StackIO.hs

1: {- Author: Richard Eisenberg, inspired by Steve Zdancewic
2:    File: Stack.hs
3:    Demonstrates ways of working with a stack, using mutable operations.
4: -}

5:
6: {-# OPTIONS_GHC -W -Wno-unused-imports #-}
7: module Stack where
8: import Control.Monad ( forM_, when )
9: import Data.Vector                        ( Vector )
10: import qualified Data.Vector as I   -- I for immutable
11: import Data.Vector.Mutable                ( IOVector )
12: import qualified Data.Vector.Mutable as M
13: import Data.IORef
14: -- This is based on the last part of Assignment 1.
15: -- In this version, we’ll use a *functional* context. That is, a context
16: -- will be a function mapping strings to ints.
17: type Ctxt = String -> Int
18: emptyCtxt v = error (v ++ " not found")
19: extendCtxt :: Ctxt -> String -> Int -> Ctxt
20:   = \\query -> if query == new_var then new_val else ctxt query
21:   -- This creates a new function that checks if the query matches the new
22:   -- binding. If so, return the new value. Otherwise, look it up in the
23:   -- original context.
24: buildContext :: [(String, Int)] -> Ctxt
25: buildContext []                  = emptyCtxt
26: buildContext ((var, val) : rest) = extendCtxt (buildContext rest) var val
27: data Insn = IPushC Int        -- push an int64 constant onto the stack
28:           | IPushV String     -- push (lookup string ctxt) onto the stack
29:           | IMul              -- multiply the top two values on the stack
30:           | IAdd              -- add the top two values on the stack
31:           | INeg              -- negate the top value on the stack
32: deriving (Eq, Show)
33: type Program = [Insn]
34: newMachine :: Int -> Program -> IO Machine
35: newMachine size prog = do
36:   st <- M.new size
37:   sp <- newIORef 0   -- SP starts at 0
38:   let insns = I.fromList prog
39:   pc <- newIORef 0   -- PC starts at 0
40:   pure (M { stack        = st
StackIO.hs

73: , spRef = sp
74: , instructions = insns
75: , pcRef = pc
76:
77: -- Print to stdout the current machine state (without the context)
78: printMachine :: Machine -> IO ()
79:
80: printMachine (M { stack = st
81: , spRef = sp
82: , instructions = insns
83: , pcRef = pc }) = do
84:   putStrLn "Stack:"
85:   sp_val <- readIORef sp
86:   putStrLn "  SP --> "
87:   forM_ (reverse [0..sp_val-1]) $ \\ stack_loc -> do
88:     stack_val <- M.read st stack_loc
89:     putStrLn ("         " ++ show stack_val)
90:   putStrLn "Instructions:"
91:   pc_val <- readIORef pc
92:   forM_ [0 .. I.length insns - 1] $ \ insn_loc -> do
93:     if (insn_loc == pc_val)
94:       then putStr "  PC --> 
95:       else putStr "         
96:     let insn = insns I.! insn_loc
97:     putStrLn (show insn)
98:
99: -- Run the machine by one step in the given context.
100: -- Returns whether the machine is done running (True means "done")
101: step :: Ctxt -> Machine -> IO Bool
102:
103: step ctxt (M { stack = st
104: , spRef = sp
105: , instructions = insns
106: , pcRef = pc }) = do
107:   -- Fetch the instruction
108:   pc_val <- readIORef pc
109:   let insn = insns I.! pc_val
110:   -- Perform the instruction
111:   case insn of
112:     IPushC n -> push n
113:     IPushV x -> push (ctxt x)
114:     IMul -> do n1 <- pop
115:                n2 <- pop
116:                push (n1 * n2)
117:     IAdd -> do n1 <- pop
118:                n2 <- pop
119:                push (n1 + n2)
120:     INeg -> do n <- pop
121:                push (-n)
122:     -- Increment the PC
123:     let new_pc_val = pc_val + 1
124:     writeIORef pc new_pc_val
125:     pure (new_pc_val == I.length insns)
126:   where
127:     push n = do
128:       sp_val <- readIORef sp
129:       when (sp_val == M.length st) $
130:         error "Out of stack space"
131:       M.write st sp_val n
132:       modifyIORef sp (+1)
133:     pop = do
134:       sp_val <- readIORef sp
135:       when (sp_val == 0) $
136:         error "stack underflow"
let new_sp_val = sp_val - 1
writeIORef sp new_sp_val
M.read st new_sp_val

-- Executing a machine means repeatedly processing instructions
execute :: Ctxt -> Machine -> IO ()
execute ctxt m = do
done <- step ctxt m
when (not done) $
  execute ctxt m

-- Extract the final, sole value from the machine. The stack must have 1 element.
answer :: Machine -> IO Int
answer (M { stack = st
  , spRef = sp }) = do
sp_val <- readIORef sp
when (sp_val /= 1) $
error ("Stack has " ++ show sp_val ++ " values at end of run.")

-- Run a program in a given context for its variables, with a given stack size
run :: Int -> Ctxt -> Program -> IO Int
run size ctxt prog = do
m <- newMachine size prog
execute ctxt m
answer m

-- Example:
p1 = [IPushC 2, IPushC 3, IMul]
answer1 = run 10 emptyCtxt p1
overflow = run 1 emptyCtxt p1