Semântica das Linguagens de Programação
(3º ano da Licenciatura em Ciências da Computação)

Criação de programas em Haskell para Simulação de Semânticas

Relatório de Desenvolvimento

Filipe Freitas (A85026)

23 de Julho de 2020

Conteúdo

1 Introdução **2**
- 1.1 Contextualização . . . 2
- 1.2 Estrutura do Relatório . . . 2
- 1.3 Estrutura do Código . . . 3

2 Estruturas da Linguagem While **4**
- 2.1 Estados . . . 4
- 2.2 Expressões Aritméticas . . . 6
- 2.3 Expressões Booleanas . . . 6
- 2.4 Comandos da Linguagem While . . . 7

3 Semântica Natural **9**
- 3.1 Expressões Aritméticas . . . 9
- 3.2 Expressões Booleanas . . . 10
- 3.3 Comandos da Linguagem While . . . 10

4 Semântica Operacional Estrutural (*Small-Step*) **13**
- 4.1 Expressões Aritméticas . . . 13
- 4.2 Expressões Booleanas . . . 14
- 4.3 Comandos da Linguagem While . . . 17

5 Máquinas Abstratas **20**
- 5.1 Máquina AM . . . 20
  - 5.1.1 Instruções e Estrutura . . . 20
  - 5.1.2 Semântica . . . 21
  - 5.1.3 Compilador de programas *While* . . . 24
  - 5.1.4 Função Semântica S_{am} . . . 26
- 5.2 Máquina AM1 . . . 27
  - 5.2.1 Memória . . . 27
  - 5.2.2 Instruções e Estrutura . . . 27
  - 5.2.3 Semântica . . . 28
  - 5.2.4 Compilador . . . 28
  - 5.2.5 Função Semântica S_{am1} . . . 29
- 5.3 Máquina AM2 . . . 30
  - 5.3.1 Instruções e Estrutura . . . 30
  - 5.3.2 Semântica . . . 31
  - 5.3.3 Compilador . . . 31
  - 5.3.4 Função semântica S_{am2} . . . 32

6 Conclusão **33**

A Parser para a linguagem While **34**

B Como correr os vários programas no interpretador GHCi **35**

C Como compilar os vários programas com recurso à Makefile **36**

Capítulo 1

Introdução

1.1 Contextualização

Na sequência da Unidade Curricular de Semântica das Linguagens de Programação, foi proposta a realização de um trabalho que consiste na implementação, em Haskell, de vários simuladores para a linguagem *While* que foi estudada nas aulas, de acordo com as semânticas naturais e operacional estruturais que foram abordadas nas aulas, bem como a implementação de simuladores para as Máquinas Abstratas AM também estudadas.

Sendo assim, este relatório pretende abordar as implementações criadas, explicar as decisões tomadas ao longo da implementação, e dar exemplos de programas, da sua simulação, e do resultado da sua execução.

1.2 Estrutura do Relatório

Este relatório está dividido em 6 partes principais, e contém também 3 anexos.

A *primeira*, correspondente a esta introdução, pretende contextualizar o trabalho e explicar a sua estrutura, e explica também a estrutura do código do trabalho.

Na *segunda parte* é feita uma introdução à linguagem *While*, e são apresentadas as estruturas de dados, correspondentes à mesma, que irão servir de apoio a todo o resto do trabalho.

Nas *terceira* e *quarta* partes são apresentadas implementações de semânticas naturais e operacional estruturais para a linguagem *While*.

Na *quinta parte* são apresentadas as máquinas abstratas AM, as suas instruções, estruturas, e correspondentes semânticas.

Na *última parte* é feita uma conclusão do trabalho realizado.

O Apêndice A contém uma explicação de um *Parser* para a linguagem *While*, que permite a tradução de texto de entrada nas estruturas de dados internas que servem de base à linguagem *While*.

Os Apêndices *B* e *C* descrevem dois modos diferentes de execução dos vários programas: recorrendo ao interpretador *GHCi* ou compilando em executáveis, respetivamente.

1.3 Estrutura do Código

O código tende a estar dividido em vários ficheiros, cada um correspondente a uma área da implementação que foi pedida; também tende a separar as estruturas de dados em ficheiros individuais, para aumentar a legibilidade; e também separa cada uma das implementações de semântica e de máquina abstrata pedidas nos seus ficheiros individuais.

Relativamente às estruturas de dados, a maior parte são implementadas recorrendo a tipos *data* ou *newtype*. Esta foi uma decisão consciente de implementação, pois assim permite a declaração de instâncias *Show* e *Read* dos mesmos, o que, em conjunção com o *Parser* que foi criado para a linguagem, torna o processo de visualização da saída do programa bastante mais simples, mesmo no *GHCi*. Isto tem a desvantagem de diminuir a legibilidade do código; no entanto, foram tomadas precauções para minimizar o impacto desta decisão na leitura do mesmo.

As implementações de instâncias *Show* são mantidas nos mesmos locais onde os respetivos tipos são declarados. No entanto, as instâncias de *Read* são declaradas no ficheiro *Parser.hs*, o que significa que só estão declaradas se este ficheiro tiver sido importado para a *scope* atual. Mais informações sobre este detalhe são dadas no Apêndice B.

Para correr cada secção do programa, existe sempre um ficheiro principal que, ao ser importado, permite uma execução correta e completa de todas as funcionalidades do respetivo programa. A indicação do ficheiro principal é sempre indicada em cada secção do relatório, sempre que pertinente.

Em cada secção deste relatório, sempre que pertinente, é dada uma indicação de onde se pode encontrar o código ao qual o relatório está a fazer referência.

Capítulo 2

Estruturas da Linguagem While

Neste capítulo vão ser apresentadas as várias estruturas de dados que servem de suporte interno à linguagem *While*.

2.1 Estados

Esta secção corresponde à resolução do exercício 8. b) ii) da ficha 1.

Antes de iniciar as estruturas de dados da linguagem *While* propriamente ditas, vamos criar uma estrutura de dados para guardar os estados que vão ser necessários durante a interpretação de comandos da linguagem.

Sendo assim, e tendo em conta que, nas aulas, estados foram definidos como sendo uma função que, a cada variável, associa o seu valor inteiro correspondente, uma representação natural em Haskell desta definição é uma lista de pares, onde a primeira componente de cada par é o nome da variável, e a segunda componente é o valor da variável no estado atual.

Temos, assim, a seguinte definição:

Ficheiro *State.hs*

```haskell
type Var = String
type Val = Int

-- Um VarState é apenas a descrição de uma variável (identificada pelo seu nome,
 ↪  uma String) e do seu valor inteiro
type VarState = (Var, Val)

-- Um Estado é apenas uma lista de variáveis, com os seus respetivos valores
 ↪  definidos
newtype State = State [VarState] deriving Eq
```

Pela razão já explicada *anteriormente*, o tipo *State*, como é visualizável na saída dos programas, é implementado com o seu próprio tipo e com o seu próprio construtor.

