Loop Convergence & Total Correctness

CS 536: Science of Programming, Fall 2022

Tue 2022-10-25: pp. 2, 3, 6 , 7

A. Why

- Diverging programs aren't useful, 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. Loop Divergence

- Aside from runtime errors, the other way that programs don't terminate 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 test there are $n-k$ iterations left for $k := 0$; ***while*** $k < n$ ***do*** …; $k := k + 1$ ***od.***
  - 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*** for the number of iterations.
- ***Definition:*** A ***bound expression*** or ***bound function*** t for a loop is a logical expression that, at each loop test, gives a strictly decreasing upper bound on the number of iterations remaining before termination. A bound expression can use program variables and logical variables.
- ***Syntax***: We'll attach the upper bound expression t to a loop using the syntax $\{$***bd*** $t\}$, so a typical loop has the form $\{$***inv*** $p\}$ $\{$***bd*** $t\}$ ***while*** B ***do*** S ***od*** $\{p \wedge \neg B\}$.
- Note we aren't required to calculate the value of t at runtime, since it's a logical expression.

Properties of Bound Functions

- For t to be a valid bound expression, it needs to meet the two following properties:
  - $p \rightarrow t \geq 0$
    - Since the invariant has to be true at each loop test, making satisfaction of p imply $t \geq 0$ is a simple way to ensure that $t \geq 0$ at every loop test.

    - Another way to phrase this is that at each loop test, there must be a nonnegative number of iterations left to do.
  - $\{p \land B \land t = t_0\}$ S $\{p \land t < t_0\}$ 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 of t at each while test, you would find a strictly decreasing sequence of nonnegative integers. (Since t can include logical variables, printing it out might not be possible.)
  - 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:

  $\{n \ge 0\}$ k := 0; s := 0;
  {***inv*** $p \equiv 0 \le k \le n \land s = sum(0, k)\}$
  {***bd*** n − k} ***while*** k < n ***do*** k := k + 1; s := s + k ***od***
  $\{s = sum(0, n)\}$

  - Check $p \rightarrow n - k \ge 0$: At the loop test, p implies $0 \le k \le n$, which implies $n - k \ge 0$.
  - Check that execution of the loop body decreases k − n: Let t_0 be our fresh logical variable, then we need $\{p \land k < n \land n-k = t_0\}$ loop body $\{n-k < t_0\}$. Since the loop body includes k := k+1 (and no other change to k), we find that $\{n-k = t_0\}$ $\{n-(k+1) < t_0\}$ k := k+1 $\{n-k < t_0\}$ is a correct full outline. [2022-10-25] A ***progress step*** is a statement that reduces the value of the bound function. (Every loop iteration needs to execute one.)

Bound Expression Properties

- The two properties we need a bound expression to have (being nonnegative and decreasing with each iteration) imply that bound expressions have other properties but also that they don't have to have other properties.
  - ***The bound expression can't be a constant***, since constants don't change values.
    - ***Example 2***: For the loop k := 0; ***while*** k < n ***do*** … ; k := k + 1 ***od***, 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 imply that the loop test holds:*** If B is the while loop test, then $t \ge 0 \rightarrow B$ would cause divergence: Since $p \rightarrow t \ge 0$, if $t \ge 0 \rightarrow B$, then $p \rightarrow B$, so B would be true at every loop test.
  - ***$p \land B \rightarrow t > 0$ is required***: When p and B hold, we run the loop body, which should decrease t but leave it ≥ 0. Equivalently, $p \land t = 0 \rightarrow \neg B$ because there's no room for the loop body to

decrease t, therefore we'd better not be able to do that iteration. $\neg p \vee \neg B \vee t > 0$ is an equivalent phrasing.

- [2022-10-25] $p \wedge \neg B \rightarrow$ ***$t = 0$*** is not required:. Since t doesn't have to be a strict upper bound, it doesn't have to be zero on termination.
  - Not required: $p \wedge t > 0 \rightarrow B$ [2022-10-25]. This is basically the converse of the previous predicate.
- Let N be the number of iterations remaining at some loop test point. Then,
  - ***N must be in O(t)*** because $t \geq$ number of iterations.
  - ***N is not required to be in Θ(t)*** because t is not required to be a strict upper bound.

