Licenciatura em Ciências da Computação

Semântica de Linguagens de Programação 2021/2022

A70373 Alexandre Rodrigues Baldé

Conteúdo:

1. Tipos de dados para a linguagem While
2. Semântica Natural
3. Semântica Operacional Estrutural
4. Detalhes de Implementação da AM1
5. Detalhes de Implementação da AM2

Tipos de dados para a linguagem While

Abaixo estão as extensões de Haskell e `import` s utilizados neste projeto.

In [ ]:
```
:ext InstanceSigs
:ext BangPatterns
```

In [ ]:
```
import qualified Data.Map   as M
import qualified Data.Maybe as Maybe

import           Data.List (intercalate)

-- For Natural Semantics
import qualified Control.Monad.State.Strict as St
import           Control.Monad              (when)
```

Dada a nomenclatura usada na disciplina, definiram-se alguns sinónimos `type` para mais fácil leitura do código.

In [ ]:
```
type Z = Integer

type Var = String
```

Vejam-se abaixo os ADTs definidos para expressões aritméticas, e booleanas.

In [ ]:
```
data Aexp
    = Num Z
    | Var Var
    | Aexp `Plus` Aexp
    | Aexp `Mul` Aexp
    | Aexp `Minus` Aexp
    deriving Eq
```

In [ ]:
```
data Bexp
    = T
    | F
    | Aexp `Eq` Aexp
    | Aexp `Le` Aexp
    | Aexp `Lt` Aexp
    | Aexp `Ge` Aexp
    | Not Bexp
    | Bexp `And` Bexp
    | Bexp `Or` Bexp
    deriving Eq
```

As funções abaixo estão definidas na biblioteca `semantics` no módulo `Core` ; as suas assinaturas explicam o seu propósito.

Num dos exercícios da ficha 1 pedia-se uma função de atualização de estado, chamada aqui de `stUpdate` .

In [ ]:
```
type State = M.Map Var Z

getSt :: State -> Var -> Z
getSt st var = Maybe.fromMaybe 0 (M.lookup var st)

stUpdate :: State -> Var -> Z -> State
stUpdate st var v = M.insert var v st
```

In [ ]:
```
subAexp :: Var -> Aexp -> Aexp -> Aexp
subBexp :: Var -> Aexp -> Bexp -> Bexp

arithEval :: Aexp -> (State -> Z)
boolEval :: Bexp -> (State -> Bool)
```

State Helpers

In [ ]:
```haskell
-- Igual ao tipo State, mas com instância de Show legível.
newtype State' = St {
    getState :: State
    } deriving (Eq)

instance Show State' where
    show = showState . getState
```

In [ ]:
```haskell
-- ADT para statements da linguagem While
data Stm
    = Var `Assign` Aexp
    | Skip
    | Stm `Comp` Stm
    | IfThenElse Bexp Stm Stm
    | WhileDo Bexp Stm
    deriving (Eq)
```

Programas e estado exemplo

Retirados da ficha 1, e usados para testar as várias funcionalidades desenvolvidas (NS, SOS, AM1, AM2).

In [ ]:
```
ex1State :: State
ex1State = M.fromList [("n", 6), ("x", 3), ("y", 2)]

swap :: Stm
swap = Comp (Comp c1 c2) c3
    where
        c1 = Assign "n" (Var "x")
        c2 = Assign "x" (Var "y")
        c3 = Assign "y" (Var "n")

minProg :: Stm
minProg = IfThenElse b1 if2 if3
    where b1 = Lt (Var "x") (Var "y")
          b2 = Lt (Var "x") (Var "z")
          b3 = Lt (Var "y") (Var "z")
          if3 = IfThenElse b3 (Assign "m" (Var "y")) (Assign "m" (Var "
z"))
          if2 = IfThenElse b2 (Assign "m" (Var "x")) (Assign "m" (Var "
z"))

expProg :: Stm
expProg = Comp assgn while
    where
        assgn = Assign "r" (Num 1)
        while = WhileDo bexp whileStm
        bexp = Var "y" `Ge` Num 1
        whileStm = Comp
            (Assign "r" (Mul (Var "r") (Var "x")))
            (Assign "y" (Minus (Var "y") (Num 1)))

fact :: Stm
fact = Comp assgn while
    where
        assgn = Assign "f" (Num 1)
        while = WhileDo bexp whileStm
        bexp = Var "n" `Ge` Num 1
        whileStm = Comp
            (Assign "f" (Mul (Var "f") (Var "n")))
            (Assign "n" (Minus (Var "n") (Num 1)))
```