É depois definida uma instância de *Show* para o tipo *State*, de acordo com a notação utilizada nas aulas:

Ficheiro *State.hs*

```haskell
instance Show State where
    show (State [])              = ""
    show (State ((var, val):xs)) = "[" ++ var ++ " -> " ++ (show val) ++ "]" ++
     ↪  (show (State xs))
```

A seguir, defini uma função para atualizar o valor de uma variável no estado, retornando um novo estado com a variável atualizada.

Ficheiro *State.hs*

```haskell
update :: VarState -> State -> State
update (x, v) (State []) = State [(x, v)]
update (x, v) (State ((var, val):s))
        | var == x  = State $ (var, v) : s
        | otherwise = let (State s') = update (x, v) (State s)
                      in  (State $ (var, val):s')
```

Defini também uma função para obter o valor de uma variável num dado estado, para simplificar algum do código que se segue.

Ficheiro *State.hs*

```haskell
fetch :: Var -> State -> Val
fetch v (State []) = 0
fetch v (State ((var, val):s)) | var == v  = val
                               | otherwise = fetch v (State s)
```

2.2 Expressões Aritméticas

Esta secção corresponde à primeira parte da resolução do exercício 8. a) i) da ficha 1.

Relativamente às expressões aritméticas, nas aulas definimos que estas podem ser uma das seguintes:

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 \times a_2 \mid a_1 - a_2$$

Como uma extensão natural, decidi acrescentar também simétricos. Temos, assim, as seguintes opções para expressões aritméticas:

$$a ::= n \mid x \mid -a \mid a_1 + a_2 \mid a_1 \times a_2 \mid a_1 - a_2$$

Portanto, defini o seguinte tipo:

Ficheiro *Aexp.hs*

```
7  data Aexp = Num  Int          -- Inteiro
8            | Var  Var          -- Variável
9            | Sim  Aexp         -- Simétrico
10           | Add  Aexp Aexp    -- Soma
11           | Mult Aexp Aexp    -- Multiplicação
12           | Sub  Aexp Aexp    -- Subtração
13           deriving Eq
```

De seguida, foi também definida uma instância de *Show* para a visualização correta das expressões aritméticas na saída da consola. Essa instância pode ser visualizada no ficheiro *Aexp.hs*, linhas 18-53. A sua aparente complexidade é apenas para garantir uma saída "bonita".

2.3 Expressões Booleanas

Esta secção corresponde à segunda parte da resolução do exercício 8. a) i) da ficha 1.

No caso das expressões booleanas, nas aulas definimos que podem ser uma das seguintes:

$$b_0 ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$$

Por extensão natural, defini também as seguintes possibilidades para expressões booleanas:

$$b ::= b_0 \mid a_1 > a_2 \mid a_1 \geq a_2 \mid a_1 < a_2 \mid b_1 \vee b_2$$

Assim, a definição completa de expressões booleanas é:

$$b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 > a_2 \mid a_1 \geq a_2 \mid a_1 < a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2$$

A estrutura em Haskell correspondente a esta estrutura é:

Ficheiro *Bexp.hs*

```haskell
data Bexp = True                -- True
          | False               -- False
          | Equals  Aexp Aexp   -- Igualdade
          | GT      Aexp Aexp   -- Maior
          | GE      Aexp Aexp   -- Maior ou Igual
          | LT      Aexp Aexp   -- Menor
          | LE      Aexp Aexp   -- Menor ou Igual
          | Not     Bexp        -- Negação
          | And     Bexp Bexp   -- Conjunção
          | Or      Bexp Bexp   -- Disjunção
          deriving Eq
```

Tal como acontece nas expressões aritméticas, é também depois definida uma instância de *Show* para o tipo *Bexp*.

2.4 Comandos da Linguagem While

Esta secção corresponde à resolução do exercício 8. b) i) da ficha 1.

Nas aulas definimos os comandos da linguagem *While* como podendo ser um dos seguintes:

$$S ::= \text{skip} \mid x := a \mid S_1; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S$$

Como extensão, eu acrescentei também um comando *if-then*, sem *else*.

Assim, a definição completa dos comandos da linguagem *While* é:

$$S ::= \text{skip} \mid x := a \mid S_1; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{if } b \text{ then } S \mid \text{while } b \text{ do } S$$

Assim, podemos definir a seguinte estrutura para os comandos da linguagem *While*:

Ficheiro *Stm.hs*

```haskell
data Stm = Skip                     -- Skip
         | Assign   Var  Aexp       -- Atribuição
         | Comp     Stm  Stm        -- Composição
         | ITE      Bexp Stm  Stm   -- If Then Else
         | If       Bexp Stm        -- If Then
         | While    Bexp Stm        -- While do
         deriving Eq
```

Tal como nas restantes estruturas, existe também uma instância de *Show* definida para a estrutura *Stm*.

Capítulo 3

Semântica Natural

Neste capítulo vai ser presente uma implementação de uma semântica natural para as expressões aritméticas, booleanas e comandos da linguagem *While*.

3.1 Expressões Aritméticas

Esta secção corresponde à primeira parte da resolução do exerício 8. a) iii) da ficha 1.

A função semântica $\mathcal{A}$ é uma função que aceita expressões aritméticas, um estado, e devolve o resultado de avaliar a expressão aritmética de entrada no estado especificado. Assim, pode ser definida em Haskell do seguinte modo:

Ficheiro *NS.hs*

```
15  evalA :: Aexp -> State -> Val
16
17  evalA (Num  n    ) _   = n
18  evalA (Var  v    ) s   = fetch v s
19  evalA (Sim  x    ) s   = -(evalA x s)
20
21  evalA (Add  e1 e2) s   = (evalA e1 s) + (evalA e2 s)
22  evalA (Mult e1 e2) s   = (evalA e1 s) * (evalA e2 s)
23  evalA (Sub  e1 e2) s   = (evalA e1 s) - (evalA e2 s)
```

Esta implementação é bastante simples: qualquer expressão que seja só um número retorna só esse número; se for uma variável, é só consultar o seu valor no estado; se a expressão for o simétrico de outra, então a função retorna o resultado simétrico do valor da expressão constituinte; somas, multiplicações e subtrações também são só o resultado da respetiva soma, multiplicação ou subtração das duas expressões constituintes da expressão.

3.2 Expressões Booleanas

Esta secção corresponde à segunda parte da resolução do exercício 8. a) iii) da ficha 1.

