IR3.hs

1: {- Author: Steve Zdancewic, translated by Richard Eisenberg
2: File: IR3.hs
3: Intermediate representation for an expression language with commands,
4: conditionals, and loops.
5: -}
6: {-# OPTIONS_GHC -W #-}
7: module IR3 where
8: import Data.Int     ( Int16 )
9: import CS350.Unique
10: import CS350.Renderable  ( Renderable(..) )
11: import HERA.Assembly     ( AssemblyProgram, AInstruction(..) )
12: import HERA.Base         ( Register(..), Condition(None, Nz), rt )
13: import Data.List    ( intercalate, nub )
14: import Data.Maybe   ( isNothing )
15: import Text.Printf  ( printf )
16: -- source language ----------------------------------------------------------
17: -- This variant of the language treats variables as mutable.
18: type VarName = String
19: -- Arithmetic expressions
20: data Exp
21:   = VarE VarName         -- string representing an object-language variable
22:   | ConstE Int16         -- a constant Int16 value
23:   | AddE Exp Exp         -- sum of two expressions
24:   | MulE Exp Exp         -- product of two expressions
25:   | NegE Exp             -- negation of an expression
26:   deriving (Show, Eq)
27: -- Abstract syntax of commands
28: data Command
29:   = Skip                       -- skip
30:   | Assign VarName Exp         -- X := e
31:   | Seq Command Command        -- c1 ; c2
32:   | IfNZ Exp Command Command   -- if (e /= 0) then cmd1 else cmd2
33:   | WhileNZ Exp Command        -- while (e /= 0) do cmd
34:   deriving (Show, Eq)
35: exampleBranch :: Command
36: exampleBranch =
37:   let x1 = "X1"
38:       x2 = "X2"
39:       vx1 = VarE x1
40:       vx2 = VarE x2
41: in
42: Seq (Assign x1 (AddE vx1 vx2))
43:   (Seq (IfNZ vx2
44:          (Assign x1 (AddE vx1 (ConstE 1)))
45:            (Assign x2 vx1))
46:            (Assign x2 (MulE vx2 vx1)))
47: exampleBranch =
```
IR3.hs
73: WhileNZ X1 DO
74:   X2 := X2 * X1;
75:   X1 := X1 + (-1);
76: DONE
77: {-
78: factorial :: Command
79: factorial =
80:   let x = "X1"
81:       ans = "X2"
82:   in
83:   Seq (Assign x (ConstE 6))
84:   (Seq (Assign ans (ConstE 1))
85:      (WhileNZ (VarE x)
86:         (Seq (Assign ans (MulE (VarE ans) (VarE x)))
87:            (Assign x   (AddE (VarE x) (ConstE (-1)))))
88:     ))
89: -------------------------------------
90: -- Intermediate representation
91: -- operands
92: type Label = Unique
93: -- operators
94: data Operand
95:   = Id Unique
96:   | Const Int16
97: deriving (Show, Eq)
98: -- binary operations
99: data BinaryOp
100:   = Add
101:   | Mul
102: deriving (Show, Eq)
103: -- comparisons
104: data CmpOp
105:   = Equal
106:   | LessThan
107: deriving (Show, Eq)
108: -- instructions
109: data Instruction
110:   = Let Unique BinaryOp Operand Operand
111:   | Load Unique VarName
112:   | Store VarName Operand
113:   | Compare Unique CmpOp Operand Operand
114: deriving (Show, Eq)
115: -- Block terminator
116: data Terminator
117:   = Return
118:   | Branch Label -- unconditional branch
119:   | CondBr Operand Label Label -- conditional branch
120: -- Basic blocks
121: data Block = Bl { instructions :: [Instruction]
122:                 , terminator   :: Terminator
123:                 }
124: -- Control flow graph: a pair of an entry block and a set of labeled blocks
125: data Program = Cfg Block [(Label, Block)]
126: -------------------------------
127: -- Pretty-printing
128: instance Renderable Operand where
129:   render (Id u)    = render u
130:   render (Const c) = render c
131: instance Renderable BinaryOp where
132:   render Add = "add"
133:   render Mul = "mul"
```
instance Renderable CmpOp where
  render Equal = "equal"
  render LessThan = "lessthan"

instance Renderable Instruction where
  render (Let u bop op1 op2) = printf "let %s = %s %s %s"
    (render u) (render bop) (render op1) (render op2)
  render (Load u x) = printf "let %s = load %s"
    (render u) ("var" ++ x)
  render (Store x op) = printf "let _ = store %s %s %s"
    (render op) ("var" ++ x)
  render (Compare u cmpop op1 op2) = printf "let %s = icmp %s %s %s"
    (render u) (render cmpop) (render op1) (render op2)

