IMP

A.A. 2021/22, Jacopo Zagoli - Cesare Montresor

IMP è un linguaggio giocattolo imperativo che permette di manipolare:

- Numeri interi,
- Valori booleani,
- Locazioni di memoria.

Inoltre, dato che durante l'esecuzione dei programmi è necessario modificare i valori associati alle locazioni, viene introdotto il concetto di *store*, che associa ad ogni locazione di memoria un numero intero.

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

Open Scope Z.

Section ImpLanguage.
```

Store

Lo store rappresenta la memoria del programma, viene realizzato utilizzando una lista di coppie (locazione, intero). Abbiamo scelto di implementare le locazioni come un nuovo tipo con un solo costruttore LOC, che data una stringa costruisce una locazione.

Costruttore per il tipo Loc

```
Inductive Loc: Type := LOC: string → Loc.
```

Definiamo per semplicità il tipo dello store come una abbreviazione di `list(Loc×Z)`.

```
Definition storeT := list (Loc × Z).
```

Definiamo inoltre una funzione che calcola se due locazioni sono uguali, cioè se hanno lo stesso nome. Per fare questo estraiamo le stringhe dalle due locazioni e le confrontiamo con la funzione di uguaglianza tra stringhe.

```
Definition locEq (loc1 loc2: Loc) : bool :=
  match loc1, loc2 with
    LOC str, LOC str' ⇒ eqb str str'
  end.
```

Questa funzione ricorsiva, data una locazione, restituisce il valore presente nello store associato a quella locazione. Per fare questo controlla lo store una coppia alla volta: se la prima locazione è uguale a quella che ci interessa restituisce il valore, altrimenti richiama la stessa funzione sul resto della lista. Se la locazione richiesta non viene trovata restituisce zero, perchè nel testo del progetto ogni store contiene tutte le possibili variabili inizializzate a zero.

```
Fixpoint readLoc (loc: Loc) (store: storeT) {struct store} : Z:=
  match store with
    | (loc',n)::store' ⇒
        if locEq loc loc' then n else readLoc loc store'
    | nil ⇒ 0 (*stato iniziale*)
  end.
```

Questa funzione viene richiamata da assignLoc e effettua l'aggiornamento dello store ricorsivamente. Scorre la lista cercando la locazione indicata, se la trova, sostituisce la coppia con una nuova coppia (loc, val). Se non trova la location si limita ad aggiungera la nuova coppia. In ogni momento tiene in memoria la parte precedente e successiva alla coppia corrente, in modo da poter ricomporre correttamente lo store.

```
Fixpoint assignLocRec (loc: Loc) (head: storeT) (tail: storeT) (n: Z) {struct tail} : storeT:=
  match tail with
```

```
    | (currloc,currn)::tail' ⇒
        if locEq currloc loc then
            app ((loc,n)::head) tail'
        else
          assignLocRec loc ((currloc,currn)::head) tail' n
    | nil ⇒ (loc,n)::head
  end.
```

Questa funzione serve per modificare il valore di una locazione nello store. Richiama la sua implementazione assignLocRec.

```
Definition assignLoc (loc: Loc) (store: storeT) (n: Z) : storeT:=
  assignLocRec loc nil store n.
```

Test delle funzioni implementate

```
Definition mem := ( (LOC "A"), 1)::( (LOC "B"), 2)::( (LOC "C"), 3)::nil.
Compute readLoc (LOC "A") mem.
Definition mem2 := assignLoc (LOC "A") mem 2.
Compute readLoc (LOC "A") mem2.
Compute readLoc (LOC "B") mem2.
Reset mem. Reset mem2.
```

Sintassi di IMP

In questa sezione definiamo i tipi induttivi del nostro linguaggio: le espressioni aritmetiche, le espressioni booleane e i comandi. Ogni costruttore definisce una espressione o un comando diverso.

Espressioni Aritmetiche

Una espressione aritmetica può essere:

- un numero
- una locazione
- somma di due espressioni aritmetiche
- sottrazione di due espressioni aritmetiche
- moltiplicazione di due espressioni aritmetiche

*a := n | var | a0 + a1 | a0 - a1 | a0 * a1*