Avaliação de Programas por Semântica Natural

In [ ]:
```
evalNS :: State -> Stm -> State
evalNS st stm = St.execState (helper stm) st
    where
        helper :: Stm -> St.State State ()
        ...
```

In [ ]:
```
evalNS ex1State fact
```

```
fromList [("f",720),("n",0),("x",3),("y",2)]
```

In [ ]:
```
evalNS ex1State expProg
```

```
fromList [("n",6),("r",9),("x",3),("y",0)]
```

Avaliação de Programas por Semântica Operacional Estrutural

In [ ]:
```
stepSOS :: State -> Stm -> Either State (Stm, State)
nstepsSOS :: State -> Stm -> Integer -> [Either State' (Stm, State')]

-- Imprimir uma configuração numa string legível
-- e.g. mapM_ (putStr . helperSOS) $ nstepsSOS ex1State fact 10
helperSOS :: Either State' (Stm, State') -> String

evalSOS :: State -> Stm -> State'
```

Sobre a implementação de SOS em `semantics.StructOpSemantics` :

- **Uma** transição (`stepSOS`) em semântica operacional estrutural:
  - ou dá origem a um estado (`Left`)
  - ou dá origem a um comando intermédio, juntamente com um novo estado (`Right`).
- Várias transições em SOS (`nstepsSOS`):
  - devolve a configuração que resulta de dar o número de passos de execução pedido, se for possível
  - o resultado é uma lista que também tem todas as transições que precederam a última.
- Avaliação de um programa While num dado estado (`evalSOS`):
  - Retorna um `State'` definido acima - igual a `State` mas com instância `String` legível.

In [ ]:
```
evalSOS ex1State fact
```

```
f := 720; n := 0; x := 3; y := 2
```

In [ ]:
```
evalSOS ex1State expProg
```

```
n := 6; r := 9; x := 3; y := 0
```

Máquina Abstrata 1 (AM1)

Os tipos abaixo são usados para implementar AM1 (alguns são reutilizados/modificados para AM2).

- O tipo `Env` é um mapeamento de variáveis a posições em memória.
- À medida que se compila o código de AM1, vão-se atribuindo posições em memória na primeira ocorrência de cada variável
  - o ADT `EnvStateAM1` e o tipo `NextAddr` servem para registar esta informação. Incialmente, o mapeamento tem que ser `Map.empty` , e o próximo endereço `0` .

In [ ]:
```haskell
type Env = M.Map Var Z

getEnv :: Env -> Var -> Z
getEnv e var = e M.! var

type NextAddr = Z

data EnvStateAM1 = EnvSt {
    getEnvSt :: !Env,
    getNxtAdr :: !NextAddr
    } deriving (Eq, Show)

getEnv :: Env -> Var -> Z
getEnv e var = e M.! var
```

In [ ]:
```haskell
data AM1Instr
    = PUSH Z
    | ADD
    | MULT
    | SUB
    | TRUE
    | FALSE
    | EQUAL
    | LE
    | GE
    | LTHAN
    | AND
    | OR
    | NEG
    | PUT Z
    | GET Z
    | NOOP
    | BRANCH AM1Code AM1Code
    | LOOP AM1Code AM1Code
    deriving (Eq, Show)

type AM1Code = [AM1Instr]
```

`aexpToAM1Code` e `bexpToAM1Code` correspondem a $\mathcal{CA}$ e $\mathcal{CB}$, respetivamente.
Para poderem fazer sempre a associação correta entre variáveis, recebem como argumento um `EnvStateAM1` que é atualizado e passado às chamadas recursivas.

Note-se que se poderia fazer uso da mónade `State` (https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-State-Strict.html#t:State), e de facto, é isto que acontece na implementação de AM2.

In [ ]:
```haskell
aexpToAM1Code :: EnvStateAM1 -> Aexp -> (AM1Code, EnvStateAM1)
aexpToAM1Code m@(EnvSt e nxtAdr) a = case a of
    Num n -> ([PUSH n], m)
        ...

bexpToAM1Code :: EnvStateAM1 -> Bexp -> (AM1Code, EnvStateAM1)
bexpToAM1Code m@(EnvSt e nxtAdr) b = case b of
    T -> ([TRUE], m)
    F -> ([FALSE], m)
        ...
```

In [ ]:
```haskell
whileToAM1 :: Stm -> (AM1Code, EnvStateAM1)
whileToAM1 stm = St.runState (helper stm) (EnvSt M.empty 0)
    where
        helper :: Stm -> St.State EnvStateAM1 AM1Code
        helper (var `Assign` aexp) = do
            ...
        helper (WhileDo b c) = do
            memSt <- St.get
            let (predCode, memSt') = bexpToAM1Code memSt b
            St.put memSt'
            loopCode <- helper c
            return [LOOP predCode loopCode]
```

Uma configuração de AM1 é de tipo $\mathrm{Code} \times \mathrm{Stack} \times \mathrm{Memory}$; abaixo definem-se os últimos dois tipos.

