最終レポート

学籍番号 1029-25-7819　　名前 五十嵐　雄

1　はじめに

本実験の実装は Haskell で行っている。不明な点があれば以下に挙げる資料を中心に参照せよ。

- https://www.haskell.org/documentation (リファレンス集)
- https://www.haskell.org/hoogle/ (Haskell 用の検索エンジン)

ディレクトリのトップで以下のコマンドを実行することにより，他の環境に影響を与えることなくインストールすることができる．

```
$ cabal sandbox init
$ cabal install
```

インストールしたバイナリは .cabal-sandbox/bin/HaSC である．scc で実行する場合はそのファイルを参照せよ．

2　課題 11

2.1　はじめに

本課題では，課題 10 で生成した解析済みの抽象構文木 A_Program を，中間表現 ICode の列に変換する処理を実装した．資料で提示されている中間表現は各種最適化を適用するには少し水準が高すぎるので，今後の拡張のことも考えて独自のものを採用した．

2.2　ICode のデータ構造

独自に定義した中間表現は以下のようである．特に説明が必要と思われるものにはコメントをつけている．

```
-- src/HaSC/Prim/IntermdSyntax.hs
type IProgram = [IDecl]
type IVar     = ObjInfo
```

```
type Label = String

data IDecl = IVarDecl IVar
             -- グローバルな変数宣言
           | IFunDecl IVar [IVar] [ICode]
             -- 関数宣言（本体の情報，引数，本体）の順である
             deriving(Show, Eq, Ord)

data ICode = ILabel     Label
           | ILet       IVar IVar
           | ILi        IVar Integer
           | IAop       String IVar IVar IVar
           | IRelop     String IVar IVar IVar
           | IWrite     IVar IVar
           | IRead      IVar IVar
           | IAddr      IVar IVar
           | IJumpTr    IVar Label
             -- IVar が 非 0 ならば指定したラベルにジャンプする
           | IJumpFls   IVar Label
             -- IVar が 0 ならば指定したラベルにジャンプする
           | IJump      Label
           | ICall      IVar IVar [IVar]
           | IReturn    IVar
           | IRetVoid
           | IPrint     IVar
             deriving(Show, Eq, Ord)
```

2.3 変換処理

2.3.1 一時変数

抽象構文木を中間表現に変換する過程で一時変数が必要となるので，以下のような処理を実装している．State モナドの中で freshVar とすることで @hoge という名前を持った IVar が取得できるようになっており，また，変数の数を状態として持つことで重複なく生成できる．さらに，独立した複文では重複した一時変数を使っても良いので，複文を出るときに collectUnuseVar を使って明示的に変数を返却できるようにした．返却された変数は Reuseable としてスタックに積まれ優先的に使用される．

```
-- src/HaSC/Prim/ASTtoIntermed.hs
type IREnv = S.StateT (Reuseable, VarNum) (State LabelNum)
type VarNum    = Int
type LabelNum  = Int
type Reuseable = [VarNum]

freshVar :: IREnv IVar
freshVar = do
  (reuse, num) <- S.get
  newnum <- case reuse of
              []     -> do S.put ([], num+1)
                           return num
              (n:ns) -> do S.put (ns, num)
                           return n
  return $ makeVar newnum
    where makeVar n = ObjInfo (("@" ++) . show $ n) Var CTemp (-1)

freshVars :: Int -> IREnv [IVar]
freshVars n = replicateM n freshVar

resetVars :: IREnv ()
resetVars = S.put ([], 0)

collectUnuseVar :: IVar -> IREnv ()
collectUnuseVar (ObjInfo ('@':varNum) _ _ _)
    = do (reuse, num) <- S.get
         when (not $ (read varNum) `elem` reuse) (put (sort $ (read varNum):reuse, num))
collectUnuseVar _ = return ()


collectUnuseVars :: [IVar] -> IREnv ()
collectUnuseVars = mapM_ collectUnuseVar
```

