2022 年度 3 回生前期学生実験 SW

レポート 3（型推論実験）

作成者: 伊藤舜一郎 (学籍番号:1029-32-7548)

提出期限：7 月 21 日 24 時 00 分　再提出日: 2022 年 7 月 21 日

1 Exercise 4.2.1

1.1 実装内容

教科書に示されている、MiniML2 のための型推論アルゴリズムを実装するためにコードに加えるべき変更を参考にしつつ，T-Int と T-Plus のケースにならって，すべての場合について型推論アルゴリズムを完成させる．また，インタプリタに変更を加え，型推論ができるようにする．

1.2 変更した箇所

cui.ml,syntax.ml,main.ml,typing.ml に以下の変更点を加えることで実装した。

ソースコード 1: cui.ml

```
1  open Typing
2  open Syntax
3
4  let rec read_eval_print env tyenv = (* New! 型環境を REPL で保持 *)
5      print_string "#␣";
6      flush stdout;
7      let decl = Parser.toplevel Lexer.main (Lexing.from_channel stdin) in
8      let ty = ty_decl tyenv decl in (* New! let 宣言のための型推論 *)
9      let (id, newenv, v) = eval_decl env decl in
10       (* New! 型を出力するように変更 *)
11       Printf.printf "val␣%s␣:␣" id;
12       pp_ty ty;
13       print_string "␣=␣";
14       pp_val v;
15       print_newline();
16       (* 型環境を次のループに渡す.let 宣言はまだないので,tyenv を新しくする必要はない. *)
17       read_eval_print newenv tyenv
18
19 (* New! initial_env のための型環境を作る *)
20 let initial_tyenv =
21     Environment.extend "i" TyInt
22       (Environment.extend "v" TyInt
23         (Environment.extend "x" TyInt Environment.empty))
```

ソースコード 2: syntax.ml

```
1  (* MiniML の型を表す OCaml の値の型 *)
2  type ty =
3    TyInt
4  | TyBool
5
6  (* ty 型の値のための pretty printer *)
7  let pp_ty typ =
8    match typ with
9      TyInt -> print_string "int"
10   | TyBool -> print_string "bool"
11
12 let string_of_ty typ =
13   match typ with
14   TyInt -> "int"
15 | TyBool -> "bool"
```

資料の内容に加えて新たに string_of_ty 関数を実装した。この関数は ty 型の変数を受け取ってそれに応じて文字列を出力する関数で、インタプリタが型推論した結果を出力することができるための関数である。

ソースコード 3: main.ml

```
(* New! initial_tyenv を REPL の最初の呼び出しで渡す *)
let _ = read_eval_print initial_env initial_tyenv
```

ソースコード 4: typing.ml

```
open Syntax

exception Error of string

let err s = raise (Error s)

(* Type Environment *)
type tyenv = ty Environment.t

let ty_prim op ty1 ty2 = match op with
    Plus -> (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: <"))

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 tyarg1 = ty_exp tyenv exp1 in
      let tyarg2 = ty_exp tyenv exp2 in
      let tyarg3 = ty_exp tyenv exp3 in
      (match tyarg1, tyarg2, tyarg3 with
        TyBool, TyInt, TyInt -> TyInt
      | TyBool, TyBool, TyBool -> TyBool
      | _, _, _ -> err ("Argument must be of bool*'t*'t : if")
      )
  | LetExp (id, exp1, exp2) ->
      let value = ty_exp tyenv exp1 in ty_exp (Environment.extend id value tyenv)exp2
  | _ -> err ("Not Implemented!")

let ty_decl tyenv = function
    Exp e -> ty_exp tyenv e
  | _ -> err ("Not Implemented!")
```

