Getting it Right: Testing, Proofs and Program Correctness

Some tough questions:

- How do we know if our program is correct?
- What does correctness mean w.r.t. programs?
- What do “real” software developers do?
- What ought “real” software developers do?
- Can we settle for something less than formal correctness?

In the Real World ...

Industrial practitioners use a variety of techniques:

- LOTS of testing, done in various ways.
- code inspections and structured walkthroughs.
- sometimes formal proofs ...
- rarely automated verification systems ...

Software still goes wrong ... but experience with safety-critical software systems is actually encouraging:

→ We can do a good job if we try hard, act like careful engineers.

Just Testing

Years ago, Dijkstra noted that testing can only ever prove the presence of errors, not the absence of them.

→ This is true, of course, and should give us pause.

→ However, in practice testing is the main way in which we discover errors, and we aren’t going to abandon it!

→ Sometimes, well-tested software turns out to have serious bugs that come to light only years later.

- race conditions show up after hardware upgrade
- software used in new situations
- lesser-used features become more used

Inspections and Walkthroughs

Code inspections and structured walkthroughs\(^a\) have also been very effective in industry for finding errors:

- Old timers can teach newbies tricks of the trade, good habits.
- Forces you to be careful if others will be reading your code.
- Fresh eyes spot mistakes.
- Readable, scrutable code is more likely to be correct. [Unsubstantiated hypothesis.]
- Social act of walkthrough draws out hidden (and sometimes wrong) assumptions into the open.

\(^a\)Recall assgt #2.
What Does “Correct” Mean?

Simply stated, correct means:

“does what it is supposed to”.

i.e., conforms to the specification

```
// Returns the index of the first occurrence of
// soughtVal in A, if it occurs.
// Returns -1 if it doesn’t occur.
public static int linSearch (int A[], int soughtVal) {
    for (int i=0; i<A.length; i++) {
        if (A[i]==soughtVal) {
            return i;
        }
    }
    return -1;
}
```

Proving correctness is a formal argument that a program correctly implements its specification. We’ll see how this is done.

Simple Proofs

Does square f have a bomb?

```
a b c d
 e f g h
 i j k l
 m n o p
 q r s t
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
</tr>
<tr>
<td>m</td>
<td>n</td>
<td>o</td>
<td>p</td>
</tr>
<tr>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
</tr>
</tbody>
</table>

bomb means bomb in this square
N means no bomb in this square
but N bombs among my neighbours
– means no bomb here or any neighbour

*Examples from Diane Horton

Proof by Contradiction

Basic idea:

→ Want to prove Q is true.

→ Assume Q is false and show that this leads to an impossible situation.

Another Example

If j has a bomb, what do we know about t?

```
a b c d
 e f g h
 2 -- --
 i j k l
 m n o p
 q r s t
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
<tr>
<td>2</td>
<td>--</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
</tr>
<tr>
<td>m</td>
<td>n</td>
<td>o</td>
<td>p</td>
</tr>
<tr>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
</tr>
</tbody>
</table>

Conditional Proof

→ Want to prove $A \Rightarrow B$

→ Assume $A$ is true.
Try to prove $B$ somehow.

Yet Another Example

Does $n$ have a bomb?

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>n</td>
<td>o</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td></td>
</tr>
</tbody>
</table>

Proof By Cases

→ Prove $Q$ is true.

→ Proof:
The following is true: $C_1 \lor C_2 \lor \cdots \lor C_N$
Prove $C_1 \Rightarrow Q$
Prove $C_2 \Rightarrow Q$
\vdots
Prove $C_N \Rightarrow Q$ 
Therefore $Q$ is true.

• Have to choose the cases carefully; make sure you get them all.
• Can sometimes combine several cases into one.

Proving Properties of Integers

• Some properties can be proved directly:

\[(x > 2) \Rightarrow (x^2 - 3x + 2 > 0)\]

In fact this is true for all real numbers too!

• What about this:

\[
\sum_{i=0}^{N} i = \frac{N \times (N+1)}{2}
\]
Induction

1. 0 is an integer.
2. if \(N\) is an integer, then so is \(N + 1\).
3. if \(N\) is an integer, then so is \(N - 1\).

Fact: any integer can be derived by repeated application of these three rules.

e.g., \(5 = 0 + 1 + 1 + 1 + 1 + 1\)

So we can characterize any integer as being 0, \(N + 1\) or \(N - 1\) where \(N\) is known to be an integer.