- O tipo Stack é uma lista de `Either Z Bool` - note-se que uma operação que espere ter um `x : Left Z` no topo da pilha e encontra `y : Right Bool` lançará erro, e vice-versa.
- Tanto aqui como na AM2, usa-se um `Map` para definir `Memory`, que é um mapeamento dos endereços calculados durante a compilação para os valores das variáveis correspondentes.

In [ ]:
```haskell
type Stack = [Either Z Bool]

type Memory = M.Map Z Z

type AM1Config = (AM1Code, Stack, Memory)
```

`stepAM1` recebe uma configuração, e dá um passo de execução da máquina AM1.
A função tem que ser definida para todos os casos de `AM1Instr`, pelo que por brevidade apenas se deixaram alguns exemplos.

In [ ]:
```
stepAM1 :: AM1Config -> AM1Config
stepAM1 conf@([], stack, mem) = conf
stepAM1 (c : cs, stack, mem) = case c of
    PUSH n -> (cs, Left n : stack, mem)
    ADD -> case stack of
        Left z1 : Left z2 : stack' ->
            (cs, Left (z1 + z2) : stack, mem)
        _ -> error "ADD: invalid stack for operation!"
    ...
    BRANCH ins ins' -> case stack of
      Right b : stack' ->
          let instr = if b then ins else ins'
          in (instr, stack', mem)
      _   -> error "BRANCH: invalid stack for operator!"

    LOOP ins ins' -> (ins ++ [BRANCH (ins' ++ [LOOP ins ins']) [NOOP]] ++
cs, stack, mem)
```

Para poder executar um programa While na máquina AM1, é preciso um estado inicial a partir do qual se carregarão os valores iniciais das variáveis em memória.

A função `initConfigAM1` faz o descrito acima, devolvendo também o `Env` calculado após compilar o programa em "*bytecode*" AM1 - isto é necessário porque:

- a compilação $\mathrm{While} \to \mathrm{AM1Code}$ produz um mapeamento $\mathrm{Var} \to \mathrm{Addr}$
- após execução da máquina abstrata em `runStmInAM1` tem-se um mapeamento entre $\mathrm{Addr} \to \mathrm{Z}$ (variáveis a valores finais)

donde, para poder devolver os valores finais de cada variável, são necessários ambos `Env` e `Memory`.

In [ ]:
```
initConfigAM1 :: State -> Stm -> (AM1Config, Env)

-- Devolve as variáveis usadas no programa, e os valores que estavam nas
respetivas
-- posições de memória aquando da terminação da execução.
-- Pode não terminar! (Halting problem).
runStmInAM1 :: State -> Stm -> M.Map Var Z
```

In [ ]:
```
runStmInAM1 ex1State minProg
runStmInAM1 ex1State swap
runStmInAM1 ex1State expProg
runStmInAM1 ex1State fact
```

```
fromList [("m",0),("x",3),("y",2),("z",0)]

fromList [("n",3),("x",2),("y",3)]

fromList [("r",9),("x",3),("y",0)]

fromList [("f",720),("n",0)]
```

Máquina Abstrata 2 (AM2)

O tipo `Env` seguinte é semelhante àquele definido em AM1, acrescentado de uma noção de *program counter*.

In [ ]:
```
type Env = M.Map Var Z

type NextAddr = Z

-- Program counter associated with each instruction.
-- Must be positive, starts at 1, each instruction has a unique PC value,
-- and strictly increases by 1 unit with every atomic instruction.
type ProgramCounter = Z

type Stack = [Either Z Bool]

type Memory = M.Map Z Z
```

Como referido na Ficha 3, as instruções `LABEL-l`, `JUMP-l`, `JUMPFALSE-l` de AM2 substituem `BRANCH` e `LOOP`.

In [ ]:
```
data AM2Instr
    = PUSH Z
    | ...
    | LABEL ProgramCounter
    | JUMP ProgramCounter
    | JUMPFALSE ProgramCounter
    deriving (Eq, Show)

type AM2Code = [AM2Instr]
```

Uma configuração de AM2 é similar a uma de AM1, acrescida do valor de $\mathrm{pc} : \mathbb{N}$.
Para representar um programa de AM2 juntamente com *program counters*, escolheu-se usar um `Map` entre o valor de `pc` para uma dada instrução, e a instrução em si.

