Loop Convergence

CS 536: Science of Programming, Fall 2019

10/27

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 lecture you should understand

- The loop bound method of ensuring termination.

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 loop test, `i := 0; while i < n do ...; i := i + 1 od` has `n-i` iterations left.
  - 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 variables from the program or proof, which is why t is also often called the **bound function**.
- **Syntax**: We'll attach the upper bound expression t to a loop using the syntax {**bd** t}.
- To show convergence of the loop {**inv** p} {**bd** t} **while** B **do** S **od** $\{p \wedge \neg B\}$, it's sufficient for the bound expression t to meets the two following properties:
  - $p \rightarrow t \geq 0$
    - The invariant guarantees that there is a nonnegative number of iterations left to do.
  - $\{p \wedge B \wedge t = \mathtt{t_0}\}\ S\ \{p \wedge t < \mathtt{t_0}\}$ where $\mathtt{t_0}$ 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 $\mathtt{t_0}$ 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 lecture.)

- **Example 1**: For the `sum(0, n)` program, we can use `n-i` for the bound:

  {`n ≥ 0`} `i := 0; s := 0;`
  {**`inv`** $p \equiv$ `0 ≤ i ≤ n ∧ s = sum(0, i)`}
  {**`bd`** `n-i`} **`while`** `i < n` **`do`** `i := i + 1; s := s + i` **`od`**
  {`s = sum(0, n)`}

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

- **Two 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 cannot have the two following properties:
  - *The bound expression can't be a constant*, since constants don't change values.
    - **Example 2**: For the loop `i := 0; while i < n do …; i := i + 1 od`, people often make an initial guess of "`n`" for the bound expression instead of `n-i`. When `i` = 0, the upper bound is indeed `n-i` = `n`, but as `i` 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 $t \geq 0 \rightarrow B$ causes divergence: Since $p \rightarrow t \geq 0$, if $t \geq 0 \rightarrow B$, then $p \rightarrow B$, so B is true at every loop test.
  - There's no requirement that when B is false, t must be zero. It's allowed but not required.
    - Similarly, $t > 0 \rightarrow B$ is allowed, but it's not required.

- **Three Non-Requirements for a Bound Expression**
  - It's often the case that people think the bound expression has to have certain properties that, though nice to have, are in fact not required.
  - First, we're only trying to prove termination; we're not figuring out the asymptotic running time, so the upper bound doesn't have to be tight.
    - **Not required**: We don't require t`-1` to not be an upper bound. More generally, using big-O notation, we don't need the running time to be $\in \Theta(t)$, just $\in O(t)$.
  - The bound expression doesn't have to ever become zero.
    - **Not required**: $p \wedge \neg B \rightarrow t =$ `0`.
  - The bound expression doesn't have to decrease by exactly one:
    - **Not required**: {$p \wedge B \wedge t =$ t_0} *loop body* {$t =$ t_0`-1`}

- **Example 3**: For binary search, if `L` and `R` are the left and right endpoints of the search, then `R-L` is a perfectly fine upper bound even though *ceiling*($\log_2$`(R-L)`) 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 $t \equiv$ `0`. If the loop body makes a variable `x` smaller, add `x` as a term in t.

- If the loop body makes a variable `y` larger, add `(-y)` as a term in t.
- If your current guess for t can be < 0, find a large value to add to it to ensure $t \geq 0$ [big constants are nice].

- **Example 4**: For a loop that sets `j := j-1`, try `j` for t.
  - If the invariant allows $j < 0$, then we need to make t larger. E.g., if the invariant implies $j \geq -10$, then it implies $j + 10 \geq 0$, so maybe adding `10` to t will help.
- **Example 5**: For a loop that sets `i := i + 1`, try `(-i)` for t.
  - If `(-i)` can be < 0, we should add something to `(-i)`. E.g., if the invariant implies $i \leq x$, then it implies $x - i \geq 0$, so maybe adding `x` to t will help.

