中間レポート 2

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。コンパイラは ghc 7.10.1 を利用しており、以前のバージョンに比べて Prelude が拡張されているなどのクリティカルな差異があるので注意せよ。不明な点があれば以下に挙げる資料を参照せよ。

- 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)

2　課題 8

資料で提示されている構文糖衣を実装する。

2.1　単項の (-) 演算子

単項の (-) を (-1) との積に変換することによって構文木を簡略化する。演算子テーブルで (-) 演算子に対応する変換関数 op_neg を実装することで実現している。オペランドを受け取り BinaryPrim (つまり (-1) との積) を返すようになっている。

```
op_neg = do {
           pos <- getPosition <* (reservedOp "-");
           return $ BinaryPrim pos "*" (Constant pos (-1)) }
```

2.2　else 節のない if 文

else 節のない if 文を、else 節が空文であるような if-else 構文に変換することで構文木を簡略化する。以下のように直感的な実装となっている。

```
ifStmt :: Parser Stmt
```

```
ifStmt = do
  pos  <- getPosition
  cond <- symbol "if" >> parens expr
  true <- stmt
  return $ IfStmt pos cond true (EmptyStmt pos)
```

2.3 for 文

for 文を資料 17 ページにあるような形式に変換する。以下のような実装になっている。初期化式、条件式、更新式の 3 つを順番にパースしてきて構文糖衣を組み立てている。初期化式が省略されている場合は EmptyStmt を、条件式が省略されている場合は定数の 1 を、更新式が省略されている場合は EmptyStmt をそれぞれ埋め込むことにしている。また forToWhile では本体が複合文であるかどうかに応じて処理を分岐している。

```
forStmt :: Parser Stmt
forStmt = do
  pos    <- getPosition <* (symbol "for" >> symbol "(")
  init   <- option (EmptyStmt pos) (liftM (ExprStmt pos) expr) <* semi
  cond   <- option (Constant pos 1) expr <* semi
  update <- option (EmptyStmt pos) (liftM (ExprStmt pos) expr) <* symbol ")"
  body   <- stmt
  return $ forToWhile pos body init cond update

forToWhile :: SourcePos -> Stmt -> Stmt -> Expr -> Stmt -> Stmt
forToWhile pos body init cond update
    = CompoundStmt pos [init, WhileStmt pos cond whileBody]
    where
      whileBody = case body of
                    (CompoundStmt p stmts) -> (CompoundStmt p
                                               (stmts ++ [update]))
                    stmt -> (CompoundStmt pos [stmt, update])
```

2.4 配列参照式

配列参照式をポインタによる間接参照式に置き換える。インデックスは先頭ポインタからのオフセットであるからポインタ演算に置き換えることができる。中では多次元配列のアクセスもカバーするような実装になっていて、添字アクセスの列を accesser に束縛しそれを外側から * 演算子と + 演

算子を利用して巻き取っていく形になっている。

```
arrayAccess :: Expr -> Parser Expr
arrayAccess h = do
  accesser <- many1 $ liftM2 (,) getPosition (brackets expr)
  return $ foldl (\acc (p, i) ->
                  UnaryPrim p "*" (BinaryPrim p "+" acc i)) h accesser
```

2.5 print 関数

資料ではプロトタイプ宣言を追加して整合性を保つとなっているが、実装の都合で解析の初期環境に print 関数を登録する形で解決した。関数の実体はないので中間表現以降で特別に分岐することになる。

```
initialEnv :: Env
initialEnv = M.fromList [(0, [("print",
                               (ObjInfo Func (CFun CVoid [CInt]) 0))])]

semanticAnalyze :: Program -> (A_Program, [String])
semanticAnalyze prog = runEnv body initialEnv
    where body = do collectGlobal prog
                    ret <- analyze prog
                    typeCheck ret >> return ret
```

3 課題 10

3.1 オブジェクト情報の収集・変数との対応付け

抽象構文木をたどって変数および関数の宣言を解析することでオブジェクトの情報を収集していく。オブジェクトの情報は ObjInfo 型のレコードで、種類、型、レベルを保持し環境へ追加していく。独立した局所環境を区別するために環境の追加と削除を繰り返しながら木をたどるので、変数との対応付けも同時に行っている。最終的には宣言と変数に関する情報を ObjInfo に変換した Analyzed_AST を定義して返している。

3.1.1 型の定義

オブジェクトの情報に関する ObjInfo に関する宣言は以下のようになっている。概ね資料の通りなので再び説明することはしない。

```
data ObjInfo  = ObjInfo { kind  :: Kind,
                          ctype :: CType,
                          level :: Level
                        }deriving(Eq, Show, Ord)

type Level = Int
data Kind  = Var | Func | FuncProto | Parm deriving(Show, Eq, Ord)
data CType = CInt
           | CVoid
           | CPointer CType
           | CArray   CType Integer
           | CFun     CType [CType]
           deriving(Ord)
```

