プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 4： ラムダ計算、高階の関数型言語

関数型言語

- ラムダ計算 (＝「関数」概念を追究した体系) に基づくプログラム言語たちのこと
- 例. Lisp, Scheme, ML (SML, OCaml, F#), Haskell
- Ruby, JavaScript, Scala などの言語も関数型言語の機能を取り込む。
- 機能の例. 関数クロージャ, Generics, Map/Reduce

ラムダ計算

λ-calculus: Alonzo Church (1936) が提案した計算モデル

- 「関数」の概念を究極までつきつめて得られた体系
- 関数の出力だけでなく入力となる変数を明記
- 「$f(x) = x^2 + 5x$ となる関数 f」 を「$\lambda x.\ x^2 + 5x$」と表示
- $f\ 10 = (\lambda x.\ x^2 + 5x)\ 10 = 10^2 + 5 \cdot 10$
- 他の「計算モデル」: チューリング機械、再帰的関数など。

書き方： 数学で $f(x)$ と書くところをラムダ計算では $f\ x$ と書く (ことが多い)

ラムダ項、ラムダ式

ラムダ項の構文:

$$
\begin{array}{lr}
M ::= x & \text{変数} \\
\quad | \ \lambda x.\ M & \text{ラムダ抽象} \\
\quad | \ M\ M & \text{関数適用}
\end{array}
$$

注. 括弧は適宜使ってよい。

ラムダ項の例: $x\ y$, $x\ (y\ z)$, $\lambda x.\ x$ (恒等関数)、$\lambda x.\lambda y.\ x(xy)$

構文を拡張したラムダ項の例: $\lambda x.\ x + 1$ (1 を足す関数)、
$\lambda x.\lambda y.\lambda z.$ if x then y else z (x が true なら y を返し、そうでなければ z を返す関数)、

ラムダ計算の強力さ

高階関数 (higher-order function)

関数を入力あるいは出力 (あるいは、その両方) とする関数のこと。数学では「汎関数」。

例

- $\lambda f.\ f\ 10$ ... 関数 f をもらって、f(10) の値を返す高階関数
- $\lambda f.\ f(f\ 10)$ .. 関数 f をもらって、f(f(10)) の値を返す高階関数
- $\lambda x.\ (\lambda y.\ x + y)$ .. x をもらって、「y をもらうと $x + y$ を返す関数」を返す高階関数
- $\lambda f.\ (\lambda x.\ f(f\ x))$ .. 関数 f をもらって、「x をもらって f(f(x)) の値を返す関数」を返す高階関数
- $\lambda f.\ (\lambda g.\ (\lambda x.\ g(f\ x)))$ .. 関数 f をもらって、「関数 g をもらって、「f と g の合成関数」を返す高階関数」を返す高階関数

ラムダ計算

ラムダ計算の表現力

ラムダ計算は、計算モデルとして、チューリング機械と同等の表現力を持つ。

つまり、チューリング機械 (や再帰的関数やその他のありとあらゆる計算モデル) をシミュレートできるし、逆にチューリング機械でラムダ計算をシミュレートすることもできる。

再帰関数を表現する「不動点コンビネータ (Curry の Y コンビネータ)」(詳細は、秋学期の「計算モデル論」の授業 (井田哲雄先生) を参照)。

関数型プログラム言語

ラムダ計算に基づく関数型プログラム言語の特徴:

- 単一代入、参照透明性 (referential transparency)
- 計算エフェクトが分離されている (ことが多い)、意味論が明快で検証しやすい
- 簡単な割に実は強力; 高階関数，データ型

関数型プログラム言語の利用

- 得意な分野: アルゴリズム，プログラム言語の処理系，記号処理システム (不定長データの複雑な処理) など
- 不得意な分野: 固定長データの数値計算，高性能計算

関数型言語

- Lisp: 長い歴史，人工知能システムや数式処理システムなど
- Scheme: Lisp の意味論を洗練したコンパクトな言語。
- ML: 広く使われている関数型言語の一族，SML, OCaml, F# などの総称
- Haskell: 遅延評価と「純粋関数型」という特徴をもつ。

Ruby, Scala, JavaScript など他のプログラム言語にも多大な影響。

関数型のプログラミング・スタイル1

手続き的スタイル: 繰返し
(for, while,...)

```
int fib (int n) {
  int i, tmp;
  int x=1, y=1;
  for (i=2; i<n; i++) {
    tmp = x;
    x = y;
    y += tmp;
  }
  return y;
}
```