```
Inductive Aexpr: Type :=
  | N : Z → Aexpr
  | VAR : Loc → Aexpr
  | SUM : Aexpr → Aexpr → Aexpr
  | SUB : Aexpr → Aexpr → Aexpr
  | MUL : Aexpr → Aexpr → Aexpr
.
```

Espressioni Booleane

Una espressione booleana può essere:

- un valore di verità
- una uguaglianza tra espressioni booleane
- un confronto minore uguale tra espressioni booleane
- una negazione di espressioni booleane
- un and logico tra espressioni booleane
- un or logico tra espressioni booleane

b := true | false | a0 == a1 | a0 ≤ a1 | ¬ b | b0 ∧ b1 | b0 ∨ b1

```
Inductive Bexpr: Type :=
  | TT
  | FF
  | EQ : Aexpr → Aexpr → Bexpr
  | LEQ : Aexpr → Aexpr → Bexpr
  | NOT : Bexpr → Bexpr
  | AND : Bexpr → Bexpr → Bexpr
  | OR : Bexpr → Bexpr → Bexpr
```

.

Comandi

Un comando può essere:

- uno skip (non fa nulla)
- un assegnamento di una espressione aritmetica ad una locazione
- sequenza di due comandi
- un costrutto if
- un costrutto while

c := skip | X := a | c0;c1 | if b then c0 else c1 | while b do c

```
Inductive Com: Type :=
  | SKIP
  | ASS : Loc → Aexpr → Com
  | SEQ : Com → Com → Com
  | IF : Bexpr → Com → Com → Com
  | WHILE : Bexpr → Com → Com
.
```

Sematica di IMP

In questa sezione definiamo come vengono valutate le espressioni aritmetiche, le espressioni booleane ed i comandi.

Semantica operazionale delle espressioni aritmentiche

Con la parola chiave `Fixpoint` definiamo una funzione ricorsiva (totale, che quindi termina sempre). La funzione evalAexpr quindi valuta ricorsivamente una espressione aritmetica secondo le regole sotto indicate, restituendo come risultato un numero intero.

```
Fixpoint evalAexpr (aexpr: Aexpr) (store: storeT) : Z :=
  match aexpr with
    | N n ⇒ n
    | VAR loc ⇒ readLoc loc store
    | SUM e1 e2 ⇒ (evalAexpr e1 store) + (evalAexpr e2 store)
    | SUB e1 e2 ⇒ (evalAexpr e1 store) - (evalAexpr e2 store)
    | MUL e1 e2 ⇒ (evalAexpr e1 store) × (evalAexpr e2 store)
  end.
```

Semantica operazionale delle espressioni booleane

La seguente funzione ricorsiva valuta una espressione booleana secondo le regole definite dalla consegna, e restituisce il valore booleano associato.

```
Fixpoint evalBexpr (bexpr: Bexpr) (store: storeT) : bool :=
  match bexpr with
    | TT ⇒ true
    | FF ⇒ false
    | EQ e1 e2 ⇒ (Z.eqb (evalAexpr e1 store) (evalAexpr e2 store))
    | LEQ e1 e2 ⇒ (Z.leb (evalAexpr e1 store) (evalAexpr e2 store))
    | NOT e1 ⇒ negb (evalBexpr e1 store)
    | AND e1 e2 ⇒ andb (evalBexpr e1 store) (evalBexpr e2 store)
    | OR e1 e2 ⇒ orb (evalBexpr e1 store) (evalBexpr e2 store)
  end.
```

Semantica operazionale dell'esecuzione dei comandi

Per la valutazione dei comandi non abbiamo potuto definire un'altra funzione ricorsiva con `Fixpoint`: dal momento che nel linguaggio IMP il `while` può non terminare, avremmo dovuto codificare questo comportamento nella funzione. Coq però non accetta funzioni ricorsive non totali. Abbiamo quindi definito l'esecuzione dei comandi come un predicato induttivo: tramite esso non è possibile eseguire un comando con `Compute`, però è possibile dimostrare proprietà sui comandi stessi (ad esempio, che essi terminano in un certo stato, che sono equivalenti ad altri ecc...). Per definire questo predicato abbiamo creato un costruttore per ogni regola della semantica, trasformandole in assiomi.

