計算機科学実験Ⅳ

1029-28-9483 勝田 峻太朗

2018 年 10 月 9 日

目 次

課題 1 (拡張構文: 任意)

階乗計算を行う MiniML プログラムを, 再帰を用いず loop 構文と組を使って書きなさい. 同じく, フィボナッチ数を求めるプログラムを,loop 構文と組を使って書きなさい.

階乗計算

```
1 (* calculate factorial *)
2 let fact n =
3 loop v = (n, 1) in
4 if v.1 > 1 then
5 recur (v.1 - 1, v.2 * v.1)
6 else v.2
```

フィボナッチ数

```
(* calculate the fibonacci number *)
let fib n =
loop v = (n, (1, 0)) in
if v.1 > 1 then
let tmp1 = v.2.1 in
let tmp2 = v.2.1 + v.2.2 in
recur (v.1 - 1, (tmp2, tmp1))
else v.2.1 + v.2.2
```

課題 2 (フロントエンド: 必須)

> MiniML の文法規則に従う MiniML プログラムを入力とし, 以下の `syntax.ml` により定義される抽象構文木を返す字句解析器・構文解析器を作成しなさい.

字句解析器・構文解析器ともに実験Ⅲの実装を参考にした.

字句解析器

課題にない実装として, コメントアウト機能を実装した.

lexer.mll

```
let reservedWords = [
  (* Keywords *)
  ("else", Parser.ELSE);
  ("false", Parser.FALSE);
  ("fun", Parser.FUN);
  ("if", Parser.IF);
  ("in", Parser.IN);
  ("let", Parser.LET);
  ("rec", Parser.REC);
  ("then", Parser.THEN);
  ("true", Parser.TRUE);
  ("loop", Parser.LOOP);
  ("recur", Parser.RECUR);
]
}

rule main = parse
  (* ignore spacing and newline characters *)
  [' ' '\009' '\012' '\n']+     { main lexbuf }
...
| "(*" { comment 1 lexbuf }
...
| "(" { Parser.LPAREN }
```

```
| ")" { Parser.RPAREN }
| ";;" { Parser.SEMISEMI }
| "+" { Parser.PLUS }
| "*" { Parser.MULT }
| "<" { Parser.LT }
| "=" { Parser.EQ }
| "->" { Parser.RARROW }
| "," { Parser.COMMA }
| "." { Parser.DOT }
...
and comment i = parse
  | "*)" { if i = 1 then main lexbuf else comment (i-1) lexbuf }
  | "(*" { comment (i+1) lexbuf }
  | _ {comment i lexbuf}
```

構文解析器

以下のようにコードを追加し, 構文解析器を作成した. LetRecExp は,let rec f = fun x -> e1 in e2 以外にも let rec f x = e1 in e2 の表現にも対応した.

parser.mly

```
open Syntax

 %}

-%token LPAREN RPAREN SEMISEMI RARROW
+%token LPAREN RPAREN SEMISEMI RARROW COMMA DOT
 %token PLUS MULT LT EQ
-%token IF THEN ELSE TRUE FALSE LET IN FUN REC
+%token IF THEN ELSE TRUE FALSE LET IN FUN REC LOOP RECUR

 %token <int> INTV
 %token <Syntax.id> ID
Expr :
    | e=LetExpr    { e }
    | e=LetRecExpr { e }
    | e=LTExpr     { e }
+   | e=LoopExpr   { e }

 LTExpr :
     e1=PExpr LT e2=PExpr { BinOp (Lt, e1, e2) }
MExpr :

 AppExpr :
     e1=AppExpr e2=AExpr { AppExp (e1, e2) }
+   | RECUR e1=AExpr { RecurExp (e1) }
    | e=AExpr { e }
```

```
27
28  AExpr :
29 AExpr :
30     | FALSE { BLit false }
31     | i=ID { Var i }
32     | LPAREN e=Expr RPAREN { e }
33 +   | LPAREN e1=Expr COMMA e2=Expr RPAREN { TupleExp(e1, e2) }
34 +   | e1=AExpr DOT i=INTV { ProjExp(e1, i) }
35
36  IfExpr :
37       IF e1=Expr THEN e2=Expr ELSE e3=Expr { IfExp (e1, e2, e3) }
38 FunExpr :
39       FUN i=ID RARROW e=Expr { FunExp (i, e) }
40
41  LetRecExpr :
42 -     LET REC i=ID EQ FUN p=ID RARROW e1=Expr IN e2=Expr
43 +     | LET REC i=ID EQ FUN p=ID RARROW e1=Expr IN e2=Expr
44 +     | LET REC i=ID p=ID EQ e1=Expr IN e2=Expr
45         { if i = p then
46             err "Name conflict"
47           else if i = "main" then
48             err "main must not be declared"
49           else
50             LetRecExp (i, p, e1, e2) }
51 +
52 +LoopExpr :
53 +   LOOP id=ID EQ e1=Expr IN e2=Expr { LoopExp (id, e1, e2) }
```

