Software Engineering II

Computer Science Tripos Part 1a (50% Option)

Lawrence C Paulson
Computer Laboratory
University of Cambridge

lcp@cl.cam.ac.uk

Contents

Engineering, as it is properly understood, is not possible for software. An engineer can design a bridge, confident that it will meet its requirements when built (most of the time!). Our theory and tools are not yet good enough to let us build software to this standard of reliability.

This course has less ambitious goals. It introduces methods for designing software systematically. It also introduces the emerging theory that may one day make *Software Engineering* a reality.

No textbooks follow this course at all closely. *Fundamentals of Software Engineering* [7] is similar in spirit and has more content than much larger books. *ML for the Working Programmer* [15] covers structural induction, used in the last lecture. Past examination questions on Software Engineering II from 1998 onward are relevant. For the last lecture, try 1993 Paper 2 question 8. There are many exercises below that you can use during revision.

Joe Hurd reported several typographic errors in these notes. Please tell me if you find any more; I'll acknowledge all corrections.

Acknowledgement. Ross Anderson supplied a book [14] that became the basis for Lect. 3.

Slide 101

Refinement, or Top-Down Design

Example: **Printing a Table of Squares**

```
fun squares() =
  while not (finished())
  do (getInputs();
      printTable())
```

Refinement is top-down design. The main task is expressed as a simple routine that delegates its work to subroutines yet to be defined. You can declare the yet uncoded subroutines as *stubs*: dummy functions that just print their name and exit. (Some systems generate stubs automatically.) With stubs, the program is always executable, even if it doesn't accomplish anything. Coding the lower-level routines adds functionality to the program, so that eventually it can do useful work.

Our task below is to design a program to print tables of squares. In each table, the range of values to be squared is specified by giving the lower bound, upper bound, and increment (or delta). Even this trivial example will highlight many points.

Slide 102

Refining A Low-Level Routine

```
fun finished() =
    (promptForInput();
     testInputStream());        return true if no more input
```

Refines to . . .

```
fun finished() =
    (print "Lower  Delta  Upper? ";
     TextIO.endOfStream TextIO.stdIn);
```

Remark. This example is coded in ML. But ML functions must be declared before they are called, while refinement means designing functions in reverse order. The declarations shown on the slides must be re-arranged to yield a valid ML program. Often, the higher-level program will need modification anyway due to insights learned when coding lower levels.

Function `finished` has the task of determining whether more tables have to be printed, i.e. whether more input remains. It uses components of `TextIO`, an ML Basis Library module for text input/output.

Do not worry about learning advanced features of ML such as these. They are used merely to complete the example.

Slide 103

Coding some Low-Level Routines

```
fun getInputs() =
  let val line = TextIO.inputLine TextIO.stdIn
  in
     case map Real.fromString (wordsOf line) of
          [SOME lower, SOME delta, SOME upper]
             => SOME(lower,delta,upper)
        | _  => (complain();  NONE)
  end;

val wordsOf = String.tokens Char.isSpace;

fun complain _ = print "Bad input line\n";
```

Function `getInputs` reads an input line and tries to decode it as three real numbers. It uses the ML Basis Library type `option` to indicate whether the input is erroneous or valid.

```
datatype 'a option = NONE | SOME of 'a;
```

The call to `map` returns a list whose elements have type `real option`. A well-formed input is mapped to `SOME` *x*, where *x* is the corresponding real number, while an ill-formed input is mapped to `NONE`.

If the input line consists of three valid real numbers, then the `case` expression returns the whole triple, again using `SOME` to indicate that it is valid. Otherwise, it prints an error message and returns `NONE`. The `case` pattern matches three-element lists whose elements are all valid.

Error reporting belongs in `getInputs`, not in the main loop. A well-structured program delegates each task to the most appropriate routine. Sometimes, the best division of labour is not obvious.

Function `complain` is the user interface. A better one would describe what was wrong with the input line. This task might be combined with the scanning of the numbers. Function `getInputs` would have to be restructured, but its callers would not be affected.

Top-down design produces many trivial functions like `wordsOf` and `complain`. Usually, such functions make the program more readable, so they should be kept separate and not integrated with their caller. Any optimizing compiler will eliminate trivial function calls during code generation: they will not slow down execution.

Slide 104

Generating the Table

```
fun printTable(lower,delta,upper) =
  let val xr = ref lower
  in
     while !xr <= upper do
       (printLine (!xr);
        xr := !xr + delta)
  end;

fun printLine x =
  print (Real.toString x ^ "    " ^
         Real.toString (x*x) ^ "\n")
```

A simple `while` loop generates numbers from the lower bound to the upper. Take some time to convince yourself that this code is correct. Look especially at its termination condition, which compares the current value with the upper bound.

We delegate the job of actually printing the squares to function `printLine`.

Notice that the interface to `printTable` has changed. Before, we hadn't thought about how it should get its data. Coding it has suggested that it should take the parameters as arguments, rather than through global variables.

Slide 105

Revised Main Program

```
fun squares() =
  while not (finished())
  do
     case getInputs() of
          SOME triple => printTable triple
        | NONE        => ();
```

We now revise the declaration of `squares` to accommodate how `getInputs` and `printTable` were finally coded. The former function returns a triple of items parsed from the input (prefixed by `SOME`) if it was valid, and the constructor `NONE` otherwise. Since `getInputs` will have already printed an error message, `squares` simply ignores the bad data.

Refinement has given us a clear program that has a natural structure.

Exercise 1 Recode `getInputs` to use exceptions instead of `SOME` and `NONE` to report bad data. Which version do you prefer?

Slide 106

Results??

```
squares();
> Lower  Delta  Upper? 0  0.1  1.5

> 0.0   0.0
> 0.1   0.01
> 0.2   0.04
and so forth . . .
> 1.2   1.44
> 1.3   1.69
> 1.4   1.96        WHAT ABOUT 1.5??
```

This sample run tells us that the program is wrong. We requested a table of squares finishing at 1.5, but this last value did not appear. The reason is that `!xr` was minutely greater than 1.5. The accumulated rounding errors were on the order of 10^{-17}.

Thanks to the structuring of the program, we can instantly see that the fault lies within function `printTable`. (Had you convinced yourself of its correctness, as you were asked to do?) The termination test uses $\leq$, and $<$ would plainly be wrong. But with rounding errors, there is little useful difference between $\leq$ and $<$ on floating-point numbers.

Other problems can be noticed. For example, `printTable` runs forever if the increment is zero or negative. These problems would have come to light before writing a single line of code if we had taken the time to specify the task formally (Lect. 5).

Slide 107

A Better Version?

```
fun printTable(lower,delta,upper) =
  if delta = 0.0 then print "Delta = 0.0!\n"
  else
  let val xr = ref lower
      and ir = ref (iters(lower,delta,upper))
  in
     while !ir > 0 do
       (printLine (!xr);
        xr := !xr + delta;  ir := !ir-1)
  end;
```

The simplest, and perhaps the best, approach is to modify `printTable`'s termination test. Instead of testing against the upper bound u, it could test against $u + \delta/2$; in other words, add half the increment to the upper bound to guard against rounding errors.

The solution shown above is more radical, and suitable for machines (such as the IBM 360 series) in which floating-point rounding errors are especially severe. It wholly abandons floating-point in the termination test. Instead, it computes the necessary number of iterations beforehand:

```
fun iters(lower,delta,upper) =
  1 + Real.round ((upper-lower)/delta);
```

This computation is less likely to go wrong. It performs just three floating-point operations, avoiding the previous version's accumulation of rounding errors. Rounding the quotient will yield the nearest integer.

Note the addition of one to the quotient. The need for an extra iteration is easily missed. Such *off by one* errors are common in programs.

The explicit use of division makes the requirement $\delta \neq 0$ obvious. This version of `printTable` tests for it, printing an error message instead of failing with division by zero.

Not checked is whether the conversion to integer (by `Real.round`) will cause an overflow. Robust code should test for every potential error. The June 1996 explosion of the Ariane 5 rocket was caused by a similar error: a conversion from a 64-bit integer to a 16-bit integer. Ironically, there was no need for the faulty code to be running at the time of the failure.

Slide 108

All OK Now?

```
printTable(˜0.4, 0.1, 0.4);
> ˜0.4   0.16
> ˜0.3   0.09
> ˜0.2   0.04
> ˜0.1   0.01
> ˜2.77555756156E˜17   7.70371977755E˜34
> 0.1   0.01
> 0.2   0.04
> 0.3   0.09
> 0.4   0.16
```

A nice thing about declaring separate routines (especially with an interactive language like ML) is that they can be tested separately. Trying out the new `printTable`, we see that it still is not perfect. There is an entry for -2.78×10^{-17}, but none for zero.

Is this error just cosmetic? It depends on precisely what is to be done with the table, but most people would agree that this output is unacceptable. It does, at least, warn us of the errors that are always present in floating point.

The simplest fix is to change function `printLine`. The floating-point quantities should be displayed to a fixed precision, here one decimal place for x and two for x^2. Both -2.78×10^{-17} and its square would then be rounded to zero.

The precision depends upon δ, so the interface to `printLine` must be changed, and our trivial program becomes surprisingly complicated.

The suggested fix could be criticised as treating the symptom rather than the cause. For printing tables, I think it is all right, but in critical applications one must work to reveal errors and not to hide them.

Exercise 2 The Standard ML Basis Library is on the World Wide Web at `http://www.dina.kvl.dk/˜sestoft/sml/sml-std-basis.html`. Find out how to scan real numbers from strings and how to display them to a fixed precision. (Knowing how to read library documentation is a useful Software Engineering skill: it stops you from reinventing the wheel.)

Exercise 3 Change `printLine` using the information gathered in the previous exercise, fixing the program as suggested above. Rather than making the precision depend upon δ, you might choose a fixed precision and restrict the range of δ accordingly.

Slide 109

Some Comments on the Design

Refinement works for *data structures*, too

Can the new `printTable` replace the other?

Incompatibilities:

- negative increments
- inexact upper bounds

Need for a precise *specification*

Inherent dangers of floating-point arithmetic

Top-down refinement lets us design program components one at a time. The final result is a program whose structure reflects the design process. If the division of tasks into subtasks is done sensibly, then each program unit will have a well-defined role. Faults in the program can often be isolated to a single program unit.

Of course, things will not always be this nice. Some faults may concern the overall structure of the program. Modifying a program unit is risky: new faults are often introduced. Even when they are not, some users may actually have depended upon the faulty behaviour and will not want to see it corrected.

