Objective Caml 3.12 のモジュール機能

Jacques Garrigue (名古屋大学)

http://www.math.nagoya-u.ac.jp/~garrigue/

with Alain Frisch (Lexifi), OCaml developper team (INRIA)

Objective Caml におけるモジュールの役割

- プログラムおよび名前空間の構造化
- インターフェスによる仕様記述・部品化
- 抽象型・プライベート型による隠蔽
- ファンクターによる抽象化

他の言語機能と密接につながり、プログラミング全体に影響する

モジュールがコア言語より優れている所

- **多相的な値の受け渡し**

  レコードやオブジェクトでも可能だったが、モジュールでは型構成子に対する多相生もある。

- **正確な多相型要求** （**例：**`val id : 'a -> 'a`）

  3.12**ではコア言語でも可能になった**

- **全ての種類の定義ができる**

  コア言語ではトップレベルでしか定義できないものや相互再帰にできないもの（クラスと通常の方など）がある。

- **存在型（抽象型）の定義**

  コア言語では全称型によってエンコードできるが、型構成子に対応できない。

Objective Caml 3.12での新機能

- **局所的**open (local open)
- **局所的抽象型** (local abstract type)
- **明示的な多相型注記** (explicit polymorphism annotations)
- **第一級モジュール** (first-class modules)
- **シグネチャの取得** (signature of a module)
- **シグネチャの破壊的代入** (destructive substitution)

第一級モジュール

- **モジュールを通常の値のように受け渡しできる**

  モジュール言語でしかできなかった操作がコア言語で利用できるようになる。

- **完全に型付けされている**

  安全に利用できる。

- **理論的には新しくない**

  10**年前に**Claudio Russo**が**Moscow ML**で実装した。**
  ただし、ファンクターとの干渉のせいで安全ではなかった。

例1　多相関数の受け渡し

```
# module type ID = sig val id : 'a -> 'a end ;;
module type ID = sig val id : 'a -> 'a end
# let f id =
    let module Id = (val id : ID) in
    (Id.id 1, Id.id true) ;;
val f : (module ID) -> int * bool = <fun>
# f (module struct let id x = print_endline "Id!"; x end : ID);;
Id!
Id!
- : int * bool = (1, true)
```

注：オブジェクトやレコードでは既にできていたが、モジュールの方が自然。

例2　実行時に実装を選ぶ

```
module type DEVICE = sig ... end
let devices : (string, (module DEVICE)) Hashtbl.t
            = Hashtbl.create 17

module PDF = struct ... end
let () = Hashtbl.add devices "PDF" (module PDF: DEVICE)
...

module Device =
  (val (try Hashtbl.find devices Sys.argv.(1)
        with Not_found -> prerr_endline "Unknown device"; exit 2)
       : DEVICE)
```

例3　型安全なプラグイン

```
module type PLUGIN = sig
  type t                                    (* プラグインごとに型が異なる *)
  val state : t
  val start : t -> unit
  val stop : unit -> t
end ;;
let plugins = ref ([] : (string * (module PLUGIN)) list) ;;

let new_instance name =
  let module P = (val List.assoc name !plugins : PLUGIN) in
  object                              (* レコードでもいいけど，クラスはだめ *)
    val mutable state = P.state
    method start = P.start state
    method stop  = state <- P.stop ()
  end ;;
val new_instance : string -> < start : unit; stop : unit > = <fun>
```

例4　クラスの動的生成・継承

```
module type Compute = sig
  class compute : object method x : int end
end
module Default = struct          (* クラスを第一級モジュールに入れる *)
  class compute = object method x = 0 end
end
let compute = ref (module Default : Compute)

let incr () =                    (* クラスを動的継承によって変更する *)
  let module M = struct
    module C = (val !compute : Compute)
    class compute = object
      inherit C.compute as super
      method x = super#x + 1
    end
  end in compute := (module M : Compute)
```

例5　型の実行時表現とGADTs

```
module TypEq : sig
  type ('a, 'b) t                                          (* 型等式 *)
  val apply : ('a, 'b) t -> 'a -> 'b
  val refl : ('a, 'a) t        val sym : ...
end = ...
module rec Typ : sig
  module type PAIR =
    type t and t1 and t2
    val eq: (t, t1 * t2) TypEq.t
    val t1: t1 Typ.typ         val t2: t2 Typ.typ
  end
  type 'a typ =
    | Int of ('a, int) TypEq.t
    | String of ('a, string) TypEq.t
    | Pair of (module PAIR with type t = 'a)    (* ∃t1 ∃t2 ... *)
end = Typ
```

例5　型の実行時表現とGADTs（続）

```
...                                  (* to_stringは多相再起を使うので型を書く *)
let rec to_string : 'a. 'a typ -> 'a -> string =
  fun (type s) t x ->                                  (* sは局所的抽象型 *)
    match t with
    | Int eq -> string_of_int (TypEq.apply eq x)
    | String eq -> Printf.sprintf "%S" (TypEq.apply eq x)
    | Pair p ->
        let module P = (val p : PAIR with type t = s) in
        let (x1, x2) = TypEq.apply P.eq x in
        Printf.sprintf "(%s,%s)"
          (to_string P.t1 x1) (to_string P.t2 x2)
```

