平成 30 年度 3 回生前期

計算機科学実験及演習 3

レポート 3　型推論

提出

2018/7/20

28 年度入学 1029288922 松島優太

4.2.1

教科書図 4.1 を main.ml に追加したが教科書通りのため割愛。図 4.2 を typing.ml に追加したうち、追記したものについて記述するものとする。

ty_prim

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: +"))
 | 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: <"))
 | Mt -> (match ty1, ty2 with
                TyInt, TyInt -> TyBool
              | _ -> 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: ||"))
 | _ -> err ("Not Implemented")
```

操作 BinOp について、その引数の型をマッチさせ出力の型を導出。それぞれ型付け規則(T-PLUS)から(T-OR)に対応する。また引数の型がマッチしない場合や操作が存在しない場合はエラーを吐く。

ここで(T-AND)と(T-OR)は

$$\frac{\Gamma \vdash e_1 : \mathrm{bool} \quad \Gamma \vdash e_2 : \mathrm{bool}}{\Gamma \vdash e_1 \ \&\& \ e_2 : \mathrm{bool}} \quad \text{(T-AND)}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{bool} \quad \Gamma \vdash e_2 : \mathrm{bool}}{\Gamma \vdash e_1 \ || \ e_2 : \mathrm{bool}} \quad \text{(T-OR)}$$

ty_exp

ty_exp は各型付け規則に対応している。

typing.ml

```
let rec ty_exp tyenv = function
    Var x ->
      (try Environment.lookup x tyenv with
          Environment.Not_bound -> err ("variable not bound: " ^ x))
```

(T-VAR)に対応する。tyenv(型付け規則の Γ)から変数 x を探し出して(Γ(x) = τ)型付けを行なう。Γ 内に存在しない場合、エラーを吐く。

```
  | ILit _ -> TyInt
  | BLit _ -> TyBool
```

(T-INT)・(T-BOOL)に対応する。

```
  | BinOp (op, exp1, exp2) ->
      let ty1 = ty_exp tyenv exp1 in
      let ty2 = ty_exp tyenv exp2 in
      ty_prim op ty1 ty2
```

ty1・ty2 に対して再帰的に ty_exp を行い型付けする($\Gamma \vdash e_{1,2} : \tau_{1,2}$)。その型を元にして ty_prim より型推論。

```
  | IfExp (exp1, exp2, exp3) ->
      let tytest = ty_exp tyenv exp1 in
      if (tytest!=TyBool)
      then err ("Argument must be boolean: 1st argument of if")
      else
      let tyexp2 = ty_exp tyenv exp2 in
      let tyexp3 = ty_exp tyenv exp3 in
      if (tyexp2!=tyexp3)
      then err ("Arguments must be same type: 2nd and 3rd arguments of if")
      else tyexp2
```

(T-IF)に対応する。まず if 式の条件である exp1 は bool 型でなければならず($\Gamma \vdash e_1 : bool$)そうでない場合はエラー。次に then 内の型と else 内の型は同一でなければならず($\Gamma \vdash e_{2,3} : \tau$)そうでない場合はエラー。以上の規則に則った結果、if 式は tyexp2(= tyexp3= τ)型となる($\Gamma \vdash$ if e_1 then e_2 else $e_3 : \tau$)。

```
  | LetExp (id, exp1, exp2) ->
      let tyvalue = ty_exp tyenv exp1 in
```

```
ty_exp (Environment.extend id tyvalue tyenv) exp2
```

(T-LET)に対応する。まず let 内 exp1 を現在の型環境において評価($\Gamma \vdash e_1 : \tau_1$)。それを環境内に組み込んだ上で in 内を評価($\Gamma, x : \tau_1 \vdash e_2 : \tau_2$)。この τ_2 が let-in 式の型である。

4.3.1

pp_ty

pp_ty は、main.ml 内実際に ML 言語による対話を行なう際の、型の表示を担う関数である。

端末

```
# fun x -> x + 1;;
val - : int -> int = <fun>
```

上記波線部の出力を決定している。

syntax.ml

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

TyInt、TyBool はその実 int、bool なのでそれを出力。

