Objective Camlがなぜバグを書かせないのか

Jacques Garrigue
名古屋大学 多元数理科学研究科
http://www.math.nagoya-u.ac.jp/~garrigue/papers/

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．今回の内容のほとんどは Standard ML**や**Haskell**にも応用できる．**

高速なコンパイラ，様々な言語拡張，豊富なツールとライブラリ

OCamlはどこで使われている

ウィキペディアやOCaml-Humpなどに登録されているOCamlで書かれた大きなソフトウェアを分類すると

- **定理証明**: Coq, HOL-Light, Focalize, Why
- **コンパイラ等**: OCaml, MTASC, CIL, CCured, XDuce, CDuce
- **ソフトウェア検証**: Frama-C, SLAM
- **ネットワーク**: MLDonkey, Unison, Marionnet, spamoracle, SKS
- **数値計算**: FFTW(**フーリエ変換**), FaCiLe(**最適化**)
- **文書処理**: ADvi, ANT(Tex), Hevea(Tex→Html), Mana(**かな変換**)
- **ライブラリ等**: FreeBSD**では**74**のコンパイル済み**package

それ以外に社内開発 (システムソフトウェアや金融業務など) や小規模なプロジェクトに多く使われている.

この発表の内容

- ▷ **型推論と多相性**
- **様々な抽象化**
- **ヴァリアント (直和型) の基本**
- **応用例：数式の処理とパージング**
- **さらなる安全性：プログラムの証明**

型推論とは

OCamlは強い型付き言語である

全ての関数や変数に型があり，実行時に型エラーがない

しかし，明示的に型を書く必要がない

型推論によって，型が自動的に推論される

```
$ ocaml                                          ← ocamlを起動
        Objective Caml version 3.11.1

# let average x y = (x + y) / 2 ;;               ← 関数定義を入力
val average : int -> int -> int = <fun>          ← 型が推論される
```

型エラー発生！

```
# average 10 20;;
- : int = 15
# average 10 "20";;
Error: This expression has type string but an
  expression was expected of type int
# print_string (average 10 20);;
Error: This expression has type int but an
  expression was expected of type string
# average 10 (int_of_string "20");;
- : int = 15
```

多相性とは

同じ関数が**複数の型に対して使える**場合がある

```
# let pair x y = (x, y) ;;          ← 対を作る関数
val pair : 'a -> 'b -> 'a * 'b = <fun>
```

- 'a, 'b は型変数で，型の解釈は $\forall\alpha\forall\beta,\ \alpha \to \beta \to \alpha \times \beta$
- 常に**最も一般性のある型**が推論される
- 注意：ここでいう多相性はオブジェクト指向言語の**部分型による多相性と似て異なる**ものである．
  特に，値自体に対して操作を行うと多相性が生じない．

データの多相性

対だけでなく，様々なデータ構造が多相性を持つ．例えば，リストや配列．

```
# let me = pair "Jacques" 38;;
val me : string * int = ("Jacques", 38)
# let append l1 l2 = l1 @ l2;;
val append : 'a list -> 'a list -> 'a list = <fun>
# let l = append [1;2] [3;4];;
val l : int list = [1; 2; 3; 4]
# let l' = append [1;2] ["Jacques"];;
Error: This expression has type string but an expression
  was expected of type int
# let first arr = arr.(0);;
val first : 'a array -> 'a = <fun>
```

汎関数と多相性

汎関数の存在が多相性の利用を飛躍的に増やす．

```
# let foldl = List.fold_left;;                    ← 演算子をリストに挟む汎関数
    foldl (⊕) a [b_1; ...; b_n] = a ⊕ b1 ⊕ ... ⊕ bn
val foldl : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

# let sum_int l = foldl (fun a b -> a + b) 0 l;;
val sum_int : int list -> int = <fun>

# let sum_float = foldl (+.) 0. ;;                ← 関数引数の省略
val sum_float : float list -> float = <fun>

# sum_float [1.5; 2.0; 3.1] ;;
- : float = 6.6
```