There are many visible differences between the two versions of `printTable`. Consider this example:

```
printTable(0.0, 0.1, 1.09);
```

The first version's table will end at 1.0, while (because of its rounding action) the second version's table will end at 1.1. Here is another example:

```
printTable(0.0, ~0.1, ~1.0);
```

The first version will run forever, while the second version table will print a table from 0.0 to -1.0, decreasing in steps of 0.1.

These differences exist because we never specified the desired behaviour of `printTable` precisely. Lecture 5 will briefly introduce formal specifications.

Exercise 4 Precisely specify the valid inputs to `printTable`. In this, use your judgement to decide what would constitute a sensible table and what is probably an error. Your specification should suffice to resolve all the points raised above.

Slide 110

Refinement of Data Structures

```
datatype client = Public  of publicClient
                | Private of privateClient;

datatype publicClient = LocalAuthority
                      | GovtAgency;

datatype privateClient = Individual
                       | SmallBusiness
                       | Corporation;
```

Data structures can also be designed using top-down refinement. In this context, a *stub* is a dummy type, as shown on the slide. The principles are the same. At all times, we have a working (if incomplete) data structure. The finished data structure will reflect the design process. Errors will normally be localized to one part.

Data and control structures can be refined together. Typically, there are different routines to handle different forms of data, so a new routine and the corresponding part of the data structure can be declared at the same time.

Always recall the distinction between abstract services and the low-level data structure used to provide them. Use objects or modules (if your programming language has them) to hide internal details. Other parts of the program should refer only to the high-level services provided by your code. (This was discussed in the course *Foundations for Computer Science*.) For example, a dictionary should support the operations *lookup* and *update*, while hiding the underlying arrays, trees, lists, etc.

Slide 201

The March of Programming Languages

Fortran (1958)

```
  I = 1
1 IF (A(I)-X) 2,3,2
2 I = I+1
  GO TO 1
3 CONTINUE
```

Pascal (1971)

```
i := 1;
while A[i] <> x do i := i+1
```

Both pieces of code have the same purpose. Each examines the elements of array *A*, searching for the first one that equals *x*. Each returns the position of that element in *i*. Fortran's use of branching and labels renders the code almost unreadable, compared with the Pascal code. The `while` statement neatly encapsulates a loop body and termination test, with a single point of exit. The boolean expression is called the *guard*. The loop body is executed only if the guard is true; otherwise, the loop terminates.

(I should like to include code written in a modern programming language, but C and Java have regressed. For example, they use the equals symbol (=) to stand for updating a variable, and they rely excessively on `break` for specifying control flow.)

During the 1970s, improvements in programming languages and the phasing out of assembly code yielded dramatic productivity increases. Researchers continue to seek better languages and other tools, but further gains are likely to be more modest.

A good programmer must understand loop design. A program's efficiency is often determined by its inner loop. Confusing loop structures make for faulty code that is hard to understand, and therefore, hard to correct.

Note. This lecture uses Pascal rather than ML. The latter's need for the `!` operator tends to make programs obscure.

Slide 202

Understanding a `while` Loop

```
Invariant holds here

    while Guard do
        Invariant holds here
        begin
        make_Progress;
        restore_Invariant
        end

Invariant & ¬Guard hold here
```

Every well-designed loop must do two things.

1. It must *make progress*, to ensure that eventually it will terminate.

2. It must maintain an *invariant*, which is an assertion describing the relationships that may hold among the variables changed in the loop. The invariant must hold at each exit point. (With `while`, this is also the start of the loop body.)

The loop body might be executed any number of times, but it will make the invariant hold after every iteration. The invariant will hold when the loop terminates. The termination condition (for `while`, the negation of the guard) will hold too. These two facts must be strong enough to guarantee whatever state of affairs the loop is intended to establish.

Many other looping constructs can be found in programming languages. For example, the exit point might be at the end or in the middle. Multiple exits are often possible (e.g. using `break` in C), but they are best avoided, since they complicate understanding. Anyway, the invariant must hold at every exit point. Unless the start of the loop is an exit point, the invariant does not have to hold upon entry.

C programmers often write termination tests that update variables. Then the invariant must hold just at the point of exit, after those updates have occurred. Such a termination test should be avoided: it cannot be regarded as stating a property, making the loop harder to understand.

Slide 203

A Trivial Loop Invariant

```
k := 0;
```

Invariant: elements $A[1], \ldots, A[k]$ equal 0

```
while k <> N do
    begin
    k := k+1;          make progress
    A[k] := 0          restore invariant
    end
```

Now $k = N$, so elements $A[1], \ldots, A[N]$ equal 0

This trivial loop initializes array elements to zero. It has an index variable K, and the invariant states that elements $A[1], \ldots, A[k]$ are equal to zero.

At first we have $k = 0$, so the assertion

$$A[1], \ldots, A[k] \text{ equal } 0$$

holds vacuously (no array elements are in the range mentioned). So the invariant holds initially, as required.

At each iteration, the loop body makes progress by increasing k. After doing so, the invariant might fail to hold: we have no reason to believe that $A[k] = 0$ for the new value of k. The loop body immediately sets that array element to zero, restoring the invariant.

Once $k = N$, the `while` loop will terminate. Combining $k = N$ with the invariant yields the desired result, that elements from 1 up to N are set to zero.

If we were working more formally, we should express the invariant using quantifiers:

$$\forall i\,(1 \le i \wedge i \le k \to A[i] = 0).$$

Exercise 5 What is the right invariant for this loop? What assumptions does the loop make about the initial value of N? What can we conclude after the loop terminates?

```
k := 0;  sum := 0.0;
while k <> N do
    begin
    k := k+1;
    sum := sum + A[k]
    end
```

Slide 204

The Outer Loop of Insertion Sort

```
k := 0;
```
$\boxed{A[1], \ldots, A[k] \text{ are an ordered permutation of } A_1, \ldots, A_k}$
```
while k <> N do
    begin
    k := k+1;          make progress
    insert(A,k)        restore invariant
    end
```
$\boxed{\text{Finally } A[1], \ldots, A[N] \text{ are an ordered permutation of } A_1, \ldots, A_N}$

Tiny modifications of the previous loop give us the outer loop of insertion sort. In reading the formulas, note that I have used A_i to refer to the value held in $A[i]$ at the start of execution. The phrase 'an ordered permutation of' is the usual specification of sorting. It expresses that sorting involves re-arranging the given elements so that they are in order. The invariant states that the first k elements of the original input (namely $A_1, \ldots, A_k$) have been sorted and now occupy the array positions $A[1], \ldots, A[k]$.

Initially $k = 0$ and the invariant holds vacuously: nothing has been sorted yet.

At each iteration, k is increased (making progress) and then the procedure call `insert(A,k)` inserts the value $A[k]$ into its proper place among the already-sorted elements $A[1], \ldots, A[k-1]$.

Finally $k = N$ and the loop terminates. The invariant now states that array elements $A[1], \ldots, A[N]$ have been sorted.

Exercise 6 Code `insert(A,k)`, showing the loop invariant. It is simplest to count downwards from `k`, moving up array elements that are greater than `A[k]`.

Slide 205

Defensive Programming (?)

```
k := 0;
while [k < N] do
    begin
    k := k+1;
    A[k] := 0
    end
```

- less risk of endless looping
- weaker termination condition, $k \geq N$

The previous slide's termination condition, $k = N$, seems correct. We need it in order to conclude that the first N array elements have been set to zero, and it is how we expect the loop to terminate.

In practice, such a strong termination condition is dangerous. If $N < 0$ then we shall never reach a point where $k = N$, so the loop will run forever. Worse, as it runs, it will set all memory cells to zero. (In a less trivial loop, where variables are updated in complicated ways, such a disaster could be hard to foresee.)

Many programmers would write our loop as `while k<N` (as shown above) to reduce the risk of endless looping. (This is one element of *defensive programming*.) Then they have to settle for the weaker termination condition of $k \geq N$. Some authorities, notably Dijkstra, insist that `while k<>N` is the correct form. The next lecture will resolve this conflict between theory and practice.

Slide 206

Many loops search through arrays for a desired element. If that element does not exist, then the loop must exit and report failure rather than causing an array subscript error. The loop could include an exit to be taken if the index variable exceeds the array bound. However, having two termination conditions is both inefficient and ugly.

The slide illustrates a technique advocated by D. E. Knuth. A suitable element—the `sentinel`—is inserted into the array's last position. The search will always be successful, and inspecting the final value of i indicates whether the item found was the sentinel or a proper array element.

A general lesson: look for ways to simplify your loops.

(Naturally, you must be sure that $A[N]$ exists. Maguire [14, page 147] gives an example where sentinels are risky. In C, it is easy for programmers to update non-existent memory. How ironic that a supposedly efficient language makes basic optimizations dangerous!)

The invariant for the `while` loop states that elements

$A[1], \ldots, A[i-1]$ each differ from x.

It holds trivially at first, and if $A[i] \neq x$ then adding one to i preserves the invariant. Upon termination, we have the invariant (for the current value of i) together with the termination condition, $A[i] = x$. These two properties together tell us that i is the position of the first element of A that equals x.

For simplicity, the example shows a search for a particular value. The same considerations apply if the search is for any value satisfying some property, such as of being a positive number.

Exercise 7 Code the procedure `insert(A,k)`, using a sentinel to eliminate the need to test for the beginning of the array (see exercise 6).

Slide 207

Algorithm Design Using Loop Invariants

```
k := 0;
while k<>N do
   begin
   writeln(k*k*k);
   k := k+1
   end
```

If multiplication is slow?

The *make progress, restore invariant* structure is found in many classic algorithms, e.g. for finding shortest paths in a graph. The idea can be used to design any loop.

Here, we consider the task of printing (using Pascal's `writeln` procedure) the cubes of the integers from 0 to $N - 1$, where $N \geq 0$. Multiplication is slow on our computer; we should like to eliminate it.

(This example was presented by W. H. J. Feijen at the Marktoberdorf Summer School, 1996.)

Refinement I: $x = k^3$

Slide 208

```
k := 0;  x := 0;
while k<>N do        Invariant: x = k^3
   begin
   writeln(x);
   x := x  +  3k^2 + 3k + 1 ;
   k := k+1
   end
```

$(k+1)^3 - k^3 = 3k^2 + 3k + 1$

We introduce the variable *x* in the hope of maintaining the invariant $x = k^3$. Because *k* is initially zero, we can satisfy the invariant at the start by also setting *x* to zero.

If the program had to work for an arbitrary lower bound instead of zero, then it is hard to see how *x* could be initialized other than by `x:=k*k*k`. Removing the multiplications from the loop body is still better than performing them at each iteration.

Calculating

$$(k+1)^3 - k^3 = (k^3 + 3k^2 + 3k + 1) - k^3 = 3k^2 + 3k + 1$$

tells us what value the loop body must add to *x* in order to preserve the invariant. This large formula does not look like an improvement over k^3, but we have reduced the degree of the polynomial.