A função semântica $\mathcal{B}$ é uma função parecida à função semântica $\mathcal{A}$. De um modo geral, podemos dizer que a função semântica $\mathcal{B}$ atua sobre expressões booleanas, em vez de expressões aritméticas. Assim, pode ser definida do seguinte modo:

Ficheiro *NS.hs*

```
27  evalB :: Bexp -> State -> Bool
28
29  evalB Bexp.True      _ = Prelude.True
30  evalB Bexp.False     _ = Prelude.False
31
32  evalB (Equals a b) s  =  (evalA a s) == (evalA b s)
33  evalB (GT     a b) s  =  (evalA a s) >  (evalA b s)
34  evalB (GE     a b) s  =  (evalA a s) >= (evalA b s)
35  evalB (LT     a b) s  =  (evalA a s) <  (evalA b s)
36  evalB (LE     a b) s  =  (evalA a s) <= (evalA b s)
37
38  evalB (Not    a  ) s  =  not $ evalB a s
39
40  evalB (And    a b) s  =  (evalB a s) && (evalB b s)
```

3.3 Comandos da Linguagem While

Esta secção corresponde à resolução dos exercícios 8. b) iii) e 8. b) iv) da ficha 1.

É agora pedida uma função que faça a avaliação de expressões da linguagem *While*. Defini essa função de acordo com as regras da semântica natural que foram apresentadas nas aulas.

Ficheiro *NS.hs*

```haskell
evalNS :: (Stm, State) -> State

evalNS (Skip, s)              = s
evalNS (Assign var exp, s)    = update (var, evalA exp s) s

evalNS (Comp c1 c2,     s)    = let s' = evalNS (c1, s)
                                in evalNS (c2, s')

evalNS (ITE b c1 c2,    s)    | cond      = evalNS (c1, s)
                              | otherwise = evalNS (c2, s)
                              where cond  = evalB b s
evalNS (If b c1,        s)    = evalNS (ITE b c1 Skip, s)

evalNS (While b c,      s)    | cond      = evalNS ((While b c), s')
                              | otherwise = s
                              where cond  = evalB b s
                                    s'    = evalNS (c, s)
```

Devido ao *Parser* que criei, e às instâncias de *Read* e *Show* criadas, testar esta função é bastante fácil. Para tal, basta, no *GHCi*, importar o ficheiro *NS.hs*, e executar a função main. Esta espera na sua entrada um programa *While* e um estado inicial, imprimindo depois o estado final, após execução do programa *While*.

Apresenta-se a seguir a saída esperada de dois programas *While*:

$ ghci NS.hs

```
*NS> main
------------------------------------------------------------
> if x > 0 then y := x else if x < 0 then y := -x else z := 1 [x -> -1]
> if x > 0 then {y := x} else {if x < 0 then {y := -x} else {z := 1}}
  [x -> -1]
> [x -> -1][y -> 1]
------------------------------------------------------------

------------------------------------------------------------
> x := 1; while b > 0 do { x := x * a; b := b - 1 } [a -> 3][b -> 2]
> x := 1; while b > 0 do {x := x * a; b := b - 1}
  [a -> 3][b -> 2]
> [a -> 3][b -> 0][x -> 9]
------------------------------------------------------------
```

Nota: após introduzir um programa *While*, a função main avalia a entrada dada, e se esta for válida, apresenta na saída o mesmo programa, sem alterações, após este ter sido convertido para as estruturas internas da linguagem *While*.

Nota 2: Para efeitos de legibilidade, o estado é sempre impresso numa linha separada das instruções.

Em alternativa à função main, é possível executar a função evalNS introduzindo manualmente os parâmetros do comando *While* e do estado. Basta, para isso, seguir o exemplo que se apresenta a seguir:

$ ghci NS.hs

```
*NS> let c = read "if x > 0 then y := x else if x < 0 then y := -x else z := 1" :: Stm
*NS> let s = read "[x -> 1]" :: State
*NS> evalNS (c, s)
[x -> 1][y -> 1]
```

Este processo encontra-se descrito no Apêndice B.

Capítulo 4

Semântica Operacional Estrutural (*Small-Step*)

Neste capítulo vai ser apresentada uma implementação de uma semântica *small-step* para as expressões aritméticas, booleanas e comandos da linguagem *While*.

4.1 Expressões Aritméticas

Esta secção corresponde à resolução do exercício 5. a) da ficha 2.

Uma semântica de transições *small-step* para as expressões aritméticas pode ser definida do seguinte modo:

Possível implementação de stepA

```
1  stepA :: (Aexp, State) -> Either Val Aexp
2  stepA (Num n,                    _)    = Left n
3  stepA (Var v,                    s)    = Left $ fetch v s
4  stepA (Sim x,                    s)    = case stepA (x, s) of
5                                               Left  x' -> Left  $ 0-x'
6                                               Right x' -> Right $ Sim x'
7
8  stepA (Add  (Num a) (Num b),     s)    = Left $ a + b
9  stepA (Add  (Num a)      b ,     s)    = case stepA (b, s) of
10                                              Left  b' -> Right $ Add (Num a) (Num b')
11                                              Right b' -> Right $ Add (Num a)      b'
12 stepA (Add       a       b ,     s)    = case stepA (a, s) of
13                                              Left  a' -> Right $ Add (Num a')     b
14                                              Right a' -> Right $ Add      a'      b
15
16 (...)
```

Este padrão da função stepA, para expressões aritméticas Add, repete-se novamente para multiplicações e subtrações. Assim, foi extraído para uma função auxiliar:

Ficheiro *SOS.hs*

```
14 stepAux f c (Num a) (Num b) s          = Left $ a `f` b
15 stepAux f c (Num a)      b  s          = case stepA (b, s) of
16                                              Left  b' -> Right $ c (Num a) (Num
                                                 ↪ b')
17                                              Right b' -> Right $ c (Num a)
                                                 ↪ b'
18 stepAux f c      a       b  s          = case stepA (a, s) of
19                                              Left  a' -> Right $ c (Num a')
                                                 ↪ b
20                                              Right a' -> Right $ c      a'
                                                 ↪ b
```

Assim, a implementação final da função stepA é a seguinte:

Ficheiro *SOS.hs*

```
22 stepA :: (Aexp, State) -> Either Val Aexp
23
24 stepA (Num n,                    _)    = Left n
25 stepA (Var v,                    s)    = Left $ fetch v s
26 stepA (Sim x,                    s)    = case stepA (x, s) of
27                                              Left  x' -> Left  $ 0-x'
28                                              Right x' -> Right $ Sim x'
29
30 stepA (Add  a b,                 s)    = stepAux (+) (Add)  a b s
31 stepA (Mult a b,                 s)    = stepAux (*) (Mult) a b s
32 stepA (Sub  a b,                 s)    = stepAux (-) (Sub)  a b s
```

4.2 Expressões Booleanas

Esta secção corresponde à resolução dos exercícios 5. b) e c) da ficha 2.