```
Inductive execCommand : Com → storeT → storeT → Prop :=
  | E_SKIP : ∀ store: storeT,
                  execCommand SKIP store store
  | E_ASS : ∀ (store: storeT) (exp:Aexpr) (loc: Loc),
                  execCommand (ASS loc exp) store (assignLoc loc store (evalAexpr exp store))
  | E_SEQ : ∀ (s s' s'': storeT) (c1 c2: Com),
                  execCommand c1 s s' →
                  execCommand c2 s' s'' →
                  execCommand (SEQ c1 c2) s s''
  | E_IF_TRUE : ∀ (s s': storeT) (c1 c2: Com) (b: Bexpr),
                  evalBexpr b s = true →
                  execCommand c1 s s' →
                  execCommand (IF b c1 c2) s s'
  | E_IF_FALSE : ∀ (s s': storeT) (c1 c2: Com) (b: Bexpr),
                  evalBexpr b s = false →
                  execCommand c2 s s' →
                  execCommand (IF b c1 c2) s s'
  | E_WHILE_TRUE : ∀ (s s' s'': storeT) (c: Com) (b: Bexpr),
                  evalBexpr b s = true →
                  execCommand c s s'' →
                  execCommand (WHILE b c) s'' s' →
                  execCommand (WHILE b c) s s'
  | E_WHILE_FALSE : ∀ (s: storeT) (c: Com) (b: Bexpr),
                  evalBexpr b s = false →
                  execCommand (WHILE b c) s s
.
```

Equivalenza fra comandi

Il concetto di equivalenza fra due comandi è qui intuitivamente definito nel seguente modo: per ogni store arbitrario, eseguendo con quello store i due programmi, essi termineranno con lo stesso store finale. Questo concetto viene utilizzato nel primo teorema.

```
Definition comEq (c1 c2: Com) : Prop :=
  ∀ (s s': storeT),
    execCommand c1 s s' ↔ execCommand c2 s s'.

End ImpLanguage.
```

Teoremi

```
Section Teoremi.
```

Teorema 1

Questo teorema chiede di dimostrare l'equivalenza tra due programmi. Il primo programma è un semplice ciclo `while` generico, mentre il secondo è l'unrolling di un ciclo dello stesso `while`, ossia la esplicita valutazione della condizione tramite un if; se la condizione è positiva si esegue il comando più il `while` originario, altrimenti `skip`.

```
 Section Teorema_1.
   Axiom b: Bexpr. (* espressione booleana generica*)
   Axiom c: Com. (* comando generico*)

   Definition w := WHILE b c.
   Definition w' := IF b (SEQ c w) SKIP.

   Theorem unroll_while : comEq w w'.
   Proof.
   unfold comEq. (* sostituisco la definizione di equivalenza*)
   unfold w'. (* sostituisco le definizioni dei due programmi*)
   unfold w.
   intros. (* elimino il quantificatore universale*)
   split. (* spezzo la doppia implicazione nei due versi*)
   (* primo verso: while equivalente all'if*)
```