関数的スタイル: 再帰
呼出し

```
let rec fib n =
  if n<=2 then 1
  else fib(n-1) + fib(n-2)
```


関数型のプログラミング・スタイル2

単一代入: 変数に対する束縛は1回限り

手続き的スタイル: 変数への値の代入

```
int foo (int x) {
  int y;
  y = x + goo(x+1);
  y += hoo(y*y);
  y = goo(y+2);
  ...
  return y;
}
```

関数的スタイル: 局所的な変数束縛

```
let foo x =
  let y = x + goo(x+1) in
  let y = y + hoo(y*y) in
  let y = goo(y+2) in
  ...
    y
```


関数型のプログラミング・スタイル2'

C 言語でできて、単一代入の言語ではできないこと：

```
int x = 10;
if (some_condition) { x = x + 5;}
printf("%d\n",x);
```

最後に印刷する際の x の値がどこで束縛されたか、静的にはわからない。
Assignment (日本語では「割当て」または「代入」、なお、substitution とはまったく別の概念)

ラムダ計算と OCaml

OCaml では $\lambda x.M$ という式は `funx -> M` と表現

```
let f = fun x -> x + 5 in
  f (f 3)
let g = fun x -> f x + 8 in
  g (f (g 3))
```

$f = \lambda x.\ x + 5$ であり $g = \lambda x.\ (f\ x) + 8$ である。
ポイント: 関数が普通のデータとなり変数 f や g に格納できる。→ 関数をデータとして扱う関数 (高階関数)

関数型のプログラミング・スタイル3

高階関数の例:

map 関数の利用

```
let foo a b lst =
  List.map
   (fun x -> x*a+b) lst
in
  foo 10 20 [1; 2; 3; 4]
==>
  [30; 40; 50; 60]
```

map 関数は他のものにも使える

```
List.map
 (fun x -> x ^ ".ml")
 ["foo"; "goo"; "a"]
==>
["foo.ml"; "goo.ml"; "a.ml"]
```


関数型のプログラミング・スタイル4

高階関数を定義する

```
let foo1 f g x =
  g (f x)
let foo2 f g =
  fun x -> g (f x)
let foo3 =
  fun f -> fun g ->
    fun x -> g (f x)
```

高階関数を定義する

```
let rec goo n f x =
  if n=0 then x
  else goo (n-1) f (f x)
in
  goo 5 (fun x -> x + 10) 7
==>
  57
```


演習

問題: 以下の関数は何をするものか言葉で答えなさい。

```
let rec foo f n x =
  if n = 0 then x
  else f (foo f (n − 1) x)
```

ヒント: foo の第一引数 f として、let add1 x = x + 1 として定義された関数 add1 や let times2 x = x ∗ 2 として定義された関数 times2 をいれたものを想定してみなさい。

オプション問題: 自然数 m,n に対して、foo (foo add1 m) n 0 を計算すると何が返るだろうか？

計算エフェクト、副作用

式の計算において、「値を返す」以外の状態変化を計算エフェクト (computational effect)、あるいは、副作用 (side effect) という。

- 例 1: 変数の値を変更 (状態の変更)
- 例 2: ファイルからの入力、ファイルや端末への出力
- 例 3: プログラムの制御を変更 (ジャンプ命令)

関数型言語と計算エフェクト

- 計算エフェクトがなければ，プログラムの解析等は容易
- 例えば f(x)+f(x) = f(x) * 2 が成立するかどうか
- 関数型言語は、副作用がない (Haskell) か、あるいは、副作用のある計算が明示される (OCaml の参照型など) ため、プログラムの解析が容易。

関数型言語の意味論と処理

関数をデータとして扱うための仕組みは？
関数クロージャ
対象言語の拡張：

```
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 * expr
  | Lambda of string * expr   (* new *)
  | Call of expr * expr (* 1st arg is expr, not string
 *)
```

式の例

```
Lambda("x",Prim("+",Var("x"),CstI(3)))
Lambda("x",Lambda("y",Prim("+",Var("x"),Var("y"))))
```

