Lecture aims

1. To critique the difference between program proof and other forms of verification, such as testing.
2. To be able to prove the correctness of simple programs against their contracts.
3. To think about programming in a more systematic and mathematical manner, with the result of improving our programming skills.

Lecture plan

Introduction to reasoning about correctness

1. Start with video: `http://www.youtube.com/watch?v=H61d_OkIht4`
2. Nulka is a "soft-kill" decoy missile, which leads homing missiles away from ships to splash harmlessly in the water. Developed in Melbourne by BAE and DSTO.
3. Nulk's flight control unit software formally verified using SPARK (at the programming level), using Hoare logic (topic for next 3-4 lectures).
4. Now deployed on over 130 Australian, Canadian and United States warships, and is the most successful Australia defence export.

Introduction to reasoning about programs

1. Semantics vs syntax. Three major types of semantics:
   (a) denotational: translate programs into denotations; e.g. sets of legal traces; Dijkstra's weakest precondition semantics.
   (b) operational: describes how programs are executed directly; e.g. as an abstract machine.
   (c) axiomatic: describes programs by the axioms that hold over them; the meaning is what is provable; e.g. Hoare logic.
2. We have specifications and designs (with contracts), but we need to implement our code such that it conforms to its contract.
3. Testing, review, and "basic" static analysis are not enough — many applications require *proof* of correctness.
4. Binary search example (below) — where is the fault?

   First version of the binary search appeared in 1946. Not until 1962 that the first *correct* version appeared. Common version "proved" by John Bentley contains a fault!

   If the best and brightest computer scientists cannot produce a correct program, and faults can remain undetected in text books for decades, what hope do we have?

A small programming language

1. Language has:
   (a) Procedures.
   (b) Expressions: numbers, arithmetic, variables reference and array reference.
   (c) Boolean expressions
   (d) Statements: variable assignment, array assignment, "skip", sequencing, branching, iteration, and procedure calls.
2. A *program* is a list of procedures and a statement that calls those procedures.

Hoare logic

1. Hoare triples.
2. Inference rules.
3. Example: increment an integer (below) — can we prove this?
4. The rules:
   (a) Assignment axiom. Example (Inc program): $\{x \geq 0\}\ x := x + 1\ \{x > 0\}$.
   Note that the rule at first seems backwards, but going forwards does not work using incremented example above.
   (b) Consequence rule: Example: complete the Inc program proof. Additional example: $\{\text{true}\}\ x := 5\ \{x \geq 5\}$.
   (c) Sequential composition rule: Example: swap program (below).
   First, assume simultaneous assignment of x and y. This motivates the use of *auxiliary* variables and pre-state values.
   Then do sequential version, in which $t = x^{\sim}$ cannot be proved. This motivates the sequential composition rule. Example: complete the swap program.
   (d) Empty statement axiom. Example: see condition rule example.
   (e) Conditional rule. Example: conditional swap (below)
   (f) Iteration rule: Example: summation (below).
   Discuss the idea and challenge of loop invariants.
   Discuss the notion of partial correctness and termination. The following program is an example of

   $\{y \geq 0\}$
   while $y \geq 0$ **do**
   $x := 0;$
   end while
   $\{y \geq 0\}$

   Apply iteration rule to prove hypothesis: $\{y \geq 0\}\ x := 0\{y \geq 0\}$, which allows us to conclude:

$\{y \geq 0\}$
while $y \geq 0$ **do**
    $x := 0;$
end while
$\{y \geq 0 \wedge \neg y \geq 0\}$

which has a contradiction. The program does not achieve this because it does not terminate, but even if it did terminate, it couldn't achieve that postcondition.

(g) Loop invariants. Requires some creativity, but is an active research area, even in the department.

Properties of loop invariant I:

i. $P \Rightarrow I$: must be *weaker* than its precondition.
ii. $I \wedge \neg B \Rightarrow Q$: *stronger* than its postcondition.
iii. $\{B \wedge I\}\ S\ \{I\}$: loop re-establishes the invariant.

Of course, these assume the loop body is correct!

Some heuristics:

i. Loop invariants generally contain most of the variables from the loop condition, loop body, the precondition, and the postcondition (where we mean the precondition/-postcondition of the loop, not necessarily the entire program).
ii. Generally state a relationship between the variables.
iii. Generally contain a predicate similar to the postcondition.
iv. Hold even if the loop condition is false.

5. Array assignment rule: Example: $\{x = y \wedge a[x] = 0\}\ a[y] := 5\ \{a[x] = 0\}$.

   Initial attempt at rule: $\{P[E/a[NE]]\}\ a[NE] := E\ \{P\}$.

   But if $x = y$, then $a[x] = 5$, but we cannot tell from the program.

   Hoare's corrected rule: $\{P[a\{NE \rightarrow E\}/a]\}\ a[NE] := E\ \{P\}$

6. Procedure call. Example: invalid example below. Proper example: factorial call.

   Programs and calls must have the following properties:

   (a) The number of arguments in a procedure call must be equal to the number of parameters in the procedure.
   (b) The formal parameter names must be distinct from each other.
   (c) The arguments for value-result variables must be distinct from each other. It is this rule that prevents the problem outlined in the counterexample above — the procedure call would be illegal.
   (d) Value parameters cannot be changed in the procedure body.
   (e) No recursion.

