計算機科学実験 レポート 1

1029-28-9483 勝田 峻太朗

2018 年 11 月 8 日

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.

クロージャ変換は, 以下のように実装した.

```
1  (* === conversion to closed normal form === *)
2  let rec closure_exp (e: N.exp) (f: cexp -> exp) (sigma: cexp
       Environment.t): exp =
3    match e with
4    | N.CompExp(N.ValExp(Var v)) ->
5      let may_fail v =
6        try
7          f (Environment.lookup ("_" ^ v) sigma)
8        with _ -> f (ValExp(Var ("_" ^ v)))
9      in may_fail v
10   | N.CompExp(N.ValExp(IntV i)) -> f (ValExp(IntV i))
11   | N.CompExp(N.BinOp(op, v1, v2)) -> f (BinOp(op, convert_val v1,
         convert_val v2))
12   | N.CompExp(N.AppExp(v1, v2)) ->
13     let new_app0 = fresh_id "closure_app" in
14     LetExp(new_app0, ProjExp(convert_val v1, 0),
15            f (AppExp(Var new_app0, [convert_val v1; convert_val v2])))
16   | N.CompExp(N.IfExp(v, e1, e2)) ->
17     (* closure_exp e1 (fun y1 ->
18         closure_exp e2 (fun y2 ->
19             CompExp(IfExp(convert_val v, f y1, f y2))) sigma) sigma *)
20     f (IfExp(convert_val v, closure_exp e1 (fun ce -> CompExp ce) sigma,
         closure_exp e2 (fun ce -> CompExp ce) sigma))
21   | N.CompExp(N.TupleExp (v1, v2)) -> f(TupleExp([convert_val v1;
         convert_val v2]))
22   | N.CompExp(N.ProjExp (v, i)) -> f(ProjExp(convert_val v, i-1)) (* {1, 2}
         -> {0, 1} *)
23   | N.LetExp(id, ce1, e2) ->
24     closure_exp (CompExp ce1) (fun y1 ->
25         LetExp(convert_id id, y1, closure_exp e2 f sigma)) sigma
26   | N.LetRecExp(funct, id, e1, e2) ->
```

```
27      let recpointer = fresh_id ("b_" ^ funct) in
28      let out_of_scope_vars = get_out_of_scope_variables e1 [id] in
29      let funct_tuple_list = (Var recpointer:: out_of_scope_vars) in
30      let rec make_tuple_env l i env =  (* make environment from id to
            projection to var in closure *)
31        match l with
32        | Var hd:: tl ->
33          let env' =  Environment.extend hd (ProjExp(convert_val (Var funct),
                i)) env in
34          make_tuple_env tl (i+1) env'
35        | [] -> env
36        | _ -> (match l with
37            | hd:: tl -> err ("unknown input in make_tuple_env" ^ "\n" ^
                  string_of_closure(CompExp(ValExp(hd))))
38            | _ -> err "none valid match") in
39      let sigma' = make_tuple_env out_of_scope_vars 1 Environment.empty in
40      let closure_contents = TupleExp(funct_tuple_list) in
41      let e1' = closure_exp e1 (fun ce -> CompExp(ce)) sigma' in
42      let e2' = LetExp(convert_id funct, closure_contents, closure_exp e2 f
            sigma') in
43      LetRecExp(recpointer, [convert_id funct; convert_id id], e1', e2')
44    | N.LoopExp(id, ce1, e2) ->
45      closure_exp (CompExp ce1) (fun y1 ->
46          LoopExp(convert_id id, y1, closure_exp e2 f sigma)) sigma
47    | N.RecurExp(v) -> RecurExp(convert_val v)
48
49  (* entry point *)
50  let convert e = closure_exp e (fun ce -> CompExp ce) Environment.empty
```

