Examen Haskell – IR3

Stéphane Vialette

28 janvier 2016

Lire le sujet en entier avant de commencer, il contient de nombreux exemples et une partie importante du travail demandé consiste en la compréhension du code fourni. Les fonctions demandées sont courtes (la question 9 demande un tout petit peu plus de code). Les questions 10 et 11 sont des questions bonus (et elles ne sont pas très difficiles !). Testez vos fonctions au fur et à mesure de l'écriture. Dernier point important, rappelez vous qu'en haskell les structures de données ne sont pas mutables. Ce qui veut dire que vous ne pouvez pas les modifier, vous devez en créer de nouvelles.

1 Introduction

Nous allons formaliser une notion d'ordinateur très simple, une *machine à registres non limités*[1] (URM pour *Unlimited Register Machine*). Il s'agit d'un modèle simplifié et abstrait du fonctionnement des appareils mécaniques de calcul. Ces machines ont une mémoire constituée d'un nombre fini de registres $R_0, R_1, \ldots, R_k$, chaque registre pouvant contenir un entier naturel arbitraire. Une URM dispose également d'un *programme* qui est une suite finie constituée de 4 types d'*instruction*, et d'un *pointeur d'instruction* qui indique la prochaine instruction du programme à exécuter. Une URM s'arrête uniquement si le pointeur d'instruction ne pointe pas vers une instruction du programme. Les instructions sont les suivantes :

1. `zero`(R_i) : Mettre le contenu du registre R_i à 0, et incrémenter ensuite le pointeur d'instruction.
2. `successor`(R_i) : Incrémenter le contenu du registre R_i, et incrémenter ensuite le pointeur d'instruction.
3. `copy`(R_i, R_j) : Copier la valeur du registre R_i dans le registre R_j (R_i conserve sa valeur), et incrémenter ensuite le pointeur d'instruction.
4. `jump`(R_i, R_j, l) : Si les valeurs des registres R_i et R_j sont égales alors placer le pointeur d'instruction sur la ligne l, sinon incrémenter le pointeur d'instruction.

Vous remarquerez que dans tous les cas (sauf éventuellement pour `jump`(R_i, R_j, l)), la prochaine instruction à exécuter est donc la suivante. On considère dans la suite que la numérotation des instructions est implicite et débute à 0. De plus, un registre R_i est simplement désigné par son index i. Un exemple devrait clarifier les choses. Considérons la machine à 4 registres (R_0, R_1, R_2, et R_3) qui calcule (dans R_3) l'addition de 2 registres (R_1 et R_2), le registre R_0 est utilisé pour effectuer des calculs intermédiaires :

```
 0 : zero(3)
 1 : zero(0)
 2 : jump(0,1,6)
 3 : successor(0)
 4 : successor(3)
 5 : jump(0,0,2)
 6 : zero(0)
 7 : jump(0,2,11)
 8 : successor(0)
 9 : successor(3)
10 : jump(0,0,7)
```

1. Consultez `http://planetmath.org/unlimitedregistermachine` ou `https://proofwiki.org/wiki/Definition:Unlimited_Register_Machine` pour plus de détails.

Suivez pas à pas ce programme et remarquez qu'une instruction `jump(i, i, l)` place toujours le pointeur d'instruction sur la ligne l (nous comparons en effet un registre avec lui même dans ce cas !).

2 Notre but

Notre but est d'écrire un ensemble de modules Haskell pour construire et simuler des URM. Voici le code que nous pourrons par exemple utiliser pour créer une URM à 4 registres et exécuter le programme précédant (la fonction `testTrace` est relative à la dernière question de l'examen) :

```
module Test
(
  test
-- uncomment the following line to export testTrace function
--, testTrace
)
where

  import qualified Instruction as I
  import qualified Register    as R
  import qualified Program     as P
  import qualified URM

  -- program
  p :: P.Program
  p =  [  I.Zero      3      -- initialisation
        , I.Zero      0      -- add R1 to R3 (use R0 to count)
        , I.Jump      0 1 6  -- loop1: enter if R0 and R1 are not equal
        , I.Successor 0      -- increment R0
        , I.Successor 3      -- increment R3
        , I.Jump      0 0 2  -- end loop1
        , I.Zero      0      -- add R2 to R3 (use R0 to count)
        , I.Jump      0 2 11 -- loop2: enter if R0 and R2 are not equal
        , I.Successor 0      -- increment R0
        , I.Successor 3      -- increment R3
        , I.Jump      0 0 7  -- end loop2
        ]

  -- registers
  rs :: [R.Value]
  rs = [0,3,2,0]

  -- URM
  urm :: URM.URM
  urm  = URM.makeURM p rs

  test :: IO ()
  test = print (URM.run urm)

  testTrace :: IO()
  testTrace = print (URM.runTrace urm)
```

