Total Correctness for Loops

Stefan Muller
based on material by Jim Sasaki

CS 536: Science of Programming, Fall 2023
Lecture 17

1 You Are Here

	Partial Correctness	Total Correctness
Programs w/o loops or array assignmnents	X	X
Loops	X (Lectures 14-15)	Today
Array assignments	Next lecture	Sort of?
(More stuff to come)		

So far, we've studied:

- How to prove partial correctness triples (and find weakest liberal preconditions) for programs without loops or array assignments.
- The $D()$ predicate for telling if a program without loops will have an error, which meant we could turn partial correctness into total correctness, for program without loops. *Note:* We also saw $D()$ for array assignments, so once we know how to show partial correctness for these, we'll immediately have total correctness too.
- Loop invariants, which allow us to prove partial correctness for programs with loops. We can also tell whether the body of a loop will have an error with the $D()$ predicate, but we still don't know how to prove that a program with a loop terminates. That's the subject of today's class.

2 Loop Bounds

- How do we tell if a loop will terminate?
- For some loops (e.g., almost all `for` loops), we can count how many iterations are left. For example, in the program below, the loop always has $n - k$ iterations left.

$k := \overline{0}$;
while $k < n$
do
  S;
  $k := k + 1$
od

- If we can tell how many iterations are left and it's a finite number, then clearly the loop terminates.
- But, of course, we can't do this for all loops, not even all loops that terminate.
- A more general observation is that we don't need the exact number of iterations left, an upper bound works.

- If we know there are *at most* N iterations left, then we still know the loop terminates, even if we don't know exactly *when*.
- Also, this isn't going to be an exact number—it's going to be an expression, possibly containing some combination of program variables, ghost variables, and mathematical operations.
- We'll call this expression the *bound expression* or *loop bound*.

3 Loop Bounds in Proofs

We'll write a loop bound in a proof as $\{\textbf{dec } t\}$, where t is our bound expression, sort of like how we write loop invariants:

$\{\textbf{inv } P\}$
$\{\textbf{dec } t\}$
while e
do
  S;
od

In our example above:

$k := \overline{0}$;
$\{\textbf{inv } 0 \leq k \leq n\}$
$\{\textbf{dec } n - k\}$
while $k < n$
do
  S
  $k := k + 1$
od

To prove that a loop terminates, we need to prove that the loop bound is actually always an upper bound on how many iterations are left. We do that by showing two properties of the loop bound t:

- $P \Rightarrow t \geq 0$. We can't have less than zero iterations left!
- $\{P \wedge e \wedge t = t_0\}\ S\ \{P \wedge t < t_0\}$. t_0 is a ghost variable storing the value of the loop bound before the loop. This triple says that every iteration of the loop causes the loop bound to decrease, which makes sense: the number of iterations we have left can't increase as we do more iterations! To show this, we get to assume the loop invariant and loop condition, since both will be true if we enter the loop body.

The idea is that if we have an expression t whose value is an integer and is set at the beginning of the loop, and running the loop decreases t by at least one, and t can't become negative, then the loop can't possibly run forever.[1] It's like driving a car: if you start out with some amount of gas in your tank and driving uses up gas, your car will stop eventually.[2]

Note: To get full total correctness, we also have to avoid runtime errors, which we can do using the $D()$ predicate we saw in the weakest precondition lectures.

Example 1: For the sum program, we can use $n - k$ for the bound. We'll write additional facts we need for total correctness in red.

[1] If we allowed real-valued loop bounds, we'd need to be more careful: e.g., if the loop bound starts out positive and is divided by two every iteration, it would be allowed to loop forever.

[2] Some languages use a similar idea of bounds to guarantee that all programs terminate. The virtual machine for the Ethereum blockchain platform calls its bound *gas* using this same analogy.

