プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 3: OCaml プログラミング、一階の関数型言語

目次

1. 再帰関数
2. 第 2 章 静的スコープの処理
3. 第 4 章 micro ML

OCaml における再帰関数

「再帰関数」は多くの言語が持つ。
C, Java, Scheme, OCaml, Haskell etc.

例: 与えられた正の整数に対して、f(1)=1 で、$x > 1$ に対しては、「x が偶数なら 2 で割り、奇数なら 3 倍して 1 を足す」ことを繰返す関数 f.

```
let is_even n =
   n mod 2 = 0
let rec f n =
  if n = 1 then 1
  else if is_even n then f (n / 2)
  else f (n * 3 + 1)
```

Collatz Problem (未解決問題): どんな正の整数 x から始めても f(x) の計算は止まる (いつか 1 になる)。

OCaml における再帰関数

f をそのまま計算するのではなく、「100 回まわったところで答えを返す」方法は？

```
let rec g n k =
  if k <= 0 then n
  else if n = 1 then 1
  else if is_even n then g (n / 2) (k - 1)
  else g (n * 3 + 1) (k - 1)
let f n = g n 100
```

割り切れるかどうかの判定

m が k から m-1 までの整数のうち 1 つ以上で割りきれるかどうかを判定する関数 h:

```
let rec h m k =
  if k >= m then false
  else if m mod k = 0 then true
  else h m (k + 1)
```

この関数は h m 2 で起動する。

m が 2 から k までの整数のうち 1 つ以上で割りきれるかどうかを判定する関数 h:

```
let rec h' m k =
  if k <= 1 then false
  else if m mod k = 0 then true
  else h' m (k − 1)
```

この関数は h' m (m-1) で起動する。

目次

1 再帰関数

2 第 2 章 静的スコープの処理

3 第 4 章 micro ML

この章の目的

- 対象言語を拡張; 変数と変数束縛
- 自由/束縛変数, スコープ、出現、代入
- スタック機械へのコンパイル

変数と変数スコープ

スコープ (scope):

```
let x = 10 in
  (let x = 20 in
    x + 10)      (* この x は 20  *)
  * (x + 3);;    (* この x は  3 *)

let x = 10 in
  let f y = y + x in
   f x ;;          (* この x は 10 *)
```

キーワード

- 静的スコープと動的スコープ
- ブロック構造言語
- スコープの入れ子 (nest)

変数と変数スコープ

C 言語の静的スコープ:

```
int z;
int foo (int x, int a[]) {
  int i;
  ... /* z x,a,i が使える */
  { int y;
    float x;
    ...
    ... /* z,a,i,y,x が使える. x は 内側の x */
  }
  ... /* z,x,a,i が使える */
}
```


変数の出現と束縛

項 e における変数 x の出現は、x をスコープに持つ束縛 (binding) があるとき、「e において束縛されている」と言う。そうでないとき「e において自由」と言う。

変数 x の出現が、複数の束縛のスコープにあるとき (入れ子のとき)、一番内側の束縛を取る。

どの x が、自由/束縛 出現か考えなさい。

```
(let x = 10 in
 (let f x = x + 3 in
 f x) + x)
+ x ;;
```


変数束縛のある言語の構文

以前の expr の定義をやめて、新たに定義しなおす。

```
e ::= i | e+e | x | let x = e in e
```

例: `let x = 1 + 2 in x + x`

```
type expr =
| CstI of int
| Prim of string * expr * expr
| Var  of string   (* 変数 *)
| Let  of string * expr * expr  (* 変数束縛 *)
```

例: `Let("x",CstI(10),Prim("+",Var("x"),CstI(20)))`
これは、OCaml の `let x=10 in x+20` という式に対応

変数と変数束縛のある言語の意味 (OCaml 版)

```
let rec eval e env =
 match e with
 | CstI i    → i
 | Var x     → lookup env x
 | Let(x, erhs, ebody)
     → let xval = eval erhs env in
        let env1 = (x, xval) :: env in
         eval ebody env1
 | Prim("+", e1, e2) → (eval e1 env) + (eval e2 env)
 | _ → failwith "unknown expression"
```

引数 env は 環境を表す。

変数と変数束縛のある言語の意味 (数式版)

式 e を環境 ϵ のもとで評価 (evaluate) した結果を $[\![e]\!]_\epsilon$ と書くことにする.

