

関数プログラミングの基本概念

胡 振江

東京大学 計数工学科

2007年10月15日, 22日

Copyright © 2007 Zhenjiang Hu, All Right Reserved.

Outline

- 1 関数プログラミング
- 2 関数プログラムの評価
- 3 関数
- 4 型
- 5 関数の定義
- 6 関数プログラムの処理系

関数プログラミング

関数プログラミングは二つ部分からなる。

- 1 関数を定義する。

$square :: Integer \rightarrow Integer$

$square\ x = x * x$

$smaller :: (Integer, Integer) \rightarrow Integer$

$smaller\ (x, y) = \mathbf{if}\ x \leq y\ \mathbf{then}\ x\ \mathbf{else}\ y$

- 2 計算機を用いて式を評価する。

? `square 3768`

14197824

? `square (smaller (5,3+4))`

25

?

Outline

- 1 関数プログラミング
- 2 関数プログラムの評価
- 3 関数
- 4 型
- 5 関数の定義
- 6 関数プログラムの処理系

関数プログラムの評価：式の簡約

関数プログラムの評価は、式を「最も単純な等価な形」に簡約し結果を表す過程である。

$square(3 + 4)$
 \Rightarrow { def. of + }
 $square\ 7$
 \Rightarrow { def. of square }
 $7 * 7$
 \Rightarrow { def. of * }
 49

簡約方法は唯一ではない

式 $square(3 + 4)$ を簡単にする方法はこのほかにもある。

$square(3 + 4)$
 \Rightarrow { def. of square }
 $(3 + 4) * (3 + 4)$
 \Rightarrow { def. of + }
 $7 * (3 + 4)$
 \Rightarrow { def. of + }
 $7 * 7$
 \Rightarrow { def. of * }
 49

止まらない簡約もある

次のプログラムを考える。

```
three :: Integer → Integer  
three x = 3
```

```
infinity :: Integer  
infinity = infinity + 1
```

式 *three infinity* の評価はどうなるか。

- *infinity* を優先的に簡約すると、無限ステップになる。
- *three* を優先的に簡約すると、結果がすぐに得られる。

関数プログラミング言語

関数プログラミング言語は、**ラムダ計算 (lambda calculus)** の概念をプログラミング言語として体現したもの。

代表的な関数プログラミング言語

- Lisp 1958- (Scheme 1970 年代後半-)
- ISWIM 1966-
- ML 1970 年代後半-
- Miranda 1985-
- **Haskell** 1987- (純粋関数プログラミング言語, Haskell の名前は論理学者 Haskell B. Curry に由来, **本講義で使う言語**)

Outline

- 1 関数プログラミング
- 2 関数プログラムの評価
- 3 関数**
- 4 型
- 5 関数の定義
- 6 関数プログラムの処理系

関数（数学）

入力 x に対して、出力 y のただ一つの値を決定する規則が与えられているとき、 y を x の関数という。対応規則を明示するときは、

$$y = f(x)$$

のように対応規則に名前を付与する。

Cの関数 \neq 関数(数学)

例：プログラミング言語 C の関数定義

```
1: int y = 10;  
2: int f (x: int) {  
3:     printf ("Hello!");  
4:     return y * x;  
5: }
```

次の点で数学の関数とは異なる。

- **副作用を持つ**：関数の処理の実行によってシステムに変化が発生する
- **状態を持つ**：引数が同じでも状況に応じて戻り値が異なる

本講義中の関数 = 関数 (数学)

関数

$$f :: A \rightarrow B$$

は型 A の引数 (argument) をとり, 型 B の結果 (result) を返すという. また, x が A の元を表しているとき, x に関数 f を適用した結果を $f(x)$ または $f x$ で表す.