instance Renderable Terminator where
  render Return                = "  ret ()"
  render (Branch lbl)          = printf "  br %s" (render lbl)
  render (CondBr op lbl1 lbl2) = printf "  cbr %s %s %s"
    (render op) (render lbl1) (render lbl2)

instance Renderable Block where
  render (Bl { instructions = insns, terminator = term }) =
    intercalate " in
" (map render insns) ++
    (if length insns > 0 then " in\n" else ")"]
  render term

instance Renderable Program where
  render (Cfg entry blocks) =
    printf "let program () = 
%s
in entry ()" $
    printf "let rec entry () =
%s" (render entry) ++ "$\n"
  intercalate "$\n"
    (map ((\(lbl, block) ->
      printf "and %s () = 
%s" (render lbl) (render block))) blocks)

-- Compilation

data Element
  = L Label
  | I Instruction
  | T Terminator

| A stream is a sequence of elements *in reverse order*
| data Stream = MkStr [Element]

{- During generation, we typically emit code so that it is in
  _reverse_ order when the stream is viewed as a list. That is,
  instructions closer to the head of the list are to be executed
  later in the program. That is because cons is more efficient than
  append.

  To help make code generation easier, we define snoc (reverse cons)
  and reverse append, which let us write code sequences in their
  natural order.

--}
IR3.hs

216: MkStr x <> y = MkStr (y : x)
217: -}
218:
219: -- Turn a single instruction into a stream
220: inst :: Instruction -> Stream
221: inst = MkStr . (:][) . I
222:
223: -- Turn a terminator into a stream
224: term :: Terminator -> Stream
225: term = MkStr . (:][) . T
226:
227: -- Turn a label into a stream
228: label :: Label -> Stream
229: label = MkStr . (:][) . L
230:
231: -- An empty stream
232: emptyStream = MkStr []
233:
234: {- Convert an instruction stream into a control flow graph.
235: Assumes that the instructions are in 'reverse' order of execution.
236: -}
237: buildCfg :: Stream -> Program
238: buildCfg (MkStr code) =
239:   let (insns, m_term, blks) = go [] Nothing [] code in
240:     case m_term of
241:       Nothing -> error "buildCfg: entry block has no terminator"
242:     Just term -> Cfg (Bl { instructions = insns, terminator = term }) blks
243:   where
244:     go insns m_term blks [] = (insns, m_term, blks)
245:     go insns m_term blks (L l : rest)
246:       | Just term <- m_term
247:       = go [] Nothing ((l, Bl { instructions = insns, terminator = term }) : blks) rest
248:       | null insns  -- this happens for a dummy (redundant) label
249:       = go [] Nothing blks rest
250:       | otherwise
251:       = error (printf "buildCfg: block labeled %s has no terminator" (render l))
252:     go insns m_term blks (T t : rest)
253:       | null insns && isNothing m_term
254:       = go [] (Just t) blks rest
255:       | otherwise
256:       = error (printf "buildCfg: block is missing label: %s" (unlines (map render insns)))
257:     go insns m_term blks (I i : rest)
258:       = go (i : insns) m_term blks rest
259:     go insns m_term blks (L 1 : rest)
260:       = go (i : insns) m_term blks rest
261:     compileBop :: BinaryOp -> Exp -> Exp -> UniqueM (Stream, Operand)
262:     compileBop bop e1 e2 = do
263:       (ins1, ret1) <- compileExp e1
264:       (ins2, ret2) <- compileExp e2
265:       ret <- newUnique "tmp"
266:       pure (ins1 <+> ins2 <+> inst (Let ret bop ret1 ret2), Id ret)
267:     compileExp :: Exp -> UniqueM (Stream, Operand)
268:     compileExp (VarE x) = do ret <- newUnique "tmp"
269:       pure (inst (Load ret x), Id ret)
270:     compileExp (ConstE c) = pure (emptyStream, Const c)
271:     compileExp (AddE e1 e2) = compileBop Add e1 e2
272:     compileExp (MulE e1 e2) = compileBop Mul e1 e2
273:     compileExp (NegE e1) = compileBop Mul e1 (ConstE (-1))
IR3.hs

