1 Predicate Logic & Correctness

- Propositional operators
    - not ($\neg$), and ($\wedge$), or ($\vee$)
    - implication ($\Rightarrow$) – $P \Rightarrow Q \equiv \neg P \vee Q$
    - equivalence ($\Leftrightarrow$) – $P \Leftrightarrow Q \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P)$
- Operator precedence (tightest-binding first): $\neg$, $\wedge$, $\vee$, $\Rightarrow$
- Quantifiers
    - for all ($\forall$)
    - there exists ($\exists$)
    - Example: $\forall\, x \in \mathbb{N} \cdot x \geq 0$
    - The type of the bound variable may be implicit, e.g. $\forall\, x \cdot x \geq 0$
- Entailment
    - If $P \Rightarrow Q$ is a tautology (always true) then P is stronger than Q
    - Equivalently: P entails Q ($P \Rrightarrow Q$)
- Substitution
    - $P[x\backslash a]$ – Substitute all occurrences of x by a in P
    - $P[x, y\backslash a, b]$ – Substitute a and b for x and y **simultaneously**
- Hoare triples – $\{P\}\, S\, \{Q\}$
    - P is the precondition, S is the program and Q is the postcondition
    - If P is true before S executes, then S will terminate and Q will be true when it does
    - The program must terminate if started in States_P (total correctness)
- Weakest preconditions
    - For a program S and postcondition Q, $wp(S, Q)$ is the unique weakest possible precondition such that the triple $\{P\}\, S\, \{Q\}$ will be true
    - $\forall\, P \cdot (\{P\}\, S\, \{Q\}) \Rightarrow (P \Rrightarrow wp(S, Q))$

2   Guarded Command Language

- **skip** – Empty Command
  - $wp(\textbf{skip}, Q) \equiv Q$
  - Hence $\{Q\}\,\textbf{skip}\,\{Q\} \equiv$ true for any Q
  - $\{P\}\,\textbf{skip}\,\{Q\}$ is false if and only if P is strictly weaker than Q
- **abort** – Chaotic Command
  - $wp(\textbf{abort}, Q) \equiv$ false
  - No precondition can guarantee a postcondition
  - Represents 'chaotic/undefined behaviour'
- **:=** – Assignment
  - $wp(x := E, Q) \equiv Q[x\backslash E]$
  - So $\{Q[x\backslash E]\}\,x := E\,\{Q\}$ is true
  - Generalises to multiple assignment: $wp((x, y := E, F), Q) \equiv Q[x, y\backslash E, F]$
- **;** – Composition/Concatenation
  - $wp(S_1;\ S_2, Q) \equiv wp(S_1, wp(S_2, Q))$
  - There exists some 'middle' predicate true after S_1 and before S_2
  - If $\{P\}\,S_1\,\{M\} \wedge \{M\}\,S_2\,\{Q\}$ then $\{P\}\,S_1;\ S_2\,\{Q\}$
- **if** – Selection
  - **if** $G_1 \to S_1$
    $[\!]\ \ G_2 \to S_2$
    . . .
    $[\!]\ \ G_n \to S_n$
    fi
  - Evaluate all guards $G_1 \ldots G_n$, choose a true guard G_i nondeterministically, and execute S_i
  - if all guards evaluate to false then **abort** is executed
  - $wp(\textbf{if}, Q) \equiv \bigvee_{i=1}^{n} G_i \wedge \bigwedge_{i=1}^{n}(G_i \Rightarrow wp(S_i), Q)$
  - The disjunction of the guards *must* be true
- **do** – Repetition
  - **do** $G_1 \to S_1$
    $[\!]\ \ G_2 \to S_2$
    . . .
    $[\!]\ \ G_n \to S_n$
    od
  - Evaluate all guards, choose a true guard nondeterministically, execute S_i and repeat
  - If all G_i are false, the loop ends and the program continues
  - Weakest precondition rule is complex, so *loop invariants* are used instead
  - Predicate I is a loop invariant if $\{I \wedge G_i\}\,S_i\,\{I\}$ for all $1 \le i \le n$
  - There are usually many possible invariants for a loop

3   Refinement & Verification

- Specification statement: $w : [P, Q]$

- P is the precondition, Q is the postcondition, w is the 'frame' of variables that may be modified

- A program C satisfies $w : [P, Q]$ if and only if

  - $\{P\}\, C\, \{Q\}$

  - C only changes variables in w

- If P is not true when C is executed it may do anything, and it need not terminate

- Mixing specification statements with GCL forms a 'wide-spectrum language'

- Refinement ($\sqsubseteq$) – a partial ordering on programs (similar to $\leq$ for reals)

  - $S \sqsubseteq S'$ means a user expecting program S would be satisfied with S'

  - $S \sqsubseteq S' \Leftrightarrow \forall\, Q \cdot wp(S, Q) \Rightarrow wp(S', Q)$

  - For a specification: $wp(x : [P, Q], Q') \mathrel{\widehat{=}} P \wedge (\forall\, x \cdot Q \Rightarrow Q')[v_0 \backslash v]$

- General approach to refining a program

  - Start with a specification $S = w : [P, Q]$

  - Use rules to replace S with S' mixing specifications with GCL

  - Each rule must preserve correctness – i.e. every program C that satisfies S' must satisfy S

  - Eventually arrive at a pure GCL program C such that $\{P\}\, C\, \{Q\} \equiv \text{true}$

3.1 Refinement Rules

- Rule 1: **Strengthen Postcondition**
  - If $P[w \backslash w_0] \wedge Q' \Rrightarrow Q$ then $w : [P, Q] \sqsubseteq w : [P, Q']$ ($P[w \backslash w_0]$ usually not needed)
- Rule 2: **Weaken Precondition**
  - If $P \Rrightarrow P'$ then $w : [P, Q] \sqsubseteq w : [P', Q]$
- Rule 3: **Skip**
  - If $P \Rrightarrow Q$ then $w : [P, Q] \sqsubseteq \mathbf{skip}$
- Rule 4: **Assignment**
  - If $P \Rrightarrow Q[x \backslash E]$ then $x : [P, Q] \sqsubseteq x := E$
- Rule 5: **Composition**
  - $w : [P, Q] \sqsubseteq w : [P, M]; \ w : [M, Q]$ (no side condition)
- Rule 6: **Following Assignment** (combined assignment and composition)
  - $w, x : [P, Q] \sqsubseteq w, x : [P, Q[x \backslash E]]; \ x := E$
- Rule 7: **Selection**
  - If $P \Rrightarrow \bigvee_{i=1}^{n} G_i$ then $w : [P, Q] \sqsubseteq$
    $\mathbf{if} \ \ G_1 \rightarrow w : [G_1 \wedge P, Q]$
    $\ldots$
    $[\!]\ \ G_n \rightarrow w : [G_n \wedge P, Q]$
    $\mathbf{fi}$
- Rule 8: **Repetition**
  - For repetition, a loop invariant I and loop variant (an integer expression) V are required
    * Let V_0 be the value of V at the start of each iteration
    * Then $0 \le V < V_0$ is true at the end of each iteration
    * (i.e. V is *strictly decreasing* on every iteration, and won't be negative before loop termination)
  - To apply the repetition rule
    1. Strengthen postcondition to $I \wedge \neg G$ (side condition: $I \wedge \neg G \Rrightarrow Q$)
    2. Use composition to perform $w : [P, I \wedge \neg G] \sqsubseteq w : [P, I]; \ w : [I, I \wedge \neg G]$
    3. Refine the first half into initialisation (e.g. an assignment)
    4. Refine the second half using the repetition rule (no side conditions!)
  - Rule: Let $G \ \widehat{=} \ \bigvee_{i=1}^{n} G_i$, then $w : [I, I \wedge \neg G] \sqsubseteq$
    $\mathbf{do} \ G_1 \rightarrow w : [I \wedge G_1, I \wedge (0 \le V < V_0)]$
    $\ldots$
    $[\!]\ \ G_n \rightarrow w : [I \wedge G_n, I \wedge (0 \le V < V_0)]$
    $\mathbf{od}$
- Rule 9: **Contract frame**
  - $w, x : [P, Q] \sqsubseteq w : [P, Q[x_0 \backslash x]]$
- Rule 10: **Remove invariant**
  - If w does not occur in I then $w : [P \wedge I, Q \wedge I] \sqsubseteq w : [P, Q]$

4   Arrays

- If A is an array, then A.len is the number of elements in A
- A_i is the zero-indexed i^{th} element of A if $0 \leq i < A$.len
  - If i is outside of $[0, A$.len$)$ then A_i is undefined
- $A_{[i,j)}$ is the subarray from containing the elements from A_i to A_{j-1}
  - $A_{[i,i)}$ is the empty array $[]$
  - If $i > j$ or $i < 0$ or $j > A$.len then the subarray is undefined

5   Derivation

- Deriving a loop based program: follow the strategy for repetition
  - Strengthen postcondition, use composition, assignment rule for initialisation, repetition rule
- Patterns for finding an invariant when deriving a loop-based program
  - **Pattern 1**: Given postcondition $Q \mathrel{\widehat{=}} Q_1 \wedge Q_2$, let Q_1 be the invariant and Q_2 be the negation of the guard
    * Use this pattern if the postcondition consists of conjunct conditions and one looks like a useful negation of the guard
    * If there is only one condition, remember the invariant could always just be true
  - **Pattern 2**: Given postcondition Q which uses some constant N, replace N with a variable x, and let the negation of the guard be $x = N$
    * Use this pattern to create an iterator variable x when there is something clear to iterate over
    * This is commonly used for array-based programs
    * In an array-based program with no obvious constants, remember that $A = A_{[0, A.\text{len})}$
  - Note that often these will just be 'starting' invariants that may require strengthening

6   Procedures

- A procedure is a named block of code used for structure and to enable reuse

- Given **procedure** $R() \mathrel{\widehat{=}} S$ and $w : [P, Q] \sqsubseteq S$, we have that $w : [P, Q] \sqsubseteq R$

- The *formal* parameter is used in the function and the *actual* parameter is what's passed in

- Parameter types

  - **value**

    * Sets the formal parameter to the value of a variable or expression when the procedure runs

    * Modifying the formal parameter in the procedure doesn't affect the actual parameter

    * Given **procedure** $R(\textbf{value } z) \mathrel{\widehat{=}} S$ and $w, z : [P, Q] \sqsubseteq S$:
      $w : [P[z \backslash a], Q[z_0 \backslash a_0]] \sqsubseteq R(a)$ where $a_0 = a[w \backslash w_0]$

    * The postcondition Q should not contain z since it is local to R

  - **result**

    * The actual parameter takes the value of the formal parameter when the procedure terminates

    * The actual parameter must be a variable, not an expression and its initial value is not defined

    * Given **procedure** $R(\textbf{result } z) \mathrel{\widehat{=}} S$ and $w, z :\sqsubseteq S$:
      $w : [P, Q[z \backslash a]] \sqsubseteq R(a)$

    * The precondition P should not contain z, and the postcondition Q should not contain z_0

  - **value result**

    * The formal parameter takes the value of the actual parameter when the procedure starts

    * The actual parameter takes the value of the formal parameter when the procedure terminates

    * Given **procedure** $R(\textbf{value result } z) \mathrel{\widehat{=}} S$ and $w, z : [P, Q] \sqsubseteq S$:
      $w, a : [P[z \backslash a], Q[z_0, z \backslash a_0, a]] \sqsubseteq R(a)$

    * There are no constraints on how z and z_0 may appear in P and Q

  - A procedure may have multiple parameters of different types

    * e.g. **procedure** $R(\textbf{result } x, y; \textbf{ value } z) \mathrel{\widehat{=}} x, y := 0, z + 1$

- Introducing procedures and procedure calls when refining

  1. Identify a suitable specification $x, y, z : [P, Q]$ and choose a name R

  2. Identify parameters and their types

     - Variables in P only are likely **value** parameters

     - Variables in Q only are likely **result** parameters

     - Variables in both are likely **value result** parameters

     procedure $R(\textbf{value } x; \textbf{ result } y; \textbf{ value result } z) \mathrel{\widehat{=}} x, y, z : [P, Q]$

