Functional and logic programming lab 7th report

Yoshiki Fujiwara, 05-191023

1　Q1: 型がつくものだけを評価

1.1　動作例

Code 1　動作例

```
Check if the constant works
# 1;;
- = 1
# true;;
- = true

Arithmetic operations
# 1 + true;;
Error = Unmatched type
# 1 - false;;
Error = Unmatched type
# 2 * true;;
Error = Unmatched type
# 3 / false;;
Error = Unmatched type
# 1 + 2;;
int
- = 3
# 2 * 3;;
int
- = 6
# 4 - 2;;
```

```
int
- = 2
# 4 / 2;;
int
- = 2

let expressions
# let x = 2;;
int
x = 2
# x;;
int
- = 2
# let y = true;;
bool
y = true
# y;;
bool
- = true
# x + y;;
Error = Unmatched type

if expressions
# if 1 then 2 else 3;;
Error = Unmatched type
# if true then 2 else false
;;
Error = Unmatched type
# if true then 2 else 4;;
int
- = 2文

fun
# fun x -> x + 2;;
(int -> int)
- = <fun>
# fun x -> 2;;
```

```
('a2 -> int)
- = <fun>

function applications
# let f = fun x -> x + 2;;
(int -> int)
f = <fun>
# f 3;;
int
- = 5
# f false;;
Error = Unmatched type

let rec in expressions
# let rec fact n = if (n = 0 ) then 1 else n * fact (n-1) in fact 5;;
int
- = 120
# let rec fact n = if (n = 0 ) then 1 else n * fact (n-1) in fact true;;
Error = Unmatched type


let rec expressions
# let rec fact n = if n = 0 then 1 else n * fact ( n - 1);;
int -> int
- = <fun>
# fact 5;;
int
- = 120
# fact true;;
Error = Unmatched type
```

1.2 考察

1.2.1 union 関数について

まず、この問題の実装を行うために union 関数を実装した。この関数は yuple のリストを二つ受け取ってそれの和集合を返す関数である。

1.2.2 infer_expr 関数について

infer_expr は、現在の型環境と expression を受け取って、その expression の返す型と制約を返す関数である。主にこの部分の定義についてまず考察していく。

四則演算の場合、ここで返す値は Int 型なので、返り値の第一要素は TyInt である。また、制約があり、その制約は、四則演算の前後の expr が、int 型でなければならないことである。よって、let v1 = infer_expr tenv e1 のように評価を行い、その第一要素で制約をかける。(fst(v1),TyInt)。また、その式の中の制約も受け継がなければならないので、union を用いて受け継ぐ。

EEq（equal expression) について、ここで返す型は bool 型なので、返り値の第一要素は TyBool である。また、制約については、比べるもの同士の型が等しいことである、それに=前後の expression の制約を union する。

EIf (if expression) について、まず、e1,e2,e3 の expression を評価する。ここで返す型は、e2 の返す型であるから、返り値の第一要素は、fst(v2) となる。If 文であることに注意すると、e1 の評価結果は Bool 型でならなければならず、また、e2 と e3 の評価結果の型は同じでなければならないことを制約として加える。

ELet(let expression) について、まず、今の環境のもとで、e1 の型と制約を求める。ここでできた環境 (newtenv = (e1,fst(v1)) :: tenv) のもとで、e2 を評価する。返り値の第一要素はその評価結果の第一要素であるし、返り値の第二要素は、e1 と e2 の評価でできた制約を union したものである。

EFun (function declaration) についてまず、新たな型変数 newvar を導入する。現在の型環境に e1 と newvar との対応を付け加える。そして、そのもとで、e2 を評価すれば良い。全体の型は、関数の形で、(newvar → e2 の評価結果の型) となる。

EApp (function application) について、まず、現在の型環境でそれぞれの expression について型と制約を求める。次に新たな型変数 newvar を導入する。返り値の第一要素は newvar であり、制約として、((e1 の評価結果の型) = (e2 の評価結果の型) → (newvar)) を与える。

ELetRec(let rec in expressions) について新しい型変数を導入する。それを newvar1 と newvar2 とする。現在の型環境に、f と newvar→newvar2 を対応づけたもの、x と newvar1 を対応づけたものを追加する。その環境で、e3(再帰関数の expression) の型と制約を求める。また、それとは別に f と newvar1 → newvar2 を対応づけたものを追加した環境で、e4(再帰関数の application) の型と制約を求める。式全体の評価は関数の返り値の型と、全ての制約を union したものになる。

1.2.3 eval_command の部分

上記で、型推論は行えるようになったので、実際にそれが合っているか否かの判定を行う関数を作成する。それが、infer_command 関数である。infer_command 関数は型が合っているかの検査を行い、あっていなかったら TyError を返す関数である。TyError の catch は eval_command で行う。

CExp の際は、eval_command に何も工夫はいらず、infer_command で検査を行う。infer に推論の結果を入れ、型制約を解く。返り値はあとで再帰関数を定義する時のために、pair にしているが、同じ要素を入れている。

CDecl の時と、CRecDecl の際は、eval_command 内で、tenv を変更する必要がある。つまり、変数の型を格納しなければならない。今、CDecl では一変数しか入らないため、type_result、すなわち、infer_command の結果を現在の環境に追加すれば良い。

CRecDecl の時は、infer_command の中で、再帰関数の処理を行わなければならない。lookup newvar1 (snd(infer)) で引数の型をとる。そして、pair で返すようにしている。なぜなら、関数の型環境の追加を行うためには、引数の型と返り値の型の両方が必要であるからである。

これによって、let rec の文で関数が定義できるようになり、また、関数評価の際に、適切でない型が来ることを防ぐことができる。

1.2.4 main.ml の変更

main.ml では、tenv という変数を作った。これは、let 文が型環境を維持できるようにするためである。これを行わないと、let x = 2;; x ;; で unbound variable の error をはいてしまう。

2 型がついても良さそうだが型がつかないプログラム

2.1 解答 1

let y = fun f $\rightarrow$ $((funx \rightarrow f(xx))(funx \rightarrow f(xx)))$;;

これは型がつきそうである。なぜなら、型推論の木が書けるからである。しかし、型の方程式を解くときに、$\alpha = \alpha \rightarrow (int \rightarrow int)$ を解かなければならず（これについては後述）、ocaml では解決できない。これは Y コンビネータと呼ばれるものである。型なしの世界では Y コンビネータの型付けはできるが、型ありの世界では型がつけられない。よっ

てこのように ocaml で解決できない。Ocaml でこれと同じ機能を持った関数を実装するためには、fix 関数を使えば良い。（第一回のスライドの fix 関数）

ここでの議論をもう少し詳しく書く。

例えば fibonacci 関数を考える。不動点コンビネータ（関数からその関数の不動点への関数）の一つである Y コンビネータについて考える。Y コンビネータは以下のように表せる。

$$\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

(x_1, x_2) の関数適用の部分について、x_1の型は、$\alpha \to (int \to int)$、x_2の型はα である。ここで、 $\alpha = \alpha \to (int \to int)$ が要求される。

2.2 解答 2

let fix g = (fun f $\to g(funx \to ffx))(funf \to g(funx \to ffx))$

これも同様の理由で定義できない例である。