

# 関数・論理型プログラミング実験

## 第2回

江口 慎悟  
酒寄 健  
塚田 武志  
松下祐介

# 講義のサポートページ

<http://www.kb.is.s.u-tokyo.ac.jp/~tsukada/cgi-bin/m/>

- 講義資料等が用意される
- レポートの提出先
- 利用にはアカウントが必要
  - アカウントの自己登録は本日で締め切ります
  - まだの方は急いでください

# 今日の内容

- 型多相性
  - Parametric Polymorphism
- ユーザ定義型
  - レコード型
  - ヴァリアント型
  - 多相データ型
- 副作用

型多相性

# 型多相性とは

- 異なる型や式などを  
どうにかしてまとめて扱う仕組み

# 型多相性のない世界

- 型ごとに別の関数を定義
    - 面倒。バグの元
  - 例：リストの先頭要素を返す関数
    - `hd_int : int list -> int`
      - 実装：`let hd_int (a::_) = a`
    - `hd_bool : bool list -> bool`
      - 実装：`let hd_bool (a::_) = a`
    - `hd_i2i : (int -> int) list -> int -> int`
      - 実装：`let hd_i2i (a::_) = a`
- この型に対して、functionを定義しないといけない。

# 解決：型でパラメトライズ

型を定義する。

型αのリストから。

型αを引出す。

- $\text{hd}[\alpha] : \alpha \text{ list} \rightarrow \alpha$

```
# let hd[α] (a :: _) = a;;
```

- 必要に応じて具体化

- $\text{hd[int]} : \text{int list} \rightarrow \text{int}$
- $\text{hd[bool]} : \text{bool list} \rightarrow \text{bool}$
- $\text{hd[int} \rightarrow \text{int]} :$   
 $(\text{int} \rightarrow \text{int}) \text{ list} \rightarrow \text{int} \rightarrow \text{int}$

# 実際のコード

```
# let hd (a::_) = a;;
Warning 8:...
val hd : 'a list -> 'a = <fun>
```

← これが書き込み式に対応している。

- OCaml では型パラメタを式に書かない  
(型推論器が自動的に補う)

注：実は3.12以降は書ける

# Parametric Polymorphism

- 型をパラメータにすることで、  
本質的に同一なものをまとめる方法  
型によらず同じ動作

- cf. subtype polymorphism
  - OOP の継承
- cf. ad-hoc polymorphism
  - 関数のオーバーロード
    - Haskell の type class
    - OCaml の比較演算

# 多相関数の例

- 恒等関数

```
# let id x = x;;
val id : 'a -> 'a = <fun>
```

```
# id 1;;
- : int = 1
```

```
# id true;;
- : bool = true
```

```
# id (fun x -> x+1);;
- : int -> int = <fun>
```

OCamlでは型パラメタを  
'a や 'b で表す

# 多相関数の例

- 組の要素の抽出（射影）

```
# let fst (x, _) = x;;
val fst : 'a * 'b -> 'a = <fun>
# let snd (_, y) = y;;
val snd : 'a * 'b -> 'b = <fun>
```

# 多相関数の例

- 前回の課題から

- `twice` : ('a -> 'a) -> 'a -> 'a
- `repeat` : ('a -> 'a) -> int -> 'a -> 'a
- `fix` : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
- `fold_right` :  
    ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
- `fold_left` :  
    ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
- `append` : 'a list -> 'a list -> 'a list

# 型の明示

- (パターン : 型)
- (式 : 型)

型を修正していく -



```
# let id (x : int) = x;;
val id : int -> int = <fun>

# let id x = (x : int);;
val id : int -> int = <fun>
```

# ユーザ定義型

レコード型

ヴァリアント型

多相データ型

# ユーザの定義型

- レコード (record)
  - ≈ 組の各要素に名前がついたもの
  - C の struct に相当
- ヴァリアント (variant)
  - 何種類かの値うち一つをとる値
  - C の struct、union、enum の組み合わせに相当
  - 操作の安全性が型検査により保障される

# ユーザ定義型

レコード型

ヴァリアント型

多相データ型

# レコード型の定義、値の構成

~~Construct := 相当~~

- 定義: type 型名 = { フィールド名: 型; … }

```
# type complex =
    { re : float; im : float };;
type complex = { re : float; im : float; }
```

- 値の生成: { フィールド名 = 値; … }

```
# let ci = { re = 3.0; im = 4.0 };;
val ci : complex = {re = 3.;im = 4.}
```

