計算機科学実験 ソフトウェア 課題3

1029-28-9483 勝田 峻太朗

2018 年 7 月 12 日

目 次

4.2.1[]

textbook を参考にして, 実装した. 型推論の example code の都合上, 課題 2 の実装から, 再帰と複数 let 宣言機能を削除し, 実装を開始した.

4.3.1[]

ty を入力とし,tyvar の MySet を返す関数として実装した. ty_in を再帰的に舐めていき,TyVar(id) を出力に追加する.

```
(* 与えられた型中の型変数の集合を返す関数 *)
let freevar_ty ty_in =
  let rec loop ty current =
    (match ty with
     | TyVar a -> MySet.insert a current
     | TyFun(a, b) -> MySet.union (loop a current) (loop b current)
     | _ -> current) in
  loop ty_in MySet.empty
```

4.3.2[]

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

Function subst_type takes a list of subst s and a type ty to apply the subst to. This is done by applying resolve_subst to every element in s, which takes one substitution and applies it to ty recursively.

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

(* apply subst:(substutution) to ty:(type) *)
let rec subst_type s ty =
  let rec resolve_subst (subst_tyvar, subst_ty) ty =
    let subst_pair = (subst_tyvar, subst_ty) in
    match ty with
    | TyVar id -> if id = subst_tyvar then subst_ty else TyVar id
    | TyFun(a, b) -> TyFun(resolve_subst subst_pair a, resolve_subst subst_pair b)
    | TyInt -> TyInt
    | TyBool -> TyBool
  in match s with
  | top :: rest ->
    subst_type rest (resolve_subst top ty)
  | [] -> ty
```

4.3.3[]

> 上の単一化アルゴリズムを val unify : (ty * ty) list -> subst として実装せよ.

教科書の資料に従って実装した.

```
(* main unification algorithm *)
let rec unify eqs: (tyvar * ty) list  =
  let rec loop lst current_subst =
    (match lst with
     | (x, y) :: rest ->
       if x = y then loop rest current_subst else
         (match x, y with
          | TyFun(a, b), TyFun(c, d) -> loop ((a, c) :: (b, d) :: rest) current_subst
          | TyVar(id), b ->
            if not (MySet.member id (freevar_ty b)) then
              let mid = unify(subst_eqs (id, b) rest) in
              (id, b):: mid
            else err "unify: could not resolve type"
          | b, TyVar(id) ->
            if not (MySet.member id (freevar_ty b)) then
              let mid = unify(subst_eqs (id, b) rest) in
              (id, b) :: mid
            else err "unify: could not resolve type"
          | _ -> err "unify: could not resolve type"
         )
     | _ -> current_subst) in
  loop eqs []
```

単一化アルゴリズムの詳細

1. $\mathcal{U}\left(\{(\tau,\tau)\}\cup X'\right)=\mathcal{U}\left(X'\right)$

   同じ型がある場合は, 読み飛ばし, 次に進む.

   ```
   if x = y then loop rest current_subst else...
   ```

2. $U(\{(\tau_{11}\to\tau_{12},\tau_{21}\to\tau_{22})\}\uplus X')=U(\{(\tau_{11},\tau_{21}),(\tau_{12},\tau_{22})\}\uplus X'$

   2 つの Fun の入力型と出力型は一致していなければいけない.

   ```
   (match x, y with
    | TyFun(a, b), TyFun(c, d) -> loop ((a, c) :: (b, d) :: rest) current_subst
   ```

3. $U(\{(\alpha,\tau)\}\cup X')\quad(\text{ if }\tau\neq\alpha)=\begin{cases}U([\alpha\mapsto\tau]X')\circ[\alpha\mapsto\tau] & (\alpha\notin FTV(\tau))\\ error & (\alpha\in FTV(\tau))\end{cases}$

   まず,(α,τ) を, 制約を, 残りの型同値 X' に適用し, それを単一化する. その後, 単一化した型代入にこの制約を追加する. $(\alpha\in FTV(\tau)$ の場合にエラーを出力する理由については, 課題 4.3.4 で述べる.

   ```
   (match lst with
    | (x, y) :: rest ->
      if x = y then loop rest current_subst else
        (match x, y with
        ...
           | TyVar(id), b ->
   ```

```
            if not (MySet.member id (freevar_ty b)) then
              let mid = unify(subst_eqs (id, b) rest) in
              (id, b):: mid
            else err "unify: could not resolve type"
            | b, TyVar(id) ->
            if not (MySet.member id (freevar_ty b)) then
              let mid = unify(subst_eqs (id, b) rest) in
              (id, b) :: mid
            else err "unify: could not resolve type"
        ...
```

4.3.4[]

単一化アルゴリズムにおいて，$\alpha \in FTV(\tau)$ という条件はなぜ必要か考察せよ.

`fun x -> x x` の型推論過程について考えてみる.

まず,`ty_exp tyenv FunExp([x], AppExp(x, x))` が実行され, その後,x に新しい `TyVar('a` とする) を追加した環境 `eval_env` を用いて, `ty_exp eval_env AppExp(x, x)` が呼び出される.

ここで,