À semelhança do que se fez para AM1,

- À medida que se compila o código de AM2, vão-se atribuindo posições em memória na primeira ocorrência de cada variável
  - o ADT `EnvStateAM2` regista esta informação. Incialmente, o mapeamento tem que ser `Map.empty`, e o próximo endereço `0`.
- À medida que se compila o código de AM2, vão-se atribuindo números inteiros positivos únicos e estritamente crescentes às instruções, começando em `1`.
  - O campo `getNxtPC :: ProgramCounter` contém o valor de PC a ser usado na próxima instrução, se existir.
  - O campo `getInstrs :: AM2AnnotatedProgram` contém o programa à medida que se traduz código `While` para **bytecode** `AM2`.

In [ ]:
```
type AM2Config = (ProgramCounter, AM2Code, Stack, Memory)

type AM2AnnotatedProgram = M.Map ProgramCounter AM2Instr
```

In [ ]:
```haskell
data EnvStateAM2 = EnvSt2 {
    getEnvSt  :: !Env,
    getNxtAdr :: !NextAddr,
    getInstrs :: AM2AnnotatedProgram,
    getNxtPC  :: ProgramCounter
    } deriving (Eq)
```

Tradução de expressões aritméticas e booleanas para "*bytecode*" AM2

`aexpToAM2Code` e `bexpToAM2Code` correspondem a $\mathcal{CA}$ e $\mathcal{CB}$ para a máquina AM2, e como referido na seção sobre AM1, fazem uso da mónade `State` para poderem atualizar o `Env` com novas associações variável $\rightarrow$ endereço.

Veja-se que por se usar a mónade `State`, pode-se utilizar funções como `modify'` (https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-State-Strict.html#v:modify-39-) para ir atualizando a estrutura `EnvStateAM2` com a instrução em que se está, e o seu valor de `pc`.

Manteve-se um excerto do código para ilustrar.

In [ ]:
```haskell
aexpToAM2Code :: Aexp -> St.State EnvStateAM2 AM2Code

bexpToAM2Code :: Bexp -> St.State EnvStateAM2 AM2Code
bexpToAM2Code b = case b of
    ...
    be `Or` be' -> do
        -- Careful with the order with which this is done - whichever is
done first
        -- puts its code on the stack first, so the second operand has to
go first.
        code' <- bexpToAM2Code be'
        code <- bexpToAM2Code be
        St.modify' (\(EnvSt2 environ nxtAdr instrs nxtPC) -> EnvSt2 envir
on nxtAdr (M.insert nxtPC OR instrs) (nxtPC + 1))
        return $ concat [code', code, [OR]]
    ...
```

In [ ]:
```haskell
:ext FlexibleContexts
```

A função `whileToAM2 :: Stm -> (AM2Code, EnvStateAM2)` gera o "*bytecode*" AM2 para um programa `While`, devolvendo também um valor de tipo `EnvStateAM2` que conterá o programa AM2 final anotado com `pc`s.

O código máquina gerado para os comandos `IfThenElse` e `WhileDo` é complexo porque:

- deve primeiro gerar o código dos subcomandos e predicados,
- e só depois colocar as instruções de salto e labels, cujo program counter terá de ser guardado antes da tradução dos subcomandos.

Veja-se a função `incrCounter` usada para obter o contador atual de `EnvStateAM2`, e incrementá-lo na estrutura sem mais nenhuma alteração.
Depois de gerado o código de e.g. predicado e "*branches*" de `IfThenElse` ou do corpo do ciclo de `WhileDo`, pode-se gerar as instruções de controlo `LABEL/JUMPFALSE/JUMP/LABEL` com os contadores obtidos.

