4.4　多相的 let の型推論

前節までの実装で実現される型 (システム) は単相的であり，ひとつの変数をあたかも複数の型を持つように扱えない．例えば，

```
let f = fun x → x  in
if f true then f 2 else 3;;
```

のようなプログラムは，f が，if の条件部では bool→bool として，また，then 節では int→int として使われているため， 型推論に失敗してしまう．本節では，上記のプログラムなどを受理するよう **let 多相** (*let polymorphism*) を実装する．

本節を理解するためには OCaml の多相型の知識があったほうがよい．以下の二つのプログラムがどのように型付けされるか，あるいはされないかがよく分からないという読者は，OCaml 入門テキストの多相性に関する節，特に let 多相に関する節を復習してから，この先を読まれたい．

- `let id x = x in (id 3, id true)`
- `(fun id -> (id 3, id true)) (fun x -> x)`

4.4.1　多相性と型スキーム

OCaml で `let f = fun x -> x;;` とすると，その型は `'a -> 'a` であると表示される．しかし，ここで現れる型変数 `'a` は，後でその正体が判明する（今のところは）未知の型を表しているわけではなく,「どんな型にでも置き換えてよい」ことを示すための，いわば「穴ボコ」につけた名前である．そのために，`'a` を `int` で置き換えて `int->int` として扱って整数に適用できる関数の型としたり，`'a` を置き換えて `bool->bool` として真偽値に適用する関数の型としたりすることができる．このように，型変数には「今のところ未確定で後で正体が判明する型変数（**単相的** (*monomorphic*) な型変数)」と「どんな型にでも置き換えてよい型変数（**多相的** (*polymorphic*) な型変数)」の二種類があることに注意しよう．

この二種類を区別するために，多相的な型変数は $\forall\alpha.$ で束縛されるものとし，$\forall\alpha$ が型の前に付けられた表現を**型スキーム** (*type scheme*) と呼ぶことにする．より正確には，型 τ の前に有限個の $\forall\alpha$ が付けられた表現 $\forall\alpha_1.\forall\alpha_2.\ldots.\forall\alpha_n.\tau$ を型スキームと呼ぶ. (この型スキームを，型変数の列 $(\alpha_1,\ldots,\alpha_n)$ をベクトル表記を借りて $\vec{\alpha}$ と書くことにして，以下では $\forall\vec{\alpha}.\tau$ と書くことにする.）例えば $\forall\alpha.\alpha\to\alpha$ は型スキームである．

型スキーム $\forall\vec{\alpha}.\tau$ は，型変数の列 $\vec{\alpha}$ に相当する型を受け取って型を返す，いわば型から型への関数のようなものと見ることができる．例えば，型スキーム $\forall\alpha.\alpha\to\alpha$ は，型 int を受け取ったら型 int→int を返し，型 bool を受け取ったら型 bool→bool を返すものと見ることができる．このように見ると，上記のプログラムでは，

- let で f が束縛された場所で f に型スキーム $\forall\alpha.\alpha\to\alpha$ を割り当て，

- `if` の条件節の式 `f true` 中では割り当てられた型スキームに `bool` を与えることで `f` を `bool→bool` 型として使い，
- `then` 節の式 `f 2` 中では，割り当てられた型スキームに `int` を与えることで `f` を `int→int` 型として使う，

ことで `f` の多相的な振る舞いを捉えることができる．

より形式的には，型 τ と型スキーム σ の定義を以下のように変更する．

$$\begin{aligned} \tau &::= \alpha \mid \texttt{int} \mid \texttt{bool} \mid \tau_1 \to \tau_2 \\ \sigma &::= \tau \mid \forall \alpha.\sigma \end{aligned}$$

新しく導入された型スキーム σ が（上の説明の通り）型 τ の前に有限個の $\forall\alpha$ がついた形になっていることを確認されたい．また，型 τ は型スキームともみなせることに注意されたい．($\forall\alpha.$ がひとつもついていない型スキームである．）型スキーム中，$\forall$ のついている型変数を**束縛されている** (*bound*) といい，束縛されていない型変数（これらは単相的な型変数である）を**自由である** (*free*)，という． 例えば $\forall\alpha.\alpha\to\alpha\to\beta$ において，α は束縛されており，β は自由である．

その上で，型環境 Γ を（変数から型への部分関数ではなく）変数から型スキームへの部分関数とする．これにより，`let` で束縛された変数には型スキーム $\forall\vec{\alpha}.\tau$ を持たせておき，使用する際に $\vec{\alpha}$ を適切な型で置き換えることで，多相的な振る舞いを型システムの上で表現することができる．なお，[*AI: 尻切れ．*] ←

型と型スキームの区別 ここまでの説明から分かるように，これから導入する型システムでは型と型スキームを区別する．この区別は，技術的には，型に相当するメタ変数 τ と型スキームに相当するメタ変数 σ を区別していることから生じており，この区別のために $(\forall\alpha.\alpha)\to(\forall\alpha.\alpha)$ のような表現は型とはみなされないようになっている．