Uma semântica de transições *small-step* para as expressões booleanas pode ser definida de um modo análogo à semântica de transições para expressões aritméticas, com as devidas alterações. Essa função fica com o seguinte tipo: `stepB :: (Bexp, State) -> Either Bool Bexp`.

Para a avaliação de expressões base, a sua implementação é simples:

Ficheiro *SOS.hs*

```
40  stepB (Bexp.True,                _)    = Left Prelude.True
41  stepB (Bexp.False,               _)    = Left Prelude.False
```

Para a avaliação de expressões que envolvem a comparação entre duas expressões aritméticas, podemos recorrer à função auxiliar stepAux definida acima, pois o padrão é o mesmo:

Ficheiro *SOS.hs*

```
43  stepB (Equals a b,               s)    = stepAux (==) (Equals) a b s
44  stepB (GT     a b,               s)    = stepAux (>)  (GT)     a b s
45  stepB (GE     a b,               s)    = stepAux (>=) (GE)     a b s
46  stepB (LT     a b,               s)    = stepAux (<)  (LT)     a b s
47  stepB (LE     a b,               s)    = stepAux (<=) (LE)     a b s
```

Para a avaliação de negações, e devido ao facto de que defini dois construtores especiais para os tipos Verdadeiro e Falso, então a implementação é um pouco mais complicada:

Ficheiro *SOS.hs*

```
49  stepB (Not Bexp.True,            s)    = Left Prelude.False
50  stepB (Not Bexp.False,           s)    = Left Prelude.True
51  stepB (Not a,                    s)    = case stepB (a, s) of
52                                               Left  a' -> Right $ Not $ toBexp
                                                  ↪  a'
53                                               Right a' -> Right $ Not a'
```

Esta implementação tem duas partes essenciais:

- Parte que envolve a avaliação da negação de uma expressão booleana já totalmente simplificada (*SOS.hs*, linhas 49-50) - Neste caso, a função stepB apenas retorna a negação da expressão booleana
- Parte que envolve a negação de uma expressão booleana não simplificada (*SOS.hs*, linhas 51-53) - Neste caso, a função stepB vai executar um passo de avaliação da expressão booleana não simplificada. Se dessa execução tiver resultado um valor booleano, então a função stepB retorna uma expressão que é a negação esse valor (após utilizar a função toBexp para converter um valor booleano do Prelude para um valor booleano do tipo Bexp). Caso contrário, a função simplesmente retorna a negação da expressão não simplificada.

Para a avaliação de conjunções ou disjunções, a implementação tem três partes essenciais:

- A avaliação de expressões onde ambos os membros estão simplificados (*SOS.hs*, linhas 55-58 e 67-71) - Neste caso, apenas retornamos o resultado dessa avaliação.

- A avaliação de expressões onde apenas o membro direito está simplificado (*SOS.hs*, linhas 59-61 e 72-74) - Neste caso, vamos executar um passo de avaliação do membro direito, retornando depois uma nova expressão que corresponde à avaliação da mesma expressão, mas desta vez com a simplificação efetuada no membro direito.
- A avaliação de expressões onde o membro esquerdo não está simplificado (*SOS.hs*, linhas 63-65 e 75-77) - Neste caso, vamos executar um passo de avaliação do membro esquerdo. A função depois retorna uma nova expressão, que corresponde à avaliação da mesma expressão, mas com a diferença de que o membro esquerdo está um pouco mais simplificado.

A implementação da conjunção fica assim:

Ficheiro *SOS.hs*

```
55 stepB (And Bexp.True  Bexp.True,  s)    = Left Prelude.True
56 stepB (And Bexp.True  Bexp.False, s)    = Left Prelude.False
57 stepB (And Bexp.False Bexp.True,  s)    = Left Prelude.False
58 stepB (And Bexp.False Bexp.False, s)    = Left Prelude.False
59 stepB (And Bexp.True  b,          s)    = case stepB (b, s) of
60                                             Left  b' -> Right $ And Bexp.True
                                                ↪ $ toBexp b'
61                                             Right b' -> Right $ And Bexp.True
                                                ↪ b'
62 stepB (And Bexp.False b,         s)     = Right Bexp.False
    ↪ -- False && x == False, para todo o x. Avaliação curto-circuito do C.
63 stepB (And a           b,         s)     = case stepB (a, s) of
64                                             Left  a' -> Right $ And (toBexp
                                                ↪ a')        b
65                                             Right a' -> Right $ And         a'
                                                ↪ b
```

A implementação da disjunção é análoga. Pode ser encontrada no ficheiro *SOS.hs*, linhas 66-77.

Relativamente à implementação curto-circuito, como podemos ver, na linha 62 acima mostrada, temos um caso especial: no caso de uma conjunção onde o lado esquerdo é falso, então a função stepB retorna imediatamente o valor falso, sem avaliar o lado direito da expressão. O mesmo acontece no caso da disjunção, mas no respetivo caso (ou seja, uma disjunção onde o lado esquerdo é verdadeiro):

Ficheiro *SOS.hs*

```
71 stepB (Or Bexp.True  b,          s)     = Right Bexp.True
    ↪ -- True || x == True, para todo o x. Avaliação curto-circuito do C.
```

4.3 Comandos da Linguagem While

Esta secção corresponde à resolução do exercício 6. da ficha 2.

É agora pedida a definição da função stepSOS. Esta função implementa a relação de transição $< C, s > \Longrightarrow \gamma$. Como γ pode ser tanto um estado final (se não for possível executar mais nada no programa de entrada) ou um novo programa e um novo estado ($< C, s >$), decidi, por sugestão da professora, tipar a função do seguinte modo: `stepSOS :: (Stm, State) -> Either State (Stm, State)`.