Slide 209

Refinement II: . . . **and** $y = 3k^2 + 3k + 1$

```
k := 0;  x := 0;  y := 1;
while k<>N do        Invariant: x = k^3 and y = 3k^2 + 3k + 1
   begin
   writeln(x);
   x := x+y;
   y := y  +  6k+6;
   k := k+1
   end
```

$$(3(k+1)^2 + 3(k+1) + 1) - (3k^2 + 3k + 1) = 6k + 6$$

Continuing as before, we introduce the variable y. Now we *strengthen* the invariant by conjoining $y = 3k^2 + 3k + 1$ to it. (It continues to demand $x = k^3$ as well.) Initially $k = 0$, so we satisfy the invariant at the start by setting y to 1.

In the loop body, we can use y for the current value of $3k^2 + 3k + 1$, but must update y so that it will keep this property at the next iteration, after k has been increased. A tedious but elementary calculation tells us that y must be increased by $6k + 6$.

Notice that we use y (to modify x) before updating it. The first assignment falsifies the invariant; the last assignment makes it true again.

Slide 210

Refinement III: ... **and** $z = 6k + 6$

```
k := 0;  x := 0;  y := 1;  z := 6;
```

Invariant: $x = k^3$ and $y = 3k^2 + 3k + 1$ and $z = 6k + 6$

```
while k<>N do
   begin
   writeln(x);
   x := x+y;
   y := y+z;
   z := z+6;
   k := k+1
   end
```

We still have the problem of computing $6k + 6$ within the loop body. Now it is obvious what to do, but let us keep the formal development. We introduce the variable z and again strengthen the invariant, conjoining $z = 6k + 6$ to it. Initially $k = 0$, so $z = 6$ is forced. The loop body, after using z, increases it by 6; this satisfies the invariant because $(6(k + 1) + 6) - (6k + 6) = 6$.

We are lucky that the variables could be updated one at a time. Occasionally, it happens that the new value of x depends on the current value of y, and the new value of x similarly depends upon x. An example is given by the *simultaneous* assignment

```
x, y := y, y+x;
```

but few languages offer this construct.

This cubes program is of historical interest. Charles Babbage designed his Difference Engine to print mathematical tables using similar ideas. The reduction of multiplication to addition is an instance of *reduction of strength*, which can often be used to remove expensive operations from loops. The reduction of *append* to *cons* (recall the course *Foundations of Computer Science*) is another such instance.

Moral: loop invariants let us design loops systematically and reason about their correctness. We can even make an existing loop more efficient.

Exercise 8 What happens to this program if the specification is changed to allow (a) an arbitrary lower bound, instead of zero, or (b) an arbitrary delta value, instead of one?

Slide 301

Fault Avoidance

BUGS CAN KILL

Prevention, not Cure

- *Keep It Simple*
  design — interfaces — code
- *Check Everything*
  compiler warnings — run-time checks

Some practitioners dislike the word *bug* because it belittles what deserved to be called a fault or defect. Whatever word you use, never underestimate bugs. Some have killed people; others have cost hundreds of millions of pounds in damage (e.g. the loss of the Ariane 5 rocket). Most programmers can recall a bug that cost them days or weeks to find.

How do you catch bugs? One survey [4] found that 80% bugs were found not using fancy debuggers but by primitive means, such as inserting `print` statements and hand simulation. Debugging tools can be valuable. A simple diagnostic technique is to reduce the problem by deleting parts of the input (or even the program) to see if the bug is still there. Sometimes, you can reduce the input to two lines, making it easy to find the bug.

Two major causes of bugs are memory overwrites and faults in vendor-supplied products. Memory problems are hard to isolate because the symptoms often appear long after the original error.

The best debugging technique is to avoid having bugs in the first place. This lecture presents techniques, mostly suggested by Maguire [14], for detecting bugs early. I have organized his many suggestions under two slogans: *Keep It Simple* (or KISS: Keep It Simple, Stupid) and *Check Everything*. Keeping it simple avoids introducing bugs. The compiler can detect many errors, and many remaining ones can be caught if your code includes thorough integrity checks.

Slide 302

Efficiency

Is it *necessary*?

Chief cause of wrong decisions

- coding tricks
- using assembly language or C

50% of runtime is spent in 5% of the code!

Use *profiling* to find that 5%

If you never consider efficiency at all, then it will be hard to make your program efficient later. Think about efficiency during the design phase, but not during the coding. Optimize your code later, after you have identified the critical parts.

Most programs have one bottleneck that determines their performance, regardless of how the rest is coded. We speak of the *inner loop*, but the bottleneck is seldom where you expect. Use *profiling tools* to discover which part of the program uses the most time and space.

I once worked on a large Pascal program that was spending one third of its time executing the following statement:

```
for i := 1 to 10 do str[i] := ' ';
```

Nobody had noticed this innocuous line, which performed string subscripting (slow on our PDP-10). We replaced it by the equivalent (but very fast) statement

```
str := '          ';
```

As you eliminate such bottlenecks, eventually you arrive at those that are inherent in the computational task. They can be improved through better algorithms or by recoding small parts in assembly language.

Efficiency-obsessed programmers write obscure code that is full of bugs. Code your functions straightforwardly. Before you resort to profiling, ask yourself whether the program is fast and small enough for its intended task. For many programs, reliability is more important than efficiency.

I know of a major project that failed because the investigator insisted upon coding it all in assembly language, on the grounds that one critical part had to be efficient. In the end, efficiency proved to be a red herring: the system seldom worked at all.

Exercise 9 Use a profiler (such as Unix's `prof`) on a program of your own. If possible, recode the bottleneck it identifies and measure the resulting improvement in performance.

Slide 303

Re-Inventing the Wheel

- *Libraries* (numerical, string-handling, . . .)
- *Parser generators* (yacc)
- *Specialized languages* (tcl/tk, perl)
- Commercial-Off-The-Shelf Software

Why write code if somebody else has done it for you?

A well-designed library might satisfy requirements and do so better than code you could write yourself. Some people don't use libraries because they find programming more fun than reading library manuals. Bad documentation increases this tendency. So, in large software systems, some functions are implemented many times, wasting space and making maintenance harder.

A *parser* is (in its most typical application) the first stage of a compiler. Its job is to analyze the source file according to the grammar of the programming language, identifying the functions, statements, etc. Coding a parser by hand is difficult and error-prone. A *parser generator*, given the required grammar, produces a parser automatically. Machine-generated parsers are often faster and smaller than hand-coded ones, and handle errors more uniformly.

You can build a user interface much quicker using tcl/tk than by calling windowing primitives directly. It will be slow, but it will do the job. For text-processing tasks, a few lines of perl can replace hundreds of lines of C.

Finally, consider whether some commercial package can do the whole job for you. During the 1960s, when computers were hugely expensive, companies generally wrote their software in-house. Now that hardware is cheap, bespoke (custom-made) software is seldom economic. But if the commercial package is a poor match to your needs, you might have to roll your own.

Slide 304

Simple Design

Resist imposed changes

Don't add *your own complications*

Beware of ACCIDENTAL FEATURES

It all has to be supported and documented

A simple design is your one best defence against software problems. Getting it simple, and keeping it simple, is hard.

There are many sad tales of projects going wrong due to changes imposed from above, especially late changes. Resist such pressure if you can. Don't complicate matters by suggesting changes yourself. Don't let the fun side of a project block your professional judgement.

During coding, new features may come 'for free' because of arbitrary implementation decisions. Nothing is free, however. If the new features are added to the design, then they are likely to stay there: you are committed to those decisions, for better or worse.

Software, for micros especially, has got more bloated in recent years. It seems slower than ever, despite faster hardware. It seems harder to use, despite masses of on-line documentation. A certain word-processing package is often cited as an example of this phenomenon.

There are examples of good practice, too. My personal favourite is *Textures*, a TEX environment for the Macintosh. Its developers seem to have concentrated on making it faster and more convenient rather than loading in features.

Slide 305

Simple Function Interfaces

Avoid MULTIPURPOSE FUNCTIONS

Flag EXCEPTIONAL CASES cleanly

Is that exception *necessary*?

Error values versus *booleans*

Consider the *caller*

When designing a suite of functions, especially ones that other programmers will call, devote some effort to making them easy to understand and use. Think about how your functions will be called in typical circumstances. (You'll need examples for the documentation, anyway.) Each function should have one clear purpose and should work harmoniously with the others.

As an example of how not to do it, Maguire [14] cites C's *realloc* function. This function performs four quite distinct operations (allocating storage, releasing it again, increasing a block's size, decreasing a block's size). The description of *realloc* goes on for paragraphs. No wonder programmers don't like reading manuals!

Functions like *realloc* come from a coding style that attempts to make sense of every combination of arguments. Instead of signalling an error, the function attempts to do something sensible. Fine judgement is needed. Functions should not impose arbitrary restrictions, but they should not accept rubbish either. That can mask bugs, making them harder to find. For instance, a function to allocate an array of size n should succeed if $n = 0$, but fail if $n < 0$. An empty array is a meaningful concept, but a negative-size array is not.

Exceptional situations must be signalled to the caller. In ML, you can return a result of type `option` (with its `NONE` and `SOME` constructors) or raise an exception. Again, judgement is needed. Using type `option` forces the caller to handle the error, when perhaps it should be handled higher up; raising an exception has the risk that nobody will handle it.

Using type `option` is an example of signalling failure by returning an error value. It works neatly with ML's `case` construct (turn back to Lect. 1). In C, error values force a clumsy coding style on the caller. A notorious example is function *getchar*, whose result type is not *char* but *int*: normally it returns a character, but it signals end-of-file by returning an error value. A separate boolean output or a separate end-of-file function would do the job better.

Slide 306

Simple Code

Keep to a *straightforward* style

Banish needless *optimizations*

Rely on *public interfaces*, not INTERNAL DETAILS

(Workarounds for *known bugs*?)

POINTERS need extra care

Straightforward code solves the task in the obvious manner. Tricky code might be more fun to write, but it will probably cost you in the end, especially if other people have to maintain your code.

Tricks such as using binary shift instead of multiplication by two can make the code non-portable: they can give the wrong answer on some types of hardware. Even when it works, the improvement in efficiency is small, but the cost in clarity is great. Leave the low-level optimizations to your optimizing compiler.

The worst piece of code I ever saw was an assembly-language routine that changed an instruction in another routine before calling it. (It was coded by an ‘ace programmer.’) This example may be laughable, but there are countless cases in which programmers exploit the internal details of another function (or data structure). When a new version of that function is installed, calls to it fail.