# レコードのフィールドの使用

- フィールドの取り出し: 式 . フィールド名

```
# ci.re;;
- : float = 3.
```

- パターンマッチング

```
# let abs { re = r; im = i } =
    sqrt (r *. r +. i *. i);;
val abs : complex -> float = <fun>
# abs ci;;
- : float = 5.
```

絶対値の計算  
をしています。

# ユーザ定義型

レコード型

ヴァリアント型

多相データ型

# ヴァリアント型の定義

- type 型名 = タグ1 of 型1  
| タグ2 of 型2  
| :  
|

型名は小文字から  
タグは大文字から  
はじまる

- 例：bool と同等な型

```
# type mybool = False | True;;
```

- 例：簡単なインタプリタの値

```
# type value = VInt of int  
| VBool of bool
```

# ヴァリアント型の値の構成

## ○ 値の構成: タグ式

```
# True;;
- : mybool = True
# False;;
- : mybool = False
```

```
# VInt 1;;
- : value = VInt 1
# VBool true;;
- : value = VBool true
```

VInt は、  
intという型をうけとく。  
valueは、この型を返すやうに  
します。

# ヴァリアント型の値の分解

- パターンマッチングで

基本的な  
解法

→ 自分で解せ。

```
# let not x =
  match x with
  | True  -> False
  | False -> True;;
val not : mybool -> mybool = <fun>
# not False;;
- : mybool = True
```

# ヴァリアント型の値の分解

- パターンマッチングで

```
# let string_of_value x =
  match x with
  | VInt i    -> string_of_int i
  | VBool b   -> string_of_bool b;;
val string_of_value : value ->
string = <fun>

# string_of_value (VBool false);;
- : string = "false"
```

# ヴァリアント型の再帰的定義

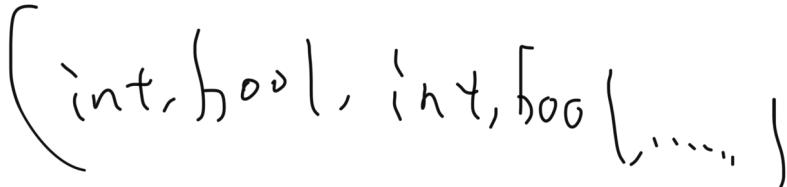
- 例：整数のリスト

```
# type int_list =
| Nil
| Cons of int * int_list;;
```

- 例：ノードに文字列を持つ二分木

```
# type str_tree =
| Leaf
| Node of string * str_tree * str_tree
;;
```

# 相互再帰的な型

- type … and … and …  

- 例：int と bool が交互に出現するリスト

```
# type ib_list = INil
          | ICons of int * bi_list
and   bi_list = BNil
          | BCons of bool * ib_list;;
```

```
# ICons (3, BCons (false, INil));;
- : ib_list = ...
```

# ユーザ定義型

レコード型

ヴァリアント型

多相データ型

# 多相データ型の必要性

- ノードの値の型ごとに二分木が考えられる

```
type str_tree =
| Leaf
| Node of string * str_tree * str_tree
```

```
type int_tree =
| Leaf
| Node of int * int_tree * int_tree
```

```
type bool_tree =
| Leaf
| Node of bool * bool_tree * bool_tree
```

- これらをまとめたい

# 多相的なヴァリアント型

- 型をパラメータに持つヴァリアント型

```
# type 'a tree =  
| Leaf  
| Node of 'a * 'a tree * 'a tree;;
```

# Node (3, Leaf, Leaf);;  
- : int tree = ...  
# Node (true, Leaf, Leaf);;  
- : bool tree = ...  
# Node(3, Node(true, Leaf, Leaf), Leaf);;  
Error: ...

# その上の関数

```
# let rec size x =
  match x with
  | Leaf -> 0
  | Node (_, l, r) ->
    1 + size l + size r;;
val size : 'a tree -> int = <fun>
```

```
# let rec sum x =
  match x with
  | Leaf -> 0
  | Node (x, l, r) ->
    x + sum l + sum r;;
val sum : int tree -> int = <fun>
```

# 複数の型パラメタ

```
# type ('a, 'b) either =
| L of 'a
| R of 'b
;;
```

# 副作用

例外

入出力

参照

副作用と型多相

# 副作用

- 式が「評価結果の値を返す」以外の振舞いをするとき、その式は副作用を持つという
- 副作用はなるべく使わない
  - コードが読みづらい
  - バグの元

# Unit 型

C {語彙の  
void的な - }

- () を唯一の値とする型

```
# ();;  
- : unit = ()
```

- 用途

- 引数が不要な関数に渡すダミー引数
- 返値に意味のない関数や式の返すダミーの値
  - 特に式に副作用のみがある場合

```
print_string : string -> unit  
read_line     : unit -> string
```

