Loop Convergence & Total Correctness

CS 536: Science of Programming, Spring 2022

A. Why

- If we want a program to produce an answer, it should terminate, so it's useful to know how to show that loops terminate.

B. Objectives

At the end of this class you should understand

- The loop bound method of ensuring termination.
- How to extend proofs of partial correctness to total correctness.

C. Partial correctness to total correctness

- We've seen how we can use the domain predicate *D(s)* to convert partial correctness into total correctness for loop-free programs (total correctness = partial + *D(s)*): the domain predicate gives the condition necessary for the program to not have runtime errors.
- We can express this with inference rules:

$$\frac{[e/x]p \Rightarrow D(e)}{\vdash [[e/x]p]x := e[p]} Assign(Backwards)$$

and so on…

- But what about while?

D. Loop Divergence

- Aside from runtime errors, the other way that programs don't terminate correctly is that they *diverge* (run forever). For our programs, that means infinite loops.
  - (For programs with recursion, we also have to worry about infinite recursion, but the discussion here is adaptable, especially if you remember that a loop is simply an optimized tail-recursive function.)
- For some loops, we can ensure termination by calculating the number of iterations left. E.g., at each loop test, *k := 0; while* ($k < n$) {…; $k := k + 1$} has $n - k$ iterations left.
  - This is fairly useful, e.g., the vast majority of *for* loops you see have a form that lets you do this.
  - But in general, we can't calculate the number of iterations for all loops (see theory of computation course for uncomputable functions).

- But we don't need the exact number of iterations — *it's sufficient to find a decreasing upper bound expression t* for the number of iterations. This t is a logical expression (we're not planning to actually evaluate it at runtime). It can contain program variables and ghost variables, and is often called the *bound function*.
- *Syntax*: We'll attach the upper bound expression t to a loop using the syntax
  $\{dec\ t\}$
- To show convergence of the loop $\{inv\ p\}\ \{dec\ t\}\ while\ e\ \{s\}\{p \wedge \neg e\}$, it's sufficient for the bound expression t to meet the two following properties:
  - $p \rightarrow t \geq 0$
    - The invariant guarantees that there is a nonnegative number of iterations left to do.
  - $\{p \wedge e \wedge t = t_0\}\ S\ \{p \wedge t < t_0\}$ where t_0 is a fresh logical variable.
    - If you compare the value of the bound expression at the beginning and end of the loop body, you find that the value has decreased. i.e., if you were to print out the value t at each while test, you would find a strictly decreasing sequence of nonnegative integers.
  - The variable t_0 is a logical variable (we don't actually calculate it at runtime). We're using it in the correctness proof to name of the value of t before running the loop body. It should be a fresh variable (one we're not already using) to avoid clashing with existing variables.
  - (Note: To get full total correctness, we also have to avoid runtime errors, which we saw in an earlier class.)
- *Example 1*: For the $sum(0, n)$ program, we can use $n - k$ for the bound:

$$\{n \geq 0\}$$

$k := 0;\ s := 0;$
$\{inv\ p \equiv 0 \leq k \leq n \wedge s = sum(0, k)\}$
$\{dec\ n - k\}$
$while\ k < n\ \{k := k + 1;\ s := s + k\}$

$$\{s = sum(0, n)\}$$

- At the loop test, we always have ≥ 0 iterations left: p implies $0 \leq k \leq n$, which implies $n - k \geq 0$.
- Execution of the loop body lowers the bound. Let t_0 be our fresh logical variable, then we need $\{p \wedge k < n \wedge n - k = t_0\}\ loop\ body\ \{n - k < t_0\}$. Since the loop body includes $k := k+1$, we know this is true: $\{n - k = t_0\}\ \{n - (k+1) < t_0\}\ k := k+1\ \{n - k < t_0\}$ by the assignment's wp, with precondition strengthening.

Hidden Requirements for a Bound Expression

- The two properties we need a bound expression to have (being nonnegative and decreasing with each iteration) imply that bound expressions have some hidden requirements to meet.
  - *The bound expression can't be a constant*, since constants don't change values.
    - *Example 2*: For the loop $k := 0;$ *while* $k < n$ $\{\ldots ; k := k + 1\}$, people often make an initial guess of "n" for the bound expression instead of $n - k$. When $k = 0$, the upper bound is indeed $n - k = n$, but as k increases, the number of iterations left decreases.
  - *A nonnegative bound can't always imply the loop condition:* If e is the while loop test, then $t \geq 0 \rightarrow e$ would cause divergence: Since $p \rightarrow t \geq 0$, if $t \geq 0 \rightarrow e$, then $p \rightarrow e$, so e would be true at every loop test.
  - $p \wedge e \rightarrow t > 0$ *is required*: When p and B hold, we run the loop body, which should decrease t but leave it ≥ 0. Equivalently, $p \wedge t = 0 \rightarrow \neg e$ because there's no room for the loop body to decrease t, therefore we'd better not be able to do that iteration.