これらは $\lambda x.\ x+3$ と $\lambda x.\lambda y.\ x+y$ に相当

関数型言語の意味論と処理

[訂正 (2018/5/18)]
授業では、「新しい言語では Letfun は不要」と書いたが、実は、新しい言語は再帰関数を定義する機能をいれていなかったので、再帰関数定義のためには Letfun は必要。

再帰的でない関数を定義する Letfun は不要; 以下の 2 つは同じ:

```
Letfun("f","x",e1,e2)
Let("f",Lambda("x",e1),e2)
```

これらを OCaml で書くとまったく同じ意味の式となる

```
let f x = e1 in e2
let f = fun x → e1 in e2
```


関数型言語の意味論と処理

意味論を新しい言語用に拡張する。
必要なのは Lambda と Call に対する拡張。

$$[\![Lambda("x", e)]\!]_\epsilon = Closure("dummy", "x", e, \epsilon)$$

関数クロージャを返すだけ。dummy は他と衝突しない変数名。

$$\begin{aligned}[\![Call(e2, e0)]\!]_\epsilon &= [\![e1]\!]_{\epsilon''} \\ \text{where } [\![e2]\!]_\epsilon &= Closure("f", "x", e1, \epsilon') \\ v &= [\![e0]\!]_\epsilon \\ \epsilon'' &= [("x", v)] \mathbin{@} \epsilon\end{aligned}$$

前との違い: e2 のところが関数名 (文字列) でなく、式になったこと。
(従って、それを lookup するのではなく、計算しなければいけない)

意味論の例 1

e =Lambda("x",Prim("+",Var("x"),CstI(3))) で $\epsilon = [\,]$ のとき
$[\![e]\!]_\epsilon$ =Closure("dummy","x",Prim("+",..)) なので、

$$\begin{aligned}[\![Call(e, CstI(5))]\!]_\epsilon &= [\![Prim("+", Var("x"), CstI(3))]\!]_{\epsilon''} \\ \text{where } [\![e]\!]_\epsilon &= Closure("dummy", "x", Prim("+", ..)) \\ v &= [\![CstI(5)]\!]_\epsilon = 5 \\ \epsilon'' &= [("x", v)] \mathbin{@} \epsilon = [("x", 5)]\end{aligned}$$

となるので、

$$\begin{aligned}[\![Call(e, CstI(5))]\!]_\epsilon &= [\![Prim("+", Var("x"), CstI(3))]\!]_{\epsilon''} \\ &= [\![Var("x")]\!]_{\epsilon''} + [\![CstI(3)]\!]_{\epsilon''} \\ &= 5 + 3 = 8\end{aligned}$$

意味論の例2

e =Lambda("x",Prim("+",Var("x"),Var("y")))
$e2$ =Let("f",e,Let("y",CstI(5),Call(Var("f"),CstI(3))) で
$\epsilon = [("y", 8)]$ のとき

$$\begin{aligned}[\![e2]\!]_\epsilon &= [\![Let("y", CstI(5), Call(Var("f"), CstI(3)))]\!]_{\epsilon 1} \\ &= [\![Call(Var("f"), CstI(3))]\!]_{\epsilon 2}\end{aligned}$$

となる。ここで
$\epsilon 1 = [("f", Closure("dummy","x", Prim(...), \epsilon))]@\epsilon$ および
$\epsilon 2 = [("y", 5)]@\epsilon 1$ である。

意味論の例2 (続き)

(前ページの続き)
$[\![Var("f")]\!]_{\epsilon 2}$ = Closure("dummy","x",Prim(...),ϵ) なので、

$$\begin{aligned}[\![Call(Var("f"), CstI(3))]\!]_{\epsilon 2} &= [\![Prim("+", Var("x"), Var("y"))]\!]_{\epsilon 3} \\ &= 3 + 8 = 11\end{aligned}$$

となる。ここで $\epsilon 3 = [("x", 3)]@\epsilon$ である。

ポイント: $\epsilon 3$ のもとになっている環境が、今の環境 $\epsilon 2$ でなく、ラムダ式が評価された時の環境 ϵ であることに注意。よって変数 y の値は 8 である。

ここまでのまとめ

- 関数プログラミング: 単一代入，高階関数，(再帰呼び出し、データ型の活用)
- 単一代入⇒副作用の分離・明示
- 高階関数と静的束縛⇒関数クロージャで処理できる

再帰と繰返し

関数型言語では、繰返しではなく再帰を多用する。

```
let rec sum n =
  if n = 0 then 0
  else n + sum (n - 1)

int sum (int n) {
  int i, res = 0;
  for (i = 0; i <= n; i++)
     res += i;
  return res;
}
```

再帰の効率は？

- 関数呼び出しをするごとに、「その呼び出しから戻ってきたあとの計算」の情報を覚えないといけない。
- 繰返し処理なら、100000 回ループしても問題ないが、再帰では、スタックがあふれて、処理が中断してしまう ???