- ***$\{p \wedge B \wedge t = t_0\}$ loop body $\{t - t_0 = 1\}$*** is not required. We must decrease t by at least 1, but more than 1 is fine.
- ***Example 3***: For searches, t = the size of the search space generally works. For binary search, if $p \rightarrow left < right$ (left and right are the left and right endpoints of the search), then right − left is a perfectly fine upper bound even though $\text{ceiling}(\log_2(right-left))$ is tighter.

D. Heuristics For Finding A Bound Expression

- ***To find a bound expression*** t, there's no algorithm but there are some guidelines.
  - First, start with a candidate $t \equiv 0$.
  - For each variable / some variable x that the loop body decreases, add x to t.
  - For each variable / some variable y that the loop body increases, subtract y from t.
  - If $t < 0$ is possible, look for a manipulation that makes the resulting term nonnegative. E.g., if for some e, we have $t \leq e$, then $e - t \geq 0$, so and use e – t as our new candidate t [2022-10-25].
- ***Example 4***: Say a loop sets $k := k-1$. First try k (i.e., add "+ k" to $t \equiv 0$) for t. If the invariant allows $k < 0$, then we need something that makes t larger. E.g., if the invariant implies $k \geq -10$, then it implies $k + 10 \geq 0$, so our new candidate bound function is $k + 10$.
- ***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 do something to increase it. E.g., if the invariant implies $k \leq e$, then it implies $e - k \geq 0$, so adding e to t should help.

E. 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)} {bd n − k}
 while k < n do
     k := k + 1;
     s := s + k
od
{s = sum(0, n)}
```

- ***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.
- ***Second and third bound functions***
  - Since $n - k \geq 0$ and decreases as k increases, $n - k$ by itself is as a bound function.
  - Similarly, $n^2 - s \geq 0$ and decreases as s increases, so $n^2 - s$ by itself is 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 is also a bound function. So it's possible to encounter $n^2+n-k-s$ by finding $n-k$ and n^2-s individually and then adding them.

F. Iterative GCD Example

- 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 $\text{gcd}(x, y)$, is the largest value that divides both x and y evenly (i.e., without remainder).
  - E.g., $\text{gcd}(300, 180) = \text{gcd}(2^2 * 3 * 5^2, 2^2 * 3^2 * 5) = 2^2 * 3 * 5 = 60$.
- Some useful gcd properties:
  - If $x = y$, then $\text{gcd}(x, y) = x = y$
  - If $x > y$, then $\text{gcd}(x, y) = \text{gcd}(x-y, y)$
  - If $y > x$, then $\text{gcd}(x, y) = \text{gcd}(x, y-x)$

- E.g., gcd(300, 180) = gcd(120, 180), gcd(120, 60) = gcd(60, 60) = 60.
- Here's a minimal proof outline for an iterative gcd–calculating loop:

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

- We have a number of predicate logic obligations
  - (x > 0 ∧ y > 0 ∧ x = X ∧ y = Y) → p
  - p ∧ x ≠ y ∧ x > y → p[x−y/x]
  - p ∧ x ≠ y ∧ x ≤ y → p[y−x/y]
  - p ∧ x = y → x = gcd(X, Y)
- With p ≡ x > 0 ∧ y > 0 ∧ gcd(X, Y) = gcd(x, y), the substitutions are
  - p[x−y/x] ≡ x−y > 0 ∧ y > 0 ∧ gcd(X, Y) = gcd(x−y, y)
  - p[y−x/y] ≡ x > 0 ∧ y−x > 0 ∧ 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 ∧ x ≠ y) → ((x > y → p[x−y/x]) ∧ (x ≤ y → 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 ≥ 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