Information hiding means using language features such as objects or modules to deny others access to internal details. No programming language can hide all details, e.g. of what a function does when its behaviour is officially undefined: the programmer must exercise discipline.

Sometimes you know that a library function returns the wrong answers in some cases. If you are forced to code a workaround, make sure your code will continue to work after the library function is fixed.

Pointers are dangerous, especially in C, where errors can corrupt memory and cause mysterious failures elsewhere. Perhaps half of Maguire’s book is devoted to preventing such problems. When using pointers, keep to the simplest and safest style possible. Avoid overwriting your caller’s memory; such a bug in Unix’s *fingerd* program allowed the Internet worm to invade thousands of machines in November 1988.

Slide 307

Check Everything (Compile Time)

Use a high-level language: *Java, Ada, Modula-3, ML*

- strong typing
- information hiding (objects or modules)
- exception handling
- clear syntax

Switch on all *compiler warnings*

Use static diagnostic tools

Let the computer catch your errors! Why spend hours tracking down a bug caused by calling a function with the wrong number of arguments, when the compiler can find such bugs itself? ML is exceptionally safe: its type system has no loopholes and it even forces variables to be initialized. The only way an ML program can fail, barring faults in the underlying system, is by not catching an exception.

ML is not suitable for every application: most implementations use too much storage (e.g. 300KB for the runtime system alone). Other languages provide a degree of safety, including Java, Modula-3 and Ada. *Strong typing* prevents your confusing a pointer with an integer, say; *information hiding* protects the integrity of data structures; *exception handling* provides a dedicated mechanism for managing run-time errors; a *clear syntax* benefits everybody who reads your program.

Object-oriented methods are popular. They are valuable because they provide information hiding (though imperfectly) and make it easy to provide libraries of reusable components. A good module system (such as ML's) gives the same advantages.

Untyped languages such as Lisp make it hard to achieve reliability. Assembly language makes it hard to achieve anything. C is riddled with pitfalls, such as the following [14]:

```
if (ch = '\t')
    ExpandTab();
```

The expression inside the `if` is not an equality test but an assignment!

Sometimes you have to use a questionable language because of legacy code, libraries, compatibility concerns, or management prejudice. Make the most of whatever language you use. Enable all compiler warnings so that all questionable C constructs are rejected. A bonus: clean code is not just more reliable but is more likely to be portable.

Slide 308

Check Everything (Run Time)

Include lots of *debugging code*

- assertion checks
- backup versions of optimized algorithms
- distinctive initial values
- active consistency checks

Debug code runs *in addition to* the real code

Test **every** new line of code

Naturally, you will enable all run-time tests, e.g. of array bounds. Augment the language's automatic checks with those of your own, using *assert* statements. These raise an error unless the supplied boolean expression evaluates to `true`.

Maguire [14] supplies a complete 'debug' version of the C store allocation routines, which record enough additional information to detect (automatically) common errors such as referring to a block after releasing it. Compaq recommend a tool called Third Degree, which checks for heap memory leaks, references to uninitialized data, etc. Electric Fence, by Bruce Perens, is a free utility that halts a program as soon as it overruns an allocated memory buffer.

Laden with debug code, your program may run two or three times as slow as it would otherwise. Such a degradation is tolerable during testing. For production runs, the debugging code can be omitted by setting the 'debug' variable to *false* and recompiling. If you can leave in some checks, so much the better.

Tricky, fast code can be checked using assertions that compare its results with that obtained by backup code, written straightforwardly. As an example, Maguire cites a spreadsheet engine.

Forgetting to initialize a variable can have drastic consequences. If your debug code initializes everything to zero, then it might mask errors, and your program will fail when the debug code is removed. Maguire suggests forcing variables to be initialized with a value carefully chosen to provoke a fault however it is used: it should be an invalid instruction, an invalid address, etc.

Do you have a complicated data structure involving lots of pointers? Consider coding an integrity checker, a function that traverses your data structure to ensure that it is well-formed. Find the bugs before they find you.

'Test your code' seems obvious. It is easy in ML: whenever you code a new function, test it on values that exercise every execution path. (It is easier to test a new three-line function than to test three lines added to a 100-line function, so try to do without the latter.) In C, Maguire suggests that you use an interactive debugger to step through every line of new code. Hand simulation is less good: if you made a mistake in the first place, you could misinterpret what the computer will do when it reaches your mistake. Build a collection of tests that can be given to your program every time you make a change.

Slide 309

That While Loop Again

```
assert (N >= 0)          precondition satisfied?
k := 0;
while  k < N  do         discourage endless looping
    begin
    k := k+1;
    A[k] := 0
    end;
assert (k = N)           postcondition satisfied?
```

This example of `assert` demonstrates safety checks for a `while` loop. As discussed in Lect. 2, we strengthen the guard from $k \neq N$ to $k < N$, preventing looping even if somehow k comes to exceed N. But we do expect $k = N$ to hold, so we add an assertion to check it. The loop's task (to zero the first N array elements) makes sense only if $N \geq 0$ initially, so we put yet another assertion before the loop.

Strengthening the guard without adding the assertions is risky. By preventing the endless looping, it would mask the underlying error, namely that the loop was entered with $N < 0$.

In this case, either assertion on its own would suffice, but the redundancy does no harm. Assertions document the relationships that hold among your variables. Comments are often wrong; assertions are machine-checked.

Some programmers dislike the sort of advice given above. Type-checking cramps their style. It all takes the fun out of programming. But nobody likes tracking down bugs, having their project cancelled or hearing that a rocket crash was their fault. The crucial question is whether this sort of advice (sermonizing, if you will) reduces the risk that those bad things will happen. In fact, hard evidence is scarce. Anecdotal evidence indicates that it does, and common sense says that actively seeking to reduce risk can only improve our safety record.

Are anecdotes and common sense a sound basis for Software Engineering? No. Precisely defined disciplines need to be developed, with scientific studies that prove their efficacy.

Slide 401

What are Formal Methods?

- *not* ‘structured methods’
- Formal specification
- Refinement to code
- Formal correctness proofs
- Rigorous code analysis
- Tool support

Formal methods are grounded in mathematics. A formal specification eliminates ambiguity, giving a precise notion of correctness.

Formal methods are sometimes taken to include graphical methods such as dataflow analysis. But unless they are fully precise, they cannot be regarded as formal. Most CASE (Computer-Assisted Software Engineering) tools support graphical methods. Formal methods also benefit from tools: to help users write syntactically correct specifications, to run simple semantic checks on them, and to help in the refinement of specifications into code.

Formally correct code can be produced in two ways. *Program derivation* or *synthesis* involves transforming a specification into code by steps guaranteed to preserve correctness. The programmer supplies the transformations; at every stage, the machine checks that the code is compatible with the specification.

Alternatively, the programmer could write a routine and submit it afterwards for proof. This is often called *program verification*, but note that *verification* is also used in the context of testing. Proving correctness requires a lot of time and skill; for most projects, it is too expensive. Unless the program was coded with verification in mind — avoiding low-level tricks — it may be practically impossible to prove correct.

Code can be analyzed systematically without constructing a completely formal proof. This technique was used to certify nuclear reactor shutdown software; see below. Real software projects seldom involve formal proofs. The main use of formal methods is in writing formal specifications.

Testing also requires correctness to be defined precisely. But testing encompasses other things, such as customer satisfaction, that lie outside the scope of formal methods.

Slide 402

What are Specifications For?

- deeper analysis of requirements
- detecting inconsistencies
- specify *what* not *how*
- communication with implementers
- communication with testing team

A formal specification is essential if you are going to prove correctness, or to support transformation into correct code. Less ambitiously, formal proof can be used to derive properties from a specification; this could reveal inconsistencies early. The specification is also useful in itself. Studies have shown that attempting to write a formal specification stimulates deeper thinking about the requirements, showing up ambiguities hidden in English.

The ConForm Project [5] is investigating the costs and benefits of using formal methods in building a small security-critical system. Two teams are independently developing a so-called trusted gateway. One team is using fairly conventional structured methods; the other augments these methods by writing a formal specification. They are using VDM, the Vienna Development Method, which has many adherents. The project is monitoring the development process, comparing the effort required to complete each phase, the quality of the documents produced, etc.

Early in the project they noticed the team using formal methods asked many more questions concerned with clarifying the requirements. The job of the trusted gateway is to take a stream of messages and forward each message either to a 'secret' or 'non-secret' output port; the decision is based upon certain keywords that may appear in messages.

Messages are limited to 10K. The formal methods team asked whether this limit included the message delimiters (it did). If a message contains both 'secret' and 'non-secret' keywords then it is regarded as secret. However, the formal methods team noticed the possibility that a 'non-secret' keyword could contain a 'secret' keyword as a substring. The developers had to go back to the customers to find out that such occurrences of 'secret' keywords should be ignored.

These are perfect examples of ambiguities that lurk in English descriptions, and that could lead to obscure errors. How many messages will be under 10K if delimiters are ignored, and over 10K if they are counted? The precision of a formal specification will help the implementers build a correct system, particularly if they have tool support. And the specification will help the testing team identify awkward cases to cover in test data.

It's not a bug, it's a feature! — formal specifications can help put an end to this (though it is partly a problem of requirements). Recall our problems in the first lecture.

Slide 403

What is a Specification Language?

- *precisely* defined syntax and semantics (meaning)
- *executable* specifications: functional or logic program, . . .
  - – rapid prototype **(Good)**
  - – implementation bias (BAD)
- specification languages for *sequential* programs:
  - – Z, VDM, Larch, . . .
- specification languages for *concurrent* systems:
  - – LOTOS, Unity, TLA, . . .

There are many specification languages, with different purposes. All have a precise definitions of their syntax and semantics. A given piece of text is either legal or not; if legal, it has a precise meaning. However, the meaning does *not* determine the implementation uniquely; rather it defines precise grounds for judging whether an implementation is correct.

A program counts as a specification. Programming languages are precisely defined (or should be), both their syntax and semantics. *Executable* specifications consist of programs written in very high-level languages paying no attention to efficiency [17]. They are precise, and (compared with a real implementation) they are easy to write, read and reason about. They also yield an executable prototype. They have many drawbacks, though. They may be too inefficient to serve even as prototypes. Making them executable will introduce implementation bias; they will not be abstract enough. They will map every input to a unique output, when normally for each input there is a set of legal outputs.

Consider a sorting program: its output should be an ordered permutation of its input. It is easier to say that than to write even a highly inefficient functional sorting program. Consider a compiler: its output is a string of machine instructions. If we specify the output uniquely, we shall not be allowed to include optimisations.

The meaning of a specification is defined in terms of mathematical abstractions. Early work concentrated on specifying data types, such as lists, stacks, queues and symbol tables; such work (e.g. Larch) was based on the theory of algebras.