Bound Function Properties That are Allowed but Not Required

- There are a number of properties that the bound function is allowed to have but aren't required.
- Not required: $p \wedge \neg e \rightarrow t = 0$. Since t doesn't have to be a strict upper bound, it doesn't have to be zero on termination (though in many cases it will be).
- Required: Number of iterations remaining $\in O(t)$. This holds because $t \geq$ *number of iterations.*
- Not required: number of iterations remaining $\in \Theta(t)$. This is because t doesn't have to be a strict upper bound.
- Required: $\{p \wedge e \wedge t = t_0\}$ *loop body* $\{t < t_0\}$. Execution of the loop body must reduce t.
- Not required: $\{p \wedge e \wedge t = t_0\}$ *loop body* $\{t - t_0 = 1\}$. We aren't required to reduce t by exactly 1. For example, with searches, t = size of search space generally works.
  - *Example 3*: For binary search, if L and R are the left and right endpoints of the search and $p \rightarrow L < R$, then $R - L$ is a perfectly fine upper bound even though $ceiling(\log_2(R - L))$ is tighter.

E. *Heuristics For Finding A Bound Expression*

- *To find a bound expression* t, there's no algorithm but there are some guidelines.
  - First, start with $t \equiv 0$.
  - For each (? some?) variable x that the loop body decreases, add x to t.
  - For each (? some?) variable y that the loop body increases, subtract y from t.

- If $t < 0$ is possible, try to find some large expression e such that $e + t \geq 0$. Basically, we need a manipulation of t that makes it more positive, and adding a large constant might be helpful.

- *Example 4*: For a loop that sets $k := k - 1$, try k (i.e., add "+ k" to $t \equiv 0$) for t.
  - If the invariant allows $k < 0$, then we need add something to t to make it larger. E.g., if the invariant implied $k \geq -10$, then it would imply $k + 10 \geq 0$, we could add 10 to our candidate bound and get $t + 10$.
- *Example 5*: For a loop that sets $k := k + 1$, try $(-k)$ (i.e., $0 - k$) for t.
  - If $-k$ can be < 0, we should add something to it. E.g., if the invariant implies $k \leq e$, then it implies $e - k \geq 0$, so maybe adding e to t will help.

F. Increasing and Decreasing Loop Variables

- We've looked at the simple summation loop

```
{n ≥ 0} k := 0; s := 0;
{inv p ≡ 0 ≤ k ≤ n ∧ s = sum(0,k)} {dec n – k}
 while k < n {
    k := k + 1;
    s := s + k
}
{s = sum(0, n)}
```

- *A first bound function*:
  - Using our heuristic, since k and s are increasing, $-k-s$ is a candidate bound function that fails because it's negative.
  - For terms to add to $-k-s$ to make it nonnegative, we know $n-k \geq 0$ because the invariant includes $k \leq n$, so let's add n and get $n - k - s$.
  - But $n - k - s$ can be negative, so we want to add some expression e such that $e + n - k - s \geq 0$. The invariant doesn't give an explicit bound for s, but from algebra we know that $0 + 1 + 2 + \ldots + n$ grows quadratically, and it's easy to verify that $n^2 - s \geq 0$ for all $n \in \mathbb{N}$.
  - This gives $n^2 + n - k - s$ as a bound function.
- *A second and third bound function*
  - Since $n - k \geq 0$ and decreases as k increases, it by itself works as a bound function.
  - Similarly, $n^2 - s \geq 0$ and decreases as s increases, so it works by itself as a bound function too.
- *Modifications to bound functions*
  - Bound functions are not unique: If t is a bound expression, then so is $a\,t^n + b$ for any positive a, b, and n. Similarly, if t_1 and t_2 are bound functions separately, then $t_1 + t_2$

and $t_1 \times t_2$ are also bound functions. So it's possible to encounter $n^2 + n - k - s$ by finding $n - k$ and $n^2 - s$ individually and then adding them.

G. Another Loop Example: Iterative GCD

- Not all loops modify only one loop variable with each iteration: Some modify multiple variables, with some being modified sometimes and others being modified another time.
- *Definition*: For $x, y \in \mathbb{N}$, $x, y > 0$, the *greatest common divisor* of x and y, written $gcd(x, y)$, is the largest value that divides both x and y evenly (i.e., without remainder).
  - E.g., $gcd(300, 180) = gcd(2^2 * 3 * 5^2, 2^2 * 3^2 * 5) = 2^2 * 3 * 5 = 60$.
- Some useful *gcd* properties:
  - If $x = y$, then $gcd(x, y) = x = y$
  - If $x > y$, then $gcd(x, y) = gcd(x - y, y)$
  - If $y > x$, then $gcd(x, y) = gcd(x, y - x)$
- E.g., $gcd(300, 180) = gcd(120, 180), gcd(120, 60) = gcd(60, 60) = 60$.
- Here's a program that calculates GCD using an algorithm developed by Euclid c. 300 BC:

```
{x > 0 ∧ y > 0 ∧ X = x ∧ Y = y}
{inv p ≡ x > 0 ∧ y > 0 ∧ gcd(X, Y) = gcd(x, y)}
{dec ???}  // to be filled-in
while x ≠ y {
   if x > y then { x := x - y } else { y := y - x }
}
{x = gcd(X, Y)}
```

