Lecture 22:
Correctness in Practice

Aims:

- To discuss what is actually happening in practice.

22.1. Developing Correct Programs

- So far, we've been looking at proving the correctness of an *already-written* program.

  Now, we will briefly look at how construction of a program and its proof can proceed hand-in-hand, so that we guarantee that a correct program is developed.

  This is *non-examinable*.

- Suppose we want a program C, such that $(\!| 0 \leq n |\!)\, C\, (\!| x^2 \leq n \wedge n < (x+1)^2 |\!)$ (the largest integer that doesn't exceed the square root of x).

  We probably need some initialisation commands and then a loop.

$(\!| 0 \leq n |\!)$
???
$(\!| Inv |\!)$
while B
{ $(\!| Inv \wedge B |\!)$
???
$(\!| Inv |\!)$
}
$(\!| Inv \wedge \neg B |\!)$While
$(\!| x^2 \leq n \wedge n < (x+1)^2 |\!)$

Decide on an invariant and a loop-test. The postcondition is a conjunction. Maybe the loop-test is the negation of one of the conjuncts, and the invariant is the other conjunct. So let

$$B =_{\text{def}} n \not< (x+1)^2$$

$$Inv =_{\text{def}} x^2 \leq n$$

$(\!| 0 \leq n |\!)$
???
$(\!| x^2 \leq n |\!)$
while $(x+1)^2 \leq n$
{ $(\!| x^2 \leq n \wedge (x+1)^2 \leq n |\!)$
???
$(\!| x^2 \leq n |\!)$
}
$(\!| x^2 \leq n \wedge n < (x+1)^2 |\!)$While
$(\!| x^2 \leq n \wedge n < (x+1)^2 |\!)$Consequence

Fill in the loop body. The obvious way to preserve the invariant is to increment x in the loop body.

$(\!| 0 \leq n |\!)$
???
$(\!| x^2 \leq n |\!)$
while $(x+1)^2 \leq n$
{ $(\!| x^2 \leq n \wedge (x+1)^2 \leq n |\!)$Invariant and loop test
$(\!| (x+1)^2 \leq n |\!)$Consequence
$x := x + 1;$
$(\!| x^2 \leq n |\!)$Assignment
}
$(\!| x^2 \leq n \wedge n < (x+1)^2 |\!)$While
$(\!| x^2 \leq n \wedge n < (x+1)^2 |\!)$Consequence

Fill in the initialisation. The obvious way to establish the invariant in the first place

is to set x to 0.

$(\!| 0 \leq n |\!)$
$(\!| 0^2 \leq n |\!)$Consequence
$x := 0;$
$(\!| x^2 \leq n |\!)$Assignment
while $(x+1)^2 \leq n$
{ $(\!| x^2 \leq n \wedge (x+1)^2 \leq n |\!)$Invariant and loop test
$(\!| (x+1)^2 \leq n |\!)$Consequence
$x := x + 1;$
$(\!| x^2 \leq n |\!)$Assignment
}
$(\!| x^2 \leq n \wedge n < (x+1)^2 |\!)$While
$(\!| x^2 \leq n \wedge n < (x+1)^2 |\!)$Consequence

I didn't show the proofs needed for Consequence, but they should also be given. Ideally, we should also have considered a variant to establish total correctness.

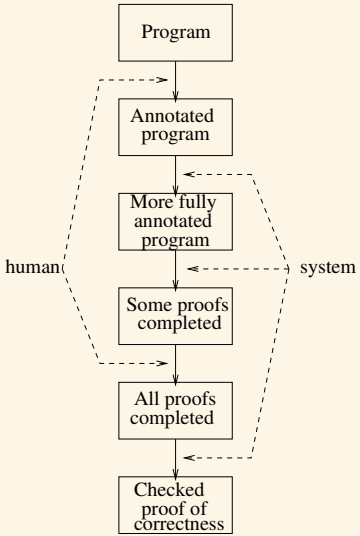
22.2. Program Correctness in Practice

- So how much program proof actually goes on in practice? Obviously, not much. But greater rigour is needed when you are developing safety-critical (e.g. life-endangering) systems. So let's look at some of the things that people do.

22.2.1. Automatic Program Verification

- For every correct program, there is a proof of its correctness.
- For even short programs, proofs are often long, fiddly and boring.
- Can we automate proofs of correctness?
  - We can write a program which, given a proof, can check the proof (a *proof checker*).
  - But the problem of writing a program to *find* proofs for arbitrary programs is non-computable.
  - However, we can write a program that assists the user by proving many useful assertions and checking the work of the user.
- A typical set-up is shown in the following diagram. The user writes some of the annotations. The system does as many proofs as it can. The user is then left to do the remaining proofs. The system then checks the proof.

