課題3レポート

芦田聖太

提出日　17/07/027

Ex4.2.1

設計方針

- syntax.ml に ty とその出力 pp_ty を追加
- main.ml に ty の評価とその出力を追加
- typing.ml に評価方法を定義する。

実装

syntax.ml

```
~~
type ty = TyInt | TyBool

let pp_ty = function  TyInt -> print_string "int"
                    | TyBool -> print_string "bool"

~~
```

以上のように追加する。テキストに書かれてあるものと同様である。

main.ml

```
~~~

let rec read_eval_print env tyenv=
  print_string "# ";
  flush stdout;
  try
  let decl = Parser.toplevel Lexer.main (Lexing.from_channel
      stdin) in
  let rec repeat env1 tyenv1 x =
    let (ty, newtyenv) = ty_decl tyenv1 x in
    let (id, newenv, v, expr) = eval_decl env1 x in
    match expr with
            Nothing -> Printf.printf "val %s : " id;
                       pp_ty ty;
                       print_string "=";
                       pp_val v;
                       print_newline();
                       read_eval_print newenv newtyenv

~~~

let initial_tyenv =
  Environment.extend "iv" TyInt
    (Environment.extend "iii" TyInt
      (Environment.extend "ii" TyInt
        (Environment.extend "i" TyInt
          (Environment.extend "v" TyInt
            (Environment.extend "x" TyInt Environment.empty)))))

let _ = read_eval_print initial_env initial_tyenv
```

read_eval_print に引数として initial_tyenv を追加する。initial_tyenv は i　v を TyInt に束縛した環境である。read_eval_print の変更点としては、まず引数 tyenv を増やすことである。次に、与えられた tyenv に関して ty_decl で評価を行い、Nothing とのマッチング部分に pp_ty で表記させる。

typing.ml

```
~~~~
let ty_prim op ty1 ty2 = match op with
    Plus -> (match ty1, ty2 with
                  TyInt, TyInt -> TyInt
                | _ -> err ("Argument␣must␣be␣of␣integer:␣+"))
  | Minus -> (match ty1, ty2 with
                  TyInt, TyInt -> TyInt
                | _ -> err ("Argument␣must␣be␣of␣integer:␣+"))
  | Mult -> (match ty1, ty2 with
                  TyInt, TyInt -> TyInt
                | _ -> err ("Argument␣must␣be␣of␣integer:␣+"))
  | Lt -> (match ty1, ty2 with
                  TyInt, TyInt -> TyBool
                | _ -> err ("Argument␣must␣be␣of␣integer:␣+"))
  | Equal -> (match ty1, ty2 with
                  TyInt, TyInt -> TyInt
                | _ -> err ("Argument␣must␣be␣of␣integer:␣+"))
  | And -> (match ty1, ty2 with
                  TyBool, TyBool -> TyBool
                | _ -> err ("Argument␣must␣be␣of␣boolean:␣+"))
  | Or -> (match ty1, ty2 with
                  TyBool, TyBool -> TyBool
                | _ -> err ("Argument␣must␣be␣of␣boolean:␣+"))
  | Cons -> err "ty_prim␣Not␣Implemented!"

let rec ty_exp tyenv = function
    Var x ->
      (try Environment.lookup x tyenv with
          Environment.Not_bound -> err ("variable␣not␣bound:␣" ^
               x))
  | ILit _ -> TyInt
  | BLit _ -> TyBool
  | BinOp (op, exp1, exp2) ->
      let tyarg1 = ty_exp tyenv exp1 in
      let tyarg2 = ty_exp tyenv exp2 in
        ty_prim op tyarg1 tyarg2
  | IfExp (exp1, exp2, exp3) ->
      let tyarg0 = ty_exp tyenv exp1 in
      if tyarg0 = TyBool then
      (let tyarg1 = ty_exp tyenv exp2 in
      let tyarg2 = ty_exp tyenv exp3 in
        if tyarg1 = tyarg2 then tyarg1 else err("ifExp␣Not␣
            Implemented!"))
      else err ("Not␣Bool")
  | LetExp (id, exp1, exp2) ->
        let tyarg = ty_exp tyenv exp1 in
        ty_exp (Environment.extend id tyarg tyenv) exp2
  | _ -> err ("ty_exp␣Not␣Implemented!")

