

Towards correct programs

Semantics

- A *correct program* is not *just more* reliable
it is **reliable**.
- A *correct program* does not *rarely* goes wrong
it **cannot go wrong**.
- A *correct program* does not *almost* solve a problem
it **solves a problem**.
- “*The correct program should be the philosopher stone for the programmer, the pole star of his efforts.*” Andrew Cumming.

How to test correctness?

- Testing may not be used as evidence of correctness for any but the most trivial of programs.
- Tests are almost never exhaustive.
- Having lots of tests which give the right results may be *reassuring* but it can never be *convincing*.
- We should relying on **reasoning (proofs)**.
- The intellectual effort involved in proving the correctness of even the simplest programs is immense (expensive).
- Formal methods have a role to play in safety-critical systems (air traffic control, military systems, financial systems...)

Functional programming

- Logical analysis of functional programs is possible.
- It is possible to make *assertions* (correctness or properties) and prove them.
- For imperative programs (i.e. those written in languages such as C, C++, or Java...) it is harder (second part of the lecture).

Square

- `fun f 0 = 0`
`| f(n) = f(n-1)+2*n-1;`

`fun g n:int = n*n;`

Prove using induction that $f(n)$ is equivalent to $g(n)$.

- – *Basis case:* $n=0$
 $f(n) = f(0) = 0$
 $g(n) = g(0) = 0^2$
 So $f(n) = g(n)$ for $n = 0$.
- *Induction hypothesis:* Assume $f(n) = g(n)$ for an arbitrary and fixed n ($n \geq 0$).
- We prove that $f(n+1) = g(n+1)$.
 $f(n+1)$
 $= f(n) + 2 * (n+1) - 1$ for $n \geq 1$.
 $= n^2 + 2(n+1) - 1$ (induction hypothesis)
 $= n^2 + 2n + 2 - 1$
 $= n^2 + 2n + 1$
 $= (n+1)^2$
 $= g(n+1)$
- *Conclusion*

Lists - Reverse

- `fun reverse(L) = if (L=[]) then L`
`else if (tl(L)=[]) then L`
`else reverse(tl(L))@[hd(L)];`

Prove by induction that the *reverse* function does what we want it to do.

- – *Basis case:* $L = nil$ or $L = [x]$.
 If the list has zero or one element then it is its own reverse (handled by the base cases of the function).
- *Induction hypothesis:*
 Assume that *reverse* works for any list of length n ($n \geq 0$).
- We prove that *reverse* works for any list of length $n+1$.

Lists - Length

We label the elements of the list of length $n+1$ as $[a_1, a_2, a_3, \dots, a_n, a_{n+1}]$.

Then $hd(L) = a_1$ and $tl(L) = [a_2, a_3, \dots, a_n, a_{n+1}]$.

So $reverse(L)$
 $= reverse(tl(L))@[hd(L)]$ (definition of reverse)
 $= reverse([a_2, a_3, \dots, a_n, a_{n+1}])@[a_1]$ (expanding $tl(L)$ and $hd(L)$)
 $= [a_{n+1}, a_n, \dots, a_3, a_2]@[a_1]$ (by induction hypothesis)
 $= [a_{n+1}, a_n, \dots, a_3, a_2, a_1]$

Proving a property

Structural induction

- `fun length nil = 0`
`| length L = 1 + length(tl(L));`
 Prove that $length(L) \geq 0$ for all list L .

- Proof on the structure of the list L .

- *Basis case:* $L = nil$
 $length(nil) = 0$
 So $length(L) \geq 0$.

- *Induction Hypothesis:* Assume that $length(L') \geq 0$ for an arbitrary and fixed list L' .

- $L = x::L'$
 $length(x::L')$
 $= 1 + length(L')$
 But $length(L') \geq 0$ (induction hypothesis)
 So $length(x::L') \geq 1$
 So $length(x::L') \geq 0$

- *Conclusion*

Lists - Take and skip

```
• fun take(L) = if L=[] then []
                else hd(L)::skip(tl(L))
and
  skip(L) = if L=[] then []
            else take(tl(L));
```

Prove by induction that the *take* and *skip* functions do what we want them to do.