# 副作用

例外

入出力

参照

副作用と型多相

# 例外処理

- エラーが発生したとき、  
現在の計算を打ち切ってエラー処理用の  
コードにジャンプする機構
  - エラーの例：ゼロ除算、hd []、  
ユーザの入力がおかしい
- C++ やJava にも同様の機構がある

# 例外の生成と補足

- 生成 : `raise` 式
- 補足 : `try ... with` パターン -> 式
  - | パターン -> 式

```
# let div x y =
  if y = 0 then raise Division_by_zero
               else x / y;;
...
# div 8 0;;
Exception: Division_by_zero
# try Some (div 8 2) with Division_by_zero -> None;;
- : int option = Some 4
# try Some (div 8 0) with Division_by_zero -> None;;
- : int option = None
```

# ユーザ定義例外

## ○ exception 例外名 of 型

```
# exception My_exception;;
exception My_exception
# raise My_exception;;
Exception: My_exception

# exception My_str_exception of string;;
引数
# raise (My_str_exception "Hello");;
Exception: My_str_exception "Hello"
# try raise (My_str_exception "Hello")
  with My_str_exception s -> s;;
- : string = "Hello"
```

# 副作用

例外

入出力

参照

副作用と型多相

# 入出力

- `print_string` : `string` -> `unit`
- `read_line` : `unit` -> `string`

```
# print_string "Foo\n";;
Foo
- : unit = ()

# let s = read_line ();;
Bar
val s : string = "Bar"
```

# 副作用

例外

入出力

参照

副作用と型多相

# 参照 (reference)

- 中身を変更可能な「入れもの」
  - C や Scheme の変数のように再代入できる
- mutable record の特殊例
  - マニュアルを参照のこと

```
# let r = ref 0;;           参照の生成
val r : int ref = {contents = 0}
# !r ;;
- : int = 0               中身を取り出す
# r := 1;;
- : unit = ()             再代入
# !r ;;
- : int = 1
```

# 複数の式の逐次実行

- 式; 式; …; 式
  - 順番に式を評価し、最後に評価した式の結果を返す

```
# print_string "> "; read_line ();;  
> (入力待ち)
```

# 副作用

例外

入出力

参照

副作用と型多相

# 副作用と型多相

- let t = ref [] を考える
  - [] の型は 'a list
  - t の型はなんであるべきか？
- t : 'a list ref とすると型安全でない！

```
# let t = ref []
  let sum () = List.fold_right (+) (!t) 0;;
...
# sum ();;
- : int = 0
# t := [true]; sum ();;
???
```

# OCaml の解決策

- 「未決定な単相型」
  - `'_a` や `'_b` など
  - いったん型が決まると、他の型としては使えない

```
# let t = ref [];;
val t : '_a list ref = {contents = []}
# let sum () = List.fold_right (+) (!t) 0;;
...
# t;;
- : int list ref = {contents = []}
# t := [true];;
Error: ...
```

'\_a が int に決定

# 「未解決な単相型」がいつ必要か

- 問題は参照型以外にも

```
# let f () =
  let r = ref [] in
  fun x -> (r := x :: !r; !r);;
val f : unit -> 'a -> 'a list = <fun>

# let g = f () in (g 3; g true);;
???
```

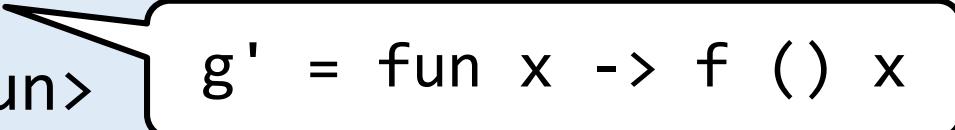
g : 'a -> 'a list ???

# OCaml の最終的な解決策

- Value Restriction

- 副作用がないことが確実である「値」にのみ多相的な型を与える
- 値である式：定数、fun式、それらの組など
- 値でない式：関数適用、参照、let式

```
# let f () x = x;;
val f : unit -> 'a -> 'a = <fun>
# let g = f ();;
val g : '_a -> '_a = <fun>
# let g' x = f () x;;
val g' : 'a -> 'a = <fun>
```



$g' = \text{fun } x \rightarrow f () x$

# 例題

理解の確認をするための課題です

提出の必要はありません

どれか一問を聞いてTAに見せることで出席とします

分からぬことがあつたら、積極的に質問しましょう

# 例題 1

- 2つの複素数の積を返す関数 prod を作れ  
 $\text{prod} : \text{complex} \rightarrow \text{complex} \rightarrow \text{complex}$
- ヴァリアント型 str\_tree の値を 3つ作れ
- ヴァリアント型 ib\_list の値を 3つ作れ

