Licenciatura em Ciências da Computação

Semântica de Linguagens de Programação 2021/2022

A70373 Alexandre Rodrigues Baldé

Conteúdo:

Tipos de dados para a linguagem While

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)
```

In [ ]:
```haskell
type Z = Integer

type Var = String

data Aexp
    = Num Z
    | Var Var
    | Aexp `Plus` Aexp
    | Aexp `Mul` Aexp
    | Aexp `Minus` Aexp
    deriving Eq
```

In [ ]:
```haskell
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
```

In [ ]:
```haskell
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 [ ]:
```haskell
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 [ ]:
```
data Stm
    = Var `Assign` Aexp
    | Skip
    | Stm `Comp` Stm
    | IfThenElse Bexp Stm Stm
    | WhileDo Bexp Stm
    deriving (Eq)
```

Example programs

Retirados da ficha 1.

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)))
```

Natural Semantics

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)]
```

Structural Operational Semantics

In [ ]:
```
-- Uma transição em semântica operacional estrutural:
-- * ou dá origem a um estado (Left)
-- * ou dá origem a um comando intermédio, juntamente com um novo estado
(Right)
stepSOS :: State -> Stm -> Either State (Stm, State)

-- Devolve a transição após o número de passos de execução pedido, se for
possível,
-- numa lista com todas as transições que lhe precederam.
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'
```

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

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

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

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

Abstract Machine 1

In [ ]:

```haskell
-- Mapping from variable names to positions in memory.
-- Used during "compilation" of While code to AM1 bytecode.
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)
```

In [ ]:

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

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]
```

In [ ]:
```haskell
aexpToAM1Code :: EnvStateAM1 -> Aexp -> (AM1Code, EnvStateAM1)
aexpToAM1Code m@(EnvSt e nxtAdr) a = case a of
    Num n -> ([PUSH n], m)
    Var var -> case M.lookup var e of
        Nothing  -> ([GET nxtAdr], EnvSt (M.insert var nxtAdr e) (nxtAdr
+ 1))
        Just adr -> ([GET adr], m)
    ae `Plus` ae' ->
        let (code, m') = aexpToAM1Code m ae
            (code', m'') = aexpToAM1Code m' ae'
        in (concat [code', code, [ADD]], m'')
    ae `Mul` ae' ->
        ...

bexpToAM1Code :: EnvStateAM1 -> Bexp -> (AM1Code, EnvStateAM1)
bexpToAM1Code m@(EnvSt e nxtAdr) b = case b of
    T -> ([TRUE], m)
    F -> ([FALSE], m)
    ae `Eq` ae' ->
        let (code, m') = aexpToAM1Code m ae
            (code', m'') = aexpToAM1Code m' ae'
        in (concat [code', code, [EQUAL]], m'')
    ae `Le` ae' ->
        ...
```

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 Skip = return [NOOP]
        helper (c1 `Comp` c2) = do
            ...
        helper (IfThenElse b c1 c2) = 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]
```

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

-- Mapping from address positions to the values they contain.
-- Should look like:
-- 0 -> n_1
-- 1 -> n_2
-- 2 -> n_3,
-- ...
-- k -> n_k
-- and so on, where n_i are integers.
type Memory = M.Map Z Z

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

In [ ]:
```haskell
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)
```

In [ ]:
```haskell
initConfigAM1 :: State -> Stm -> (AM1Config, Env)
initConfigAM1 initSt stm =
    let code :: AM1Code
        envSt :: EnvStateAM1
        (code, envSt) = whileToAM1 stm

        environ = getEnvSt envSt

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

-- Dado um estado inicial e um comando da linguagem while, simula a sua e
xecução
-- na máquina abstrata AM1.
-- 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
runStmInAM1 initSt stm =
    let (init@(initCode, initStack, initMemory), environ) = initConfigAM1
initSt stm
        run :: AM1Config -> AM1Config
        run !cfg =
            let cfg' = stepAM1 cfg
            in if cfg' == cfg then cfg else run cfg'
        (finalCode, finalStack, finalMemory) = run init

        varsToValues = M.fromList [(var, finalMemory M.! (environ M.! va
r)) | var <- M.keys environ]

    in varsToValues
```

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)]
```

Abstract Machine 2

In [ ]:
```
-- Mapping from variable names to positions in memory.
-- Used during "compilation" of While code to AM1 bytecode.
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]

-- Mapping from address positions to the values they contain.
-- Should look like:
-- 0 -> n_1
-- 1 -> n_2
-- 2 -> n_3,
-- ...
-- k -> n_k
-- and so on, where n_i are integers.
type Memory = M.Map Z Z
```

