assertion is a *critical assertion* if it is not within an await statement. An *assignment action* is either an assignment statement not within an await statement or an await statement containing one or more assignment actions.

Local non-interference: an assignment action, a, does not interfere with an assertion, C, if {pre(a)∧C}a {C} is a theorem.

Global non-interference: a set of proofs is interference free if for each critical assertion, C, in the set of proofs, and each assignment action, a, appearing in a process other than the one containing C, a does not interfere with C.

Now note the second difference between IF and AWAIT: assertions within an await are not critical assertions, and assignments within an await are assumed to execute atomically for the purpose of proving non-interference with other threads.

Summary

You may be wondering what we have accomplished by introducing Programming Logic for concurrent programs. We have established a framework for arguing that our programs behave correctly that is more systematic and more concise than where we started. When we started we could could only argue about all the possible executions of our concurrent programs: if this happens first and then that happens we conclude ..., but if that happens first and then this happens we conclude ... and remember that there were exponentially states to consider.

But these proofs are still big: we still have quadratically many proof obligations in order to establish non-interference between the proofs of the various threads. (For m processes with n critical assertions and p assignment actions each, about m^2np non-interference proofs).

Writing interference-free proofs

The primary requirement is that we construct the theorems {Pi} Si {Qi}so that their proofs are interference-free. What can we do to make this process as painless as possible?

The first obvious approach: use *disjoint variable sets.* If there are no shared variables, there is no interference. This is a property of the programs alone (doesn't involve the assertions) that is easily implemented and checked. Unfortunately, it is not sufficient because the resulting programs have no interaction between the threads :-). But it is a starting point for thinking about concurrent programs whose threads only interact using explicit communications (send/recv e.g.) - more later in the semester on this approach. Also, using shared variables only when necessary reduces the number of non-trivial non-interference proofs.

Weakened assertions - remember that interference is a property of proofs, not of programs alone. We can avoid interference by changing (weakening) assertions so that potentially-interfering assignments don't interfere.

Example: Problem 2.14

At-most-once? – discuss – Clearly the program given does not satisfy the at most once property because of the 3rd co branch.

What are the final values of x and y? – discuss –

Proposed proof outline:

```
{true}
x=1
y=1
{(x==1)∧(y==1)}
co
(C3) {(x==1∨x==0)∧(y==1∨y==0)}  weakened assertion
(A1) <x = x+y>
(C4) {(x==1∨x==0∨x==2)∧(y==1∨0)}
//
(C0) {true}
(A0) y = 0
(C1) {y==0}
//
(C5) {(x==1∨x==2)∧(y==1∨y==0)}  weakened assertion
(A2) x = x-y
(C6) {(x==1∨x==0∨x==2)∧(y==1∨0)}
oc
```

If we can show non-interference in this proof, then we can conclude at the end that x is in [0..2] and y==0.

First, all non-interference theorems involving C0, C1, and A0 are trivial. That leaves the following theorems to be checked:

```
{C3∧C5} A2 {C3} (NI1)
{C4∧C5} A2 {C4} (NI2)
{C3∧C5} A1 {C5} (NI3)
{C3∧C6} A1 {C6} (NI4)

NI1:
{x==1∧(y==1∨y==0)} x=x-y {(x==1∨x==0)∧(y==1∨y==0)}
NI2:
{(x==1∨x==2)∧(y==1∨y==0)}
x=x-y
{(x==1∨x==0∨x==2)∧(y==1∨0)}

NI3:
{x==1∧(y==1∨y==0)} x=x+y {(x==1∨x==2)∧(y==1∨y==0)}

NI4:
```

```
{(x==1∨x==0)∧(y==1∨y==0)}
x=x+y
{(x==1∨x==0∨x==2)∧(y==1∨0)}
```

What effect does making A1 atomic have? – discuss – none that I can discern.

Global invariants - combine with disjoint variable sets to get a powerful technique: identify sets of global variables with each set being related by a global invariant. Confine updates to these variables to procedures associated with each particular set of variables. (This begins to look like OO programming, doesn't it?) Now interference can only occur between proofs of the procedures belonging to a particular object.