let ty_decl tyenv = function
    Exp e -> (ty_exp tyenv e, tyenv)
  | Decl (x, e) -> let v = ty_exp tyenv e in (v, Environment.
```

```
      extend x v tyenv)
  | _ -> err ("ty_decl␣Not␣Implemented!")
```

ty_prim では Plus から Mult までは TyInt と TyInt を受け取って TyInt を返し、Lt と Equal は TyInt と TyInt を受けて TyBool を返す。また、And と Or では TyBool と TyBool を受け取って TyBool を返す。それ以外のときはエラーを起こす。
ty_exp は ILit から BinOp まではテキストのものと同様である。LetExp では exp1 を評価して、id に s その結果を束縛して exp2 を評価する。if では exp1 を評価して TyBool であれば、exp2 と exp3 を評価し、型が同じであればその型を出力する。ty_decl には Decl を追加している、これによって let 宣言も型推論が可能となっている。

Ex4.3.1

設計方針

- pp_ty には TyVar と TyFun を追加。
- freevar_ty は TyFun と TyVar を探索していき出てきた型をリスト化して行く。

実装

pp_ty

```
let rec pp_ty = function
        ...........
         | TyVar x -> print_string ("'a" ^ string_of_int x)
         | TyFun (x, y) -> print_string "(" ; pp_ty x ;
             print_string "␣->␣";  pp_ty y ; print_string ")"
```

型変数は「'a 数字」の形で表すように設定した。TyFun に対しては括弧付きで表記するようにして、x と y に対してもう一度 pp_ty を行うことで型を探索して行くようにした。

freevar_ty

```
let rec freevar_ty = function
        TyVar x -> MySet.singleton x
        | TyFun (x, y) -> let tyx = freevar_ty x in
                                    let tyy = freevar_ty y in
                                    MySet.union tyx tyy
        | _ -> MySet.empty
```

TyInt と TyFun のときは型変数が存在しないので、empty を返す。TyVar のときは変数をリストにして返す。TyFun のときは x と y に関して freevar_ty を適用して、その結果を union をして統合する。

Ex4.3.2

設計方針

代入のリストと適用する型を引数にとり、代入されて型を出力させる。TyInt と TyBool には代入を適用しても意味がないので、TyVar と TyFun に関して操作を行う。

実装

subst, subst_type

```
type subst = (tyvar * ty) list

let rec subst_type a b = match b with
                         TyVar c ->
                         (match a with
                           [] -> b
                           | (d, e) :: rest -> if d = c then
                               subst_type rest e else subst_type
                               rest b )
                       | TyFun (f, g) -> TyFun ( subst_type a f,
                           subst_type a g )
                       | _ -> b
```

- 適用される側の型が TyVar のとき、リストが空でなければ、その先頭の要素を取り出す。取り出した組の 1 項目が TyVar の tyvar と等しければ、代入を適用して組の 2 項目に subst_type をさらに適用する。等しくなければ先頭要素を除いた型代入リストと代入される型を引数にして subst_type を行う。
- 適用される側の型が TyFun のとき、両項に subst_type を同じ型代入リストで適用する。

以上の操作により型代入を行うことができる。

Ex4.3.3

設計方針

与えられた等式集合の考えられるパターンに対してそれぞれ操作を定義して行く。明らかに等式なり得ない組が現れた場合はエラーとして検出する。

実装

unify