# 例題 2

- 型 iexpr を次のように定める

```
type iexpr =
| EConstInt of int
| EAdd          of iexpr * iexpr
| ESub          of iexpr * iexpr
| EMul          of iexpr * iexpr
```

- 式を評価する関数 eval を書け

eval : iexpr -> int

- オーバーフローは気にしなくてよい

# 例題 2 (つづき)

- 入出力例

```
# eval (EConstInt 10);;
- : int = 10
# eval (EAdd (EConstInt 10, EConstInt (-3)));;
- : int = 7
# let e1 = (EAdd (EConstInt 3, EConstInt 4));;
val e1 : iexpr = ...
# let e2 = (EMul (e1, EConstInt 8));;
val e2 : iexpr = ...
# eval (ESub (e2, EConstInt 6));;
- : int = 50
```

# 例題 3

- 整数を受け取って、  
ひとつ前に受け取った整数を返す関数を書け  
■ 初期値は 0

```
# let f = (適当な定義を書く) ;;
val f : int -> int = <fun>
# f 10;;
- : int = 0
# f (-5);;
- : int = 10
# f 3;;
- : int = -5
```

# レポート課題 2

締切：2019/5/7 13:00(JST)

# 問 1

- 自然数を表す型 nat を以下で定義する

```
type nat = Z | S of nat
```

以下の関数を書け

- 加算 add : nat -> nat -> nat
- 減算 sub : nat -> nat -> nat
- 乗算 mul : nat -> nat -> nat
- 累乗 pow : nat -> nat -> nat
- n2i : nat -> int
- i2n : int -> nat
  - 演算は int を経由しないこと

## 問 2

- 二分木を受け取り、以下に挙げる探索順で全要素を並べたリストを生成する関数を書け
  - 行きがけ順 (pre-order)
  - 通りがけ順 (in-order)
  - 帰りがけ順 (post-order)
- 二分木の定義は次の通り

```
type 'a tree =
| Leaf
| Node of 'a * 'a tree * 'a tree
```

親ノードを返す関数

## 問 3



- 二分木を受け取り、以下に挙げる探索順で全要素を並べたリストを生成する関数を書け
  - レベル順 (level-order)
    - 幅優先探索の順番
- 二分木の定義は問 2 と同じ

# 問 4

- int と bool 型の値を計算する単純な式 E の構文を以下で定める

```
E ::= 0 | 1 | 2 | ...
| E + E | E - E | E * E | E / E
| true | false
| E = E | E < E
| if E then E else E
```

- 評価結果の値の型を以下で定める

```
type value = VInt of int | VBool of bool
```

## 問4（つづき）

- このとき E の抽象構文器に対応する型 expr を定義せよ
  - 以下を完成させればよい

```
type expr =
| EConstInt    of int
| EAdd          of expr * expr
| ESub          of expr * expr
| ...
| EConstBool   of bool
| ...
```

# 問 5

- 前問で定義した式を評価する関数

eval : expr -> value

を定義せよ

- value は前問を参照
- expr は前問の定義

- 例外 Eval\_error を定義し、評価時のエラーが発生した場合には Eval\_error を投げること
  - bool を int に足す、など

# 発展 1

参照を使う方法

variantを使う←いいえ。



func g y

= g (lrg) y in

fix f : func  
(lrg) f .

- 関数  $f$  を受け取って、  
 $f$  を再帰的に無限回合成したもの を返す

関数 fix を、let rec を用いずに書け

- 参照を用いてもよい
- let rec による定義は次の通り

```
# let rec fix f x = f (fix f) x;;
val fix : (('a -> 'b) -> 'a -> 'b) ->
'a -> 'b = <fun>
```

Curry-Howard (-Lambek) 対応.

## 発展 2

直観論理記入

- 型 `false_t`, `not_t`, `and_t`, `or_t` を  
次のように定義する

```
type false_t = { t : 'a. 'a }
type 'a not_t = 'a -> false_t
type ('a, 'b) and_t = 'a * 'b
type ('a, 'b) or_t = L of 'a | R of 'b
```

- `false_t` 型の値は存在しない

# 発展 2 (つづき)

- 次のページの型について、  
それぞれの型を持つ式を
  - 再帰、副作用を用いずに定義できるか？  
できない場合は理由を述べよ
  - call/cc を許す場合はどうか  
 $\text{callcc} : ((\text{'a} \rightarrow \text{false\_t}) \rightarrow \text{'a}) \rightarrow \text{'a}$
- ただし以下を満たすこと
  - 定義する式の中で発生する例外は、  
その式中で補足すること
  - 標準ライブラリの関数を用いないこと