資料の内容に加え、関数 ty_prim に T-Mult と T-Lt のケースに対応できるよう実装を行った。また、関数 ty_exp に T-If と T-Let のケースに対応することができるように実装を加えた。T-If のケースでは exp1, exp2, exp3 のそれぞれについての評価し、exp1 の評価結果が TyBool かつ、exp2,exp3 の評価結果が共に TyInt または TyBool であれば exp2,exp3 の評価結果を出力し、それ以外の場合ではエラーを出力するように実装した。T-Let のケースでは exp1 の評価結果と id を環境に加えてから exp2 を評価するように実装した。関数 ty_exp はこの後の課題で変更が加えられているので、4.2.1 に取り組んだ時の実装の内容はコメントアウトされている。

1.3 実行例

正しく動作するか確認すると、以下のようになった。

```
# 2<3;;
val - : bool = true
```

このようにインタプリタが型推論を行えていることがわかる。

2 Exercise 4.3.1

2.1 実装内容

string_of_ty と freevar_ty を完成させる.string_of_ty は ty -¿ string 型の関数で，ty 型の値を受け取るとその文字列での表現を返す．TyVar 0 が ”’a”，TyVar 1 が ”’b” のように，OCaml での型変数の文字列表現と合わせること．freevar_ty は，与えられた型中に現れる型変数の集合を返す関数で，型は ty -¿ tyvar MySet.t とする．型 ’a MySet.t は mySet.mli で定義されている型 ’a の値を要素とする集合を表す値の型である．

2.2 変更した箇所

syntax.ml に以下の変更点を加えることで実装した。

ソースコード 5: syntax.ml

```
1  open MySet
2
3  let fresh_tyvar =
4    let counter = ref 0 in (* 次に返すべき tyvar 型の値を参照で持っておいて, *)
5    let body () =
6      let v = !counter in
7        counter := v + 1; v (* 呼び出されたら参照をインクリメントして,古い counter の参照先の
             値を返す *)
8    in body
9
10 let rec freevar_ty ty =
11   match ty with
12     TyVar var -> MySet.singleton var
13     |TyFun (ty1,ty2) ->
14       union (freevar_ty ty1) (freevar_ty ty2)
15     |_ -> MySet.empty
16
17 let rec string_of_ty typ =
18   match typ with
19   TyInt -> "int"
20 | TyBool -> "bool"
21 | TyVar tyvar ->
22   let n = tyvar / 26 in (*何週目のa〜z かを求める*)
23     let id = tyvar - 26*n + 97 in (*
           char 型に変換し、その後 string 型に変換したときに対応する文字が出力できるように数を調整
           *)
24     let ch = char_of_int id in (*char 型に変換*)
25     let st = Char.escaped ch in (*string 型に変換*)
26     "'" ^ st ^ string_of_int n
27 | TyFun (ty1 , ty2) -> "(" ^ string_of_ty (ty1) ^ "->" ^ string_of_ty(ty2) ^ ")" (*(
       ty1 を文字列に変換したもの−>ty2 を文字列に変換したもの)を出力する*)
28 |_ -> ""
```

```
29
30 let pp_ty ty =
31   print_string (string_of_ty ty)
```

string_of_ty は TyInt を受け取ったら”int”を、TyBool を受け取ったら”bool”を出力する。また、引数が TyVar 型であれば、その引数がもつ int 型の tyvar に応じて”’a0”から順に出力するできるようにした。

3 Exercise 4.3.2

3.1 実装内容

型代入に関する以下の型，関数を typing.ml 中に実装する．type subst = (tyvar * ty) list val subst_type : subst -> ty -> ty

3.2 変更した箇所

typing.ml に以下の変更点を加えることで実装した。

ソースコード 6: typing.ml

```
1  type subst = (tyvar * ty) list
2
3  let hd (x::rest) = x (*list の先頭の要素を返す関数*)
4  let tl (x::rest) = rest (*list の先頭以外の要素を返す関数*)
5
6  let rec subst_type s ty = (*subst -> ty -> ty*)
7    match s with
8      [] -> ty
9      | _ -> (match ty with
10                 TyFun(a,b) -> TyFun((subst_type s a), (subst_type s b))
11                 | TyVar x -> let(var, typ) = hd s in
12                              if x=var then subst_type (tl s) typ (*置き換えを行い、
                                    s の残りの要素について subst_type を計算*)
13                              else subst_type (tl s) ty (*置き換えを行わず、
                                    s の残りの要素について subst_type を計算*)
14                 | _ -> ty)
```

