

関数・論理型プログラミング実験

第3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
- 名前/学籍番号/希望アカウント名をメールを
 tsukada@kb.is.s.u-tokyo.ac.jp
 までメールしてください。
 - 件名は「FL/LP実験アカウント申請」
 - アカウント名/パスワードを返信
 - PCからのメールを受け取れるように

今日の内容

- OCaml のモジュールシステムについて
 - Structure
 - Signature
 - Functor
- OCaml の (分割) コンパイル

モジュール 分割コンパイル

モジュールシステム

Structure

Signature

Functor

大規模ソフトウェアの プログラミングは難しい

- 人の記憶できるプログラムの量には
限度があるから
- OCaml の処理系のソースプログラム全てを記憶し
ている人は（多分）いない
- Linux カーネルのソースプログラム全てを記憶して
いる人は（多分）いない

Q: ではどうするか？

- A: 複数人でプログラムする
- Q: どれくらい効率的？
 - 10人でやれば一人あたり $1/10$ の作業量
 - 100人でやれば一人あたり $1/100$ の作業量
 - 10000人でやれば一人あたり $1/10000$ の作業量
 - ...

Q: ではどうするか？

- A: 複数人でプログラムする
- Q: どれくらい効率的？
 - 10人でやれば一人あたり $1/10$ の作業量
 - 100人でやれば一人あたり $1/100$ の作業量
 - 10000人でやれば一人あたり $1/10000$ の作業量
 - ...

にはならない

最悪のシナリオ

- 他人の書いたコードなんて読めるか！
- 自分で書いた方が早い



- 似たようなプログラムが大量にできあがる



- プログラムの改善・保守が困難になる
 - 似たプログラムを全て修正する必要がある
 - 修正が及ぶ影響が予測できない

どう避ける？

- プログラムを「モジュール化」する
 - モジュール：再利用可能なプログラム部品
- モジュールの仕様と実装を切り分ける

仕様と実装を切り分ける

- 仕様：モジュールの外からの使われ方
 - どんな関数がある？
 - それらの型は？
 - ...
- 実装：仕様の具体的な実現方法
 - この関数の定義は...

なぜ仕様と実装を分離？

- モジュールの利用が容易に
 - 利用者は仕様だけ見ればよい
- モジュールの実装の修正が容易に
 - モジュールの仕様さえ守ればよい

OCaml のモジュールシステム

- Structure
 - モジュールの実装
 - 名前空間の切り分け
- Signature
 - Structure の仕様
 - 提供する型と、関数名およびその型
- Functor
 - Structure から Structure を作る関数のようなもの

モジュールシステム

Structure

Signature

Functor

Structure

モジュール名を定義.

- モジュールの実装を定義

- 構文

```
module モジュール名 =  
    struct 内容 end
```

モジュールオブジェクトを作成する.
内容を入力する.

- 内容の部分に型や関数の定義を書く
- モジュール名の先頭は大文字

例：多重集合

モジュールの主役となる型は
慣習的に t と名づける

```
module Multiset =
  struct
    type 'a t = 'a list
    let empty = []
    let add a xs = a::xs
    let rec remove a xs =
      match xs with
      | [] -> []
      | y::ys -> if a = y then ys
                   else y::(remove a ys)
    let rec count_sub a xs k =
      match xs with
      | [] -> k
      | y::ys -> if a = y then count_sub a ys (k+1)
                   else count_sub a ys k
    let count a xs = count_sub a xs 0
  end
```

empty は空集合を表し,
list が Multiset 型だという
ことを示す。

この定義は、
「この型は Multiset の要素」とい
うことを示す。

Structure の使い方

- 中の型や関数などを使うには:

モジュール名 . 型名
モジュール名 : 変数名

```
# let e = Multiset.empty;;
val e : 'a list = []

# let s = Multiset.add 5 e;;
val s : int list = [5]

# Multiset.count 5 s;;
- : int = 1
```

モジュール名の省略

- open することでモジュール名を省略可

```
open モジュール名
```

```
# open Multiset;;
# let s = add 5 empty;;
val s : int list = [5]
# count 5 s;;
- : int = 1
```

標準ライブラリのモジュール

- List, String, Printf, ...
 - 詳しくはマニュアルの Part IV 参照

```
# List.length [1; 2; 3];;
- : int = 3

# String.sub "abcde" 2 3;;
- : string = "cde"

# Printf.printf "%04d %s\n" 12 "XXX";;
0012 XXX
- : unit = ()
```

モジュールシステム

Structure

Signature

Functor

Signature

- モジュールのインターフェース
 - Signature に書いた型や関数のみが外部から利用可
 - モジュールの「型」
- 構文

```
module type シグニチャ名 =
    sig 内容 end
```

