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.

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, \( k := 0; \text{while } k < n \text{ do } \ldots \); \( k := k + 1 \) \text{ od} \ has \( n-k \) 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 \( \{\text{inv } p}\ \{bd \ t\} \text{ while } B \text{ do } S \text{ od } \{p \land \neg B\} \), 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.
    - \( \{p \land B \land t = b_0\} \ S \{p \land t < b_0\} \) where \( b_0 \) 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 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 $\text{sum}(0, n)$ program, we can use $n-k$ for the bound:

\[
\begin{align*}
\{n \geq 0\} & \quad k := 0; \quad s := 0; \\
\{\text{inv } p \equiv 0 \leq k \leq n \land s = \text{sum}(0, k)\} \\
\{bd n-k\} & \quad \text{while } k < n \ do \quad k := k + 1; \quad s := s + k \ od \\
\{s = \text{sum}(0, n)\}
\end{align*}
\]

• At the loop test, we always have $\geq 0$ iterations left: ($p \rightarrow n-k \geq 0$) because $p$ implies $0 \leq k \leq n$.

• Execution of the loop body lowers the bound. Let $t_0$ be our fresh logical variable, then we need $\{d \land k < n \land n-k = t_0\}$ 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.

• 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 $k := 0; \ while \ k + 1 \ do \ \ldots \ k := k + 1 \ od$, 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 $t \geq 0 \rightarrow B$ 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: $p \land \neg B \rightarrow t = 0$.

    • The bound expression doesn't have to decrease by exactly one:

    • Not required: $\{d \land B \land t = t_0\}$ loop body $\{t = t_0\}$

• 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 $\text{ceiling}(\log_2(R-L))$ 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 $k := k-1$, try $k$ for $t$.
  - If the invariant allows $k < 0$, then we need to make $t$ larger. E.g., if the invariant implies $k \geq -10$, then it implies $k + 10 \geq 0$, so maybe adding 10 to $t$ will help.

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

E. Increasing and Decreasing Loop Variables

- We’ve looked at the simple summation loop
  
  $\{ n \geq 0 \}$ $k := 0; s := 0$
  $\{ \text{inv} \} \{ \text{bd} \} \{ \text{od} \} \{ q \}$

  $\{n \geq 0 \} k := 0; s := 0$
  $\{\text{inv} \} \{ k \leq n \land s = \text{sum}(0,k) \} \{ \text{bd} \}$

  while $k < n$
  $\{ k := k + 1; \}$
  $\{ s := s + k \}$

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

- 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 $n^2 - s + n - k$ is also a loop bound.
More generally, if the invariant includes $k \leq e$, where $k$ increases with each iteration, then $e-k$ is a good bound function; if $e \leq k$ where $k$ decreases, then $k-e$ is a bound function; if $e \geq 0$, then just $k$ works also.

F. 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 $gcd(x, y)$, is the largest value that divides both $x$ and $y$ evenly (i.e., without remainder).

- E.g., $gcd(300, 180) = gcd(2^2 \cdot 3 \cdot 5^2, 2^2 \cdot 3^2 \cdot 5) = 2^2 \cdot 3 \cdot 5 = 60$.

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 minimal proof outline for an iterative $gcd$-calculating loop:

\[ \{x > 0 \land y > 0 \land x = x_0 \land y = y_0\} \]
\[ \{inv\ p = x > 0 \land y > 0 \land gcd(x_0, y_0) = gcd(x, y)\} \]
\[ \{bd ???\} // to be filled-in \]
\[ while \ x \neq y \ do \]
\[ \quad if \ x > y \ then \ x := x - y \ else \ y := y - x \ fi \]
\[ \od \]
\[ \{x = gcd(x_0, y_0)\} \]

Here's a full proof outline for partial correctness.

\[ \{x > 0 \land y > 0 \land x = x_0 \land y = y_0\} \]
\[ \{inv\ p = x > 0 \land y > 0 \land gcd(x_0, y_0) = gcd(x, y)\} \]
\[ \{bd ???\} // to be filled-in \]
\[ while \ x \neq y \ do \]
\[ \quad \{p \land x \neq y\} \]
\[ \quad if \ x > y \ then \]
\[ \quad \quad \{p \land x \neq y \land x > y\} \{p[x/y/x]\} \{x := x - y \{p\} \}
\[ \quad else \]
\[ \quad \quad \{p \land x \neq y \land x \leq y\} \{p[y/x]\} \{y := y - x \{p\} \}
\[ \quad fi \]
\[ \od \{p \land x = y\} \{x = gcd(x_0, y_0)\} \]

