7 型推論

これまでの経験で，プログラムの中に型を書かなくても，ocaml がそれを自動的に推論してくれることが分かったはずである．

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

この推論によって，エラーが指摘されることもある．

```
# f 1.5;;
This expression has type float but is here used with type int
```

しかし，エラーの原因が分かりにくい場合もある．

```
# let g x = x in (g 1, g true);;
- : int * bool = (1, true)
# let f g = (g 1, g true);;
This expression has type bool but is here used with type int
# let rec nth n (x,l) =
    if n = 0 then x else f (n-1) l ;;
This expression has type 'a but is here used with type 'b * 'a
# let rec f x = g (x+1)
  and g x = x *. 2. ;;
This expression has type int but is here used with type float
# let rec f x = h (x+1)
  and g x = x *. 2.
  and h x = g x ;;
This expression has type int but is here used with type float
```

これを理解するのに，型推論がどうやって行われるかを (ある程度) 知らなければならない．

型の内部表現　型は木構造として見ることができる．

```
type etype =
    Var of string
  | Funct of etype * etype
  | Prod of etype list
  | Named of string * etype list

# let int = Named ("int", []) ;;
val int : etype = Named ("int", [])
# let list t = Named ("list", [t]) ;;
val list : etype -> etype
```

代入　型の中に変数があれば，それを具体的な型に変えることができる．

```
type bindings = (string * etype) list
let rec subst (bl : bindings) t =
  match t with
  | Var x  -> (try List.assoc x bl with Not_found -> t)
  | Funct (t1, t2) -> Funct (subst bl t1, subst bl t2)
  | Prod tl        -> Prod (List.map (subst bl) tl)
  | Named (s, tl)  -> Named (s, List.map (subst bl) tl)
```

```
val subst : bindings -> etype -> etype

# let t0 = Funct (Var "a", Var "b")
  let t0' = subst ["a", int] t0 ;;
val t0' : etype = Funct (Named ("int", []), Var "b")

let add_bindings bl bl' = bl @ List.map (fun (x, tx) -> (x, subst bl tx)) bl'
val subst_bindings : bindings -> (string * etype) list -> (string * etype) list
```

単一化　二つの型が同じでなければならないと分かったとき，それらを単一化する．両側が同じになるような代入を生成する

$$bl = \texttt{unify}\ t_1\ t_2 \quad \Rightarrow \quad \texttt{subst}\ bl\ t_1 = \texttt{subst}\ bl\ t_2$$

ただし，循環を含む代入を定義してはいけないので要注意.

```
let rec occur x t =
  match t with
    Var y -> if y = x then failwith "occur"
  | Funct (t1,t2) -> occur x t1; occur x t2
  | Prod tl | Named(_,tl) -> List.iter (occur x) tl
val occur : string -> etype -> unit
let rec unify t1 t2 =
  match (t1, t2) with
  | (Var x, Var y) when x = y -> []
  | (Var x, t) | (t, Var x)   -> occur x t; [x, t]
  | (Funct(u1,v1), Funct(u2,v2)) ->
      let bl = unify u1 u2 in
      let bl' = unify (subst bl v1) (subst bl v2) in
      add_bindings bl' bl
  | (Prod tl1, Prod tl2) -> unify_list [] tl1 tl2
  | (Named(s1,tl1), Named(s2,tl2)) when s1 = s2 ->
      unify_list [] tl1 tl2
  | _ -> failwith "unify"
and unify_list bl tl1 tl2 =
  match (tl1, tl2) with
    ([], []) -> bl
  | (t1::tl1, t2::tl2) ->
      let bl' = unify (subst bl t1) (subst bl t2) in
      unify_list (add_bindings bl' bl) tl1 tl2
  | _ -> failwith "unify_list"
val unify : etype -> etype -> bindings =
val unify_list : bindings -> etype list -> etype list -> bindings list

# let bl = unify t0 (Funct(int, Var "a")) ;;
val bl : bindings = [("b", Named ("int", [])); ("a", Named ("int", []))]
# let t0' = subst bl t0 ;;
val t0' : etype = Funct (Named ("int", []), Named ("int", []))
```

多相型　let で定義された関数は多相型を持つこともある．使う度に異なる型を得るために，変数を新しいものにする必要がある．そのために，あらかじめ多相にできる変数を調べておく．

```
let diff l1 l2 = List.filter (fun a -> not (List.mem a l2)) l1
val diff : 'a list -> 'a list -> 'a list
let union l1 l2 = diff l1 l2 @ l2
val union : 'a list -> 'a list -> 'a list
```

```
let rec free t =
  match t with
    Var x -> [x]
  | Funct (t1, t2) -> union (free t1) (free t2)
  | Prod tl | Named(_, tl) ->
      List.fold_left union [] (List.map free tl)
val free : etype -> string list
type env = (string * (string list * etype)) list
let free_env (env : env) =
let free_env env =
  List.fold_left union [] (List.map (fun (s,(vl,t)) -> diff (free t) vl) env)
val free_env : env -> string list
let generalize env t =
  let vl = diff (free t) (free_env env) in
  (vl, t)
val generalize : env -> etype -> string list * etype
let c = ref 0
let newvar s = assert (s <> ""); incr c; Printf.sprintf "%c%d" s.[0] !c
val newvar : string -> string
let instance (vl, t) =
  let bl = List.map (fun s -> (s, Var(newvar s))) vl in
  subst bl t
val instance : string list * etype -> etype
let t = instance (["a";"b"], Funct(Prod[Var"a";Var"b"], Var"a")) ;;
val t : etype = Funct (Prod [Var "a1"; Var "b2"], Var "a1")
```