- 内容部分に型の宣言や関数の型を書く
- シグニチャ名の先頭は慣習的に大文字

C言語ひいう.h的なもの イメージ\

例：多重集合

```
module type MULTISSET =
sig
  type 'a t
  val empty : 'a t
  val add : 'a -> 'a t -> 'a t
  val remove : 'a -> 'a t -> 'a t
  val count : 'a -> 'a t -> int
end
```

Signature の適用

- Signature を structure に当て嵌める

```
module モジュール名 : シグニチャ  
    = 元モジュール
```

```
module モジュール名  
    = (元モジュール : シグニチャ)
```

- 実体は元モジュールと同じ
- モジュール外からは signature で示された型や関数しか利用できない

例

```
# module AbstMultiset : MULTISSET
= Multiset;;
module AbstMultiset : MULTISSET
# AbstMultiset.empty;;
- : 'a AbstMultiset.t = <abstr>
# AbstMultiset.add 1 Abstmultiset.empty;;
- : int AbstMultiset.t = <abstr>
```

さきは 'a t [first, tail],
リストで実装されてるところ
2- フラグは使へなくなつた

実体が
外部から隠匿

list であることが
外部から隠匿

これに付してできることは
AbstMultiset で定義された
もののみ。

例（つづき）

- count_sub は MULTISET にないので利用不可

```
# AbstMultiset.count_sub;;
Error: Unbound value AbstMultiset.count_sub
```

- 実体は同じでも違う型とみなされる

```
# AbstMultiset.add 0 Multiset.empty;;
  AbstMultiset.add 0 Multiset.empty
                           ^^^^^^^^^^^^^^
```

```
Error: This expression has type 'a list but an
expression was expected of type int
AbstractMultiset.t
```

相互再帰的なモジュール

- 相互再帰的なモジュールも定義できる

```
module rec モジュール名 : シグニチャ = struct 内容 end  
and モジュール名 : シグニチャ = struct 内容 end ...
```

```
module rec Even : sig val f : int -> bool end =  
  struct  
    let f n = if n = 0 then true else Odd.f (n-1)  
  end  
and Odd : sig val f : int -> bool end =  
  struct  
    let f n = if n = 0 then false else Even.f (n-1)  
  end
```

モジュールシステム

Structure

Signature

Functor

Functor

- モジュールを受け取りモジュールを返す
関数のようなもの

```
functor (仮引数 : シグニチャ)  
    -> モジュール
```

例：多重集合

```
type order = LT | EQ | GT
module type ORDERED_TYPE = 型を定めた抽象型
sig
  type t
  val compare : t -> t -> order
end このモジュール で用いる型を定義。

module Multiset2 =
  functor (T: ORDERED_TYPE) -> struct
    type t = T.t list
    let rec remove a xs =
      match xs with
      | [] -> []
      | y::ys -> (match T.compare a y with
                    | EQ -> ys
                    | _ -> y :: (remove a ys))
    (* 以下略 *)
  end
```

例：多重集合

```
type order = LT | EQ | GT
module type ORDERED_TYPE =
sig
  type t
  val compare : t -> t -> order
end

module Multiset2 (T: ORDERED_TYPE) =
struct
  type t = T.t list
  let rec remove a xs =
    match xs with
    | [] -> []
    | y::ys -> (match T.compare a y with
                  | EQ -> ys
                  | _ -> y :: (remove a ys))
  (* 以下略 *)
end
```

Functor の適用

- Functor にモジュールを渡す

ファンクタ（モジュール）

- 括弧は必要

例

```
module OrderedString =
  struct
    type t = string
    let compare x y =
      if x < y then LT
      else if x > y then GT
      else EQ
  end

  module StringMultiset = Multiset2 (OrderedString)
```

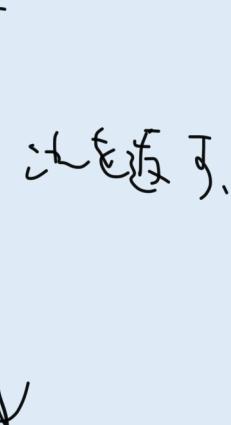
Functor に対する signature

- Functor にも signature が作れる

```
functor ( 仮引数 : シグニチャ )  
  -> シグニチャ
```

```
module type MULTISSET2 =  
  functor (T: ORDERED_TYPE) ->  
    sig  
      type t  
      val empty : t  
      val add   : T.t -> t -> t  
      val remove : T.t -> t -> t  
      val count  : T.t -> t -> int  
    end
```