```
  | TyFun (ty1, ty2) -> (match ty1 with
      TyFun (tyunder1, tyunder2) -> (print_string "(";
                                     pp_ty ty1;
                                     print_string ")";
                                     print_string " -> ";
                                     pp_ty ty2)
    | _ -> pp_ty ty1;
           print_string " -> ";
           pp_ty ty2)
```

TyFun は「入力型 -> 出力型」と表示する。この入力型・出力型の表示は pp_ty により再帰的に決定すれば良い。

高階関数である場合、つまり入力型もしくは出力型が TyFun である場合、入力型・出力型自体が「型 -> 型」という形になる。

このとき「->」は右結合であるため、出力型が TyFun の場合は問題なく「型 -> 型 -> 型」であるが、入力型が TyFun の場合は「(型 -> 型) -> 型」としなければならない。

```
  | TyVar var -> (print_string "'a";
```

```
                    print_int var)
```

TyVar は型変数であるが、「'a」「'b」等アルファベット順に型変数を表示するとアルファベットの範囲を超えてしまう可能性がある。そのため「'a」に対する添え字として、内部で持っている型変数の ID(int 型)を付けることで表示を表現することとした。

freevar_ty

freevar_ty は与えられた型中の型変数の集合を返す関数。

syntax.ml

```
let rec freevar_ty ty =
  match ty with
    TyInt -> MySet.empty
  | TyBool -> MySet.empty
```

判定する型が TyInt・TyBool としてもう既に求まっている時は、その型が他の型変数を含んでいることはない。そのため空集合を返す。

ここで MySet.empty は空集合([])。

```
  | TyVar tyvar -> MySet.singleton tyvar
```

判定する型が TyVar の時、それに含まれる型変数はその tyvar そのもの 1 つだけである。よって tyvar のみからなる集合を返す。

ここで MySet.singleton は 1 つ要素を受け取り、その要素 1 つだけからなるリストを返す関数。

```
  | TyFun (tya, tyb) -> MySet.union (freevar_ty tya) (freevar_ty tyb)
```

判定する型が TyFun の時、それに含まれる型変数は tya と tyb に含まれる型変数の和集合である。

例えば TyFun (int, int)なら[]。TyFun (TyVar 0, TyFun (TyVar 1, TyVar 2))なら[0, 1, 2]。

よって tya と tyb に対し freevar_ty をかけてそれぞれに含まれる型変数を求め、それらの和集合を取れば良い。

ここで MySet.union は 2 つのリストから、その要素の和集合を持つリストを返す関数。同要素が 2 つのリストの両方にあるときは 1 つのみ出力リストに含める。

mySet.ml

```
let union xs ys =
  List.fold_left (fun zs x -> insert x zs) ys xs
```

4.3.2

subst

型代入を表すデータ構造 subst を次のように定義。

typing.ml

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

これは ID として tyvar を持つ型変数は、型 ty が代入されることを示している。

subst_type

関数 subst_type は subst と型を引数として取り、subst を型に適用する。

typing.ml

```
let rec subst_type sb ty =
   match ty with
      TyInt -> TyInt
   | TyBool -> TyBool
```

与えられた型が TyInt・TyBool として既に決定している場合、適用するまでもなく TyInt・TyBool。

```
   | TyVar var -> (match sb with
        [] -> TyVar var
     | (alpha, reaty) :: tl -> if var=alpha then subst_type tl reaty
                                            else subst_type tl ty)
```

与えられた型が TyVar の場合、sb が[]なら何もしなくて良い。
そうでない時、sb の第一要素(alpha, reaty)をまず適用、var＝alpha なら TyVar var は reaty と等しいことがわかる。もしくは var≠alpha では TyVar var は TyVar var のまま。
これで第一要素を適用し終えた後、再帰的に sb の残りの tl を適用。

```
   | TyFun (ty1, ty2) -> TyFun (subst_type sb ty1, subst_type sb ty2)
```

与えられた型が TyFun の場合、その入力型 ty1・出力型 ty2 に対して再帰的に subst_type をかけることで、それぞれに適用させる。

4.3.3

eqs_of_subst

typing.ml

```
let eqs_of_subst s = List.map (fun sx -> (TyVar (fst sx), snd sx)) s
```

subst を引数として取り、型の等式集合(同値の型 2 つのペアからなるリスト)を返す関数。
subst の要素 1 つを取ると((alpha, reaty)とする。ここで alpha＝fst sx、reaty＝snd sx)、

これは型変数 TyVar alpha が最終的に型 reaty となることを示しているため TyVar alpha と reaty は同値。
よって出力として(TyVar alpha, reaty)へと変換。
これを subst の全要素に行なう。

subst_eqs

typing.ml