Un simple test depuis l'interpréteur haskell pour vérifier $3 + 2 = 5$ (ici et dans la suite `lambda>` est le prompt de l'interpréteur) :

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
lambda> :load Test
[1 of 5] Compiling Register         ( Register.hs, interpreted )
[2 of 5] Compiling Instruction      ( Instruction.hs, interpreted )
[3 of 5] Compiling Program          ( Program.hs, interpreted )
[4 of 5] Compiling URM              ( URM.hs, interpreted )
[5 of 5] Compiling Test             ( Test.hs, interpreted )
Ok, modules loaded: Instruction, Test, Register, Program, URM.
lambda> test
[2,3,2,5]
lambda>
```

La function `run` retourne les valeurs des registres après exécution du programme (dans cet exemple, le résultat est dans le dernier registre). Vous remarquez également l'utilisation de 4 modules haskell (**Instruction**, **Register**, **Program** et **URM**), modules que nous allons développer.

3 Registres

Le module **Register** (incomplet ici) pour manipuler un ensemble de registres est donné ci-après :

```haskell
module Register
(
-- types
  Value
, Index

-- read registers
, readRegister

-- alter registers
, zero
, successor
, copy
)
where

  -- Register value type definition
  type Value = Int

  -- Register index type definition
  type Index = Int

  -- Return the value stored in register 'i'
  readRegister :: [Value] -> Index -> Value
  readRegister = (!!)
  -- Note from your teacher. Note that the definition of the above function
  -- is strictly equivalent to "readRegister rs i = rs !! i" as !! is an infix
  -- function and hence (!!) is a prefix function. Therefore, via eta-reduction,
  -- one obtains:
  -- readRegister rs i = rs !! i == readRegister rs i = (!!) rs i
  --                             == readRegister = (!!)

  -- Alter a register according to an altering function
  alterRegister :: (Value -> Value) -> [Value] -> Index -> [Value]
  alterRegister f = go
    where
      go :: [Value] -> Index -> [Value]
      go (x:xs) i
        | i == 0    = f x : xs
        | otherwise = x   : go xs (i-1)
```

```haskell
  -- Store 0 in a specified register.
  -- zero :: [Value] -> Index -> [Value]
  -- To be implemented...

  -- Increment the value stored in a specified register.
  -- successor :: [Value] -> Index -> [Value]
  -- To be implemented...

  -- Copy the value of one register to another register.
  -- copy :: [Value] -> Index -> Index -> [Value]
  -- To be implemented...
```

Question 1. Écrire la fonction `zero :: [Value] -> Index -> [Value]` qui place 0 dans un registre donné. Vous pouvez ou non utiliser la fonction `alterRegister` pour écrire cette fonction. Par exemple :

```
lambda> R.zero [1,2,3] 0
[0,2,3]
lambda> R.zero [1,2,3] 1
[1,0,3]
lambda> R.zero [1,2,3] 2
[1,2,0]
```

Question 2. Écrire la fonction `successor :: [Value] -> Index -> [Value]` qui incrémente la valeur d'un registre donné. Il est impératif d'utiliser la fonction `alterRegister`. Par exemple :

```
lambda> R.successor [1,2,3] 0
[2,2,3]
lambda> R.successor [1,2,3] 1
[1,3,3]
lambda> R.successor [1,2,3] 2
[1,2,4]
```

Question 3. Écrire la fonction `copy :: [Value] -> Index -> Index -> [Value]` qui copie la valeur d'un registre dans un autre. Vous pouvez ou non utiliser la fonction `alterRegister` pour écrire cette fonction. Par exemple :

```
lambda> R.copy [1,2,3] 0 1
[1,1,3]
lambda> R.copy [1,2,3] 1 2
[1,2,2]
lambda> R.copy [1,2,3] 2 0
[3,2,3]
lambda> R.copy [1,2,3] 0 0
[1,2,3]
```

4 Instructions

Le module **Instruction** est donné complet. Il ne contient que la définition du type **Instruction** pour représenter une instruction d'un programme destiné à une URM.

```haskell
module Instruction
(
  -- type
  Instruction(..)
)
where
```

```
  import qualified Register as R

  data Instruction =
    -- Zero i : Replace the number in Ri by 0
     Zero R.Index
    -- Successor i : Add 1 to the number in Ri
    | Successor R.Index
    -- Copy i j : Replace the number in Rj by the number in Ri
    -- (leaving the one in Ri as it was)
    | Copy R.Index R.Index
    -- Jump i j k : If the numbers in Ri and Rj are equal, go
    -- to instruction number k, otherwise go to the next instruction
    | Jump R.Index R.Index Int
    deriving (Show)
```

5 Programme

Le module **Program** est également donné complet. Il ne contient en fait que la définition du type **Program** pour représenter un programme destiné à une URM.