汎関数の応用と抽象化

```
# let foldl2 = List.fold_left2;;                    ← 2つのリストを使うfoldl
val foldl2 :
  ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a = <fun>

# let scalar v1 v2 = foldl2 (fun r x y -> r +. x *. y) 0. v1 v2;;
val scalar : float list -> float list -> float = <fun>

# let scalar' ~plus ~mult ~zero v1 v2 =                ← 演算子を抽象化
  foldl2 (fun r x y -> plus r (mult x y)) zero v1 v2;;
val scalar' : plus:('a -> 'b -> 'a) -> mult:('c -> 'd -> 'b) ->
              zero:'a -> 'c list -> 'd list -> 'a = <fun>

# let scalar_float = scalar' ~plus:(+.) ~mult:( *.) ~zero:0. ;;
val scalar_float : float list -> float list -> float = <fun>
```

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てくれる．

```
plus:('a -> 'b -> 'a) -> mult:('c -> 'd -> 'b) ->
zero:'a -> 'c list -> 'd list -> 'a
```

- この関数は`plus`, `mult` と `zero` という名前の引数と，2つのリストをもらう．
- 2つのリストの各要素は `mult` によって組み合わされ，その結果は `plus` によって `zero` と組み合わされる．

さらに，型が期待していたものと異なったとき，バグの発見に役立つ．

- 一般性が足りないとき，どこかで値が混同され，正しくない処理が行われている．
- 引数の型変数が他で現れないとき，その引数が使われていない．型エラーにはならないので，要注意．

型推論 : 汎関数のすすめ

- 計算の途中の状態が隠蔽される
  → プログラムを状態のないものとして理解できる.
- 配列の外にアクセスするようなことは起きない.
- 抽象化によって共通のコードが共有できるので，バグが修正しやすい.
- 抽象化で推論された型を見ると，バグが発見できる

```
# let scalar' ~plus ~mult ~zero v1 v2 =
  foldl2 (fun r x y -> mult x y) 0. v1 v2;;
val scalar' :
  plus:'a -> mult:('b -> 'c -> float) ->
  zero:'d -> 'b list -> 'c list -> float
```

型を見れば，`plus` や `zero` が使われていないことが分かる.

オブジェクトによる抽象化

Objective Caml**は名前の通り，オブジェクトを持っている，型推論もできる．**

```
let scalar'' ~ops v1 v2 =
  foldl2 (fun r x y -> ops#plus r (ops#mult x y)) ops#zero v1 v2
val scalar'' :
  ops:<mult:'a -> 'b -> 'c; plus:'d -> 'c -> 'd; zero:'d; .. > ->
  'a list -> 'b list -> 'd = <fun>
class float_ops = object
  method plus x y = x +. y
  method mult x y = x *. y
  method zero = 0.
end
let scalar_float = scalar'' ~ops:(new float_ops)
val scalar_float : float list -> float list -> float = <fun>
```

モジュールによる抽象化

モジュールによって型を引数に取る複雑な構造が表現できる

```
module type PseudoRing = sig          ← モジュールの引数は推論されない
  type t                              ← 環の元の型
  val plus : t -> t -> t              ← 各演算子の型
  val neg  : t -> t
  val zero : t
  val mult : t -> t -> t
end

module Vector (R : PseudoRing) = struct          ← 環を引数にとる
  type t = R.t list                              ← 環の上のベクトル
  let plus : t -> t -> t = List.map2 R.plus      ← 型は省略してもいい
  let scalar = foldl2 (fun r x y -> R.plus r (R.mult x y)) R.zero
end
```

型推論と抽象化のまとめ

型推論は型安全なプログラムの開発を楽にする

- ◇ 変数ごとに型を書く必要がない
- ◇ 小さなコードでも簡単に共有できる
- ◇ 多相性によって利用範囲が広がる

多相型は情報の宝庫

- ◇ データの流れが直接的にに見える
- ◇ 汎関数によって多相型が増やせる

様々な抽象化の方法がある

- ◇ 汎関数は手軽
- ◇ クラスを使うと拡張しやすい
- ◇ モジュールは堅牢なデザインにつながる

この発表の内容

型推論と多相性

様々な抽象化

▷ **ヴァリアント (直和型) の基本**

応用例：数式の処理とパージング

さらなる安全性：プログラムの証明

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**
- ◇ **プログラムを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**

