Functional and logic programming lab 6th report

Yoshiki Fujiwara, 05-191023

1 Q1: fun 文

1.1 動作例

Code 1 動作例

```
# fun x -> x + 2;;
- = <fun>
# (fun x -> x + 2) 2;;
- = 4
```

1.2 考察

ここでは関数が定義できるようにすることと、関数を適応できるようにすることを行った。

例題で行ったように、VFun を name * expr * (name * value) list の型で定義する。

この問題では主に、eval.ml の部分を変更する。

1.2.1 関数の適用

この問題ではこちらから考えた方が考えやすいので、こちらから考察する。関数の適用については EApp を実装すれば良い。まず、関数と引数の expr を評価する。そのあとに、関数評価をする。ここで、関数を評価する環境は関数を宣言した時の環境であることに注意する。評価は、変数の宣言と同様のことを行った環境で評価するだけで良い。

1.2.2 関数の宣言

関数の宣言については、EFun の部分について実装を行えば良い。EFun について評価が行われたあとは、CExp にいくことが構文木 (parser.mly) からわかる。今は fun の宣

言ができるようにすればいいので、型を合わせるだけで良い。

最後に、print できるようにするために、VFun が来た時に”fun”を出力するというコードを書けば良い。

2 Q2: 再帰関数の実装（参照を用いた場合）

2.1 動作例

Code 2 動作例

```
# let rec fact n = if n = 0 then 1 else fact ( n - 1 ) * n;;
- = <fun>
# fact 3;;
- = 6
# let rec fact n = if n = 0 then 1 else fact ( n - 1 ) * n in fact 3;;
- = 6
```

2.2 考察

まず、再帰関数 closure の生成を行う。これは eval.ml の ELetRec で行われている。まず、ダミーの環境 oenv を作り、v = VFun (x,e1,oenv) として（この時点では oenv は [ ] のポインタ)、oenv := extend f v env; と eval_expr (extend f v env) e2 を実行する。oenv := extend f v env; によって、v の中の oenv が指す内容も変化する。なぜなら、oenv は reference を用いて宣言しているからである。これによって再帰の構造ができた。

次に再帰関数の評価についてみる。再帰関数の評価は EApp で行われている。評価の時には再帰関数の引数を対応させるだけで良い。なぜなら、oenv に再帰関数の環境が入っているからである。これは Q4 でみる方法と大きく違う点である。

3 Q3: 相互再帰

3.1 動作例

Code 3 動作例

```
# let rec even n =
  if n = 0 then true
  else odd (n-1)
```

```
    and
    odd n =
    if n = 0 then false
    else even (n-1)
;;
  - = <fun>
# even 2;;
  - = true
# odd 5;;
  - = true
# even 3;;
  - = false
# odd 4;;
  - = false

# let rec even n =
    if n = 0 then true
    else odd (n-1)
    and
    odd n =
    if n = 0 then false
    else even (n-1)
    in
    even 2;;
    - = true
```

3.2 考察

参照を用いて循環的 closure を作成する方法を用いて実装した。循環的 closure を作成するには三つのステップがある。まず、ダミー環境の oenv を作り、次に、各宣言につき、closure を作り、最後に、oenv の環境を (f1,v1)::(f2,v2):: ... ::(fn,vn)::env に書き換えれば良い。

まず、宣言を順番に評価できるように、list を順に評価できるような関数を match 文で作る。それが eval.ml の中の update 関数に相当する。update 関数で行いたいことは oenv を書き換えていくことである。参照を用いているため、v1 から評価したとして、v1 の評価直後 (v1 = VFun (x1, e1, oenv)) は、vn の環境が入っていないように見える。だが、参照を用いているため、vn の評価によって、oenv が書き換えられると、その影響を

受けることになる。

こうすることで、相互再帰に対応できるようになる。

4 Q4: 関数適用の工夫を用いる方法

4.1 動作例

Code 4 動作例

```
# let rec fact n = if n = 0 then 1 else fact ( n - 1 ) * n;;
- = <fun>
# let rec fact n = if n = 0 then 1 else fact ( n - 1 ) * n in fact 4;;
- = 24
```