	$\{n \geq 0\}$
$s := \overline{0}$;	$\{n \geq 0 \wedge s = 0\}$
$k := \overline{0}$;	$\{n \geq 0 \wedge s = 0 \wedge k = 0\} \Rightarrow \{s = sum(0,k) \wedge k \leq n\}$
$\{\textbf{inv}\ s = sum(0,k) \wedge k \leq n\}$	
$\{\textbf{dec}\ n - k\}$	
while$(k < n)$ do	$\{s = sum(0,k) \wedge k \leq n \wedge k < n \wedge n - k = t_0\}$
	$\Rightarrow \{s + k = sum(0, k+1) \wedge k + 1 \leq n \wedge n - k - 1 < t_0\}$
$\quad s := s + k$;	$\{s = sum(0, k+1) \wedge k + 1 \leq n \wedge n - k - 1 < t_0\}$
$\quad k := k + \overline{1}$	$\{s = sum(0,k) \wedge k \leq n \wedge n - k < t_0\}$
od	$\{s = sum(0,k) \wedge k \leq n \wedge k \geq n\} \Rightarrow \{s = sum(0,n)\}$

If we use $t = n - k$ and we add $t = t_0$ to the precondition of the loop body and $t < t_0$ the postcondition, and propagate this backward through the loop body using weakest preconditions, we end up with the proof obligation

$$n - k = t_0 \Rightarrow n - k - 1 < t_0$$

which is clear.

We also have to prove that the first requirement of a loop bound is met:

$$s = sum(0,k) \wedge k \leq n \Rightarrow n - k \geq 0$$

Again, this is clear, because $k \leq n$.

4 Side Note: Requirements for a Bound Expression

We talked about two conditions for a bound expression (being nonnegative and decreasing with each iteration). There are some other conditions that are implied by these, as well as some conditions that often hold, but don't need to.

4.1 Hidden Requirements

- *The bound expression can't be constant* (constants don't decrease!)

  Example 2. For the sum program, people often guess n as a loop bound. Indeed, n is initially an upper bound on the number of iterations of the loop. But as k increases, we get a tighter bound. The number of iterations left needs to decrease as we do more iterations.

- *A nonnegative bound can't always imply the loop condition.* One might think that we should have $t \geq 0 \Rightarrow e$. After all, we want $t \geq 0$ while the loop is running. However, we also have $P \Rightarrow t \geq 0$, which would then mean that $P \Rightarrow e$, and since P is always true during the loop, this means that the loop condition never becomes false, and the loop doesn't terminate. Instead of proving termination, we've now actually proven **non**termination!

- *If the loop condition is true, the bound must be strictly positive*, that is $P \wedge e \Rightarrow t > 0$ (or an equivalent fact, $P \wedge t = 0 \Rightarrow \neg e$). If e is true, we're going to do another iteration of the loop, but there's no room for the loop bound to decrease. This is like still having your foot on the gas pedal when the tank is empty: you're going to have a problem.

4.2 Properties Allowed but not Required

- *The bound does not have to go down to zero*, i.e. $P \wedge \neg e \Rightarrow t = 0$ is not required (we just need an upper bound, it can be a loose one).

- **But:** the number of iterations remaining *does* have to be $\in O(t)$ (by definition of big-O)

- *The number of iterations remaining doesn't have to be* $\in \Theta(t)$. There's no requirement at all on how tight an upper bound the loop bound is: 2^{n-k} also works as a loop bound for the "sum" program: it's always nonnegative and decreases at every iteration of the loop.
- *The bound doesn't need to decrease by exactly one*, i.e. $\{P \wedge e \wedge t = t_0\}\ S\ \{t - t_0 = 1\}$ is *not* required.

  Example 3: For search problems, we're generally decreasing the size of the search space on each iteration. For a linear search, we're decreasing the size by 1 each time. For binary search, however, $R-L$ (where L and R are the left endpoints of the search space) works great as a loop bound.

5 Finding a Bound Expresssion

As with finding a loop invariant, there's no algorithm for finding a bound expression. In fact, there fundamentally *cannot* be, as this would solve the Halting Problem. As an example, consider the following program:

```
while n > 1 do
  if n%2 = 0 then
    n := n/2
  else
    n := 3 * n + 1
  fi
od
```

The existence of a bound expression for this loop is known as the *Collatz Conjecture*, and it's a famous unsolved problem in mathematics (though not as long-standing as the Goldbach Conjecture, which we saw on HW1).

