Hoare Logic Proof Assistant

Chiara Fanco, Emilian Postolache, Andrea Proietto, Alessandro Steri

August 19, 2017

Abstract

Il progetto prevede lo sviluppo in ML di un *proof assistant* interattivo per la logica di Hoare. Lo scopo del documento è fornire le istruzioni per l'uso dell'applicazione, oltreché descrivere e motivare le scelte progettuali adottate, evidenziandone limiti e migliorie.

1 Overview dell'applicazione

L'applicazione supporta l'utente nel provare la correttezza parziale di un programma imperativo scritto nel linguaggio *Imp*. A tal fine prende in input una formula F della logica di Hoare che rappresenta la specifica del programma e ne dimostra la verità costruendo l'albero di derivazione mediante le regole di inferenza, denominate "*tactic*" in quanto segue, specificate dall'utente.

In ogni momento, l'applicazione mantiene in memoria lo stato corrente della dimostrazione. Esso consiste di una lista di formule che rappresentano le foglie dell'albero di derivazione costruito fino a quel momento.

Lo stato iniziale s_0 della dimostrazione contiene la sola formula F fornita dall'utente. A partire da uno stato s_t, l'utente genera lo stato successivo s_{t+1} specificando una *tactic* T ed una formula f di s_t a cui applicarla. s_{t+1} è ottenuto sostituendo f in s_t con le premesse derivate.

Se una formula è un assioma, viene rimossa dallo stato; ne consegue che F è dimostrata quando si raggiunge uno stato vuoto.

Alcune tactic introducono delle meta-variabili, ovvero dei segnaposti, laddove non sono in grado di calcolare la precondizione o la postcondizione all'interno di una formula. Andando avanti nella dimostrazione è possibile inserire delle espressioni

2 Struttura dell'applicazion

L'applicazione consiste di 6 moduli:

- **Hoare**: definisce la sintassi astratta del linguaggio Imp e delle formule di Hoare. Include, inoltre, delle funzioni di utility per la manipolazione delle formule.

- **Lexer**: implementa un analizzatore lessicale che si occupa di suddividere in token le stringhe passate in input.
- **Parser**: analizza i token prodotti dal Lexer e li traduce in formule applicando i costruttori sintassi astratta.
- **Printer**: converte le formule in stringhe
- **Rule**: definisce le funzioni che implementano le tactic.
- **Controller**: espone i comandi mediante i quali l'utente interagisce con l'applicazione.

3 Sintassi astratta

La sintassi astratta è rappresentata sottoforma dei seguenti datatype:

```
datatype numExp = Num  of int
                | Var  of string
                | Plus of numExp * numExp

datatype boolExp = Bool    of bool
                 | Not     of boolExp
                 | Meta    of string
                 | MetaVal of string * boolExp
                 | And     of boolExp * boolExp
                 | Or      of boolExp * boolExp
                 | Impl    of boolExp * boolExp
                 | Minor   of numExp * numExp
                 | Equal   of numExp * numExp

datatype prog = Skip
              | Comp   of prog * prog
              | Assign of string * numExp
              | While  of boolExp * prog
              | If     of boolExp * prog * prog

datatype form = Prop   of boolExp
              | Triple of boolExp * prog * boolExp
```

Va notato che rispetto alla sintassi astratta della sezione 10.2 delle Note sono stati aggiunti per le espressioni booleane due costruttori: Meta e MetaVal.

Il costruttore Meta definisce una meta-variabile "$\_a$" che segna il posto per un'espressione booleana all'interno di una formula nel caso in cui tale espressione non possa essere calcolata dalla tactic nello stato corrente.

Quando si applica una tactic ad una formula contenente una meta-variabile "$\_a$" per cui sia possibile calcolarne l'espressione booleana exp, si definisce mediante il costruttore MetaVal un meta-valore ovvero una coppia $(\_a, exp)$ che associa exp ad "$\_a$" e che nell'applicazione è scritta come "$\_a : exp$".