課題 3 (インタプリタ・型推論: 任意)

> 実験 3 で作成した ML^4 言語のインタプリタと型推論器を基に,MiniML 言語のインタプリタと型推論器を作成しなさい.

インタプリタ・型推論器ともに実験Ⅲを参考に作成した.

loop, recur 式への対応

loop 式,recur 式のインタプリタと型推論器を作成するのは困難であるため, これらはすべてインタプリタ・型推論器に通す前に let rec 式に変換した.

一般的に,

$$loop\ v\ = e_1\ in\ e_2$$

は, 新しい変数 f を用いて,

$$let\ rec\ f\ v = e_{2[recur\ e \to f\ e]}\ in\ f\ e_1$$

と表現できる.

例えば,

```
loop v = (1, 0) in
 if v.1 < 101
 then recur (v.1 + 1, v.1 + v.2)
 else v.2;;
```

は,

```
let rec f = fun v ->
if v.1 < 101 then f (v.1 + 1, v.1 + v.2)
else v.2
```

に変換される.

この操作を構文解析後の段階で, インタプリタ・型推論器に入る前に実装した.

main.ml

```
(* create fresh variable *)
let fresh_loopvar =
  let counter = ref 0 in
  let body () =
    let v = !counter in
    counter := v + 1;
    "f_" ^ string_of_int v
  in body

(* replace loop expressions with letrec *)
let recprog_of_loop p =
  (* replace recur e with f e *)
  let recur_subst newf e =
    let rec recur_subst_loop = function
      | FunExp(id, e) -> FunExp(id, recur_subst_loop e)
      | ProjExp(e, i) -> ProjExp(recur_subst_loop e, i)
      | BinOp(op, e1, e2) -> BinOp(op, recur_subst_loop e1, recur_subst_loop e2)
      | LetExp(id, e1, e2) -> LetExp(id, recur_subst_loop e1, recur_subst_loop e2)
      | AppExp(e1, e2) -> AppExp(recur_subst_loop e1, recur_subst_loop e2)
      | LetRecExp(i1, i2, e1, e2) -> LetRecExp(i1, i2, recur_subst_loop e1, recur_subst_loop e2)
      | LoopExp(id, e1, e2) -> LoopExp(id, recur_subst_loop e1, recur_subst_loop e2)
      | TupleExp(e1, e2) -> TupleExp(recur_subst_loop e1, recur_subst_loop e2)
      | IfExp(cond, e1, e2) -> IfExp(recur_subst_loop cond, recur_subst_loop e1, recur_subst_loop e2)
      | RecurExp(e) -> AppExp(newf, e)
      | _ as e -> e in
    recur_subst_loop e in
  let rec recexp_of_loop = function
    | LoopExp(v, e1, e2) ->
      let new_funct: id = fresh_loopvar () in
      let rece1 = recur_subst (Var new_funct) (recexp_of_loop e2) in
      let rece2 = AppExp(Var new_funct, e1) in
      LetRecExp(new_funct, v, rece1, rece2)
    | _ as e -> e in
  match p with
```

```
39    | Exp e -> Exp (recexp_of_loop e)
40
41  let rec read_eval_print env tyenv =
42    print_string "# ";
43    flush stdout;
44    try
45      let decl = Exp(Parser.toplevel Lexer.main (Lexing.from_channel stdin)) in
46      (* remove loop exp from program *)
47      let decl' = recprog_of_loop decl in
48      (match decl' with
49       | Exp e -> string_of_exp e |> print_endline);
50      let ty, new_tyenv = ty_decl tyenv decl' in
51      let (id, newenv, v) = eval_decl env decl' in
52      ...
```

インタプリタ

インタプリタでは, 新しい表現である tuple と proj に対応した.

