Mechanising Turing Machines and Computability Theory in Isabelle

Jian Xu Xingyuan Zhang

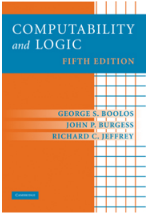
PLA University of Science and Technology

Christian Urban

King's College London

Why Turing Machines?

- At the beginning, it was just a student project about computability.



Computability and Logic (5th. ed)
Boolos, Burgess and Jeffrey

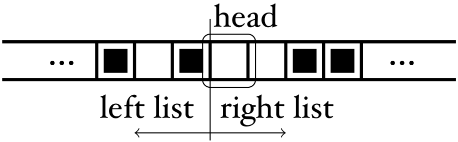
- found an inconsistency in the definition of halting computations (Chap. 3 vs Chap. 8)

Some Previous Work

- Norrish formalised computability theory in HOL starting from the lambda-calculus
  - for technical reasons we could not follow him
  - some proofs use TMs (Wang tilings)
- Asperti and Ricciotti formalised TMs in Matita

Turing Machines

- tapes are lists and contain *0*s or *1*s only

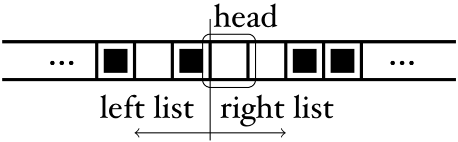


- *steps* function:

  What does the TM claclulate after it has executed *n* steps?

Turing Machines

- tapes are lists and contain *0*s or *1*s only



- *steps* function:

  What does the TM claclulate after it has executed *n* steps?

- designate the *0*-state as □alting stateänd remain there forever, i.e. have a *Nop*-action

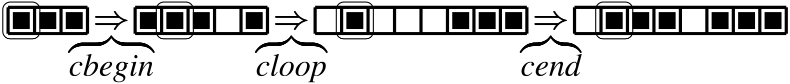
Register Machines

- instructions

$$
\begin{array}{rcll}
I & ::= & \mathit{Inc}\ R & \text{increment register } R \text{ by one} \\
& | & \mathit{Dec}\ R\ L & \text{if content of } R \text{ is non-zero,} \\
& & & \text{then decrement it by one} \\
& & & \text{otherwise jump to instruction } L \\
& | & \mathit{Goto}\ L & \text{jump to instruction } L
\end{array}
$$

Copy Turing Machines

- TM that copies a number on the input tape



$$cbegin \stackrel{def}{=} [(W_0, 0), (R, 2), (R, 3), (R, 2), (W_1, 3), (L, 4), (L, 4), (L, 0)]$$

$$cloop \stackrel{def}{=} [(R, 0), (R, 2), (R, 3), (W_0, 2), (R, 3), (R, 4), (W_1, 5), (R, 4), (L, 6), (L, 5), (L, 6), (L, 1)]$$

$$cend \stackrel{def}{=} [(L, 0), (R, 2), (W_1, 3), (L, 4), (R, 2), (R, 2), (L, 5), (W_0, 4), (R, 0), (L, 5)]$$

Hoare Logic for TMs

- Hoare-triples and Hoare-pairs:

$\{P\}\ p\ \{Q\} \stackrel{def}{=}$

$\forall\ tp.$

if $P\ tp$ holds then

$\exists\ n.$ such that

$is\_final\ (steps\ (1,\ tp)\ p\ n)\ \wedge$

$Q\ holds\_for\ (steps\ (1,\ tp)\ p\ n)$

$\{P\}\ p \uparrow \stackrel{def}{=}$

$\forall\ tp.$

if $P\ tp$ holds then

$\forall\ n.\ \neg\ is\_final\ (steps\ (1,\ tp)\ p\ n)$

Hoare Reasoning

- reasoning is still quite demanding;
  the invariants of the copy-machine:

$I_1\ n\ (l, r) \stackrel{def}{=} (l, r) = ([], 1^n)$		(starting state)
$I_2\ n\ (l, r) \stackrel{def}{=} \exists\, i\, j.\ 0 < i \wedge i + j = n \wedge (l, r) = (1^i, 1^j)$		
$I_3\ n\ (l, r) \stackrel{def}{=} 0 < n \wedge (l, tl\ r) = (0{::}1^n, [])$		
$I_4\ n\ (l, r) \stackrel{def}{=} 0 < n \wedge (l, r) = (1^n, [0, 1]) \vee (l, r) = (1^{n-1}, [1, 0, 1])$		
$I_0\ n\ (l, r) \stackrel{def}{=} 1 < n \wedge (l, r) = (1^{n-2}, [1, 1, 0, 1]) \vee$ $n = 1 \wedge (l, r) = ([], [0, 1, 0, 1])$		(halting state)
$J_1\ n\ (l, r) \stackrel{def}{=} \exists\, i\, j.\ i + j + 1 = n \wedge (l, r) = (1^i, 1{::}1{::}0^j @ 1^j) \wedge 0 < j \vee$ $0 < n \wedge (l, r) = ([], 0{::}1{::}0^n @ 1^n)$		(starting state)
$J_0\ n\ (l, r) \stackrel{def}{=} 0 < n \wedge (l, r) = ([0], 1{::}0^n @ 1^n)$		(halting state)
$K_1\ n\ (l, r) \stackrel{def}{=} 0 < n \wedge (l, r) = ([0], 1{::}0^n @ 1^n)$		(starting state)
$K_0\ n\ (l, r) \stackrel{def}{=} 0 < n \wedge (l, r) = ([0], 1^n @ 0{::}1^n)$		(halting state)

Recursive Functions

- addition, multiplication, ...
- logical operations, quantifiers...
- coding of numbers (Cantor encoding)
- UF

Recursive Functions

- addition, multiplication, ...
- logical operations, quantifiers...
- coding of numbers (Cantor encoding)
- UF

- Recursive Functions $\Rightarrow$ Register Machines
- Register Machines $\Rightarrow$ Turing Machines

Sizes

- UF (size: *140843*)
- Register Machine (size: *2* Mio instructions)
- UTM (size: *38* Mio states)

old version: RM (*12* Mio) UTM (*112* Mio)

Separation Algebra

- introduced a separation algebra framework for register machines and TMs
- we can semi-automate the reasoning for our small TMs
- we can assemble bigger programs out of smaller components
- looks awfully like ``real'' assembly code

Separation Algebra

- introduced a separation algebra framework for register machines and TMs
- we can semi-automate the reasoning for our small TMs
- we can assemble bigger programs out of smaller components

- looks awfully like ``real'' assembly code
- Conclusion: we have a playing ground for reasoning about low-level code; we work on automation