4 Tactics

(`DA FARE`, scrivere i tipi definiti in Rule e poi le signature delle tactic, inoltre sarebbe opportuno indicare da qualche parte la sintassi astratta affinche l'utente sappia come inserire l'input da stringa, operatori associativi a destra per disambiguare le parentesi, si mette l'end alla fine di if e di while, i programmi non vogliono le parentesi e che comp ha un problema?; e' sparito il pezzo che diceva per quale motivo usavamo meta-valori e non ho messo l'uno nella guida ma tanto abbiamo detto che esiste)

Ci sono due tipi di tactic: le base tactic e le meta tactic. Le prime non introducono meta-variabili e sono:

1. tacAxiom
2. tacSkip
3. tacAssign
4. tacWhile
5. tacIf

Le seconde introducono meta-variabili e sono:

6. tacStr
7. tacWeak
8. tacComp

La tactic tacAxiom verifica che la formula su cui è invocata sia effettivamente un assioma e, in tal caso, la elimina dallo stato. Le due tattiche tacSkip e tacAxiom, pur corrispondendo a degli assiomi sono diverse da tacAxiom poichè sono utilizzate per calcolare le precondizioni o le post condizioni quando al loro posto sono presenti meta-variabili sui programmi Skip e negli assegnamenti. Tutte le altre tactic sono semplicemente un implementazione delle corrispondenti regole di inferneza.
A questi due gruppi si aggiungono altre due tactic "speciali":

9. tacNorm
10. tacMeta

La tactic (9), applicata ad una formula che contiene un meta-valore della forma "$\_a : exp$", sostituisce con exp tutte le meta-variabili e i meta-valori di nome "$\_a$" presenti nello stato. Tale processo prende il nome di "normalizzazione".

La tactic (10), prende come argomenti una stringa "$\_a$" ed un espressione booleana exp e sostituisce con exp tutte le meta-variabili e i meta-valori di nome "$\_a$" presenti nello stato. Questa tactic permette all'utente di fornire manualmente il valore associato ad una meta-variabile.

5 Guida pratica all'utilizzo

In questa sezione si descrive l'interfaccia esposta dal Controller dopodichè si da un esempio pratico di utilizzo.

- **goal**: prende in input una tripla di Hoare come stringa ed inizializza lo stato.
- **by**: prende in input una tactic e l'indice della formula su cui la si vuole applicare e aggiorna lo stato con le nuove premesse. In caso di fallimento restituisce un opportuno messaggio. La tactic (10) non può essere passata come argomento a questa funzione. La si può applicare chiamando meta.
- **meta**: prende in in input il nome di una meta-variabile ed un espressione booleana ed invoca tacMeta.
- **pr**: stampa lo stato corrente.
- **undo**: anulla l'ultima modifica allo stato ripristinando quello precedente.
- **getState**: restituisce lo stato corrente.

Come esempio di utilizzo deriviamo la seguente tripla di Hoare:

$$\{x = 1\} skip;\ if(x < 0)\ then\ x := x + 1\ else\ x := x + 2\ end \{x = 2\}$$

Il primo passo consiste nel posizionarsi con il terminale nella directory del progetto ed avviare il REPL di ML:

```
Standard ML of New Jersey v110.78 [built: Thu Jul 23 11:21:58 2015]
-
```

Si aprono i file dell'applicazione:

```
Standard ML of New Jersey v110.78 [built: Thu Jul 23 11:21:58 2015]
-  use "files.sml";
...
```

Viene aperta la structure Controller per ottenere l'interfaccia:

```
- open Controller;
opening Controller
  val pr : unit -> unit
  val getState : unit -> Rule.state
  val goal : string -> unit
  val by : Rule.tactic -> unit
  val meta : string -> string -> unit
  val undo : unit -> unit
-
```

Inseriamo la formula da dimostrare utilizzando `goal`. Ad ogni passo della dimostrazione viene stampato lo stato attuale numerando le formule presenti. In questo caso si ha solo il goal iniziale:

```
- goal "{x=0}skip;if(x<0)then x:=x+1 else x:=x+2 end{x=2}";
1. {x = 0}skip; if (x < 0) then x := x + 1 else x := x + 2 end{x = 2}
val it = () : unit
-
```

