Simulateur de machines de Turing

Florent Marchand de Kerchove

29 janvier 2011

Une machine de Turing est une construction volontairement simple, puisque destinée à être manipulée par un humain, et non par un automate capable d'effectuer des centaines de transitions par seconde. Un simulateur est donc assez court à décrire.

Nous avons choisi le Common Lisp pour écrire ce programme, sans raison pratique particulière ; la justification est plutôt culturelle. Le Lisp est un ancêtre parmi les langages de programmation, tout comme les machines de Turing sont les ancêtres des ordinateurs, et pourtant tous deux sont loin d'être obsolètes.

Implémentation en Common Lisp

Pour simuler une machine, il nous faut la représenter en mémoire. En Lisp, la structure de liste est favorite, mais pas forcément efficace. Nous utiliserons donc également des tableaux et des vecteurs fournis par le Common Lisp.

Afin de définir la machine, nous n'avons besoin que de ses transitions et de son état initial ; tous les symboles de l'alphabet d'entrée, de sortie, et tous les états qui ne sont pas mentionnés dans les transitions ne seront pas utiles au simulateur, puisque jamais utilisés par la machine. Si l'on souhaite définir des machines acceptantes, il nous faudra également préciser la liste des états finaux. Enfin, nous laissons la possibilité de définir le symbole blanc, en utilisant le # par défaut, conventionnellement.

Les règles de transition seront donc passés dans une liste de vecteurs. Chaque vecteur spécifie une règle de la forme : `#(p a q b x)`, où p, q sont des états, a, b des symboles et $x \in$ "L", "R". Notons que le Lisp n'impose pas de restriction sur les types. Nos états et symboles peuvent donc être spécifiés par des entiers, des chaînes de caractères, des listes ou même des machines de Turing ! Bien sûr, dans la pratique il sera plus aisé d'utiliser un type atomique comme les entiers ou les chaînes. Par exemple, la règle suivante : `#(0 "a" 1 "b" "" "R")` remplace le symbole a par b sur la bande, si l'état courant est 0, et passe alors à l'état 1, en décalant la tête de lecture sur la droite.

L'état initial est spécifié par le symbole qui le représente dans les règles de transition, et les états finaux sont passés dans une liste, qui peut être vide dans le cas d'une machine en mode calculateur. Enfin le symbole blanc spécial est également librement choisi, mais nous utiliserons # dans nos exemples.

Tout ceci nous permet de modéliser la machine, mais il nous manque un élément crucial : la bande d'entrée et de sortie. Cette bande sera représentée par un tableau à dimension variable (finie, sauf si vous disposez d'une machine à mémoire infinie), et la position de la tête de lecture sera conservée dans un entier qui indice la bande à partir de zéro.

Ces informations en main, il nous suffit, pour simuler la machine sur une bande donnée, de faire une boucle qui exécute la transition suivante selon la donnée présente sous la tête de lecture et l'état courant, tant qu'une telle transition est possible. Lorsque la boucle s'arrête, la bande modifiée par le machine est affichée. Cette boucle contiendra en plus un test qui vérifiera si l'état courant est un état final, auquel cas le programme affichera le message « calcul acceptant ».

Nous ne prenons pas de précautions particulière pour garantir le « bon » fonctionnement de la machine : l'utilisateur est responsable des machines à simuler, et nous du simulateur. En particulier, c'est à lui de s'assurer que l'état initial figure bien dans les transitions, de même pour les éáts finaux. La programme vérifie un point : que la tête de lecture ne passe pas à gauche de l'origine de la bande, ce qui n'est pas accepté sur les machines du modèle standard.