We have a number of predicate logic obligations

- $(x > 0 \land y > 0 \land x = x_0 \land y = y_0) \rightarrow p$
- $p \land x \neq y \land x > y \rightarrow p[x/y/x]"
• \( p \land x \neq y \land x \leq y \rightarrow p[y\times y] \)
• \( p \land x = y \rightarrow x = gcd(x_0, y_0) \)

- With \( p = x > 0 \land y > 0 \land gcd(x_0, y_0) = gcd(x, y) \), the substitutions are
  - \( p[x\times y/x] = x > 0 \land y > 0 \land gcd(x_0, y_0) = gcd(x, y) \)
  - \( p[y\times y/y] = x > 0 \land y > 0 \land gcd(x_0, y_0) = gcd(x, y-x) \)

- 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

\[
\{ p \land x \neq y \} \quad // \text{where } p \text{ is } x > 0 \land y > 0 \land gcd(x_0, y_0) = gcd(x, y) \\
\{ (x > y \rightarrow p[x\times y/x]) \land (x \leq y \rightarrow p[y\times y/y]) \} \\
if x > y then \\
\{ p[x\times y/x] \} x := x \times \{ p \} \\
else \\
\{ p[y\times y/y] \} y := y \times \{ p \} \\
fi \{ p \}
\]

- We get one larger predicate logic obligations instead of two smaller ones:
  - \( (p \land x \neq y) \rightarrow ((x > y \rightarrow p[x\times y/x]) \land (x \leq y \rightarrow p[y\times y/y])) \)

- 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 \( x+y \) for the possible bound. The loop body always reduces one of \( x \) and \( y \), so it always reduces \( x+y \).

  - So we have \( \{ 0 \land x \neq y \land x+y = t_0 \} \) loop body \( \{ x+y < t_0 \} \)

  - To show that the loop bound is nonnegative, we need \( p \rightarrow x+y \geq 0 \). Since \( p \) implies \( x > 0 \) and \( y > 0 \), this follows easily.

- So our final minimally-annotated program is

\[
\{ x > 0 \land y > 0 \land x = x_0 \land y = y_0 \} \quad // \text{initial values of } x \text{ and } y \\
\{ inv \ p = x > 0 \land y > 0 \land gcd(x_0, y_0) = gcd(x, y) \} \\
\{ bd \ x+y \} \\
while x \neq y do \\
\qquad if x > y then x := x-y else y := y-x \ fi \\
\{ od \} \\
\{ x = gcd(x_0, y_0) \}
\]

- To expand to a full proof outline for total correctness, we add the material in green below, for the loop bound. We use \( t_0 \) 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.)

\[
\{ x > 0 \land y > 0 \land x = x_0 \land y = y_0 \} \\
\{ inv \ p = x > 0 \land y > 0 \land gcd(x_0, y_0) = gcd(x, y) \land x+y \geq 0 \} \\
\{ bd \ x+y \} 
\]
while $x \neq y$ do
\{ $p \land x \neq y \land x + y = t_0$ \}

if $x > y$ then
\{ $p \land x \neq y \land x > y \land x + y = t_0$ \} $\{ p[x/y/x] \land (x-y)+y < t_0 \}$ $x := x-y \{ p \land x + y < t_0 \}$
else
\{ $p \land x \neq y \land x \leq y \land x + y = t_0$ \} $\{ p[y/x/y] \land x+(y-x) < t_0 \}$ $y := y-x \{ p \land x + y < t_0 \}$
fi
\{ $p \land x + y < t_0$ \}
\od\{ $p \land x = y$ \} $\{ x = \gcd(x_0, y_0) \}$

• 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 $(x-y)+y = x < x+y$ (since $y$ is positive) and $x+(y-x) = y < x+y$ (since $x$ is positive.)