まず, 新しい tuple データ型を追加した.

eval.ml

```
1  type exval =
2    | IntV of int
3    | BoolV of bool
4    | TupleV of exval * exval
5    | ProcV of id * exp * dnval Environment.t ref
6  and dnval = exval
```

また,eval_exp に対応する項目を追加した.

eval.ml

```
1  let rec eval_exp env = function
2  ...
3  | TupleExp(e1, e2) ->
4      let v1 = eval_exp env e1 in
5      let v2 = eval_exp env e2 in
6      TupleV(v1, v2)
7  | ProjExp(e, i) ->
8      (match eval_exp env e with
9      | TupleV(v1, v2) -> if i = 1 then v1
10        else if i = 2 then v2
11        else err "ProjExp: index not valid"
12     | _ -> err "error: projection of non-tuple")
13   | _ -> err "eval_exp: should not enter this match"
```

型推論器

型推論器にも,tuple と proj の型推論を追加した.

typing.ml

```
| TupleExp(e1, e2) ->
    let tyarg1, tysubst1 = ty_exp tyenv e1 in
    let tyarg2, tysubst2 = ty_exp tyenv e2 in
    let main_subst = unify(eqls_of_subst tysubst1 @ eqls_of_subst tysubst2) in
    let ty1 = subst_type main_subst tyarg1 in
    let ty2 = subst_type main_subst tyarg2 in
    (TyTuple(ty1, ty2), main_subst)
  | ProjExp(e, i) ->
    (let tyarg, tysubst = ty_exp tyenv e in
     let t1 = TyVar(fresh_tyvar()) in
     let t2 = TyVar(fresh_tyvar()) in
     let main_subst = unify(eqls_of_subst tysubst @ [(tyarg, TyTuple(t1, t2))]) in
     let ty1 = subst_type main_subst t1 in
     let ty2 = subst_type main_subst t2 in
     if i = 1 then (subst_type tysubst ty1, tysubst)
     else if i = 2 then (subst_type tysubst ty2, tysubst)
     else err "non valid projection target")
```

課題 4 (recur 式の検査: 必須)

> syntax.ml 中の recur_check 関数を完成させることにより,recur 式の検査を実装しなさい. parser.mly 中の呼び出している箇所を見ると分かるとおり, recur_check 関数は unit 型の値を返す. 末尾位置ではないところに書かれた recur 式を発見したら, 即座に例外を投げコンパイル処理を中断すること.

recur_check の内部に,Syntax.exp と, その expression が末尾位置であるかを示す is_tail を引数に取る再帰関数を定義し,recur 式が正しい位置にあるかどうか確認する.

normal.ml

```
(* ==== recur 式が末尾位置にのみ書かれていることを検査 ==== *)
(* task4: S.exp -> unit *)
let rec recur_check e is_tail: unit =
  let recur_err () = err "illegal usage of recur" in
  S.(match e with
      | RecurExp _ ->
        if is_tail then ()
        else recur_err ()
      | LoopExp (x, e1, e2) ->
        recur_check e1 false;
        recur_check e2 true
      | IfExp(e1, e2, e3) ->
```

```
175           recur_check e1 false;
176           recur_check e2 is_tail;
177           recur_check e3 is_tail
178         | LetExp(x, e1, e2) ->
179           recur_check e1 false;
180           recur_check e2 is_tail
181         | LetRecExp(f, x, e1, e2) ->
182           recur_check e1 false;
183           recur_check e2 is_tail
184         | FunExp(_, e) | ProjExp(e, _) ->
185           recur_check e false
186         | BinOp(_, e1, e2) | AppExp(e1, e2) | TupleExp(e1, e2) ->
187           recur_check e1 false;
188           recur_check e2 false
189         | _ -> () (* Var, ILit, BLit *)
190       )
191
192   (* ==== entry point ==== *)
193   let rec convert prog =
194     recur_check prog false;
195     normalize prog
```

課題 5 (正規形への変換: 必須)

言語 C への変換と, 正規形への変換を同時に行う,`normal.ml` 中の `norm_exp` 関数を完成させよ. 関数は次に示す形で実装すること. 引数 `f` を適切に用いれば各場合分けで数行書くだけで完成する.

課題 6 (クロージャ変換: 必須)

`closure.ml` の `convert` 関数を完成させることにより, クロージャ変換を実装しなさい.