square :: *Integer* → *Integer*

smaller :: (*Integer*, *Integer*) → *Integer*

three :: *Integer* → *Integer*

関数の外延性

関数の相等

$f = g$ iff 任意の x に対して、 $f\ x = g\ x$ が成り立つ。

次の二つの関数が等しい。

$double, double' :: Integer \rightarrow Integer$

$double\ x = x + x$

$double'\ x = 2 * x$

カーリー化

カーリー化

構造をもつ値の引数を単純な引数の列に置き換える方法。

- 引数が構造を持つ

$$\begin{aligned} \text{smaller} &:: (\text{Integer}, \text{Integer}) \rightarrow \text{Integer} \\ \text{smaller } (x, y) &= \mathbf{if } x \leq y \mathbf{ then } x \mathbf{ else } y \end{aligned}$$

- 引数が構造を持たない

$$\begin{aligned} \text{smallerc} &:: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \\ \text{smallerc } x \ y &= \mathbf{if } x \leq y \mathbf{ then } x \mathbf{ else } y \end{aligned}$$

カリー化の利点：部分関数

関数定義

$$\begin{aligned} \text{smallerc} &:: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \\ \text{smallerc } x \ y &= \mathbf{if} \ x \leq y \ \mathbf{then} \ x \ \mathbf{else} \ y \end{aligned}$$

は

$$\begin{aligned} \text{smallerc} &:: \text{Integer} \rightarrow (\text{Integer} \rightarrow \text{Integer}) \\ (\text{smallerc } x) \ y &= \mathbf{if} \ x \leq y \ \mathbf{then} \ x \ \mathbf{else} \ y \end{aligned}$$

と同じである。従って、

$$\text{smallerc } 5$$

は関数（どうな関数？）を表している。

二項演算子のセクション表現

セクション：括弧でくくられた演算子

$$\begin{aligned} (+) & \quad :: \text{Int} \rightarrow \text{Int} \rightarrow \text{int} \\ (+) x y & = x + y \end{aligned}$$

括弧でくくられた **infix 演算子** が普通の **prefix 関数** のように引数を適用することができる。また、引数として関数に渡したりすることができる。

$$\text{both } f \ x = f \ x \ x$$

と定義すると、

$$\text{both } (+) \ 3 \Rightarrow (+) \ 3 \ 3 \Rightarrow 3 + 3 \Rightarrow 6$$

二項演算子のセクション表現

更に拡張： 引数を演算子とともに括弧でくくる。

$$(x\oplus) y = x \oplus y$$

$$(\oplus y) x = x \oplus y$$

例：

(*2)： 2倍する関数

(1/)： 逆数を求める関数

(/2)： 2分する関数

(+1)： つぎの値を得る関数

関数の合成

関数合成

$$\begin{aligned}(\cdot) &:: (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma) \\(f \cdot g) x &= f (g x)\end{aligned}$$

例：

$$\begin{aligned}quad &:: Integer \rightarrow Integer \\quad &= square \cdot square\end{aligned}$$

性質

$$\begin{aligned}id \cdot f &= f \cdot id = f \\(f \cdot g) \cdot h &= f \cdot (g \cdot h)\end{aligned}$$

Outline

- 1 関数プログラミング
- 2 関数プログラムの評価
- 3 関数
- 4 **型**
- 5 関数の定義
- 6 関数プログラムの処理系

型

型 (type) はある種類の値の集まりである。

- 基本型

- 整数型 Integer/Int: $\{\dots, -3, 2, 0, 10, \dots\}$
- 浮動小数点数型 Double/Float: $\{\dots, -3.14, 3, 0, 123.45, \dots\}$
- 論理型 Bool: $\{True, False\}$
- 文字型 Char: $\{'a', 'b', \dots, 'A', 'B', \dots\}$
- 文字列型 String: $\{"HelloWorld", \dots\}$

- 合成型 (派生型)

- リスト型 $[t]$: $\{[], [2, 5, 4], \dots\}$
- 組型 (t_1, t_2) : $\{(1, 2), ('a', 3), \dots\}$
- 関数型 $t_1 \rightarrow t_2$: $\{square, smaller\ c\ 5, \dots\}$