末尾呼び出し、末尾再帰

Tail call: 関数呼び出しが、処理の「末尾」のみであるもの：

```
let rec sum n =
  if n = 0 then 0
  else n + sum (n - 1)  ;; NG
let rec sum2 n m =
  if n = 0 then m
  else sum2 (n - 1) (n + m) ;; OK
let rec power n x =
  if n = 0 then 1
  else if (even n) then square(power (n/2) x)
  else x * (power (n-1) x)  ;; NG
let rec power2 n x m =
  if n = 0 then m
  else if (even n) then
       power2 (n/2) (square x) m
  else power2 (n-1) x (n*m) ;; OK
```


末尾呼び出し、末尾再帰

末尾呼び出しの処理の最適化:

- 関数呼び出しのあと、「戻ってくる」必要がない。
- →繰返しと同様の処理が可能となる。(スタックを消費しない。)

現実のプログラム言語の処理系:

- Scheme: 処理系は必ず末尾再帰の最適化
- ML など多くの関数型言語: 末尾再帰の最適化をするのが普通
- C,JavaScript など： 末尾再帰の最適化は必ずしもしない

→ 自分の好きな言語の処理系で、「末尾再帰を 1 億回おこなう」コードを書いて試してください。(同じ言語でも、処理系ごとに違うことがあります。)

今年の授業では「継続」はやっていません。以下のものは参考資料です。

継続渡し方式

高階関数を用いた、ある種のプログラムの記述方法:

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n-1)  ;; NG

let rec fact_cps n k =
  if n = 0 then k 1
  else fact_cps (n-1) (fun x -> k (n * x))  ;; OK

fact_cps 5 (fun x -> x)
=> fact_cps 4 (fun x -> 5 * x)
=> fact_cps 3 (fun x -> 5 * 4 * x)
=> ...
=> fact_cps 0 (fun x -> 5 * 4 * 3 * 2 * 1 * x)
=> 5 * 4 * 3 * 2 * 1 * 1
```

末尾再帰になっている。

継続渡し方式

前ページのプログラムへの疑問：

- k は何だろうか？
- 普通のプログラムを、継続渡し方式にするのはどうしたらいいだろうか？
- 何のメリットがあるのだろうか？

→ 教科書 11 章を参照。

継続

継続 (continuation): プログラム実行時の、「残りの計算」を表す概念。

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n-1)  ;; NG
```

fact 5	この時点での継続 []
5 * (fact 4)	この時点での継続 5 * []
5 * 4 * (fact 3)	この時点での継続 5 * (4 * [])

継続渡し方式

継続渡し方式: Continuation-Passing Style

- 「継続」を関数として表現して、明示的に表す。
- もともとの関数は、「継続」を表す引数を取る高階関数となる。

```
let rec fact_cps n k =
  if n = 0 then k 1
  else fact_cps (n-1) (fun x -> k (n * x))  ;; OK
fact_cps 5 (fun x -> x)
=> fact_cps 4 (fun x -> 5 * x)
=> fact_cps 3 (fun x -> 5 * 4 * x)
=> ...
=> 5 * 4 * 3 * 2 * 1 * 1
```

CPS の利点:

- 末尾再帰、 全ての途中結果に名前が付いている
- →コンパイラの中間言語 (Appel, “Compiling with Continuations”)
- さまざまな制御を表現可能

継続渡し方式での制御

```
let rec sqrt_multiply lst =
  match lst with
  | [] -> 1
  | h :: t -> (sqrt h) * (sqrt_multiply t)

let rec sqrt_multiply_cps lst k =
  match lst with
  | [] -> k 1
  | h :: t ->
     if h >= 0 then
       sqrt_multiply_cps t
           (fun x -> k ((sqrt h) * x)
     else -1
```

リスト中に負の数があると、その途端に (残りの計算をやらずに) 抜けだす→ ML や Java の exception に相当することを、特別なしかけなしに実現。