2.3.2 ラベル

中間表現に変換する過程で処理をジャンプさせるためのラベルが必要になるので，ラベルを生成する機構を実装している．実装は State モナドを使った簡単なものなので，ここでは freshLabel で重

複のないラベルが得られるということだけを述べておく．

2.3.3 実際の変換処理

以下に自明ではない変換を行っている部分の説明を記述する．変換処理は convStmt，convExpr で行われており，いずれの関数もそこに現れた変数のリストと変換後の中間表現のリストの組を返している．注意点としては，返された変数のリストの先頭に，最終結果が格納された変数が配置されることが挙げられる．実装をこのように統一することで条件節に指定すべき変数などを容易に得ることができる．

- A_IfStmt の変換
  条件節が真ならば，そのまま処理を進めて True 節が終わったところで末尾へジャンプする．条件節が偽ならば，IJumpFls によって False 節にジャンプして処理を進める．

```
convStmt (A_IfStmt _ cond tr fls)
    = do (varsCond,  stmtCond)  <- convExpr cond
         (varsTrue,  stmtTrue)  <- convStmt tr
         (varsFalse, stmtFalse) <- convStmt fls
         (lfls:lexit:[])    <- freshLabels 2
         return (varsCond ++ varsTrue ++ varsFalse,
                 stmtCond
                 ++ [IJumpFls (result varsCond) lfls]
                 ++ stmtTrue
                 ++ [IJump lexit,
                     ILabel lfls]
                 ++ stmtFalse
                 ++ [ILabel lexit])
```

- A_WhileStmt の変換
  先頭で条件節を評価して，その値が偽であるときだけ末尾へジャンプして終了する．条件節が真の場合は本体を実行し，その後，条件節を更新してジャンプの手前まで処理を戻す．条件節の更新は注意が必要で，変数は変更する必要がないがラベルを含む場合にそれらをフレッシュな値に書き換える必要がある．詳細はコードを参照のこと．

```
convStmt (A_WhileStmt _ cond body)
    = do (varsCond, stmtCond) <- convExpr cond
         (varsBody, stmtBody) <- convStmt body
         updateCond           <- renameLabel stmtCond
         (lloop:lexit:[])     <- freshLabels 2
         return (varsCond ++ varsBody,
```

```
                stmtCond
                ++ [ILabel lloop]
                ++ [IJumpFls (result varsCond) lexit]
                ++ stmtBody
                ++ updateCond
                ++ [IJump lloop, ILabel lexit])
```

- A_AssignExpr の変換
  左辺がポインタの場合は IWrite (メモリ書き込み)，変数の場合は ILet (単純な代入) というように分岐する．これ以外の形は意味解析で弾かれているので実装する必要はない．

```
convExpr (A_AssignExpr _ dest src)
    = case dest of
        (A_UnaryPrim _ "*" dst) -> do (vars1, stmts1) <- convExpr dst
                                      (vars2, stmts2) <- convExpr src
                                      return (vars2 ++ vars1,
                                              stmts1 ++ stmts2
                                             ++ [IWrite (result vars1)  (result vars2)])
        (A_IdentExpr vdst) -> do (vars, stmts) <- convExpr src
                                 return (vars, stmts ++
                                         [ILet vdst (result vars)])
```

- && の変換
  資料で提示されているように論理演算子は MIPS の命令に存在しないので，等価な if 文相当の中間表現列に変換する．ひとつ目の式を評価して偽ならば結果をゼロにする部分へジャンプ，真ならば処理を進めてふたつ目の式に対して同様の処理を行う．いずれの値も真ならば結果に 1 を代入して末尾へジャンプする．|| も同様の考え方で実装されている．

```
do (vars1, stmts1)  <- convExpr e1
   (vars2, stmts2)  <- convExpr e2
   dest             <- freshVar
   (lfls:lexit:[])  <- freshLabels 2
   return (dest:(vars1 ++ vars2),
           stmts1
           ++ [IJumpFls (result vars1) lfls]
           ++ stmts2
           ++ [IJumpFls (result vars2) lfls,
               ILi dest 1,
               IJump lexit,
```

```
                    ILabel lfls,
                    ILi dest 0,
                    ILabel lexit])
```