Most modern specification languages treat computation as a whole, though still abstractly. A *sequential* program can be regarded as a function from inputs to outputs, or more generally as a relation between inputs and acceptable outputs. Z and VDM specify programs by modelling their data structures using elementary set theory.

A *concurrent* program is normally viewed as a system of communicating agents. This requires an abstract notion of agent behaviour, based upon something like a process algebra. Temporal logic is usually involved, for making statements about time dependencies: *A* and *B* cannot happen simultaneously; if *A* happens then *B* must happen eventually, etc.

Slide 404

Seven Myths of Formal Methods

1. *Formal methods guarantee perfection.*
2. *They work by proving correctness.*
3. *They are only good for critical systems.*
4. *They involve complex mathematics.*
5. *They increase costs.*
6. *They are incomprehensible to clients.*
7. *Nobody uses them for real projects.*

This classic paper [8] by Anthony Hall of Praxis Systems is based upon industrial usage of formal methods. Here is a summary of how he refutes each myth.

1. All human methods are fallible. In particular, the specification could be an inadequate model of the real world. Errors can also occur in machine-checked proofs. The proving program could itself be faulty. Using it to prove itself ('verifying the verifier') does not solve the problem; as an extreme case, suppose it regarded all programs as correct?

But formal specifications do help find errors, because they eliminate arguments about what the specification actually says.

2. This myth reflects the US emphasis. European work is more oriented towards specification.

3. Praxis uses formal methods merely to help ensure high quality, even for non-critical software.

4. Formal methods are in fact based on (the easier parts of) discrete mathematics: set theory and logic. Staff training only takes about three weeks. Compare with the complexity of programming languages and client applications! But correctness proofs require more complex mathematics.

5. Development may be *cheaper* with formal methods. However, the requirements phase may take longer and cost more. It takes time to write any specification at all. The initial specification can usually be simplified as the problem is better understood. Time spent here is repaid during the implementation and maintenance phases.

6. You can paraphrase the specification in natural language and use it to derive consequences of the requirements.

7. Hall describes applications by IBM, Tektronix, Rolls-Royce as well as his own firm. Since his article was published, many other industrial uses have been reported — see below.

There is still much disagreement on whether formal methods are useful or not. For every devotee, there is an arch-sceptic.

Slide 405

Experience with Formal Methods

- **SSADM tool set**, by Praxis Systems. 37,000 lines of code
- **CICS transaction system**, by IBM Hursley. 50,000 lines
- **Cobol/SF** by IBM Federal Systems. 800,000 lines
- **Air Traffic Collision-Avoidance System**, by FAA
- **UEPS** electronic funds transfer product
- **Air Traffic Control Information System**. 197,000 lines of code

Not many systems have been built with the help of formal methods, but some examples are noteworthy. Three examples will be discussed in separate slides below: the Darlington nuclear power plant, the Paris Metro signalling system and a British air traffic control information system. A major study by Susan Gerhart and others [6] investigated twelve projects involving the use of formal methods, including commercial systems and some projects involving critical software.

Some of the projects reported by Gerhart started in the early 1980s, using methods now obsolete. Some used archaic tools or no tools at all. A more recent collection of articles has been compiled by Hinchey and Bowen [11].

The SSADM design tool built by Praxis inspired Hall's paper [8]. It involved 450 staff-weeks of effort, two devoted to writing the Z specification.

IBM's Customer Information Control system is large, 800,000 lines of code. IBM is now using the Z specification language to re-engineer this system; the 50,000 lines quoted above were developed in this way.

Cobol/SF is a tool for tidying up old Cobol programs while preserving their meaning. IBM built it using the Cleanroom methodology, which is based upon (informal) proof.

The US Federal Aviation Authority (FAA) hired Nancy Leveson to apply formal methods to subsystems of TCAS (Traffic Alert and Collision Avoidance System) because they were worried about the 'loss of intellectual control over the specification.' She applied a graphical formal method (a variant of Statecharts).

The Universal Electronic Payment System (UEPS) used smartcards. Its funds-transfer operations were protected by security protocols. It is thought to be the first financial system whose protocols were analyzed using a formal method: the BAN logic. The system was a commercial success and in 1996 was adopted by Visa as the COPAC electronic purse.

Slide 406

Darlington Nuclear Power Station

- two independent shutdown systems
- 26,000 lines of code (including assembler!)
- formal methods used to certify *existing* code
  – formalise requirements as specification tables
  – analyze code as program-function tables
  – compare the tables
- *No tool support*
- cost $2-4 million Canadian

This nuclear power station is roughly 40 miles from Toronto, Canada. Lauren Wiener's account of the project [18] is quite different in tone from Craigen et al.'s [2].

Emergency shutdown systems are normally controlled using 'switches and relays and analogue meters' [18] . The Darlington nuclear power station, unusually, built its emergency shutdown systems in software. There were 6,000 lines of assembly, 7,000 lines of Fortran and 13,000 lines of Pascal among the two systems. The Canadian authorities refused to licence the plant after problems were found in the software.

A formal code inspection was organised by David Parnas using the SCR method (Software Cost Reduction). Each process was analyzed by three independent teams. One used the informal requirements document to generate a specification table. The second examined the existing code and generated program-function tables. The third examined the two sets of tables and reported discrepancies. The work was tedious and labour-intensive. They effected a hundred or so minor changes to the system, but found no serious errors.

A remarkable feature of this work was that it dealt with existing code, including assembly language. It involved rigorous analysis but not formal proof.

Wiener [18] claims that certifying the software delayed the plant's opening by six months, at a cost of $20 million per month in lost production (Canadian dollars). The software verification cost $2-4 million. A hardware shutdown system costing $1 million would therefore have been much cheaper. That is an argument against using software in nuclear power stations. It is no argument against formal methods, without which the software might not have been approved at all. One has to ask what safety criteria are used to certify traditional control systems?

Paris Metro Signalling

Slide 407

- reduce train separation from 2:30 to 2 minutes
- *by GEC Alsthom. 9,000 lines of verified code*
- 4-stage validation process
  – requirements validation
  – testing
  – safety/hazard studies
  – certification
- *Hoare logic, for proving correctness*
- B method, for refinement

The Paris Metro's new signalling system allows trains to run two minutes apart, a savings of 30 seconds. The increased capacity has eliminated the need for another railway line. The project was funded in 1982, a prototype was finished in 1985 and the system was deployed in 1989. Initially the developers used Hoare logic for correctness proofs, as the best available technique in 1982. Hoare logics are the basis for most approaches to proving correctness of software, but they can be complicated to use. The developers were unsure how to apply them on such a large scale. Jean-Raymond Abrial (one of the developers of Z) helped them to re-specify and re-verify the software.

Validation was divided into four stages: *validation of requirements*, *verification and testing*, *operations and maintenance*, and *certification*. They used other tools such as SADT (Structured Analysis and Design Technique) and performed hazard studies using fault-tree analysis. They used extensive testing, finding many problems with the specification. Testing is the only way to find out whether a program meets its real-world requirements; a correctness proof can only show that a program meets its specification.

Hoare logic [13] concerns statements of the form $\{P\}S\{Q\}$, meaning 'if P holds beforehand, and if execution of S terminates, then Q will hold afterwards.' In its pure form it says nothing at all if S fails to terminate, but it can be augmented to prove termination as well. It is not a specification language but a method for proving properties of code.

The B method models a process as an abstract machine. One abstract machine can be implemented by means of another. This accounts for the different levels of abstraction found in computer systems (machine language, operating systems functions, library functions, modules, subsystems, etc.). It supports development by top-down refinement, where an abstract machine is implemented in terms of increasingly lower-level machines.

GEC Alsthom, the developer, is now using the approach for other railway products. One is a safety system covering all electrified lines in the French railways.

Slide 408

Air Traffic Control Information System

- displays information to controllers: flights, weather, . . .
- *requirements*: high reliability, guaranteed response times
- built with a mixture of structured and formal methods
  – dataflow diagrams
  – VDM (Vienna Development Method)
  – VVSL (a variant of VDM for modules)
  – CSP (Communicating Sequential Processes)

This information system, known as CDIS [9], was developed by Praxis and delivered to the Civil Aviation Authority to support air traffic controllers working in the London area. It supports over 50 workstations used by controllers and other staff. It is a safety-critical system with high reliability requirements: for instance, it must be available 99.97 percent of the time. It is a real-time system: information must be displayed within one or two seconds of receipt. It is also a large system, with different aspects such as concurrency and the user-interface.

Several different formal methods were employed in its development. In addition to standard software-engineering methods such as dataflow and entity-relationship diagrams, the developers employed VDM to specify the system's requirements. They found that formalizing the specification forced them to ask questions that helped them to understand the requirements more thoroughly. But the specification was unable to distinguish between requirements that were essential and those that were merely desirable.

VDM alone could not suffice to specify this system because much of its functionality concerned the user interface, which VDM could not help with. To cope with the sheer size—the specification comprised 150 operations—a variant of VDM known as VVSL was used to structure the description into modules.

Concurrency was also an issue because inputs could arrive from many sources at the same time. This aspect of the system was specified using CSP, Hoare's language for specifying Communicating Sequential Processes [12].

Hall claimed that the specification phase was successful, but he also noted numerous deficiencies. The specification was hard to read and yet gave only an approximation of the desired behaviour.

The delivered system comprised 197,000 lines of code and the specification documents were 1,200 pages long.

Slide 409

Research into Formal Correctness

- model checking
- *hardware verification*
- system verification
- *protocol verification*
- program design calculi

Model checking is complementary to formal proof; it works for finite-state systems. It simply consists of enumerating all possible states (10^9 or more) and checking the desired property. Current research is investigating ways to prove properties of infinite-state systems by viewing them as finite-state systems.

Hardware verification is well advanced. The most successful method, based on higher-order logic, is due to M. J. C. Gordon here at Cambridge. Correctness properties have been proved for many real chips.

System verification involves proving the correctness of subsystems, and of their integration, so that the whole system is proved correct. Bevier et al. [1] describe the proof of a 'stack' of components ranging from a simple high-level language to a microprocessor design. The aim is to have a computer system that is entirely free of logical errors, and that can only fail due to environmental conditions. (Note that for real-world applications, environmental conditions will remain a significant cause of failures.)

Protocols are used in consumer electronics (e.g. remote controls) and telecommunications. *Cryptographic* protocols are used in security-critical systems, for example to deliver encryption keys. They are a common source of errors, since they are usually designed to work in the presence of unreliable media. Many protocols have been verified: the task is easier than verifying the software itself. Proofs depend on a model of unreliability. We assume, for example, that a network may re-order or lose messages, but not corrupt them.