Applichiamo la tactic `tacComp` sull'unica formula presente. Per fare ciò si usa la funzione `by`. Poichè `tacComp` è una meta rule si ha l'aggiunta della metavariabile "$\_a$":

```
- by(Rule.tacComp 1);
1. {x = 0}skip{_a}
2. {_a}if (x < 0) then x := x + 1 else x := x + 2 end{x = 2}
val it = () : unit
-
```

Siccome nello stato sono presenti due premesse si può scegliere quale delle due attacare. In questo caso applichiamo la tactic `tacIf` sulla seconda formula:

```
- by(Rule.tacIf 2);
1. {x = 0}skip{_a}
2. {_a & x < 0}x := x + 1{x = 2}
3. {_a & x >= 0}x := x + 2{x = 2}
val it = () : unit
-
```

Applichiamo successivamente `tacStr` sulla terza formula e `tacAssign` sulla formula 4 del nuovo stato.

Si nota che la meta-variabile "_b" é diventata il meta-valore "_b : $x + 2 = 2$ essendo stato calcolato dalla tactic `tacAssign`:

```
- by(Rule.tacStr 3);
1. {x = 0}skip{_a}
2. {_a & x < 0}x := x + 1{x = 2}
3. _a & x >= 0 -> _b
4. {_b}x := x + 2{x = 2}
val it = () : unit
- by(Rule.tacAssign 4);
1. {x = 0}skip{_a}
2. {_a & x < 0}x := x + 1{x = 2}
3. _a & x >= 0 -> _b
4. {_b : x + 2 = 2}x := x + 2{x = 2}
val it = () : unit
-
```

Normalizziamo la meta-variabile "_b" utilizzando `tacNorm`; tutte le occorenze di "_b" diventano l'espressione $x + 2 = 2$:

```
- by(Rule.tacNorm 4);
1. {x = 0}skip{_a}
2. {_a & x < 0}x := x + 1{x = 2}
3. _a & x >= 0 -> x + 2 = 2
4. {x + 2 = 2}x := x + 2{x = 2}
val it = () : unit
-
```

Siccome la formula 4 è un assioma, può essere eliminato con `tacAxiom`:

```
- by(Rule.tacAxiom 4);
1. {x = 0}skip{_a}
2. {_a & x < 0}x := x + 1{x = 2}
3. _a & x >= 0 -> x + 2 = 2
val it = () : unit
-
```

Osserviamo che la formula 3 non è una tripla di Hoare ma una espressione aritmetica. Per poter eliminare la formula 3 come assioma, la meta-variabile "_a" deve acquistare un valore. Tale valore deve essere fornito dall'utente utilizzando la funzione `meta`. In questo caso il processo di normalizzazione è automatico:

```
- meta "_a" "x = 0";
1. {x = 0}skip{x = 0}
2. {x = 0 & x < 0}x := x + 1{x = 2}
3. x = 0 & x >= 0 -> x + 2 = 2
val it = () : unit
```

A partire dall'ultimo stato, applicando in sequenza `by(Rule.tacStr 2)`, `by(Rule.tacAssign 3)` e `by(Rule.tacNorm 3)` si ottiene il seguente stato (notare l'utilizzo della funzione `pr`):

```
- pr();
1. {x = 0}skip{x = 0}
2. x = 0 & x < 0 -> x + 1 = 2
3. {x + 1 = 2}x := x + 1{x = 2}
4. x = 0 & x >= 0 -> x + 2 = 2
val it = () : unit
-
```

Ovviamiente tutti e 4 sono degli assiomi e possono essere eliminati facilmente con *tacAxiom*. Supponendo che la formula numero 1 è l'ultima da eliminare finiamo la nostra dimostrazione nel seguente modo:

```
- pr();
1. {x = 0}skip{x = 0}
val it = () : unit
- by(Rule.tacAxiom 1);
No subgoals left! Milner says: <<Good job bro!>>
val it = () : unit
-
```