```
  - intro. inversion H.
  (* inversion: For a inductively defined proposition, inversion introduces a goal
  for each constructor of the proposition that isn't self-contradictory.
  Each such goal includes the hypotheses needed to deduce the proposition.*)
    + apply E_IF_TRUE. assumption.
(* applico i costruttori del comando più esterno: devo poi dimostrare la testa dell'implicazione*)
      apply E_SEQ with (s':=s''); assumption.
    + apply E_IF_FALSE. assumption. constructor.
  (* secondo verso: if equivalente al while*)
  - intro. inversion H.
    + inversion H6. subst. (*subst fa la riscrittura in entrambi i versi eliminando ipotesi inutili*)
      apply E_WHILE_TRUE with (s'':=s'1); assumption.
    + inversion H6. (* perchè il goal sia vero in questo caso ovviamente s deve essere uguale a s'*)
      subst.
      apply E_WHILE_FALSE; assumption.
  Qed.
End Teorema_1.
```

Teorema 2

Questo teorema chiede di dimostrare che il programma seguente:

```
σ = []
...
σ[2/x][3/y]
while ( 1 <= x ) {
    y = y * 2
    x = x - 1
}
```

Dato uno store arbitrario σ, inizializzato con x = 2 e y = 3, il programma termina in N passi
in un nuovo stato σ*.

```
Section Teorema_2.
  (* shortcut per LOC e VAR*)
  Definition x := (LOC "x").
  Definition y := (LOC "y").
  Definition var_x := (VAR x).
  Definition var_y := (VAR y).
  (* Utility per iniziare un qualsiasi store con stato iniziale e finale *)
  Definition initStore (store: storeT) := ( x, 2%Z )::( y, 3%Z)::store.
  Definition finalStore (store: storeT) := ( x, 0%Z )::( y, 12%Z)::store.
  (* Programma:
    while ( 1 <= x ) {
      y = y * 2
      x = x - 1
    }
  *)
  Definition prog :=
    WHILE (LEQ (N 1) var_x )
    ( SEQ
      ( ASS y (MUL var_y (N 2) ) )
      ( ASS x (SUB var_x (N 1) ) )
    ).

  (* Dato un qualsiasi store iniziale con include x = 2 e y = 3
     Esiste uno store fianle (aka il while termina sempre )
  *)
  Theorem while_step:
    ∀ s:storeT, ∃ s':storeT,
      execCommand prog (initStore s) s'.
  Proof.
    intros. (* estraggo il PerOgni store S*)
    ∃ (finalStore s). (* Definisco uno stato finale per exist *)
    eapply E_WHILE_TRUE. (* sostituisco il WHILE nei 3 subgoal di Inductive execCommand *)
    reflexivity. (* risolvo l'espressione booleana del WHILE: true -> esegui body *)
    - eapply E_SEQ. (* sostituisco il SEQ dividendo i due assegnamenti per la prima iterazione *)
      + apply E_ASS. (* sostituisco aggiorno lo store con il nuovo valore: Y = 6 *)
      + apply E_ASS. (* sostituisco aggiorno lo store con il nuovo valore: X = 1 *)
    - eapply E_WHILE_TRUE. (* sostituisco il WHILE nei 3 subgoal di Inductive execCommand *)
      reflexivity. (* risolvo l'espressione booleana del WHILE: true -> esegui body *)
      + eapply E_SEQ. (* sostituisco il SEQ dividendo i due assegnamenti per la seconda iterazione *)
        × apply E_ASS. (* sostituisco aggiorno lo store con il nuovo valore: Y = 12 *)
        × apply E_ASS. (* sostituisco aggiorno lo store con il nuovo valore: X = 0 *)
```

```
        + eapply E_WHILE_FALSE. (* sostituisco il WHILE con il goal definito in Inductive execCommand *)
          reflexivity. (* risolvo l'espressione booleana del WHILE: false -> fine *)
     Qed.
  End Teorema_2.
End Teoremi.

Close Scope Z.

(*Generate doc:*)
(* .\coqdoc.exe --no-externals --parse-comments --no-index --short -utf8 -
d "C:\Users\Jacopo\OneDrive\scuola\ragaut\ImpCoq\doc" "C:\Users\Jacopo\OneDrive\scuola\ragaut\ImpCoq\imp.v"*)
```