Assim, a implementação desta função apenas segue as regras da semântica *small-step* que foram ensinadas nas aulas:

Ficheiro *SOS.hs*

```
80  stepSOS :: (Stm, State) -> Either State (Stm, State)
81
82  stepSOS (Skip,                  s)    = Left s
83  stepSOS (Assign var (Num n),    s)    = Left s'
84                                        where s' = update (var, n) s
85  stepSOS (Assign var exp,        s)    = case stepA (exp, s) of
86                                              Left  n    -> Right (Assign var
                                                ↪  (Num n), s)
87                                              Right exp' -> Right (Assign var
                                                ↪  exp',    s)
88
89  stepSOS (Comp c1 c2,            s)    = case stepSOS (c1, s) of
90                                              Left       s'  -> Right (c2,
                                                ↪  s')
91                                              Right (c3, s') -> Right (Comp c3
                                                ↪  c2, s')
92
93  stepSOS (ITE Bexp.True  c1 c2,  s)    = Right (c1, s)
94  stepSOS (ITE Bexp.False c1 c2,  s)    = Right (c2, s)
95  stepSOS (ITE   b        c1 c2,  s)    = case stepB (b, s) of
96                                              Left b' -> Right (ITE (toBexp b')
                                                ↪  c1 c2, s)
97                                              Right b' -> Right (ITE b' c1 c2,
                                                ↪  s)
98
99  stepSOS (If    b        c1,     s)    = stepSOS (ITE b c1 Skip, s)
100
101 stepSOS (While b        c,      s)    = Right (ITE b (Comp c (While b c))
    ↪  Skip, s)
```

Com recurso à função de *bind* do *Monad Either*, a função nstepsSOS pode ser escrita do seguinte modo:

Ficheiro *SOS.hs*

```
105 nstepsSOS 1 (c, s)                    = stepSOS (c, s)
106 nstepsSOS n (c, s)                    = stepSOS (c, s) >>= (nstepsSOS (n-1))
```

A função evalSOS também pode ser escrita em apenas uma linha devido ao *binding* monádico do *Either*:

Ficheiro *SOS.hs*

```
110 evalSOS (c, s)                        = stepSOS (c, s) >>= evalSOS
```

Isto funciona pois existe a garantia de que a função stepSOS apenas retorna apenas um estado se e só se não é possível executar mais passos no programa que foi dado como entrada.

Novamente, para testar esta função, podemos executar com recurso ao *GHCi*:

$ ghci SOS.hs

```
1  *SOS> main
2  ------------------------------------------------------------
3  > if x > 0 then y := x else if x < 0 then y := -x else z := 1 [x -> -1]
4  > if x > 0 then {y := x} else {if x < 0 then {y := -x} else {z := 1}}
5    [x -> -1]
6  > if -1 > 0 then {y := x} else {if x < 0 then {y := -x} else {z := 1}}
7    [x -> -1]
8  > if false then {y := x} else {if x < 0 then {y := -x} else {z := 1}}
9    [x -> -1]
10 > if x < 0 then {y := -x} else {z := 1}
11   [x -> -1]
12 > if -1 < 0 then {y := -x} else {z := 1}
13   [x -> -1]
14 > if true then {y := -x} else {z := 1}
15   [x -> -1]
16 > y := -x
17   [x -> -1]
18 > y := 1
19   [x -> -1]
20 > [x -> -1][y -> 1]
21 ------------------------------------------------------------
22 ------------------------------------------------------------
23 > x := 1; while b > 0 do { x := x * a; b := b - 1 } [a -> 3][b -> 2]
24 (... Saída omitida devido ao tamanho ...)
25 > skip
26   [a -> 3][b -> 0][x -> 9]
27 > [a -> 3][b -> 0][x -> 9]
28 ------------------------------------------------------------
```

Como podemos ver, com recurso à função main, é possível consultar a saída de todos os passos individuais da execução de um programa *While*.

No entanto, tal como descrito no capítulo anterior, também é possível executar as funções individualmente e manualmente, segundo o processo descrito no Apêndice B.

Capítulo 5

Máquinas Abstratas

Nesta secção é descrito o processo de construção de uma implementação da semântica das três máquinas abstratas AM, AM1 e AM2.

5.1 Máquina AM

Antes de começar a resolver o exercício 5 da ficha 3, achei pertinente implementar a máquina AM. A maior parte do código poderá depois ser facilmente adaptado para as máquinas AM1 e AM2.

5.1.1 Instruções e Estrutura

Sendo assim, comecei por criar uma estrutura correspondente a todas as instruções que a máquina AM consegue executar. Visto que estendi a linguagem *While* com comandos extra, estendi também as instruções da máquina com instruções compatíveis com as extensões introduzidas na linguagem *While*:

Ficheiro *AMCode.hs*

```
7  data Inst = PUSH Val
8            | ADD
9            | MULT
10           | SUB
11           | TRUE
12           | FALSE
13           | EQ
14           | GT
15           | GE
16           | LT
17           | LE
18           | AND
19           | OR
20           | NEG
21           | FETCH Var
22           | STORE Var
23           | NOOP
24           | BRANCH Code Code
25           | LOOP Code Code
```

Assim, código da máquina AM é representado pelo seguinte tipo:

Ficheiro *AMCode.hs*

```
27 newtype Code = Code [Inst]
```

Tal como explicado anteriormente, o tipo *Code* é implementado com recurso a *newtype* para ser possível criar uma implementação de *Show* para o mesmo. Não existe implementação de *Read* para este tipo, pois todo o código deste tipo é gerado a partir de código da linguagem *While*.

Defini também um tipo para a *Stack*:

Ficheiro *Stack.hs*

```
5 newtype Stack = Stack [Val]
```

O tipo *Val* é definido no ficheiro *State.hs*. É utilizado em todo o lado em vez de utilizar diretamente *Int*, apenas para manter a consistência de tipos em todo o código. Assim, se fosse necessário alterar, essa alteração apenas tinha de ser feita num lugar.

A stack da máquina AM é apenas uma stack de inteiros. Isto é uma simplificação em relação ao que demos nas aulas. Valores booleanos são representados de acordo com a convenção da linguagem C:

- Verdadeiro - Qualquer valor inteiro diferente de 0
- Falso - Valor inteiro 0

Assim, a máquina AM é um triplo: `(Code, Stack, State)`.

5.1.2 Semântica

A semântica das instruções foi atribuida de acordo com o ensinado nas aulas.

As instruções PUSH, ADD, MULT, SUB, TRUE e FALSE são as mais simples de atribuir semântica:

Ficheiro *AM.hs*

```
15 stepAM (Code ((PUSH n):c), Stack e,       s) = (Code c, Stack $ n:e,   s)
16 stepAM (Code (ADD:c),      Stack (a:b:e), s) = (Code c, Stack $ a+b:e, s)
17 stepAM (Code (MULT:c),     Stack (a:b:e), s) = (Code c, Stack $ a*b:e, s)
18 stepAM (Code (SUB:c),      Stack (a:b:e), s) = (Code c, Stack $ a-b:e, s)
19 stepAM (Code (TRUE:c),     Stack e,       s) = (Code c, Stack $ 1:e,   s)
20 stepAM (Code (FALSE:c),    Stack e,       s) = (Code c, Stack $ 0:e,   s)
```

As instruções de comparação (EQ, GT, GE, LT, LE) seguem todas a seguinte estrutura, com as devidas adaptações:

