Objective Caml による

プログラミング

Jacques Garrigue
Kyoto University
garrigue@kurims.kyoto-u.ac.jp

なぜ Objective Caml？

数多くある言語の中にObjective Camlを選ぶ理由

- 型推論と多相性で型のついた簡潔なプログラムが書ける
- パターンマッチングやクロージャなど、豊富な表現力
- 当然GC(自動メモリ管理)がある
- タイプチェッカーはうるさいが、実行時に型エラーがなく
  コンパイルできるプログラムにはバグがない
  という気持にさせられる

以上の性質は型付関数型言語の共通項である。このチュートリ

アルの内容のほとんどはStandard MLやHaskellにも応用できる

このチュートリアルの内容

□OCamlプログラミングの基礎

□ヴァリアント(直和型)の基本

□汎関数のすすめ

□応用例 ： 数式の処理とパージング

□型システム探検 ： 多相ヴァリアント

□抽象型、オブジェクト、ファンクター

OCaml プログラミングの基礎 ： 関数

基礎 ： 定義と型推論

定義にはletを使う

```
let x = 1 + 2 ;;
val x : int = 3
```

チュートリアルでは、例示のために、対話型コンパイラを使う

灰色 はプログラマの入力　「;;」で区切る

橙色 はコンパイラの出力　「val」または「-:」で始まる

　　推論された型 および 計算結果 を示す

```
let pair = (3, "three");;
val pair : int * string = (3, "three")
```

□これはintとstringの対を定義している

基礎 ： 関数と適用

関数適用は並置で書く

```
let f x y = x+y+1 ;;
val f : int -> int -> int = <fun>
f 10 18;;
- : int = 29
```

関数も値であり、以下のようにも書ける

```
let f = fun x y -> x+y+1 ;;
val f : int -> int -> int = <fun>
(fun x y -> x+y+1) 10 18;;
- : int = 29
```

基礎 ： 再帰とループ

最大公約の計算　let rec は再帰的な定義を表わす

```
let rec gcd x y =
  if x > 0 then gcd (y mod x) x else y ;;
val gcd : int -> int -> int : <fun>
```

破壊的代入を使ったバージョン

```
let gcd' x y =
  let m = ref x and n = ref y in
  while !m > 0 do let x = !m in m := !n mod x; n := x done;
  !n ;;
val gcd' : int -> int -> int = <fun>
```

- 多くの場合では、再帰の方が短かくてかつ証明しやすい
- ループは入出力など、元々副作用が避けられないときに有効

基礎 ： 再帰関数

ifの代わりにパターンマッチングを使った例

```
let rec fact = function
  | 0 | 1 -> 1
  | x -> x * fact (x - 1) ;;
val fact : int -> int
```

相互再帰の例

```
let rec up x =
  if x mod 2 <> 0 then up (x*2) else down (x+1)
and down x =
  if x < 10 then x else up (x/3) ;;
val up : int -> int = <fun>
val down : int -> int = <fun>
```

基礎 ： 型推論と多相性

関数の定義が引数の型に依存しない場合、多相型が得られる

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

'aや'bは型変数であり、任意の型に変えられる

もっと数学的に書くと ∀'a'b. 'a * 'b -> 'a になる

利用時も正しい型が自動的に推論される

```
fst ("France", 33) ;;
- : string = "France"
fst (42, true) ;;
- : int = 42
```

このときのfstの型はそれぞれ

string * int -> string と int * bool -> int

基礎 ： 汎関数

関数が値なので、自由に引数として渡せる

```
let map_array f arr =
  let len = Array.length arr in
  let arr' = Array.create len (f arr.(0)) in
  for i = 1 to len-1 do arr'.(i) <- f arr.(i) done;
  arr' ;;
val map_array : ('a -> 'b) -> 'a array -> 'b array

let scalar_prod x v = map_array (fun y -> x*y) v ;;
val scalar_prod : int -> int array -> int array
```

