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
  - $wp(\mathbf{skip}, Q) \equiv Q$
  - Hence $\{Q\}\,\mathbf{skip}\,\{Q\} \equiv$ true for any Q
  - $\{P\}\,\mathbf{skip}\,\{Q\}$ is false if and only if P is strictly weaker than Q
- **abort** – Chaotic Command
  - $wp(\mathbf{abort}, Q) \equiv$ 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** $G_1 \rightarrow S_1$
$[]\ \ G_2 \rightarrow S_2$
...
$[]\ \ G_n \rightarrow S_n$
fi
  - Evaluate all guards $G_1 \dots G_n$, choose a true guard G_i nondeterministically, and execute S_i
  - if all guards evaluate to false then **abort** is executed
  - $wp(\mathbf{if}, Q) \equiv \bigvee_{i=1}^{n} G_i \wedge \bigwedge_{i=1}^{n} (G_i \Rightarrow wp(S_i), Q)$
  - The disjunction of the guards *must* be true
- **do** – Repetition
  - **do** $G_1 \rightarrow S_1$
$[]\ \ G_2 \rightarrow S_2$
...
$[]\ \ G_n \rightarrow S_n$
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 $1 \leq i \leq n$
  - 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'
  - $S \sqsubseteq S' \Leftrightarrow \forall\, Q \cdot wp(S, Q) \Rrightarrow wp(S', Q)$
  - 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 $\{P\}\, C\, \{Q\} \equiv$ 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 $w : [P, Q] \sqsubseteq$ **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 $w : [P, Q] \sqsubseteq$
    if $G_1 \rightarrow w : [G_1 \wedge P, Q]$
    $\ldots$
    $[\!]\ G_n \rightarrow w : [G_n \wedge P, Q]$
    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 $0 \leq V < V_0$ 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 $G \mathrel{\widehat{=}} \bigvee_{i=1}^{n} G_i$, then $w : [I, I \wedge \neg\, G] \sqsubseteq$
    do $G_1 \rightarrow w : [I \wedge G_1, I \wedge (0 \leq V < V_0)]$
    $\ldots$
    $[\!]\ G_n \rightarrow w : [I \wedge G_n, I \wedge (0 \leq V < V_0)]$
    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 $0 \leq i < A.\text{len}$
  - If i is outside of $[0, A.\text{len})$ 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 $j > A.\text{len}$ 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** $\mathit{Factorial}(\textbf{value } n, \textbf{result } f) \mathrel{\widehat=} n : f[n \geq 0, f = n_0!]$.

Let n be the variant and introduce $n = N$ into the precondition (for the initial call).
$n, f : [n \geq 0 \wedge n = N, f = n_0!]$
$\sqsubseteq$ {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
$\sqsubseteq$ {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: $n, f : [n \geq 0 \wedge n > 0 \wedge n = N, f = n_0!]$
$\sqsubseteq$ {Contract frame: n}
$f : [n \geq 0 \wedge n > 0 \wedge n = N, f = n!]$
$\sqsubseteq$ {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:
$f : [n \geq 0 \wedge n > 0 \wedge n = N, (f \times n) = n!]$
$\sqsubseteq$ {Apply substitution}
$f : [n \geq 0 \wedge n > 0 \wedge n = N, (f \times n) = n!]$
$\sqsubseteq$ {Divide both sides of $f \times n = n!$ by n}
$f : [n \geq 0 \wedge n > 0 \wedge n = N, f = (n-1)!]$
$\sqsubseteq$ {Weaken precondition: $n \geq 0 \wedge n > 0 \wedge n = N \Rrightarrow n - 1 \geq 0 \wedge (0 \leq n - 1 < N)$}
$f : [n - 1 \geq 0 \wedge (0 \leq n - 1 < N), f = (n-1)!]$
$\sqsubseteq$ {Apply substitution backwards}
$f : [(n \geq 0 \wedge (0 \leq n < N))[n \backslash n - 1], (f = n!)[n_0, f \backslash n_0 - 1, f]]$
$\sqsubseteq$ {Introduce recursive call with **value** parameter n and **result** parameter f}
$\mathit{Factorial}(n-1, f)$

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

7 Modules

- Modules provide a way to store data structures and procedures
- Example:
  module *UniqueNumberAllocator*
      export *Acquire*, *Reset*
      import *Choose*

      var u : **set** $[0, N)$

      procedure *Acquire*(**result** t) $\widehat{=}$
          $Choose([0, N) - \text{u} \ , t);\ u := u \cup \{t\}]$

      procedure *Reset*() $\widehat{=}$ $u := \{\}$

      procedure *Choose*(**value** s; **result** e) $\widehat{=}$ $e : [s \neq \{\}, e \in s]$

      initially $u = \{\}$
  End
- Syntax
  - Modules are declared with **module** and have a unique name
  - Module-level variables are declared in the **var** clause and given a type
  - The initial condition of module variables is given the predicate in the **initially** clause
  - Modules may define procedures which make use of its variables
  - Modules list which procedures are exported publicly with an **export** clause (if there is no **export** clause, all procedures are exported)
  - The variables and procedures of another module may be used if they are included in the **import** clause
    - Imported variables must be redeclared exactly as in their source module
    - Imported procedures must be redeclared: the original declaration must refine the redeclaration
    - Imported procedures cannot refer to the local variables of the module they are imported into
    - Circular import/export is not well defined