ソースコード 1　closure.ml

LetRecExp の実装

関数クロージャに必要な変数の発見

関数クロージャに必要な参照範囲外の変数を発見するため, get_out_of_scope_variables 関数を定義した. この関数は, 探索対象の N.exp を e として受け取り, すでにスコープに入っている変数名を included で受け取り, スコープ外の変数を list で返す. included には自分の関数名, 自分の引数を渡す.

```
1   let get_out_of_scope_variables (e: N.exp) (included: N.id list): value
        list =
2   let rec loop_e ex accum incl =
3     let rec loop_ce cex caccum incl =
4       N.(match cex with
5           | ValExp v | ProjExp(v, _) ->
6             (match (List.find_opt (fun x -> Var x = v) incl), v with
7              | Some x, _ -> caccum
8              | None, Var _ -> MySet.insert v caccum
```

```
9                | None, _ -> caccum)
10             | BinOp(_, v1, v2) | AppExp(v1, v2) | TupleExp(v1, v2) ->
11               MySet.union (loop_ce (ValExp v1) caccum incl) (loop_ce (ValExp
                     v2) caccum incl)
12             | IfExp(v, e1, e2) ->
13               MySet.union (loop_ce (ValExp v) caccum incl) (MySet.union
                     (loop_e e1 accum incl) (loop_e e2 accum incl)))
14       in
15       N.(match ex with
16           | LetExp(i, cex, ex) ->
17             MySet.union (loop_ce cex accum incl) (loop_e ex accum (i::incl))
18           | LoopExp(i, cex, ex) ->
19             MySet.union (loop_ce cex accum (i:: incl)) (loop_e ex accum (i::
                   incl))
20           | LetRecExp(i1, i2, e1, e2) ->
21             MySet.union (loop_e e1 accum (i2::incl)) (loop_e e2 accum
                   (i2::incl))
22           | CompExp(ce) -> loop_ce ce accum incl
23           | _ -> accum
24         )
25     in
26     List.map convert_val (MySet.to_list (loop_e e MySet.empty included))
```

ソースコード 2　スコープ外変数の発見

関数クロージャの生成と関数本体式の変換

得られたスコープ外変数が, 関数本体式内で正しくクロージャの要素として参照されるような変換を施すため, make_tuple_env 関数を用いて, 変数と ProjExp を対応付ける環境を生成する.

例えば, get_out_of_scope_variables で得られたスコープ外変数の列が,

```
[Var "x"; Var "y"; Var "z"]
```

であった場合, make_tuple_env は,

環境 sigma'

```
[(Var "x", funct.1); (Var "y", funct.2); (Var "z", funct.3)]
```

を生成する.

これを用いて関数雨本体式を closure_exp で変換することで, 関数本体式の変数はクロージャを参照するようになる.

また, このクロージャ変換の段階で, もとの言語での `ProjExp` は, 最初の要素を取り出すのに 1 を使っていたが, ここからすべて 0 で統一する.

全体の流れ

LetRec 式においては, 関数の id だけだったものが, 関数クロージャと関数ポインターの 2 つが必要になる.

そこで, 以下のような操作をしている.

1. 関数ポインターを recpointer に,`fresh_id ("b_"^ funct)` で生成する.(関数クロージャは関数名を引き継ぐ.) (ただし `funct` は関数名)

2. funct_tuple_list に,get_out_of_scope_variables を用いて, クロージャに必要な値を代入する.
3. 関数本体式内でクロージャのインデックスを用いてスコープ外変数を参照しなければいけないので, 自由変数とクロージャからインデックスで値を取り出す表現の対応を make_tuple_env を用いて生成し,sigma' 代入する.

```
| N.LetRecExp(funct, id, e1, e2) ->
    let recpointer = fresh_id ("b_" ^ funct) in
    let out_of_scope_vars = get_out_of_scope_variables e1 [funct; id] in
    let funct_tuple_list = (Var recpointer:: out_of_scope_vars) in
    let rec make_tuple_env l i env =  (* make environment from id to
        projection to var in closure *)
      match l with
      | Var hd:: tl ->
        let env' =  Environment.extend hd (ProjExp(convert_val (Var funct),
            i)) env in
        make_tuple_env tl (i+1) env'
      | [] -> env
      | _ -> (match l with
          | hd:: tl -> err ("unknown input in make_tuple_env" ^ "\n" ^
              string_of_closure(CompExp(ValExp(hd))))
          | _ -> err "none valid match") in
    let sigma' = make_tuple_env out_of_scope_vars 1 Environment.empty in
    let closure_contents = TupleExp(funct_tuple_list) in
    let e1' = closure_exp e1 (fun ce -> CompExp(ce)) sigma' in
    let e2' = LetExp(convert_id funct, closure_contents, closure_exp e2 f
        sigma') in
    LetRecExp(recpointer, [convert_id funct; convert_id id], e1', e2')
```

ソースコード 3　LetRec 式の実装

AppExp の変換

AppExp では, 関数のクロージャではなく, クロージャの第一要素である関数ポインタに対して関数適用することになる.

