Total Correctness: Avoiding Errors and Divergence

CS 536: Science of Programming, Spring 2023

2023-03-22: pp. 2,3,6-8; 2023-04-06: pp.2, 3

A. Why

- To argue that a program is totally correct, we need it to be partially correct, avoid runtime errors, and not diverge.
- To avoid runtime errors, we use domain predicates.
- To show that a loop terminates, we define a weak upper bound for the number of iterations left.

B. Objectives

At the end of this class you should understand

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

C. Rules for Total Correctness

- Up until now, we've been studying rules for partial correctness. Recall when we started to look at proof rules for correctness triples (back in Class 14), we had the definition for a general proof system. We can be more specific now and say:
  - ***Definition***: A ***proof system for partial correctness*** is the set of logical formulas determined by the sets of axioms and rules of inference for partial correctness.
  - ***Notation:*** $\vdash \{ p \} S \{ q \}$ means the correctness triple can be proved to be partially correct, using the rules we've seen. Since so far all we've been discussing is partial correctness, there hasn't been much call for the $\vdash$ notation, and we've been just saying "$\{ p \} S \{ q \}$ is provable" instead of "$\vdash \{ p \} S \{ q \}$".
- Now we'll look at proving total correctness. We'll take the rules for partial correctness and add avoidance of runtime errors and divergence of loops, and we'll have
  - ***Definition***: A ***proof system for total correctness*** is the set of logical formulas determined by the sets of axioms and rules of inference for total correctness.
  - ***Notation:*** $\vdash_{\text{tot}} \{ p \} S \{ q \}$ means the correctness triple can be proved to be totally correct, using the rules we've seen.

D. Avoiding Runtime Errors

- In class 11, we looked at domain predicates for expressions, statements, and predicates: $D(e)$, $D(S)$, and $D(p)$ guarantees that evaluation of e, S, or p won't cause a runtime error.

- One basic differences between rules for $\vdash$ and $\vdash_{tot}$ is that when a predicate p appears in a condition, we need it to be safe in the sense that $D(p)$ also holds. This comes up often enough that it's worth some notation:
  - ***Definition:*** A predicate p is ***safe*** if $p \to D(p)$. A correctness triple $\{p\}\,S\,\{q\}$ is ***safe*** if p and q are safe. The ***safe version*** of p, written $\downarrow p$, is $p \wedge D(p)$ and the ***safe version*** of $\{p\}\,S\,\{q\}$ is $\{\downarrow p\}\,S\,\{\downarrow q\}$. [We won't prove this, but if p is safe, then $D(p)$ is safe.]
- The second basic difference between rules for $\vdash$ and $\vdash_{tot}$ is that to conclude $\vdash_{tot} \{\downarrow p\}\,S\,\{\downarrow q\}$, we need the precondition to imply $D(S)$.
- In general, we have that for partial correctness, $\{wlp(S, q)\}\,S\,\{q\}$. For total correctness, $\{wp(S, \downarrow q)\}\,S\,\{\downarrow q\}$, where $wp(S, \downarrow q) \Leftrightarrow D(S) \wedge \downarrow wlp(S, \downarrow q)$. [2023-03-22]
  - (Note in class 11, we had $wp(S, q) \Leftrightarrow D(S) \wedge wlp(S, q) \wedge D(wlp(S, q)).$)[1]
- ***Definition:*** In outline form, the total correctness rules for the non-loop statements are:
  - $\{\downarrow p\}$ ***skip*** $\{\downarrow p\}$
  - $\{D(e) \wedge \downarrow wlp(x := e, \downarrow p)\}\ x := e\ \{\downarrow p\}$ [2023-03-22] the wp(x:=e,↓p) → D(e)]
  - $\{D(e) \wedge \downarrow p \wedge x = x_0\}\ x := e\ \{(D(e) \wedge \downarrow p)[x_0/x] \wedge x = e[x_0/x]\}$.
  - $\{D(S_1) \wedge \downarrow p\}\,S_1\,;\{D(S_2) \wedge \downarrow q\}\,S_2\,\{\downarrow r\}$.
  - $\{\downarrow B \wedge (\downarrow B \to D(S_1)) \wedge (\downarrow \neg B \to D(S_2)))\}$ [2023-03-22] include a precondition p?
    if B ***then*** $\{\downarrow B \wedge D(S_1)\}\,S_1\,\{\downarrow q_1\}$ ***else*** $\{\downarrow \neg B \wedge D(S_2)\}\,S_2\,\{\downarrow q_2\}$ ***fi*** $\{\downarrow q_1 \wedge \downarrow q_2\}$.
- ***Example 1***: What wp can we use for p in $\{p\}\,x := sqrt(y/z)\,\{x^2 \le x\}$?
  - The postcondition is already safe, so it needs no modification.
  - For $S \equiv x := sqrt(y/z)$, we have $D(S) \Leftrightarrow z \neq 0 \wedge y/z \ge 0$.
  - $w \equiv wlp(S, x^2 \le x) \Leftrightarrow sqrt(y/z)^2 \le sqrt(y/z)$ [2023-04-06] and $D(w) \Leftrightarrow z \neq 0 \wedge y/z \ge 0$
  - So $p \equiv D(S) \wedge w \wedge D(w)$
    $\Leftrightarrow (z \neq 0 \wedge y/z \ge 0) \wedge (sqrt(y/z)^2 \le y/z) \wedge (z \neq 0 \wedge y/z \ge 0)$
    $\Leftrightarrow z \neq 0 \wedge y/z \ge 0 \wedge sqrt(y/z)^2 \le y/z$.