```
let rec subst_eqs s eqs = match eqs with
                          [] -> []
                        | (a, b) :: rest -> (subst_type s a,
                            subst_type s b) :: (subst_eqs s rest
                            )

let rec unify a =
  match a with
      [] -> []
    | x :: rest ->
        match x with
            (TyInt , TyInt) -> unify rest
          | (TyBool, TyBool) -> unify rest
          | (TyVar tyvar, TyInt) -> [ (tyvar, TyInt) ] @ unify (
              subst_eqs [(tyvar, TyInt)] rest )
          | (TyVar tyvar, TyBool) -> [ (tyvar, TyBool) ] @ unify
              ( subst_eqs [(tyvar, TyBool)] rest )
          | (TyVar tyvar, TyFun(b, c)) -> let varlist =
              freevar_ty (TyFun (b, c)) in
                                    if MySet.member tyvar
                                        varlist then err("can'
                                        t␣unify1")
                                    else  [(tyvar, TyFun(b, c)
                                        )] @ unify ( subst_eqs
                                         [(tyvar, TyFun(b, c))
                                        ] rest )
          | (TyInt, TyVar tyvar) -> [ (tyvar, TyInt) ] @ unify (
              subst_eqs [(tyvar, TyInt)] rest )
          | (TyBool, TyVar tyvar) -> [ (tyvar, TyBool) ] @ unify
              ( subst_eqs [(tyvar, TyBool)] rest )
          | (TyFun(a, b), TyVar tyvar) -> let varlist =
              freevar_ty (TyFun (a, b)) in
                                    if MySet.member tyvar
                                        varlist then err("can'
                                        t␣unify2")
                                    else [ (tyvar, TyFun(a,b))
                                         ] @ unify ( subst_eqs
                                         [(tyvar, TyFun(a,b))]
                                         rest )
          | (TyFun (x1, y1), TyFun (x2, y2)) -> unify( [(x1, x2)
              ; (y1, y2)] @ rest)
          | (TyVar tyvar, TyVar b) -> if (tyvar = b) then unify
              rest else
                                  [ (tyvar, TyVar b) ] @ unify (
                                       subst_eqs [(tyvar, TyVar