Exemples d'utilisation

Pour tester le simulateur, il faut tout d'abord lancer un interpréteur de Common Lisp. Nous utilisons celui fourni par GNU [1]. Ensuite, il faut charger le fichier "turing.lisp" :

```
> (load "turing.lisp")
;; Loading file turing.lisp ...
;; Loaded file turing.lisp
T
```

Ensuite, il suffit d'appeler la fonction *run-machine* sur une machine construite avec la fonction *make-machine* et une bande retournée *make-tape*, comme ceci :

```
> (run−machine (make−machine '(#(0 "#" 1 0 "R")
                               #(0 0 0 0 "R")) 0)
             (make−tape '(0 0 0)))
"0000"
```

Cette machine calcule le successeur du nombre passé en entrée, sous la forme standard 0^n. Le premier argument de *make-machine* est la liste des transitions. La première, (0 "#"1 0 "R"), transforme le # en 0 si l'état courant est l'état 0, qui passe alors à l'état 1, en déplaçant la tête de lecture sur la droite. La seconde règle est une boucle sur l'état 0 qui lit les 0 sans les effacer. Le second argument de *make-machine* est l'état initial, 0. Les états finaux ne sont pas utilisés dans cette machine en mode calculateur, donc pas passés en argument. Le seul argument de *make-tape* est la donnée d'entrée sur la bande sous forme de liste de symboles de l'alphabet d'entrée ; ici ce sont 3 zéros. Enfin, l'interpréteur Lisp renvoie la chaîne "0000" qui représente la bande en fin de calcul (les # superflus de gauche et de droite sont omis).

Nous avons décrit dans la suite quelques autre machines élémentaires pour illustrer le fonctionnement du simulateur.

```
;; n −> n−1
> (run−machine (make−machine '(#(0 0 1 "#" "R")) 0)
             (make−tape '(0 0 0)))
"00"

;; n −> 0 (n=0) | 1
```

1. `http://clisp.cons.org`

```
> (run−machine (make−machine '(#(0 0 1 0 "R")
                                #(1 0 1 "#" "R")) 0)
               (make−tape '(0 0)))
"0"

;; n −> 1 (n=0) | 0
> (run−machine (make−machine '(#(0 0 1 "#" "R")
                                #(0 "#" 1 0 "R")
                                #(1 0 1 "#" "R")) 0)
               (make−tape '(0 0)))
""

;; x −> x mod 2
> (run−machine (make−machine '(#(0 0 1 "#" "R")
                                #(1 0 0 "#" "R")
                                #(1 "#" 0 0 "R")) 0)
               (make−tape '(0 0 0 0 0)))
"0"

;; (x,y) −> x
> (run−machine (make−machine '(#(0 0 0 0 "R")
                                #(0 1 1 "#" "R")
                                #(1 0 1 "#" "R")) 0)
               (make−tape '(0 0 1 0 0 0)))
"00"

;; (x,y) −> y
> (run−machine (make−machine '(#(0 0 0 "#" "R")
                                #(0 1 1 "#" "R")) 0)
               (make−tape '(0 0 1 0 0 0)))
"000"

;; (x,y) −> x+y
> (run−machine (make−machine '(#(0 0 1 "#" "R")
                                #(0 1 1 "#" "R")
                                #(1 1 1 0 "R")
                                #(1 0 1 0 "R")) 0)
               (make−tape '(0 0 1 0 0 0)))
"00000"

;; n −> even(n) = (n+1) mod 2
;; n is in binary little−endian.
> (run−machine (make−machine '(#(0 0 1 0 "R")
                                #(0 1 1 1 "R")
                                #(1 "#" 2 "#" "L")
                                #(2 0 2 1 "R")
                                #(2 1 2 0 "R")
                                #(1 0 3 0 "L")
                                #(1 1 3 1 "L")
                                #(3 0 0 "#" "R")
                                #(3 1 0 "#" "R")) 0)
               (make−tape '(1)))
"0"
```

```
;; Accept every word of L = { a^n b^n | n > 0 }.
> (run-machine (make-machine '(#(0 "a" 1 "A" "R")
                               #(1 "a" 1 "a" "R")
                               #(1 "B" 1 "B" "R")
                               #(1 "b" 2 "B" "L")
                               #(2 "a" 2 "a" "L")
                               #(2 "B" 2 "B" "L")
                               #(2 "A" 0 "A" "R")
                               #(0 "B" 3 "B" "R")
                               #(3 "B" 3 "B" "R")
                               #(3 "#" 4 "#" "R")) 0 '(4))
             (make-tape '("a" "a" "b" "b")))
"Calcul acceptant"
"AABB"
```