- □よく使うコードは一回定義するだけでいい
- □間違える心配がなくなる
- □副作用に頼る部分は隠蔽できる

基礎 ： 関数の型を読む

コンパイラが推論した型は関数について多くのことを教える

val map_array : ('a -> 'b) -> 'a array -> 'b array

- map_array は2つの引数を取る
- その1つ目は関数である
- 'b arrayの要素は'a arrayからその関数を使って計算される

  'a array以外に'a型の値はないし、他に'b型の値を作る方法もない

バグも発見できる

val map_array : ('a -> 'a) -> 'a array -> 'a array

- 多相性不足。多分、入力の要素をそのまま出力に使っている

val map_array : ('a -> 'b) -> 'a array -> 'c array

- 多相性過剰。'c型の値が作れないので、結果がからっぽ

ヴァリアント(直和型)の基本

ヴァリアント(直和型)

関数型言語における中心的なデータ構造

- 自由代数の理論に基づく
- パターンマッチングの利用で強力に

型安全性(バグ防止)

- C のunionと違い、タグと中身の関係は明示的
- 中身の型を間違えることはない

パターンマッチング

- タグの確認と中身の抽出を同時に行う(冗長性がない)
- 完全性の検査 (忘れたタグがないか)
- 冗長性の検査 (利用されない場合)

ヴァリアント ： リストの例

中身のない Nil と頭部と後部を持つ Cons

中身の型が未定なので多相型

```
type 'a list = Nil | Cons of 'a * 'a list
```

```
let l = Cons (1, Cons (2, Nil))
val l : int list
```

パターンマッチングは型定義に似ている

```
let rec length = function
    Nil -> 0
  | Cons(hd, tl) -> 1 + length tl
val length : 'a list -> int
```

ヴァリアント： 汎関数のすすめ

汎関数を使って再利用可能な反復関数を定義できる

```
let rec map f = function
    Nil -> Nil
  | Cons(hd, tl) -> Cons (f hd, map f tl)
val map : ('a -> 'b) -> 'a list -> 'b list

map (fun x -> x+1) l ;;
- : int list = Cons (2, Cons (3, Nil))
```

□中身に依存しないので、多相型
□定義は一回だけで済む
□型推論があるので、利用が簡単
□プログラムが構造的な部分と論理的な部分に分けられる

ヴァリアント： コード変更の支援

リストの定義を拡張して見よう

```
type 'a list =
    Nil | Cons of 'a * 'a list | Append of 'a list * 'a list
```

すると、Appendを扱わない全てのパターンマッチングが指摘される

```
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Append (_, _)
```

指摘された個所に必要なコードを追加すればいいだけ

```
  | Append(l1, l2) -> length l1 + length l2
```

ヴァリアント： まとめ

安全なプログラミングを支援
- □型安全性
- □パターンマッチングで様々な検査

コードの再利用をすすめる
- □汎関数による構造と論理の分解
- □コードの変更も支援される

でもタグは一個の型のみに関連付けられる
- □型を拡張する場合、コードを変更しなければならない
- □このために、多相ヴァリアントも提供される (後半参照)

応用例 ： 数式を処理する

数式を処理する ： 型の定義

簡単な数式を考える

式 ::= 整数 | 変数 | 式 ＋ 式 | 式 × 式 | (式)

例

5 * 2 + 3　　(3 + y) * 12

型を定義する

```
type expr =
    Num of int
  | Var of string
  | Plus of expr * expr
  | Mult of expr * expr
```


数式を処理する ： 基本操作

汎関数を定義する

```
let map_expr f e =
  match e with
  | Num _ | Var _ -> e
  | Plus (e1, e2) -> Plus (f e1, f e2)
  | Mult (e1, e2) -> Mult (f e1, f e2)
val map_expr : (expr -> expr) -> expr -> expr
```

変数の代入

```
let rec subst env = function
  | Var x when List.mem_assoc x env -> List.assoc x env
  | e -> map_expr (subst env) e
val subst : (string * expr) list -> expr -> expr
```

□再帰的でないmapも役に立つ

数式を処理する ： 計算

式の評価 map_exprとevalの「相互」再帰

```
let rec eval e =
  match map_expr eval e with
  | Plus (Num x, Num y) -> Num (x + y)
  | Mult (Num x, Num y) -> Num (x * y)
  | e' -> e'
val eval : expr -> expr = <fun>
```

例

```
let e = subst ["x", Num 3; "y", Var "x"]
            (Plus (Var "y", Mult (Var "x", Num 2)));;
val e : expr = Plus (Var "x", Mult (Num 3, Num 2))
let e' = eval e;;
val e' : expr = Plus (Var "x", Num 6)
```