関数 subst_type は subst 型の変数 s と ty 型の変数 ty を受け取って、s が空リストの場合は ty を返す。そうでない場合は ty が TyFun(a,b) の場合は a,b にそれぞれ subst_type を適用させたときの ty を返す。また、TyVar x の場合は s の先頭の要素 (var, typ) について x と var が等しかったら subst_type (tl s) typ を返すことで置き換えを行い、そうでない場合は subst_type (tl s) ty を返す。またそれ以外の場合は ty を返す。

4 Exercise 4.3.3

4.1 実装内容

単一化アルゴリズムを (ty * ty) list -> subst 型の関数 unify として実装する．

4.2 変更した箇所

typing.ml に以下の変更点を加えることで実装した。

ソースコード 7: typing.ml

```
1  (*subst_eqs: subst -> (ty * ty) list -> (ty * ty) list 型の等式集合に型代入を適用する関数. *)
2  let rec subst_eqs s eqs =
3    match eqs with
4      [] -> []
5      |(ty1,ty2)::rest -> (subst_type s ty1, subst_type s ty2)::(subst_eqs s rest) (*ty1,
           ty2 に型代入を行い、リストの残りの要素についても再帰的に行う。*)
6
7  (*単一化を行う関数*)
8  let rec unify l = (* (ty * ty) list -> subst *)
9    match l with
10     [] -> [] (* 単一化の 1番目*)
11     |(ty1,ty2)::rest ->
12       if ty1 = ty2 then unify rest (* 単一化の 2番目*)
13       else (match ty1, ty2 with
14         TyFun(ty11, ty12), TyFun(ty21, ty22) -> unify((ty11,ty21)::(ty12,ty22)::rest) (*単一化
               の 3番目*)
15         |TyVar tyvar1, _ (*単一化の 4番目*) ->
16           if (member tyvar1 (freevar_ty ty2)) then err("Can not unify") (*オカーチェック*)
17           else let subst_tys = [(tyvar1, ty2)] in
18                 let allsub = subst_eqs subst_tys rest in (*残りの制約中の α に τ を代入した制約を
                       求める*)
19                 subst_tys @ unify(allsub)
20         |_, TyVar tyvar2 (*単一化の 5番目*) ->
21           if (member tyvar2 (freevar_ty ty1)) then err("Can not unify") (*オカーチェック*)
22           else let subst_tys = [(tyvar2, ty1)] in
23                 let allsub = subst_eqs subst_tys rest in (*残りの制約中の α に τ を代入した制約を
                       求める*)
24                 subst_tys @ unify(allsub)
25         |_ -> err("Can not unify")) (*単一化の 6番目*)
```

subst_eqs は型の等式集合に型代入を適用する関数であり、subst 型の値と (ty * ty) list 型の値を受け取って (ty * ty) list 型の値を返す。unify は (*単一化を行う関数であり、(ty * ty) list 型の値を受け取って subst 型の値を返す。資料に提示されている単一化の方法と同様の事を unify は行っている。オカーチェックについては、member tyvar1 (freevar_ty ty2) で行っていて、ty2 中に現れる型変数の集合に ty1 が含まれているかどうかを if 文内で判定している。