E. 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.***

(Note to self: Verify that $wp(S, q) \to D(q)$. If not, beef up wp in class 11 and here.)

  - 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 natural number [2023-03-22] 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 $\{\ \textbf{bd}\ t\ \}$, so a typical loop has the form $\{\ \textbf{inv}\ p\ \}\ \{\ \textbf{bd}\ t\ \}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{\ p \wedge \neg B\ \}$.
- Note we aren't required to calculate the value of t at runtime, since it's a logical expression.

F. Properties of Bound Functions

- For t to be a valid bound expression, it needs to meet the two following properties:
  - $p \to t \geq 0$
    - 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.
  - $\{\ p \wedge B \wedge t = t_0\ \}\ S\ \{\ t < t_0\ \}$ where t_0 is a fresh logical variable. [2023-04-06] (For termination, we only need to prove $t < t_0$. When proving partial correctness, we need to prove p.
    - 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.

- ***Example 2***: For the $sum(0, n)$ program, we can use $n–k$ for the bound:

  $\{\ n \geq 0\ \}\ k := 0;\ s := 0;$
  $\{\ \textbf{inv}\ p \equiv 0 \leq k \leq n \wedge s = sum(0, k)\ \}$
  $\{\ \textbf{bd}\ n–k\ \}\ \textbf{while}\ k < n\ \ \textbf{do}\ k := k+1;\ s := s+k\ \ \textbf{od}$
  $\{\ s = sum(0, n)\ \}$

  - We need $p \to n–k \geq 0$: At the loop test, p implies $0 \leq k \leq n$, which implies $n–k \geq 0$.
- ***Definition:*** A ***progress step*** is a statement that reduces the value of the bound function. Every loop iteration needs to execute a progress step.

- We need the loop body to contain a progress step: Here, we need to decrease $k–n$: 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 $k := k+1$ (and no other change to k), it works as a progress step.
  - In symbols, we find that $\{ n–k = t_0 \}\ \{ n–(k+1) < t_0 \}\ k := k+1\ \{ n–k < t_0 \}$ is a correct full outline.

Other 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 3***: For the loop $k := 0;$ ***while*** $k < n$ ***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 $t \geq 0 \rightarrow B$ would cause divergence: Since $p \rightarrow t \geq 0$, if $t \geq 0 \rightarrow B$, then $p \rightarrow B$, so B would be true at every loop test.
- $\boldsymbol{p \wedge B \rightarrow t > 0}$ ***is required***: When p and B hold, we run the loop body, which should decrease t but leave it nonnegative. Equivalently, $p \wedge t = 0 \rightarrow \neg B$ because if t is zero, then 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.
- $\boldsymbol{p \wedge \neg B \rightarrow t = 0}$ ***is not required***:. Since t doesn't have to be a strict upper bound, it doesn't have to be zero on termination. Also not required: $p \wedge t > 0 \rightarrow B$. This is effectively the contrapositive of $p \wedge \neg B \rightarrow t = 0$.
  - 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*** $\Theta(t)$ because t is not required to be a strict upper bound.

- $\boldsymbol{\{ p \wedge B \wedge t = t_0 \}}$ *loop body* $\boldsymbol{\{ t – t_0 = 1 \}}$ ***is not required***. We must decrease t by at least one, but more than one is fine.
- ***Example 4***: For searches, t is often the size of the search space. For binary search, if $p \rightarrow left < right$ (where $left$ and $right$ are the endpoints of the search), then $right–left$ is a perfectly fine upper bound even though $ceiling(\log_2(right–left))$ is tighter.

G. 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 or some variable x that the loop body decreases, add x to t.

- For each variable or 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 $t \le e$, then $e - t \ge 0$, so we can use $e - t$ as our new candidate t.

- ***Example 5***: 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 $k \ge -10$, then it implies $k + 10 \ge 0$, so our new candidate bound function is $k + 10$.
- ***Example 6***: For a loop that sets $k := k + 1$, try $(-k)$ (i.e., $0 - k$) for t.
  - If $-k$ can be negative, we should do something to increase it. E.g., if the invariant implies $k \le e$, then it implies $e - k \ge 0$, so adding e to t would help.

