Teoria della computabilità in HOL

Marco Maggesi
(lavoro in collaborazione con Andrea Gabrielli)
Seminario di Logica e Filosofia della Scienza
Firenze, 5 maggio 2017

Macchine di Turing

- Alfabeto: **`One, Blank, Mark : alph`**.
- Nastro: Sequenza di caratteri
  (Esempio: **`One`**, **`Blank`**, **`One`**, **`One`**, **`One`**, **`Mark`**, **`One`**, **`Blank`**, **`...`**)
  (Scrittura compatta: **`<|.111.M1... … …|>`**)
  (Tecnicamente: funzione **`num -> alph`**)
- Stato: un numero naturale.

Macchine di Turing

- Configurazione: **`Conf s x f`**
  - **`s`** = stato
  - **`x`** = cursore
  - **`f`** = nastro

- Istruzione: **`Instr s u v d t`**
  - **`s`** = stato iniziale
  - **`u`** = carattere letto
  - **`v`** = carattere scritto
  - **`d`** = direzione (**`T`** = destra, **`F`** = sinistra)
  - **`t`** = stato finale

- Programma: **`p`**

  insieme (finito) di istruzioni

Esempio: successore

```
|- HALT { Instr 0 One   One   T 0,
         Instr 0 Blank One   F 1,
         Instr 1 One   One   F 1,
         Instr 1 Blank Blank T 2 }
(Conf 0 1 <|.111|>)
(Conf 2 1 <|.1111|>)
```

```
let RELEVANT_RULES,RELEVANT_INDUCT,RELEVANT_CASES =
  new_inductive_definition
  `!s x f v d t. RELEVANT (Conf s x f) (Instr s (f x) v d t)`;;

let MOVE = new_recursive_definition bool_RECURSION
  `MOVE T = SUC /\ MOVE F = PRE`;;

let NEXT = new_definition
  `NEXT c i = Conf (final i)
                   (MOVE (move i) (cursor c))
                   (UPDATE (tape c) (cursor c) (write i))`;;
```

```
let STEP_RULES,STEP_INDUCT,STEP_CASES = new_inductive_definition
  `!p i c. i IN p /\ RELEVANT c i ==> STEP p c (NEXT c i)`;;

let EXEC = new_definition `EXEC p = RTC (STEP p)`;;

let FINAL = new_definition
  `FINAL p c <=> ~(?i. i IN p /\ RELEVANT c i)`;;

let HALT = new_definition
  `HALT p c c' <=> EXEC p c c' /\ FINAL p c'`;;
```

Polimorfismo

- Variabili di tipo:
  A = tipo dell'alfabeto
  S = tipo degli stati

- **`Conf s x f       : (A,S)conf`**

- **`Instr s u v d t : (A,S)instr`**

- Esempio 1 (Il nostro caso di studio: alfabeto di 3 simboli e stati numerici):
  A = {**One**, **Blank**, **Mark**},
  S = **num**

- Esempio 2 (Alfabeto con 2 simboli e 256 stati)
  A = **bool**,
  B = **char**

- Alcune modifiche permetterebbero ulteriori interessanti generalizzazioni.
  (Esempi: nastro bilatero, nastro bidimensionale, nastro finito, …)

STEP: definizione

```
let STEP_RULES,STEP_INDUCT,STEP_CASES = new_inductive_definition
  `!p i c. i IN p /\ RELEVANT c i ==> STEP p c (NEXT c i)`;;

STEP_RULES
|- !p i c. i IN p /\ RELEVANT c i ==> STEP p c (NEXT c i)

STEP_INDUCT
|- !p R. (!i c. i IN p /\ RELEVANT c i ==> R c (NEXT c i))
         ==> (!c c'. STEP p c c' ==> R c c')

STEP_CASES
|- !p c c'. STEP p c c' <=>
            (?i. c' = NEXT c i /\ i IN p /\ RELEVANT c i)
```

STEP: alcuni corollary

```
STEP_MONO
|- !p p' c c'. p SUBSET p' /\ STEP p c c' ==> STEP p' c c'

STEP
|- (!c c'. ~STEP {} c c') /\
   (!i p c c'.
      STEP (i INSERT p) c c' <=>
      RELEVANT c i /\ c' = NEXT c i \/ STEP p c c')

STEP_IMP_STEP2
|- !p si sf x y f g.
     STEP p (Conf si x f) (Conf sf y g)
     ==> STEP {Instr (2*s) u v d (2*t) | Instr s u v d t IN p}
              (Conf (2*si) x f)
              (Conf (2*sf) y g)
```

EXEC: alcuni corollari

```
EXEC_REFL
|- !p c. EXEC m c c

EXEC_MONO
|- !p p' c1 c2. p SUBSET p' /\ EXEC p c1 c2 ==> EXEC p' c1 c2

EXEC_EMPTY
|- !c c'. EXEC {} c c' <=> c = c'

EXEC_STEP_LEFT
|- !p c c'. EXEC p c c' <=>
            c = c' \/ (?b. STEP p c b /\ EXEC p b c')

EXEC_IMP_EXEC2
|- !p c1 c2.
     EXEC p c1 c2
     ==> EXEC {Instr (2*s) u v d (2*t) | Instr s u v d t IN p}
              (Conf (2 * status c1) (cursor c1) (tape c1))
              (Conf (2 * status c2) (cursor c2) (tape c2))
```

Mechanising Turing Machines and Computability Theory in Isabelle/HOL

Jian Xu[1], Xingyuan Zhang[1], and Christian Urban[2]

[1] PLA University of Science and Technology, China
[2] King's College London, UK

Abstract. We formalise results from computability theory in the theorem prover Isabelle/HOL. Following the textbook by Boolos et al, we formalise Turing machines and relate them to abacus machines and recursive functions. We "tie the knot" between these three computational models by formalising a universal function and obtaining from it a universal Turing machine by our verified translation from recursive functions to abacus programs and from abacus programs to Turing machine programs. Hoare-style reasoning techniques allow us to reason about concrete Turing machine and abacus programs.