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

type AM2Code = [AM2Instr]
```

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

type AM2AnnotatedProgram = M.Map ProgramCounter AM2Instr
```

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

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

In [ ]:
```
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 environ nxtAdr (M.insert nxtPC OR instrs) (nxtPC + 1))
        return $ concat [code', code, [OR]]
    ...
```

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

In [ ]:

```
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
            code <- aexpToAM2Code aexp
            EnvSt2 environ nxtAdr instrs nxtPC <- St.get
            case M.lookup var environ of
                Nothing -> do
                    let instr = PUT nxtAdr
                    St.put $ EnvSt2 (M.insert var nxtAdr environ) (nxtAdr
+ 1) (M.insert nxtPC instr instrs) (nxtPC + 1)
                    return $ code ++ [instr]
                Just n  -> do
                    let instr = PUT n
                    St.put $ EnvSt2 environ nxtAdr (M.insert nxtPC instr
instrs) (nxtPC + 1)
                    return $ code ++ [instr]
        helper Skip = do
            ...
        helper (c1 `Comp` c2) = do
            code1 <- helper c1
            code2 <- helper c2
            return $ code1 ++ code2

        -- O código máquina gerado para o comando IfThenElse e para o com
ando WhileDo
        -- é complexo porque deve primeiro gerar o código dos subcomandos
e predicados,
        -- e só depois colocar as instruções de salto e labels - cujo pro
gram counter
        -- terá de ser guardado antes da tradução dos subcomandos.
        --
        -- Ver incrCounter.
        helper (IfThenElse b c1 c2) = do
            predCode <- bexpToAM2Code b
            jzProgCounter <- incrCounter
            thenCode <- helper c1
            elseProgCounter <- incrCounter
            elseCode <- helper c2
            afterIfProgCounter <- incrCounter
            let ifJump      = JUMPFALSE elseProgCounter
                elseLabel   = LABEL elseProgCounter
                jumpToRest  = JUMP afterIfProgCounter
                restLabel   = LABEL afterIfProgCounter
            EnvSt2 environ nxtAdr instrs _ <- St.get
            let jumps = M.fromList [(jzProgCounter, ifJump), (elseProgCou
nter, jumpToRest), (afterIfProgCounter, restLabel)]
            -- Incrementa-se o contador de código devido ao LABEL final,
que apontará
            -- para o código depois do IfThenElse, se existir.
            St.put $ EnvSt2 environ nxtAdr (instrs `M.union` jumps) (afte
```

```
rIfProgCounter + 1)
            return $ predCode ++ [ifJump] ++ thenCode ++ [jumpToRest] ++
elseCode ++ [restLabel]

        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]
```

In [ ]:

```
-- Given an AM2 configuration, execute a single instruction
-- and transition into the next configuration.
stepAM2 :: AM2Config -> AM2AnnotatedProgram -> AM2Config
stepAM2 conf@(_, [], stack, mem) _ = conf
stepAM2 (pc, c : cs, stack, mem) ann = case c of
    PUSH n -> (pc', cs, Left n : stack, mem)
    ADD -> case stack of
        Left z1 : Left z2 : stack' ->
            (pc', cs, Left (z1 + z2) : stack, mem)
        _ -> error "ADD: invalid stack for operation!"
    ...
    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
```

In [ ]:
```haskell
-- 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.
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 . getInstrs $ envSt) code

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

-- Dado um estado inicial e um comando da linguagem while, simula a sua execução
-- 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)]
```