Ragionamento Automatico

Progetto 1

Michele Dalla Chiara - VR464051
Davide Zampieri - VR458470

A.A. 2020 - 2021

1 Sintassi di IMP

1.1 Insiemi

IMP è un linguaggio modello di tipo imperativo. Tale linguaggio permette di lavorare su:

- Valori numerici N.
- Valori booleani B.
- Celle di memoria (o locazioni) Loc.

Esiste inoltre la nozione di stato della memoria, ovvero una mappa che assegna ad ogni cella di memoria il rispettivo valore. Andremo quindi a denotare:

- Espressioni aritmetiche $Aexp$.
- Espressioni booleane $Bexp$.
- Comandi Com.

1.2 Librerie

```
Require Import String.
Require Import List.
Require Import ZArith.
Require Import Unicode.Utf8.
```

1.3 Regole di formazione delle espressioni aritmetiche

Sia a un arbitrario elemento di Aexp: $a ::= n|X|a_0 + a_1|a_0 - a_1|a_0 \ * \ a_1$

```
Inductive Aexp : Set :=
| num: nat→Aexp
| id: string→Aexp
| add: Aexp→Aexp→Aexp
| sub: Aexp→Aexp→Aexp
| mul: Aexp→Aexp→Aexp.
```

1.4 Regole di formazione delle espressioni booleane

Sia b un arbitrario elemento di Bexp: $b ::= true|false|a_0 = a_1|a_0 \leq a_1|\neg b|b_0 \wedge b_1|b_0 \vee b_1$

```
Inductive Bexp: Set :=
| btrue: Bexp
| bfalse: Bexp
| eq: Aexp→Aexp→Bexp
| le: Aexp→Aexp→Bexp
| not: Bexp→Bexp
| and: Bexp→Bexp→Bexp
| or: Bexp→Bexp→Bexp.
```

1.5 Regole di formazione dei comandi

Sia c un arbitrario elemento di Com: $c ::= skip|c_0; c_1|if\ b\ then\ c_0\ else\ c_1|X := a|while\ b\ do\ c$

```
Inductive Com: Set :=
| skip: Com
| seq: Com→Com→Com
| if_else: Bexp→Com→Com→Com
| ass: string→Aexp→Com
| while_do: Bexp→Com→Com.
```

1.6 Stato della memoria

La memoria è una lista di coppie (locazione, valore).

```
Record state : Set := ST {loc: string; value: nat}.

Fixpoint lookup (a: string) (l: list state) : nat :=
  match l with
    | b :: m => if eqb (loc b) a then value b else lookup a m
    | nil => 0
  end.
```

```
Fixpoint update' (l: string) (v: nat) (head: list state) (tail: list state): list
state :=
  match tail with
    | h :: tail' => if eqb (loc h) l then app head ((ST l v)::tail') else update' l v
(h::head) tail'
    | nil => app head ((ST l v)::nil)
  end.

Definition update (l: string) (v: nat) (mem: list state): list state :=
  update' l v nil mem.
```

2 Semantica operazionale di IMP

2.1 Valutazione delle espressioni aritmetiche

Valutando un'espressione aritmetica si produce un numero secondo la seguente relazione di valutazione: $\langle a, \sigma \rangle \to nat$ dove $\sigma \in \Sigma = \{\sigma | \sigma : Loc \to nat\}$.

```
Fixpoint aeval (st: list state) (a : Aexp) : nat :=
  match a with
  | num n => n
  | id x => lookup x st
  | add a0 a1 => (aeval st a0) + (aeval st a1)
  | sub a0 a1 => (aeval st a0) - (aeval st a1)
  | mul a0 a1 => (aeval st a0) * (aeval st a1)
  end.
```

2.2 Valutazione delle espressioni booleane

Valutando un'espressione booleana si produce un valore booleano secondo la seguente relazione di valutazione: $\langle b, \sigma \rangle \to \{true, false\}$ dove $\sigma \in \Sigma = \{\sigma | \sigma : Loc \to nat\}$.

```
Fixpoint beval (st: list state) (b : Bexp) : bool :=
  match b with
  | btrue => true
  | bfalse => false
  | eq a0 a1 => (aeval st a0) =? (aeval st a1)
  | le a0 a1 => (aeval st a0) <=? (aeval st a1)
  | not b0 => negb (beval st b0)
  | and b0 b1 => andb (beval st b0) (beval st b1)
  | or b0 b1 => orb (beval st b0) (beval st b1)
  end.
```

2.3 Esecuzione dei comandi

L'esecuzione di un comando in un determinato stato termina in un nuovo stato secondo la seguente relazione di esecuzione: $\langle c, \sigma \rangle \to \sigma'$ dove $\sigma \in \Sigma = \{\sigma | \sigma : Loc \to nat\}$.

```
Inductive exec : list state→Com→list state→Prop :=
| skip_exec : ∀ s:list state, exec s skip s

| ass_exec : ∀ (s:list state) (loc:string) (n:nat) (a:Aexp),
                    aeval s a = n →
                    exec s (ass loc a) (update loc n s)

| seq_exec : ∀ (s s' s'':list state) (c1 c2:Com),
                    exec s c1 s' → exec s' c2 s'' →
                    exec s (seq c1 c2) s''

| while_do_false_exec : ∀ (s:list state) (c:Com) (b:Bexp),
                    beval s b = false → exec s (while_do b c) s

| while_do_true_exec : ∀ (s s' s'':list state) (c:Com) (b:Bexp),
                    beval s b = true →
                    exec s c s' →
                    exec s' (while_do b c) s'' →
                    exec s (while_do b c) s''

| if_else_true_exec : ∀ (s s':list state) (c1 c2:Com) (b:Bexp),
                    beval s b = true →
                    exec s c1 s' →
                    exec s (if_else b c1 c2) s'

| if_else_false_exec : ∀ (s s':list state) (c1 c2:Com) (b:Bexp),
                    beval s b = false →
                    exec s c2 s' →
                    exec s (if_else b c1 c2) s'
.
```