$$
\begin{aligned}
[\![CstI(i)]\!]_\epsilon &= i \\
[\![Var(x)]\!]_\epsilon &= \texttt{lookup}(\epsilon, x) \\
[\![Let(x, e1, e2)]\!]_\epsilon &= [\![e2]\!]_{\epsilon'} \\
&\qquad \text{where } \epsilon' = [(x, [\![e1]\!]_\epsilon)] \texttt{ @ } \epsilon \\
[\![Prim(" + ", e1, e2)]\!]_\epsilon &= [\![e1]\!]_\epsilon + [\![e2]\!]_\epsilon
\end{aligned}
$$

ここで, $\texttt{lookup}(\epsilon, x)$ は環境 ϵ において変数 x に対応する値をあらわす.
$L1$ @ $L2$ はリスト $L1$ と $L2$ を連結したリストをあらわす.

実行時の環境 (environment) とその操作

初期値 (空の環境) `[ ]`
環境の拡張

```
 [("x",10)] @ [("y",20); ("x",30)]
-> [("x",10); ("y",20); ("x",30)]
```

環境からの値の取り出し (lookup)

```
 lookup([("x",10); ("y",20); ("x",30)],  "x")
-> 10     (30 ではない)
```

後から入れたものが優先される.

評価の例 1 (数式版)

式 $Prim(" + ", CstI(20), CstI(10))$ を空環境で評価
(この式は `10 + 20` を表す。)

$$
\begin{aligned}
[\![Prim(" + ", CstI(20), CstI(10))]\!]_{[\,]} &= [\![CstI(20)]\!]_{[\,]} \; + \; [\![CstI(10)]\!]_{[\,]} \\
&= 20 \; + \; [\![CstI(10)]\!]_{[\,]} \\
&= 20 \; + \; 10 \\
&= 30
\end{aligned}
$$

評価の例 2 (数式版)

式 $Let("x", CstI(20), Var("x"))$ を空環境で評価

$$
\begin{aligned}
[\![Let("x", CstI(20), Var("x"))]\!]_{[\,]} &= [\![Var("x")]\!]_{[("x", [\![CstI(20)]\!]_{[\,]})]\texttt{ @ }[\,]} \\
&= [\![Var("x")]\!]_{[("x",20)]\texttt{ @ }[\,]} \\
&= [\![Var("x")]\!]_{[("x",20)]} \\
&= \texttt{lookup}([("x", 20)], "x") \\
&= 20
\end{aligned}
$$

評価の例 3 (数式版)

式 $Let("y", CstI(10), Let("x", CstI(20), Var("y")))$ を空環境で評価

$$
\begin{aligned}
&[\![Let("y", CstI(10), Let("x", CstI(20), Var("y")))]\!]_{[\,]} \\
&= [\![Let("x", CstI(20), Var("y"))]\!]_{[("y",10)]} \\
&= [\![Var("y")]\!]_{[("x",20)]\ @\ [("y",10)]} \\
&= [\![Var("y")]\!]_{[("x",20);("y",10)]} \\
&= 10
\end{aligned}
$$

評価の例 4 (数式版)

式 $Let("y", CstI(10), Let("x", CstI(20), Prim("+", Var("x"), Var("y"))))$ を空環境で評価

$$
\begin{aligned}
&[\![Let("y", CstI(10), Let("x", CstI(20), Prim("+", Var("x"), Var("y"))))]\!]_{[\,]} \\
&= [\![Let("x", CstI(20), Prim("+", Var("x"), Var("y")))]\!]_{[("y",10)]} \\
&= [\![Prim("+", Var("x"), Var("y"))]\!]_{[("x",20);("y",10)]} \\
&= [\![Var("x")]\!]_{[("x",20);("y",10)]} + [\![Var("y")]\!]_{[("x",20);("y",10)]} \\
&= 20 + 10 \\
&= 30
\end{aligned}
$$

評価の例 5 (数式版)

式 $Let("x", Prim("+", CstI(10), CstI(20)), Var("x"))$ を空環境で評価

$$
\begin{aligned}
&[\![Let("x", Prim("+", CstI(10), CstI(20)), Var("x"))]\!]_{[\,]} \\
&= [\![Var("x")]\!]_{[("x",[\![Prim("+",CstI(10),CstI(20))]\!]_{[\,]})]} \\
&= [\![Var("x")]\!]_{[("x",10+20)]} \\
&= [\![Var("x")]\!]_{[("x",30)]} \\
&= 30
\end{aligned}
$$