型と型スキームを区別して型システムを設計するのは，主に型推論問題の決定可能性の要請から来ている．

$$\tau ::= \alpha \mid \texttt{int} \mid \texttt{bool} \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau.$$

のように型スキームも型とみなせるように型を定義して型システムを設計すると，より多くのプログラムを型付け可能とすることができ，型システムの表現力は上がるのだが，素朴な同一視をするだけでは型推論問題が決定不能になることが知られている．型と型スキームを区別し（あとで見るように）多相性のある変数を導入できる場所を `let` や `let rec` に制限することで，実行時型エラーを含まない十分に多くのプログラムを型付け可能とすることができ，なおかつ型推論問題を決定可能とすることが可能となる．

図 4.1 上半分に型スキームの実装上の定義を示す．関数 `freevar_tysc` は，Exercise **??**で実装した関数 `freevar_ty` の拡張で，型スキーム σ を受け取り，σ に自由に出現する型変数の集合を計算する関数である．$\forall\vec{\alpha}$ が型変数列 $\vec{\alpha}$ を束縛するため，型スキーム $\forall\vec{\alpha}.\tau$ 中に出現する自由な型変数の集合は，型 τ 中に出現する型変数の集合から $\vec{\alpha}$ 中の型変数をすべて除いたものになる．

4.4.2 型付け規則の拡張

変数のための規則

次に，型付け規則をどのように拡張すればよいのか考えてみよう．型環境が変数から型スキームへの部分関数であることを思い出されたい．変数のための規則は以下の通りになっている．

$$\frac{(\Gamma(x) = \forall\alpha_1, \ldots, \alpha_n.\tau)}{\Gamma \vdash x : [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]\tau} \qquad \text{(T-PolyVar)}$$

すなわち，型環境中で x が束縛されている先の型スキームを $\forall\alpha_1, \ldots, \alpha_n.\tau$ とすると，τ 中の $\alpha_1, \ldots, \alpha_n$ を任意の型 $\tau_1, \ldots, \tau_n$ で置き換えて得られる型を x の型としてよい．（まさにこれがやりたかったことである．）例えば，$\Gamma(\mathsf{f}) = \forall\alpha.\alpha{\to}\alpha$ とすると，

$$\Gamma \vdash \mathsf{f} : \mathtt{int}{\to}\mathtt{int}$$

や

$$\Gamma \vdash \mathsf{f} : (\mathtt{int}{\to}\mathtt{int}){\to}(\mathtt{int}{\to}\mathtt{int})$$

といった型判断を導出することができる．[1]

let (rec) 式に関する規則

さて，let に関しては，大まかには以下のような規則になるはずである．

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \forall\alpha_1.\cdots\forall\alpha_n.\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \qquad \text{(T-PolyLet?)}$$

これは，e_1 の型から型スキームを作って，それを使って e_2 の型付けをすればいいことを示している．さて，残る問題は $\alpha_1, \ldots, \alpha_n$ としてどんな型変数を選べばよいかである．もちろん，τ_1 に現れる型変数に関して $\forall$ をつけて，「未知の型」から「任意の型」に役割変更をするのだが，どんな型変数でも変更してよいわけではない．役割変更してよいものは **Γ に自由に出現しないもの**である．Γ 中に (自由に) 現れる型変数は，その後の型推論の過程で正体がわかって特定の型に置き換えられる可能性があるので，任意におきかえられるものとみなしてはまずいのである．例えば，

$$\mathsf{let\ f\ x} = (\mathsf{(let\ g\ y} = \mathsf{(x,\ y)\ in\ g\ 4)},\ \mathsf{x} + 1)\ \mathsf{in}\ \ldots$$

という式を考え，その型推論の経過を書くと，

1. x の型を α とし，式 ((let g y = (x, y) in g 4), x + 1) の型推論をする．

[1] なお，あとで定義する型推論アルゴリズムは，プログラム全体に型が付くように適切な $\tau_1, \ldots, \tau_n$ を選ばなければならない．しかしながら，型付け規則を定義する段階においては，適切な $\tau_1, \ldots, \tau_n$ は何なのかを気にする必要はない．このように，型付け規則は数学的にきれいな形で書いておいて，型推論アルゴリズムで適切な $\tau_1, \ldots, \tau_n$ を選ぶ等の作業を頑張るといった役割分担は，型システム関係の文献では頻出である．

2. 第 1 要素の式 `let g y = (x, y) in g 4` の型推論を行う．このために，関数 `g` のパラメータ `y` の型を β とし，型推論を行う．関数の型として $\beta \to \alpha * \beta$ が得られる．

ここで，`g` は `let` で束縛されているので，推論された型 $\beta \to \alpha * \beta$ から型スキームを作り，g をこの型スキームに束縛する必要がある．ここで $\forall$ で束縛してよい（つまり多相的に使って良い）型変数はどれであろうか．

もし $\forall\alpha.\forall\beta.\beta \to \alpha * \beta$ のように α についてまで束縛して（$\forall$ をつけて）しまったとしよう．すると，関数 `f` は型 $\forall\alpha.\forall\beta.\beta \to \alpha * \beta$ を持つことになる．これは `f` に対してどのような型の引数でも渡せることを意味するから，`f true` といった式が `in` の後に書かれていても許されることになる．しかし，`f` の定義中に `x + 1` という式が現れているため，これでは `true + 1` を実行中に評価することになってしまい，実行時に型エラーが起こってしまう．