Example of Induction

Prove:

\[
\sum_{i=0}^{N} i = \frac{N \times (N + 1)}{2}
\]

Part 1: Basis case.
Prove \(P\) holds for \(N = 0\).

Part 2: Induction case.
Assume \(P\) holds for \(N\).
Now prove it also holds for \(N + 1\).
Correctness of Programs

- OK, so we have seen a cheesy game and some simple mathematical formulas.
  - What about programs?
- Well, we prove a program to be correct if we can show that the program correctly implements its specification (what it is supposed to do).
- We’ll consider only programs with declarations, assignment, ifs, and loops.

Program Specification

- A specification of what a program is supposed to do includes:
  - pre-condition — states restrictions on use
  - post-condition — states what program does
- Good practice: State the specification of all non-trivial methods informally in a comment just before it. Be as precise as you can.
  - e.g., linear search
  - binary search
  - sqrt
  - sort
- Take CS400 if you are interested in this topic.

Pre- and Post-Conditions

Pre-Conditions:

- State limitations on how program/method may be used.
  - e.g., positive input, list sorted or not
- Only state what must be true.
  - make as logically weak as possible while mentioning all required assumptions and limitations.

Post-Conditions:

- State what effects the program has on the variables/parameters.
- Say as much as you can:
  - Make as logically strong as possible.
- May need to refer to before- and after-values of variables.

Correctness: Hoare Triples

\[ \{ P \} \ S \{ R \} \]

- P and R are logical formulas. P is the pre-condition and R is the post-condition.
- S is a program fragment (in Java, say).

\{ P \} \ S \{ R \} is an assertion of fact.

It says that:

- if you know that P is true, then
- if you run program S and it eventually halts, then you are guaranteed that R will be true.

- e.g., \{ x>0 \} \ x=x+1; \{ x>1 \}
- e.g., \{ true \} \ z=y; if (y<x) \ z=x; \{ z=max(x,y) \}

- How do you figure out what the right P and R should be?
  [Ans: The same way you get to Carnegie Hall.]
- How do you prove \{ P \} \ S \{ R \} holds?
  [Ans: We’ll skim it. See CS280 and CS400 for details.]
Partial and Total Correctness

- If decide your specification should be pre-condition \( P \) and post-condition \( R \), and if you manage to prove \( \{ P \} S \{ R \} \) holds, you have proved partial correctness.

- To establish total correctness, you must also prove that the program will eventually halt or terminate.
  - Obviously, any program without loops or recursion\(^a\) will terminate.
  - To prove that a loop terminates, we usually try to find some value that gets smaller and smaller\(^b\) until it reaches zero or some known stopping state.
  - For a recursive function, we need an argument that any recursive call will eventually reach a basis case.

- We won’t consider termination in much detail.

\(^a\)including “mutual recursion”: \( A \) calls \( B \) calls ... calls \( A \)
\(^b\)smaller, not smaller-or-stays-same

How do We Prove Programs Correct?

1. assignment statements:
   → straightforward logic

2. if statements:
   → logic plus argument by cases

3. loops (for, while):
   → This is harder! We use a kind of induction.
   → Key idea: loop invariant.
   → a loop invariant is a logical statement about the vars/params in a loop that is true at the beginning of each loop iteration. Conceptually, it represents the state of the computation so far.
   → loop body may break invariant temporarily as long as it is fixed by the end of the current iteration.

Structure of a Correctness Argument for a Loop

```java
// Pre-condition P
// initialization code
// Invariant: I
while (B) {
    S  // loop body
}
// Post-condition Q
```

To prove a loop correct, prove:

1. If \( P \) is true, then executing initialization code will ensure \( I \) is true.

2. If \( B \) is true, then executing loop body \( S \) will preserve truth of \( I \)
   \( i.e., \{ I \land B \} S \{ I \} \)

3. If \( B \) is false, then must be able to establish \( Q \)
   \( i.e., (I \land \neg B) \Rightarrow Q \)

4. Argument that loop will terminate if \( P \) is true.

An Example

Overall goal:

→ Store in \( x \) the sum of \( A[0], \ldots, A[N-1] \)

```java
// Precondition: \( N = A.length \)
int k = 0;
int x = 0;
// Invariant:
while (true) {
    }
// Post-condition:
```