数式を処理する ： 印刷

Formatモジュールで強力なプリティプリンタが作れる

```
let rec print_expr ?(prio=0) ppf e =
  let printf fmt = Format.fprintf ppf fmt in
  match e with
  | Num x -> printf "%d" x
  | Var x -> printf "%s" x
  | Mult (e1, e2) ->
      printf "@[%a *@ %a@]" (print_expr ~prio:1) e1
        (print_expr ~prio:1) e2
  | Plus (e1, e2) as e ->
      if prio > 0 then (printf "(%a)") print_expr e else
      (printf "@[%a +@ %a@]") print_expr e1 print_expr e2
val print_expr : ?prio:int -> Format.formatter -> expr -> unit
```

□?prio: はオプショナルなラベルであり、?(prio=0)はデフォルト引数である

□それを正しく処理するにはここで (printf <fmt>) を括弧でくくる必要がある

数式を処理する ： 印刷

Formatは適切なところで改行し、インデントもする

```
print_expr Format.std_formatter (big 10);;
((2 + 2) * (5 + 3 + 7) + 3 * (8 + 1) + 5 + 8 + 5) *
((9 + 6 + 7) * (3 * (8 + 5) + 4 + 0 + 0) +
 (6 + 7) * (6 + 6 + 1) + 5 * (8 + 8) + 0 + 2 + 2)
```

トップレベルのプリンタとして設定することもできる

```
let print_expr' ppf = print_expr ppf;;
val print_expr' : Format.formatter -> expr -> unit
#install_printer print_expr';;
e;;
- : expr = x + 3 * 2
```


パージングについて

Objective Caml では様々な方法で入力を解析できる

- 手で解析: input_line と Str(regexp) モジュール
  PERL風に一行ずつ処理する
- 固定フォーマット： scanf
  完全に固定されたデータに便利
- stream (camlp4でプリプロセス)
  手軽に本格的なパージング
- 正攻法： ocamllex と ocamlyacc
  伝統的なLALRパージング
  速度が早い

パージング ： scanf

```
let input_point ic =
  Scanf.sscanf (input_line ic)  " %f %f %f" (fun x y z -> (x,y,z));;
val input_point : in_channel -> float * float * float = <fun>
let input_data ic =
  let rec loop accu n =
    if n = 0 then List.rev accu else
    loop (input_point ic :: accu) (n-1)
  in Scanf.sscanf (input_line ic) " %d" (loop [])
val input_data : in_channel -> (float * float * float) list = <fun>

input_data stdin;;
2
1 1.4 5
1.3 2 3
- : (float * float * float) list = [(1., 1.4, 5.); (1.3, 2., 3.)]
```


パージング ： stream パーザ

簡単な字句解析器が提供されている

```
#load"camlp4o.cma";;
        Camlp4 Parsing version 3.07+beta 2
open Genlex;;
let lexer = Genlex.make_lexer ["+";"*";"(";")"]
val lexer : char Stream.t -> Genlex.token Stream.t = <fun>
```

streamマッチングによるパーザの動作

```
let s = lexer (Stream.of_string "1 2 3 4");;
val s : Genlex.token Stream.t = <abstr>
(parser [< ' x >] -> x) s ;;
- : Genlex.token = Int 1
(parser [< 'Int 1 >] -> "ok") s ;;
Exception: Stream.Failure
(parser [< 'Int 1 >] -> "one" | [< 'Int 2 >] -> "two") s ;;
- : string = "two"
```

stream パーザ ： 汎関数のすすめ

リストをパーズしながら結果を蓄積

```
let rec accumulate parse accu = parser
  | [< e = parse accu; s >] -> accumulate parse e s
  | [< >] -> accu
val accumulate : ('a -> Genlex.token Stream.t -> 'a) ->
  'a -> Genlex.token Stream.t -> 'a
```

左結合の演算子を定義

```
let left_assoc parse op wrap =
  let parse' accu =
    parser [< 'Kwd k when k = op; s >] -> wrap accu (parse s) in
  parser [< e1 = parse; e2 = accumulate parse' e1 >] -> e2
val left_assoc : (Genlex.token Stream.t -> 'a) ->
  string -> ('a -> 'a -> 'a) -> Genlex.token Stream.t -> 'a
```