Code source

```
;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;; Tape functions.

;; Return an infinite one-dimensional tape with 'contents' placed at
;; its origin.
(defun make-tape (contents)
  (make-array (length contents) :initial-contents contents
              :adjustable t :fill-pointer t))

;; Return the symbol of 'tape' at 'index'. If 'index' is outside the
;; tape bounds, the machine special blank symbol is returned.
(defun get-symbol (machine tape index)
  (if (< index (length tape))
      (elt tape index)
    (machine-blank machine)))

;; Set 'x' to be the symbol of 'tape' at 'index', and return
;; 'tape'. 'index' should have a value not greater than the tape
;; length.
(defun set-symbol (tape index x)
  (progn
    (if (< index (length tape))
        (setf (elt tape index) x)
      (vector-push-extend x tape))
    tape))

;; Move 'index' according to 'switch' direction (either "L" or "R"),
;; and return the new index.
(defun move-head (index switch)
  (if (equal "L" switch)
      (if (= 0 index)
          "Error - can't go left of origin."
          (- index 1))
    (+ index 1)))

;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
;; Rules functions.

;; Return a Turing machine rule with given 'from state', 'from
;; symbol', 'next state', 'next symbol' and direction 'switch'.
(defun make-rule (from-state from-symbol next-state next-symbol switch)
  (vector state symbol next-state next-symbol switch))

;; Return the 'rule' 'from state'.
(defun rule-from-state (rule)
  (elt rule 0))

;; Return the 'rule' 'from symbol'.
(defun rule-from-symbol (rule)
  (elt rule 1))

;; Return the 'rule' 'next state'.
(defun rule-next-state (rule)
  (elt rule 2))

;; Return the 'rule' 'next symbol'.
(defun rule-next-symbol (rule)
  (elt rule 3))

;; Return the 'rule' direction switch ("L" or "R").
(defun rule-switch (rule)
  (elt rule 4))

;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;; Machine functions.

;; Return a Turing machine with the given rules, initial state,
;; final states and 'blank' symbol (default to "#" if absent).
(defun make-machine (rules initial-state &optional (final-states '()) (blank "#"))
  (vector rules initial-state final-states blank))

;; Return a list of the 'machine' rules.
(defun machine-rules (machine)
  (elt machine 0))

;; Return the 'machine' initial state.
(defun machine-initial-state (machine)
  (elt machine 1))

;; Return a list of the 'machine' final states.
(defun machine-final-states (machine)
  (elt machine 2))

;; Return the special blank symbol of 'machine'.
(defun machine-blank (machine)
  (elt machine 3))

;; Return a rule of the given Turing machine applicable to the current
;; state and symbol.
(defun find-rule (machine state symbol)
```

```
  (find (list state symbol) (machine-rules machine)
        :test 'equal
        :key (lambda (rule) (list (rule-from-state rule) (rule-from-symbol rule)))))

;; Whether 'state' is a final state of 'machine'.
(defun is-final-state? (machine state)
  (not (null (find state (machine-final-states machine)
                   :test 'equal))))

;; The simulator function. Takes a machine and a tape as arguments.
(defun run-machine (machine tape
                            &optional (state (machine-initial-state machine))
                            (tape-head 0))
  (let* ((symbol (get-symbol machine tape tape-head))
         (rule (find-rule machine state symbol)))
    (when (is-final-state? machine state)
      (print "Calcul acceptant"))
    (if (null rule) (pretty-print tape (machine-blank machine))
      (run-machine machine
                   (set-symbol tape tape-head (rule-next-symbol rule))
                   (rule-next-state rule)
                   (move-head tape-head (rule-switch rule))))))

;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;; Main course !

;; Transform a list tape into a string, with the special blank symbol
;; (default to '#') trimmed.
(defun pretty-print (tape blank)
  (string-trim blank
               (reduce (lambda (a b)
                         (concatenate 'string
                                      (format nil "~a" a)
                                      (format nil "~a" b)))
                       tape)))
```