3.1.2 オブジェクト情報の収集

以下にアルゴリズムの概要を示す。2 と 3 について、環境に入る操作と環境を削除して出てくる操作はアトミックな操作で一般化できるので、Python の with 文にならって withEnv というコンビネータを用意している。そこがブロック解析のエントリーポイントとなるので、コード理解の助けとしてほしい。

1. まず collectGlobal 関数によってグローバルな宣言を全て集める。(グローバルな宣言はプログラムのどこからでもアクセスすることができるので特別扱いする必要がある。またこのとき重複した宣言やプロトタイプ宣言の整合性などをチェックしていく)
2. グローバルな環境を持った状態で個別の関数の環境へ再帰的に入って情報を収集する。
3. 関数のブロックに限らず複合文の個別な閉居を抜ける時は、その環境の変数は無効になるので削除してから出る。

3.1.3 変数とオブジェクト情報の対応付け

変数とオブジェクト情報の対応付けは、上のオブジェクト情報の収集と並行して行っている。上記フローの 2 で個別の環境 (つまり Stmt 以下の構造) に入り、変数に当たった時は 環境に対して find をして変数情報を引き出して新たな木に組み込む。単純に木を変換する以上の処理が必要になるのは以下の 4 点である。これ以外では木の形は変わらないので愚直な実装になっている。

- analyzeStmt で CompoundStmt を読んだ時、withEnv で下の階層へ入る。
- analyzeStmt で DeclStmt を読んだ時、宣言されている情報を環境に追加する。
- analyzeExpr で ApplyFunc を読んだ時、環境から関数名を find して木に情報を埋め込む。変数を引いた場合は型検査で落とされるのでここでは気にしない。

- analyzeExpr で IdentExpr を読んだ時、環境から変数名を find して木に情報を埋め込む。このとき関数を引いた場合はエラーとする。

変数情報を埋め込んだ AnalyzedAST の定義は以下のようになる。各ノードの役割は中間レポート 1 で解説しているのでそちらを参照していただきたい。

```
type A_Program = [A_EDecl]
type A_Idnentifier = (Identifier, ObjInfo)

data A_EDecl = A_Decl A_Idnentifier
             | A_Func SourcePos A_Idnentifier [A_Idnentifier] A_Stmt
               deriving(Eq, Show)

data A_Stmt = A_EmptyStmt
            | A_ExprStmt     A_Expr
            | A_DeclStmt     [A_Idnentifier]
            | A_CompoundStmt [A_Stmt]
            | A_IfStmt       SourcePos A_Expr A_Stmt A_Stmt
            | A_WhileStmt    SourcePos A_Expr A_Stmt
            | A_ReturnStmt   SourcePos A_Expr
            | A_RetVoidStmt  SourcePos
              deriving(Eq, Show)

data A_Expr = A_AssignExpr SourcePos A_Expr A_Expr
            | A_UnaryPrim  SourcePos String A_Expr
            | A_BinaryPrim SourcePos String A_Expr A_Expr
            | A_ApplyFunc  SourcePos A_Idnentifier [A_Expr]
            | A_MultiExpr  [A_Expr]
            | A_Constant   Integer
            | A_IdentExpr  A_Idnentifier
              deriving(Eq, Show)
```

3.2 重複定義の検査

重複定義の検査は環境を拡張する extendEnv 関数に組み込まれているので、特に明示しなくても検査されるようになっている。あるレベルにオブジェクトを追加する前に find を実行して、そのレベルでオブジェクトが見つかる場合とそうでない場合に分岐している。関係するプログラムは以下の

部分である。

```
findAtTheLevel :: Level -> Identifier -> StateEnv (Maybe ObjInfo)
findAtTheLevel lev name = liftM (\e -> M.lookup lev e >>= lookup name) get
```

```
{- 重複を調べずに与えられた info を環境に追加する -}
addEnv :: Level -> (Identifier, ObjInfo) -> StateEnv ()
addEnv lev info_entry = liftM (M.insertWith (++) lev [info_entry]) get >>= put
```

```
{- 重複を調べてから環境を拡張する -}
extendEnv :: SourcePos -> Level -> (Identifier, ObjInfo) -> StateEnv ()
extendEnv p lev (name, info)
    = do {
        dupInfo <- findAtTheLevel lev name;
        case dupInfo of
          Nothing  -> tellShadowing p name lev >> addEnv lev (name, info)
          (Just _) -> duplicateError p name; }
```