2.4 Notazioni personalizzate

```
Declare Scope notazioni.
Notation "a0 '+' a1" := (add a0 a1) : notazioni.
Notation "a0 '-' a1" := (sub a0 a1) : notazioni.
Notation "a0 '*' a1" := (mul a0 a1) : notazioni.
```

```
Notation "a0 '==' a1" := (eq a0 a1) (at level 60) : notazioni.
Notation "a0 '<=' a1" := (le a0 a1) : notazioni.
Notation "'!' b0" := (not b0) (at level 60) : notazioni.
Notation "b0 'AND' b1" := (and b0 b1) (at level 60) : notazioni.
Notation "b0 'OR' b1" := (or b0 b1) (at level 60) : notazioni.

Notation "c0 ';' c1" := (seq c0 c1) (at level 60) : notazioni.
Notation "'IF{' b '}THEN' c0 'ELSE' c1" := (if_else b c0 c1) (at level 60) : notazioni.
Notation "X ':->' a" := (ass X a) (at level 60) : notazioni.
Notation "'WHILE' b 'DO' c" := (while_do b c) (at level 60) : notazioni.

Notation "s '[' l '\' v ']'" := ((ST l v)::s) (at level 60, right associativity) : notazioni.
Open Scope notazioni.
```

3 Equivalenza dei comandi

L'equivalenza fra due comandi c_0 e c_1 è garantita se a partire dalla stessa memoria σ entrambi riescono a convergere sulla memoria risultato σ', ovvero:

$$c_0 \sim c_1 \iff (\forall \sigma, \sigma' \in \Sigma. \langle c_0, \sigma \rangle \to \sigma' \Leftrightarrow \langle c_1, \sigma \rangle \to \sigma')$$

```
Definition Equivalence (c0 c1 :Com) : Prop :=
  ∀ (s s' : list state), (exec s c0 s' ↔ exec s c1 s').
```

3.1 Dimostrazione

Se $w \equiv \mathit{while}\ b\ \mathit{do}\ c$ e $w' \equiv \mathit{if}\ b\ \mathit{then}\ c; w\ \mathit{else}\ \mathit{skip}$, allora $w \sim w'$.

```
Section Domanda_2.
Variable b : Bexp.
Variable c : Com.
Definition w : Com := WHILE b DO c.
Definition w' : Com := IF{ b }THEN (c ; w) ELSE skip.

Lemma Domanda_2: Equivalence w w'.
Proof.
   unfold Equivalence.  unfold w.  unfold w'.  unfold w.  intros.  split.
   - intros.
    inversion H.
    + subst.
    apply if_else_false_exec.  assumption.
    apply skip_exec.
    + subst.
    apply if_else_true_exec.  assumption.
    apply seq_exec with (s':=s'0); assumption.
   - intros.
    inversion H.
```

```
    + subst.  inversion H6.  subst.
    apply while_do_true_exec with (s':=s'0); assumption.
    + subst.  inversion H6.  subst.
    apply while_do_false_exec.  assumption.
Qed.
End Domanda_2.
```

Nota: la tattica `inversion` sui predicati induttivi esplicita quali sono i vincoli per l'utilizzo dei costruttori che potrebbero aver prodotto una determinata ipotesi.

3.2 Dimostrazione

Sia $w \equiv while\ 0 < x\ do\ (y := 2 * y; x := x - 1)$, allora $\forall\sigma.\exists\sigma^*.\ \langle w, \sigma[2/x][3/y]\rangle \to \sigma^*$.

```
Section Domanda_3.
Definition x : string := "X".
Definition y : string := "Y".

Definition sigma' (s : list state) : list state := (s[x\0][y\12]).

Definition Domanda_3 := WHILE (!((id x) <= (num 0)))
  DO ((y :-> ((id y) * (num 2)));(x :-> ((id x) - (num 1)))).

Lemma Dimostrazione_3: ∀ s: list state, ∃ s': list state,
                        exec (s[x\2][y\3]) Domanda_3 s'.
Proof.
   unfold Domanda_3.
   intros.
   exists (sigma' s).
   eapply while_do_true_exec.  reflexivity.
   - eapply seq_exec.
    + now apply ass_exec with (n:=6).
    + now apply ass_exec with (n:=1).
   - eapply while_do_true_exec.  reflexivity.
    + eapply seq_exec.
    * now apply ass_exec with (n:=12).
    * now apply ass_exec with (n:=0).
    + apply while_do_false_exec.  reflexivity.
Qed.
End Domanda_3.
```

Nota: la tattica `eapply` consente di non dover specificare l'istanza della variabile che rappresenta lo stato intermedio necessaria ad alcune regole di esecuzione dei comandi; piuttosto, usa un segnaposto per indicare il fatto che l'istanza può essere trovata in seguito nella prova.