問 同じ優先順位の演算子が複数あればどうすればいい？

数式を処理する ： stream パーザ

主関数

```
let rec parse_simple = parser
  | [< 'Int n >] -> Num n
  | [< 'Ident x >] -> Var x
  | [< 'Kwd"("; e = parse_expr; 'Kwd")" >] -> e
and parse_mult s =
  left_assoc parse_simple "*" (fun e1 e2 -> Mult(e1,e2)) s
and parse_expr s =
  left_assoc parse_mult "+" (fun e1 e2 -> Plus(e1,e2)) s
val parse_simple : Genlex.token Stream.t -> expr = <fun>
val parse_mult : Genlex.token Stream.t -> expr = <fun>
val parse_expr : Genlex.token Stream.t -> expr = <fun>
```

数式を処理する ： stream パーザ

文字列からのパーザ

```
let parse_string s =
  match lexer (Stream.of_string s) with parser
    [< e = parse_expr; _ = Stream.empty >] -> e
val parse_string : string -> expr = <fun>
```

パージングの例

```
let e = parse_string "5+x*(4+x)";;
val e : expr = Plus (Num 5, Mult (Var "x", Plus (Num 4, Var "x")))

eval (subst ["x",3] e);;
- : expr = Num 26
```


数式を処理する ： まとめ

この例では次の機能を利用した

□数式の内部表現はヴァリアント(直和型を利用)

□再帰関数と汎関数の組み合わせで計算

□プリティプリンタを定義し、インストールした

□パージングにはstreamパーザを利用

□そこも再帰関数と汎関数を組み合わせて定義

型システム探検 ： 多相ヴァリアント

多相ヴァリアント ： 基礎

通常のヴァリアント(直和)と違って、型定義なしに使える

```
let a = ‘Apple
val a : [> ‘Apple ]
let b = ‘Orange "spain"
val b : [> ‘Orange of string ]
```

型は含まれる値を表わす

```
let l = [a; b]
val l : [> ‘Apple | ‘Orange of string ] list
```

関数の型は扱える場合を表わす

```
let show1 = function
    ‘Apple -> "apple"
  | ‘Orange s -> s ^ " orange"
val show1 : [< ‘Apple | ‘Orange of string ] -> string
```

多相ヴァリアント ： ディスパッチ

asパターンを使って、ヴァリアントがディスパッチできる

```
type round = [ ‘Apple | ‘Orange of string]
function #round as x -> x ;;
- : [< round ] -> [> round ] = <fun>
let show2 = function
    #round as x -> show1 x
  | ‘Pear -> "pear"
val show2 : [< ‘Apple | ‘Pear | ‘Orange of string ] -> string
List.map show2 [‘Pear; ‘Apple; ‘Orange "navel"] ;;
- : string list = ["pear"; "apple"; "navel orange"]
```

通常のヴァリアントが直和を提供しているのに対し、
多相ヴァリアントは場合の和集合を提供している

言語の拡張問題 ： モジュラーな方法

多相ヴァリアントを使えば、コードを書き変えずに言語を拡張できる

まずは整数だけの言語から

```
type num = [ ‘Num of int]
let eval_num (‘Num n : num) = n
val eval_num : num -> int = <fun>
```

さらなる拡張を可能にするために、「開かれた再帰」を使う

```
type ’a pexpr = [ num | ‘Plus of ’a * ’a]
let eval_pexpr eval_rec (e : ’a pexpr) =
  match e with
    #num as x -> eval_num x
  | ‘Plus (e1, e2) -> eval_rec e1 + eval_rec e2
val eval_pexpr : (’a -> int) -> ’a pexpr -> int = <fun>
```

言語の拡張問題 ： 再帰を閉じる

通常の評価関数を作るために再帰を閉じなければならない

```
let rec eval1 e = eval_pexpr eval1 e
val eval1 : ('a pexpr as 'a) -> int = <fun>
```

例

```
eval1 (‘Plus (‘Num 1, ‘Plus(‘Num 2, ‘Num 3)));;
- : int = 6
```

積算だけの言語も

```
type 'a mexpr = [ num | ‘Mult of 'a * 'a]
let eval_mexpr eval_rec (e : 'a mexpr) =
  match e with
    #num as x -> eval_num x
  | ‘Mult (e1, e2) -> eval_rec e1 * eval_rec e2
val eval_mexpr : ('a -> int) -> 'a mexpr -> int = <fun>
```