関数の型・多様型

関数プログラミングにおいて、関数は値であり、他の値と対等のものである。関数を引数として関数に渡したり、結果として返したりする。

$$\begin{aligned} \text{apply} &:: (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \\ \text{apply } f \ x &= f \ x \end{aligned}$$
$$\begin{aligned} \text{curry} &:: ((\alpha, \beta) \rightarrow \gamma) \rightarrow \alpha \rightarrow \beta \rightarrow \gamma \\ \text{curry } f \ x \ y &= f \ (x, y) \end{aligned}$$
$$\begin{aligned} \text{uncurry} &:: (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha, \beta) \rightarrow \gamma \\ \text{uncurry } f \ (x, y) &= f \ x \ y \end{aligned}$$

Outline

- 1 関数プログラミング
- 2 関数プログラムの評価
- 3 関数
- 4 型
- 5 関数の定義**
- 6 関数プログラムの処理系

関数の定義 (1/3)

条件の表現：

$smaller :: (Integer, Integer) \rightarrow Integer$
 $smaller(x, y) = \mathbf{if} \ x \leq y \ \mathbf{then} \ x \ \mathbf{else} \ y$

または

$smaller :: (Integer, Integer) \rightarrow Integer$
 $smaller(x, y)$
| $x \leq y = x$
| $x > y = y$

関数の定義 (2/3)

再帰の表現：

$$\begin{aligned} \text{fact} &:: \text{Integer} \rightarrow \text{Integer} \\ \text{fact } n &= \mathbf{if } n == 0 \mathbf{ then } 1 \mathbf{ else } n * \text{fact } (n - 1) \end{aligned}$$

評価例：

$$\begin{aligned} &\text{fact } 1 \\ = &\quad \{ \text{def. of fact} \} \\ &\mathbf{if } 1 == 0 \mathbf{ then } 1 \mathbf{ else } 1 * \text{fact } (1 - 1) \\ = &\quad \{ \text{since } 1 == 0 \text{ evaluates to False} \} \\ &1 * \text{fact } (1 - 1) \\ = &\quad \{ 1 - 1 = 0 \} \\ &1 * \text{fact } 0 \\ = &\quad \{ \text{many steps ...} \} \\ &1 \end{aligned}$$

関数の定義 (3/3)

局所定義：

$$\begin{aligned} f &:: (\text{Float}, \text{Float}) \rightarrow \text{Float} \\ f(x, y) &= (a + 1) * (b + 2) \\ &\mathbf{where} \\ &\quad a = (x + y)/2 \\ &\quad b = (x + y)/3 \end{aligned}$$

または、

$$\begin{aligned} f &:: (\text{Float}, \text{Float}) \rightarrow \text{Float} \\ f(x, y) &= (a + 1) * (b + 2) \\ &\mathbf{where } a = (x + y)/2; b = (x + y)/3 \end{aligned}$$

Outline

- 1 関数プログラミング
- 2 関数プログラムの評価
- 3 関数
- 4 型
- 5 関数の定義
- 6 関数プログラムの処理系

関数プログラムの処理系

本講義で関数プログラミング言語 Haskell (コア部分)
<http://www.haskell.org/haskellwiki/Haskell>
を使う。

- Haskell のコンパイラ: ghc
<http://www.haskell.org/ghc/>
- Haskell のインタプリタ: hugs
<http://haskell.org/hugs/>

Haskell のインタプリタ: Hugs

- Mark P. Jones らによって開発された Haskell のインタプリタで、現在の最新版は Hugs 98 である。
- Hugs は、多くの Unix や Windows 上で動くことが確認されており、コンパイル済のバイナリコードが Win32, Linux, Machintosh の各プラットフォームに用意されている。
- Windows 用の Hugs (winhugs) は GUI を備えており、初心者でも扱いやすくなっている。
- Hugs のインストールは簡単である。

宿題

- 教科書第一章を復習し、演習問題をやる。
- Hugs システムをインストールする。
- 「Learning Haskell in 10 Minutes」を自習する。

http://www.haskell.org/haskellwiki/Learn_Haskell_in_10_minutes