                                      b)] rest )
          | (_, _) -> err ("can't␣unify3")
```

等式集合の先頭要素から順に見て行く。先頭要素のパターンでそれぞれマッチングして行く。

(TyInt, TyInt) または (TyBool, TyBool) のとき

もとから型が同じなので型代入を形成する必要がない。残りの等式集合を unify する。

(TyVar, TyInt), (TyVar, TyBool), (TyInt, TyVar), (TyBool, TyVar) のとき

TyVar tyvar とする。tyvar から TyInt または TyBool への型代入 A を作る。先頭要素を除いた等式集合に型代入 A を subst_eqs で適用したものを unify して A につづけてリストを返す。(sebst_eqs は等式集合の要素となっているそれぞれの組に対して、組の両辺に型代入を適用するものである。)

(TyVar a, TyVar b) のとき

a と b の型変数が等しい時は型代入を形成する必要はないので、残りの等式集合を unify する。等しくないときは tyvar から b への型代入 A を作る。先頭要素を除いた等式集合に型代入 A を subst_eqs で適用したものを unify して A につづけてリストを返す。

(TyVar a, TyFun) または (TyFun, TyVar a) のとき

freevar_ty で TyFun の中にある型変数のリスト A を入手する。Myset.member で A のなかに a があるかをチェックする。あればエラーを発生させる。なければ a から TyFun への型代入 B を作る。先頭要素を除いた等式集合に型代入 B を subst_eqs で適用したものを unify して B につづけてリストを返す。

(TyFun(x1, y1), TyFun(x2, y2) のとき

x1 と x2、y1 と y2 は同じ型でないといけないので、(x1, x2),(y1,y2) の等式と先頭要素を除いた等式集合をつなげて unify を行う。

Ex4.3.4

型代入の代入後に代入前の型変数が含まれていると、無限ループが発生してしまう。例えば、等式集合 [(TyVar 'a0, TyFun('a1, 'a0)) ; (TyVar 'a0, TyInt)] を考える。unify を行うと型代入 ('a0, TyFun('a1, 'a0)) を (TyVar 'a0, TyInt) に行うことになる。代入結果は (TyFun('a1, 'a0), TyInt) となる。この結果には'a0 が含まれているので、さらに型代入 ('a0, TyFun('a1, 'a0)) をする。この操作は無限に行われる。したがって、制約が必要となる。

Ex4.3.5

型推論手続き

T-IF

if e1then e2 else e3 の型推論

1. Γ,e1 を入力として型推論を行い, S_1, τ_1 を得る。

2. Γ,e2 を入力として型推論を行い, S_2, τ_2 を得る。
3. Γ,e3 を入力として型推論を行い, S_3, τ_3 を得る。
4. S_1,S_2,S_3 を等式集合と考えて、$S_1 \cup S_2 \cup S_3 \cup \{(\tau_1, bool), (\tau_2, \tau_3)\}$ を単一化し、型代入 S_4 を得る。
5. S_4 と、τ_2 に S_4 を適用したものを出力する。

T-LET

let id = e1 in e2 の型推論

1. Γ,e1 を入力として型推論を行い, S_1, τ_1 を得る。
2. id を τ_1 で拡張した型環境 Γ′ を作成。
3. Γ′,e2 を入力として型推論を行い S_2, τ_2 を得る。
4. S_1,S_2 を等式集合と考えて、$S_1 \cup S_2$ を単一化し、型代入 S_3 を得る。
5. S_3 と、τ_2 に S_3 を適用したものを出力する。

T-ABS

fun x -> e の型推論

1. x を新たな型変数 τ_1 で拡張した型環境 Γ′ を作成。
2. Γ′,e を入力として型推論を行い S , τ_2 を得る。
3. 型代入 S と、τ_1 に S を適用したものから τ_2 への関数型を出力する。

T-APP

e1 e2 の型推論

1. Γ,e1 を入力として型推論を行い, S_1, τ_1 を得る。
2. Γ,e2 を入力として型推論を行い, S_2, τ_2 を得る。
3. 新たな型変数 τ_3 を作成。
4. 等式 eqs を (関数型 τ_2 から τ_3) = τ_1 とする。
5. S_1,S_2 を等式集合と考えて、$S_1 \cup S_2 \cup eqs$ を単一化し、型代入 S_3 を得る。
6. 型代入 S_3 と、τ_3 に S_3 を適用したものを出力する。

設計方針

上記の手続きを元に typing.ml を改造する。

実装

typing.ml

