Functional and logic programming lab 9th report

Yoshiki Fujiwara, 05-191023

課題 1,2,3 を 1-3 という名前の folder にまとめて実装した。

1 Q.1-Q.3: 組・リスト

1.1 解答

Code 1 動作例

```
Pair
# (2,3);;
int*int
- = (2,3)
# (2,true);;
int*bool
- = (2,true)
# (2,(true,false));;
int*bool *bool
- = (2,(true,false))

list
# [];;
a1list
- = []

# 1::[];;
intlist
- = VCons
# 2::(true::[]);;
Error = Unmatched type
```

```
# true :: (false :: (false :: []));;
bool list
- = VCons


# let a = match 1 with 1 -> 2 | 2 -> 1;;
int
a = 2
# let a = match 1 with 1 -> 2 | 2 -> true;;
Error = Unmatched type
# let a = match 1 with 1 -> 2 | true -> false ;;
Error = Unmatched type
```

1.2 考察

1.2.1 Q1 の考察

まず、Q1 について考察する。Q1 は Pair と空リストと cons について実装する課題である。まず、Syntax.ml も type expression に Pair を表す EPair と空リストを表す ENil と、cons を表す ECons の型を追加した。また、value 型にも同様に、VPair、VNil、VCons を追加した。それに応じて、tySyntax.ml の中の Print_type も変更した。

最後に eval.ml の eval_expr で、追加した expression に対する評価を記述する。EPair がきた場合は、VPair の一つ一つに対する評価を行えば良い。

VPair((eval_expr env e1),(eval_expr env e2))。ECons についても同様で、ENil については VNil を返せば良い。

1.2.2 Q2 の考察

次に、Q2 について考察する。Q2 では、Pattern match の評価を行えるようにした。

match 文を表現する expression の EMatch を新たに定義した。

まず、Pattern の照合を行う関数、find_match 関数を定義する。これは Pattern と照合する値を受け取って、成功すれば追加する環境を返し、失敗すれば None を返す関数である。この関数を使って match 文を評価する。

次に、eval_expr の中について説明する。match させる expression をまず評価する。また、search_match という、先ほど定義した find_match を pattern それぞれについて行う関数を定義した。

ここでは前のパターンから順に照合させていき、最後まで照合に成功しなければ

EValErr を raise する。パターンの照合に成功すれば、結果を環境に追加して、expression を評価する。

1.2.3 Q3 の考察

最後に、Q3 について考察する。Q3 では型推論を行なった。

まず、pair の型推論について考察する。Pair はそれぞれの要素を評価して、その評価の家庭で出てきた制約の union を取れば良い。Cons の型推論については、要素として追加する第一要素を list にしたものは第二要素の list の型と一致している必要がある。よってその制約と、式の評価途中の制約を union すれば良い。

EMatch についての型推論を行う。

そのために、まず、find_match_type という関数を作った。この関数は、パターンを受け取って、パターンの型と制約と追加される型環境を返す関数である。この関数の作成はスライド通りに行った。注意すべきは変数パターンと cons パターンである。変数パターンでは新しい型変数、それを型環境に追加しなければならない。cons パターンでは、新しい変数を導入して、、制約に list の肩が等しいということを追加しなければならない。

本題の infer_expr について考察する。まず、match の前の型と制約を求める。これを v1 とする。そして、新しい型変数 newvar を導入する。各パターンについて、find_match_pattern を実行して、パターン pi, 制約 Ci, 追加される環境 Γi を求める。

環境 (この環境は infer し始めた時の環境) を Γi で拡張した後、パターンに紐づけられている *expression* を型推論する。それを vi とする。そして、(union [(newvar, (fst(vi)))] (union [((fst(v1)), ti)](union (snd(vi)) (union ci const)))) として、制約を拡張していくことを繰り返す。($v1$ と vi が混同しやすいのでもう一度記述すると、$v1$ は *match* 文の *with* の前の *expression* の評価結果で、vi は各パターンの *expression* の評価結果。) この繰り返しによって、スライドに書かれている制約を得ることができる。