```
module Program
(
-- type
  Program
)
where

  import qualified Instruction   as I

  -- A program is a list of instructions
  type Program = [I.Instruction]
```

6 URM

Considérons maintenant le module (incomplet cette fois ci) **URM** suivant :

```
module URM
(
-- type
  URM()
, Trace
-- URM construction
, makeURM

-- run URM
, run
, runTrace
)
where

  import qualified Data.List   as L
  import qualified Instruction as I
  import qualified Register    as R
  import qualified Program     as P

  -- URM type definition
```

```haskell
data URM = URM { program        :: P.Program -- list of instructions
               , instructionPtr :: Int       -- current instruction index
               , registers      :: [R.Value] -- list of registers
               } deriving (Show)

-- trace type definition
type Trace = (I.Instruction, [R.Value])

-- Construct an URM from program and value registers.
makeURM :: P.Program -> [R.Value] -> URM
makeURM p rs = URM { program = p, instructionPtr = 0, registers = rs }

-- execute a Zero instruction
-- executeZero :: URM -> R.Index -> URM
-- To be implemented...

-- Execute a Succesor instruction
-- executeSuccessor :: URM -> R.Index -> URM
-- To be implemented...

-- Execute a Copy instruction
-- executeCopy :: URM -> R.Index -> R.Index -> URM
-- To be implemented...

-- Execute a Jump instruction
-- executeJump :: URM -> R.Index -> R.Index -> Int -> URM
-- To be implemented...

-- Return the number of instructions in the program.
numberOfInstructions :: URM -> Int
numberOfInstructions urm = L.length is
  where
    is = program urm

-- Return True iff the instruction pointer is valid (i.e. the instruction
-- pointer is heading an instruction in the program).
-- validInstructionPtr :: URM -> Bool
-- To be implemented...

-- Return True iff the instruction pointer is not valid (i.e. the instruction
-- pointer is not heading an instruction in the program).
invalidInstructionPtr :: URM -> Bool
invalidInstructionPtr = not . validInstructionPtr

-- Alias for invalidInstructionPtr
quit :: URM -> Bool
quit = invalidInstructionPtr

-- Return the current instruction.
-- Warning: instructionPtr urm may raise an exception.
currentInstruction :: URM -> I.Instruction
currentInstruction urm = program urm !! instructionPtr urm

-- Run the URM and return registers at termination (if any).
run :: URM -> [R.Value]
run urm
```

```
    | quit urm  = registers urm
    | otherwise = execute (currentInstruction urm)
    where
      execute :: I.Instruction -> [R.Value]
      execute (I.Zero i)      = run $ executeZero      urm i
      execute (I.Successor i) = run $ executeSuccessor urm i
      execute (I.Copy i j)    = run $ executeCopy      urm i j
      execute (I.Jump i j k)  = run $ executeJump      urm i j k