ORDERED_TYPEを引数に,
↓



OCaml の (分割) コンパイル

OCaml のコンパイラ

- 二種類
 - ocamlc : バイトコードコンパイラ
 - OCaml 仮想マシン (OCaml バイトコードインタプリタ)用コードを生成
 - ocamlopt : ネイティブコードコンパイラ
 - x86など、実際のマシン用コードの生成
- モジュール単位での分割コンパイルが可能

FILES

- ソースファイル

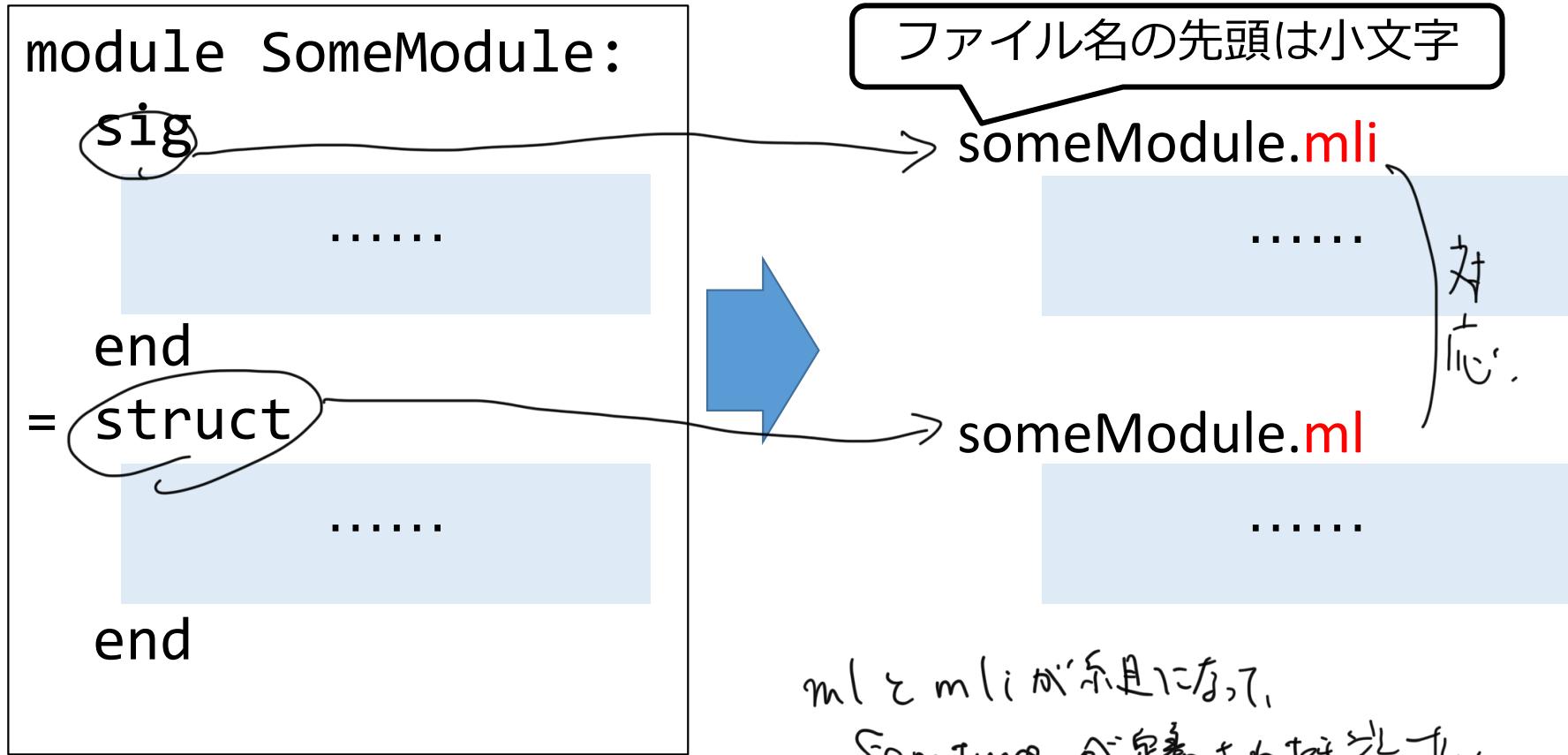
- .ml モジュールの実装
- .mli モジュールのシグニチャ

- オブジェクトファイル

- .cmo 実装のバイトコード
- .cmi I/F のバイトコード
- .o 実装のネイティブコード
- .cmx .o の附加情報
- .a, .cma, .cmxa ライブラリ