演習問題

以下の式を空環境で評価せよ.

```
Let("x", CstI(10),
    Prim("+",Let("x",CstI(20),
                 Prim("+",Var("x"),Cst(30))),
        Var("x")))
```

(この式は `let x=10 in (let x=20 in x + 30) + x` を表す。)

目次

1. 再帰関数
2. 第 2 章 静的スコープの処理
3. 第 4 章 micro ML

micro ML 言語

- 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」ための制限)

```
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 rec f x = x + 10 in f 20
```


micro ML プログラミング

```
microML
  Prim("+", Var("z"), CstI(8))
OCaml
  z + 8

microML
  Letfun("f","x",
    Prim("+",Var("x"),CstI(7)),
    Call(Var("f"),CstI(2)))
OCaml
  let rec f x = x + 7 in f 2
```

micro ML での Letfun は再帰関数の定義 (OCaml の let rec)。

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

関数クロージャ(関数閉包): 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

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

これは、`f y =x + y` と環境をセットにしたもの。
f が使われるとき、x=10 という環境のもとで評価される。

microML の意味 (数式版)

$[\![e]\!]_\epsilon$ は式 e を環境 ϵ のもとで評価した結果:
(前と同じ)

$$[\![CstI(i)]\!]_\epsilon = \ldots$$
$$[\![Var(x)]\!]_\epsilon = \ldots$$
$$[\![Let(x, e1, e2)]\!]_\epsilon = \ldots$$
$$[\![Prim("+", e1, e2)]\!]_\epsilon = \ldots$$

(新しい)

$$[\![CstB(b)]\!]_\epsilon = b$$
$$[\![If(e0, e1, e2)]\!]_\epsilon = [\![e1]\!]_\epsilon \text{ where } [\![e0]\!]_\epsilon = true$$
$$[\![If(e0, e1, e2)]\!]_\epsilon = [\![e2]\!]_\epsilon \text{ where } [\![e0]\!]_\epsilon = false$$

microML の意味 (数式版)

(新しい)

$$[\![Letfun("f","x",e1,e2)]\!]_\epsilon = [\![e2]\!]_{\epsilon'}$$
$$\text{where } \epsilon' = [("f", Closure("f","x",e1,\epsilon))] \mathbin{@} \epsilon$$
$$[\![Call("f",e0)]\!]_\epsilon = [\![e1]\!]_{\epsilon''}$$
$$\text{where } \texttt{lookup}(\epsilon,"f") = Closure("f","x",e1,\epsilon')$$
$$v = [\![e0]\!]_\epsilon$$
$$\epsilon'' = [("x", v)] \mathbin{@} \epsilon'$$

1. "f" に対応する関数クロージャを ϵ から探す
2. 引数 e0 を現在の環境 ϵ で評価 (その結果を v とする)
3. 仮引数"x"を v とする束縛を ϵ' に追加
4. その環境 ϵ'' のもとで、関数本体 e1 を評価

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
```

上記で f 30 を計算する瞬間の環境 env1

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

env1 のもとでの (f 30) の実行：

1. f の値を env1 から探す。
2. 引数 30 を評価する。
3. y を 30 に束縛して環境 [("x",10)] に追加
5. f の本体をその環境で評価

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 = 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",...,env0))] @ env0
env2 = [("x",40;("g",Closure("g","z",...,env1))] @ env1
```

計算過程:

- (g 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。
- f の本体 x+y の計算を環境 (y=100;env0) で行う。
- 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 (OCaml 版)

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

関数クロージャの処理 (OCaml 版)

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
```

動的束縛

動的に (実行時に) 束縛が決まる

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

動的束縛では、関数本体 x+y における x は、関数が実行される時点での環境で束縛されるので、x=20 となる。(上記の式の計算結果は 50)

動的束縛にした microML の意味 (数式版)

以下の定義 (のみ) を変更すれば動的束縛になる

$$[\![Call("f", e0)]\!]_\epsilon = [\![e1]\!]_{\epsilon''}$$
$$\text{where } \texttt{lookup}(\epsilon, "f") = Closure("f", "x", e1, \epsilon')$$
$$v = [\![e0]\!]_\epsilon$$
$$\epsilon'' = [("x", v)] \texttt{@} \epsilon$$

変更点は、最後の行で ϵ' だったものを ϵ にした点のみ。
1. 静的束縛では、ϵ'' を作る「もと」は、クロージャの環境 ϵ' であった。
2. 動的束縛では、ϵ'' を作る「もと」は、現在の環境 ϵ である。(そもそもクロージャを作る必要がない。)

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