Intermediate representation for an expression language with commands, conditionals, and loops.
IR3.hs

```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: -- Intermediate representation
90: 91: type Label = Unique
92: 93: -- operands
94: 95: data Operand
96:   = Id Unique
97:   | Const Int16
98:   deriving (Show, Eq)
99: 100: -- binary operations
101: 102: data BinaryOp
103:   = Add
104:   | Mul
105:   deriving (Show, Eq)
106: 107: -- comparisons
108: 109: data CmpOp
110:   = Equal
111:   | LessThan
112:   deriving (Show, Eq)
113: 114: -- instructions
115: 116: data Instruction
117:   = Let Unique BinaryOp Operand Operand
118:   | Load Unique VarName
119:   | Store VarName Operand
120:   | Compare Unique CmpOp Operand Operand
121:   deriving (Show, Eq)
122: 123: -- Block terminator
124: 125: data Terminator
126:   = Return
127:   | Branch Label -- unconditional branch
128:   | CondBr Operand Label Label -- conditional branch
129: 130: -- Basic blocks
131: 132: data Block = Bl { instructions :: [Instruction]
133:                       , terminator   :: Terminator
134:                      }
135: 136: -- Control flow graph: a pair of an entry block and a set of labeled blocks
137: 138: data Program = Cfg Block [(Label, Block)]
139: 140: -------------------------------
141: -- Pretty-printing
142: instance Renderable Operand where
143:   render (Id u)    = render u
144:   render (Const c) = render c
145: 146: instance Renderable BinaryOp where
147:   render Add = "add"
148:   render Mul = "mul"
```

instance Renderable CmpOp where
  render Equal = "equal"
  render LessThan = "lessthan"

instance Renderable Instruction where
  render (Let { u = u, bop = bop, op1 = op1, op2 = op2 }) =
    printf "let %s = %s %s %s"
      (render u) (render bop) (render op1) (render op2)
  render (Load { u = u, x = x }) =
    printf "let %s = load %s"
      (render u) ("var" ++ x)
  render (Store { x = x, op = op }) =
    printf "let _ = store %s %s"
      (render op) ("var" ++ x)
  render (Compare { u = u, cmpop = cmpop, op1 = op1, op2 = 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 = lbl }) = printf "  br %s" (render lbl)
  render (CondBr { op = op, lbl1 = lbl1, lbl2 = lbl2 }) = printf "  cbr %s %s %s"
      (render op) (render lbl1) (render lbl2)

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

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

-- 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.

-- append in reverse order
(<++>) :: Stream -- this one logically comes first
    -> Stream -- this one logically comes second
    -> Stream
MkStr x <++> MkStr y = MkStr (y ++ x)