型推論　プログラムに対して型推論を行なう．プログラムも木構造として見ることができる．

```
type expr =
    Const of int
  | Ident of string
  | Fun of string * expr
  | Tuple of expr list
  | Let of string * expr * expr
  | Letrec of (string * expr) list * expr
  | App of expr * expr

let subst_poly bl (vl,t) =
  let bl = List.filter (fun (x,_) -> not (List.mem x vl)) bl in
  (vl, subst bl t)
val subst_poly :
  (string * etype) list -> string list * etype -> string list * etype
let subst_env bl (env : env) : env =
  List.map (fun (x, pt) -> (x, subst_poly bl pt)) env
val subst_env : (string * etype) list -> env -> env
let rec typing env e =
  match e with
    Const _ -> ([], int)
  | Ident x -> ([], instance (List.assoc x env))
  | Fun (x, e) ->
      let v = newvar x in
      let (bl, t) = typing ((x,([],Var v))::env) e in
      (bl, Funct (subst bl (Var v), t))
  | Tuple el -> let (bl, tl) = type_list env [] [] el in (bl, Prod tl)
  | App (e1, e2) ->
      let (bl, t1) = typing env e1 in
      let (bl', t2) = typing (subst_env bl env) e2 in
      let bl = add_bindings bl' bl in
      let r = newvar "r" in
```

```
      let bl' = unify t1 (Funct(t2, Var r)) in
      (add_bindings bl' bl, Var r)
  | Let (x, e1, e2) ->
      let (bl, t1) = typing env e1 in
      let env = subst_env bl env in
      typing ((x, generalize env (subst bl t1))::env) e2
  | Letrec (xel, e) ->
      let env' = List.map (fun (x,e) -> (x,([],Var(newvar x)))) xel in
      let (bl,tl) = type_list (env' @ env) [] [] (List.map snd xel) in
      let bl =
        unify_list bl (List.map (subst bl) tl)
          (List.map (fun (_,(_,t)) -> subst bl t) env') in
      let env = subst_env bl env
      and env' = subst_env bl env' in
      let env'' = List.map (fun (x,(_,t)) -> (x, generalize env t)) env' in
      typing env'' e
and type_list env bl tl el =
  match el with
    [] -> (bl, tl)
  | e :: el ->
      let (bl', t)  = typing (subst_env bl env) e in
      type_list env (add_bindings bl' bl) (t :: tl) el
val typing : env -> expr -> bindings * etype
val type_list :
  env -> bindings -> etype list -> expr list -> bindings * etype list
# typing [] (Let("g", Fun("x",Ident"x"),
                 Tuple[App(Ident"g",Const 1);App(Ident"g",Tuple[])])) ;;
- : bindings * etype =
  ([("r7", Prod []); ("x6", Prod []); ("r5", Named ("int", []));
    ("x4", Named ("int", []))], Prod [Var "r5"; Var "r7"])
# typing [] (Letrec(["f", Fun("x", Fun("y", Tuple[App(Ident"g", Ident"x");
                                                  App(Ident"g", Ident"y")]));
                     "g", Fun("x", Ident"x")], Ident"f")) ;;
- : bindings * etype =
  ([], Funct (Var "r15", Funct (Var "r15", Prod [Var "r15"; Var "r15"])))
```

実習課題 (7 回目)

1. unify までのコードを入力し，様々な型を単一化してみよ．

2. ocamlbrowser で List.fold_left の意味を確認せよ．ヒント：List.fold_left の型を表示させてから，Intf ボタンを押してみる．

3. 今まで書いたプログラムを ocamlbrowser の File–Open コマンドで開き，Compiler–Typecheck で型推論を行い，内部の変数の型を確認してみよ．

4. expr 型の値 (プログラム) を計算する関数を定義せよ．
   `val interp : (string * expr) list -> expr -> expr`
   結果の外の部分は Const か Fun か Tuple でなければならない．以下の部分が与えられる．それ以外の場合を追加せよ．関数に自由変数がないと仮定してもいい，そして Letrec を無視してもいい (Tuple，App と Let だけをあつかえばいい)

   ```
   let interp env e =
     match e with
       Const _ | Fun _ -> e
     | Ident x -> List.assoc x env
     ...
   ```