```
let rec ty_exp tyenv = function
    ..............
  | IfExp (exp1, exp2, exp3) ->
      let (s1, ty1) = ty_exp tyenv exp1 in
      let (s2, ty2) = ty_exp tyenv exp2 in
      let (s3, ty3) = ty_exp tyenv exp3 in
      let eqs1 = eqs_of_subst s1 in
      let eqs2 = eqs_of_subst s2 in
      let eqs3 = eqs_of_subst s3 in
      let eqs = eqs1 @ eqs2 @ eqs3 @ [(ty2, ty3) ; (ty1, TyBool)
          ] in
      let s4 = unify eqs in (s4, subst_type s4 ty2)

  | LetExp (id, exp1, exp2) ->
        let (a, b) = ty_exp tyenv exp1 in
        let (c, d) = ty_exp (Environment.extend id b tyenv) exp2
             in
        let s = unify( eqs_of_subst (c @ a) ) in (s, subst_type
            s d )

  | FunExp (id, exp) ->
    let domty = TyVar (fresh_tyvar ()) in
    let s, ranty =
      ty_exp (Environment.extend id domty tyenv) exp in
       (s, TyFun (subst_type s domty, ranty))

  | AppExp (exp1, exp2) ->
        let (s1, ty1) = ty_exp tyenv exp1 in
        let (s2, ty2) = ty_exp tyenv exp2 in
        let eqs1 = eqs_of_subst s1 in
        let eqs2 = eqs_of_subst s2 in
        let midty1 = TyVar (fresh_tyvar()) in
        let midty2 = TyVar (fresh_tyvar()) in
        let eqs = eqs1 @ eqs2 @ [ (TyFun (midty1, midty2) , ty1)
             ; (midty1, ty2) ] in
        let s3 = unify eqs in (s3, subst_type s3 midty2)

  | _ -> err ("ty_exp␣Not␣Implemented!")
```

IfExp

exp1〜exp3 についてそれぞれ ty_exp で型の評価を行う。等式集合に (ty2, ty3) と (ty1, TyBool) を加える (ty1〜ty3 はそれぞれ exp1〜exp3 の評価結果の型)。追加することで、unify を行うときに exp1 が bool かどうかということと exp2 と exp3 の型が等しいかどうかを確認できる。得られた等式集合を単一化して型代入のリストと ty2 に型代入を適用したものを返す。

LetExp

exp1 を評価してその結果を id に束縛して新たな環境を構築。得られた環境で exp2 を評価。exp1 と exp2 で得られた型代入を unify したもの

を exp2 で得られた型に適用する。最後に、unify された型代入と exp2 に型代入した結果を返す。

AppExp

exp1 と exp2 を評価する。(評価結果はそれぞれ ty1, ty2) 新たな型変数 a,b を作り、新たな TyVar a と TyVar b を作る。等式集合に (TyFun (TyVar a, TyVar b) , ty1) と (TyVar a, ty2) を追加する。追加することで、exp1 が関数であるかどうかということと exp2 が exp1 の引数として妥当かどうかということが確認できる。得られた等式集合を単一化して型代入のリストと TyVar b に型代入を適用したものを返す。

4.3.6

設計方針

ty_exp に LetRec を追加する。多相型を目指したので、tysc になっているが、実装はできなかった。

実装

```
..............

  | LetRecExp (id, para, exp1, exp2) ->
        let dummyTy = TyVar( fresh_tyvar() ) in
        let ty1 = TyVar( fresh_tyvar() ) in
        let newenv1 = Environment.extend para (TyScheme([], ty1)
            ) (Environment.extend id (TyScheme([], dummyTy))
            tyenv) in
        let (s1, ty2) = ty_exp newenv1 exp1 in
        let eqs = [(dummyTy, TyFun(ty1, ty2))] in
        let subst1 = unify (eqs @ (eqs_of_subst s1)) in
        let ty3 = subst_type subst1 dummyTy in
        let tysc = closure ty3 newenv1 subst1 in
        let (s2, ty4) = ty_exp (Environment.extend id tysc
            newenv1) exp2 in
        let subst2 = unify(eqs_of_subst (subst1 @ s2)) in  (
            subst2, subst_type subst2 ty4)

..................
```

新たな型変数で、id と para を拡張した環境で exp1 を評価する。TyFun(para の型, 評価結果) と id の型を等式集合として評価から得られた s1 と unify を行い subst1 を得る。id に束縛されている dummyTy を subst1 で型代入を行い ty3 を得て、closure で tysc を得る（多相にする際ここで問題が発生していると思われる。)。id を tysc で拡張した環境で、exp2 を評価。あとは LetExp と同様。

Ex.4.4.1

設計方針

- freevar_tysc, freevar_tyenv, closure を構成。
- ty_exp で Var x, LetExp, FunExp を改良し、(LetRecExp を追加改良)

実装

freevar_tysc

```
let rec freevar_tysc (tysc :tysc) =
                match tysc with
                      TyScheme(tyvars, ty) -> let in_tyvars =
                          freevar_ty ty in
                        MySet.diff in_tyvars  (MySet.from_list
                            tyvars)
```

ty 中の tyvar を feevar_ty で List 化したのち、MySet.diff でその中から tyvars の要素を除去する。

freevar_tyenv