  3. If the formal parameter appears only in the precondition, use a **value** parameter

  4. Refine the body of R to code

  5. Refine the main program with variables a, b, c to the specification:
     $b, c : [P[x, z \backslash a, c], Q[x_0, y, z_0, z \backslash a_0, b, c_0, c]]$

  6. Replace the above specification with $R(a, b, c)$

6.1   Recursion

- Procedures may be called recursively (i.e. from within themselves), as per any procedure call

- Need to use a variant V to ensure the recursion reaches a base case

  - The variant may refer to variables including parameters (aside from **result** parameters)

  - Given procedure R with specification $w : [P, Q]$, let $V = N$ when the procedure is first called

  - Then to refine to a call outside the function, R has specification $w : [P \wedge (V = N), Q]$

  - To refine to a recursive call within R, we require the specification $w : [P \wedge (0 \leq V < N), Q]$

Example: **procedure** $Factorial(\textbf{value } n, \textbf{result } f) \ \widehat{=}\ n : f[n \geq 0, f = n_0!]$.

Let n be the variant and introduce $n = N$ into the precondition (for the initial call).
$n, f : [n \geq 0 \wedge n = N, f = n_0!]$
$\sqsubseteq \{\text{Selection: } n \geq 0 \wedge n = N \Rrightarrow n = 0 \vee n > 0\}$
if $n = 0 \to n, f : [n \geq 0 \wedge n = 0 \wedge n = N, f = n_0!]$
$[]\ \ n > 0 \to n, f : [n \geq 0 \wedge n > 0 \wedge n = N, f = n_0!]$
fi
$\sqsubseteq \{\text{Assignment: } n \geq 0 \wedge n = 0 \wedge n = N \Rrightarrow (f = n_0!)[f\backslash 1]\}$
if $n = 0 \to f := 1$
$[]\ \ n > 0 \to n, f : [n \geq 0 \wedge n > 0 \wedge n = N, f = n_0!]$
fi

The remaining specification $n, f : [n \geq 0 \wedge n > 0 \wedge n = N, f = n_0!]$ is refined as follows: $n, f : [n \geq 0 \wedge n > 0 \wedge n = N, f = n_0!]$
$\sqsubseteq \{\text{Contract frame: } n\}$
$f : [n \geq 0 \wedge n > 0 \wedge n = N, f = n!]$
$\sqsubseteq \{\text{Following assignment: } f := f \times n\}$
$f : [n \geq 0 \wedge n > 0 \wedge n = N, (f = n!)[f\backslash f \times n]];\ f := f \times n$

The remaining specification $f : [n \geq 0 \wedge n > 0 \wedge n = N, (f \times n) = n!]$ is refined into a recursive call:
$f : [n \geq 0 \wedge n > 0 \wedge n = N, (f \times n) = n!]$
$\sqsubseteq \{\text{Apply substitution}\}$
$f : [n \geq 0 \wedge n > 0 \wedge n = N, (f \times n) = n!]$
$\sqsubseteq \{\text{Divide both sides of } f \times n = n! \text{ by } n\}$
$f : [n \geq 0 \wedge n > 0 \wedge n = N, f = (n - 1)!]$
$\sqsubseteq \{\text{Weaken precondition: } n \geq 0 \wedge n > 0 \wedge n = N \Rrightarrow n - 1 \geq 0 \wedge (0 \leq n - 1 < N)\}$
$f : [n - 1 \geq 0 \wedge (0 \leq n - 1 < N), f = (n - 1)!]$
$\sqsubseteq \{\text{Apply substitution backwards}\}$
$f : [(n \geq 0 \wedge (0 \leq n < N))[n\backslash n - 1], (f = n!)[n_0, f\backslash n_0 - 1, f]]$
$\sqsubseteq \{\text{Introduce recursive call with } \textbf{value} \text{ parameter } n \text{ and } \textbf{result} \text{ parameter } f\}$
$Factorial(n - 1, f)$

This produces the final program:
if $n = 0 \to f := 1$
$[]\ \ n > 0 \to Factorial(n - 1, f);\ f := f \times n$
fi

7   Modules