However, as with funding a loop invariant, there are some heuristics that will often work, and they're generally easier to apply than the heuristics for finding loop invariants.[3]

Here's the basic idea:

1. Start with $t = 0$.
2. For a variable x that decreases in the loop body, add x to t.
3. For a variable y that increases in the loop body, substract y from t.
4. If t may become negative, try to find some large expression e (often a constant) such that $e + t \geq 0$ and add it to t.

Example 4: For a loop that sets $k := k - 1$, try $t = k$. If this might let t become negative, try to find a constant to add (e.g., maybe the loop condition is $k \geq -10$, so k won't work as a bound expression, but $k + 10$ would).

Example 5: For a loop that sets $k := k + 1$, try $t = -k$. This makes it even more likely that t can be negative, so we need to find something to add to the bound. For example, in our "sum" loop, we know that k doesn't go above n, so we use $n - k$ as the bound instead of $-k$.

Example 6: Consider our sum program again. Using our heuristic, our first candidate might be $-k - s$, since both k and s increas in the loop, but that doesn't work because it's negative. We need to find something to add to t so that it's always nonnegative. We can add n, which helps because $k \leq n$, so let's try $n - k - s$. Unfortunately, this still fails: n "cancels out" k, but $-s$ can still cause this to become negative. However, we can also bound s: we know it's the sum of the first n numbers. It turns out this is always less than n^2, so $n^2 - n - k - s$ works as a bound expression.

A given loop can have many valid bound expressions. We've already seen that $n-k$ works as a bound for the "sum" loop. In addition, $n^2 - s$ by itself would work too: it decreases each iteration (because s increases and it's always nonnegative).

[3] In fact, you've already seen these in action. We glossed over this in class last time, but Dafny actually requires you to prove total correctness, not partial correctness? So why did it say that our programs were verified even though we didn't supply a bound expression or prove that it decreases? Because it was able to figure it out itself.

5.1 Modifying Bound Functions

There are a few modifications we can do to bound expressions that result in valid bound expressions:

- If t is a bound expression, then so is $a \cdot t^n + b$ for any positive a, b, and n. This is why we can always add a constant to a bound expression.
- If t_1 and t_2 are bound expressions, then so are $t_1 + t_2$ and $t_1 \times t_2$. So, if we knew that $n - k$ and $n^2 - s$ are bound functions, we know $n^2 + n - k - s$ is as well.

6 Example: Iterative GCD

In this example, we present an efficient algorithm for computing greatest common divisor (this algorithm was developed by Euclid around 300BC).

Definition: For $x, y \in \mathbb{N}$, where $x, y > 0$, the *greatest common divisor* of x and y (written $\gcd(x, y)$) is the largest value that divides both x and y evenly. For example, $\gcd(300, 180) = 60$.

Some useful properties about the GCD (we won't prove these):

- If $x = y$, then $\gcd(x, y) = x = y$.
- If $x > y$, then $\gcd(x, y) = \gcd(x - y, y)$.
- If $y > x$, then $\gcd(x, y) = \gcd(x, y - x)$.

Here's the algorithm, which uses those properties:

$\{x > 0 \wedge y > 0 \wedge x_0 = x \wedge y_0 = y\}$
$\{\textbf{inv}\ P \triangleq x > 0 \wedge y > 0 \wedge \gcd(x_0, y_0) = \gcd(x, y)\}$
$\{\textbf{dec}\ ???\}$
while $x \neq y$ **do**
  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:

$\{x > 0 \wedge y > 0 \wedge x_0 = x \wedge y_0 = y\}$
$\{\textbf{inv}\ P \triangleq x > 0 \wedge y > 0 \wedge \gcd(x_0, y_0) = \gcd(x, y)\}$
$\{\textbf{dec}\ ???\}$
while $x \neq y$ **do** $\{P \wedge x \neq y\}$
  if $x > y$ **then** $\{P \wedge x > y\} \Rightarrow \{[x - y/x]P\}$
    $x := x - y$ $\{P\}$
  else $\{P \wedge x < y\} \Rightarrow \{[y - x/y]P\}$
    $y := y - x$ $\{P\}$
  fi $\{P\}$
od $\{P \wedge x = y\} \Rightarrow \{x = \gcd(x_0, y_0)\}$