最後に、unification アルゴリズムを少し変更する。pair と list について要素同士の統合に unification すればいいので、constraintSolver.ml のに記述したような実装になる。

以上の実装を行えば型推論が行われるようになる。

2　Q.4: 名前呼び

2.1　解答

実際に名前呼びが行われたかどうかを確認するため、EAdd が呼ばれると expression を出力するようにした。以下の動作例には四つの場合を記した。

1) (fun x → x * x) (2 + 3) を名前呼びで実行した場合。

2)(fun x → x * x) (2 + 3) を値呼びで実行した場合。

3) fact(1+0) を名前呼びで実行した場合。

4) fact(1+0) を値呼びで実行した場合。

1) の場合と 2) の場合を比べると、名前呼びで実行した場合に、(2+3) の計算が二度されていることがわかる。EAdd が二回行われているからである。よって、名前呼びにすることに成功しているとわかる。

3) と 4) を比べる。名前呼びで実行した場合では、n の値が調べられるたびに EAdd が実行されている。これもまた、名前呼びに成功している証拠となっている。

Code 2　動作例

```
1) execute (fun x -> x * x) (2 + 3) in call by name
# (fun x -> x * x) (2 + 3);;
int
EAdd (2,3)
EAdd (2,3)
- = 25

2) execute (fun x -> x * x) (2 + 3) in call by value
# (fun x -> x * x) (2 + 3);;
int
EAdd (2,3)
- = 25

3)execute fact (0+1) in call by name
# let rec fact n = if n = 0 then 1 else n * fact(n-1) in fact (1 + 0);;
int
EAdd (1,0)
EAdd (1,0)
```

```
EAdd (1,0)
- = 1

4)execute fact (0+1) in call by value
# let rec fact n = if n = 0 then 1 else n * fact(n-1) in fact (1 + 0);;
int
EAdd (1,0)
- = 1
```

2.2 考察

値呼びにするために、再帰でない関数と再帰関数の文について実装を行なった。

2.2.1 再帰でない関数

Thunk を用いて実装を行なった。Thunk の型は、expr * env である。これは環境に入れることになり、環境の型は、(var * thunk) list である。まず、変数をどう eval すればいいかについて書く。変数が来た時は、変数に対応する thunk を探す。この時、thunk の中身は環境と式であることに注意する。値を返さなければならないので、この時に評価が行われる。thunk の中に入っている環境で、thunk の中の式を評価することを行えば良い。

次に、関数適用をどう行うかについて書く。e1 e2 の形で関数適用が行われるとする。まず、e1 を評価すると、e1 は関数の式なので、VFun が出てくるはずである。それを VFun(x1,expr1,env_thunk) とする。e2 を thunk にする。そのさいの環境は元の環境であることに注意する。この thunk を thunk1 とする。thunk1 を関数の引数 x に対応させる。(extend x1 thunk1 env_thunk) がそれに当たる。そのもとで、expr1 を評価すれば、名前呼びが実現できる。

2.2.2 再帰関数

まず、再帰関数の適用部分については、f の環境を拡張して、その環境で評価する値を評価すれば良い。今回の実装でこの新しい環境は newenv として定義した。

f には Thunk を対応させる。f は、今の環境で Thunk(ERecFun(....)) を評価したものとなる。ここの env がなぜ今の環境で良いかというと、評価の時に環境の拡張を行えば良いからである。また、ここで新しく ERecFun というものが追加された。ERecFun は、VRecFun を expression 型と紐づけるために新たに作成したものである。

最後に VRecFun をどのように評価するか説明する。今回の実装では、この部分で

環境を拡張しているため、この部分が再帰関数を実現させる肝となっている。まず、thunk1 という再帰関数 closure を作成する。この closure には再帰関数を表す ERecFun と VRecFun に入っていた env(env_thunk) を入れている。あとは、apply する引数部分を現在の環境で評価する closure を作って環境に追加して評価すれば良い。

3 Q.5: 必要呼び

3.1 動作例

前問と同じく、EAdd を呼び出した回数を測るために、EAdd の部分で print_expr を行なった。

Code 3 動作例

```
# (fun x -> x * x) (2 + 3);;
int
EAdd (2,3)
- = 25
# let rec fact n = if n = 0 then 1 else n * fact(n-1) in fact (1 + 0);;
int
EAdd (1,0)
- = 1
```