In [ ]:
```haskell
whileToAM2 :: Stm -> (AM2Code, EnvStateAM2)
whileToAM2 stm = St.runState (helper stm) (EnvSt2 M.empty 0 M.empty 1)
    where
        incrCounter = do
            EnvSt2 e nA is nxtPC <- St.get
            St.put $ EnvSt2 e nA is $ nxtPC + 1
            return nxtPC

        helper :: Stm -> St.State EnvStateAM2 AM2Code
        helper (var `Assign` aexp) = do
            ...
        helper Skip = do
            ...
        helper (c1 `Comp` c2) = do
            code1 <- helper c1
            code2 <- helper c2
            return $ code1 ++ code2

        helper (IfThenElse b c1 c2) = do
            ...

        helper (WhileDo b c) = do
            boolTestCounter <- incrCounter
            predCode <- bexpToAM2Code b
            jzProgCounter <- incrCounter
            loopCode <- helper c
            jumpCounter <- incrCounter
            afterWhileCounter <- incrCounter
            let whileLabel = LABEL boolTestCounter
                whileJump  = JUMPFALSE afterWhileCounter
                loopJump   = JUMP boolTestCounter
                restLabel  = LABEL afterWhileCounter
            EnvSt2 environ nxtAdr instrs _ <- St.get
            let jumps = M.fromList [(boolTestCounter, whileLabel), (jzPro
gCounter, whileJump), (jumpCounter, loopJump), (afterWhileCounter, restLa
bel)]
            St.put $ EnvSt2 environ nxtAdr (instrs `M.union` jumps) (afte
rWhileCounter + 1)

            return $ [whileLabel] ++ predCode ++ [whileJump] ++ loopCode
++ [loopJump] ++ [restLabel]
```

Como para AM1 com `stepAM1`, a função `stepAM2` faz uma transição -se possível - a partir de uma configuração AM2.
Deixam-se apenas alguns casos para ilustrar o seu funcionamento, e nota-se o seguinte:

- Nas instruções de AM2 que requerem saltos referentes a valores de `pc`, o campo `getInstrs :: AM2AnnotatedProgram` em `EnvStateAM2` permite obter segmentos do "*bytecode*" AM2 ao filtrar o `Map ProgramCounter AM2Instr` pelas chaves que correspondam ao programa a partir de uma certa `label :: ProgramCounter` (ver `initConfigAM2`)

In [ ]:
```haskell
stepAM2 :: AM2Config -> AM2AnnotatedProgram -> AM2Config
stepAM2 conf@(_, [], stack, mem) _ = conf
stepAM2 (pc, c : cs, stack, mem) ann = case c of
    LABEL lab -> (pc', cs, stack, mem)
    JUMP lab -> case M.lookup lab ann of
        Nothing    -> error "JUMP: invalid label!"
        Just instr ->
            let instrs = M.elems $ M.dropWhileAntitone (<= lab) ann
            in (lab, instr : instrs, stack, mem)
    JUMPFALSE lab -> case stack of
        Right b : stack' -> if b
                then (pc', cs, stack', mem)
                else case M.lookup lab ann of
                    Nothing    -> error "JUMPFALSE: invalid label!"
                    Just instr ->
                        let instrs = M.elems $ M.dropWhileAntitone (<= la
b) ann
                        in (lab, instr : instrs, stack', mem)
        _           -> error "JUMPFALSE: invalid stack for operation"

    where
        pc' = pc + 1
```

A função `initConfigAM2` produz, para um programa `While` e um estado inicial:

- a configuração inicial do programa para máquina AM2, acompanhada do
- `Env` resultante da compilação, pelos mesmos motivos que AM1, e
- o mesmo programa AM2, mas anotado com `pc` s.

A configuração inicial de um programa para AM2 precisa vir acompanhada de um Map com a associação entre cada instrução e o seu program counter, porque no caso das instruções de salto em que é possível "regredir" no programa, usar só uma lista para instruções não o permitirá.

A função `runStmInAM2` é a equivalente de `runStmInAM1` em AM2.

In [ ]:
```haskell
initConfigAM2 :: State -> Stm -> (AM2Config, Env, AM2AnnotatedProgram)
initConfigAM2 initSt stm =
    let code :: AM2Code
        envSt :: EnvStateAM2
        (code, envSt) = whileToAM2 stm

        environ = getEnvSt envSt
        annotatedByteCode = getInstrs envSt--M.fromList $ zip (M.keys . g
etInstrs $ envSt) code

        memory :: Memory
        memory = M.fromList [(getEnv environ variable, getSt initSt varia
ble) | variable <- M.keys environ]
    in ((1, code, [], memory), environ, annotatedByteCode)

-- Dado um estado inicial e um comando da linguagem while, simula a sua e
xecução
-- na máquina abstrata AM2.
-- Devolve as variáveis usadas no programa, e os valores que estavam nas
respetivas
-- posições de memória aquando da terminação da execução.
-- Pode não terminar! (Halting problem).
runStmInAM2 :: State -> Stm -> M.Map Var Z
```

In [ ]:
```haskell
runStmInAM2 ex1State expProg
runStmInAM2 ex1State fact
```

```
fromList [("r",9),("x",3),("y",0)]

fromList [("f",720),("n",0)]
```