5 Exercise 4.3.4

単一化アルゴリズムにおいて，オカーチェックの条件はなぜ必要か考察する．例えば $\alpha = \alpha ->\alpha$ という制約を許すと、型 $\alpha ->\alpha$ は $(\alpha ->\alpha)$ -> ($\alpha ->\alpha$) に変換され、そしてさらにこの型は... と無限に続いてしまう。この型推論器において型は有限の大きさとしているが、τ に α が表れることはこれに反してしまうのでオカーチェックの条件は必要である。

6 Exercise 4.3.5

6.1 実装内容

他の型付け規則に関しても同様に型推論の手続きを与える．そして，typing.ml に加えるべき変更の解説を参考にして，型推論アルゴリズムの実装を完成させる．

6.2 他の型付け規則

6.2.1 T-Var

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \text{T-Var}$$

- $\Gamma(x)$ の x が環境に束縛されているか確認する
- 束縛されているなら空の型代入とその型を出力する

6.2.2 T-Int

$$\frac{}{\Gamma \vdash n : \mathbf{int}}\ \text{T-Int}$$

- 空の型代入と int を出力として返す

6.2.3 T-Bool

$$\frac{b = \mathbf{true} \text{ or } b = \mathbf{false}}{\Gamma \vdash b : \mathbf{bool}}\ \text{T-Bool}$$

- b が true または false と等しいか確認する
- 空の型代入と bool を出力として返す

6.2.4 T-Mult

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 * e_2 : \mathbf{int}}\ \text{T-Mult}$$

- Γ, e_1 を入力として型推論を行い，θ_1，τ_1 を得る．
- Γ, e_2 を入力として型推論を行い，θ_2，τ_2 を得る．
- 型代入 θ_1, θ_2 を $\alpha = \tau$ という形の方程式の集まりとみなして，$\theta_1 \cup \theta_2 \cup \{\tau_1 = \mathbf{int}, (\tau_2, \mathbf{int})\}$ を単一化し，型代入 θ_3 を得る．
- θ_3 と **int** を出力として返す．

6.2.5 T-Lt

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 < e_2 : \mathbf{bool}}\ \text{T-Lt}$$

- Γ, e_1 を入力として型推論を行い，θ_1，τ_1 を得る．
- Γ, e_2 を入力として型推論を行い，θ_2，τ_2 を得る．
- 型代入 θ_1, θ_2 を $\alpha = \tau$ という形の方程式の集まりとみなして，$\theta_1 \cup \theta_2 \cup \{\tau_1 = \mathbf{int}, (\tau_2, \mathbf{int})\}$ を単一化し，型代入 θ_3 を得る．
- θ_3 と **bool** を出力として返す．

6.2.6 T-If

$$\frac{\Gamma \vdash e : \mathbf{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau}\ \text{T-If}$$

- Γ, e を入力として型推論を行い，θ，τ を得る．
- Γ, e_1 を入力として型推論を行い，θ_1，τ_1 を得る．
- Γ, e_2 を入力として型推論を行い，θ_2，τ_2 を得る．
- 型代入 $\theta, \theta_1, \theta_2$ を $\alpha = \tau$ という形の方程式の集まりとみなして，$\theta \cup \theta_1 \cup \theta_2 \cup \{(\tau, \mathbf{bool}), (\tau_1, \tau_2)\}$ を単一化し，型代入 θ_3 を得る．
- θ_3 と τ_2 に θ_3 を型代入したものを出力として返す．

6.2.7 T-Let

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}\ \text{T-Let}$$

- Γ, e_1 を入力として型推論を行い，θ_1，τ_1 を得る．
- x と τ_1 で拡張した Γ, e_2 を入力として型推論を行い，θ_2，τ_2 を得る．
- 型代入 θ_1, θ_2 を $\alpha = \tau$ という形の方程式の集まりとみなして，$\theta_1 \cup \theta_2$ を単一化し，型代入 θ_3 を得る．
- θ_3 と τ_2 に θ_3 を型代入したものを出力として返す．

6.2.8 T-Fun

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathbf{fun} x \to e : \tau_1 \to \tau_2}\ \text{T-Fun}$$

- Γ, x を入力として，x が τ_1 を持つと仮定する。
- x と τ_1 で拡張した Γ, e_2 を入力として型推論を行い，θ_2，τ_2 を得る.
- 型代入 θ_1, θ_2 を $\alpha = \tau$ という形の方程式の集まりとみなして，$\theta_1 \cup \theta_2$ を単一化し，型代入 θ_3 を得る.
- θ_3 と 関数型 (τ_1 に θ_3 を型代入したもの, τ_2) を出力として返す.