E. ☞☞ The Five Things You Need To Prove About a Loop

- To show that a loop of the form $\{ p_0 \}\ S_0$ `;` $\{$ **inv** $p \}\{$ **bd** $t \}$ **while** B **do** S **od** $\{ q \}$ doesn't diverge or cause runtime errors, you need to show:
  1. Initialization establishes the invariant: $\{ p_0 \}\ S_0 \{ p \}$ (because $p_0 \rightarrow wp(S_0, p)$ or $sp(p_0, S_0) \rightarrow p$).
  2. The loop body maintains the invariant: $\{ p \wedge B \}\ S \{ p \}$
  3. The loop postcondition implies the outer postcondition: $p \wedge \neg B \rightarrow q$
  4. The invariant ensures that at the **while** test, the bound is nonnegative ($p \rightarrow t \geq 0$) and evaluation of the loop test won't fail ($p \rightarrow D(B)$) and the loop body won't fail ($p \wedge B \rightarrow D(S)$)
  5. Evaluation of the loop body decreases the upper bound: $\{p \wedge B \wedge t = t_0\}\ S\ \{t < t_0\}$

F. Increasing and Decreasing Loop Variables

- We've looked at the simple summation loop

```
{n ≥ 0} i := 0; s := 0;
{inv p ≡ 0 ≤ i ≤ n ∧ s = sum(0,i)} {bd n-i}
while i < n do
      i := i + 1;
      s := s + i
od
{s = sum(0, n)}
```

- It's easy to find bound functions for simple loops like this one that always increment (or decrement) a loop variable but keep it within some range.
- For this particular loop, `s` is also getting larger, and since it's easy to verify that $n^2 \geq s$, we can use $n^2 - s$ as a loop bound. Adding two bound expressions yields a bound expression, so $n^2 - s + n - i$ is also a loop bound.
- More generally, if the invariant includes $i \leq e$, where `i` increases with each iteration, then $e - i$ is a good bound function; if $e \leq i$ where `i` decreases, then $i - e$ is a bound function; if $e \geq 0$, then just `i` works also.

G. Another Loop Example: Iterative GCD

- Not all loops modify only one loop variable with each iteration; they might modify many variables, they might modify some variables sometimes and other variables other times.

- **Definition**: For $x, y \in \mathbb{N}$, $x, y > 0$, the greatest common divisor of x and y, written $\texttt{gcd}(x, y)$, is the largest value that divides both x and y evenly (i.e., without remainder).
  - E.g., $\texttt{gcd}(300, 180) = \texttt{gcd}(2^2 * 3 * 5^2, 2^2 * 3^2 * 5) = 2^2 * 3 * 5 = 60$.
- Some useful `gcd` properties:
  - If $x = y$, then $\texttt{gcd}(x, y) = x = y$
  - If $x > y$, then $\texttt{gcd}(x, y) = \texttt{gcd}(x - y, y)$
  - If $y > x$, then $\texttt{gcd}(x, y) = \texttt{gcd}(x, y - x)$
- E.g., $\texttt{gcd}(300, 180) = \texttt{gcd}(120, 180)$, $\texttt{gcd}(120, 60) = \texttt{gcd}(60, 60) = 60$.
- Here's a minimal proof outline for an iterative `gcd`-calculating loop:

  $\{\texttt{x} > 0 \wedge \texttt{y} > 0 \wedge \texttt{x} = \texttt{X} \wedge \texttt{y} = \texttt{Y}\}$
  $\{\textbf{inv}\ p \equiv \texttt{x} > 0 \wedge \texttt{y} > 0 \wedge \texttt{gcd(X, Y)} = \texttt{gcd(x, y)}\}$
  $\{\textbf{bd}\ ???\}$ // to be filled-in
  while $\texttt{x} \neq \texttt{y}$ **do**
    if $\texttt{x} > \texttt{y}$ **then** $\texttt{x} := \texttt{x-y}$ **else** $\texttt{y} := \texttt{y-x}$ **fi**
  od
  $\{\texttt{x} = \texttt{gcd(X, Y)}\}$
- Here's a fully annotated version of the program (i.e., a full proof outline):

