CIS 575. Introduction to Algorithm Analysis
Material for February 12, 2024

Loop Invariants to Prove Iterative Programs Correct

1 Why Correctness Proofs

The *Howell* textbook states in its preface:

> This book is motivated in part by the author's belief that people do not fully understand an algorithm until they are able to prove its correctness.

This belief is shared by numerous other computer scientists (including the instructor of this course) and is further explained on p.8 of that textbook:

> The ability to prove algorithm correctness is quite possibly the most underrated skill in the entire discipline of computing. First, knowing how to prove algorithm correctness also helps us in the design of algorithms. Specifically, once we understand the mechanics of correctness proofs, we can design the algorithm with a proof of correctness in mind. This approach makes designing correct algorithms much easier. Second, the exercise of working through a correctness proof — or even sketching such a proof — often uncovers subtle errors that would be difficult to find with testing alone. Third, this ability brings with it a capacity to understand specific algorithms on a much deeper level. Thus, the ability to prove algorithm correctness is a powerful tool for designing and understanding algorithms. Because these activities are closely related to programming, this ability greatly enhances programming abilities as well.

In this set of notes, we shall work out detailed correctness proofs for simple algorithms. Later in the course, as we develop more advanced algorithms, we shall still aim to design them "with a proof of correctness in mind" though we shall seldom do rigorous proofs but rather settle for informal proof sketches — which is far better than not thinking about correctness at all (we should not let the perfect be the enemy of the good).

2 Correctness Proofs for Iterative Algorithms

In this set of notes, we shall mostly focus on algorithms that are built around **while** loops and thus of the form

P
while G
  B

where P is the *preamble*, G the *loop guard*, and B the *loop body*. To reason about such a loop, a crucial step is to find an assertion Φ that is a **loop invariant** in that it holds *every time* the loop guard is evaluated.

To check that a given assertion Φ is indeed a proper loop invariant, it suffices to verify that

1. Φ holds the first time that G is evaluated, that is, the preamble P will **establish** Φ
2. if Φ holds before one iteration then Φ also holds before the next, that is, executing (from a state satisfying Φ and also G) the body B will **maintain** Φ (though Φ may be temporarily violated inside B, it must be restored at the end).

If Φ is a proper invariant, we know that when (if) the loop guard G becomes false we have not just $\neg G$ but also Φ. Hence we can infer that

3. at loop exit, the assertion $\Phi \wedge \neg G$ holds

provided that there is no other way to exit the loop than through the loop guard becoming false, as is the case for a *structured program*. For languages that allow control to jump out of loops, programs using that feature are very hard to reason about, as explained already in 1968 by Edsger Dijkstra in his classical letter *Goto Statement Considered Harmful*.

Assuming that we work with structured programs, we can present conditions that suffice for ensuring that if we execute a `while` loop of the above form from a state satisfying a given precondition *Pre*, the resulting state will satisfy a given postcondition *Post*. The conditions are (other terminology may be used by other writers)

1. (**Establish**) $\{Pre\}\ P\ \{\Phi\}$ (executing the preamble P in a state satisfying *Pre* will establish the loop invariant Φ)
2. (**Maintain**) $\{\Phi \wedge G\}\ B\ \{\Phi\}$ (executing the body B in a state satisfying the invariant Φ and also the loop guard G will result in a state that satisfies the loop invariant Φ)
3. (**Correctness**) $\Phi \wedge \neg G$ logically implies *Post*.

These 3 conditions suffice for *partial* correctness: **if** the program terminates **then** the result is as required. But we will very often want also *total* correctness: the program **always** terminates **and** with result as required. To ensure that, we need to add an extra item to the checklist:

4. (**Terminate**) For all states satisfying the loop invariant Φ, the loop guard G will eventually become false.

In the subsequent notes, we shall apply that checklist to verify the correctness of several iterative algorithms.