7.1 Module Refinement

- A module M' refines some module M with exported procedures E, imported procedures I and initialisation condition *init* when
  - M' has the same local and imported variables as M
  - The exported procedures E' refine those in E (there may be more procedures in E', but not fewer)
  - The imported procedures I' refine those in I (there may be fewer procedures in I' but not more)
  - The initialisation $init'$ is stronger than $init$ – i.e. $init' \Rrightarrow init$
- To refine modules with different variables, data refinement is required

7.2 Data Refinement

- The local state of a module cannot be accessed from the outside, so it may be changed provided the difference cannot be detected by use of the exported procedures
- Rule 1: **Introducing new variables**
  - Relationships between new and existing variables are maintained via a **coupling invariant** CI
    - E.g. $CI \mathrel{\widehat{=}} p = q + r$
  - The initialisation $init$ becomes $init \wedge CI$
    - E.g. if initialisation was $p = 1$, it would become $p = 1 \wedge p = q + r$
  - Any specification $w : [P, Q]$ becomes $w, c : [P \wedge CI, Q \wedge CI]$ where c is the list of new variables
    - E.g. $p : [p > 0, p < p_0]$ becomes $p, q, r : [p > 0 \wedge p = q + r, p < p_0 \wedge p = q + r]$
  - Every assignment in the module $w := E$ becomes $w, c := E, F$ provided that $CI \Rrightarrow CI[w, c \backslash E, F]$
    - E.g. $p := p + 1$ becomes $p, q := p + 1, q + 1$, or alternately $p, r := p + 1, r + 1$
  - Every guard in the module G becomes G' provided that $CI \Rrightarrow (G \Leftrightarrow G')$
    - $G' \mathrel{\widehat{=}} CI \wedge G$ is always suitable
    - E.g. the guard $p > 0$ could become $p > 0 \wedge p = q + r$, or alternately $p = q + r \Rightarrow p > 0$
- Rule 2: **Removing an existing variable**
  - Only **auxiliary variables** may be removed, i.e. they must only appear in:
    - Assignments
    - Specifications which modify only auxiliary variables
  - The initialisation $init$ becomes $\exists\, a \cdot init$ where a is the auxiliary variable
    - The 'one-point rule' may be used to remove the existential quantifier: $\exists\, x \cdot P \wedge x = n \equiv P[x \backslash n]$
    - E.g. given initialisation $p = 1 \wedge p = q + r$, it would become $\exists\, p \cdot p = 1 \wedge p = q + r \equiv q + r = 1$
  - All specifications $w, a : [P, Q]$ become $w : [\exists\, a \cdot P, \forall\, a_0 \cdot P[w, a \backslash w_0, a_0] \Rightarrow (\exists\, a \cdot Q)]$
    - A similar one-point rule may be used to remove the universal quantifier:
      $\forall\, x \cdot P \wedge x = n \Rightarrow Q \equiv P[x \backslash n] \Rightarrow Q[x \backslash n]$
    - E.g. $p, q, r : [p > 0 \wedge p = q + r, p < p_0 \wedge p = q + r]$ can be refined to:
      $q, r : [\exists\, p \cdot p > 0 \wedge p = q + r, \forall\, p_0 \cdot p_0 > 0 \wedge p_0 = q_0 + r_0 \Rightarrow (\exists\, p \cdot p < p_0 \wedge p = q + r)]$
      $\sqsubseteq$ {Apply $\exists$ one-point rule to pre and postconditions}
      $q, r : [q + r > 0, \forall\, p_0 \cdot p_0 > 0 \wedge p_0 = q_0 + r_0 \Rightarrow q + r < p_0]$
      $\sqsubseteq$ {Apply $\forall$ one-point rule to postcondition}
      $q, r : [q + r > 0, q_0 + r_0 > 0 \Rightarrow q + r < q_0 + r_0]$
  - Any assignment $w, a := E, F$ where E contains no variables from a can be replaced by $w := E$
    - E.g. $p, q := p + 1, q + 1$ can be replaced by $q := q + 1$
  - Normally the coupling invariant CI relates each concrete state to a unique abstract state (e.g. $p = q + r$), in which case the following rule applies:
    - Given abstract variables a and concrete variables c, if $CI \mathrel{\widehat{=}} a = f(c) \wedge P(c)$ then a guard G may be replaced by $G[a \backslash f(c)] \wedge P(c)$, or simply by $G[a \backslash f(c)]$
    - E.g. $p > 0 \wedge p = q + r$ can be replaced by $q + r > 0 \wedge q + r = q + r \equiv q + r > 0$