4.2 考察

まず、再帰関数 closure の生成を行う。この実装方法では、生成の部分ではなく、評価の時に複雑な処理を行うことで再帰関数を実現しているため、生成の部分はとてもシンプルである。具体的には、元々の環境に、関数名 f と、新しく作った value 型 VRecFun の対応を付け加えるだけである。

再帰関数の関数適用は、EApp で行われる。再帰関数に対して、VRecFun を対応づけた環境に、変数の対応づけを行う

env2 = extend x v2 (extend f (VRecFun(f,x,e,oenv)) oenv)

そして、この環境の下で評価を行う。

評価をその場で行わない関数宣言は、CRetDecl で行った。これは、この考察の最初に述べた再帰関数 closure の生成と同じことをしている。

5 Q5: 関数定義のための略記法

5.1 動作例

Code 5 動作例

```
# ( fun x y z -> x * y - z );;
- = <fun>
# ( fun x y z -> x * y - z ) 5 4 2;;
- = 18
```

```
# let f x y = x + y in f 5 2;;
- = 7
```

5.2 考察

1 をベースに実装を行った。引数をリストにすれば良い。

引数をリストにするために、parser.mly で lvar と lexpr を定義した。var を lvar に変えたため、CDecl も変更することになった。CDecl については、変数が複数のリストを受け取りたくないので、複数受け取った場合はエラーが起きるようにした。

大きく変更すべきなのは、EApp と ELet である。

まず、EApp については、関数適用時の引数の一つ一つを対応づけていくという形を取れば良い。変数はリストで受け取り、変数に代入する expression もリストで受け取っている。そのリストをうまく処理する関数が envapp として定義されているものである。envapp は 1 で行ったことと同じことをリストで行うだけである。

次に、ELet について。ELet は変数をリストとして受け取るが、その変数リストの第一要素は関数名である。なので、第一要素とそれ以外に分けて、VFun の然るべき場所にそれぞれ入れれば良い。環境、(extend f (VFun(rest,e2,env)) env) で IN 以降の expression を評価することになる。

6 Q6: 値の再帰的定義を使った再帰関数の実装

6.1 動作例

Code 6 動作例

```
# let rec fact n = if n = 0 then 1 else n * fact (n-1);;
- = <fun>
# let rec fact2 n = if n = 0 then 1 else n * fact (n-1);;
- = <fun>
# let a = 10;;
a = 10
# let rec fact3 n = if n = 0 then a else n * fact3 (n-1);;
- = <fun>
# fact3 3;;
- = 60
# let a = 3;;
a = 3
```

```
# fact3 2;;
- = 20
# a;;
- = 3
```

6.2 考察

let rec oenv = (f, VFun(x,e1,oenv)) :: env として、oenv を定義して、その環境で関数の適用を行えば良い。

このコードであれば、oenv には f と VFun の対応関係が入っていて、再帰できるようになっている。

7 Q7: 動的束縛の関数定義

7.1 動作例

Code 7 動作例

```
# let a = 10;;
a = 10
# let f = (dfun x -> x + a);;
f = <fun>
# let a = 20;;
a = 20
# f 10;;
- = 30
# let rec fact n = if n = 0 then 1 else n * fact (n-1);;
- = <fun>
# fact 5;;
- = 120
```

7.2 考察

まず、動的束縛の関数を定義する。そのための DVFun を作った。DVFun の関数適用は eval_expr (extend x v2 env) e　のように行い、関数定義時の環境ではなく、関数適用時の環境をベースに行うことに注意する。この工夫によって動的束縛の関数が書ける。

次に、動的束縛を用いて、再帰関数を定義する。動的束縛の場合、関数適応時の環境で関数を評価するので、newenv = extend f (DVFun(x,e1)) env とした、newenv の環境で

は、いつでも再帰関数の評価を行うことができる。それは DVFun (x, e) → eval_expr (extend x v2 env) e の部分から明らかである。(動的束縛では関数定義の環境 (f が入っていない) に依存しない。)

一部の言語では動的束縛が用いられている。再帰関数も letrec などと区別して書く必要がない。静的束縛を用いるか動的束縛を用いるかは言語によって異なる。それは言語デザインの段階で決まっていて、それはその言語の目的に依存している。python で今回の Q7 と同様のコードを実装すると同じ結果となる。