- *Basis cases:*
List of length 0:
 $\text{take}([]) = []$
 $\text{skip}([]) = []$
- *Induction hypothesis:*
Assume *take* and *skip* work on lists of length n ($n \geq 0$).
- We prove that *take* and *skip* work on lists of length $n + 1$.
Let $L = [s_1, s_2, s_3, \dots, s_n, s_{n+1}]$.
So $\text{tl}(L) = [s_2, s_3, \dots, s_n, s_{n+1}]$.

* **Part 1: Prove that *take* works.**

```
take(L)
= s1 :: skip(tl(L)) (by definition)
= s1 :: skip([s2, s3, ..., sn, sn+1])
```

If n is even,
 $= s_1 :: [s_3, s_5, \dots, s_{n-1}, s_{n+1}]$ (induction hypothesis)
 $= [s_1, s_3, s_5, \dots, s_{n-1}, s_{n+1}]$

If n is odd,
 $= s_1 :: [s_3, s_5, \dots, s_n]$ (induction hypothesis)
 $= [s_1, s_3, s_5, \dots, s_n]$

* **Part 2: Prove that *skip* works.**

```
skip(L)
= take(tl(L)) (by definition)
= take([s2, s3, ..., sn, sn+1])
```

If n is even,
 $= [s_2, s_4, \dots, s_n]$ (induction hypothesis)

If n is odd,
 $= [s_2, s_4, \dots, s_{n-1}, s_{n+1}]$ (induction hypothesis)

Imperative programming

Preconditions and Postconditions

- // PRECONDITION
PROGRAM CODE
// POSTCONDITION
– A precondition is the condition that is true before the code executes.
– A postcondition is the condition that is true if the precondition is true and the code executes completely.

• Examples:

```
// Pre: x < 0
y = x * -2;
// Post: x < 0 AND y > 0 AND y = -2x
```

```
// Pre: x >= 0
x = x % 5;
// Post: x' >= 0 AND x' <= 4
```

Assignment Rule

- Given:

```
// P
x = E;
// Q
```

– where x is a variable, E is an expression, P is a precondition and Q is a postcondition.

- The **Assignment rule** is the following:
Derive P by replacing all occurrences of x in Q with E .
NOTE: x might be represented as x' in the postcondition Q .
- The assignment rule is a means of reasoning backward from a goal back to the required starting conditions.

IF statement

- See handout.

Loop Invariant

- An **invariant** is a condition that is true before and after a statement executes.
- A **loop invariant** I is true in the following four locations:

```
// I
while (C)
{
    // I
    loop body
    // I
}
// I
```

- Example:

```
sum = 0; J = 0;      J      sum
while (J < N)      0      0
{
    J = J + 1;      1      1
    sum = sum + J;  2      1+2
}
                    3      1+2+3
                    4      1+2+3+4
```

$I = \{ \text{sum} = 1+2+\dots+J \}$

NOTE: When $J=0$, $1+\dots+J$ is vacuous.

WHILE rule

- // P
 $S1$;
while (C)
{
 $S2$;
}
// Q

where C is a Boolean expression, $S1$ and $S2$ contain assignment statements, P is a precondition, Q is a postcondition and I is the loop invariant.

- The **While** rule is the following:
 - 1. $\{P\} S1 \{I\}$
 - 2. $\{I\} S2 \{I\}$
 - 3. $I \text{ AND NOT } C \rightarrow Q$
 - 4. The loop terminates.

Limitations of formal reasoning

Can we prove everything?

- All mathematical truths CANNOT be determined by following a valid logical proof procedure.
- Kurt Goedel (1930) (Czech mathematician) proved this result known as **Incompleteness Theorem of Goedel**.

Turing's Halting problem

- Alan Turing (1930) proved the incompleteness of algorithmic reasoning.
- Turing posed the following problem (since called the Halting Problem):

It is **impossible** to write a computer program H that is passed a single parameter P , where P is itself a program, such that H always returns true if P halts and false if P does not.

Ideas of the proof

- Assume we could write such a program H . Then let $H(X)$ be a library routine callable by any other program.

- Consider the following program:

```
program Contrary;  
  
#include H;  
  
begin  
  if H(Contrary)  
  then  
    while true  
      { /*Infinite Loop */}  
    else  
      halt;  
end;
```

- If H claims that the program P passed as parameter terminates, it deliberately enters an infinite loop.
- If H claims that the program P passed as parameter will not halt, it immediately halts.