モジュールと分割コンパイル

- モジュールの signature と structure を別のファイルとして分割コンパイルできる



モジュールの分割コンパイル

- .mli ファイルをコンパイル
 - .cmi が生成される
- .ml ファイルを ocamlc でコンパイル
 - .cmo が生成される
 - .mli ファイルがあれば .cmi を用いて型検査
- .ml ファイルを ocamlopt でコンパイル
 - .cmx と .o が生成される
 - .mli があれば .cmi を用いて型検査

.mli, .ml によるモジュールの例

- strSet.ml, strSet.mli
 - 文字列の順序付き多重集合のモジュールの定義
- sort.ml
 - 上で定義される StrSet モジュールを利用し、ソートを行うプログラム本体

サポートページよりダウンロード可能

分割コンパイルの例

順番が大事 : mliが先

```
$ ocamlc -c strSet.mli
$ ocamlc -c strSet.ml
$ ocamlc -c sort.ml
$ ls -F *.cm*
sort.cmi sort.cmo strSet.cmi strSet.cmo
$ ocamlc -o sort strSet.cmo sort.cmo
$ ls -F sort
sort
```

順番が大事 : sort.ml で
StrSet を使っているので、
strSet.cmo が先

sortの実行例

```
$ ./sort <<END
```

```
> bbb  
> ccc  
> aaa  
> bbb  
> END
```

```
aaa  
bbb  
bbb  
ccc
```

.cmo をインタプリタで使う

```
$ ocaml
          OCaml version 4.01.0

# #load "strSet.cmo";;
# StrSet.empty ;;
- : StrSet.t = <abstr>

# StrSet.count_sub ;;
Error: Unbound value StrSet.count_sub

# open StrSet;;
# add "abc" empty;;
- : StrSet.t = <abstr>
```

注意

- レポート課題で実行にコンパイルが必要なら、ビルド方法の記述も提出すること
 - Makefile を用いてもよい
 - OCamlMakefile を用いてもよい
 - どちらの場合も「makeせよ」とは書くこと

例題

理解の確認をするための課題です

どれか一問を聞いてTAに見せることで出席とします

分からぬことがあれば、積極的に質問しましょう

例題

- 「整数の多重集合」のモジュールを以下の手順で作れ
 - signature ORDERED_TYPE を満たすように module OrderedInt を作る
 - 作ったモジュールに ファンクタ Multiset2 を適用

Multiset2 のコードは
サポートページから入手できる

例題（つづき）

- 2を1個、5を2個 要素を持つ多重集合を作れ
- 作った多重集合が5を2個要素を持つことを確認せよ

例題（つづき）

```
module OrderedInt =  
  struct  
    type t = int  
    let compare x y = (* ここに適切な実装を書く *)  
  end  
  
module IntMultiset = Multiset2 (OrderedInt)  
(* IntMultiset を使って多重集合を作る *)
```

レポート課題 3

締切：2019/5/14 13:00(JST)

ソートコンパイルを試してみよ。

問 1

- sort の例を自分で試せ
 - 例にそって実行ファイルを生成・実行せよ
 - .cmo をインタプリタで利用せよ
 - .mli をコンパイルしないとどうなるか
 - 最後のリンク時にファイルの順番を変えるとどうなるか
 - OCamlMakefile を用いてみよ
 - その他いろいろ試してみよ

ターゲットを変えてみよ。

※ 今後課題で Makefile / OCamlMakefile を用いてもよい

問 2

- スタックを扱うモジュールを実装せよ
 - 以下の関数をサポートすること
 - pop : 'a スタック型 -> ('a * 'a スタック型)
 - push : 'a -> 'a スタック型 -> 'a スタック型
 - empty : 'a スタック型
 - size : 'a スタック型 -> int
 - シグニチャを適切に与え抽象化すること
 - スタックの実装を 'a list から
'a list * int に変えててもよいように

 $\overbrace{\quad}^{\text{数をハサウメーション}}$

問 3

- signature MULTISET2 を持つ
ファンクタを実装せよ
- 配布したものと異なる実装であること
(例えば 2 分木を用いた実装)

問 4

- 連想配列を扱うモジュールを作成せよ
 - 連想配列のキーは
ORDERED_TYPE で表される型とする
 - 参照は用いないこと
 - 利用するのに十分なだけの関数を用意すること
 - empty, add, remove, lookup など

問 5

- functor を用いて、次のシグネチャから行列とベクトルの演算を定義するモジュールを作成せよ

```
module type SEMIRING = sig
  type t
  add : t -> t -> t
  mul : t -> t -> t
  unit : t
  zero : t
end
```

問5 (つづき)