6.2.9 T-App

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{T-App}$$

- Γ, e_1 を入力として型推論を行い，θ_1，τ_1 を得る.
- Γ, e_2 を入力として型推論を行い，θ_2，τ_2 を得る.
- 新しい型変数 tyvar を作成する
- 型代入 $\theta, \theta_1, \theta_2$ を $\alpha = \tau$ という形の方程式の集まりとみなして, $\theta \cup \theta_1 \cup \theta_2 \cup \{(\tau_1, TyFun(\tau_2, tyvar)\}$ を単一化し，型代入 θ_3 を得る.
- θ_3 と tyvar に θ_3 を型代入したものを出力として返す.

6.3 変更した箇所

typing.ml に以下の変更点を加えることで実装した。

ソースコード 8: typing.ml

```
1  (* New! eqs_of_subst : subst -> (ty * ty) list
2      型代入を型の等式集合に変換.型の等式制約 ty1 = ty2 は (ty1,ty2) という
3      ペアで表現し,等式集合はペアのリストで表現. *)
4  let rec eqs_of_subst s =
5    match s with
6      [] -> []
7     |(tyvar, ty)::rest -> (TyVar tyvar, ty)::eqs_of_subst rest
8
9  let ty_prim op ty1 ty2 = match op with
10       Plus -> ([(ty1, TyInt); (ty2, TyInt)], TyInt)
11     | Mult -> ([(ty1, TyInt); (ty2, TyInt)], TyInt)
12     | Lt -> ([(ty1, TyInt); (ty2, TyInt)], TyBool)
13
14 (*型環境 tyenv と式 exp を受け取って,型代入と exp の型のペアを返す *)
15 let rec ty_exp tyenv exp =
```

```
16    match exp with
17      Var x −>
18       (try ([], Environment.lookup x tyenv) with
19            Environment.Not_bound −> err ("variable␣not␣bound:␣" ^ x))
20    | ILit _ −> ([], TyInt)
21    | BLit _ −> ([], TyBool)
22    | BinOp (op, exp1, exp2) −>
23        let (s1, ty1) = ty_exp tyenv exp1 in
24        let (s2, ty2) = ty_exp tyenv exp2 in
25        let (eqs3, ty) = ty_prim op ty1 ty2 in
26             (* s1 と s2 を等式制約の集合に変換して,eqs3 と合わせる *)
27        let eqs = (eqs_of_subst s1) @ (eqs_of_subst s2) @ eqs3 in
28             (* 全体の制約をもう一度解く. *)
29        let s3 = unify eqs in (s3, subst_type s3 ty)
30    | IfExp (exp1, exp2, exp3) −>
31        let (s1, ty1) = ty_exp tyenv exp1 in
32        let (s2, ty2) = ty_exp tyenv exp2 in
33        let (s3, ty3) = ty_exp tyenv exp3 in
34        let eqs = (eqs_of_subst s1) @ (eqs_of_subst s2) @ (eqs_of_subst s3) @ [(ty1, TyBool)] @ [(
              ty2,ty3)] in
35        let s4 = unify eqs in
36        (s4, subst_type s4 ty2)
37    | LetExp (id, exp1, exp2) −>
38        let (s1,ty1) = ty_exp tyenv exp1 in
39        let (s2,ty2) = ty_exp (Environment.extend id ty1 tyenv) exp2 in
40        let eqs = (eqs_of_subst s1) @ (eqs_of_subst s2) in
41        let s3 = unify eqs in
42        (s3,subst_type s3 ty2)
43    | FunExp (id, exp) −>
44        (* id の型を表す fresh な型変数を生成 *)
45        let domty = TyVar (fresh_tyvar ()) in
46             (* id : domty で tyenv を拡張し,その下で exp を型推論 *)
47        let s, ranty =
48          ty_exp (Environment.extend id domty tyenv) exp in
49          (s, TyFun (subst_type s domty, ranty))
50    | AppExp (exp1, exp2) −>
51        let (s1, ty1) = ty_exp tyenv exp1 in
52        let (s2, ty2) = ty_exp tyenv exp2 in
53        let tyvar1 = TyVar (fresh_tyvar()) in
54        let eqs = (eqs_of_subst s1) @ (eqs_of_subst s2) @ [(ty1, TyFun(ty2, tyvar1))]in
55        let s3 = unify eqs in
56        (s3,subst_type s3 tyvar1)
57    | _ −> err ("Not␣Implemented!")
58
59  let ty_decl tyenv = function
60    Exp e −> let (_, ty) = ty_exp tyenv e in ty
61    | Decl (id, exp) −> let (_, ty) = ty_exp tyenv exp in ty
62    | _ −> err ("Not␣Implemented!")
```