詳しくはマニュアルおよびKyseliov等のML Workshop 2010の論文

第一級モジュールの制限

- **ファンクターの中では第一級モジュールを開くことはできない**

  OCamlのファンクターは applicative (同じ引数に二回適用すると同じ型を生成する) なので、それを許すと安全でないプログラムが書ける。

```
module type S = sig type t val x : t end
let r = ref (module struct type t = int let x = 0 end : S)
module F(X:sig end) = (val !r : S)
module A = struct end
module M = F(A) ;;
module M : sig type t = F(A).t val x : t end
r :=  (module struct type t = float let x = 0. end : S) ;;
module N = F(A) ;;
module N : sig type t = F(A).t val x : t end          (* tが変わらない *)
```

  しかし関数としてgenerativeなファンクターが書ける (共有が弱い)

- **型とpack・unpackを全て書かなければならない** 次ページ

第一級モジュールの型推論　(implicit-unpack)

多相メソッドと同じ機構を使って、多くの型注記が省ける。

```
module type ID = sig val id : 'a -> 'a end ;;
let f (module Id:ID) = (Id.id 1, Id.id true);; (* unpackパターン *)
f (module struct let id x = x end);;          (* 型注記なしでpack *)

let rec to_string : 'a. 'a typ -> 'a -> string =
  fun (type s) (t : s typ) x ->                    (* 型を伝搬する *)
    match t with
    | Int eq -> string_of_int (TypEq.apply eq x)
    | String eq -> Printf.sprintf "%S" (TypEq.apply eq x)
    | Pair (module P) ->                        (* 型注記なしでunpack *)
        let (x1, x2) = TypEq.apply P.eq x in
        Printf.sprintf "(%s,%s)"
          (to_string P.t1 x1) (to_string P.t2 x2) ;;
```

Objective Camlのシグネチャ言語

- モジュールを多く使うと、その型であるシグネチャを効率よく書かなければならない。
- 今までのOCamlのシグネチャ言語は表現力が乏しかった
  - ライブラリのモジュールのシグネチャが取得できなかった
  - シグネチャを組み合わせて新しいシグネチャをうまく作れなかった

モジュールとシグネチャの相互変換

- **シグネチャをモジュールに変換する**

  GADT で見たように、型宣言だけなら `module rec` を使えばいい。

```
module rec M : S = M
```

- **モジュールのシグネチャを取得する**

  新たに追加された `module type of` を使う。

```
module MyList : sig
  include module type of List
  val remove : 'a -> 'a list -> 'a list
end = struct
  include List
  let rec remove a = ...
end
```

シグネチャの合成

同じ型に対する2つのシグネチャが与えられたときにその合成を作りたい

```
module type Printable =
  sig type t  val print : t -> unit end
module type Comparable =
  sig type t  val compare : t -> t -> int end

module type PrintableComparable = sig                    (* 欲しい結果 *)
  type t
  val print : t -> unit
  val compare : t -> t -> int
end
```

includeによる合成の問題点

すなおにincludeで合成を取ろうとすると、同じ型が何度も定義されていると言って怒られる。

```
module type PrintableComparable = sig
  include Printable
  include Comparable with type t = t
end
Error: Multiple definition of the type name t.
       Names must be unique in a given structure or signature.
```

ファンクターによる解決方法

ファンクターで問題が解決できるものの、シグネチャと雰囲気が大分違う。

```
module type T = sig type t end
module PrintableF(X:T) = struct
  module type S = sig val print : t -> unit end
end
module ComparableF(X:T) = struct
  module type S = sig val comparable : t -> t -> int end
end
module PrintableComparableF(X:T) = struct
  module type S =
    sig include PrintableF(X).S include ComparableF(X).S end
end
```

シグネチャの破壊的代入

with **制約の構文を拡張し、型やモジュールパスを代入した後に定義をシグネチャから削除するという破壊的代入を追加した。**

```
module type ComparableInt = Comparable with type t := int ;;
module type ComparableInt =
  sig val compare : int -> int -> int end
```

この構文は `ComparableF(Int).S` **と同等の効果があるが、元々** `Comparable` **から** `ComparableF` **を生成できなかった。**

```
  module ComparableF(X:T) = struct
    module type S = Comparable with type t := X.t
  end
```

破壊的代入の応用

- シグネチャの合成：共通の型を破壊的に代入する

```
module type PrintableComparable = sig
  include Printable
  include Comparable with type t := t
end
```

- 型やモジュールの削除：定義を繰り返せばいい

```
module type PrintableInt' =
  (Printable with type t = int) with type t := int
```

- 名前の付け替え

```
module type Printable' = sig
  type printable
  include Printable with type t := printable
end
```

破壊的代入の制限

破壊的代入の実装方法により、制限が生じる。

- 代入をかけられるのは一番上のレベルの型とモジュールのみ。
- パス（定義されているもの）しか代入できない。
- 型のパラメターの数・順番は変えられない。
- 削除によって、構文的にスコープが有効でないシグネチャになったりする。ただし、意味に異常はない。

まとめ

- Objective Caml 3.12**ではモジュールとコア言語の関係とシグネチャが大きく強化された。**
- **これによって今まで不可能または冗長だった様々なパターンが利用可能になった。**
  - **プラグイン**
  - GADTs
  - **シグネチャの合成**
- **これによって言語全体が使いやすくなったはず**