~~(F)~~ ~~(A)~~)

- 作った functor を利用して、
次を要素を持つような行列モジュールを作れ
 - 加算が or 、乗算が and な真偽値
 - 加算が min 、乗算が + な 整数 $\cup \{\infty\}$

tropical semi ring

- 得られたモジュールを利用した計算を行ってみよ

$(\text{F}, \text{A}, \text{B}, \dots, \infty)$
 $\left[\begin{smallmatrix} \text{F}, \text{A}, \text{B}, \dots \\ \text{C} \end{smallmatrix} \right]$

補足

- 半環 $(R, 0, 1, +, \times)$ に対して、
行列 $(R^{m,m}, |^{m,m}, \cdot)$ はモノイド
 - 「matrix semiring」 等で検索すれば多数の例が見つかる

例：最短路の長さ

- $M_{i,j}$: i と j の間の枝の重み
 - 枝がなければ∞
- $M_{i,j}$ は半環 $(Z \cup \{\infty\}, \infty, 0, \min, +)$ 上の行列
- $(M_{i,j})^{|V|}$ が各点間の最短路の長さ
 - $(M_{i,j})^n$ は高々 n 点を経由するときの最短路の長さ
 - 注意：素直に実装すると $O(n^3 \log n)$ かかる
 - もっと工夫をすると Floyd-Warshall 法になる

$\forall t_1 t_2 \in \text{equal}$

発展 1

⇒ 今は “ $t_1 = t_2$ の正誤”
いじる。

○以下のシグネチャ EQ を持つ

モジュール Eq を定義せよ

- ただし、各関数は呼ばれれば停止し、例外が発生しないようにすること

```
module type EQ = sig
  type ('a, 'b) equal
  val refl : ('a, 'a) equal
  val symm : ('a, 'b) equal -> ('b, 'a) equal
  val trans :
    ('a, 'b) equal -> ('b, 'c) equal -> ('a, 'c) equal
  val apply : ('a, 'b) equal -> 'a -> 'b
end
```

$a = b$ の正誤を調べ
二型ペアで

正誤判定マージ

apply の実現が何をするか(セミト)

発展 1 (つづき)

ただし、割り算はなし
(0除算を避けるため)

- 前回の問4および問5の簡単な言語を Eq を用いて以下のように定義する

```
type 'a value =
| VBool of (bool, 'a) Eq.equal * bool
| VInt of (int, 'a) Eq.equal * int
type 'a expr =
| EConstInt of (int, 'a) Eq.equal * int
| EAdd of (int, 'a) Eq.equal * int expr * int expr
| EIF of bool expr * 'a expr * 'a expr
| EEq of (bool, 'a) Eq.equal * int expr * int expr | ...
```

- この式を評価する関数 eval を定義せよ

- eval : 'a expr -> 'a value
- 例外を発生させないこと (**実行時エラーはない**)

発展 1 (つづき)

- 関数定義は次のように型を陽に書くとよい

```
let rec eval : 'a. 'a expr -> 'a value = ...
```

- 式を作るには次のようにする

```
let c1 = EConstInt (Eq.refl, 1)
let c2 = EConstInt (Eq.refl, 2)
let add12 = EAdd (Eq.refl, c1, c2)
let ct = EConstBool (Eq.refl, true)
let cif = EIF (ct, add12, c2)
```

補足

- この工夫によって型が整合しない式を除ける

```
# let c1 = EConstInt (Eq.refl, 1);;
val c1 : int expr = EConstInt (<abstr>, 1)

# let ct = EConstBool (Eq.refl, true);;
val ct : bool expr = EConstBool (<abstr>, true)

# let cadd = EAdd (Eq.refl, c1, ct);;
let cadd = EAdd (Eq.refl, c1, ct)
          ^^
```

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

補足

- OCaml 4.00 以降では、
このようなことは GADT を用いて行える
- マニュアルの "Language Extensions" を参照

発展 2 (さらに進んだEQ)

- 次のシグネチャ EQ2 を持つ
モジュール Eq2 を定義せよ
 - ただし、各関数は呼ばれれば停止し、
例外が発生しないようにすること

```
module type EQ2 = sig
  type ('a, 'b) equal
  val refl : ('a, 'a) equal
  val symm : ('a, 'b) equal -> ('b, 'a) equal
  val trans :
    ('a, 'b) equal -> ('b, 'c) equal -> ('a, 'c) equal
  val apply : ('a, 'b) equal -> 'a -> 'b
  module Lift : functor (F: sig type 'a t end) -> sig
    val f : ('a, 'b) equal -> ('a F.t, 'b F.t) equal
  end
end
```