Ficheiro *AM.hs*

```
23 stepAM (Code (EQ:c), Stack (a:b:e), s) | a == b    = (Code c, Stack $ 1:e, s)
24                                        | otherwise = (Code c, Stack $ 0:e, s)
```

As instruções de conjunção e disjunção seguem a seguinte estrutura, com as devidas adaptações:

Ficheiro *AM.hs*

```
35 stepAM (Code (AND:c), Stack (a:b:e), s) | a /= 0 && b /= 0 = (Code c, Stack $
    ↪ 1:e, s)
36                                         | otherwise        = (Code c, Stack $
                                            ↪ 0:e, s)
```

A instrução de negação também tem uma implementação simples:

Ficheiro *AM.hs*

```
39 stepAM (Code (NEG:c), Stack (a:e),   s) | a /= 0           = (Code c, Stack $
    ↪ 0:e, s)
40                                         | otherwise        = (Code c, Stack $
                                            ↪ 1:e, s)
```

As instruções FETCH e STORE necessitam de aceder ao estado. Para isso, fazem uso das funções *fetch* e *update* definidas no ficheiro *State.hs*:

Ficheiro *State.hs*

```
20 update :: VarState -> State -> State
21 update (x, v) (State []) = State [(x, v)]
22 update (x, v) (State ((var, val):s))
23         | var == x  = State $ (var, v) : s
24         | otherwise = let (State s') = update (x, v) (State s)
25                       in  (State $ (var, val):s')
26
27 fetch :: Var -> State -> Val
28 fetch v (State []) = 0
29 fetch v (State ((var, val):s)) | var == v  = val
30                                | otherwise = fetch v (State s)
```

Assim, FETCH e STORE definem-se como:

Ficheiro *AM.hs*

```
42 -- Semântica para a instrução FETCH
43 stepAM (Code ((FETCH v):c), Stack e, s)     = (Code c, Stack $ (fetch v s):e, s)
44
45 -- Semântica para a instrução STORE
46 stepAM (Code ((STORE v):c), Stack (x:e), s) = let s' = update (v, x) s
47                                              in (Code c, Stack e, s')
```

A instrução NOOP simplesmente não atua sobre o estado nem a stack. Por ser tão trivial, não irei apresentar a sua definição.

A instrução BRANCH pode ser definida do seguinte modo:

Ficheiro *AM.hs*

```
53 stepAM (Code ((BRANCH (Code c1) (Code c2)):c), Stack (t:e), s)
54         | t /= 0 = (Code $ c1 ++ c, Stack e, s)
55         | otherwise = (Code $ c2 ++ c, Stack e, s)
```

Por último, a instrução LOOP é definida à custa da instrução BRANCH, tal como ensinado nas aulas:

Ficheiro *AM.hs*

```
58 stepAM (Code ((LOOP (Code c1) (Code c2)):c), e, s) = (Code $ c1 ++ [BRANCH (Code
    ↪  $ c2 ++ [LOOP (Code c1) (Code c2)]) (Code [NOOP])], e, s)
```

5.1.3 Compilador de programas *While*

Um compilador é, neste contexto, apenas uma função com o seguinte tipo: `cmpS :: Stm -> Code`.

Mas antes de ser possível compilar programas *While* completos, temos de ser capazes de compilar expressões aritméticas. Assim, a compilação de expressões aritméticas dá-se do seguinte modo:

Ficheiro *AM.hs*

```
71  cmpA :: Aexp -> Code
72
73  cmpA (Num n)       = Code [PUSH n]
74  cmpA (Sim n)       = cmpA $ Sub (Num 0) n -- -x == 0-x
75  cmpA (Var v)       = Code [FETCH v]
76
77  cmpA (Add a1 a2)  = cmpAux (cmpA) ADD a1 a2
78  cmpA (Mult a1 a2) = cmpAux (cmpA) MULT a1 a2
79  cmpA (Sub a1 a2)  = cmpAux (cmpA) SUB a1 a2
```

A função cmpAux apresentada é uma abstração de um padrão comum que se repete várias vezes ao longo do compilador, onde as instruções geradas têm o seguinte padrão: op2:op1:OP, onde op2 são as instruções que levam à produção do operando 2 no topo da stack, op1 são as instruções que levam à produção do operando 1 no topo da stack, e OP é a instrução corresponde à operação que se pretende efetuar (como por exemplo uma soma). Está definida assim:

Ficheiro *AM.hs*

```
66  cmpAux f i x1 x2 = Code $ x2' ++ x1' ++ [i]
67                     where (Code x1') = f x1
68                           (Code x2') = f x2
```

Também temos de ser capazes de compilar expressões booleanas:

Ficheiro *AM.hs*

```
82  cmpB :: Bexp -> Code
83
84  cmpB Bexp.True       = Code $ [TRUE]
85  cmpB Bexp.False      = Code $ [FALSE]
86
87  cmpB (Equals  a1 a2) = cmpAux (cmpA) (EQ) a1 a2
88  cmpB (Bexp.GT a1 a2) = cmpAux (cmpA) (AMCode.GT) a1 a2
89  cmpB (Bexp.GE a1 a2) = cmpAux (cmpA) (AMCode.GE) a1 a2
90  cmpB (Bexp.LT a1 a2) = cmpAux (cmpA) (AMCode.LT) a1 a2
91  cmpB (Bexp.LE a1 a2) = cmpAux (cmpA) (AMCode.LE) a1 a2
92
93  cmpB (Not b)         = Code $ b' ++ [NEG]
94                         where (Code b') = cmpB b
95
96  cmpB (And b1 b2)     = cmpAux (cmpB) (AND) b1 b2
97  cmpB (Or  b1 b2)     = cmpAux (cmpB) (OR) b1 b2
```

Podemos agora compilar qualquer expressão da linguagem *While*:

Ficheiro *AM.hs*

```
100 cmpS :: Stm -> Code
101
102 cmpS Skip            = Code $ [NOOP]
103
104 cmpS (Assign x a)    = Code $ a' ++ [STORE x]
105                        where (Code a') = cmpA a
106
107 cmpS (Comp s1 s2)    = Code $ s1' ++ s2'
108                        where (Code s1') = cmpS s1
109                              (Code s2') = cmpS s2
110
111 cmpS (ITE b s1 s2) = Code $ b' ++ [BRANCH (cmpS s1) (cmpS s2)]
112                       where (Code b') = cmpB b
113 cmpS (If  b s)     = cmpS (ITE b s Skip)
114
115 cmpS (While b s)   = Code $ [LOOP (cmpB b) (cmpS s)]
```

5.1.4 Função Semântica S_{am}

A função semântica S_{am} pode ser definida do seguinte modo:

Ficheiro *AM.hs*

```
118 sAM :: Stm -> (State -> State)
119 sAM c = \s -> let (_, _, s') = evalAM (cmpS c, Stack [], s)
120               in s'
```

Esta função pode, como de costume, ser testada no *GHCi*:

$ ghci AMRepl.hs

```
1 *AMRepl> sAM (read "while x > 0 do x := x - 1") (read "[x -> 3]")
2 [x -> 0]
```

5.2 Máquina AM1

Esta secção corresponde à resolução dos exercícios 5. a), b) e c) da ficha 3.

