3. Programming Logic (ch. 3)

Operational Semantic Model

- The meaning of a program is given by a set E(P) of infinite sequences:
  - $R \in E(P)$ is an execution sequence: (s_0,l_0) (s_1,l_1) ...
  - R_i is the i'th element of the sequence
  - R_i.state is the program state <u>before</u> the i'th step
  - R_0.state is the initial program state (one of many—e.g., some variables may not be initialized)
  - R_i.label is the program statement selected in the i'th step
- Constraints over the set E(P):
  - <u>deterministic</u>
    R_i.state and R_i.label fully determine R_{i+1}.state
  - <u>weakly fair</u>
    $\langle \forall\ s,i :: \langle \exists\ k:: R_k.label=s \rangle$
    $\wedge R_i.label=s \Rightarrow \langle \exists\ j : j>i :: R_j.label=s \rangle \rangle$
- Discussion:
  - If fair interleaving and atomicity are preserved, the number of parallel machines executing the program does not alter its semantics
  - Infinite sequences are needed in order to talk about fairness—one cannot prove properties of a fair sequence by proving properties of its finite subsequences
  - This model is not actually used for reasoning about programs
  - Proofs involve reasoning about assertions

Proving properties of assignment statements

- Assignment axiom

  $\{p\}\ x := exp\ \{q\} \equiv p \Rightarrow q^{x}_{exp} \equiv p \Rightarrow wp(x:=exp,q)$

  - Given a state in which p holds, the execution of the assignment s produces a state in which q holds
  - This is a well understood concept in sequential programming
  - The statement must be terminating and deterministic
- Simple multiple assignments

  $\{y=k\}\ x,y := 0,y+1\ \{x>-3 \wedge y>k\}$

$$(y=k) \Rightarrow (x>-3 \wedge y>k)^{x;y}_{0;y+1}$$

$\equiv$

$$(y=k) \Rightarrow (0>-3 \wedge y+1>k)$$

- Conditional multiple assignments

exp ≡ E0 if B0 ~ E1 if B1 ~ ... ~En if Bn

$$q^{x}_{exp} \equiv (B0 \Rightarrow q^{x}_{E0}) \wedge \ldots \wedge (Bn \Rightarrow q^{x}_{En}) \wedge ((\neg B0 \wedge \ldots \wedge \neg Bn) \Rightarrow q)$$

{true} x := -1 if x<0 ~ 0 if x=0 ~ 1 if x>0 {-1≤x≤1}

requires one to prove

```
{true ∧ x<0}  x := -1   {-1≤x≤1}
{true ∧ x=0}  x := 0    {-1≤x≤1}
{true ∧ x>0}  x := 1    {-1≤x≤1}
(true ∧ false)   ⇒      (-1≤x≤1)
```

- Assignments involving arrays
  - (A; i:u) ≡ same as array A except for the value u being associated with index i

⟨ ‖ i : 0≤i≤N :: A[i] := A[N-i] ⟩ {⟨ ∀ j : 0≤j<N :: A[j] < A[j+1] ⟩}

using the substitution rule

⟨ ∀ j : 0≤j<N ::
  ⟨ A; i : 0≤i≤N :: A[N-i] ⟩[j] < ⟨ A; i : 0≤i≤N :: A[N-i] ⟩[j+1] ⟩

≡

⟨ ∀ j : 0≤j<N :: A[N-j] < A[N-(j+1)] ⟩

≡

⟨ ∀ j : 0≤j<N :: A[(N-j)] < A[(N-j)-1] ⟩

≡

⟨ ∀ j : 0<j≤N :: A[j] < A[j-1] ⟩

Assertions about programs

- Assignment axiom revisited
  - s is a statement in P and has a finite execution

$$\{p\}\ s\ \{q\} \equiv \langle \forall R,i : R \in \mathbf{E}(P) \wedge i\geq 0 :: (p(R_i.state) \wedge R_i.label=s) \Rightarrow q(R_{i+1}.state) \rangle$$

- General properties

{p} s {true}

{false} s {q}

{p} s {false} — *hypothesis* — inference rule

¬p — *conclusion*

{p} s {q}, {p'} s {q'}

$\{p \wedge p'\}$ s $\{q \wedge q'\}$, $\{p \vee p'\}$ s $\{q \vee q'\}$

$$\frac{p' \Rightarrow p,\ \{p\}\ s\ \{q\},\ q \Rightarrow q'}{\{p'\}\ s\ \{q'\}}$$

- Quantification over the set of program statements
  - non-decreasing x ≡ a safety property (nothing goes wrong)
    `⟨ ∀ s :: {x=k} s {x≥k} ⟩`
  - eventually-increasing x ≡ a progress property (something does happen)
    `⟨ ∃ s :: {x=k} s {x>k} ⟩`
  - This is all we need really!