We have a number of predicate logic proof obligations:

- $x > 0 \wedge y > 0 \wedge x_0 = x \wedge y_0 = y \Rightarrow P$. This is clear since $x_0 = x \wedge y_0 = y$ implies $\gcd(x_0, y_0) = \gcd(x, y)$.
- $P \wedge x > y \Rightarrow [x - y/x]P$, that is

  $$x > 0 \wedge y > 0 \wedge \gcd(x_0, y_0) = \gcd(x, y) \wedge x > y \Rightarrow x - y > 0 \wedge y > 0 \wedge \gcd(x_0, y_0) = \gcd(x - y, y)$$

  This holds by the second fact about GCD.

- $P \wedge x < y \Rightarrow [y - x/y]P$. This is similar and holds by the third fact.
- $P \wedge x = y \Rightarrow x = \gcd(x_0, y_0)$. If $x = y$, then by the first fact about GCD, we have $x = \gcd(x, y)$, and by the loop invariant, this is equal to $\gcd(x_0, y_0)$.

Now how about a loop bound? Applying the heuristics, the loop body decreases x and y, so let's try $x + y$. Both x and y are always positive, so it's clear that $x + y$ is always nonnegative. The fact that it always decreases is a bit more subtle, but also true: while x or y might remain the same each iteration, *one* of them always decreases (and the other stays the same), so $x + y$ always decreases.

Let's expand the partial correctness proof outline into a proof outline for total correctness, using this bound.

	$\{x > 0 \wedge y > 0 \wedge x_0 = x \wedge y_0 = y\}$
$\{\textbf{inv}\ P \triangleq x > 0 \wedge y > 0 \wedge \gcd(x_0, y_0) = \gcd(x, y)\}$	
$\{\textbf{dec}\ x + y\}$	
while $x \neq y$ do	$\{P \wedge x \neq y \wedge x + y = t_0\}$
if $x > y$ then	$\{P \wedge x > y \wedge x + y = t_0\} \Rightarrow \{[x - y/x]P \wedge x - y + y < t_0\}$
$x := x - y$	$\{P \wedge x + y < t_0\}$
else	$\{P \wedge x < y \wedge x + y = t_0\} \Rightarrow \{[y - x/y]P \wedge x + y - x < t_0\}$
$y := y - x$	$\{P \wedge x + y < t_0\}$
fi	$\{P \wedge x + y < t_0\}$
od	$\{P \wedge x = y\} \Rightarrow \{x = \gcd(x_0, y_0)\}$

Our additional proof obligations are:

- $P \rightarrow x + y \geq 0$. This is clear.
- $P \wedge x > y \wedge x + y = t_0 \Rightarrow x - y + y < t_0$. Because $y > 0$ and $x + y = t_0$, we have $x < t_0$.
- $P \wedge x < y \wedge x + y = t_0 \Rightarrow x + y - x < t_0$. As above, we have $y < t_0$.

7 Additional facts about covergence

Let $W \triangleq \{\textbf{inv}\ P\}\{\textbf{dec}\ t\}$while e do S od. If we can prove total correctness of W, that is, if

$$\vdash [\ P \wedge e \wedge t = t_0\ ]\ S\ [\ P \wedge t < t_0\ ]$$

then if $\sigma \vDash P \wedge e \wedge t = t_0$, we have $M(S, \sigma, \neq)\emptyset$.

In addition, we now have a total correctness rule for While loops:

$$\frac{\vdash [\ P \wedge e \wedge t = t_0\ ]\ s\ [\ P \wedge t < t_0\ ] \qquad P \Rightarrow t \geq 0 \qquad P \Rightarrow D(e)}{\vdash [\ P\ ]\ \{\textbf{inv}\ P\}\{\textbf{dec}\ t\}\text{while } e \text{ do } s \text{ od } [\ P \wedge \neg e\ ]}\ (\textsc{While})$$

Above, proving $[\ P \wedge e \wedge t = t_0\ ]\ s\ [\ P \wedge t < t_0\ ]$ just means proving partial correctness and a lack of runtime errors.