Synchronization - combining multiple assignments under one await statement reduces the number of critical assertions and the number of assignment actions.

Reading and problems for next time

Section 2.8 and Chapter 2 Historical notes. Chapter 3 Intro and section 3.1

Problem 2.33

Solution to assigned problem handed in last time:

```
(1)  {true}
(6)  i = 0
(7)  {i==0}
(8)  x = 0
(9)  {i==0 ∧ x==0)
(4)  {x=∑_{j=1}^{i} a[j] ∧ i≤n} -- Invariant
(3)  while (i<n)
(10)     {x=∑_{j=1}^{i} a[j] ∧ i<n}
(12)     i = i+1
(13)     {x=∑_{j=1}^{i-1} a[j] ∧ i-1<n}
(14)     {(x+a[i])=∑_{j=1}^{i} a[j] ∧ i≤n}
(12)     x = x + a[i]
(11)     {x=∑_{j=1}^{i} a[j] ∧ i≤n} -- Invariant
(5)  {x=∑_{j=1}^{i} a[j] ∧ i≤n ∧ i≥n}
(2)  {x=∑_{i=1}^{n} a[i]}
```

Lines are numbered in the order that I wrote them. In any problem like this, the lines that are most important to put into your answer are

- 1 and 2 together with the program text, which make up the overall theorem {P} S {Q}

- 4, 10, 11, and 5 which show the application of the WHILE rule
- 13 or 14 which are equivalent; whichever one you choose it is involved in the two assignments of the loop body and therefore key to showing that the invariant is established at 15

Why did I write the proof in this particular order?

- 1 and 2 come from the theorem to be proved
- 3 and 4 capture my decision about what test to use in the loop and what invariant I will use
- once I have 3 and 4, 5 follows from the WHILE rule. Note that 2 follows from 5 using the rules of arithmetic and logic
- 6, 7, 8, and 9 establish the invariant prior to the while loop
- 10 and 11 are required by the WHILE rule
- the two assignment statements at lines 12 are needed to re-establish the invariant
- lines 13 and 14 show that the assignments do in fact re-establish the invariant using the assignment axiom.
- I might have chosen to work on the loop body before the initialization. There is nothing sacred about the order of doing things, but I find it easier to let the order be dictated by the rules, making as few choices as possible. After 5, another good approach would be to write 10 and 11 immediately before proceeding to either the initialization or the body.

Comments on the submitted solutions:

For the most part they were valiant efforts but nobody nailed it. Things to watch out for:

- check your assertions to make sure they say what you want them to say. In particular, an assertion that is obviously equivalent to *false* is almost certainly incorrect. It can be interpreted as meaning "execution of this program must never reach this point". Likewise an assertion that is obviously equivalent to *true* is almost certainly incorrect except as the first assertion of the program. It can be interpreted as "up to this point the program hasn't done anything interesting".
- If you use program variable names as bound variables in quantifiers or summations be *very careful*. Remember that in the scope of the quantifier the variable is not free and therefore does not refer to the execution state.

- Programming symbols (=, +=, ++) appear only in the program statements. They are *never* used in the first order assertions.
- If you are going to use programming symbols that are not covered by the existing rules, for example, +=, ++, FOR, you need to explain how the existing rules can be applied to the new features.
- Note that the assignment was to write a *sequential* program. Those who wrote concurrent programs as required in the book need to read the assignment more carefully. If you find the assignment confusing or ambiguous as to what is required, please send e-mail to ask for clarification.

Solution to 2.20b

I think this is easiest if you first define an abbreviation:

onezero(ar,maxindex) $\equiv \exists i(1 \leq i \leq \text{maxindex}) \bullet (\text{ar}[i] == 0 \wedge \forall j\ (1 \leq j \leq \text{maxindex}) \bullet j \neq i \Rightarrow \text{ar}[j] \neq 0))$

The required answer is then

(onezero(a,m)$\vee$onezero(b,n)) $\wedge$ $\neg$(onezero(a,m)$\wedge$onezero(b,n))

Even more compactly we could define an abbrevation for xor

$P \oplus Q \equiv (P \vee Q) \wedge (\neg P \vee \neg Q)$

and write onezero(a,m)$\oplus$ onezero(b,n)