Commonly used assertions

Sample program: Comparing two ascending sequences

- Given two ascending sequences of numbers, determine if they represent the same set
- Assumptions

```
f[0] = g[0]
f[N] = g[N]
f[N] > f[N-1]
g[N] > g[N-1]
⟨ ∀ i : 0≤i<N :: f[i] ≤ f[i+1] ⟩
⟨ ∀ i : 0≤i<N :: g[i] ≤ g[i+1] ⟩

Program  Compare
   declare   u,v : integer
   initially u,v = 0,0
   assign
   s1      u := u+1          if u<N ∧ f[u]=f[u+1]
   s2  []  v := v+1          if v<N ∧ g[v]=g[v+1]
   s3  []  u,v := u+1,v+1    if v<N ∧ u<N ∧ f[u+1]=g[v+1]
end
```

UNLESS relation

- If p and ¬q hold, either they hold forever or q eventually holds

```
    (p∧¬q)    (p∧¬q)    (p∧¬q) ...    (p∧¬q) ...
or
    (p∧¬q)    (p∧¬q)    (q)    ...?   ...?   ...?
```

- which is stated operationally as

$p[R_i] \Rightarrow \langle \forall j : j \geq i :: (p \wedge \neg q)[R_j] \rangle$ -- holds forever
$\vee$
$\langle \exists k : k \geq i :: q[R_k] \wedge \langle \forall j : i \leq j < k :: (p \wedge \neg q)[R_j] \rangle \rangle$

Note: we extended the notion to use p(R) ≡ p(R.state)

- and as an inference rule

$\langle \forall s : s \text{ in } P :: \{p \wedge \neg q\}\ s\ \{p \vee q\} \rangle$
----
p **unless** q

- Other safety properties derived from **unless**

```
stable p ≡ p unless false
       ...?      ...?      (p)...(p)    ...

const. p ≡ stable p ∧ stable ¬p

       (p)...(p)...(p)...
   or
       (¬p)   ...(¬p)   ...(¬p)   ...

inv. p ≡ stable p ∧ (init ⇒ p)
       (p)...(p)...(p)...
```

Note that (**const**. x=3) does not mean (**inv**. x=3)

- Invariant Substitution Axiom≡an invariant is a theorem in the context of the program; if we have

```
inv. p ≡ true
```

we can replace p and true anywhere

- This is why we can prove the following inference rule

p **unless** q, **inv**. ¬q
----
stable p

¬q ≡ true	substitution axiom	(1)
p **unless** q	premise	
p **unless** false	using (1)	
stable p	definition	

- Illustrations (program Compare)

```
const. N>7
const. f[3]=g[8]
u=k unless u>k
stable u=N
inv.   0≤u≤N ∧ 0≤v≤N
inv.   f[u]=g[v]
inv.   ⟨ set i : 0≤i≤u :: f[i] ⟩ = ⟨ set i : 0≤i≤v :: g[i] ⟩
inv.   u=N ≡ v=N
```

ENSURES relation

- It states that (p **unless** q) holds and there is one statement which, when executed, guarantees that q is established
- The fairness requirement guarantees that the statement is eventually executed
- It provides a way of proving progress (see the existential quantification)

```
(p∧¬q)    (p∧¬q)    s      (q)    ...?   ...?   ...?
```

- which is stated operationally as

$$p[R_i] \;\Rightarrow\; \langle \exists\, k : k \ge i :: q[R_k] \wedge \langle \forall\, j : i \le j < k :: p[R_j] \rangle \rangle$$
$$\wedge\; \langle \exists\, s :: \{p \wedge \neg q\}\ s\ \{q\} \rangle$$

- and as an inference rule

  p **unless** q, ⟨ ∃ s : s in P :: {p ∧ ¬q} s {q} ⟩
  ----
  p **ensures** q

- Illustrations (program Compare)

  `u=k` **`ensures`** `u>k`

  - False. The unless part holds but u may not be incremented.

  `f=g ∧ u=k<N` **`ensures`** `u>k`

  - False. Consider the case in which selecting s1 will not increment u

3	u->3	4
v->3	3	4

  `f[u]=f[u+1] ∧ u=k<N` **`ensures`** `u>k`

  - True. The unless part holds. s1 can increment u.

  `f[u]=f[u+1]≠g[v+1] ∧ u=k<N` **`ensures`** `u>k`

  - True. The unless part holds. Only s1 can increment u.