```
let subst_eqs s eqs = List.map (fun (eqsxf, eqsxs) -> (subst_type s eqsxf, subst_type s eqsxs)) eqs
```

subst である s を、等式集合 eqs に適用する関数。
eqs の 1 要素(eqsxf, eqsxs)について、そのそれぞれについて s を適用させる。

unify

単一化アルゴリズムを実現した関数。等式集合 l を引数として取り、subst を返す。

typing.ml

```
let rec unify l = match l with
    [] -> []
```

$\mathcal{U}(\varphi)=\varphi$ に対応。

```
  | (tya, tyb) :: tl ->
      if tya=tyb then unify tl
```

$\mathcal{U}(\{(\tau, \tau)\}\uplus X')=\mathcal{U}(X')$に対応。等式の両項が既に等しければそれを除いた tl(=X')のみ残る。

```
      else match (tya, tyb) with
        (TyFun(ty11, ty12), TyFun(ty21, ty22)) ->
          unify ((ty11, ty21) :: (ty12, ty22) :: tl)
```

$\mathcal{U}(\{(\tau_{11}\to\tau_{12}, \tau_{21}\to\tau_{22})\}\uplus X')=\mathcal{U}(\{(\tau_{11}, \tau_{21}), (\tau_{12}, \tau_{22})\}\uplus X')$に対応。TyFun の場合はその両辺の型がそれぞれ等しい必要がある。

```
      | (TyVar alpha, ty) ->
          if MySet.member alpha (freevar_ty ty) (* ty 内に alpha があるかどうか *)
            then err ("Type error")
            else (* ペアの両方に subst_type  α→ty するような関数 *)
                  (alpha, ty) :: unify (subst_eqs [(alpha, ty)] tl)
```

$\mathcal{U}(\{(\alpha, \tau)\}\uplus X')\ (\text{if } \tau\neq\alpha)=\begin{cases}\mathcal{U}([\alpha\mapsto\tau]X')\circ[\alpha\mapsto\tau]\ (\alpha\notin FTV(\tau))\\ \text{error (その他)}\end{cases}$に対応。

$\tau \neq \alpha$ の条件については既に上で $\tau = \alpha$ の if を作っており、その else 節内にいるので問題ない。
freevar_ty で ty(=τ)内に alpha(=α)が存在するかチェック、あればエラー。
無ければ subst に(alpha, ty)を追加すると共に、これから unify する残りの等式集合内から、TyVar alpha を ty に変換しておく。

```
| (ty, TyVar alpha) ->
    if MySet.member alpha (freevar_ty ty)
    then err ("Type error")
    else (alpha, ty) :: unify (subst_eqs [(alpha, ty)] tl)
```

$\mathcal{U}(\{(\tau, \alpha)\} \uplus X')$ (if $\tau \neq \alpha$) $= \begin{cases} \mathcal{U}([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin FTV(\tau)) \\ \text{error (その他)} \end{cases}$ に対応。上と同様。

```
| _ -> err("Type error")
```

$\mathcal{U}(\{(\tau_1, \tau_2)\} \uplus X') = \text{error}$ に対応。その他の場合はエラーである。

4.3.4

$\alpha \in FTV(\tau)$なる τ が等式集合内に(α, τ)として存在していた場合、実際この型は無限に型代入を繰り返し決定されない型である。
例えば型の等式(α, α→α)を用いて α→int の型を決定しようとすると、

$$\alpha \to int$$
$$(\alpha \to \alpha) \to int$$
$$((\alpha \to \alpha) \to \alpha \to \alpha) \to int$$
$$\vdots$$

となり、無限に決定されない。よってこのような型代入はされるべきでない。
しかしながら、この単一化アルゴリズムではこのような型代入は一度のみ行なわれ、原理上可能となってしまう。そのためこれを検知しエラーとしなければならない。

4.3.5

subst の導入により ty_exp は型だけではなく subst も返すようにする必要が出てくる。そのため、ty_prim と ty_exp を更新する。

ty_prim

typing.ml

```
let ty_prim op ty1 ty2 = match op with
    Plus -> ([(ty1, TyInt); (ty2, TyInt)], TyInt)
```

```
(中略)
  | Or -> ([(ty1, TyBool); (ty2, TyBool)], TyBool)
  | _ -> err "Not Implemented!"
```

それぞれ型付け規則として、型チェックの代わりに型の等式集合として表現。型の等式集合と型のペアを出力する。

もしこの等式集合が他の型代入と衝突した場合、後に合わせて型代入とする際に単一化アルゴリズムでエラーとなるため、関数 unify がエラーを吐いてくれる。

ty_exp

型代入 subst と型のペアを出力する。

typing.ml

```
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)
```

ほとんど更新なし。既に決定しているので subst は空集合。