Program design calculi provide a precise way of constructing code to meet a formal specification. Many calculi are under investigation. Some use functional programming languages, which are particularly easy to reason about. Other methods apply to the usual (imperative) sort of language, although languages like C are hard to model. A popular line of research involves deriving programs from suitably constructive proofs.

Slide 501

The Z Specification Language

Schemas used to define

- the legal state space
- operations that change the state
- operations that inspect the state
- special cases of an operation

Incremental development of a specification

Data described using set theory

This lecture is based on Spivey [16]. It presents his trivial example, the *Birthday Book*, a system that can record people's birthdays and issue a reminder for them.

Schemas are peculiar to Z. They are a bit like record operations: they describe a collection of named fields (which are the program variables), with an associated set of constraints. The constraints can specify a number of things, including relations that hold among the variables and actions affecting the variables. You can define a schema for each operation. But an operation can, in fact, be defined in terms of several schemas: one schema for the normal case, and other schemas for various exceptional cases. Schemas can be introduced one at a time.

Another popular specification language is VDM (the Vienna Development Method). VDM is unusual for its use of a three-valued logic, as a way of reasoning about definedness (particularly, termination). VDM includes methods to help refine the specification into code.

Z was developed at Oxford University by Jean-Raymond Abrial, Bernard Sufrin, Carroll Morgan and others. VDM was developed at the IBM Laboratory in Vienna by Cliff Jones, Dines Bjørner, Peter Lucas and others. The two languages look quite different, but in most essential respects they are the same.

One key difference is the treatment of an operation's *precondition*: a property that must hold before the operation may be invoked. In VDM, you specify the precondition directly. In Z, if an operation is built out of several schemas, the precondition is specified in bits and pieces.

Both languages use basic concepts from set theory to describe data and operations. This is called the *model-oriented* approach; such a specification is a bit like an implementation in set theory (so, of course, it is not executable). So-called *property-oriented* specification languages involve stating the desired properties of a module without exhibiting a mathematical model for it.

Slide 502

Some Z Notation

$\mathbb{P}X$ is the set of subsets of X

$x \in A$ means x is an element of A (and $x \notin A$ is its negation)

$A \subseteq B$ means A is a subset of B

$A \cup B$ is the union of A and B

$f : A \rightarrow\!\!\!\!\!+ B$ means f is a *partial* function from A to B

$\operatorname{dom} f$ is the domain of f

$f \cup \{x \mapsto y\}$ extends f to map x to y

Z includes a formal mathematical language.

$f : A \rightarrow B$ means f is a *total* function from A to B: it maps *all* elements of A to elements of B. It is not used below, but is the natural way of specifying arithmetic operations, for instance.

$f : A \rightarrow\!\!\!\!\!+ B$ is used below to represent a table. We specify a *partial* function as we do not expect a table to contain an output for every conceivable input.

$\operatorname{dom} f$ is not interesting for total functions; if $f : A \rightarrow B$ then $\operatorname{dom} f = A$. But if f is a partial function, then $x \in \operatorname{dom} f$ if and only if $f(x)$ is defined.

$f \cup \{x \mapsto y\}$ is the function that agrees with f except that its domain is enlarged to map x to y. Here $\{x \mapsto y\}$ is a trivial function whose domain is $\{x\}$. Since a function is a set of pairs, $\{x \mapsto y\}$ is simply a nicer syntax for the ordered pair of x and y. Also $f \cup g$ combines the functions f and g, but the result will not be a function unless f and g agree where their domains intersect.

More generally, $f \oplus g$ combines f and g, with g overriding f where their domains intersect. So $f \oplus g$ will always be a function provided f and g are. The function $f \oplus \{x \mapsto y\}$ is a version of f *modified* to map x to y. It can be used to modify any function (partial or total), or to extend a partial function's domain.

This sort of abstract notation allows us to express data without concern for the implementation. A partial function could be implemented as an array, a list, a tree, a B-tree on disc, etc.; such decisions are taken later in the design stage.

Z includes many more symbols: for sequences, Cartesian products, tuples, etc. In addition, there are all the logical symbols: and, or, not, implies, etc. Unfortunately, VDM frequently uses different symbols for the same concepts. Both languages often differ from standard mathematical usage.

Slide 503

Defining the State Space

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \nrightarrow DATE$$

$$known = \operatorname{dom} birthday \qquad \boxed{\textit{Invariant}}$$

State variables

- *known*: a set of *NAME*s
- *birthday*: a partial map from *NAME*s to *DATE*s

Z *schemas* allow specifications to be structured and combined. Specifications could be written using the mathematical language alone, but schemes are more compact and more natural.

Our description on the slide is very abstract. We have not specified anything about the structure of a *NAME* or *DATE*. We have placed no limit on the number of names stored. Such points can be specified later. But since *birthday* is a function, we have specified that a name can be assigned at most one birthday.

A state space has two key features. The *state variables* are the components that make up the state. The *invariant* is the relation that must hold of the components. For the birthday book, the state has two components, *known* and *birthday*, where *known* is entirely determined by *birthday*.

A more realistic system would have a more complicated relationship among its components. We could add a new component, mapping names to addresses say, with the restriction that you can only record an address if you also record the same person's birthday.

BirthdayAndAddressBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \nrightarrow DATE$$
$$address : NAME \nrightarrow ADDRESS$$

$$known = \operatorname{dom} birthday \land \operatorname{dom} address \subseteq known$$

We could have expressed this schema by combining *BirthdayBook* with a small schema specifying *address*. It is hardly worth the trouble here, but for larger specifications the ability to combine schemas is invaluable.

Every operation on the state must *preserve the invariant*: it may assume that the invariant holds at the start, and must ensure that it holds at the finish. The concept of invariant is not specific to Z, but is fundamental to Computer Science. The ConForm Project [5] found that specifying the invariant helped the designers identify pathological cases.

Slide 504

A State-Changing Operation

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{AddBirthday}\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}}\\
\Delta \mathit{BirthdayBook}\\
\mathit{name}? : \mathit{NAME}\\
\mathit{date}? : \mathit{DATE}\\
\hline
\mathit{name}? \notin \mathit{known} \qquad\qquad \boxed{\text{Precondition}}\\
\mathit{birthday}' = \mathit{birthday} \cup \{\mathit{name}? \mapsto \mathit{date}?\} \qquad \boxed{\text{Operation}}\\
\hline
\end{array}$$

Invariant: implicitly present

AddBirthday adds *name*? to the state, assigning to it the birthday *date*?. Since this operation changes the state, we specify it using a Δ schema that includes *BirthdayBook*. The schema contains two copies of *BirthdayBook*'s state. The variables *known* and *birthday* represent the initial values, while the primed variables known' and $\mathit{birthday}'$ represent the final values.

Variables ending with a question mark, such as *name*? and *date*?, represent the operation's inputs. Output variables end with an exclamation mark; this schema has none, but see below. An equation such as

$$\mathit{birthday}' = \mathit{birthday} \cup \{\mathit{name}? \mapsto \mathit{date}?\},$$

looks like an assignment statement, but actually it *defines* a final value in terms of initial values and inputs. The equation specifies that the *birthday* function will be extended to map *name*? to *date*?. The relation between initial and final states does not have to be given by equations, especially if the input state does not constrain the final state uniquely.

The schema *AddBirthday* is subject to the precondition $\mathit{name}? \notin \mathit{known}$: the name must not already have a birthday assigned. Otherwise $\mathit{birthday}'$ might assign two different birthdays to *name*?; it would no longer be a function! A schema specifies an operation *provided* the precondition holds.

The invariants are added implicitly: $\mathit{known} = \mathrm{dom}\,\mathit{birthday}$ is part of the precondition, while $\mathit{known}' = \mathrm{dom}\,\mathit{birthday}'$ is part of the effect. The latter equation allows us to derive an explicit value for known':

$$\begin{aligned}
\mathit{known}' &= \mathrm{dom}(\mathit{birthday} \cup \{\mathit{name}? \mapsto \mathit{date}?\})\\
&= \mathrm{dom}\,\mathit{birthday} \cup \mathrm{dom}\{\mathit{name}? \mapsto \mathit{date}?\}\\
&= \mathrm{dom}\,\mathit{birthday} \cup \{\mathit{name}?\}
\end{aligned}$$

Using the invariants, we obtain $\mathit{known}' = \mathit{known} \cup \{\mathit{name}?\}$. We have also used basic properties of domains, $\mathrm{dom}(f \cup g) = \mathrm{dom}\,f \cup \mathrm{dom}\,g$ and $\mathrm{dom}\{x \mapsto y\} = \{x\}$.

Formally, a schema consists of variables that are constrained by a formula. The precondition, invariant and operation are simply parts of that formula, with no special interpretation of their own.

Slide 505

A State-Inspecting Operation

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FindBirthday}} \\
\Xi \textit{BirthdayBook} \\
\textit{name}? : \textit{NAME} \\
\textit{date}! : \textit{DATE} \\
\hline
\textit{name}? \in \textit{known} \qquad \boxed{\textit{Precondition}} \\
\textit{date}! = \textit{birthday}(\textit{name}?) \qquad \boxed{\textit{Operation}}
\end{array}
$$

No effect on state — instead, yields an *output*

FindBirthday looks up $name?$ in the state, returning the associated birthday as $date!$. Since this operation never changes the state, we specify it using a Ξ schema that includes *BirthdayBook*. Strangely enough, this schema also contains two copies of *BirthdayBook*'s state, just as a Δ schema would. But it also contains implicit constraints that the state cannot change: $known' = known$ and $birthday' = birthday$. This means that Δ and Ξ schemas have the same internal structure, allowing them to be combined easily.

The equation

$$date! = birthday(name?)$$

defines the output variable $date!$ in terms of the input variable $name?$ and the state variable *birthday*.

The schema *FindBirthday* is subject to the precondition $name? \in known$: the name must have a birthday assigned. If it does not, $birthday(name?)$ is undefined. Several schemas for one operation, specifying different preconditions, can be combined to yield a more general operation; we can specify error situations separately.

Slide 506

More Schemas

Remind

$\Xi BirthdayBook$
$today? : DATE$
$cards! : \mathbb{P}\, NAME$

$cards! = \{\, n : known \mid birthday(n) = today? \,\}$

Remind is a sort of inverse to *FindBirthday*: it looks up the date *today*? in the state, returning the associated names as the set *cards*!. This set is specified to consist of all names in *known* whose birthday equals *today*?. We are not constrained to find the set of names by searching, as the formula may suggest; any implementation technique, such as hashing, is acceptable. (The variable is called *cards*! because it will hold the names of people you must send cards to.)