  In practice, the process is iterative.



22.2.2. Informal Proofs

- Another possibility in practice is for the human to do proofs but to do only very high-level, informal proofs.

  For example, the human might do the following:

  - Come up with an invariant.
  - Prove informally that the initialisation steps make the invariant true.
  - Prove informally that, if the invariant is true at the start of the loop body, then it will be true at the end of the loop body.

22.3. Adding Assertions to Programs

- Programming languages such as Eiffel and (recent releases of) Java allow the programmer to include assertions in the program.
- An *assertion* is a statement that allows you to record and test assumptions about your program.
- In Java, you write:

```
assert ⟨Boolean-expression⟩ ;
```

  or, more helpfully,

```
assert ⟨Boolean-expression⟩ : ⟨error string⟩ ;
```

- When encountered during execution, the Boolean expression is evaluated. An error is thrown if the expression does not evaluate to **true**. The error string, if included, is displayed.
- The claimed advantages are

> "By verifying that the boolean expression is indeed true, the assertion confirms your assumptions about the behavior of your program, increasing your confidence that the program is free of errors.
>
> "Experience has shown that writing assertions while programming is one of the quickest and most effective ways to detect and correct bugs. As an added benefit, assertions serve to document the inner workings of your program, enhancing maintainability." [jav03].

- Ideal candidates are: preconditions, postconditions and loop invariants. In other words, even if you are not going to all the trouble of proving the correctness of your program, a little thought about preconditions, postconditions and loop invariants can be tremendously useful. Include them as assertions in your program. Put the precondition at the start. Put the postcondition at the end. Put the loop invariant (a) just prior to the **while** command, (b) at the end of the loop body; and (c) after the **while** loop.

- E.g.

```
private static int intDiv(int x, int y)
{    assert x >= 0 && y > 0;
     int q = 0;
     int r = x;
     assert x == q * y + r && 0 <= r;
     while (r >= y)
     {    q++;
          r = r - y;
          assert x == q * y + r && 0 <= r;
     }
     assert x == q * y + r && 0 <= r;
     assert x == q * y + r && 0 <= r &&
                                   r < y;
     return q;
}
```

(Obviously, in this particular example, there wasn't much point including both the invariant after the loop and the postcondition, since they were so similar. Just the postcondition would suffice here. But, in general, you might include both.)

- In case this (new) feature of Java interests you, here are some more notes about using

it.

- If your program contains assertions, then you must compile your program in a special way. This will not work:

  ```
  javac  MyProgram.java
  ```

  Instead, you must use:

  ```
  javac  -source  1.5  MyProgram.java
  ```

  or, if that doesn't work, try:

  ```
  javac  -source  1.4  MyProgram.java
  ```

  (This is why you should **not** include assertions in programs you are writing for other lecturers in this year's modules. When they are grading your program, they may not realise that they need to compile it in a special way.)

- Assertion checking can be enabled or disabled (so that it incurs no performance penalty once the program is finished). By default, it is disabled. So if you run your program like this:

  ```
  java  MyProgram
  ```

  assertion checking is disabled. More verbosely, the following gives the same effect:

  ```
  java  -da  MyProgram
  ```

  To *enable* assertion checking, run your program like this:

  ```
  java  -ea  MyProgram
  ```

  (Of course, you'll need to write a buggy program —one whose assertions are not true— to see anything happen. But that shouldn't be a problem ;)

  - For further information about Java assertions and where and how to use them, read [jav03].

- A final remark is that assertion-checking was a late addition to the Java language. The Eiffel language is an object-oriented language in which, by contrast, assertion-checking was included from the outset. The facilities for assertion-checking in Eiffel are therefore much better-developed. Great care has gone into thinking about the way that assertion-checking interacts with object-oriented features such as inheritance, for example.

  In Eiffel circles, the use of assertions has been elevated to an approach to the design and implementation of programs and goes under the name of Design-By-Contract. If you are interested, the classic textbook is [Mey97].

22.4. Other Lines of Work

- Verification of hardware
- More sophisticated languages for specification of systems
  - Hoare triples are suitable only for simple programs
  - To specify whole systems of various kinds, use Z, B, VDM, Larch, CCS, CSP, modal logics, etc.
- Formal specification refinement.

Acknowledgements

I used material from [Kal90] and [jav03].

Clip Art (of head with bomb) licensed from the Clip Art Gallery on DiscoverySchool.com.

References

[jav03] java.sun.com. Programming with assertions. http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html, Accessed 02/10/03.

[Kal90] A. Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice Hall, 1990.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.