{-
| I Instruction
| T Terminator

|
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:
246:     go insns m_term blks (L l : rest)
247:       | Just term <- m_term
248:       = go [] Nothing ((L l) { instructions = insns, terminator = term }) : blks) rest
249:       | null insns -- this happens for a dummy (redundant) label
250:       = go [] Nothing blks rest
251:       | otherwise
252:       = error (printf "buildCfg: block labeled %s has no terminator" (render l))
253:     go insns m_term blks (T t : rest)
254:       | null insns && isNothing m_term
255:       = go [] (Just t) blks rest
256:       | otherwise
257:       = error (printf "buildCfg: block is missing label: %s" (unlines (map render insns)))
258:     go insns m_term blks (I i : rest)
259:       = go (i : insns) m_term blks rest
260: compileBop :: BinaryOp -> Exp -> Exp -> UniqueM (Stream, Operand)
261: compileBop bop e1 e2 = do
262:   (is1, ret1) <- compileExp e1
263:   (is2, ret2) <- compileExp e2
264:   ret <- newUnique "tmp"
265:   pure (ins1 <+> ins2 <+> inst (Let ret bop ret1 ret2), Id ret)
266: compileExp :: Exp -> UniqueM (Stream, Operand)
267: compileExp (VarE x) = do ret <- newUnique "tmp"
268:   pure (inst (Load ret x), Id ret)
269: compileExp (ConstE c) = pure (emptyStream, Const c)
270: compileExp (AddE e1 e2) = compileBop Add e1 e2
271: compileExp (MulE e1 e2) = compileBop Mul e1 e2
272: compileExp (NegE e1) = compileBop Mul e1 (ConstE (-1))
273: compileCmd :: Command -> UniqueM Stream
274: compileCmd Skip = pure emptyStream
275: compileCmd (Assign v e) = do
276:   (is, op) <- compileExp e
277:   pure (is <+> inst (Store v op))
278: compileCmd (Seq c1 c2) = do
279:   str1 <- compileCmd c1
280:   str2 <- compileCmd c2
281:   pure (str1 <+> str2)
282: compileCmd (IfNZ e c1 c2) = do
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:
296: pure (is <++]
297: -- Compute the guard result
298: inst (Compare guard Equal result (Const 0)) <++>
299: term (CondBr (Id guard) z_branch nz_branch) <++>
300: 301: -- guard is non-zero
302: label nz_branch <++>
303: c1_insns <++>
304: term (Branch merge) <++>
305: 306: -- guard is zero
307: label z_branch <++>
308: c2_insns <++>
309: term (Branch merge) <++>
310: 311: label merge)
312:
313: compileCmd (WhileNZ e c) = do
314: (is, result) <- compileExp e
315: c_insns <- compileCmd c
316: guard <- newUnique "guard"
317: entry <- newUnique "entry"
318: body <- newUnique "body"
319: exit <- newUnique "exit"
320:
321: pure (term (Branch entry) <++>
322: label entry <++>
323: is <++>
324: inst (Compare guard Equal result (Const 0)) <++>
325: term (CondBr (Id guard) exit body) <++>
326: label body <++>
327: c_insns <++>
328: term (Branch entry) <++>
329: label exit)
330:
331: compile :: Command -> UniqueM Program
332: compile cmd = do
333: str <- compileCmd cmd
334: pure (buildCfg (str <++> term Return))
335:
336: --------------------------------------------------------
337: -- Compilation to HERA
338:
339: {- Strategy:
340: - each named variable becomes a named data location in the data segment
341: - each temporary corresponds to a memory location, where the temporary number
342: is the address of the memory location.
343: - No function calls here, so we don’t have to worry about interactions with, e.g.,
344: the function stack.
345: -}
346: -}
347: 348: compileProgram :: Program -> AssemblyProgram
349: compileProgram (Cfg entry labeled_blocks)
350: = data_insns ++
351: compileBlock entry ++
352: concatMap compileLabeledBlock labeled_blocks
353: where
354: all_variables = collectVariables (entry : map snd labeled_blocks)
355: data_insns = concatMap mk_data_insns all_variables
356:
357: mk_data_insns var_name = [ ADlabel var_name
358:       , AInteger 0 ]
359:
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:   = [ ASel R1 var_name
379:     , ALoad R1 0 R1
380:     , AStore R1 0 R2 ]
381: compileInstruction (Store var_name op)
382:   = getOperand R1 op1 ++
383:     getOperand R2 op2 ++
384:     bopInstruction bop R1 R1 R2 ++
385:     [ ASet R2 (fromIntegral $ uniqueNumber u)
386:     , AStore R1 0 R2 ]
387: compileInstruction (Compare u cmpop op1 op2)
388:   = getOperand R1 op1 ++
389:     getOperand R2 op2 ++
390:     cmpInstruction cmpop R1 R1 R2 ++
391:     [ ASet R2 (fromIntegral $ uniqueNumber u)
392:     , AStore R1 0 R2 ]
393:
394: compileTerminator :: Terminator -> AssemblyProgram
395: compileTerminator Return = [AHalt]
396: compileTerminator (Branch lbl) = [ABr None (uniqueString lbl)]
397: compileTerminator (CondBr op lbl1 lbl2)
398:   = getOperand R1 op ++
399:     [ AFlags R1
400:     , ABr Nz (uniqueString lbl1)
401:     , ABr None (uniqueString lbl2) ]
402:
403: getOperand :: Register -> Operand -> AssemblyProgram
404: getOperand reg (Id u) = [ ASet rt (fromIntegral $ uniqueNumber u)
405:   , ALoad reg 0 rt ]
406: getOperand reg (Const i) = [ ASet reg (fromIntegral i) ]
407:
408: bopInstruction :: BinaryOp -> Register -> Register -> Register -> AssemblyProgram
409: bopInstruction Add rd ra rb = [AAdd rd ra rb]
410: bopInstruction Mul rd ra rb = [AMul rd ra rb]
411:
412: cmpInstruction :: CmpOp -> Register -> Register -> Register -> AssemblyProgram
413: cmpInstruction Equal  rd ra rb = [ ACmp ra rb
414:   , ASavef rd
415:   , ASr rd rd
416:   , ASetlo rt 1
417:   , AAnd rd rt rd ]
418: cmpInstruction LessThan rd ra rb = [ ACmp ra rb
419:   , ASavef rd
420:   , ASetlo rt 1
421:   , AAnd rd rt rd ]
422:
423: collectVariables :: [Block] -> [String]
424: collectVariables blocks = nub $ concatMap collect1 blocks
425:   where
426:     collect1 (Bl { instructions = insns }) = concatMap collect_insn insns
427:     collect_insn (Load _ var_name)  = [var_name]
428:     collect_insn (Store var_name _) = [var_name]
429:     collect_insn _                 = []