InitBirthdayBook

$BirthdayBook$

$known = \emptyset$

InitBirthdayBook is a schema to specify the initial state for *BirthdayBook*. This is an example of extending an existing schema with additional constraints, here $known = \emptyset$. Writing it in this way is more concise than writing out the *BirthdayBook* schema and including the additional equation.

The invariant, $known = \mathrm{dom}\, birthday$, is still present. Since *InitBirthdayBook* specifies $known = \emptyset$ we obtain $\mathrm{dom}\, birthday = \emptyset$. Therefore $birthday = \emptyset$; initially, no birthdays are recorded. (The empty set, $\emptyset$, is also the empty function.)

Slide 507

Success versus Failure

Success

$result! : REPORT$

$result! = ok$

Typical use: $AddBirthday \land Success$

We shall deal with exceptional situations by augmenting each operation to return a status report. The report can be *ok* or an error value such as *already_known*.

The trivial schema *Success* simply returns a report indicating success. It is useless by itself. But we can express a schema that combines *AddBirthday* with a success report by the conjunction $AddBirthday \land Success$. This denotes the schema whose state variables are those of the two schemas combined, and whose logical specifications are joined using $\land$. The new schema does everything that *AddBirthday* does, and also reports $result! = ok$.

Slide 508

Specifying Exceptional Cases

$$
\begin{array}{|l}
\hline
\textit{AlreadyKnown} \\
\Xi \textit{BirthdayBook} \\
\textit{name}? : \textit{NAME} \\
\textit{result}! : \textit{REPORT} \\
\hline
\textit{name}? \in \textit{known} \\
\textit{result}! = \textit{already\_known} \\
\hline
\end{array}
$$

The schema *AlreadyKnown* handles the case of attempting to add a birthday for a name already present. Its precondition, $name? \in known$, is the negation of *AddBirthday*'s. We use a Ξ schema to specify that the state does not change; instead, the output variable $result!$ receives the value *already_known*. We may interpret this as an error condition; Z (unlike VDM) has no built-in notion of exception.

A *robust* operation to add birthdays, which handles the error condition, can be defined to be a combination of the schemas presented above:

$$\textit{RobustAddBirthday} \mathrel{\widehat{=}} (\textit{AddBirthday} \wedge \textit{Success}) \vee \textit{AlreadyKnown}$$

If $name? \in known$ then the specified effect is $result! = already\_known$; otherwise it adds the birthday and yields $result! = ok$. Specifying an operation in pieces, as here, has many advantages over writing one huge specification that covers all error conditions. It is easier to read, easier to write, easier to extend and modify.

Spivey [16] goes on to define *RobustFindBirthday* in precisely the same manner. Finally he defines $\textit{RobustRemind} \mathrel{\widehat{=}} \textit{Remind} \wedge \textit{Success}$; since *Remind* has no precondition, all we must do is make it report success.

One problem with Z is understanding what a schema really means. At first, schemas were regarded as shorthand for long formulæ. Later it was decided that schemas required some kind of a formal semantics, and this has taken many years to get right. Intuitively, a schema abbreviates a formula of the form *precondition* implies *effects*, where *effects* contains all specified constraints on the final state and output variables.

Slide 509

More on Z

- Other schema operations
  - $\mathit{Schema}1 \fatsemi \mathit{Schema}2$
- Refining the design
- Tool support
- Related methods
  - object Z
  - the B method

Z contains many other means of building new schemas. For example, $\mathit{Schema}1 \fatsemi \mathit{Schema}2$ is intended to specify the effect of applying $\mathit{Schema}1$ followed by $\mathit{Schema}2$. It expands to a schema that equates $\mathit{Schema}1$'s final state variables with $\mathit{Schema}2$'s initial state variables, without specifying their actual values. (It does this using existential quantifiers.) Both schemas' input and output variables are gathered together to form the inputs and outputs of $\mathit{Schema}1 \fatsemi \mathit{Schema}2$. From the schema $\mathit{AddBirthday} \fatsemi \mathit{FindBirthDay}$ one can derive $\mathit{date}! = \mathit{date}?$. This illustrates Z's power and complexity — as with a programming language, one must use this power with care.

Refinement. Z does not supply a method of refining the specification into a design, but it can be used for this purpose. Spivey [16] describes how to write more concrete Z schemas for the birthday book that use arrays to implement the *birthday* function, and to show that a concrete type (here arrays) faithfully implements the abstract type (functions).

Tool support. Part of the effort of writing a Z specification is neat presentation. These lecture notes were produced with the help of `zed-csp.sty`, a LaTeX style file. More elaborate tools perform type checking and other simple consistency checks. Z is not directly concerned with theorem proving, but there has been some research into support for Z using theorem provers such as HOL and Isabelle. Commercial tools (suitably priced!) are available too.

Z has been under development for a long time, and the Z Standard is nearing maturity. But research is continuing; methods under development include Object Z and B.

Object Z [3] extends Z with object-oriented features. 'The main reason for this extension is to improve the clarity of large specifications through enhanced structuring.' Object-Z introduces a class structure with a private state schema, packaged together with the operations that may affect that state. This attacks the problem, also found in programming, that a global state can be modified by any operation anywhere.

The B method, developed by J.-R. Abrial, has been mentioned in a previous lecture. Sophisticated tools have been developed to support it.

Slide 601

Structural induction on lists

To show $\phi(l)$ for every list l, prove that

- $\phi([])$ is true *(base case)*
- whenever $\phi(l')$ is true, then so is $\phi(x :: l')$ for all x, l' *(induction step)*

$\phi([x_1, \ldots, x_n])$ **is true after** n **steps**

This lecture concerns proving theorems about ML programs. In general, proving programs correct is extremely difficult. It becomes simple if we restrict attention to terminating, purely functional code. We can regard such ML programs as mathematical functions and reason about them by induction.

We begin with an introduction to list induction (sometimes called structural induction). For an extended discussion of such material, please see Chapter 6 of my ML book [15].

Why is list induction sound? In other words, why do the base case and induction step together imply $\phi(l)$ for all l?

It suffices to show that we have $\phi([x_1, \ldots, x_n])$ for arbitrary length n and elements $x_1, \ldots, x_n$. The base case yields $\phi([])$. Applying the induction step to x_n and $[]$, we have $\phi([x_n])$. Applying the induction step to x_{n-1} and $[x_n]$, we have $\phi([x_{n-1}, x_n])$. Eventually we reach $\phi([x_1, \ldots, x_n])$.

Slide 602

No List Equals Its Own Tail

Prove $l \neq x :: l$ by structural induction on l

Base case: $[] \neq x :: []$ (obvious!)

Induction step: Show $y :: l \neq x :: y :: l$ by contradiction.

If $y :: l = x :: y :: l$ then $y = x$ and $l = y :: l$.

So $l = x :: l$

But induction hypothesis says $l \neq x :: l$ CONTRADICTION

A simple example of structural induction is to prove that no list equals its own tail: $l \neq x :: l$. The proof requires some obvious properties of lists:

- constructors are distinct, $[] \neq x :: l$
- constructors are injective, if $x :: l = x' :: l'$ then $x = x'$ and $l = l'$

These are sometimes called 'freeness' properties. Their analogues hold for any tree-like data structure. They express that there is only one way of taking the data structure apart.

Theorem. For every list l we have $l \neq x :: l$.

Proof By structural induction on the list l. The base case is $[] \neq x :: []$, which is immediate by freeness.

The induction step is to show $y :: l \neq x :: y :: l$ from the induction hypothesis $l \neq x :: l$. It suffices to assume $y :: l = x :: y :: l$ and derive a contradiction. By freeness we get $y = x$ and $l = y :: l$. Therefore $l = x :: l$, contradicting the induction hypothesis. □

If there were infinite lists, then the list $[1, 1, \dots]$ would equal its own tail. Infinite lists can be defined mathematically, but their induction principles are too weak to prove the theorem above. This is not surprising; the justification of structural induction is that each list is constructed in finitely many steps.

Slide 603

Append is Associative: the Base Case

```
fun app([], ys)    = ys
  | app(x::xs, ys) = x :: app(xs,ys)
```

Prove $app(app(xs, ys), zs) = app(xs, app(ys, zs))$

By induction on *xs*:

$app(app([], ys), zs) = app(ys, zs) = app([], app(ys, zs))$

Structural induction is often used to prove properties of recursive functions. A classic example is to prove that the append function is associative:

Theorem. For all lists *xs*, *ys* and *zs*, we have

$$app(app(xs, ys), zs) = app(xs, app(ys, zs)).$$

Proof By structural induction on the list *xs*.

The base case is $app(app([], ys), zs) = app([], app(ys, zs))$. It follows because

$$app(app([], ys), zs) = app(ys, zs) = app([], app(ys, zs)).$$

The induction step assumes

$$app(app(l, ys), zs) = app(l, app(ys, zs))$$

as the induction hypothesis and requires proving

$$app(app(x :: l, ys), zs) = app(x :: l, app(ys, zs)).$$

(Continued on next slide.)

Slide 604

Append: the Inductive Step

$$\begin{aligned} app(app(x :: l, ys), zs) &= app(x :: app(l, ys), zs) \\ &= x :: app(app(l, ys), zs) \\ &= x :: app(l, app(ys, zs)) \qquad \text{[IND HYP]} \\ &= app(x :: l, app(ys, zs)). \end{aligned}$$

Other steps by the definition of *app*

Simplify both sides. Substituting by the definition of *app*, the left side becomes

$$\begin{aligned} app(app(x :: l, ys), zs) &= app(x :: app(l, ys), zs) \\ &= x :: app(app(l, ys), zs) \\ &= x :: app(l, app(ys, zs)). \end{aligned}$$

The last step above used the induction hypothesis. The right side becomes

$$app(x :: l, app(ys, zs)) = x :: app(l, app(ys, zs)).$$

Both sides are equal. □

This sort of proof is often routine. The secret is to set up the induction properly, choosing the right induction formula and induction variable. Induction on *xs* opens up the recursive definition of *app*, which is recursive in its first argument. Induction on *ys* or *zs* would not open up the definition.

Showing that append is associative justifies replacing `(l1@l2)@l3` by `l1@(l2@l3)`, which we might do in order to make a program run faster. In a moment, we shall see some more demanding program proofs.

Exercise 10 Prove xs @ $[] = xs$ (in other words, $app(xs, []) = xs$) by structural induction.