5.2.1 Memória

A arquitetura da máquina AM1 substitui o estado por uma memória, análoga à memória de um computador tradicional: é apenas uma grande lista de valores inteiros. Assim, defini uma estrutura especializada para esta memória, bem como funções para a manipular:

Ficheiro *Memory.hs*

```
5  type Address = Val
6  newtype Memory = Memory [Val] deriving Eq
```

Ficheiro *Memory.hs*

```
13 put :: (Address, Val) -> Memory -> Memory
14 put (0, v) (Memory []) = Memory [v]
15 put (n, v) (Memory []) = Memory $ (replicate n 0) ++ [v]
16 put (n, v) (Memory mem) | length mem <  n = Memory $ mem ++ (replicate (n - 1 -
    ↪  length mem) 0) ++ [v]
17                         | length mem == n = Memory $ (take (n) mem) ++ [v]
18                         | otherwise       = Memory $ memBefore ++ [v] ++
                            ↪  memAfter
19                                             where memBefore = take (n) mem
20                                                   memAfter = drop (n+1) mem
21
22 get :: Address -> Memory -> Val
23 get _ (Memory []) = 0
24 get n (Memory m) = m !! n
```

A função get é bastante simples: apenas consulta na memória o valor do endereço que foi passado de entrada.

A função put já é mais complexa: dado um endereço e um valor, tem de colocar na memória esse valor, mas no endereço certo. Se, por acaso, a lista que representa a memória for demasiado pequena, então a função put tem de aumentar o tamanho da lista. Cada caso especial é tratado individualmente na função.

5.2.2 Instruções e Estrutura

Segundo o pedido no exercício, a máquina AM1 deixa de ter as instruções FETCH-x e STORE-x, passando a ter, respetivamente, as instruções GET-n e PUT-n.

Isto é representado na lista de instruções da máquina AM1 pelas seguintes alterações:

Ficheiro *AM1Code.hs*

```
22  | PUT Address
23  | GET Address
```

5.2.3 Semântica

Mantém-se a maior parte da semântica anteriormente atribuída. No entanto, foram removidas as implementações de semântica relativas às instruções FETCH e STORE, sendo introduzida semântica para as instruções GET e PUT:

Ficheiro *AM1.hs*

```
44  -- Semântica para a instrução GET
45  stepAM1 (Code ((GET a):c), Stack e, Memory m) = (Code c, Stack $ (get a (Memory
    ↪  m)):e, Memory m)
46
47  -- Semântica para a instrução PUT
48  stepAM1 (Code ((PUT a):c), Stack (n:e), m) = let m' = put (a, n) m
49                                               in (Code c, Stack e, m')
```

5.2.4 Compilador

O compilador desta máquina é, geralmente, igual ao compilador da máquina AM. No entanto, o compilador de AM1 não aceita apenas um programa *While*; aceita também um Estado. Um estado é uma abstração conveniente que já tinha sido anteriormente criada, e que se adequa perfeitamente às nossas necessidades: este estado será responsável por associar às variáveis do programa *While* localizações na memória.

A função de atribuição de endereço a uma variável é simples:

Ficheiro *Memory.hs*

```
40  env :: Var -> State -> Val
41  env v (State []) = 0
42  env v s = case lookupVar v s of
43                Nothing -> 1 + numVars s
44                Just  n -> n
```

Esta função é simples: consulta o estado à procura da variável pedida. Se por acaso essa variável não existir, então atribui-lhe um endereço: esse endereço é simplesmente o número de variáveis já com endereço atribuído + 1.

Assim, para compilar Assignments da linguagem *While*, é necessário ter em conta os endereços:

Ficheiro *AM1.hs*

```
107 cmpSAux (Assign x a, s)       = (Code $ a' ++ [PUT addr], s'')
108                                 where addr = env x s
109                                       s'   = update (x, addr) s
110                                       (Code a', s'') = cmpA (a, s')
```

Nas restantes equações das funções de compilação, é apenas necessário ter cuidado para gerir corretamente o estado, atualizando-o sempre que necessário, ou sempre que possam ter ocorrido alterações no mesmo em execuções recursivas da função de compilação.

5.2.5 Função Semântica S_{am1}

A função semântica S_{am1} pode ser definida do seguinte modo:

Ficheiro *AM1.hs*

```
130 sAM1 :: Stm -> (Memory -> Memory)
131 sAM1 c = \s -> let (_, _, s') = evalAM1 (fst $ cmpS c, Stack [], s)
132                in s'
```

Esta função pode, como de costume, ser testada no *GHCi*:

$ ghci AM1Repl.hs

```
1 *AM1Repl> sAM1 (read "while x > 0 do x := x - 1") (setupMemory (read "[x -> 3]") (read
   ↪  "[x -> 0]") (Memory []))
2 0
```

A função setupMemory recebe dois estados: o primeiro é o estado inicial das variáveis do programa While; o segundo é a associação entre o nome de uma variável e o seu endereço de memória. O terceiro argumento é para permitir uma recursividade na definição da função; basta ser inicializado como vazio.

5.3 Máquina AM2

Esta secção corresponde à resolução dos exercícios 5. d), e) e f) da ficha 3.

5.3.1 Instruções e Estrutura

A máquina AM2 introduz, em relação à AM1, um *program counter* para que não seja necessária modificação do código da máquina. Esta máquina substitui também as instruções BRANCH() e LOOP() por instruções de LABEL-ℓ, JUMP-ℓ e JUMPFALSE-ℓ. Assim, a semântica desta máquina nunca modifica o código da mesma; apenas modifica o *program counter*.

As instruções JUMP-ℓ e JUMPFALSE-ℓ saltam para o endereço onde se encontra a instrução LABEL-ℓ. Sendo assim, quando encontra uma destas instruções, a máquina tem pelo menos duas opções:

1. Procurar em todo o código onde se encontra o LABEL pretendido;
2. A própria instrução JUMP-ℓ e JUMPFALSE-ℓ pode incluir o endereço relativo da instrução LABEL-ℓ pretendida.

A minha implementação opta pela segunda opção.

