Convergence; Finding Invariants; Array Assignments; Disjoint Parallelism

CS 536: Science of Programming, Fall 2019

Due Tue Nov 26, 11:59 pm, Solution to be posted Sat Nov 30

11/19

A. Instructions

- Problem 6 involves drawing a graph, so it may be easier for you to write that answer out on a separate sheet from your typed-in answers to the other problems. If you have multiple files, please scan your graph and submit a zip file containing both/all of your files.

B. Why?

- To avoid runtime errors, we use domain predicates; to avoid infinite loops, we use bound functions.
- There is no algorithm for finding loop invariants, but there are some heuristics.
- We can handle array assignments if we extend our notion of substitution.
- Adding parallelism makes program execution more interesting and complicated.

C. Outcomes

After this homework, you should be able to

- State the conditions for proving loop convergence; use heuristics for finding bound functions for simple loops.
- Generate possible invariants using heuristics.
- Calculate the *wp* of an array element assignment using substitution for an array element.
- Recognize parallel programs, disjoint parallel programs, and parallel programs with disjoint conditions.

D. Problems [60 points total]

Lecture 18: Loop Convergence [16 points]

1. [6 = 3 * 2 points] Consider the loop

   {**inv** *p* } {**bd** *t* } **while** `j` ≤ `n` **do** … `j := j+1` **od**.

   Assume the invariant *p* implies `n` ≥ `0` ∧ `0` < `C` ≤ `j` ≤ `n+C`) (where `C` is a named constant). For each of the following expressions, say whether or not it can be used as the bound expression *t* above (if not, briefly explain why).

   a. `n – j`
   b. `n + j + C`
   c. `n – j + 2*C`

2. [10 points] Expand the minimal outline below into a full proof outline for total correctness. (I.e., avoid runtime errors and loop divergence.)

   - You'll need to define a bound expression. (Hint: both `k` and `x` are increasing, so look for an find upper bound for one/both of them.

- Not all of the conditions marked ??? need to be filled in; it depends on whether you use *wp* or *sp* in that part of the outline.
- If you want, feel free to define other predicates ("Let *name* ≡ *predicate it stands for*.")
- Include a list of predicate logic obligations and show the expansion of any substitutions.

```
{ b[j] ≥ 1      }        // (You may or may not need something to ensure bound expr ≥ 0.)
                         // (Use wp or sp on initialization code; if you use wp, it goes here.)
x := 1;
{???}                    // (Whether you use wp or sp on the init. code, you need a condition here.)
k := 0;
                         // (If you use sp on initialization code, it goes here.)
{inv p ≡ x = 2^k ≤ b[j] ∧ ??? }     // Augment p to include bound expression ≥ 0
{bd ???}                 // (Bound expression)
while 2*x ≤ b[j] do
    {p ∧ 2*x ≤ b[j] ∧ ???}       // (Loop inv, test, and bound expression = logical variable.)
                         // (Use wp or sp on loop body; if you use wp, it goes here.)
    k := k+1;
    {???}                // (Whether you use wp or sp on the loop body, you need a condition here.)
    x := 2*x
                         // (If you use sp on loop body, it goes here.)
    {p ∧ ???}            // (Loop invariant and bound expr < logical variable.)
od
{ p ∧ 2*x > b[j] }
{ x = 2^k ≤ b[j] < 2^(k+1) }
```

Lecture 19: Finding Invariants [14 points]

For Problems 3 and 4, let $q \equiv x \geq 0 \wedge z = 2^x \leq n < 2^{(x+1)}$ where `n` is constant and `x` is a variable. Write each invariant candidate in the form

{*initial precondition*} *initialization code* **;** {**inv** *invariant*} **while** *loop test*.

You can include an educated guess about the range of the new variable, but it's not required. If you can't find an initial precondition or initialization code for some case, explain why. Feel free to write `2^x` as 2^x if you prefer.

3. [8 = 4 * 2 points] There are six possible invariants obtained by replacing a constant by a variable in *q*. Two of them involve replacing the `2` in `2^x` and `2^(x+1)` and don't lead to reasonable invariants.
   (a), (b), (c), (d) Give the other four candidate invariants obtained by replacing a constant by a variable in *q*.

4. [6 = 2 * 3 points] If we take $z = 2^x \leq n < 2^{(x+1)}$ as meaning $z = 2^x \wedge 2^x \leq n \wedge n < 2^{(x+1)}$, then there are three possible invariants obtained by dropping a conjunct of *q*. One of those conjuncts is $z = 2^x$, which definitely doesn't work, since the loop would use **while** $z \neq 2^x$.
   (a), (b), Give the other two candidate invariants obtained by dropping a conjunct of *q*.

Lecture 20: Array Assignments [14 points]

5. [14 = 2*3 + 2*4 points] Complete each of the full proof outlines for partial correctness by using *wp* to give definitions for *q* and then *p*. Logically simplify as you go.
   a. { *p* } `b[j] := a;` { *q* } `b[i] := c` { `b[j]` ≤ `b[i]` }
   b. { *p* } `b[j] := b[m];` { *q* } `b[k] := b[n]` { `b[j]` < `b[k]` ∧ `j` ≠ `k` }
      (Hint: `j` ≠ `k` will help you simplify as you go.)

Lecture 21: Parallel Programs [16 points]

6. [7 points] Let $S \equiv$ [`x := x+5; y := x/2` ‖ `z := x/3`] and σ = {`x` = 12}. (Note that in the first thread the assignments are done sequentially.) (a) Draw an evaluation graph for ⟨*S*, σ⟩ and (b) Give *M*(*S*, σ).

7. [9 = 3 * 3 points] Write out a table in the style of Activity 12 to show whether the three triples below have pairwise disjoint programs and conditions. For each pair of triples, list the *Change*, *Var*, and *Free* sets and whether the pair are parallel disjoint and/or have disjoint conditions.
   - {`x` ≠ `y`} `x := u ; y := u` {`x` = `y`}
   - {`v` = `z`} `z := z+1 ; v := v+1` {`v` = `z`}, and
   - {`w` ≥ `u+x`} `w := u+1; w := v` {`w` > `u+x`}