- Here's a full proof outline for partial correctness.

```
                                                    {x > 0 ∧ y > 0 ∧ X = x ∧ Y = y}
{inv p ≡ x > 0 ∧ y > 0 ∧ gcd(X, Y) = gcd(x, y)}
{dec ???}  // to be filled-in
while x ≠ y {                                       {p ∧ x ≠ y}
   if x > y then {                                  {p ∧ x ≠ y ∧ x > y} => {[x-y/x]p}
      x := x - y                                    {p}
   } else {                                         {p ∧ x ≠ y ∧ x ≤ y} =>{p[y - x/y]}
      y := y - x                                    {p}
   }                                                {p}
}                                                   {p ∧ x = y} => {x = gcd(X, Y)}
```

- We have a number of predicate logic obligations

- $(x > 0 \wedge y > 0 \wedge x = X \wedge y = Y) \rightarrow p$
- $p \wedge x \neq y \wedge x > y \rightarrow [x - y/x]p$
- $p \wedge x \neq y \wedge x \leq y \rightarrow [y - x/y]p$
- $p \wedge x = y \rightarrow x = gcd(X, Y)$

- With $p \equiv x > 0 \wedge y > 0 \wedge gcd(X, Y) = gcd(x, y)$, the substitutions are
  - $[x - y/x]\ p \equiv x - y > 0 \wedge y > 0 \wedge gcd(X, Y) = gcd(x - y, y)$
  - $[y - x/y]p \equiv x > 0 \wedge y - x > 0 \wedge gcd(X, Y) = gcd(x, y - x)$
- (There are other full outline expansions, for example, one using the *wp* of the entire *if–fi*, which is
  - $(p \wedge x \neq y) \rightarrow ((x > y \rightarrow p[x - y/x]) \wedge (x \leq y \rightarrow p[y - x/y]))$
  - But these other outlines produce logic obligations of roughly the same proof difficulty.
- What about convergence?
  - The loop body contains code that makes both x and y smaller, so our heuristic gives us $x + y$ as a candidate bound function. Non–negativity is easy to show: the invariant implies $x, y > 0$, so $x + y \geq 0$.
  - Reduction of $x + y$ is slightly subtle: Though the loop body doesn't always reduce x or always reduce y, it always reduces one of them, so $x + y$ is always reduced.

- So our final minimally–annotated program is

```
{x > 0 ∧ y > 0 ∧ X = x ∧ Y = y}          // X and Y are the initial values of x and y
{inv p ≡ x > 0 ∧ y > 0 ∧ gcd(X, Y) = gcd(x, y)}
{dec x + y}
while x ≠ y {
   if x > y then { x := x – y } else { y := y – x}
}
{x = gcd(X, Y)}
```

- We can add the material in *green* below to fill in the full outline for total correctness.

```
                                                        [ x > 0 ∧ y > 0 ∧ X = x ∧ Y = y ]
{inv p ≡ x > 0 ∧ y > 0 ∧ gcd(X, Y) = gcd(x, y) ∧ x+y ≥ 0}
{dec x+y}
while x ≠ y {                                           [ p ∧ x ≠ y ∧ x+y = t0 ]
   if x > y then {                                      [ p ∧ x ≠ y ∧ x > y ∧ x+y = t0 ]
                                                        => [ p[x – y/x] ∧ (x–y)+y < t0 ]
      x := x – y                                        [p ∧ x+y < t0 ]
   } else {                                             [ p ∧ x ≠ y ∧ x ≤ y ∧ x+y = t0 ]
                                                        => [ p[y – x/y] ∧ x+(y–x) < t0 ]
      y := y – x                                        [ p ∧ x+y < t0 ]
```



```
    }                                   [ p ∧ x+y < t0 ]
}                                       [ p ∧ x = y] => [ x = gcd(X, Y) ]
```

- For this to work, we need $x + y = t_0$ to imply either $(x - y) + y$ or $x + (y - x) < t_0$ (depending on the *if–else* branch). These hold because $(x - y) + y = x < x + y$ and $x + (y - x) = y < x + y$ (because both x and y are positive).

H. Semantics of Convergence

- Here's a semantic assertion about bound functions and loop termination.
- *Lemma (Loop Convergence)*: Let $W \equiv$ *{inv p} {dec t} while e {s}* be an loop annotated with an invariant and bound function. Assume we can prove partial correctness of W and total correctness of [$p \wedge e \wedge t = t_0$] s [$p \wedge t < t_0$]. Then if $\sigma \models p \wedge e \wedge t = t_0$, then $\perp_d \notin M(W, \sigma)$. (Proof omitted.)

*I. *** Total Correctness of a Loop ****

While Loop Rule for Total Correctness

$$\frac{\vdash [\, p \wedge e \wedge t = t_0 \,] s [\, p \wedge t < t_0 \,] \qquad p \Rightarrow t \geq 0 \qquad p \Rightarrow D(e)}{\vdash \{inv\ p\}\{dec\ t\}\ while\ e\ \{s\}\ [\, p \wedge \neg e \,]}\ While$$