# 発展 2 (つづき)

（つづき）  
前回の通りでありますから、今日は、

1.  $('a \rightarrow 'b) \rightarrow ('b \rightarrow 'c) \rightarrow ('a \rightarrow 'c)$
2.  $('a, ('b, 'c) \text{ and\_t}) \text{ or\_t} \rightarrow$   
 $(('a, 'b) \text{ or\_t}, ('a, 'c) \text{ or\_t}) \text{ and\_t}$
3.  $(('a, 'b) \text{ or\_t}, ('a, 'c) \text{ or\_t}) \text{ and\_t} \rightarrow$   
 $('a, ('b, 'c) \text{ and\_t}) \text{ or\_t}$
4.  $('a, 'a \text{ not\_t}) \text{ or\_t}$ 
  - もしくは  $('a \rightarrow 'c) \rightarrow ('a \text{ not\_t} \rightarrow 'c) \rightarrow 'c$
5.  $('a, 'a \text{ not\_t}) \text{ and\_t}$ 
  - もしくは  $('a \rightarrow 'a \text{ not\_t} \rightarrow 'c) \rightarrow 'c$
6.  $(('a \rightarrow 'b) \rightarrow 'a) \rightarrow 'a$ 
  - cf. Peirce 則

## 発展 2 (つづき)

- 残念ながら OCaml には call/cc がない
- 課題では call/cc の型のみが大切なので  
次を call/cc の実装に代えて使う

```
let rec callcc
  : (('a -> false_t) -> 'a) -> 'a
  = fun f -> callcc f
```

- OCaml に call/cc を加えた実装もある

# 補足

- 部分関数（停止しなかったり、例外を投げたりする関数）を使えば簡単

```
# let rec f x = f x;;
val f : 'a -> 'b = <fun>
# let f x = let Some y = None in y;;
val f : 'a -> 'b = <fun>
# let f x = failwith "Hey!"
val f : 'a -> 'b
```

# 発展 3

- ' $a * b \rightarrow c$ ' 型の関数を  
 $'a \rightarrow 'b \rightarrow 'c$  型の関数とみなすことを  
**Curry化 (Currying)** といい、  
逆を **uncurry** という

```
# let curry f x y = f (x, y);;
val curry
  : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# let uncurry f (x, y) = f x y;;
val uncurry
  : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

# 発展 3 (つづき)

- 実は OCaml においては  
curry と uncurry は逆演算ではない
  - $\text{uncurry} (\text{curry } f) = f$  だが  
 $\text{curry} (\text{uncurry } f) = f$  ではない
- 次の式が違う値を返すような  $h$  と  $f$  を作れ

```
# h f;;
```

```
# h (curry (uncurry f));;
```

# 発展 3 (つづき)

- つまり、以下の左右の結果の異なる適切な式 E1 と E2 を探せ
  - ocaml のインタプリタを新に起動して環境をリセットしていることに注意

```
> ocaml  
# let f = E1;;  
# let h = E2;;  
# h f;;
```

```
> ocaml  
# let f = E1;;  
# let h = E2;;  
# h (curry (uncurry f));;
```

# おまけ：高ランク多相

- ML の型推論器は引数は単相型だと仮定する

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# (id 1, id 2.0, id true);;
- : int * float * bool = (1, 2., true)

# let f h = (h 1, h 2.0, h true);;
Characters 18-21:
let f h = (h 1, h 2.0, h true);;
                           ^^^
```

Error: This expression has type float but  
an expression was expected of type int

# おまけ：高ランク多相

- 多相関数を引数にしたいときは？

## 高ランク多相

- (ひとつの) 解決策はレコードを使うこと

```
# type t = { h : 'a. 'a -> 'a };;
type t = { h : 'a. 'a -> 'a; }

# let f x = (x.h 1, x.h 2.0, x.h true);;
val f : t -> int * float * bool = <fun>

# f { h = fun x -> x };;
- : int * float * bool = (1, 2., true)
```

# おまけ：前回の発展 3

- "普通" にやっては型が付かない
  - 高ランク多相を使うと "普通の手法" に型が付く

```
type church = { t : 'a. ('a -> 'a) -> 'a -> 'a }
```

- このとき `sub : church -> church -> church`
- 厳密には問題の要件を満たさない
- 他の OCaml の機能を使う
  - 整数型とその演算（とてもつまらない方法）
  - リスト（それなりに面白い方法もある？）
  - 参照