This homework is concerned with the basics of assertional reasoning.

There are 3 simple problems, preceded by a sample problem and its solution.

Keep your answers brief, as in the example problem's solution.

You will get full marks: if an answer is not adequate or too long, I'll ask you to do it again.

The construct `sum(f(j): j in` α`)`, where α is a set, sequence or bag, equals the sum of `f(j)` where `j` ranges over α

- e.g.: `sum(f(j): j in 1..7)` equals `f(1)+f(2)+···+f(7)`
- e.g.: `sum(f(j): j in 1..0)` equals `0`

When answering a question of the form "Does program `Z` satisfy *Inv* P", do the following:

- If you answer yes, give a predicate, say B, such that
  - B satisfies the invariance induction rule (see `proofrules.pdf`), and
  - $B \Rightarrow P$ holds.
- If you answer no, give a finite allowed evolution ending in a state where P does not hold.

When answering a question of the form "Does program `Z` satisfy P *leads-to* Q", do the following:

- If you answer yes, give one or more leads-to assertions (each of the form X *leads-to* Y) such that
  - each leads-to assertion holds via the weak-fair or strong-fair rule (see `proofrules.pdf`), and
  - P *leads-to* Q follows from the closure (see `proofrules.pdf`) of the leads-to assertions
- If you answer no, give an allowed evolution of `Z` that satisfies `Z`'s progress assumption and does not satisfy P *leads-to* Q.

Sample Problem

```
program Z(int N) {
  int x ← 0;
  int y ← 0;
  while (y < N)
 a1: •x ← x + y + 1;
      y ← y + 1;
  return;

  atomicity assumption {as given by the '•'}
  progress assumption {weak fairness}
}
```

From the atomicity assumption, the following are atomic

- start-to-a1
- a1-to-a1
- a1-to-exit

Part a Does Z satisfy *Inv* A_0, where

A_0 : y = N ⇒ x = sum(j: j in 1..N)

Part b Does Z satisfy L_0, where

L_0 : y = 0 *leads-to* y = N

Sample Problem Solution

Part a

B_0 : `x = sum(j: j in 1..y)`

B_0 satisfies the invariance induction rule and implies A_0

This part is not required for the answer.

Details on how B_0 satisfies the invariance induction rule

- start-to-`a1`:
  zeros `x` and `y`, which establishes B_0
- `a1`-to-`a1`, starting with `y` equal to, say, `p`:
  B_0.lhs increases by `p+1`; `y` increases from `p` to `p+1`, adding `p+1` to the sum in B_0.rhs. So B_0 is preserved.
- `a1`-to-exit, starting with `y` equal to `N-1`:
  same as in `a1`-to-`a1`.

Details on how $B_0 \Rightarrow A_0$ holds

- B_0 with `y=N` yields `x = sum(j: j in 1..N)`.

Part b

L_1 : `(thread at a1) and (y = j < N)` *leads-to* `(thread at a1) and (y = j+1)`

L_1 holds via weak-fair rule. L_0 follows from closure of L_1.

This part is not required for the answer.

Details on how L_1 satisfies the weak-fair rule

- L_1.lhs (conjunct 1) implies that step start-to-`a1` is enabled.
- Step `a1`-to-`a1` establishes B_0.rhs from B_0.lhs
- Step `a1`-to-exit establishes B_0.rhs from B_0.lhs

Problem 1

```
program Z(int N) {
  // note: N can be 0 or negative                                        // now explicit
  int x ← 0;                                                                  // changed
  Tid[N] t;
  for (j in 0..N-1)
     t[j] ← startThread(up(j));
  return;

  function up(j)
  a1: • x ← x + j + 1;
     return;

  atomicity assumption {as given by the '•'s}
  progress assumption {weak fairness}
}
```

Part a Does Z satisfy *Inv* A_0, where

A_0 : x ≤ sum(j: j in 1..N)

Part b Does Z satisfy L_0, where

L_0 : true *leads-to* x = sum(j: j in 1..N)

By convention, sum(empty set) is 0.

Problem 2

```
program Z(int N) {
  int x ← 4;
  t1 ← startThread(fn1());
  t2 ← startThread(fn2());
  t3 ← startThread(fn3());
  return;

  function fn1()
     repeat
 a1:  • x ← 1;
 b1: until  • (x = 3);
     return;

  function fn2()
     repeat
 a2:  • if (x = 1)  x ← 2;
        else  x ← 4;
 b2: until  • (x = 3);
     return;

  function fn3()
     repeat
 a3:  • if (x = 2)  x ← 3;
        else  x ← 4;
 b3: until  • (x = 3);
     return;

  atomicity assumption {as given by the '•'s}
  progress assumption {weak fairness}
}
```

Part a Does Z satisfy *Inv* A_0, where

A_0 : (not (t1.alive or t2.alive or t3.alive)) ⇒ x = 3

Part b Does Z satisfy *Inv* A_1, where

A_1 : (not t1.alive or not t2.alive or not t3.alive) ⇒ x = 3

Part c Does Z satisfy L_0, where

L_0 : ((t1 at a1) and (t2 at a2) and (t3 at a3)) *leads-to* not (t1.alive or t2.alive or t3.alive)

Problem 3

Consider the following distributed shortest-distance algorithm for a network of nodes and node-to-node fifo channels. There are `N` nodes, with ids $1, \cdots, N$. There are channels for a given subset `E` of node pairs, i.e., there is a channel from `i` to `j` iff `[i,j]` is in `E`. The channel from `i` to `j` has a nonnegative cost `D[i,j]`. The graph of the nodes and channels may not be fully connected. For every node `i` that is reachable from node `1`, let `D[i]` denote the shortest distance from `1` to `i`

Every node `i` has a variable `dist[i]`, indicating the current estimate of the shortest distance from node `1` to node `i`. Node `1` starts the computation by sending on every outgoing channel `[1,j]` the message `[D[1,j]]`. When node `i` receives a message `[d]`, if `d` is less than `dist[i]` then node `i` sets `dist[i]` to `d` and sends on every outgoing channel `[i,j]` the message `[d+D[i,j]]`.

The program below models the above within a single system. Variable α`[i,j]` has the sequence of messages in transit. Also, the activity is defined by rules, rather than explicity threads. Also, ∞ denotes “max int”.

```
program Z(int N, E, D) {
  ic {N > 0 and
      (E subsetOf set([i,j]: i,j in [1..N], i ≠ j))
     }

  init:
    for ([i,j] in E)
       α[i,j] ← [];
    dist[1..N] ← ∞;
    dist[1] ← 0;
    for ([1,j] in E)
       append [D[1,j]] to α[1,j];

  rule rcv(i,j), for [i,j] in E:
    await (α[i,j] ≠ [])
       remove [d] from α[i,j].head;
       if (d < dist[j])
          dist[j] ← d;
          for ([j,k] in E)
            append [d+D[j,k]] to α[j,k].tail;
     return;

  atomicity assumption {init, each rule}
  progress assumption {weak fairness}
}
```

Part a Does `Z` satisfy *Inv* A_0, where

A_0 : `((i in 2..N) and dist[i]` $\neq \infty$`)` $\Rightarrow$ (there is a path from `1` to `i` of length `dist[i]`) // added `i in 2..N`

Part b Does `Z` satisfy L_0, where

L_0 : `((i in 2..N) and (i reachable from 1))` *leads-to* `dist[i] = D[i]` // added `i in 2..N`

Part c Does `Z` satisfy L_1, where

L_1 : `((in in 2..N) and ([i,j] in E))` *leads-to* α`[i,j] = D[i]` // added `i in 2..N`