このコードは、値呼びと同じ EAdd の呼び出し回数となっているので、必要呼びに成功しているであろうとわかる。

3.2 考察

必要呼びは名前呼びから少し変更するだけで実装できる。変更する点は以下の 2 点である。まず一つ目は、環境の型の変更。二つ目は、変数を評価するときに、式であれば評価を実行して、値を保存し直すことである。

環境の型を変更するのは、式と環境の pair である dthunk と、値を同時に持ちたいからである。一度評価された thunk がもう一度評価されることはない。よって型は

env = (name * dval ref) list and dval = DThunk of expr * env | DVal of value

となる。

次に、変数評価についてであるが、変数 x に対する dval ref を環境から見つける。それを searched として、searched が値であればそのまま返し、searched が thunk であれば、

thunk の中身を評価して、環境を値に更新して、値を返すことをすれば良い。

以上の工夫を名前呼びに施すことで完成する。

4 Q6: インタプリタの機能拡張

4.1 動作例

前問と同じく、EAdd を呼び出した回数を測るために、EAdd の部分で print_expr を行なった。

Code 4 動作例

```
# let a = ref 2;;
val a : int ref {contents = 2}
# (!a);;
- : int ref = 2
# a := true;;
Error: This expression has type bool but an expression was expected of type in
# a := 3;;
- : int = 3
# a;;
- : int ref = {contents = 3}
# 1::2::[];;
- : intlist = [1;2]
```

4.2 考察

実装したものを順に書いていく。

4.2.1 エラー処理

エラー処理の改良を行なった。

まず、parser がエラーを起こした時にどの部分でエラーを起こしたのか表示してくれるようにした。これは parser.mly の一部にエラーが起きた時の実装を行うことで実装ができた。

また、型エラーが起きた時に、どの型を予測していたのにどの型が来たのかという情報をだすようにした。Ocaml の記法に習って This expression has ... のように出力した。

次に、環境に指定された変数がなかった時に、Unbound error が起きるが、その時にどの変数が足りないのか表示するようにした。

Error の色の変更も行なった。色のコードを調べ、Printf.printf 関数の中に、埋め込むことで色の変更ができる。

細かいことだが、Ocaml の version 表示も行うようにした。今回提出した interpreter の出力は Ocaml の出力にできる限り即している。

4.2.2 list, cons の出力

こちらも出力の話になるが、list と cons の出力を行えるようにした。print_list と print_value を相互再帰で実装する。list は [] で囲わなければならないので、最初の [を print_value で実装し、最後の] を print_list で実装した。

4.3 ref の実装

reference を interpreter で解釈できるように実装した。ref として実装したのは、let 文による reference の宣言、exclamation mark による、reference の contents の取り出し、値の再代入の三つである。

まず、parser と lexer を変更する。上記三つの構文をそれぞれ、CRef, ERefSub, RefExcalim を実装した。

次に、reference の型と reference を表す value を作る。それぞれ TyRef と VRef である。VRef は値の再代入が行えるように、value ref の型にした。

それぞれの実装について解説する。

まず、let 文による reference の宣言を表す CRef は、ref に格納する値を評価して、そのあとに、それを VRef の中に入れて環境に追加すればよい。ここで、前までの実装と、値を print する場所が異なっているが、これは Ocaml の出力に即するために、contents の部分を表示させる必要があったためである。

次に、代入を表す ERefSub について、環境の中から、VRef を探して来て、その値を変えればよい。値を変えることができるのは、環境に入れているのが reference であるからである。

ERefExclam については環境から値を見つけて取ることをすればよい。

最後に、型検査が通るようにしなければならない。基本的には TyRef の型として型環境に追加していけばよい。ただ、値の再代入の時は注意が必要である。再代入する際に reference に入れられていたものと、代入するものの型が異なれば、型エラーを出すべき

である。よってこれを制約に加えなければならないことに注意する。

以上のようにして、reference を実装した。