   Look at recursive procedures and the induction rule.

Mechanisation

1. Annotate the program with the necessary predicates. Automatic for all except loop invariant.
2. Use a program to generate *verification conditions*.
3. Prove using theorem prover.
4. Left over involve user input (but can still be automated).
5. All dischared implies correct. One or more failed implies incorrect. One or more undischarged either way implies unknown.

Dijkstra's weakest precondition semantics

1. Weakest precondition: For statement S and postcondition R, the weakest precondition, written $wp(S, R)$, is the weakest predicate that establishes R if S is executed and terminates.
2. The transformation rules: show table.
3. Note that $\{wp(S, Q)\}\ S\ \{Q\}$.
4. To prove $\{P\}\ S\ \{Q\}$, need to just prove $P \rightarrow wp(S, Q)$.

```
int binarySearch(int [] list, int target)
{
  int low = 0;
  int high = len(list) - 1;
  int mid;

  while(low <= high) {
    mid = (low + high)/2;
    if (list[mid] < target) {
      low = mid + 1;
    }
    else if (list[mid] > target) {
      high = mid - 1;
    }
    else {
      return mid;
    }
  }
  return -1;
}
```

```
procedure Inc (X: in out Integer)
   Pre => (X >= 0),
   Post => (X > 0);
is begin
   X := X + 1;
end Inc;
```

```
procedure Swap (X, Y : in out Float)
with
    Post => (X = Y'Old and Y = X'Old);
is
    T : Float;
begin
    T := X;
    X := Y;
    Y := T;
end Swap;
```

```
1   procedure ConditionalSwap (X, Y : in out Float)
2   with
3       Post => (X >= Y);
4   is
5       T : Float;
6   begin
7       if X < Y then
8           T := X;
9           X := Y;
10          Y := T;
11      endif;
12  end Swap;
```

```
1   procedure Summation (N : in Integer; A : in Array; Sum : out Integer)
2   with
3     Pre => (N >= 0),
4     Post => (Sum = ∑_{j=0}^{N-1} A(J));
5   is begin
6      I := 0;
7      Sum := 0;
8      while I < N do
9         Sum := Sum + A[I];
10        I := I + 1;
11     done;
12  end Summation;
```

Assume:	$p(x, y) \mathrel{\widehat{=}} x := y + 1$		(1)
	$\{y + 1 = y + 1\}\ x := y + 1\ \{x = y + 1\}$	[Assignment axiom]	(2)
	$true\ \Rightarrow\ y + 1 = y + 1$	[Logical theorem]	(3)
From 2, 3:	$\{true\}\ x := y + 1\ \{x = y + 1\}$	[Consequence rule]	(4)
From 1, 4:	$\{true\}\ p(x, y)\ \{x = y + 1\}$	[Procedure rule]	(5)
From 5:	$\{true\}\ p(z, z)\ \{z = z + 1\}$	[Procedure rule]	(6)

```
procedure InsertionSort (A : in out Array)
with
  Post => (for all m in (0 .. A'Length-2) => A(J) < A(J + 1));
is begin


   while I /= N - 2 do



      J := I;


      while J /= 1 and A[J - 1] > A[J] do



          swap(A, J - 1, J);


          J := J - 1;


      done;


   done;


{for all m in (0 .. A'Length-2) => A(J) < A(J + 1)}
end Summation;
```

$\{P[E/x]\}\ x := E\ \{P\}$ (assignment axiom)

$\{P\}$ **skip** $\{P\}$ (empty statement axiom)

$$\frac{P' \Rightarrow P,\ Q \Rightarrow Q',\ \{P\}\ S\ \{Q\}}{\{P'\}\ S\ \{Q'\}}$$ (consequent rule)

$$\frac{\{P\}\ S_1\ \{R\},\ \ \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1;\ S_2\ \{Q\}}$$ (sequential composition rule)

$$\frac{\{P \wedge B\}\ S_1\ \{Q\},\ \ \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{endif}\ \{Q\}}$$ (conditional rule)

$$\frac{\{P \wedge B\}\ S\ \ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{\neg B \wedge P\}}$$ (iteration rule)

$$\frac{\{P\}\ S\ \{Q\}}{\{P[E_1/v_1, E_n/v_n]\}\ p(E_1, \ldots, E_n)\ \{Q[E_1/v_1, \ldots, E_n/v_n]\}}$$ (procedure call rule)

$\{P[a\{NE \rightarrow E\}/a]\}\ a[NE] := E\ \{P\}$ (array assignment rule)

Dijkstra's predicate transformers

Rule	**Input**	**Output**
Skip	$wp(\textbf{skip}, R)$	R
Assignment	$wp(x := E, R)$	$R[E/x]$
Sequence	$wp(S_1;\ S_2, R)$	$wp(S_1, wp(S_2, R))$
Conditional	$wp(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ endif}, R)$	$B \rightarrow wp(S_1, R) \wedge \neg B \rightarrow wp(S_2, R)$
While loop	$wp(\textbf{while } B \textbf{ do } S \textbf{ done})$	$\exists\, k.(k \geq 0 \wedge P_k)$ where $P_0 \equiv \neg B \wedge Q$ $P_{k+1} \equiv B \wedge wp(S, P_k)$