```
let rec freevar_tyenv (tyenv :tysc Environment.t) =
        let f v a =
           (match v with
                TyScheme(tyvar, ty) -> MySet.union (freevar_tysc
                    v) a
           | _ -> err("Error␣of␣freevar_tyenv\n")) in
                Environment.fold_right f tyenv (MySet.empty)
```

fold_right と freevar_tysc を使って tyenv 中の各要素 TyScheme 中にある Γ には自由にでてこない型変数を取り出す。

closure

```
let closure ty tyenv subst =
  let fv_tyenv' = freevar_tyenv tyenv in
  let fv_tyenv =
    MySet.bigunion
      (MySet.map
        (fun id -> freevar_ty (subst_type subst (TyVar id)))
        fv_tyenv') in
  let ids = MySet.diff (freevar_ty ty) fv_tyenv in
      (* let rec printer vars = (match vars with
                                 [] -> print_string("closure:::\n")
                               | x :: rest -> print_int x;
                                   print_string(" ") ; printer rest)
    in printer (MySet.to_list fv_tyenv');   *)
  TyScheme (MySet.to_list ids, ty)
```

テキストの仕様をそのまま引用。

ty_exp

```
type tyenv = tysc Environment.t

...........

let rec ty_exp tyenv = function
    Var x ->
      (try
       let TyScheme (vars, ty) = Environment.lookup x tyenv in
        let s = List.map (fun id -> (id, TyVar (fresh_tyvar ()))
            )
          vars in  ([], subst_type s ty)
        with Environment.Not_bound -> err ("variable␣not␣bound:␣
            " ^ x))

..............

  | LetExp (id, exp1, exp2) ->  let (a, b) = ty_exp tyenv exp1
      in
                                let tysc = closure b tyenv a in
                                let (c, d) = ty_exp (Environment
                                    .extend id tysc tyenv) exp2
                                    in
                                let s = unify( eqs_of_subst (c @
                                     a) ) in (s, subst_type s d
                                    )

  | FunExp (id, exp) ->
    let domty = TyVar (fresh_tyvar ()) in
      let s, ranty =
        ty_exp (Environment.extend id (TyScheme( [], domty ))
            tyenv) exp in
        (s, TyFun (subst_type s domty, ranty))

  | _ -> err ("ty_exp␣Not␣Implemented!")
```

まず初めに、環境を tysc の Environment.t とする。次に Var x についてはテキストと同様の変更を加える。LetExp では exp1 を評価し、関数 closure を使って型スキームを入手する。その後、得た型スキームで id を束縛したのち exp2 を評価。返す値はそれまでのものと同様。FunExp の変更は環境の拡張を型スキームにするのみである。

全体テスト

```
# let s = fun x -> fun y -> if x y then y + 1 else y - 9;;
val s : ((int -> bool) -> (int -> int)) = <fun>

# fun x -> x x;;
can't unify
```

以上のテストはきちんと型推論ができている。

1. let 宣言,fun 式,if 式, 関数適用がうまく機能していることが確認できている。また、x が int -> bool で y が int になっていることから型変数から型への変換が unify できちんと行われていることが確認できる。

2. 課題 4.3.4 でもあったように無限の操作を防ぐために unify で型代入の変換先に変換前の型変数があるとエラーを起こすように設定してある。この式ではそれが確認できている。

感想

補助関数の定義に苦労した。補助関数の定義を行うためには全体を理解していないといけなかったが、あまり理解せずに作ってしまった。そのため本体の定義の部分で関数が間違っているのか式本体が誤っているかがはっきりとしなかった。特に苦労したのは unify だった。自分で作った最初の定義では、型代入の順番が正しい代入とは正反対になっておりそれに気付くのにかなりの時間がかかった。unify がうまくいってからはそれほど苦労はしなかった。きちんと全体の見通しをつけてからプログラムを組むことの大事さをこの演習を通して気付くことができた。