Slide 605

Universal Quantifiers

$\forall x\, \phi(x)$ means $\phi(x)$ is true for all x

Infer it if $\phi(x)$ holds for ARBITRARY x

Use it to conclude $\phi(t)$ for any t

$m \times n = 0$: a statement about m and n

$\forall n\, [m \times n = 0]$: a statement about m

$\forall m\, n\, [m \times n = 0]$: a (false) *sentence*

The symbol $\forall$ is called the *universal quantifier*, and $\forall x\, \psi(x)$ means that $\psi(x)$ is true for all x. To prove $\forall x\, \psi(x)$, we must prove $\psi(x)$ for an arbitrary x, making no assumption about its value. If we know that $\forall x\, \psi(x)$ is true, then we have $\psi(t)$ for every term t.

For instance, we have proved $l \neq x :: l$ above. Since l and x are arbitrary in the proof, we may conclude the universally quantified formula $\forall x\, l\ l \neq x :: l$. This states that the inequality holds for all l and x. To use it later, we may replace l and x by suitable terms.

If we prove $y \times (x/y) = x$ under the assumption $y \neq 0$, then y is *not* arbitrary: we have assumed it to equal zero! So it is wrong to conclude $\forall x\, y\, [y \times (x/y) = x]$. (The Part 1b course *Logic and Proof* will revisit these matters.)

Contrast the three formulae on the slide:

- $m \times n = 0$ an assertion about a particular m and n. (From what we know about multiplication, either variable must equal zero.)

- $\forall n\, [m \times n = 0]$ is an assertion about m alone. (Here, by further reasoning, we can conclude $m = 0$.)

- $\forall m\, n\, [m \times n = 0]$ does not refer to any particular variables, and it is false: for example, $2 \times 6 = 12 \neq 0$.

Sometimes the induction formula must involve quantifiers, as we shall now see.

Slide 606

Generalizing an Induction Formula

```
fun nlength []      = 0
  | nlength (x::xs) = 1 + nlength xs

fun addlen (k, [ ])  = k
  | addlen (k, x::xs) = addlen (k+1, xs)
```

Want to show $addlen(0, l) = nlength(l)$ *Too weak*

Try $addlen(k, l) = k + nlength(l)$ *Too rigid: k must vary*

$\forall k\; addlen(k, l) = k + nlength(l)$ **Correct!**

Frequently we must strengthen an assertion before applying induction. Consider proving $addlen(0, l) = nlength(l)$. This cannot be a useful induction formula, however, as it says nothing about the role of argument k in *addlen*. Evaluation of $addlen(0, l)$ involves $addlen(k, l')$ for various positive integers k and lists l'.

The formula $addlen(k, l) = k + nlength(l)$ precisely describes the relationship between *addlen* and *nlength*, and putting $k = 0$ gives the result we require. But even this formula is not useful for induction. The induction step would get stuck:

$$addlen(k, x :: l') = addlen(k + 1, l').$$

The induction hypothesis $addlen(k, l') = k + nlength(l')$ could not be used. The problem is that k varies during the evaluation of $addlen(k, l)$. We have an induction hypothesis about k, but need one about $k + 1$.

The right induction formula is $\forall k\,[addlen(k, l) = k + nlength(l)]$. This formula states that $addlen(k, l) = k + nlength(l)$ holds for all values of k. Note that k is a bound variable; the formula asserts a property of l alone. For a given l, it asserts that *addlen* and *nlength* are in the correct relationship *for all k*.

As an induction hypothesis, the formula $\forall k\,[addlen(k, l) = k + nlength(l)]$ lets us replace k by anything we please, dropping the quantifier. Generally speaking, the induction formula can be universally quantified over all variables except the induction variable, making the induction hypothesis as flexible as possible. But the proofs below use only those quantifiers that are actually necessary.

Slide 607

Correctness of `addlen`

$\forall k \; addlen(k,l) = k + nlength(l)$

Base case: $addlen(k,[]) = k = k + 0 = k + nlength[]$

Induction step:

$$\begin{aligned} addlen(k, x :: l) &= addlen(k+1, l) \\ &= k + 1 + nlength(l) \qquad [\text{IND HYP}, k \mapsto k+1] \\ &= k + nlength(x :: l). \end{aligned}$$

Theorem. For every list xs, we have $addlen(0, xs) = nlength\, xs$.

Proof It follows by putting $k = 0$ in the formula below, which we prove by list induction on l:

$\forall k \, addlen(k,l) = k + nlength(l)$

The base case is $\forall k \, addlen(k,[]) = k + nlength([])$. To prove a universally quantified statement, we simply drop the quantifier. For all k we clearly have

$$addlen(k,[]) = k = k + 0 = k + nlength([]).$$

The induction step assumes $\forall k \, addlen(k,l') = k + nlength(l')$ for the induction hypothesis, and requires proving

$\forall k \, addlen(k, x :: l') = k + nlength(x :: l')$.

This is true (for all k) because

$$addlen(k, x :: l') = addlen(k+1, l') = k + 1 + nlength(l') = k + nlength(x :: l').$$

The crucial step above is to invoke the induction hypothesis with $k+1$ in place of k, getting $addlen(k+1, l') = (k+1) + nlength(l')$. We may do this because the hypothesis is universally quantified: it holds for all k. $\square$

Slide 608

An Induction Formula for Reverse

```
fun nrev  []    = []
  | nrev(x::xs) = (nrev xs) @ [x]

fun revApp ([], ys)    = ys
  | revApp (x::xs, ys) = revApp (xs, x::ys)
```

Want to show $revApp(xs, []) = nrev(xs)$ *Too weak*

Try $revApp(xs, ys) = nrev(xs) \mathbin{@} ys$ *Too rigid*

$\forall ys\ revApp(xs, ys) = nrev(xs) \mathbin{@} ys$ **Correct!**

Proving $revApp(xs, []) = nrev(xs)$ involves an inductive argument similar to the one we have just examined. The arguments of *revApp* vary, just as those of *addlen* do. We prove the quantified formula $\forall y\ revApp(xs, ys) = append(nrev\, xs, ys)$.

Slide 609

Correctness of `revApp`

Base case: $revApp([], ys) = ys = [] \mathbin{@} ys = nrev[] \mathbin{@} ys$

Induction step:

$$\begin{aligned} revApp(x :: xs, ys) &= revApp(xs, x :: ys) \\ &= nrev(xs) \mathbin{@} (x :: ys) \\ &\qquad [\text{IND HYP}, xs \mapsto x :: ys] \\ &= nrev(xs) \mathbin{@} [x] \mathbin{@} ys \\ &= nrev(x :: xs) \mathbin{@} ys. \end{aligned}$$

The details of this proof are in *ML for the Working Programmer* [15], pages 227–8. Recall that @ is the same function as *app*.

A similar induction principle applies to binary trees and other recursive datatypes. The equivalence between `inord` and `inorder`, for example, is proved just like the examples above. (Those functions were mentioned in *Foundations of Computer Science*.)

Exercise 11 Prove $nrev(xs \mathbin{@} ys) = nrev\, ys \mathbin{@} nrev\, xs$ by structural induction.

Exercise 12 Prove $nrev(nrev\, xs) = xs$ by structural induction. *Hint*: use the previous exercise as a lemma.

Exercise 13 Prove $take(xs, k) \mathbin{@} drop(xs, k) = xs$ for every integer k and list xs.

Slide 610

Other Examples

$$nlength(xs \mathbin{@} ys) = (nlength\, xs) + (nlength\, ys)$$

$$nrev(xs \mathbin{@} ys) = (nrev\, ys) \mathbin{@} (nrev\, xs)$$

$$nrev(nrev(xs)) = xs$$

$$xs \mathbin{@} [] = xs$$

$$(map\, f) \circ (map\, g) = map\ (f \circ g)$$

Correctness-preserving program transformations

What does all this have to do with Software Engineering?

Real engineering consists of proven, practical techniques backed up by theory. For software, we don't have enough useful theory to build systems with confidence. We can be confident that software *won't* work first time and count ourselves lucky if it can be got to work in time and on budget.

But here, we see properties established of a form of software: functional ML programs. We can be sure, for example, that *revApp* gives a correct method of implementing list reversal, which is specified by *nrev*. The equations shown on the slide above can also be proved easily. They tell us that certain changes to programs, such as replacing `xs@[]` by `xs`, are safe.

There is a close relationship between the inductive proofs of this lecture and the loop invariants of Lect. 2. Here is a simple example. Function *addlen* corresponds to an obvious `while` loop for counting a list's elements: while a list variable *lv* (initially, the whole list) is non-empty, add one to the counter *k* (initially, zero). The loop invariant is $k + \mathrm{length}(lv) = \mathrm{length}(l)$, where *l* is the original list. The reasoning needed to prove correctness of this loop is quite similar to the inductive proof of *addlen*.

Real programs in real languages (like C) are not easily amenable to this sort of proof. Many programs rely on hardware features in an uncontrolled way, and furthermore, are very large and complex. In the real world, program proving is still restricted to small library functions.

References

[1] William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, 1989.

[2] Dan Craigen, Susan Gerhart, and Ted Ralston. Case study: Darlington nuclear generating station. *IEEE Software*, pages 30–32, January 1994.

[3] R. Duke, P. King, G. A. Rose, and G. Smith. The Object-Z specification language: Version 1. Technical Report 91-1, Department of Computer Science, University of Queensland, St. Lucia 4072, Australia, April 1991.

[4] Marc Eisenstadt. My hairiest bug war stories. *Communications of the ACM*, 40(4):30–37, April 1997.

[5] J. S. Fitzgerald, P. G. Larsen, T. M. Brookes, and M. A. Green. Developing a security-critical system using formal and conventional methods. In Hinchey and Bowen [10], pages 333–356.

[6] Susan Gerhart, Dan Craigen, and Ted Ralston. Experience with formal methods in critical systems. *IEEE Software*, pages 21–28, January 1994.

[7] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, 1991.

[8] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[9] Anthony Hall. Using formal methods to develop an ATC information system. *IEEE Software*, 13(2):66–76, March 1996.

[10] Michael Hinchey and Jonathan P. Bowen, editors. *Applications of Formal Methods*. Prentice-Hall, 1995.

[11] Michael Hinchey and Jonathan P. Bowen, editors. *Industrial-Strength Formal Methods in Practice*. Springer, 1999.

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[13] C. A. R. Hoare. An axiomatic basis for computer programming. In C. A. R. Hoare and C. B. Jones, editors, *Essays in Computing Science*, pages 45–58. Prentice-Hall, 1989. Originally published in 1969.

[14] Steve Maguire. *Writing Solid Code*. Microsoft Press, 1993.

[15] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[16] J. M. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 4(1):40–50, January 1989.

[17] D. A. Turner. Functional programs as executable specifications. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 29–54. Prentice-Hall, 1985.

[18] Lauren Ruth Wiener. *Digital Woes: Why We Should Not Depend on Software*. Addison-Wesley, 1993.