Smallstep Semantics & Forward & Backward Simulation

(CS428/528 Lecture 10)

Smallstep Semantics

```
Record semantics : Type := Semantics_gen {
  state: Type;
  genvtype: Type;
  step : genvtype -> state -> trace -> state -> Prop;
  initial_state: state -> Prop;
  final_state: state -> int -> Prop;
  globalenv: genvtype;
  symbolenv: Senv.t
}.
```

Global Environments

```
Record t: Type := mkgenv {
  genv_public: list ident;     (* which symbol names are public *)
  genv_symb: PTree.t block;    (* mapping symbol -> block *)
  genv_defs: PTree.t (globdef F V);    (* mapping block -> definition *)
  genv_next: block;      (* next symbol pointer *)
  genv_symb_range:
        forall id b, PTree.get id genv_symb = Some b -> Plt b genv_next;
  genv_defs_range:
        forall b g, PTree.get b genv_defs = Some g -> Plt b genv_next;
  genv_vars_inj:
        forall id1 id2 b, PTree.get id1 genv_symb = Some b ->
           PTree.get id2 genv_symb = Some b -> id1 = id2
}.
```

Global Definitions & Variables

```
Inductive globdef (F V: Type) : Type :=
  | Gfun (f: F)
  | Gvar (v: globvar V).

Record globvar (V: Type) : Type := mkglobvar {
  gvar_info: V;    (* language-dependent info, e.g. a type *)
  gvar_init: list init_data;    (* initialization data *)
  gvar_readonly: bool;  (* read-only variable? (const) *)
  gvar_volatile: bool  (* volatile variable? *)
}.
```

Program & Program Module

```
Record program (F V: Type) : Type := mkprogram {
  prog_defs: list (ident * globdef F V);
  prog_public: list ident;
  prog_main: ident
}.
```

Symbol Environments

```
Record t: Type := mksenv {

  find_symbol: ident -> option block;
  public_symbol: ident -> bool;

  invert_symbol: block -> option ident;

  block_is_volatile: block -> bool;

  nextblock: block;

}.
```

Clight Function Definition

```
Inductive fundef : Type :=
  | Internal: function -> fundef
  | External: external_function -> typelist -> type
                                          -> calling_convention -> fundef.
Record function : Type := mkfunction {
  fn_return: type;
  fn_callconv: calling_convention;
  fn_params: list (ident * type);
  fn_vars: list (ident * type);
  fn_temps: list (ident * type);
  fn_body: statement
}.
```

Asm Function Definition & Program Module

```
Definition code := list instruction.

Record function : Type :=

            mkfunction { fn_sig: signature; fn_code: code }.

Definition fundef := AST.fundef function.

Definition program := AST.program fundef unit.
```

Forward Simulation

```
Record fsim_properties (L1 L2: semantics) (index: Type)
                       (order: index -> index -> Prop)
                       (match_states: index -> state L1 -> state L2 -> Prop) : Prop := {
    fsim_order_wf: well_founded order;
    fsim_match_initial_states:
      forall s1, initial_state L1 s1 ->
      exists i, exists s2, initial_state L2 s2 /\ match_states i s1 s2;
    fsim_match_final_states:
      forall i s1 s2 r,
      match_states i s1 s2 -> final_state L1 s1 r -> final_state L2 s2 r;
    fsim_simulation:
      forall s1 t s1', Step L1 s1 t s1' ->
      forall i s2, match_states i s1 s2 ->
      exists i', exists s2',
         (Plus L2 s2 t s2' \/ (Star L2 s2 t s2' /\ order i' i))
      /\ match_states i' s1' s2';
    fsim_public_preserved:
      forall id, Senv.public_symbol (symbolenv L2) id = Senv.public_symbol (symbolenv L1) id
  }.
```

Forward Simulation (cont'd)

```
Inductive forward_simulation (L1 L2: semantics) : Prop :=
  Forward_simulation
    (index: Type)
    (order: index -> index -> Prop)
    (match_states: index -> state L1 -> state L2 -> Prop)
    (props: fsim_properties L1 L2 index order match_states).
```

Backward Simulation

```
Record bsim_properties (L1 L2: semantics) (index: Type)
                       (order: index -> index -> Prop)
                       (match_states: index -> state L1 -> state L2 -> Prop) : Prop := {
    bsim_order_wf: well_founded order;
    bsim_initial_states_exist:
      forall s1, initial_state L1 s1 -> exists s2, initial_state L2 s2;
    bsim_match_initial_states:
      forall s1 s2, initial_state L1 s1 -> initial_state L2 s2 ->
      exists i, exists s1', initial_state L1 s1' /\ match_states i s1' s2;
    bsim_match_final_states:
      forall i s1 s2 r,
      match_states i s1 s2 -> safe L1 s1 -> final_state L2 s2 r ->
      exists s1', Star L1 s1 E0 s1' /\ final_state L1 s1' r;
```

Backward Simulation (cont’d)

```
bsim_progress:
  forall i s1 s2,
  match_states i s1 s2 -> safe L1 s1 ->
  (exists r, final_state L2 s2 r) \/
  (exists t, exists s2', Step  L2 s2 t s2');
bsim_simulation:
  forall s2 t s2', Step L2 s2 t s2' ->
  forall i s1, match_states i s1 s2 -> safe L1 s1 ->
  exists i', exists s1',
     (Plus L1 s1 t s1' \/ (Star L1 s1 t s1' /\ order i' i))
  /\ match_states i' s1' s2';
bsim_public_preserved:
  forall id, Senv.public_symbol (symbolenv L2) id = Senv.public_symbol (symbolenv L1) id
}.
```

Safety

```
Definition safe (L: semantics) (s: state L) : Prop :=
  forall s',
  Star L s E0 s' ->
      (exists r, final_state L s' r)
       \/ (exists t, exists s'', Step L s' t s'').
```

Backward Simulation (cont'd)

```
Inductive backward_simulation (L1 L2: semantics) : Prop :=
  Backward_simulation
    (index: Type)
    (order: index -> index -> Prop)
    (match_states: index -> state L1 -> state L2 -> Prop)
    (props: bsim_properties L1 L2 index order match_states).
```

Compiler Correctness

```
Theorem transf_c_program_correct:
  forall p tp,
  transf_c_program p = OK tp ->
  backward_simulation (Csem.semantics p) (Asm.semantics tp).
```