```
| AppExp(exp1, exp2) ->
    let ty_exp1, tysubst1 = ty_exp tyenv exp1 in
    let ty_exp2, tysubst2 = ty_exp tyenv exp2 in
    (* make new var *)
    let ty_x = TyVar(fresh_tyvar()) in
    let subst_main = unify([ty_exp1, TyFun(ty_exp2, ty_x)] @ eqls_of_subst tysubst1 @ eqls_of_subst tys
    ...
```

以上のコードより,`unify([('a, 'a -> 'b)])` が実行されるが, ここで,$\alpha \in FTV(\tau)$ の条件をチェックしていないと,

```
| TyVar(id), b ->
            if not (MySet.member id (freevar_ty b)) then
              let mid = unify(subst_eqs (id, b) rest) in
              (id, b):: mid
            else err "unify: could not resolve type"
```

このコードの if 分に入ることになるが, これでは,

```
('a, 'a -> 'b) ---> ('a, ('a -> 'b) -> 'b) --->
('a, (('a -> 'b) -> 'b) -> 'b) ---> ('a, ((('a -> 'b) -> 'b) -> 'b) -> 'b)
```

というように無限ループしてしまう.

これは, 型同値のペアの両方に, 同じ型変数が入っていることによるので, これは検出して, 無限ループを防がなければいけない.

4.3.5[]

ty_decl

ty_decl は,main.ml において型推論のために最初に呼び出される関数であり, 型環境と表現を受け取り, 型の評価結果を返す.

main.ml

```
let rec read_eval_print env tyenv =
  ...
  try
    ...
    let ty, new_tyenv = ty_decl tyenv decl in
    let (id, newenv, v) = eval_decl env decl in
    Printf.printf "val %s : " id;
    ...
  with e ->
    ...
```

typing.ml

```
let rec ty_decl tyenv = function
  | Exp e ->
    let (type_, _) = ty_exp tyenv e in
    (type_, tyenv)
  | Decl(id, e) ->
    let e_ty, _ = ty_decl tyenv (Exp e) in
    let new_tyenv = Environment.extend id e_ty tyenv in
    (e_ty, new_tyenv)
```

ty_decl は, 最初に表現を受け取ったら, それが宣言を伴わない表現である場合は, 以下に示す ty_exp に従って表現の型を評価する. 表現が宣言である場合は, 宣言の内容を ty_exp を用いて型推論し, それを現在の型環境に追加した新しい型環境を返す.

ty_exp

ty_exp は, 型環境 tyenv と, 評価したい表現を入力とする. 各表現ごとに, 処理を説明していく.

ILit, BLit

```
  | ILit _ -> (TyInt, [])
  | BLit _ -> (TyBool, [])
```

型としてそれぞれ TyBool, TyInt を返す. また, 型代入は返さなくてよい.

BinOp, LogicOp

```
let ty_prim op (ty1:ty) (ty2:ty) = match op with
  | Plus -> (TyInt, (ty1, TyInt) :: (ty2, TyInt) :: [])
  | Mult -> (TyInt, (ty1, TyInt) :: (ty2, TyInt) :: [])
  | Lt  -> (TyBool, (ty1, TyInt) :: (ty2, TyInt) :: [])

let rec ty_exp tyenv = function
    ...
    | BinOp (op, exp1, exp2) ->
        let tyarg1, tysubst1 = ty_exp tyenv exp1 in
        let tyarg2, tysubst2 = ty_exp tyenv exp2 in
        let ty3, eqs3 = ty_prim op tyarg1 tyarg2 in
        let eqs = (eqls_of_subst tysubst1) @ (eqls_of_subst tysubst2) @ eqs3    in
        let main_subst = unify eqs in
        (ty3, main_subst)
    ...
```

まず 2 つの引数の型と型代入を計算する. その後,ty_prim に 2 つの引数の型を代入し, 式全体の型と, 型同値の集合を得る. 最後に,2 つの式を単一化し, 型と型代入を返す.

LogicOp についても同様である.

IfExp