  $\{\texttt{x} > 0 \wedge \texttt{y} > 0 \wedge \texttt{x} = \texttt{X} \wedge \texttt{y} = \texttt{Y}\}$
  $\{\textbf{inv}\ p \equiv \texttt{x} > 0 \wedge \texttt{y} > 0 \wedge \texttt{gcd(X, Y)} = \texttt{gcd(x, y)}\}$
  $\{\textbf{bd}\ ???\}$ // to be filled-in
  while $\texttt{x} \neq \texttt{y}$ **do**
    $\{p \wedge \texttt{x} \neq \texttt{y}\}$
    if $\texttt{x} > \texttt{y}$ **then**
      $\{p \wedge \texttt{x} \neq \texttt{y} \wedge \texttt{x} > \texttt{y}\}\ \{p[\texttt{x-y/x}]\}\ \texttt{x} := \texttt{x-y}\ \{p\}$
    else
      $\{p \wedge \texttt{x} \neq \texttt{y} \wedge \texttt{x} \leq \texttt{y}\}\ \{p[\texttt{y-x/y}]\}\ \texttt{y} := \texttt{y-x}\ \{p\}$
    fi $\{p\}$
  od $\{p \wedge \texttt{x} = \texttt{y}\}\ \{\texttt{x} = \texttt{gcd(X, Y)}\}$
- We have a number of predicate logic obligations
  - $(\texttt{x} > 0 \wedge \texttt{y} > 0 \wedge \texttt{x} = \texttt{X} \wedge \texttt{y} = \texttt{Y}) \rightarrow p$
  - $p \wedge \texttt{x} \neq \texttt{y} \wedge \texttt{x} > \texttt{y} \rightarrow p[\texttt{x-y/x}]$
  - $p \wedge \texttt{x} \neq \texttt{y} \wedge \texttt{x} \leq \texttt{y} \rightarrow p[\texttt{y-x/y}]$
  - $p \wedge \texttt{x} = \texttt{y} \rightarrow \texttt{x} = \texttt{gcd(X, Y)}$
- With $p \equiv \texttt{x} > 0 \wedge \texttt{y} > 0 \wedge \texttt{gcd(X, Y)} = \texttt{gcd(x, y)}$, the substitutions are
  - $p[\texttt{x-y/x}] \equiv \texttt{x-y} > 0 \wedge \texttt{y} > 0 \wedge \texttt{gcd(X, Y)} = \texttt{gcd(x-y, y)}$
  - $p[\texttt{y-x/y}] \equiv \texttt{x} > 0 \wedge \texttt{y-x} > 0 \wedge \texttt{gcd(X, Y)} = \texttt{gcd(x, y-x)}$

- The given annotation combines the *sp* of the **if-else** test with the *wp* of the branches. For an example of a different annotation, if we use the *wp* of the entire **if-else**, we get

  $\{p \wedge \mathtt{x} \neq \mathtt{y}\}$ // where p is $\mathtt{x} > 0 \wedge \mathtt{y} > 0 \wedge \mathtt{gcd(X, Y)} = \mathtt{gcd(x, y)}$
  $\{(\mathtt{x} > \mathtt{y} \rightarrow p[\mathtt{x-y/x}]) \wedge (\mathtt{x} \leq \mathtt{y} \rightarrow p[\mathtt{y-x/y}])\}$
  if $\mathtt{x} > \mathtt{y}$ **then**
    $\{p[\mathtt{x-y/x}]\}$ $\mathtt{x} := \mathtt{x-y}$ $\{p\}$
  else
    $\{p[\mathtt{y-x/y}]\}$ $\mathtt{y} := \mathtt{y-x}$ $\{p\}$
  fi $\{p\}$

