Lecture 10:
Introduction to Correctness

Aims:

- To look at the different types of errors that programs can contain;
- To look at how we might detect each of these errors;
- To look at the difficulty of detecting run-time errors using testing.

10.1 Bugs

- Correctness. Informally, an algorithm or program is correct if it is free from error. In particular, it must map the parameters to the return values in accordance with the problem specification. (We'll give a more formal definition later in the module, where we will distinguish partial correctness and total correctness).
- Three types of program error.

  Compile-time errors. These are errors that are detected by the compiler when it is attempting to translate from the source language to the target language. Hence, these are errors that are detected *in advance of execution of the program*. They arise when the programmer, through ignorance or carelessness, fails to follow the syntactic and static semantic rules of the programming language.

  - Syntax errors. Examples in Java are failing to match up parentheses or braces, omitting semi-colons, mistyping the word **public** as pubic, and so on.
  - Static semantics errors. Examples in Java are failing to initialise local variables, certain mismatched type errors (such as assigning a value that is of type **boolean** into a variable that is of type **int**), declaring in the header of a method that it returns a **boolean** but in the body of the method returning an **int** or returning nothing, and so on.

  It is not uncommon the first time you compile a program to get over 100 compiler-errors. As you have probably realised, one mistake early in the program can wrong-foot the compiler causing it to find fault with many subsequent lines of the program.

  There are two simple pieces of advice (which I know you won't follow) if you want to avoid being faced with overwhelming numbers of error messages from the compiler. Firstly, compile at frequent intervals, rather than typing in a vast program and only compiling once the whole thing has been typed in. Secondly, just before compiling, read through your code: you will easily spot the majority of the errors.

  Run-time errors (*dynamic semantics errors*). Run-time errors arise during execution of the program. They cannot be detected at compile-time. When detected during execution, they cause the program to crash. Examples in Java include division of an **int** by zero and index out of bounds errors. An index out of bounds error arises when your program attempts to access a non-existent cell of a data structure. For example, if a is a Java array of length 10, then the cells are $a[0] \ldots a[9]$. If your program attempts to access, e.g., $a[-1]$ or $a[10]$, this gives rise to an index out of bounds error. Run-time errors can also arise when your program is attempting to communicate with some external device. For example, if your program is trying to write data to a file on disk, the file may not exist, the program may not have write-permission for the file, the disk may be full, or the disk drive may have crashed. Any of these will mean that your program's attempt to write data will fail. Unless your program includes special checks for these situations and code to recover from them, then, if they arise, your program will crash.

  Since these errors cannot be detected in advance by the compiler, the conventional way of discovering them is through program testing, which we discuss below.

  Logic errors. Logic errors are errors which do not result in error messages from the compiler or the run-time environment. We are not notified of them at all. They are conceptual errors: our algorithm or program does not match the problem specification. The actual results are not the same as the expected results. For example, consider this problem specification:

  Problem 10.1.
  Parameters: *An integer, n.*
  Returns: 2^n

  An algorithm or program that is supposed to solve this problem contains logic errors if, for example, it apparently fails to terminate; or it always seems to give answers that are off by one, e.g. for $n = 3$, it returns the answer 7; or it gets the right answers for all non-negative n, but not for negative n, e.g. for $n = -3$ it returns 8 instead of 0.125.

  Again the conventional way of discovering these errors is through program testing, discussed below.

  Of course, if our programming language is interpreted instead of compiled, then syntax errors and static semantics errors will only be discovered at run-time, like dynamic semantics errors. The ability to find errors in advance of execution is one of the advantages of compilers.

- We also cannot rule out the possibility of *hardware errors* or *systems software errors* (e.g. errors in the compiler or interpreter). Thankfully, these days hardware errors are a rarity. Errors in systems software, however, are a genuine problem.
- The dangers of releasing buggy software include loss of money, livelihood, life, etc. and damage to health, environment, etc.
- We're going to discuss testing below. But there's an obvious weakness with testing. Our test cases may simply fail to create the circumstances that give rise to some of the more subtle errors. They will go undetected, and may only reveal themselves when the program is in use. Even programs which have executed successfully for many years may still contain bugs that would cause them to fail in certain circumstances.
- There's an obvious advantage in getting the compiler to do as much error-checking as possible. Errors will then be detected in advance of execution.
- So a natural question arises: how much can the compiler do? For example, can we get the compiler to detect more of the dynamic semantics
- Let's see how easy or hard it would be for the compiler to detect possible division by zero errors.

- Which of these would ordinarily cause run-time errors and which could a compiler easily detect?

```
public int method(int x)
{   return 3 / 0;
}
```

```
public int method(int x)
{   return 3 / x;
}
```

```
public int method(int x)
{   if (x > 0)
    {   return 3 / x;
    }
    else
    {   return -1;
    }
}
```

```
public int method(int x)
{  return 3 / (x - x);
}
```

```
public int method(int x)
{   int y = x - x;
    return 3 / y;
}
```

```
public int method(int x, int y)
{   for (int i = 0; i < y; i++)
    {   x += -2;
    }
    return 3 / x;
}
```

- Can we write a compiler to work out which of these could cause run-time errors? Some of the faulty ones are easy to spot syntactically. Others could be found from running the program on a few well-chosen test cases (although this somewhat defeats the object of asking a compiler to find the errors). Others would require the compiler to *reason* quite extensively about operations on integers and about control flow.
- In fact, recall from the introductory lecture that some problems are non-computable (unsolvable). Well, here is one of them.