288: (is, result) <- compileExp e
289: c1_insns <- compileCmd c1
290: c2_insns <- compileCmd c2
291: guard <- newUnique "guard"
292: nz_branch <- newUnique "nz"
293: z_branch <- newUnique "z"
294: merge <- newUnique "merge"
295: pure (is +++
296:   -- Compute the guard result
297:   inst (Compare guard Equal result (Const 0)) +++
298:   term (CondBr (Id guard) z_branch nz_branch) +++
299:   -- guard is non-zero
300:   label nz_branch +++
301:   c1_insns +++
302:   term (Branch merge) +++
303:   -- guard is zero
304:   label z_branch +++
305:   c2_insns +++
306:   term (Branch merge) +++
307:   label merge)
308: compileCmd (WhileNZ e c) = do
309:   (is, result) <- compileExp e
310:   c_insns <- compileCmd c
311:   guard <- newUnique "guard"
312:   entry <- newUnique "entry"
313:   body  <- newUnique "body"
314:   exit  <- newUnique "exit"
315:   pure (term (Branch entry) +++
316:     label entry +++
317:     is +++
318:     inst (Compare guard Equal result (Const 0)) +++
319:     term (CondBr (Id guard) exit body) +++
320:     label body +++
321:     c_insns +++
322:     term (Branch entry) +++
323:     label exit)
324: compile :: Command -> UniqueM Program
325: compile cmd = do
326:   str <- compileCmd cmd
327:   pure (buildCfg (str +++ term Return))
328: compileProgram :: Program -> AssemblyProgram
329: compileProgram (Cfg entry labeled_blocks)
330:   = data_insns ++
331:     compileBlock entry ++
332:     concatMap compileLabeledBlock labeled_blocks
333:   where
334:     all_variables = collectVariables (entry : map snd labeled_blocks)
335:     data_insns = concatMap mk_data_insns all_variables
336:     mk_data_insns var_name = [ ADlabel var_name
337:                              , AInteger 0 ]
IR3.hs

360: compileLabeledBlock :: (Label, Block) -> AssemblyProgram
361: compileLabeledBlock (lbl, block)
362: = [ALabel (uniqueString lbl)] ++
363:     compileBlock block
364:
365: compileBlock :: Block -> AssemblyProgram
366: compileBlock (Bl { instructions = insns, terminator = term })
367: = concatMap compileInstruction insns ++
368:     compileTerminator term
369:
370: compileInstruction :: Instruction -> AssemblyProgram
371: compileInstruction (Let u bop op1 op2)
372: = getOperand R1 op1 ++
373:     getOperand R2 op2 ++
374:     bopInstruction bop R1 R1 R2 ++
375:     [ ASet R2 (fromIntegral $ uniqueNumber u)
376:     , AStore R1 0 R2 ]
377: compileInstruction (Load u var_name)
378: = [ ASetl R1 var_name
379:     , ALoad R1 0 R1
380:     , ASet R2 (fromIntegral $ uniqueNumber u)
381:     , AStore R1 0 R2 ]
382: compileInstruction (Store var_name op)
383: = getOperand R1 op ++
384:     [ ASet1 R2 var_name
385:     , AStore R1 0 R2 ]
386: compileInstruction (Compare u cmpop op1 op2)
387: = getOperand R1 op1 ++
388:     getOperand R2 op2 ++
389:     cmpInstruction cmpop R1 R1 R2 ++
390:     [ ASet R2 (fromIntegral $ uniqueNumber u)
391:     , AStore R1 0 R2 ]
392: compileTerminator :: Terminator -> AssemblyProgram
393: compileTerminator Return = [AHalt]
394: compileTerminator (Branch lbl) = [ABr None (uniqueString lbl)]
395: compileTerminator (CondBr op lbl1 lbl2)
396: = getOperand R1 op ++
397:     [ AFlags R1
398:     , ABr Nz (uniqueString lbl1)
399:     , ABr None (uniqueString lbl2) ]
400: getOperand :: Register -> Operand -> AssemblyProgram
401: getOperand reg (Id u) = [ ASet rt (fromIntegral $ uniqueNumber u)
402:                         , ALoad reg 0 rt ]
403: getOperand reg (Const i) = [ ASet reg (fromIntegral i) ]
404: bopInstruction :: BinaryOp -> Register -> Register -> Register -> AssemblyProgram
405: bopInstruction Add rd ra rb = [AAdd rd ra rb]
406: bopInstruction Mul rd ra rb = [AMul rd ra rb]
407: cmpInstruction :: CmpOp -> Register -> Register -> Register -> AssemblyProgram
408: cmpInstruction Equal    rd ra rb = [ ACmp ra rb
409:                                    , ASavef rd
410:                                    , ALsr rd rd
411:                                    , ASetlo rt 1
412:                                    , AAnd rd rt rd ]
413: cmpInstruction LessThan rd ra rb = [ ACmp ra rb
414:                                    , ASavef rd
415:                                    , ASetlo rt 1
416:                                    , AAnd rd rt rd ]
417: collectVariables :: [Block] -> [String]
418: collectVariables blocks = nub $ concatMap collect1 blocks
419:   where
420:     collect1 (Bl { instructions = insns }) = concatMap collect_insn insns
421:     collect_insn (Load _ var_name)  = [var_name]
422:     collect_insn (Store var_name _) = [var_name]
423:     collect_insn _                  = []