CSE 127 Notes: Proofs of Correctness and Recursion

Bennet Yee

Mar 3-5, 2003

1 Proof Techniques

In previous lectures, we have been introduced to the notions of preconditions, postconditions, and the use of loop invariants to prove the correctness of loops. The proof technique of using loop invariants corresponds to the notion of *weak induction* in mathematics: let $P(i)$ is some property of the integer i. Then then statements

1. $P(k)$ is true ("base case")
2. If $P(n)$ is true, then so is $P(n+1)$ ("induction step")

implies that $\forall n \geq k : P(n)$ holds.

Weak induction is often compared to a ladder: you climb from one rung of the ladder (integer n) to the next rung $(n+1)$.

2 Recursive Algorithms

To prove recursive algorithms correct, we will generally use a proof technique that corresponds to *strong induction* in mathematics:

1. $P(k)$ is true ("base case");
2. If $P(k), P(k+1), \ldots, P(n)$ is true, then so is $P(n+1)$ ("induction step")

proves that $\forall n \geq k : P(n)$ holds.

Figure 1: `quicksort` utility: `median_to_front`

```
1   #define SWAP(x,n)       do {t=x[n];x[n]=x[n+1];x[n+1]=t;} while(0)
2   #define SSWAP(x,n)      do {if (x[n] > x[n+1]) SWAP(x,n);} while(0)
3
4   /* nelt >= 3 */
5   static void     median_to_front(int     *base,
6                                   int     nelt)
7   {
8           int     t;
9           SSWAP(base,0); SSWAP(base,1); SSWAP(base,0);
10          SWAP(base,0); /* cheap version -- not quite median value */
11  }
```

To how this is used, let us look at its use in the concrete example of *quicksort*. Figures 1 and 2 shows the source file `quicksort.c` contents split across two pages.

Figure 2: `quicksort` code

```
1  /* 0 <= nelt < INT_MAX, base[0..nelt-1] valid */
2  void    quicksort(int   *base,
3                    int   nelt)
4  {
5          int pivot, t, hi, lo;
6
7          switch (nelt) { /* base cases */
8          case 2: SSWAP(base,0); /* fall through */
9          case 0: case 1: return;
10         }
11         /* 3 <= nelt */
12         median_to_front(base, nelt); pivot = base[0];
13         lo = 1; hi = nelt-1;
14         /* lo < hi */
15         /* base[0..lo-1] <= pivot < base[hi+1..nelt-1] */
16         while (lo <= hi) {
17                 while (lo < nelt && base[lo] <= pivot)
18                         lo++;
19                 /* 0 <= lo == nelt || base[lo] > pivot */
20                 while (0 <= hi && pivot < base[hi])
21                         --hi;
22                 /* -1 == hi < nelt || base[hi] <= pivot */
23                 if (lo < hi) {
24                         t = base[lo]; base[lo] = base[hi]; base[hi] = t;
25                 }
26         }
27         /* hi < lo */
28         /* base[0..lo-1] = base[0..hi] <= pivot < base[hi+1..nelt-1]
29          *                                     = base[lo..nelt-1]
30          * 0 < lo = hi+1; lo=nelt possible if all entries identical */
31         /* t = base[0]; */ base[0] = base[lo-1]; base[lo-1] = pivot;
32         /* lo-1 < nelt */
33         while (0 <= hi && base[hi] == pivot)
34                 hi--;
35         /* hi < 0 || (0 < hi && base[hi] < pivot) */
36         if (0 < hi) quicksort(base,hi+1);
37         if (lo < nelt)
38                 quicksort(base+lo,nelt-lo); /* nelt-1 - (hi+1) + 1 */
39 }
```

The `median_to_front` function in Figure 1 is only an approximation of the desired *linear-time median find* algorithm, so this particular implementation of *quicksort* will have poor worse-case behavior. (Can you see why? Since we are primarily concerned with correctness—so the algorithm won't read memory that it's not supposed to and reveal secrets, or write to memory that it's not supposed to and permit unintended program state changes—we are not going to look into this aspect more closely.) We will not cover the "proper" version of this algorithm in class. N.B.: many *quicksort* implementations do not use the lineary-time median find algorithm: even though it has $O(n)$ run time, its constant factors are relatively large; something similar to the `median_to_front` function works quite well in practice (average case complexity), such as randomly picking a pivot element or taking the median of a slightly larger sample.

Figure 2 contains the meat of the *quicksort* code. It implements the variant of the *quicksort* algorithm discussed in class: at each iteration, we choose a pivot, and separate the array into three partitions—at the left, a partition containing elements less than or equal to the pivot; at the right, a partition containing elements greater than the pivot; and in the middle, a partition containing as-yet-unexamined elements. Initially the left and right partitions are empty, and we grow these by scanning into the middle. This is not a "stable" sort (what does this mean?), but is nice in that it requires very little auxilliary storage.

This code contains preconditions and postconditions as C language comments. When I wrote this code, I debugged it by backtracking from desired postconditions to determine the (weakest) precondition, and adding *guards* such as the `if (lo < nelt)` in line 37 in Figure 2, which by establishing `lo < nelt`, ensures that `nelt-lo` $\geq$ `0` holds. This is a precondition for invoking `quicksort`.

3 Security Audits

The tools developed for proving code correct are applied when performing a code security audit/review. For example, in the code fragment in Figure 3, we reason *backwards* from line 13. In order for it to execute correctly, we must ensure the code actually ensured its weakest precondition, that `dptr` and `dptr->read` both have valid values.

Figure 3: Code fragment for security review

```
1   /*
2    * necessary precondition:
3    *  major within range of valid
4    *  indices for devsw
5    */
6   dptr = &devsw[major];
7   /* desired postcondition of this: dptr is valid */
8   ...
9   /* precondition for use of dptr:
10   *  dptr must be valid, and
11   *  dptr->read must be valid
12   */
13  return (*dptr->read)(...);
```

By backtracking through the code, we see that `dptr` is initialized by computing the address of an element of the `devsw` array. Continuing this process, we discover that `major` should be properly range checked. A simple guard statement, such as

```
1   if ((unsigned) major >= sizeof devsw/sizeof devsw[0]) {
2           /* do cleanup */
3           return ENODEV;
4   }
```

would protected this code from exceeding the array bounds. Of course, ensuring that all array entries have valid function pointers is also necessary for the `dptr->read` invocation to operate correectly, e.g., by initializing the `devsw` array so that all entries are valid (use an `unimplemented` function to fill in slots, where `unimplemented` simply returns the appropriate error code), or include an explicit check:

```
1   if (0 == dptr->read) {
2           /* do cleanup */
3           return ENOSYS;
4   }
```