  $u<u_0 \wedge v<v_0 \wedge u+v=k$ **ensures** $u+v>k$

  - False. (Assume u_0 and v_0 represent the largest index in subarrays containing the same set of values.)

LEADS-TO relation

- Once p holds, either q holds or will hold sometime in the future
- The property p need not hold in-between
- This is the usual way of specifying progress (The **ensures** relation is too strong and too close to the statement level.)

  `(p)    ...?    ...?    (q)    ...?`

- which is stated operationally as

$$p[R_i] \;\Rightarrow\; \langle \exists\, k : k \ge i :: q[R_k] \rangle$$

- and deduced using the following inference rules

  p **ensures** q

  $p \rightarrow q$

  $p \rightarrow q,\ q \rightarrow r$ (transitivity)

  $p \rightarrow r$

$\langle \forall$ m : m∈W :: p(m) → q $\rangle$ (disjunction--for any set W)

$\langle \exists$ m : m∈W :: p(m) $\rangle$ → q

- Illustrations (program Compare)

  $u<u_0 \wedge v<v_0 \wedge u+v=k \rightarrow u+v>k$

  - True. (Assume u_0 and v_0 as before.)

    $u<u_0 \wedge v<v_0 \wedge f[u]=f[u+1]\neq g[v+1] \wedge u+v=k \rightarrow u+v>k$ (ensures s1)

    $u<u_0 \wedge v<v_0 \wedge g[v]=g[v+1]\neq f[u+1] \wedge u+v=k \rightarrow u+v>k$ (ensures s2)

    $u<u_0 \wedge v<v_0 \wedge f[u]=g[v] \wedge f[u+1]=g[u+1] \wedge u+v=k \rightarrow u+v>k$ (ensures s3)

    $u<u_0 \wedge v<v_0 \wedge u+v=k \rightarrow u+v>k$ (disjunction)

  creates disjoint cases

3	**3**	4		3	**3**	4
3	3	**4**		3	**3**	4

  $u,v=0,0 \rightarrow u,v=u_0,v_0$

  - True. (Transitivity—with some induction to be discussed later.)

FIXED POINT

- A program reaches a fixed point if the program state can no longer change

  $FP[R_i] \Rightarrow \langle \forall$ k : k≥i :: R_k.state = R_i.state $\rangle$

- A program may have more than one fixed point

  ```
  x:=1 if x=0 [] x:=2 if x=0
  ```

- It is a concept useful for terminating programs but not for reactive programs
- In UNITY the fixed point may be characterized syntactically

  FP ≡ $\langle \forall$ s : s in P ∧ (s is x:=exp) :: x = exp $\rangle$

- Simple examples

  ```
  k := k+1
     FP ≡ k=k+1 ≡ false -- there is none!

  k := k+1 if k<N
     FP ≡ k<N ⇒ k=k+1 ≡ k<N ⇒ false ≡ k≥N
     Note: k may be initially (N+3)
  ```

- Illustrations (program Compare)

  - the fixed point is (using p ⇒ q ≡ ¬p ∨ q)

    ```
    FP ≡
        (u≥N ∨ f[u]≠f[u+1]) ∧
        (v≥N ∨ g[v]≠g[v+1]) ∧
        (u≥N ∨ v≥N ∨ f[u+1]≠g[v+1])
    ```

- using the invariants

```
inv. u=N ≡ v=N
inv. ⟨ set i : 0≤i≤u :: f[i] ⟩ = ⟨ set i : 0≤i≤v :: g[i] ⟩
```

- FP simplifies as follow

  Case 1: u=N ∧ v=N

  ```
  ⟨ set i : 0≤i≤N :: f[i] ⟩ = ⟨ set i : 0≤i≤N :: g[i] ⟩
  ```

  Case 2: u<N ∧ v<N

  ```
  (f[u]≠f[u+1]) ∧ (g[v]≠g[v+1]) ∧ (f[u+1]≠g[v+1])

  ⟨ set i : 0≤i≤N :: f[i] ⟩ ≠ ⟨ set i : 0≤i≤N :: g[i] ⟩
  ```

f[u+1]<g[v+1]	assume to be the case
f[u]<f[u+1]	ascending
g[v]=f[u]	invariant above
g[v]<f[u+1]<g[v+1]	the three lines above
f[u+1] not in g	

- We can conclude

```
FP ⇒
    (u=N ≡ v=N) ∧
    (u=N ≡ ⟨ set i : 0≤i≤N :: f[i] ⟩ = ⟨ set i : 0≤i≤N :: g[i] ⟩ )
```