Implementazione del linguaggio Imp

Francesco Cecconello VR457796

24 Giugno 2023

1 Introduzione

In questo progetto verrà presentata l'implementazione del linguaggio Imp attraverso Coq, utilizzando le regole di valutazione delle espressioni e dei comandi definite all'interno della consegna. Inoltre, il linguaggio appena definito verrà utilizzato per dimostrare alcune proposizioni riguardanti il costrutto While.

2 Funzioni

Utilizzando le regole, è possibile definire delle funzioni di valutazione per ogni espressione aritmetica o booleana e per ogni comando.

2.1 AExp

Le espressioni aritmetiche ritornano sempre un naturale. Le espressioni contenute in altre espressioni vengono valutate, rendendo il procedimento ricorsivo.

```
Fixpoint aeval (a : aexp) (st : stato) : nat :=
  match a with
  | ANum n => n
  | AId x => st x
  | APiu a1 a2 => (aeval a1 st) + (aeval a2 st)
  | AMeno a1 a2 => (aeval a1 st) - (aeval a2 st)
  | APer a1 a2 => (aeval a1 st) * (aeval a2 st)
  end.
```

Ad esempio, per applicare la regola *APiu*, viene valutata a_1, poi viene valutata a_2 e, infine, quando queste convergono a due numeri n_0 e n_1, si può farne la somma.

2.2 BExp

Le espressioni booleane ritornano sempre un booleano. Le espressioni contenute in altre espressioni vengono valutate, rendendo il procedimento ricorsivo.

```
Fixpoint beval (b : bexp) (st : stato) : bool :=
  match b with
  | BVero => true
  | BFalso => false
  | BNot b' => negb (beval b' st)
  | BUguale a1 a2 => (aeval a1 st) =? (aeval a2 st)
  | BMinoreUguale a1 a2 => (aeval a1 st) <=? (aeval a2 st)
  | BAnd b1 b2 => andb (beval b1 st) (beval b2 st)
  | BOr b1 b2 => orb (beval b1 st) (beval b2 st)
  end.
```

Ad esempio, per applicare la regola *BUguale*, viene valutata a_1, poi viene valutata a_2 e, infine, quando queste convergono a due numeri n_0 e n_1, si può confrontarle, ritornando *true* o $false$.

2.3 Com

I comandi ritornano sempre uno stato.

```
Inductive com : Type :=
  | CSkip : com
  | CAss : id -> aexp -> com
  | CSeq : com -> com -> com
  | CIf : bexp -> com -> com -> com
  | CWhile : bexp -> com -> com.
```

2.3.1 Valutazione non funzionante

Inizialmente si è pensato di implementare la valutazione dei comandi seguendo la stessa logica delle precedenti funzioni di valutazione.

```
Fixpoint ceval (c : com) (st : stato) : stato :=
  match c with
  | CSkip => st
  | CAss x a1 => aggiorna_stato st x (aeval a1 st)
  | CSeq c1 c2 => ceval c2 (ceval c1 st)
  | CIf b c1 c2 => if beval b st then ceval c1 st else ceval c2 st
  | CWhile b c1 => if beval b st then ceval (CWhile b c1) (ceval c1 st) else st
  end.
```

Si è però presentato un problema con la valutazione del comando CWhile, poiché CoqIde ritornava l'errore "Cannot guess decreasing argument of fix". Deducendo che Coq non prevedesse un limite alla ricorsione delle funzioni di

punto fisso - e che non potesse quindi accertarsi della terminazione del ciclo - il primo tentativo di correzione ha previsto l'inserimento di un limite di ricorsione (sotto forma di numero intero) per imporre manualmente una quantità massima di memoria disponibile per l'esecuzione del comando, oltre la quale il ciclo sarebbe stato terminato forzatamente.

```
Fixpoint ceval_limited (c : com) (st : stato) (limite : nat) : stato :=
  match limite with
  | O => st
  | S n =>
    match c with
    | CSkip => st
    | CAss x a1 => aggiorna_stato st x (aeval a1 st)
    | CSeq c1 c2 => ceval_limited c2 (ceval_limited c1 st n) n
    | CIf b c1 c2 => if beval b st then ceval_limited c1 st n else ceval_limited c2 st n
    | CWhile b c1 => if beval b st then ceval_limited (CWhile b c1) (ceval_limited c1 st n)
      else st
    end
  end.
```