- We get one larger predicate logic obligations instead of two smaller ones:
  - $(p \wedge \mathtt{x} \neq \mathtt{y}) \rightarrow ((\mathtt{x} > \mathtt{y} \rightarrow p[\mathtt{x-y/x}]) \wedge (\mathtt{x} \leq \mathtt{y} \rightarrow p[\mathtt{y-x/y}]))$
- What about convergence?
  - The loop body sometimes (but not always) makes **x** smaller; it also sometimes (but not always) makes **y** smaller. Using the heuristic, let's add **x** and **y** to our possible bound expression. This gives us $\mathtt{x} + \mathtt{y}$ for the possible bound. The loop body always reduces one of **x** and **y**, so it always reduces $\mathtt{x} + \mathtt{y}$.
    - So we have $\{p \wedge \mathtt{x} \neq \mathtt{y} \wedge \mathtt{x} + \mathtt{y} = \mathtt{t_0}\}$ *loop body* $\{\mathtt{x} + \mathtt{y} < \mathtt{t_0}\}$
  - Show show that the loop bound is nonnegative, we need $p \rightarrow \mathtt{x} + \mathtt{y} \geq 0$. Since p implies $\mathtt{x} > 0$ and $\mathtt{y} > 0$, this follows easily.
- So our final minimally-annotated program is

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

- Expanding to a full proof outline, the material in green below is for the loop bound. We use $\mathtt{t_0}$ as a logical constant for the value of **x+y** at the top of the loop body. (As always, the actual name isn't interesting; it's the logical constant aspect that is.)

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

fi $\{p \wedge \texttt{x+y} < \texttt{t}_0\}$
od $\{p \wedge \texttt{x} = \texttt{y}\}$ $\{\texttt{x} = \texttt{gcd(X, Y)}\}$

- For this to work, we need $\texttt{x+y} = \texttt{t}_0$ to imply either $\texttt{(x-y)+y}$ or $\texttt{x+(y-x)} < \texttt{t}_0$ (depending on the **if-else** branch). These hold because $\texttt{(x-y)+y} = \texttt{x} < \texttt{x+y}$ (since y is positive) and $\texttt{x+(y-x)} = \texttt{y} < \texttt{x+y}$ (since x is positive.)

Loop Termination

CS 536: Science of Programming

A. Why

- Runtime errors make our programs not work, so we want to avoid them.
- Diverging programs aren't useful, so it's useful to know how to show that loops terminate.

B. Objectives

At the end of this activity you should be able to

- Calculate the domain predicate of an expression.
- Show what domain predicates need to hold within a program.
- Generate possible loop bounds for a given loop.
- State the extra obligations required to prove that a partially correct program is totally correct.

C. Questions

1. Consider the triple $\{\textbf{inv}\ p\}\ \{\textbf{bd}\ e\}\ \textbf{while}\ \texttt{i} < \texttt{n}\ \textbf{do}\ \ldots\ \texttt{i} := \texttt{i+1}\ \textbf{od}\ \{p \wedge \texttt{i} \geq \texttt{n}\}$. Assume $p \rightarrow \texttt{n} \geq \texttt{i}$. To show that this loop terminates, we need

   (1) $p \rightarrow \texttt{n-i} \geq 0$ (which holds by assumption) and

   (2) $\{p \wedge \texttt{i} < \texttt{n} \wedge e = \texttt{t}_0\}\ \ldots; \texttt{i} := \texttt{i+1}\ \{e < \texttt{t}_0\}$

   a. Does condition (2) hold if $e \equiv \texttt{n-i}$? (If so, we can use `n-i` as a bound expression.)

   b. Can we use `n-i+1` as a bound expression?

   c. Can we use `2*n-i` as a bound expression?

2. Use the same program as in Question 3 but assume $p \rightarrow \texttt{n} \geq \texttt{i-3}$, not $\texttt{n} \geq \texttt{i}$.

   a. Why does `n-i` now fail as a bound expression?

   b. Give an example of a bound expression that does work.

3. Consider the loop below. (Assume `n` is a constant and the omitted code does not change `j`.)

   a. Why does using just `j` as the bound function fail?

   b. Find an expression that involves `j` and prove that it's a loop bound. (You'll need to augment p.)

   $\{\texttt{n} \geq \texttt{-1}\}$
   `j := n;`
   $\{\textbf{inv}\ p \wedge \texttt{\_\_\_\_\_}\ \}\ \{\textbf{bd}\ \texttt{\_\_\_\_\_}\ \}$
   `while` `j` $\geq$ `-1`
   `do` … `j := j-1` … **`od`**

4. What is the minimum expression (i.e., closest to zero) that can be used as a loop bound for

   {**inv** n ≤ x+y} {**bd** …} **while** x+y > n **do** … y := y-1 **od** ?