3.3 式の形の検査

式の形の検査は型検査の一部として行っている。

3.3.1 & 演算子のオペランドのチェック

& 演算子のオペランドは変数参照式のみであるので以下のようにチェックしている。

```
checkAddressReferForm :: SourcePos -> A_Expr -> StateEnv ()
checkAddressReferForm _ (A_IdentExpr _) = wellTyped
checkAddressReferForm p _ = addrFormError p
```

3.3.2 代入文の形の検査

代入文の左辺には $*e$ の形の式か変数のみが現れるので以下のようにチェックしている。変数でも配列参照である可能性があるためパターンマッチで排除している。

```
checkAssignForm :: SourcePos -> A_Expr -> StateEnv()
checkAssignForm p (A_IdentExpr (name, ObjInfo kind ty _))
    = case (kind, ty) of
        (Var, (CArray _ _))  -> assignError p
```

```
        (Var, _)             -> wellTyped
        (Parm, (CArray _ _)) -> assignError p
        (Parm, _)            -> wellTyped
        (Func, _)            -> assignError p
        (FuncProto, _)       -> assignError p
checkAssignForm p (A_UnaryPrim _ "*" _) = wellTyped
checkAssignForm p _ = assignError p
```

3.4 型検査

AnalyzedAST をたどって資料のとおりに検査をしている。return が省略された関数の返り値の型を判定するために文にも擬似的に型を付けた点が工夫した点である。

3.4.1 文の型

Small C の定義によれば return 文の無い関数は自動的に void 型が付くことになっているが、全体に return 文がないかを調べるには構文木を再帰的にたどって結果をトップレベルまで伝播させなければいけない。その過程で Statement にも型を与えたほうが都合が良いので以下のように定義する。

- *returne*; には e の型が付く
- RetVoidSttm には void 型が付く
- CompoundStmt は return 文が含まれていればその型が、それ以外は void 型が付く
- IfStmt は 二つの節の型を併合した型が付く
- WhileStmt には本体の節の型が付く
- それ以外には void 型が付く

(注: 型の併合は CType を Ord 型クラスのインスタンスとして max 関数で行っている。これは return によって型が明示されれば暗黙の void が消えるという性質をよく満たす。テストの関係で Ord インスタンスは導出してあるので利用した)

3.5 サンプル

オブジェクト情報を埋め込んだあとの抽象構文木を示す。

```
int k;
int main(int j){
  int i;
  i = 3;
  if(i == 3){
    return k;
```

```
  }else{
    return j;
  }
}
```

実行結果が以下

```
([A_Decl ("k",ObjInfo {kind = Var, ctype = int, level = 0}),
  A_Func "test/test.c" (line 2, column 1)
    ("main",ObjInfo {kind = Func, ctype = (int) -> int, level = 0})
      [("j",ObjInfo {kind = Parm, ctype = int, level = 1})]
        (A_CompoundStmt
          [A_DeclStmt [("i",ObjInfo {kind = Var, ctype = int, level = 2})],
           A_ExprStmt (A_AssignExpr "test/test.c" (line 4, column 5)
                       (A_IdentExpr ("i",ObjInfo {kind = Var, ctype = int, level = 2}))
                       (A_Constant 3)),
           A_IfStmt "test/test.c" (line 5, column 3)
            (A_BinaryPrim "test/test.c" (line 5, column 8) "=="
              (A_IdentExpr ("i",ObjInfo {kind = Var, ctype = int, level = 2}))
              (A_Constant 3))
            (A_CompoundStmt
              [A_ReturnStmt "test/test.c" (line 6, column 5)
                (A_IdentExpr ("k",ObjInfo {kind = Var, ctype = int, level = 0}))])
            (A_CompoundStmt
              [A_ReturnStmt "test/test.c" (line 8, column 5)
                (A_IdentExpr ("j",ObjInfo {kind = Parm, ctype = int, level = 1}))])])],[])
```

4 感想

3 時間くらいで終わると思ったが 3 週間近くかかってしまった。パーサーとは違って実装方法もたくさんあり、それぞれの長所と短所をよく考えながらじっくりと取り組んだ。後の実装のことを良く理解しないまま間違ったコードを書いてしまうことも多かったので、全体の見通しを立ててからコードを書き始めるのは大事だと感じた。型検査を間違えて State で実装してしまい、あとで Either に差し替えたのだが、型を少し合わせるだけで動いてしまい Haskell の抽象とその一貫性に深く感動した。一方で破壊的代入が使えない難しさも痛感するところで、良いことばかりではないという思いも持った。ここからは実装の難しさというよりは知識の問題も大きいと思うのでしっかりと勉強していきたい。