Il codice si è rivelato effettivamente funzionante per quanto riguarda la valutazione dei comandi, ma si è presentato un problema dal punto di vista della complessità della dimostrazione sia del teorema che dell'esempio, rendendo troppo ostica la prova di entrambe le proposizioni. La causa di tale difficoltà risiede probabilmente nella scelta errata del tipo dato a **ceval_limited**: mentre **aeval** e **beval** rappresentano effettivamente la valutazione di espressioni aritmetiche o booleane, l'esecuzione di un comando non è effettivamente una pura e semplice valutazione, ma porta al cambiamento dello stato corrente. Per questo motivo si è rivelato più utile vedere l'esecuzione di un comando non come una funzione con tipo di ritorno **int** o **bool** come **aeval** e **beval**, ma piuttosto come una funzione assimilabile ad un metodo "void", rendendo necessaria la definizione di **ceval** come un tipo di dato induttivo.

```
Inductive ceval : com -> stato -> stato -> Prop :=
| E_Skip : forall st,
    esegui CSkip in st ritorna st
| E_Ass : forall st a1 n x,
    aeval a1 st = n ->
    esegui (x ::= a1) in st ritorna (aggiorna_stato st x n)
| E_Seq : forall c1 c2 st st' st'',
    esegui c1 in st ritorna st' ->
    esegui c2 in st' ritorna st'' ->
    esegui (c1 ;; c2) in st ritorna st''
| E_IfTrue : forall st st' b c1 c2,
    beval b st = true ->
    esegui c1 in st ritorna st' ->
    esegui (CIf b c1 c2) in st ritorna st'
```

```
| E_IfFalse : forall st st' b c1 c2,
    beval b st = false ->
    esegui c2 in st ritorna st' ->
    esegui (CIf b c1 c2) in st ritorna st'
| E_WhileFalso : forall b st c,
    beval b st = false ->
    esegui (CWhile b c) in st ritorna st
| E_WhileVero : forall st st' st'' b c,
    beval b st = true ->
    esegui c in st ritorna st' ->
    esegui (CWhile b c) in st' ritorna st'' ->
    esegui (CWhile b c) in st ritorna st''

where "'esegui' c1 'in' st 'ritorna' st'" := (ceval c1 st st').
```

Per rendere più leggibile il codice, è stata introdotta una notazione che renda più chiaro ciò che sta avvenendo. In accordo con le regole, le valutazioni dell'If e del While sono state suddivise nei due casi in cui la guardia sia vera oppure falsa.

2.4 Stati

La struttura che assomiglia di più ad uno stato è il dizionario, il quale deve avere come chiavi le locazioni e come valori dei numeri interi, corrispondenti ai valori delle variabili assegnati a tali locazioni. In particolare, le locazioni sono così definite

```
Inductive loc : Type :=
  | Loc : nat -> loc.
```

In questo caso, la definizione del costruttore Loc non rappresenta direttamente il valore che si desidera inserire nella locazione di memoria n, ma assegna la posizione n a una variabile di tipo loc. In altre parole, Loc n crea un valore di tipo loc che rappresenta la locazione di memoria n, ma non include il valore effettivo da memorizzare in quella locazione. Per assegnare un valore a tale locazione bisogna utilizzare la struttura del dizionario, che associ una locazione di memoria a un valore.

```
Definition dizionario (A:Type) := loc -> A.
```

La definizione di stato vuoto inizializza tutte le locazioni allo stesso valore (nel nostro caso sarà 0).

```
Definition dizionario_vuoto {A:Type} (v : A) : dizionario A :=
  (fun _ => v).
```

Per aggiornare un dizionario bisogna definire una funzione di aggiornamento che vada a inserire il nuovo valore associato ad una locazione **x**, oppure che vada ad aggiornarlo se la locazione è già esistente.

```
Definition aggiorna_stato {A:Type} (m : dizionario A)
                    (x : loc) (v : A) :=
  fun x' => if loc_eq x x' then v else m x'.
```

Uno stato, quindi, è un dizionario che accetta come valori solo elementi di tipo Nat.

```
Definition stato := dizionario nat.
```

Infine, si può definire lo stato iniziale come uno stato vuoto, ovvero un dizionario in cui tutti i valori delle locazioni siano 0.

```
Definition stato_vuoto : stato :=
  dizionario_vuoto 0.
```

3 Esercizi

3.0.1 Esercizio 1

Data la definizione di equivalenza fra comandi proposta nelle slide, si è proceduto a tradurla in Coq.

```
Definition cequiv (c1 c2 : com) : Prop :=
   forall (st st' : stato),
   (esegui c1 in st ritorna st') <=> (esegui c2 in st ritorna st').
```

Il primo esercizio richiedeva di dimostrare il seguente teorema:

```
Theorem equivalenza_if_while: forall (b: bexp) (c: com),
  cequiv
    (While b do c)
    (If b then (c;; While b do c) else Skip).
Proof.
```

La dimostrazione viene descritta nel dettaglio direttamente sotto forma di commento nel file *imp.v*.

3.0.2 Esercizio 2

Il secondo esercizio chiedeva di dimostrare che, dato il programma

```
    while 1 <= x do (y:=2*y;x:=x-1)
```

e dato lo stato σ tale che per ogni locazione i, $\sigma(i) = 0$, allora $\exists \sigma* : \langle w, \sigma[2/x][3/y]\rangle \rightarrow \sigma*$. In pratica, bisognava dimostrare che, a partire da uno stato vuoto, ponendo come precondizione $X = 2$ e $Y = 3$, il programma sarebbe terminato in uno stato ben definito $\sigma*$, dove $\sigma * (X) = 0$ e $\sigma * (Y) = 12$. Per farlo, si è reso necessario seguire i cambiamenti di stato durante l'esecuzione,

ovvero si è dovuto fornire uno stato finale che tenesse conto di tutte le varie assegnazioni effettuate durante la processazione del ciclo while. Per semplicità nella trascrizione del codice, la variabile **stato_finale** nel codice rappresenterà l'effettivo stato finale riportato all'interno della dimostrazione, ovvero

```
Definition stato_finale : stato :=
(aggiorna_stato
 (aggiorna_stato
  (aggiorna_stato
   (aggiorna_stato
    (aggiorna_stato
     (aggiorna_stato stato_vuoto X 2) Y 3) X 1) Y 6) X 0) Y 12).
```

La dimostrazione viene descritta nel dettaglio direttamente sotto forma di commento nel file *imp.v*.