   (Assume x and n are constant.)

5. Consider the loop {n > 0} k := n; {**inv** ???} **while** k > 1 **do** … k := k/2 **od** {…}
   a. Argue that *ceiling*(log_2 k) is a loop bound. (Augment the invariant as necessary.)
   b. Argue that k is a loop bound.
   c. Argue that *ceiling*(log_2 n) is **not** a loop bound. (Trick question.)

6. Let's look at the general problem of convergence of {**inv** p} **while** B **do** S **od** {q}. For each property below, briefly discuss whether it is (1) required, (2) allowable but not required, or (3) incompatible with the requirements.
   a. $p \rightarrow t \geq 0$
   b. $t < 0 \rightarrow \neg p$
   c. $\{p \wedge B \wedge t = t_0\}\ S\ \{t = t_0 - 1\}$
   d. $p \wedge t > 0 \rightarrow B$
   e. $\neg B \rightarrow t = 0$
   f. $\{p \wedge B \wedge t = t_0\}\ S\ \{t < t_0\}$

Solution to Activity 18 (Loop Termination)

1. (Termination of {**inv** *p*} {**bd** `n-i`} **while** `i < n` **do** … `i := i+1` **od**)
   a. Yes: $\{p \wedge \texttt{i} < \texttt{n} \wedge \texttt{n-i} = \texttt{t}_0\}$ … $\{\texttt{n-(i+1)} < \texttt{t}_0\}$ `i := i+1` $\{\texttt{n-i} < \texttt{t}_0\}$ requires `n-(i+1) < n-i`, which is true.
   b. Yes: Decrementing `i` certainly decreases `n-i+1`, and `n-i+1 > n-i ≥ 0`, which is the other requirement.
   c. Yes, but only if `n ≥ 0`: We know `n-i ≥ 0`, so `2*n-i ≥ n`, which is `≥ 0` if `n ≥ 0`. (If `n < 0` then `2*n-i` might be negative.)

2. If `n ≥ i-3`, then we only know `n-i ≥ -3`. (Note `n-i+3` works as a bound, however.)

3. (Decreasing loop variable)
   a. We can't just `j` as the bound expression because we don't know `j ≥ 0`. In fact, the loop terminates with `j = -2`.
   b. Since `j` is initialized to `n`, we can add `-2 ≤ j ≤ n` to the invariant and use `j+2` as the bound expression.
   c. We need to know that the invariant implies `j+2 ≥ 0` and that the loop body decreases `j+2`.

4. The smallest loop bound is `x+y-n`. We know it's ≥ 0 because `n ≤ x+y`, and we know it decreases by 1 each iteration, so at loop termination, `x+y-n = 0`, which implies that nothing `< x+y-n` can work as a bound.

5. (Θ(*log* `n`) loop)
   a. Add `0 ≤ k ≤ n ∧ n > 0` to the invariant. Since `k > 1`, we know *ceiling*(log_2 `k`) > 0, and halving `k` decreases *ceiling*(log_2 `k`) by one, so *ceiling*(log_2 `k`) ≥ 0. Thus *ceiling*(log_2 `k`) works as a loop bound.
   b. Since `k > 1`, halving `k` decreases it but leaves it ≥ 0.
   c. *ceiling*(log_2 `n`) doesn't decrease, since n is a constant, and constants aren't loop bounds.

6. (Loop convergence) Required are (a) $p \rightarrow t \geq 0$, (b) $t < 0 \rightarrow \neg p$ [i.e., the contrapositive of (a)], and (f) $\{p \wedge B \wedge t = \texttt{t}_0\}\ S\ \{t < \texttt{t}_0\}$. Property (c) $\{p \wedge B \wedge t = \texttt{t}_0\}\ S\ \{t = \texttt{t}_0\texttt{-1}\}$ is allowable but not required: It implies (f) but is stronger than we need. Property (e) $\neg B \rightarrow t = 0$ is allowable but not required. Property (d) $p \wedge t > 0 \rightarrow B$ is incompatible with the requirements (it would cause an infinite loop).