何がおかしかったのだろうか．α が `g` のスコープの外側で宣言されている `x` の型であったことである．スコープの外側で宣言されている変数の型は，その変数が外側でどのように使われているかに依存して決まるため，後になって特定の型にしなければならない場合がある．（実際にこの例では `x` が外側で `x+1` のように整数との加算に用いられているため，`x` の型を `int` としなければならないことが，後になって分かる．）そのため，$\forall$ をつけて多相性を持たせてはならないのである．というわけで，正しい型付け規則は，付帯条件をつけて，

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \forall\alpha_1.\cdots\forall\alpha_n.\tau_1 \vdash e_2 : \tau_2 \quad (\alpha_1, \ldots, \alpha_n \text{ は } \tau_1 \text{ に自由に出現する型変数で } \Gamma \text{ には自由に出現しない})}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \quad \text{(T-PolyLet)}$$

となる．「Γ に自由に出現しない」という条件で，スコープ外で宣言された変数の型として使われている型変数が多相性を持たないように制限している．

4.4.3 型推論アルゴリズム概要

ここまでのところが理解できれば，実は型推論の実装に対する変更はそんなに多くはない．メジャーな変更が必要なのは変数式に関するケースと let 式に関するケースである．図 4.2 にコードの変更点を示す．

まず，変数式に関するケースを考えよう．型変数に代入する型（型付け規則中の $\tau_1, \ldots, \tau_n$）はこの時点では未知であり，変数が他の部分でどう使われるかに依存して決定される．そのため，ここでは $\tau_1, \ldots, \tau_n$ に相当する新しい型変数を用意し，それらを使って具体化を行う．

次に let 式のケースである．ここでは，e_1 の型推論で得られた e_1 の型 τ を型スキーム化する必要がある．型スキームする際には，T-PolyLet の付帯条件を満たすように多相性を持たせる型変数を決定する必要がある．この計算を行うための補助関数として図 4.2 中で `closure` を定義している．これは，型 τ と型環境 Γ と型代入 S から，条件「$\alpha_1, \ldots, \alpha_n$ は τ に自由に出現する型変数で $S\Gamma$ には自由に出現しない」を満たす型スキーム $\forall\alpha_1.\cdots.\forall\alpha_n.\tau$ を求める関数である．型代入 S を引数に取るのは，型推論の実装に便利なためである．

syntax.ml

```
    (* type scheme *)
    type tysc = TyScheme of tyvar list * ty

    let tysc_of_ty ty = TyScheme ([], ty)

    let freevar_tysc tysc = ...
```

main.ml

```
    ...
    let rec read_eval_print env tyenv =
      print_string "# ";
      flush stdout;
      let decl = Parser.toplevel Lexer.main (Lexing.from_channel stdin) in
      let (newtyenv, ty) = ty_decl tyenv decl in
      let (id, newenv, v) = eval_decl env decl in
        Printf.printf "val %s : " id;
        pp_ty ty;
        print_string " = ";
        pp_val v;
        print_newline();
        read_eval_print newenv newtyenv
```

図 4.1: 多相的 let のための型推論の実装 (1)

Exercise 4.4.1 [⋆⋆] 図 4.1, 4.2 を参考にして，多相的 let 式・宣言ともに扱える型推論アルゴリズムの実装を完成させよ．

Exercise 4.4.2 [⋆] 以下の型付け規則を参考にして，再帰関数が多相的に扱えるように，型推論機能を拡張せよ．

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e_1 : \tau_2 \qquad \Gamma, f : \forall \alpha_1, \ldots, \alpha_n.\tau_1 \to \tau_2 \vdash e_2 : \tau \quad (\alpha_1, \ldots, \alpha_n \text{ は } \tau_1 \text{ もしくは } \tau_2 \text{ に自由に出現する型変数で } \Gamma \text{ には自由に出現しない})}{\Gamma \vdash \mathsf{let\ rec}\ f = \mathsf{fun}\ x \to e_1\ \mathsf{in}\ e_2 : \tau} \quad \text{(T-PolyLetRec)}$$

Exercise 4.4.3 [⋆⋆⋆] OCaml では，「: ⟨型⟩」という形式で，式や宣言された変数の型を指定することができる．この機能を扱えるように処理系を拡張せよ．

Exercise 4.4.4 [⋆⋆⋆] 型推論時のエラー処理を，プログラマにエラー箇所がわかりやすくなるように改善せよ．

typing.ml

```
type tyenv = tysc Environment.t

let rec freevar_tyenv tyenv = ...

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
    TyScheme (MySet.to_list ids, ty)

let rec subst_type subst = ...

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 ty_decl tyenv = function
    Exp e -> let (_, ty) = ty_exp tyenv e in (tyenv, ty)
  | Decl (id, e) -> ...
```

図 4.2: 多相的 let のための型推論の実装 (2)