H. 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 $n - k \ge 0$ because the invariant includes $k \le n$, 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 $e + n - k - s \ge 0$. 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 $n^2 - s \ge 0$ for all $n \in \mathbb{N}$.
  - This gives $n^2 + n - k - s$ as a bound function.
- ***Second and third bound functions***
  - Just $n - k$ by itself is as a bound function: $n - k$ is nonnegative and decreases each iteration.
  - Similarly, just $n^2 - s$ by itself is a bound function: $n^2 - s$ is nonnegative and decreases with each iteration.
- ***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. If we had found bound functions $n-k$ and n^2-s individually, we could have combined them to show that $n^2+n-k-s$ is a loop function.

I. 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 $gcd(x, y)$, is the largest value that divides both x and y evenly (i.e., without remainder).
- If you know the prime factorizations of x and y, you can easily find their gcd. E.g., $gcd(300, 180) = gcd(2^2 * 3 * 5^2, 2^2 * 3^2 * 5) = 2^2 * 3 * 5 = 60$. But in general, finding prime factorizations is difficult, and it's been known since ancient times that there are faster simpler ways to calculate a gcd.
- The technique relies on 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 correctness of an iterative gcd-calculating loop. Since a bound function has yet to be found, the outline is for partial correctness only.

  $\{x > 0 \wedge y > 0 \wedge X = x \wedge Y = y\}$ [2023-03-22] X and Y are the same as x_0 and y_0
  $\{\textbf{inv}\ p \equiv x > 0 \wedge y > 0 \wedge gcd(X, Y) = gcd(x, y)\}$
  $\{\textbf{bd}\ ???\}$
  while $x \neq y$ ***do***
    if $x > y$ ***then*** $x := x - y$ ***else*** $y := y - x$ ***fi***
  od
  $\{x = y = gcd(X, Y)\}$ [2023-03-22]

- Expanding the minimal outline gives a full outline for partial correctness.

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

- We have a number of predicate logic obligations:

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

- With $p \equiv x > 0 \wedge y > 0 \wedge gcd(X, Y) = gcd(x, y)$, the substitutions are
  - $p\,[x{-}y/x] \equiv x{-}y > 0 \wedge y > 0 \wedge gcd(X, Y) = gcd(x{-}y, y)$
  - $p\,[y{-}x/y] \equiv x > 0 \wedge y{-}x > 0 \wedge gcd(X, Y) = gcd(x, y{-}x)$
- 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 the various predicate logic obligations are of basically the same proof difficulty.
- [2023-03-22] We don't have to worry about runtime errors because nothing in the program can cause one.
- 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 minimal outline for total correctness of the 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
  $\{x = gcd(X, Y)\}$

- To get a full outline for total correctness, we can take a full outline for partial correctness and add the termination requirements, shown *in blue* below.

  $\{x > 0 \wedge y > 0 \wedge X = x \wedge Y = y\}$ // X and Y are the initial values of x and y
  $\{$ ***inv*** $p \equiv x > 0 \wedge y > 0 \wedge gcd(X, Y) = gcd(x, y) \wedge x{+}y \geq 0\}$
  $\{$ ***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).

J. Semantics of Convergence

- Here's a semantic assertion about bound functions and loop termination.
- ***Lemma (Loop Convergence)***: Let $W \equiv \{ \textbf{inv } p \} \{ \textbf{bnd } t \}$ ***while*** B ***do*** S ***od*** be a loop annotated with an invariant and bound function. Assume we can prove $\{ p \} W \{ p \wedge \neg B \}$ (partial correctness of the loop) and [2023-03-22] (p→t≥0) and $\{ p \wedge B \wedge t = t_0 \} S \{ p \wedge t < t_0 \}$ (total correctness of the loop body). Then if $\sigma \models p \wedge B \wedge t = t_0$, then $\perp_d \notin M(W, \sigma)$.
  - Proof omitted. (It's by induction on t and pretty straightforward.)

K. *** Total Correctness of a Loop ***

- To show total correctness of $\{ p_0 \} S_0 ; W$ where $W \equiv \{ \textbf{inv } p \} \{ \textbf{bd } t \}$ ***while*** B ***do*** S ***od*** $\{ q \}$, we need
  - Partial correctness: $\{ p_0 \} S_0 ; W \{ q \}$. [2023-03-22] change indentation
    1. Initialization establishes the invariant: $\{ p_0 \} S_0 \{ p \}$. E.g., $\{ p_0 \} \{ wp(S_0, p) \} S_0 \{ p \}$ 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: $\models_{tot} \{ p_0 \} S_0 ; W \{ T \}$
    4. No runtime errors during initialization $(p_0 \rightarrow D(S_0))$ nor 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. $p \rightarrow D(B) \wedge (B \rightarrow D(S))$, where p and B are safe
2. $p \rightarrow \downarrow(t \geq 0)$ [2023-03-22] drop and D(t)
3. $\vdash_{tot} \{ p \wedge B \wedge t = t_0 \} S \{ p \wedge t < t_0 \}$
4. $\vdash_{tot} \{ \textbf{inv } p \} \{ \textbf{bnd } t \}$ ***while*** B ***do*** S ***od*** $\{ p \wedge \neg B \}$ while 1, 2, 3

answer line