関数 ty_prim,ty_exp は型付け規則と同様の事を行えるように実装した。

6.4 実行例

実際に型推論機能ができているか確認する。

```
#let a = 1;;
val a : int = 1
# let x = fun y -> if y then 3 else 4;;
val x : (bool->int) = <fun>
# x (3 < 5);;
val - : int = 3
# let f = fun g -> if g then true else 5;;
Fatal error: exception Miniml.Typing.Error("Can not unify")
```

上記の実行例から型推論機能が実装できていることがわかる。

7 工夫した点

入力した環境を保存できるように cui.ml,typing.ml,typingTestGenerator.ml を変更した。

7.1 変更した箇所

ソースコード 9: cui.ml

```
1  let rec read_eval_print env =
2    print_string "#␣";
3    flush stdout;
4    try(let decl = Parser.toplevel Lexer.main (Lexing.from_channel stdin) in
5    let (id, newenv, v) = eval_decl env decl in
6    Printf.printf "val␣%s␣=␣" id;
7    pp_val v;
8    print_newline();
9    read_eval_print newenv)
10 ...
```

ソースコード 10: typing.ml

```
1  let ty_decl tyenv = function
2    Exp e -> let (_, ty) = ty_exp tyenv e in (tyenv, ty)
3    | Decl (id, exp) -> let (_, ty) = ty_exp tyenv exp in
4                        let newtyenv = Environment.extend id ty tyenv in
5                        (newtyenv, ty)
6    | _ -> err ("Not␣Implemented!")
```

ソースコード 11: typingTestGenerator.ml

```
1  let typing input =
2    Exec.exec
3      (fun env program ->
4          (* let env, ty = ty_decl env program *)
5          let env, ty = ty_decl env program in
6          env, ty)
7      Cui.initial_tyenv
8      input
```

ty_decl の返り値に環境の情報を含めることで入力した環境が次の入力にも用いられるようになった。

7.2 実行例

変更前

```
#let a = 1;;
val a : int = 1
#a;;
Miniml.Typing.Error("Variable not bound: a")
```

変更後

```
#let a = 1;;
val a : int = 1
#a;;
val - : int = 1
```

上記の実行例から型推論機能が実装できていることがわかる。

8 実験の感想

自分にとって型推論は仕組みをある程度理解することができても、実装を行うとなると手が動かなくなるような難しい単元だったので、TA の人や友人に質問をすることで型推論器、そして Ocaml の理解を深めた。そもそもプログラミング自体が苦手なので、この実験は全体を通してつらかった。