```
  | BinOp (op, exp1, exp2) ->
      let (s1, ty1) = ty_exp tyenv exp1 in
      let (s2, ty2) = ty_exp tyenv exp2 in
      let (eqs3, ty) = ty_prim op ty1 ty2 in
      let eqs = (eqs_of_subst s1) @ (eqs_of_subst s2) @ eqs3 in
      let s3 = unify eqs in
      (s3, subst_type s3 ty)
```

まず演算の両項exp1・exp2を現在の型環境の元で型推論し、ty1・ty2を得る。これをty_primに入れて出力の型 ty を得る。

これで得られた s1・s2・eqs3 は、型を決定づける型代入/等式集合であり、これらを全て反映した型代入を作成するため、s1・s2 を一度等式集合に変換し eqs3 と全部統合した上で unify する。

最後にこの s3 と、ty に s3 を適用したものを返す。

```
  | IfExp (exp1, exp2, exp3) ->
      let (stest, tytest) = ty_exp tyenv exp1 in
      let (s2, ty2) = ty_exp tyenv exp2 in
```

```
    let (s3, ty3) = ty_exp tyenv exp3 in
    (* 制約条件(等式集合)にして、unify に投げる。解いてもらって subst にする *)
    let eqs =
      (tytest, TyBool) :: (ty2, ty3) :: (eqs_of_subst stest) @ (eqs_of_subst s2) @
(eqs_of_subst s3) in
    let resultsubst = unify eqs in
    (resultsubst, subst_type resultsubst ty2)
```

まず exp1・exp2・exp3 を現在型環境で型推論、tytest・ty2・ty3 を得る。
上記同様 stest・s2・s3 を統合するが、ここでさらに型の等式条件として

- stest は TyBool 型であること(e_1: bool)
- ty2 と ty3 は同型であること(e_2: τ, e_3: τ)

が if 文として成立するために必要である。これらを等式集合に追加。
あとは unify して、これを元に ty2(または ty3)を推論して返す。

```
  | 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))
```

id に型変数が存在していないため、fresh_tyvar を用いてまだ使われていない型変数 TyVar domty を生成。
ここで現在型環境に id=domty であるということを追加した環境下で、exp を型推論、型代入 s と ranty を得る。
最後に s と、domty に s を適用したもの(=τ_1)を入力型、ranty(=τ_2)を出力型とする TyFun($\tau_1 \rightarrow \tau_2$)を返す。

```
  | AppExp (exp1, exp2) ->
    let (s1, ty1) = ty_exp tyenv exp1 in
    let (s2, ty2) = ty_exp tyenv exp2 in
    let tyf2 = TyVar (fresh_tyvar ()) in
    let resultsubst =
      unify ((ty1, TyFun (ty2, tyf2)) :: (eqs_of_subst s1) @ (eqs_of_subst s2)) in
    (* ty1 は最終 TyFun になる条件をつけて unify  ↑ *)
    (resultsubst, subst_type resultsubst tyf2)
```

まず exp1・exp2 を現在型環境で型推論、ty1・ty2 を得る。
ここで出力型(τ_2)に型変数が存在していないためまだ使われていない型変数 TyVar tyf2 を生成。

s1 と s2 を統合するが、ここでさらに型の等式条件として

- ty1 は最終的に TyFun の形となり、その入力型は ty2、出力型は tyf2 であること(e_1: $\tau_1 \rightarrow \tau_2$)

が必要である。これらを等式集合に追加。
最後にこれを unify したもの resultsubst と、tyf2 に resultsubst を適用したもの(=τ_2)を返す。

read_eval_print

typing.ml の ty_exp を更新したことで、対話形式の表示に関する関数を更新する必要があった。

main.ml

```
let rec read_eval_print env tyenv =
  print_string "# ";
(中略)
  let (s, ty) = ty_decl tyenv decl in
(中略)
    Printf.printf "val %s : " id;
    pp_ty ty;
(以下略)
```

subst は内部のみにおいて必要で、表示しない。

感想

- 予めプログラミング言語処理系で学んでいたことで、実験をスムーズに行なう手助けになったと感じた
- 逆に、プログラミング言語処理系で学んだ事を実践することにより、講義で理解しがたかった内容を理解することができた部分も多くあった
- インタプリタが行なっていることについては講義を受けたことがあったが、その中身については見たことがなかったため、より言語について深く知れたし、興味を喚起された
- 再帰についてコードを書くことが多く、より再帰を使いこなせることに近づいたように感じる