Sendo assim, as novas instruções introduzidas são:

Ficheiro *AM2Code.hs*

```
25  | LABEL Val
26  | JUMP Val Val -- JUMP-l: o primeiro Val é o l, e o segundo Val é o endereço
    ↪  relativo da localização da instrução LABEL-l respetiva
27  | JUMPFALSE Val Val -- JUMPFALSE-l: o primeiro Val é o l, e o segundo Val é o
    ↪  endereço relativo da localização da instrução LABEL-l respetiva
```

5.3.2 Semântica

A semântica da maior parte das instruções foi alterada para não modificar o código, mas sim o *program counter*.

Às instruções de JUMP e JUMPFALSE foi-lhes atribuida a seguinte semântica:

Ficheiro *AM2.hs*

```
63  -- Semântica para a instrução JUMP-l
64  stepAM2Aux (pc, JUMP l n, e, s) = (pc + n, e, s)
65
66  -- Semântica para a instrução JUMPFALSE-l
67  stepAM2Aux (pc, JUMPFALSE l n, Stack (x:e), s) | x == 0    = (pc + n, Stack e,
        s)
68                                                 | otherwise = (pc + 1, Stack e,
                                                       s)
```

5.3.3 Compilador

O compilador da linguagem *While* para os comandos If-Then-Else, If-Then, e While tiveram de ser alterados para acomodar as novas instruções.

O número do último *label* atribuído é guardado no Estado que guarda os endereços das variáveis, sob o nome "$LABEL". Sempre que é necessário criar um novo *label*, o estado é consultado para saber qual deve ser o próximo número de *label* a atribuir.

No caso do If-Then-Else, foi criado um *label* no código que é o corpo do Then e o corpo do Else. As instruções geradas são, portanto, a avaliação da condição b; depois um JUMPFALSE, que, se b for falso, salta para o corpo do Else, recorrendo ao *label* lá colocado. caso contrário, a execução continua normalmente.

Ficheiro *AM2.hs*

```
128 cmpSAux (ITE b s1 s2, s) = (Code $ b' ++ [JUMPFALSE labelElse offsetElse] ++ s1'
        ++ [JUMP labelAfter offsetAfter, LABEL labelElse] ++ s2' ++ [LABEL
        labelAfter], s''')
129                              where labelElse        = fetch "$LABEL" s
130                                    labelAfter       = labelElse + 1
131                                    (Code b', s')    = cmpB (b, s)
132                                    (Code s1', s'')  = cmpSAux (s1, update
                                           ("$LABEL", 1 + labelAfter) s')
133                                    (Code s2', s''') = cmpSAux (s2, s'')
134                                    offsetElse       = 1 + length s1'
135                                    offsetAfter      = 2 + length s2'
```

O caso do While contém dois *labels*: um dos *labels* corresponde ao código que se encontra APÓS TODO o ciclo While; outro *label* encontra-se exatamente no início do ciclo While. O While é depois implementado com dois saltos: um JUMPFALSE que salta para o *label* final, no caso de a condição do While ser falsa; e, no final da execução do corpo do While, encontra-se um JUMP incondicional que salta sempre para o *label* que se encontra no início de todo o ciclo While.

Ficheiro *AM2.hs*

```
139 cmpSAux (While b c, s) = (Code $ [LABEL labelBefore] ++ b' ++ [JUMPFALSE
    ↪ labelAfter offsetAfter] ++ c' ++ [JUMP labelBefore offsetBefore, LABEL
    ↪ labelAfter], s'')
140                           where labelBefore    = fetch "$LABEL" s
141                                 labelAfter     = labelBefore + 1
142                                 (Code b', s')  = cmpB (b, s)
143                                 (Code c', s'') = cmpSAux (c, update ("$LABEL", 1
                                     ↪ + labelAfter) s')
144                                 offsetBefore   = -(length c') - 1 - (length b') -
                                     ↪ 1
145                                 offsetAfter    = 1 + (length c') + 1
```

5.3.4 Função semântica S_{am2}

A função semântica S_{am2} é idêntica à função semântica induzida por AM1:

Ficheiro *AM2.hs*

```
151 sAM2 :: Stm -> (Memory -> Memory)
152 sAM2 c = \s -> let (_, _, _, s') = evalAM2 (0, fst $ cmpS c, Stack [], s)
153                in s'
```

Esta função pode, como de costume, ser testada no *GHCi*:

$ ghci AM2Repl.hs

```
1 *AM2Repl> sAM2 (read "while x > 0 do x := x - 1") (setupMemory (read "[x -> 3]") (read
    ↪ "[x -> 0]") (Memory []))
2 0
```

Capítulo 6

Conclusão

Este trabalho permitiu uma profunda consolidação dos conhecimentos sobre todas as semânticas que estudamos ao longo das aulas da UC; permitiu também uma profunda consolidação de conhecimentos sobre a linguagem de programação Haskell.

Apêndice A

Parser para a linguagem While

No ficheiro *Parser.y* está incluido um *Parser* para a linguagem *While* que estudamos, bem como a notação que utilizamos para os estados das variáveis. Este *Parser* é baseado numa gramática independente de contexto, e utiliza a *library* Happy para o seu funcionamento.

A sua compilação num ficheiro Haskell é simples: após instalação, basta executar o comando *happy Parser.y*. Por conveniência, é já incluído um ficheiro Parser.hs, derivado a partir da descrição da gramática, no ficheiro ZIP que contém todo o trabalho.

Apenas com a inclusão do *Parser* é que são declaradas instâncias de *Read* para os tipos da linguagem *While*; se este *Parser* não for incluido, não é possível recorrer às instâncias de *Read*.

Apêndice B

Como correr os vários programas no interpretador GHCi

Para correr um dos programas no *GHCi* é só necessário carregar o ficheiro correto no *GHCi*. Cada semântica tem o seu ficheiro principal:

- Semântica Natural - NS.hs
- Semântica Small-Step - SOS.hs
- Máquina AM - AMRepl.hs
- Máquina AM1 - AM1Repl.hs
- Máquina AM2 - AM2Repl.hs

Se for incluido um destes ficheiros, é sempre possível chamar a função main, que inicia um Read-Eval-Print-Loop (REPL), que espera, no stdin, por programas da linguagem *While*, e depois, dependendo do programa carregado, executa os vários passos de execução desse programa de acordo com a semântica que está carregada, ou no caso das máquinas, compila o código, e executa na respetiva máquina.

A execução de qualquer um destes ficheiros também incluí todos os símbolos necessários à construção de expressões para a respetiva semântica e/ou máquina.

Apêndice C

Como compilar os vários programas com recurso à Makefile

Com recurso à Makefile, é possível executar *make* para compilar um executável por programa.

Podem ser compilados os seguintes programas:

- make NS
- make SOS
- make AM
- make AM1
- make AM2

A execução de um programa compilado deste modo funciona do mesmo modo do que a execução da função main descrita no Apêndice B.