-- Run the URM in trace mode.
-- runTrace :: URM -> [Trace]
-- To be implemented...
```

Question 4. Écrire la fonction `validInstructionPtr :: URM -> Bool` qui retourne vrai si et seulement si le pointeur d'intruction est positionné sur une instruction du programme (c'est-à-dire que le pointeur d'instruction est un entier positif qui n'excède pas le nombre d'instructions du programme moins un (puisque l'on compte à partir de 0)).

Question 5. Écrire la fonction `executeZero :: URM -> R.Index -> URM` (appelée depuis la fonction `run :: URM -> [R.Value]`). Cette fonction est appelée lorsque l'URM exécute une instruction de type `Zero`.

Question 6. Écrire la fonction `executeSuccessor :: URM -> R.Index -> URM` (appelée depuis la fonction `run :: URM -> [R.Value]`). Cette fonction est appelée lorsque l'URM exécute une instruction de type `Successor`.

Question 7. Écrire la fonction `executeCopy :: URM -> R.Index -> R.Index -> URM` (appelée depuis la fonction `run :: URM -> [R.Value]`). Cette fonction est appelée lorsque l'URM exécute une instruction de type `Copy`.

Question 8. Écrire la fonction `executeJump :: URM -> R.Index -> R.Index -> Int -> URM` (appelée depuis la fonction `run :: URM -> [R.Value]`). Cette fonction est appelée lorsque l'URM exécute une instruction de type `Jump`.

Question 9. Écrire la fonction `runTrace :: URM -> [Trace]` qui exécute un programme en mode *trace*. Dans le mode *trace*, la valeur retournée (de type `[Trace]`) donne une vision détaillée de l'exécution du programme. Plus précisément, la fonction `runTrace` retourne une liste dont l'élément `i` est la paire `(inst, rs)` où `inst` est l'instruction exécutée à l'itération `i` du programme et `rs` est la valeur de tous les registres après exécution de cette instruction. Par exemple, en considérant la fonction `testTrace` du module `Test` :

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
lambda> :load Test
[1 of 5] Compiling Register         ( Register.hs, interpreted )
[2 of 5] Compiling Instruction      ( Instruction.hs, interpreted )
[3 of 5] Compiling Program          ( Program.hs, interpreted )
[4 of 5] Compiling URM              ( URM.hs, interpreted )
[5 of 5] Compiling Test             ( Test.hs, interpreted )
Ok, modules loaded: Instruction, Test, Register, Program, URM.
lambda> testTrace
[(Zero 3,[0,3,2,0]),      (Zero 0,[0,3,2,0]),      (Jump 0 1 6,[0,3,2,0]),
(Successor 0,[1,3,2,0]), (Successor 3,[1,3,2,1]), (Jump 0 0 2,[1,3,2,1]),
(Jump 0 1 6,[1,3,2,1]),  (Successor 0,[2,3,2,1]), (Successor 3,[2,3,2,2]),
(Jump 0 0 2,[2,3,2,2]),  (Jump 0 1 6,[2,3,2,2]),  (Successor 0,[3,3,2,2]),
(Successor 3,[3,3,2,3]), (Jump 0 0 2,[3,3,2,3]),  (Jump 0 1 6,[3,3,2,3]),
(Zero 0,[0,3,2,3]),       (Jump 0 2 11,[0,3,2,3]), (Successor 0,[1,3,2,3]),
(Successor 3,[1,3,2,4]), (Jump 0 0 7,[1,3,2,4]),  (Jump 0 2 11,[1,3,2,4]),
(Successor 0,[2,3,2,4]), (Successor 3,[2,3,2,5]), (Jump 0 0 7,[2,3,2,5]),
(Jump 0 2 11,[2,3,2,5])]
lambda>
```

Question 10. (Bonus) À la manière du module **Test**, écrire un module **TestMin** exportant une unique fonction `testMin :: [R.Value] -> IO ()` simulant un programme URM spécifié de la façon suivante : $R_0 = \min\{R_1, R_2, R_3\}$. Un exemple d'utilisation :

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
lambda> :load TestMin
[1 of 5] Compiling Register         ( Register.hs, interpreted )
[2 of 5] Compiling Instruction      ( Instruction.hs, interpreted )
[3 of 5] Compiling Program          ( Program.hs, interpreted )
[4 of 5] Compiling URM              ( URM.hs, interpreted )
[5 of 5] Compiling TestMin      ( TestMin.hs, interpreted )
Ok, modules loaded: Instruction, TestMin, Register, Program, URM.
lambda> testMin [0,2,4,6]
[2,2,4,6]
lambda> testMin [0,4,2,6]
[2,4,2,6]
lambda> testMin [0,6,4,2]
[2,6,4,2]
lambda> testMin [0,3,3,3]
[3,3,3,3]
lambda>
```

Question 11. (Bonus) Il s'agit d'utiliser le module **URM** pour calculer le minimum des éléments d'une liste d'entiers. Pour ce faire, écrire un module **TestMultiMin** exportant une unique fonction `testMultiMin :: [Int] -> IO ()` simulant un programme URM spécifié de la façon suivante : $R_0 = \min\{R_1, R_2, \ldots R_n\}$, où n est la longueur de la liste en entrée et les R_i sont les valeurs des éléments de la liste en entrée. Un exemple d'utilisation :

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
lambda> :l TestMultiMin.hs
[1 of 5] Compiling Register         ( Register.hs, interpreted )
[2 of 5] Compiling Instruction      ( Instruction.hs, interpreted )
[3 of 5] Compiling Program          ( Program.hs, interpreted )
[4 of 5] Compiling URM              ( URM.hs, interpreted )
[5 of 5] Compiling TestMultiMin     ( TestMultiMin.hs, interpreted )
Ok, modules loaded: Instruction, TestMultiMin, Register, Program, URM.
lambda> testMultiMin [4]
[4,4]
lambda> testMultiMin [4,2]
[2,4,2]
lambda> testMultiMin [4,1,2]
[1,4,1,2]
lambda> testMultiMin $ L.reverse [1..10]
[1,10,9,8,7,6,5,4,3,2,1]
lambda>
```