Problem 10.2.
Parameters: *A program, P*
Returns: *YES if P will ever perform a division by zero, NO otherwise.*

In other words, we cannot write a program that can take in any other program as input and infallibly tell us whether the input program will, when executed on certain inputs, attempt a division by zero.

The justification for this bold statement is not because I've given quite a few examples, and some of them look quite hard.

The justification for this bold statement comes in the form of a *proof* that this problem is non-computable. However, we won't prove it here. A proper discussion of non-computable problems comes later in the module. There, we'll discover that lots of problems that involve checking programs are non-computable.

This result seems to drive us back towards a reliance on testing. Let's look at testing in detail to see its limitations.

10.2 Testing

- You need to run your program on test data and compare the actual answers with the expected answers.
  - Rarely is it practical to test exhaustively. How many different actual parameter values do these have?

```
public boolean method(int x)
```

```
public boolean method(int x, int y)
```

  - Instead, design a test-suite with good *coverage*: favour quality over quantity
- In black-box testing,
  - Design test-suite on the basis of the problem specification.
  - Use *equivalence partitioning*: choose test cases that are representative of collections of data.
    - normal, boundary and exceptional values

What test values would you use for the following:

Problem 10.3.
Parameters: *A list of integers, L.*
Returns: *The sum of the integers in L.*

- In random testing,
  - Test-suite is chosen randomly.
  - Allows large test suites.
  - Avoids subconsciously under-testing parts you're not sure of
- In glass-box testing (also called white-box testing),
  - The test-suite is chosen based on analysis of the control flow of the program itself
  - Choose test data that exercises *all commands* in the program.
  - If possible, choose test data that exercises *all paths* through the program.

- E.g. Devise values of x that will exercise all commands and all paths in the following:

```
if (x = 0)
{   System.out.println(``A'');
}
else
{   System.out.println(``B'');
}
if (y = 0)
{   System.out.println(``C'');
}
else
{   System.out.println(``D'');
}
```

- Note the way in the last example that the number of paths starts to multiply up.

  All-paths testing is quickly impractical. Instead, you have to test some representative subset of the paths.

10.2.1 Testing is Inconclusive

- In any discussion of testing, remember

  Testing cannot demonstrate the absence of errors, only their presence.

- Even if you've done a very great amount of testing with no apparent errors, you still cannot conclude that your program is error-free. Perhaps there is some bizarre error that occurs for the numerous inputs that you haven't tested.

  Always assume that there remains at least one more error.

- So, if none of your tests has yet found an error, then the best thing to conclude is that your testing hasn't yet been demanding enough to find that last remaining error that inevitably lurks in the program somewhere! So you need some better test data in order to find that error.

  A successful test is one that finds an error.

- Since we mentioned a non-computable problem earlier, let's mention another one here.

Problem 10.4.
Parameters: *A program, P*
Returns: *An adequate set of test data for program P.*

This is non-computable. In other words, no algorithm exists that can take in an arbitrary program and return an adequate set of test data. 'Adequate' here means that if the program produces no errors on the test data then it will produce no errors on any input data.

10.3 Proving Correctness

- As we have seen, testing is inconclusive (except in those rare occasions when you can test exhaustively on all possible inputs). For safety-critical software that can endanger health and life, bugs are disastrous. It would be nice to have stronger guarantees than testing can give. It would be nice to *prove* that the program is correct. In other words, to prove that, for all legal inputs, it terminates and produces the right answer.
- Any correct algorithm can be proved to be correct.
- Of course, there is a risk of a mistake in the proof! But at least a proof will increase confidence in the program's correctness.
- And let's mention a third non-computable problem! You might hope that we can write a program to do the proofs for us. Well, sorry to dash this hope: you cannot fully automate correctness proofs.

Problem 10.5.
Parameters: *A problem specification, PS, and a program P.*
Returns: *A proof that P solves PS (if it does solve it), otherwise* ***fail.***

We cannot write a program that can take in an arbitrary problem specification and program and output a correctness proof, if the program meets the specification.

Just because we cannot (fully) automate the problem doesn't invalidate the usefulness of doing correctness proofs. It just means we'll have to do more of the work ourselves, manually. (It is possible to partly automate the task, and this can take a lot of the pain out of doing it.)

- The next part of this module shows you how to do correctness proofs. Before we can look at the details of the proofs, we have to take a digression to recap our understanding of logic.

Acknowledgements

I based some of the later parts of the lecture on material from [Har92] and [GL82].

References

[GL82] L. Goldschlager and A. Lister. *Computer Science: A Modern Introduction*. Prentice-Hall, 1982.

[Har92] D. Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley, 2nd edition, 1992.