StackIO.hs

1: {- Author: Richard Eisenberg, inspired by Steve Zdancewic
2: File: Stack.hs
3:
4: Demonstrates ways of working with a stack, using mutable operations.
5: -}
6:
7: {-# OPTIONS_GHC -W -Wno-unused-imports #-}
8: module Stack where
9:
10: import Control.Monad ( forM_, when )
11:
12: import Data.Vector                        ( Vector )
13: import qualified Data.Vector as I   -- I for immutable
14: import Data.Vector.Mutable                ( IOVector )
15: import qualified Data.Vector.Mutable as M
16: import Data.IORef
17:
18: -- This is based on the last part of Assignment 1.
19:
20: -- In this version, we'll use a *functional* context. That is, a context
21: -- will be a function mapping strings to ints.
22: type Ctxt = String -> Int
23:
24: -- An empty context maps all strings to error
25: emptyCtxt :: Ctxt
26: emptyCtxt v = error (v ++ " not found")
27:
28: -- Extend a context with a new binding
29: extendCtxt :: Ctxt -> String -> Int -> Ctxt
30: extendCtxt ctxt new_var new_val
31:   = \query -> if query == new_var then new_val else ctxt query
32:   -- This creates a new function that checks if the query matches the new
33:   -- binding. If so, return the new value. Otherwise, look it up in the
34:   -- original context.
35:
36: -- Build a context from a list of (String, Int) pairs.
37: buildContext :: [(String, Int)] -> Ctxt
38: buildContext []                  = emptyCtxt
39: buildContext ((var, val) : rest) = extendCtxt (buildContext rest) var val
40:
41: -- One instruction in our stack machine
42: data Insn
43:   = IPushC Int        -- push an int64 constant onto the stack
44:   | IPushV String     -- push (lookup string ctxt) onto the stack
45:   | IMul              -- multiply the top two values on the stack
46:   | IAdd              -- add the top two values on the stack
47:   | INeg              -- negate the top value on the stack
48: deriving (Eq, Show)
49:
50: -- A stack program is just a list of instructions.
51: type Program = [Insn]
52:
53: -- A stack machine has a data space for the stack, as well as the index of one
54: -- past the top of the stack (the bottom is always 0), the instructions,
55: -- and the program counter (PC), which tells us which instruction to run
56: -- next.
57: data Machine = M { stack        :: IOVector Int
58:                  , spRef        :: IORef Int
59:                  , instructions :: Vector Insn
60:                  , pcRef        :: IORef Int }
61:
62: -- Create a new machine of the given size, ready to execute a given program
63: -- in the given context.
64: newMachine :: Int -> Program -> IO Machine
65: newMachine size prog = do
66:   st <- M.new size
67:   sp <- newIORef 0   -- SP starts at 0
68:   let insns = I.fromList prog
69:   pc <- newIORef 0   -- PC starts at 0
70:   pure (M { stack = st
71:   })
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:   putStrLn "Stack:"
84:   sp_val <- readIORef sp
85:   putStrLn "  SP --> "
86:   forM_ (reverse [0..sp_val-1]) $ \ stack_loc -> do
87:     stack_val <- M.read st stack_loc
88:     putStrLn ("         " ++ show stack_val)
89:   putStrLn "Instructions:"
90:   pc_val <- readIORef pc
91:   forM_ [0 .. I.length insns - 1] $ \ insn_loc -> do
92:     if (insn_loc == pc_val)
93:       then putStr "  PC --> "
94:       else putStr "         
95:     let insn = insns I.! insn_loc
96:     putStrLn (show insn)
97:   putStrLn ""
98:   putStrLn ""
99:   putStrLn ""
100:   putStrLn ""
101:   putStrLn ""
102: -- Run the machine by one step in the given context.
103: -- Returns whether the machine is done running (True means "done")
104: step :: Ctxt -> Machine -> IO Bool
105: step ctxt (M { stack = st
106:                , spRef = sp
107:                , instructions = insns
108:                , pcRef = pc }) = do
109:   -- Fetch the instruction
110:   pc_val <- readIORef pc
111:   let insn = insns I.! pc_val
112:   case insn of
113:     IPushC n -> push n
114:     IPushV x -> push (ctxt x)
115:     IMul -> do n1 <- pop
116:             n2 <- pop
117:             push (n1 * n2)
118:     IAdd -> do n1 <- pop
119:             n2 <- pop
120:             push (n1 + n2)
121:     INeg -> do n <- pop
122:             push (-n)
123:   -- Increment the PC
124:   let new_pc_val = pc_val + 1
125:   writeIORef pc new_pc_val
126:   pure (new_pc_val == I.length insns)
127: where
128:   push n = do
129:     sp_val <- readIORef sp
130:     when (sp_val == M.length st) $ 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) $ error "stack underflow"
StackIO.hs

145:
146:   let new_sp_val = sp_val - 1
147:   writeIORef sp new_sp_val
148:   M.read st new_sp_val
149:
150:   -- Executing a machine means repeatedly processing instructions
151: execute :: Ctxt -> Machine -> IO ()
152: execute ctxt m = do
153:   done <- step ctxt m
154:   when (not done) $*
155:     execute ctxt m
156:
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
161:   sp_val <- readIORef sp
162:   when (sp_val /= 1) $*
163:     error (*Stack has " ++ show sp_val ++ " values at end of run."*)
164:
165:   M.read st 0
166:
167:   -- Run a program in a given context for its variables, with a given stack size
168: run :: Int -> Ctxt -> Program -> IO Int
169: run size ctxt prog = do
170:   m <- newMachine size prog
171:   execute ctxt m
172:   answer m
173:
174:   -- Example:
175: p1       = [IPushC 2, IPushC 3, IMul]
176: answer1  = run 10 emptyCtxt p1
177: overflow = run 1 emptyCtxt p1
178: