

誰も知らない OCaml

僕も知らない

KMCID: taisei

5/9/2022

やること

- ここ数年 OCaml ばかり書いてる
- ここ数年で言語の更新や新機能の追加が加速した気がする
- OCaml 書いててこんなんでできるんだ～ってなったを紹介



OCaml

- OCaml¹ : Objective Caml
 - Caml : Categorical Abstruct Machine Language²らしい
- ML³ 方言 関数型プログラミング言語の一つに分類されがち
- 計算機コースの人は必修

¹知らない人は [2], [3] 辺りを参照

²Meta Language だと思っていたけれど違った

³Machine Learning ではない

こんなの

適当な例 1

```
type point = { x: int; y: int } (* x, y を持つ record の宣言 *)
(* 2つの point を持つ Square と, point と int を持つ Circle の variant *)
type shape = Square of point * point | Circle of point * int
let pi = 4.0 *. atan 1.0 (* 定数定義 *)
let area sp = match sp with (* 関数定義. sp が Shape か Circle かで分岐 *)
  | Square ({x=px; y=py}, q) -> float_of_int ((q.y-py) * (q.x-px))
  | Circle (_, r) -> let r = float_of_int r in r *. r *. pi
```

Circle と Square の 2 種類の shape を定義し, shape の面積を求める関数 area を定義

こんなの

適当な例 2

```
module List = struct
  type 'a list = Nil | Cons of 'a * 'a list (* 多相型 *)
  let rec map : ('a -> 'b) -> 'a list -> 'b list = (* 再帰関数定義 *)
    fun f -> function
      | Nil -> Nil
      | Cons (h, t) -> Cons (f h, map f t)
end
let double_list : int list -> int list =
  List.map (fun x -> x * 2) (* 部分適用 *)
```

⁴関数 map を持つモジュール List を定義し, List で定義したリストの各要素を 2 倍にする関数 double_list を定義

⁴function は fun x -> match x with の構文糖衣

もくじ

- Extensible Variant Type
- Polymorphic Variant
- Monadic bind
- PPX
- Type Construct over Signatures & Substituting inside a Signature

Extensible Variant Type

- コンストラクタを後で追加できる (???)

Extensible Variant Type

```
type t = .. (* 省略していません!! *)
type t += Str of string
type t +=
  | Int of int
  | Float of float
```

Extensible Variant Type

- よく Result モナドの Error.t 型の方で使われる

Extensible Variant Type

```
module Error = struct
  type t = ..
  type t += Fatal of string
end
type 'a result = Ok of 'a | Error of Error.t

module Arith = struct
  type Error.t += Overflow | ZeroDivision
  let div a b = if b = 0 then (Error ZeroDivision) else Ok (a/b)
end
```


Polymorphic Variant

- どの型にも属さないコンストラクタ 通称タグ
 - 先頭が ‘(バッククォート) で始まる
- 型宣言なしで使えるが、型注釈してもよい
 - pattern-match で便利になるかも

Polymorphic Variant

```
type myvariant = ['Tag1 of int | 'Tag2 of bool]
let f = function
  | #myvariant -> "myvariant" (* myvariant 型に属する tag 全てにマッチ *)
  | 'Tag3 -> "Tag3"
=====
val f : [< 'Tag1 of int | 'Tag2 of bool | 'Tag3 ] -> string
```

Polymorphic Variant

- [$<..$]: サブセットを表す型
- [$>..$]: スーパーセットを表す型

Polymorphic Variant

```
type t = ['A | 'B | 'C]
let f = function
  | #t as t -> t
  | 'D -> 'E
let g = function
  | #t as t -> t
  | x -> x
=====
val f : [< 'A | 'B | 'C | 'D ] -> [> 'A | 'B | 'C | 'E ] = <fun>
val g : ([> t ] as 'a) -> 'a = <fun>
```

Monadic Bind

- これまで OCaml で Monad を書くと bind 演算子祭りになって辛かった
 - 関数がネストして見辛い
 - 型が合わないエラー等の場所が分かりづらくなっていた

Monadic Bind

```
let bind x f = match x with Some x -> f x | None -> None
let both x y = match (x, y) with Some x, Some y -> Some (x, y) | _ -> None
let return x = Some x
let (>>=) = bind

both (return 1) (return 2) >>= (fun (x, y) ->
return 3 >>= (fun z ->
return (x + y + z)))
```

Monadic Bind

- `let(symbol+)` と `and(symbol+)` を演算子として定義できる
- `let` 文っぽく書いて特殊な感じで展開してくれる

Monadic Bind

```
let (let*) = bind
let (and*) = both
```

```
let* x = return 1 and* y = return 2 in
let* z = return 3 in
return (x + y + z)
```

: (* 展開されると *)

```
(let*) ((and*) (return 1) (return 2)) (fun (x, y) ->
(let*) (return 3) (fun z ->
return (x + y + z)))
```

PPX (PreProcessor eXtension)[4]

- literal, attribute, extension point を埋め込める
- コードの AST から AST へ変換する関数を前処理で実行できる
 - 型付け前なので型情報は無い
- マクロ, リテラル, コード生成, monadic なキーワード定義などに

ppx

```
123456z ==> Z.of_int 123456 (* literal. 多倍長整数へ変換 *)
```

```
type point3d = float * float * float  
[@@deriving show] (* attribute. show 関数を自動定義 *)
```

```
match%lwt x with ... ==> [%lwt match x with ...] (* exptention point *)  
==> Lwt.bind x (function ...) (* モナドで持ち上がった x を展開 *)
```

Type Construct Over Signatures⁵

- module 名 with type $t = u$ とすると, 型同士の制約を追記できる

Type Construct over signatures

```
module type Monad = sig
  type 'a t (* 抽象型 *)
  val return : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t
end
module type OptionMonad : Monad with type 'a t = 'a option
=====
module type OptionMonad = sig
  type 'a t = 'a option
  val return : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t
end
```

⁵それっぽい呼び方見つからなかったので適当につけた

Substituting inside a Signature

- module 名 with type $t := u$ とすると, t を u で書き換える (??) ⁶

Substituting inside a signature

```
module type Monad = sig
  type 'a t (* 抽象型 *)
  val return : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t
end
module type OptionMonad : Monad with type 'a t := 'a option
=====
module type OptionMonad = sig
  val return : 'a -> 'a option
  val bind : 'a option -> ('a -> 'b option) -> 'b option
end
```

⁶最近 module type を module type で代入できるようになった (??)

紹介しなかったトピック

- OCaml の O の部分
 - 未だに書いたこと無い...
- GADT [5]
 - Explicit polymorphic Annotations + Locally Abstract Type の謎記法
- Pretty Printer[6]

最後に

Quiz

```
type _ expr =  
| Int : int -> int expr  
| Add : (int -> int -> int) expr  
| App : ('a -> 'b) expr * 'a expr -> 'b expr  
  
let rec eval [% ??? ] = function  
| Int n -> n  
| Add -> (+)  
| App (f, x) -> eval f (eval x)
```

`[% ???]` に入れてコンパイルが通るのは次のうちどれでしょう？

最後に

- 1 (type t) : t expr -> t
- 2 : type t. t expr -> t
- 3 : t. t expr -> t
- 4 : 't. 't expr -> 't
- 5 実は何も書かなくてもいける

REFERENCES I

- [1] <https://ocaml.org/manual/index.html>
- [2] https://hackmd.io/@aigarashi/r1az0w0HP/%2FBsXr0ToESu-Q30kiM_PAmg
- [3] <https://github.com/kuis-isle3sw/IoPLMaterials/blob/master/textbook/slides/ocaml.pdf>
- [4] <https://dailambda.jp/slides/2021-04-09-ppx.html#/what-is-ppx-1>

REFERENCES II

- [5] <https://www.math.nagoya-u.ac.jp/~garrigue/papers/ml2011-show.pdf>
- [6] <https://qnighy.hatenablog.com/entry/2017/01/26/215948>