この発表の内容

- 型推論と多相性
- 様々な抽象化
- ヴァリアント (直和型) の基本
- ▷ 応用例：数式の処理とパージング
- さらなる安全性：プログラムの証明

数式を処理する ： 型の定義

簡単な数式を考える

$$\textbf{式} ::= \textbf{整数} \mid \textbf{変数} \mid \textbf{式} + \textbf{式} \mid \textbf{式} \times \textbf{式} \mid (\textbf{式})$$

例

$$5 \times 2 + 3 \qquad\qquad (3 + y) \times 12$$

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` が再利用のこつ

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 =      ← ?prio はデフォルト引数
  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 ->          ↓ (printf <fmt>) は prio のため
      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
```

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;;                    ← prio を消す
val print_expr' : Format.formatter -> expr -> unit
#install_printer print_expr';;                          ← プリンターを登録
e;;
- : expr = x + 3 * 2
```

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"dynlink.cma";;
#load"camlp4o.cma";;
        Camlp4 Parsing version 3.11.1

open Genlex;;
let lexer = Genlex.make_lexer ["+";"*";"(";")"] ;;
val lexer : char Stream.t -> Genlex.token Stream.t = <fun>
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
```

数式を処理する ： stream パーザ

```
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 =                          ← 主関数
  left_assoc parse_mult "+" (fun e1 e2 -> Plus(e1,e2)) s

let parse_string s = parse_expr (lexer (Stream.of_string s));;
val parse_string : string -> expr = <fun>
let e = parse_string "5+x*(4+x)";;
val e : expr = Plus (Num 5, Mult (Var "x", Plus (Num 4, Var "x")))
```

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**

この発表の内容

型推論と多相性

様々な抽象化

ヴァリアント (直和型) の基本

応用例：数式の処理とパージング

▷ **さらなる安全性：プログラムの証明**

プログラムの証明

当然ながら，奇麗な書き方をしても，型システムだけでプログラムの正しさは保証できない．

更なる安全性のために，次のどちらかが必要になる

- **網羅的なテストによる安全性の獲得**
  現実的な方法だが，入力の範囲が制限できない場合，どれだけ多くのテストケースを用意すればよいかは判断しにくい．
- **証明されたプログラムの生成**
  多大な労力が必要だが，絶対的な安全性が得られる．

Coqでのプログラム開発

Coq は OCaml で開発された**定理証明支援系**

- 使用している論理は**型理論**に基づいている．
- Calculus of Inductive Constructionsは高階述語論理をも包含する**非常に強い直観主義論理**．
- **同じ言語**の中でプログラム，命題および証明を書く．
- プログラムが完成すれば，証明された OCaml **のコードとして抽出**できる．

この方法でいくつかのプログラムが証明されている．

- OCamlに標準添付の**有限集合ライブラリ**と同機能のコード
- ANSI-C **のコンパイラー** (CompCert)

OCamlを証明する？

OCaml 自体は複雑なプログラムになっている．

- 特に型推論のために自明でないアルゴリズムを多く使っている．
- 結果的に型推論の健全性と完全性が保証しにくい．

OCaml の型検証器を Coq で書けば確実な保証が得られる．

- 私は２年前からそれをやっている．
- 効率性の観点から，現在の実装に代わるものは作れないが，参考実装を目指す．
- 現状ではオブジェクトとヴァリアントを含んだ型検証器の完全な検証に成功している．
- OCaml にはそれ以外に多くの機能があるので，完成はまだ遠い

まとめ

Objective Caml **は強力なプログラミング言語で，正しく使えば型システムが多くのバグを検出してくれる．**

今回紹介機能の中で，関数型言語に共通なものがあれば，特徴的なものもある．

ほとんどのプログラムにおいて，ヴァリアント（再帰データ型）と汎関数を利用すれば，簡潔かつ安全な開発ができる．

再利用をさらに増やしたい場合，オブジェクトや多相ヴァリアントが役に立つが，大規模になるとはまりやすいので慎重に使った方がいい．

絶対的な信頼が必要なときには，Coq **とのインターフェースが魅力的．**

```
http://caml.inria.fr/
http://coq.inria.fr/
http://www.math.nagoya-u.ac.jp/~garrigue/home-j.html
```