言語の拡張問題 ： 二重継承

ディスパッチを使って簡単に二重継承ができる

```
type 'a expr = [ 'a pexpr | 'a mexpr]
let eval_expr eval_rec (e : 'a expr) =
  match e with
    #pexpr as x -> eval_pexpr eval_rec x
  | #mexpr as x -> eval_mexpr eval_rec x
val eval_expr : ('a -> int) -> 'a expr -> int = <fun>
```

例

```
let rec eval e = eval_expr eval e
val eval : ('a expr as 'a) -> int = <fun>
eval (‘Plus (‘Num 3, ‘Mult(‘Num 5, ‘Num 2)));;
- : int = 13
```

多相ヴァリアント ： まとめ

多相ヴァリアントを使うと

□同じタグは複数の型で使える

□型定義自体は避けられる

□通常のヴァリアントより詳細な型情報

□開かれた再帰と組み合わせると継承が可能

言語拡張問題は現存の型付きオブジェクト指向言語では解けない

抽象型・オブジェクト・ファンクター

抽象型 ： signature

Camlのモジュールを利用して、抽象データ型が定義できる

```
module type PERSON = sig
  type t
  val create : name:string -> age:int -> t
  val birthday : t -> unit
  val name : t -> string
  val age : t -> int
end
```

上記のsignatureでは、t が抽象型になっている

createで作ることができるが、読み出しと変更は残りの3関数からしかできない

□name: はラベルと言い、分かりやすさのために使う

抽象型 ： 実装

signatureに対する実装を定義する

```
module Person : PERSON = struct
  type t = { name: string; mutable age: int }
  let create ~name ~age =
    assert (age>=0); {name=name; age=age}
  let birthday p = p.age <- p.age + 1
  let name p = p.name
  let age p = p.age
end
```

実際の t はレコード型だが、制約「: PERSON」によって外から抽象型として見える

中でageを変更可能だが、外からはbirthdayからしかできない

□name の前の ~ は引数をラベル付きにする

オブジェクト： クラス

OCaml はクラスベースのオブジェクトシステムを含む

```
class person ~name ~age = object
  val mutable age = assert (age>=0); age
  method birthday = age <- age+1
  method age = age
  method name : string = name
end ;;
class person : name:string -> age:int ->
  object
    val mutable age : int
    method age : int
    method birthday : unit
    method name : string
  end
```

この定義で抽象型と同じようにageが隠蔽される

オブジェクト： 継承

抽象型と違い、クラスは継承ができる

```
class person_feb29 ~name ~age : person =
  object
    inherit person ~name ~age
    method birthday = age <- age + 4
  end ;;
class person_feb29 : name:string -> age:int -> person
```

例

```
let family =
  [new person "kazuo" 24; new person_feb29 "kazuko" 24];;
val family : person list = [<obj>; <obj>]
List.map (fun x -> x#birthday; x#age) family;;
- : int list = [25; 28]
```

和夫と和子はどちらもperson型になる ： クラスと型は異なる

ファンクター

クラスには継承のようがあれば、モジュールには関手がある

標準ライブラリのSet.Makeは引数にOrderedTypeを取る

```
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end
module IntOrder = struct
  type t = int
  let compare x y = x-y
end
module S = Set.Make(IntOrder) ;;
let set = S.add 1 (S.add 2 (S.add 1 S.empty)) ;;
val set : S.t = <abstr>
Set.elements set ;;
- : S.elt list = [1; 2]
```


まとめ

本チュートリアルではObjective Camlの様々な機能を紹介した

関数型言語に共通なものがあれば、特徴的なものもある

豊富過ぎる機能の中でどれを選べばいいかと悩むかも知れない

そのときはまず簡単な方を選ぶべきでしょう
多相ヴァリアント以降に紹介した機能は型システムの
深い理解を必要とする場合がある

簡単なものの中に、やはり汎関数と多相性を多用するといい
より汎用的な定義をすることで、問題の解析が進み、

Objective Caml に関する情報源

開発元 INRIA でのサイト

`http://caml.inria.fr/`

日本語での OCaml 情報

`http://www.ocaml.jp/`

京都大学でのサイト (ftpミラーとライブラリ・ツールの配布)

`http://wwwfun.kurims.kyoto-u.ac.jp/soft/`