```
{$x > 0 \wedge y > 0 \wedge X = x \wedge Y = y$}          // X and Y are the initial values of x and y
{**inv** $p \equiv x > 0 \wedge y > 0 \wedge gcd(X, Y) = gcd(x, y)$}
{**bd** $x + y$}
**while** $x \neq y$ **do**
  **if** $x > y$ **then** $x := x-y$ **else** $y := y-x$ **fi**
**od**
{$x = gcd(X, Y)$}
```

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

```
{$x > 0 \wedge y > 0 \wedge X = x \wedge Y = y$}
{**inv** $p \equiv x > 0 \wedge y > 0 \wedge gcd(X, Y) = gcd(x, y) \wedge x+y \geq 0$}
{**bd** $x+y$}
**while** $x \neq y$ **do**
  {$p \wedge x \neq y \wedge x+y = t_0$}
  **if** $x > y$ **then**
    {$p \wedge x \neq y \wedge x > y \wedge x+y = t_0$} {$p[x-y/x] \wedge (x-y)+y < t_0$} $x := x-y$ {$p \wedge x+y < t_0$}
  **else**
    {$p \wedge x \neq y \wedge x \leq y \wedge x+y = t_0$} {$p[y-x/y] \wedge x+(y-x) < t_0$} $y := y-x$ {$p \wedge x+y < t_0$}
  **fi** {$p \wedge x+y < t_0$}
**od** {$p \wedge x = y$} {$x = gcd(X, Y)$}
```

- 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).

G. Semantics of Convergence

- Here's a semantic assertion about bound functions and loop termination.
- ***Lemma (Loop Convergence)***: Let $W \equiv$ {***inv*** p} {***bnd*** t} ***while*** B ***do*** S ***od*** be a loop annotated with an invariant and bound function. Assume we can prove partial correctness of W and total correctness of $\{p \wedge B \wedge t = t_0\}\ S\ \{p \wedge t < t_0\}$. Then if $\sigma \vDash p \wedge B \wedge t = t_0$, then $\bot_d \notin M(W, \sigma)$. (Proof omitted. [2022-10-25] It's not that hard - it's by induction on t.)

*H. *** Total Correctness of a Loop ****

- To show total correctness of $\{p_0\}\ S_0$; {***inv*** p}{***bd*** t} ***while*** B ***do*** S ***od*** $\{q\}$ we need
  - Partial correctness: $\vDash \{p_0\}\ S_0;\ W\ \{q\}$, where W is the loop proper.
    1. Initialization establishes the invariant: $\{p_0\}\ S_0\ \{p\}$. Typical proofs are $\{p_0\}\ \{wp(S_0, p)\}\ S_0\ \{p\}$ or $\{p_0\}\ S_0\ \{sp(p_0, S_0)\}\ \{p\}$.
    2. The loop body maintains the invariant: $\{p \wedge B\}\ S\ \{p\}$.

3. The loop establishes the final postcondition: $p \wedge \neg B \rightarrow q$

- Termination: $\vDash_{tot}$ $\{p_0\}$ S_0; W $\{T\}$
  4. No runtime errors during initialization $(p_0 \rightarrow D(S_0))$ or during loop evaluation: $(p \rightarrow D(B))$ and $(p \wedge B \rightarrow D(S)))$.
  5. No divergence: The bound function is nonnegative $(p \rightarrow t \geq 0)$ and evaluation of the loop body decreases the bound: $\{p \wedge B \wedge t = t_0\}$ S $\{t < t_0\}$.
- For a formal proof rule, let's concentrate on the loop itself and not worry about initialization or finalization. This leaves partial correctness (line 2 above) and termination (lines 4 and 5 above).

While Loop Rule for Total Correctness

1. $p \rightarrow D(B) \wedge (B \rightarrow D(S))$
2. $p \rightarrow t \geq 0$
3. [2022-10-25] $\vdash_{tot}$ $\{p \wedge B \wedge t = t_0\}$ S $\{p \wedge t < t_0)$
4. $\vdash_{tot}$ $\{$***inv*** $p\}$ $\{$***bnd*** $t\}$ ***while*** B ***do*** S ***od*** $\{p \wedge \neg B\}$ ***while*** 1, 2, 3