よって, 呼び出す関数のポインタを新しい変数に代入し, それに対して関数適用することとなる. 関数クロージャから関数ポインタを得るには, 関数クロージャの先頭要素を取れば良い.

```
| N.CompExp(N.AppExp(v1, v2)) ->
    let new_app0 = fresh_id "closure_app" in
    LetExp(new_app0, ProjExp(convert_val v1, 0),
           f (AppExp(Var new_app0, [convert_val v1; convert_val v2])))
```

ソースコード 4　AppExp の実装

課題 7 (平滑化: 必須)

flat.ml の flatten 関数を完成させることにより，正規形コードを平滑化しなさい.

```
(* ==== フラット化：変数参照と関数参照の区別も同時に行う ==== *)
let convert_id (i: C.id): id = i
let convert_id_list (il: C.id list): id list = il

let get_flat_exp ex =
  (* === helper functions === *)
  let fun_list = ref (MySet.empty: C.id MySet.t) in
  let append_fun v = fun_list := MySet.insert v !fun_list in
  let search_fun v = MySet.member v !fun_list in
  let decl_list = ref ([]: decl list) in
  let append_decl d = decl_list := (d :: !decl_list) in
  let convert_val (v: C.value): value =
    match v with
    | C.Var id -> if search_fun id
      then Fun(id)
      else Var(convert_id id)
    | C.IntV i -> IntV(i) in
  let convert_val_list (vl: C.value list): value list = List.map
      convert_val vl in
  let rec flat_exp (e: C.exp) (f: cexp -> exp): exp =
    match e with
    | C.CompExp(C.ValExp v) -> f (ValExp(convert_val v))
    | C.CompExp(C.BinOp(op, v1, v2)) ->
      let v1' = convert_val v1 in
      let v2' = convert_val v2 in
      f (BinOp(op, v1', v2'))
    | C.CompExp(C.AppExp(v, vl)) ->
      let v' = convert_val v in
      let vl' = convert_val_list vl in
      f (AppExp(v', vl'))
    | C.CompExp(C.IfExp(v, e1, e2)) ->
      let v' = convert_val v in
      (* flat_exp e1 (fun y1 ->
          flat_exp e2 (fun y2 ->
              f (IfExp(v', f y1, f y2)))) *)
      f (IfExp(v', flat_exp e1 (fun ce -> CompExp ce), flat_exp e2 (fun ce
          -> CompExp ce)))
    | C.CompExp(C.TupleExp(vl)) -> f (TupleExp(convert_val_list vl))
    | C.CompExp(C.ProjExp(v, i)) -> f (ProjExp(convert_val v, i))
    | C.LetExp(id, ce, e) ->
      flat_exp (CompExp ce) (fun cy1 ->
          LetExp(convert_id id, cy1, flat_exp e f))
    | C.LetRecExp(funct, idl, e1, e2) ->
      append_fun funct;
      let letrec' = RecDecl(convert_id funct, convert_id_list idl, flat_exp
```

```
            e1 (fun x -> CompExp x)) in
      append_decl letrec';
      flat_exp e2 f
    | C.LoopExp(id, ce, e) ->
      let id' = convert_id id in
      flat_exp (CompExp ce) (fun cy1 ->
          LoopExp(id', cy1, flat_exp e f))
    | C.RecurExp(v) -> RecurExp(convert_val v)
  in let converted = flat_exp ex (fun x -> CompExp x) in
  (converted, !decl_list)

let flatten exp =
  let toplevel_content, decl_list = get_flat_exp exp in
  decl_list @ [RecDecl("_toplevel", [], toplevel_content)]
```

LetRec 式の取り出しと Fun 変数型への対応

以下のポインタを定義し, 変換をする.

fun_list Fun 変数型へ対応するために, 関数へのポインタの集合を追加していく. convert_val では, id が fun_list に存在すれば Fun 変数型に変換され, そうでなければ Var 型に変換される.

decl_list LetRec 式を入れ子から取り出し, 並べるため,decl_list に追加していく.

LetRec 式の平滑化処理は次のように行う.

1. 関数ポインタの id を fun_list に追加する.
2. 平滑化した LetRec 関数を取り出し, decl_list に追加する.
3. 続きの exp を flat_exp に適応する.

返り値

decl_list と, 残った式を, 以下のように `decl list` にして返す.

```
let flatten exp =
  let toplevel_content, decl_list = get_flat_exp exp in
  decl_list @ [RecDecl("_toplevel", [], toplevel_content)]
```