1 Introduction

We like to enable Isabelle/HOL users to reason about computability theory. Reasoning about decidability of predicates, for example, is not as straightforward as one might think in Isabelle/HOL and other HOL theorem provers, since they are based on classical logic where the law of excluded middle ensures that $P \vee \neg P$ is always provable no matter whether the predicate P is constructed by computable means.

Norrish formalised computability theory in HOL4. He choose the λ-calculus as the starting point for his formalisation because of its "simplicity" [7, Page 297]. Part of his formalisation is a clever infrastructure for reducing λ-terms. He also established the computational equivalence between the λ-calculus and recursive functions. Nevertheless he concluded that it would be appealing to have formalisations for more operational models of computations, such as Turing machines or register machines. One reason is that many proofs in the literature use them. He noted however that [7, Page 310]:

> *"If register machines are unappealing because of their general fiddliness,*

Rappresentazione concreta del nastro

- **`CONST a`** funzione costante uguale ad **`a`**

```
|- !a x. CONST a x = a
```

- Nastro *vuoto*: **`CONST Blank`** (<|…… … … … … |>)

- **`INS x f`** aggiunge **`x`** in testa ad **`f`**

```
|- !x f.   INS x f 0       = x
|- !x f n. INS x f (SUC n) = f n
```

- Esempio:

```
`<|.11.1|>` = `INS Blank (INS One (INS One (INS Blank
                                    (INS One (CONST Blank)))))`
```

UPDATE

```
let UPDATE = new_definition
  `!x v f. UPDATE f x v =
           (\y. if y = x then v else f y)`;;
```

```
UPDATE_CONST
|- (!a n. UPDATE (CONST a) n a = CONST a) /\
   (!a b. UPDATE (CONST a) 0 b = INS b (CONST a)) /\
   (!a b n. UPDATE (CONST a) (SUC n) b =
            INS a (UPDATE (CONST a) n b))

UPDATE_INS
|- (!f a b. UPDATE (INS a f) 0 b = INS b f) /\
   (!f a b n. UPDATE (INS a f) (SUC n) b = INS a (UPDATE f n b))
```

Computabilità

```
let TAPE_OF_NUM = new_definition
  `TAPE_OF_NUM n =
   INS Blank (ITER n (INS One) (INS One (CONST Blank)))`;;
```

Esempio: `|- TAPE_OF_NUM 3 = <|.111|>`

```
let TURING_COMPUTABLE = new_definition
  `TURING_COMPUTABLE (f:num->num) <=>
   ?p s t. !n.
     HALT p
          (Conf s 1 (TAPE_OF_NUM n))
          (Conf t 1 (TAPE_OF_NUM (f n)))`;;
```

Esempio: `|- TURING_COMPUTABLE SUC`

Composizione

```
|- !f g. TURING_COMPUTABLE f /\
        TURING_COMPUTABLE g
        ==> TURING_COMPUTABLE (f o g)
```

Esempio:

```
|- TURING_COMPUTABLE (\x. x + 2)
```

Esecuzione certificata

Esempio: Somma

Input: `` `2 + 2` ``

Output: `|- 2 + 2 = 4`

Nel nostro caso

Input: `` `HALT p c` ``

Output: `|- HALT p c = {c1, c2, …}`

(Equivalente a

`|- HALT p c c1 /\ HALT p c c2 /\ …`)

Esecuzione certificata

Input:

```
`HALT {Instr 0 One   One   T 0,
       Instr 0 Blank One   F 1,
       Instr 1 One   One   F 1,
       Instr 1 Blank Blank T 2}
      (Conf 0 1 <|.111|>)`
```

Output:

```
|- HALT {Instr 0 One   One   T 0,
        Instr 0 Blank One   F 1,
        Instr 1 One   One   F 1,
        Instr 1 Blank Blank T 2}
       (Conf 0 1 <|.111|>)
   =
   { Conf 2 1 <|.1111|> }
```

Esecuzione certificata

Esempio: macchina non deterministica

Input:

```
`HALT {Instr 1 Blank Blank T 0,
       Instr 1 Blank One   T 0}
      (Conf 1 1 (CONST Blank))`
```

Output:

```
|- HALT {Instr 1 Blank Blank T 0,
        Instr 1 Blank One   T 0}
       (Conf 1 1 <||>)
   =
   { Conf 0 2 <||>,
     Conf 0 2 <|.1|> }
```

Esecuzione certificata

Esempio: non-terminazione

Input:

```
`HALT {Instr 1 Blank One T 1}
 (Conf 1 1 (CONST Blank))`
```

Output:

Esempio: scorrimento a sx

```
|- HALT {Instr  0 One   Blank T  1, Instr  1 One   One   F  2,
        Instr  2 Blank Blank F  2, Instr  2 Mark  Mark  T  3,
        Instr  2 One   One   T  3, Instr  3 Blank One   T  4,
        Instr  4 Blank Blank T  4, Instr  4 One   Blank T  1,
        Instr  1 Blank Blank F 22, Instr 22 Blank Blank F 22,
        Instr 22 Mark  Mark  T 33, Instr 22 One   One   T 33,
        Instr 33 Blank One   F  5, Instr  5 One   One   F  5,
        Instr  5 Mark  Blank T  6}
(Conf 0 5 <|M.......111|>)
=
{Conf 6 1 <|.111|>}
```