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: extendCtxt ctxt new_var new_val
21:   = \query -> if query == new_var then new_val else ctxt query
22:   -- This creates a new function that checks if the query matches the new
23:   -- binding. If so, return the new value. Otherwise, look it up in the
24:   -- original context.
25: buildContext :: [(String, Int)] -> Ctxt
26: buildContext []                  = emptyCtxt
27: buildContext ((var, val) : rest) = extendCtxt (buildContext rest) var val
28: data Insn
29:   = IPushC Int        -- push an int64 constant onto the stack
30:   | IPushV String     -- push (lookup string ctxt) onto the stack
31:   | IMul              -- multiply the top two values on the stack
32:   | IAdd              -- add the top two values on the stack
33:   | INeg              -- negate the top value on the stack
34: deriving (Eq, Show)
35: type Program = [Insn]
36: newMachine :: Int -> Program -> IO Machine
37: newMachine size prog = do
38:   st <- M.new size
39:   sp <- newIORef 0   -- SP starts at 0
40:   let insns = I.fromList prog
41:   pc <- newIORef 0   -- PC starts at 0
42:   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: printMachine (M { stack = st
80: , spRef = sp
81: , instructions = insns
82: , pcRef = pc }) = do
83:
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: putStrLn "
93: putStrLn "PC --> 
94: forM_ [0 .. I.length insns - 1] $ \ insn_loc -> do
95: let insn = insns I.! insn_loc
96: putStrLn (show insn)
97: putStrLn "Run the machine by one step in the given context.
98: -- Returns whether the machine is done running (True means "done")
99: step :: Ctxt -> Machine -> IO Bool
100: step ctxt (M { stack = st
101: , spRef = sp
102: , instructions = insns
103: , pcRef = pc }) = do
104: -- Fetch the instruction
105: pc_val <- readIORef pc
106: let insn = insns I.! pc_val
107: case insn of
108:   IPushC n -> push n
109:   IPushV x -> push (ctxt x)
110:   IMul -> do n1 <- pop
111:             n2 <- pop
112:             push (n1 * n2)
113:   IAdd -> do n1 <- pop
114:             n2 <- pop
115:             push (n1 + n2)
116:   INeg -> do n <- pop
117:             push (-n)
118:   -- Increment the PC
119:   let new_pc_val = pc_val + 1
120:   writeIORef pc new_pc_val
121:   pure (new_pc_val == I.length insns)
122: case insn of
123:   IPushC n -> push n
124:   IPushV x -> push (ctxt x)
125:   IMul -> do n1 <- pop
126:             n2 <- pop
127:             push (n1 * n2)
128:   IAdd -> do n1 <- pop
129:             n2 <- pop
130:             push (n1 + n2)
131:   INeg -> do n <- pop
132:             push (-n)
133:   -- Increment the PC
134:   let new_pc_val = pc_val + 1
135:   writeIORef pc new_pc_val
136:   pure (new_pc_val == I.length insns)
137: where
138:   push n = do
139:     sp_val <- readIORef sp
140:     when (sp_val == M.length st) $ 
141:       M.write st sp_val n
142:     modifyIORef sp (+1)
143:     pop = do
144:       sp_val <- readIORef sp
145:       when (sp_val == 0) $ 
146:         error "stack underflow"
StackIO.hs

145: let new_sp_val = sp_val - 1
146: writeIORef sp new_sp_val
147: M.read st new_sp_val
148: -- Executing a machine means repeatedly processing instructions
149: execute :: Ctxt -> Machine -> IO ()
150: execute ctxt m = do
done <- step ctxt m
when (not done) $
execute ctxt m

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

165: M.read st 0
166: run :: Int -> Ctxt -> Program -> IO Int
167: run size ctxt prog = do
m <- newMachine size prog
execute ctxt m
answer m

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