**Program Correctness – Arrays, Functions, Procedures**

* Array – a function mapping indices to values
  + Indices = {i . i : N | 0 ≤ i < arraySize }
  + B : Indices → Value
  + B[i] – refers to the ith value in the array
* Arrays used on the RHS – reference of value in array
  + Use assignment rule:

assert(P[B[x]/Var])

Var := B[x];

assert(P); % asn

* Arrays used on the LHS – assignment of value to array
  + i.e. one pair in the array is overridden
  + When B is a function and S contains only one pair:
    - (B ⊕ {(i, e)})[k]
      * = e if i = k
      * = B[k] if ¬(i = k)
    - In proofs:

3) i = j

4) (B ⊕ {(i, e)})[j] = e by set % defn of override, 3

* + Array assignment rule:

assert(P[B ⊕ {(i, e)}/B]);

B[i] = e;

assert(P); % array asn

* Vacuously true – a universal quantification where no values satisfy the premise of the implication
  + e.g. ∀k . k < 0 ∧ k > 0 ⇒ P
  + There are no values of k that satisfy the conditions of the quantification
* Logical variable introduction
  + For any references to variable w, a logical variable w0 can be created to record the value of w at any point in program execution
  + The new logical variable must not have already been used in the program
  + i.e.

P(w)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ log var intro

P(w0) ∧ w = w0

* Functions
  + Formal parameters – identifiers that represent the values passed to a function/procedure
    - Used in function/procedure declaration
  + Actual parameters – values that are passed to a function/procedure
    - Used in function/procedure call
  + Functions do not change parameter values or global variables, only local variables
  + Functions – annotated program:

fun f(x1, x2) { % x1, x2 = formal parameters

assert(Pre: R); % precondition (in terms of x1, x2, ret)

var ret;

C; % body of function

return ret;

assert(Post: S); % postcondition (in terms of x1, x2, ret)­

};

…

assert(P);

y := f(a1, a2); % a1, a2 = actual parameters

assert(Q); % derived asn (VC 1)

Function rule: for the function call f(a1, a2), the function rule gives us:

Lemma 1: (R ⇒ S)[a1/x1, a2/x2, f(a1, a2)/ret]

% VC 1

P |− Q[f(a1, a2)/y]

* + - Use the lemma to prove the VC for derived asn
    - Ex:

R → x ≥ 0

S → ret = x + y

d := f(b, c);

Lemma 1: b ≥ 0 ⇒ f(b, c) = b + c

* Procedures
  + Procedures can change the values of global variables
  + Procedures – annotated program:

proc p(x1, x2) {

assert(Pre: R);

C:

assert(Post: S):

}

…

assert(P);

assert(H ∧ R[a1/x1, a2/x2]): % implied (VC 1)

p(a1, a2);

assert(H ∧ S[a1/x1, a2/x2]); % procedure

assert(Q); % implied (VC 2)

% VC …

* + - H must not contain any variables changed by p
    - H may contain actual parameters; H may be nothing (i.e. true)
    - Choose H based on P & Q (to make their VC’s easier to prove)
    - Ex:

R → x = x0

S → x > x0

…

assert(q = y);

assert(q0 = y ∧ q = q0); % log var intro

proc(q);

assert(q0 = y ∧ q > q0); % procedure

assert(q > y); % implied(arith)

* Recursive functions/procedures
  + Use induction
    - Base case – function satisfies its specs when there are no recursive calls
    - Induction step – assume function satisfies its specs for all function calls; then it satisfies its specs
  + See examples
* Termination
  + A program can not terminate due to:
    - Non-terminating while loop
    - Non-terminating function/procedure recursion
  + Total correctness
    - |= tot assert(P); C; assert(Q);
    - i.e. for all program executions that satisfy P, C is guaranteed to terminate and Q is satisfied
  + While loop termination
    - To show that a loop terminates, identify a variant/bounding function:
    - An integer expression that
      * Is guaranteed to be nonnegative
      * Decreases with every loop iteration, and
      * Makes the loop guard become false at some point as it approaches 0
    - Ex:

assert(n > 0);

j := 0;

while (j != n) do {

j := j + 1

}

* + - Bounding function is n – j since n doesn’t change, j is incremented, and the loop stops when n = j
  + Recursion termination
    - Determine bounding function that is nonnegative and decreases with each recursive call, in terms of the arguments of the function
    - Show there is a section of the function that does not make recursive calls and is entered at some point as the bounding function approaches 0
    - See examples