```
| IfExp (exp1, exp2, exp3) ->
    let tyarg1, tysubst1 = ty_exp tyenv exp1 in
    let cond_type = get_type tyarg1 in
    (* if condition part is valid *)
    if cond_type = "tybool" || cond_type = "tyvar" then
      let new_eqs = (tyarg1, TyBool) :: eqls_of_subst tysubst1 in
      let tyarg2, tysubst2 = ty_exp tyenv exp2 in
      let tyarg3, tysubst3 = ty_exp tyenv exp3 in
      let main_subst = unify ((eqls_of_subst tysubst2) @ (eqls_of_subst tysubst3) @ new_eqs @ [(tyarg2,
      (subst_type main_subst tyarg2, main_subst)
    else err "condition must be boolean: if"
```

まず条件部分を ty_exp で評価する. この型が TyBool または,TyVar である場合は, 条件部分が TyBool であるという制約を加え,2 つの実行部分を ty_exp に適用する. 最後に, この 2 つの型が等しいという制約とこれまでの型制約を単一化し, 得られた型代入を用いて式全体の型を再評価し, 返す.

MultiLetExp

```
| MultiLetExp (params, exp) ->
    let rec extend_envs_from_list current_tyenv current_subst p =
      match p with
      | (id, e) :: rest ->
        let e_type, e_subst = ty_exp tyenv e in
```

```
      let new_tyenv = Environment.extend id e_type current_tyenv in
      let new_subst = current_subst @ e_subst in
      extend_envs_from_list new_tyenv new_subst rest
    | [] -> current_tyenv, current_subst in
  let eval_tyenv, eval_subst = extend_envs_from_list tyenv [] params in
  let exp_ty, exp_subst = ty_exp eval_tyenv exp in
  let main_subst = unify (eqls_of_subst eval_subst @ eqls_of_subst exp_subst) in
  (subst_type main_subst exp_ty, main_subst)
```

MultiLetExp は,`let x1 = e1 and x2 = e2 and ... in e` の式などの複数 let 宣言 (単数を含む) を表し,`parser.mly` によって `MultiLetExp([(x1, e1), (x2, e2), ...], e)` と parse される.

1. 宣言部の各値を `ty_exp` によって評価した型を現在の型環境に加えた型環境 `eval_tyenv` と, それによって判明した型代入 `eval_subst` を求める.
2. 求めた `eval_tyenv` のもとで,`e` を型を評価する.
3. 1. の型代入と 2. の型代入を単一化し,`e` の型を再評価する.

FunExp

```
| FunExp(params, exp) ->
    let rec extend_envs_from_list current_env p =
      (match p with
       | id :: rest ->
         let new_tyvar = TyVar (fresh_tyvar()) in
         let new_env = Environment.extend id new_tyvar current_env in
         extend_envs_from_list new_env rest
       | [] -> current_env ) in
    (* get environment with new tyvar for each params to evaluate the main function *)
    let eval_tyenv = extend_envs_from_list tyenv params in
    (* evaluate main function in the created environment *)
    let res_ty, res_tysubst = ty_exp eval_tyenv exp in
    (* make output ( re-evaluate args ) *)
    let rec eval_type p e =
      (match p with
       | top :: rest ->
         (try
            let arg_tyvar = Environment.lookup top eval_tyenv in
            let arg_ty =  subst_type res_tysubst arg_tyvar in
            TyFun(arg_ty, eval_type rest e)
          with _ -> err "error in fun exp")
       | [] -> e) in
    (eval_type params res_ty, res_tysubst)
```

`FunExp` は,`fun x1 x2 x3 -> e` が,`FunExp([x1, x2, x3], e)` と評価される.

1. 各引数に新しい型変数 `TyVar(fresh_tyvar())` を割り当てて, 現在の型環境に追加した型環境 `eval_tyenv` を作成する.
2. `eval_tyenv` のもとで,`e` を評価する.
3. 2. で得られた型代入から,1. で振り当てた新しい型変数の型を評価し, 全体の型を得る.

AppExp

```
| AppExp(exp1, exp2) ->
    let ty_exp1, tysubst1 = ty_exp tyenv exp1 in
    let ty_exp2, tysubst2 = ty_exp tyenv exp2 in
    (* make new var *)
    let ty_x = TyVar(fresh_tyvar()) in
    let subst_main = unify([ty_exp1, TyFun(ty_exp2, ty_x)] @ eqls_of_subst tysubst1 @ eqls_of_subst tys
    let ty_3 = subst_type subst_main ty_x in
    (ty_3, subst_main)
```

1. 適用する表現と適用される表現の型をそれぞれ `ty_exp` で評価する.
2. 関数適用式全体の型を, 新しい型 `ty_x` で置く.
3. (ty_exp1, TyFun(ty_exp2, ty_x)) の型同値を 1. で得られた型代入と単一化し, この型代入で `ty_x` を再評価する.

実験の感想

ソフトウェア実験は, 日頃仲良くしているプログラムが, どのように処理されて実行されているかを知る良い機会となった.

また,ocaml を通じて, 静的な型検査, 型推論や, 関数型プログラミングの便利さを実感し,~~python が嫌いになってしまった.~~ 動的型検査, 手続き型言語を書くのが不快に感じるようになってしまった.