3 課題 14

3.1 はじめに

本課題ではアドレス割り当て処理の説明を記述する．アドレス割り当ては MIPS 特有の操作となるので，MCode という別立ての中間表現を作って IVar が入っていたところにアドレスを入れていく，というように実装している．

3.2 MIPS 中間コード

MIPS 中間コードは以下のようになる．汎用中間表現との違いは，関数オブジェクトを関数名に差し替えた点と，MFuncDecl に関数フレームのサイズを追加した点である．アドレスは MVar というデータ構造で表現しており，今のところ $fp と $gp についてオフセット値を格納できるようになっている．

```
type MProgram = [MDecl]
type Label = String

data MVar  = Fp Int
           | Gp Int
             deriving(Eq, Ord)

instance Show MVar where
    show (Fp n)    = show n ++ "($fp)"
    show (Gp n)    = show n ++ "($gp)"

data MDecl = MVarDecl MVar
           | MFunDecl Int String [MVar] [MCode]
             deriving(Show, Eq, Ord)

data MCode = MLabel     Label
           | MLi        MVar Integer
           | MLet       MVar MVar
           | MAop       String MVar MVar MVar
```

```
  | MRelop     String MVar MVar MVar
  | MWrite     MVar MVar
  | MRead      MVar MVar
  | MAddr      MVar MVar
  | MJumpTr    MVar Label
  | MJumpFls   MVar Label
  | MJump      Label
  | MCall      MVar String [MVar]
  | MReturn    MVar
  | MRetVoid
  | MPrint     MVar
    deriving(Show, Eq, Ord)
```

3.3 アドレス割り当て機構

基本的な考え方は (fp オフセットの最小値, gp オフセットの最小値, IVar と MVar の対応テーブル) という状態を持ちまわりながら変数をアドレスに置き換えていくというものである．中間表現に直す段階で構造自体はほとんど崩しているので，コードに新しい変数が現れるたびに getAddr でフレッシュなアドレスを生成して返せば良い．
違う関数を評価するときには fp の値とテーブルをリセットしなければいけないのだが，テーブルにグローバル変数のエントリは残さなければいけないので多少の処理を加えている．

4 課題 16

4.1 はじめに

以下では MIPS コード生成の実装を説明する．といっても，関数フレームの形成の仕方などは講義資料で述べられており，実装も対応する文字列を素直に実装するだけなので特筆することはほとんどない．

4.2 実装の注意

現在レジスタ割り当てを行っていないので全ての変数はメモリを経由して使用される．それでも，現在見ている命令の結果が $t0 に格納されるように，統一したコーディングを行っている．

5　提案

実験で提案されている中間表現は少し水準が高すぎる．このままでは機械独立の最適化がやりづらく，またターゲット言語を複数にしようとしたところで重複コードが生まれる．モジュール分割の境界はいわゆる 3 アドレスコードレベルにあると思われるので，本レポートで提示したデータ構造を参考にしていただきたい．

6　感想・総評

今回のコンパイラは今まで書いてきたものの中でも最大規模となり非常に学びが多かった．ポインタ演算など統一的ではない演算に対して愚直に実装を行う力や，厳密ではない型に対してなるべく正しそうな解釈を与えてそれらしくする力など，コード片を眺めて再現するだけでは得られない経験を得られたように思う．

最大の反省として，全体の仕組みを見通さないまま目先の実装だけに取り組みつづけてスパゲッティコードを生産してしまったことが挙げられる．結局ほとんどを書きなおして体裁は整えたが，データ構造の設計などはよくよく考えてから行わなければいけないと強く感じた．

中間表現やディレクトリ構造を整理して，あとの最適化やターゲット言語の変更につなげることができているので最終的な結果としては満足している．コンパイラの面白いところはこれからなので，夏休みをつぎ込んで改良を重ね有識者に送りつけていきたいと思う．