プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 5： 一階の関数型言語

これから現れる言語たち

- micro ML (4 章): 一階、関数型言語
- 名前なし (5 章): 高階、関数型言語
- micro C (7 章): 命令型言語
- オブジェクト指向言語

micro ML

Sestoft テキストでの micro ML:

- ML や F#の小さなサブセット
- 一階の言語 (not 高階言語)
- 関数型言語 (not 命令型言語)

一階 (first-order) vs 高階 (higher-order)

- 高階関数: 関数をもらったり、関数を返したりする関数
- 一階関数: 高階でない関数 (数をもらって、文字列を返す関数など)

micro ML の構文

```
type expr =
  | CstI of int   | CstB of bool    | Var of string
  | Let of string * expr * expr
  | Prim of string * expr * expr
  | If of expr * expr * expr
  | Letfun of string * string * expr * expr
  | Call of expr * expr
```

ただし、`Call(e1,e2)` において `e1` は、`Var s` の形でなければいけない。
(「一階言語である」ことの制限)

```
CstB true
If(Prim("=",Var "x",CstI 20),CstI 30,CstI 40)
   ... if x=20 then 30 else 40
Letfun("f", "x", Prim("+", Var "x", CstI 10),
       Call(Var "f", CstI 20))
   ... let f x = x + 10 in f 20
```

micro ML

micro ML の特徴

- 「プログラム=(値を返す) 式」 である (「プログラム=命令」でない)
- (再帰) 関数の定義、関数呼出しがある
- 関数そのものは値とならない (変数に格納できない)
- 副作用はない (変数の値の書換え、入出力)

micro ML プログラミング

例:

```
z + 8
let f x = x + 7 in f 2
let f x = if x = 0 then 1 else 2 + f(x−1) in f 7
```

注意点: micro ML での Letfun は再帰関数を定義する (OCaml での let ではなく let rec に相当する)。

関数クロージャの必要性

例:

```
let x = 10 in
let f y = x + y in
let x = 20 in
  f 30
```

質問: f の中の x は 10 なのか 20 なのか?

- 静的束縛 (lexical, static binding)
- 動的束縛 (dynamic binding)

静的束縛を実現するには、`f` が `y -> x + y` であるという情報だけでは不足!

関数クロージャの必要性

関数クロージャ(関数閉包, function closure): 関数定義と環境をセットにしたもの。

```
let x = 10 in
let f y = x + y in
let x = 20 in
  f 30
```

上記の f に対応する関数クロージャ

```
(f y = x + y, [(x,10)])
```

評価器での表現

```
type value =
  | Int of int
  | Closure of string * string * expr * value env
```

上記の f に対応する関数クロージャ

```
Closure("f", "y", Prim("+",...), [(x,Int(10))])
```

関数適用後の「関数本体の計算」で、現在の環境ではなく、関数クロージャに格納していた環境を拡張したもので計算。

関数クロージャを使った計算

```
let x = 10 in
let f y = x + y in
let x = 20 in
  f 30
```

環境に格納するのは、x=10 の形、もしくは、関数クロージャ：

```
env0 = [("x",Int(10))]
env1 = [("x",Int(20));("f",Closure("f","y",Prim(..),env0)
   );("x",Int(10))]
```

env1 のもとでの (f 30) の実行：

1. f の値を env1 から拾う。
2. 引数 30 を評価する。
3. f の仮引数 y を 30 に束縛する。
4. それを env0 に追加する (それを env2 とする)
5. f の本体を env2 で評価する。

関数クロージャを使った計算

課題: 以下を手で計算せよ。

```
let x = 10 in let f y = x + y in
let x = 30 in let g z = x + z + (f z) in
let x = 40 in f (g 100)
```

```
env0 = (x=10)
env1 = (x=30; f=Closure(f,y,x+y,env0); env0)
env2 = (x=40; g=Closure(g,z,x+z+f(z),env1); env1)
```

計算過程:

- 100 の計算では env2 を使う。
- g の本体 x+z+f(z) の計算を環境 (z=100;env1) で行う。
- その中の x を計算して 30 を、z を計算して 100 を得る。
- f の本体で x+y の計算を環境 (y=100;env0) で行う。(結果は 110)
- x+z+f(z) の計算結果は 240 である。
- f(240) の計算を env2 で行なう。
- f の本体 x+y の計算を環境 (y=240;env0) で行う。(結果は 250)

関数クロージャの処理

関数クロージャへの対応 1. (関数定義への対応)

```
let rec eval e env =
 match e with
 | CstI i    → i
 ...
 | Letfun(f, x, fBody, letBody) →
   let bodyEnv = (f, Closure(f, x, fBody, env)) :: env
   in eval letBody bodyEnv
 ...
```

関数クロージャを作って、それを環境に格納するだけ。

関数クロージャの処理

関数クロージャへの対応 2. (関数適用への対応)

```
...
| Call(Var f, eArg) →
  let fClosure = lookup env f in
  match fClosure with
  | Closure (f,x,fBody,fDeclEnv) →
    let xVal = Int(eval eArg env) in
    let fBodyEnv = (x,xVal)::(f,fClosure)::fDeclEnv in
       eval fBody fBodyEnv
```

- f に対応する値を、環境 env から取得
- それを Closure(...) とパターンマッチ
- f の実引数 eArg を「現在の」環境 env で評価 (計算)
- f の仮引数 x を実引数の評価結果に束縛 (環境に格納)
- (microML の関数は再帰関数なので) f 自身の値を再度環境に格納
- f の本体 fBody を新しい環境のもとで評価

疑問

いろいろな疑問：

- これで本当に静的束縛になるのか？
- 動的束縛にするのにはどうしたらよいか？
- 関数クロージャは，動的に (実行時に) 作られるが，メモリを消費するのではないか？

動的束縛

```
let x = 10 in
let f y = x + y in
let x = 20 in
  f 30
```

動的束縛では、評価結果が 30 になる。

動的束縛への対応

eval の定義を 1 か所変更する：関数クロージャにおける「環境」を無視する。
静的束縛に対応する eval の定義の一部 (再掲)

```
| Call(Var f, eArg) →
  let fClosure = lookup env f in
  match fClosure with
  | Closure (f,x,fBody,fDeclEnv) →
    let xVal = Int(eval eArg env) in
    let fBodyEnv = (x,xVal)::(f,fClosure)::fDeclEnv in
       eval fBody fBodyEnv
```

動的束縛: 上記の下から 2 番目の行を以下に置きかえればよい。

```
    let fBodyEnv = (x,xVal)::env in
```

あるいは、最初から、関数クロージャを作らずに、関数の本体の定義だけを覚えるという実装でもよい。

まとめ

- 拡張: 関数定義と関数適用を含むプログラム言語
- 静的束縛: 関数クロージャが必要
- 動的束縛: 関数クロージャは不要 (前回のインタプリタと同様の仕組みで単純に実装できる)

来週の演習にそなえて，スライドの復習 (と、できれば、Sestoft 教科書の第 4 章を参照) をしておいてください。テキストのプログラムは、coins マシンの~kam/plm/ の下に既に置いてあります。

来週は「演習のみ」なので、1 階の計算機室に集合です。注意してください。