Objective Caml NG集
型推論とオブジェクト指向の複雑な関係

Jacques Garrigue
名古屋大学 多元数理科学研究科

Synopsis

- λ計算と型推論
- 多相性
- オブジェクトの型付け
- バグ１ 引数のない仇名型
- バグ２ 計算時間の爆発
- バグ３ 型引数の共変性
- バグ４ 第一級多相型と再帰型

λ計算と関数型言語

λ計算自身の定義は簡単だが，一般的なプログラムを書くのに十分である

e ::=	x	**変数**
|	c	**定数**
|	$e\ e$	**関数適用**
|	fun $x \to e$	**関数抽象**
|	let $x = e$ in e	**定義**

意味は書き換えによって定義できる

$$(\text{fun } x \to e_1)\ e_2 \longrightarrow e_1[x := e_2]$$
$$\text{let } x = e_2 \text{ in } e_1 \longrightarrow e_1[x := e_2]$$

関数型言語(LISP,ML,Haskellなど)はλ計算を核としている

型付λ計算と型推論

型は三種類 型変数・基本型・関数型

$$\tau ::= \alpha \mid b \mid \tau \to \tau$$

以下の規則によって証明ができるものが正しい型付λ式

Var
$\Gamma \vdash x : \Gamma(x)$

Const
$\Gamma \vdash c : T(c)$

App
$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

Abs
$$\frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \mathsf{fun}\ x \to e : \tau_1 \to \tau_2}$$

定理 (一般性) $\Gamma \vdash e : \tau$ が成り立つならば，$\Gamma \vdash e : \tau_0$ が存在し，任意の $\Gamma \vdash e : \tau'$ に関して $(\exists \theta : \mathcal{V} \to \mathcal{T})\ \tau' = \theta(\tau_0)$

型推論の例

$$\emptyset \vdash \mathsf{fun}\ f\ x\ y \rightarrow f\ y\ x : \alpha$$

型推論の例

$$\frac{\Gamma = f{:}\beta, x{:}\gamma, y{:}\delta \vdash f\ y\ x : \eta}{\emptyset \vdash \mathsf{fun}\ f\ x\ y \rightarrow f\ y\ x : \alpha = \beta \rightarrow \gamma \rightarrow \delta \rightarrow \eta}$$

型推論の例

$$\dfrac{\dfrac{\Gamma \vdash f\ y : \gamma \to \eta \qquad \Gamma \vdash x : \gamma}{\Gamma = f{:}\beta, x{:}\gamma, y{:}\delta \vdash f\ y\ x : \eta}}{\emptyset \vdash \mathsf{fun}\ f\ x\ y \to f\ y\ x : \alpha = \beta \to \gamma \to \delta \to \eta}$$

型推論の例

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash f : \beta = \delta \to \gamma \to \eta \quad \Gamma \vdash y : \delta}{\Gamma \vdash f\ y : \gamma \to \eta} \qquad \Gamma \vdash x : \gamma}{\Gamma = f{:}\beta, x{:}\gamma, y{:}\delta \vdash f\ y\ x : \eta}}{\emptyset \vdash \mathsf{fun}\ f\ x\ y \to f\ y\ x : \alpha = \beta \to \gamma \to \delta \to \eta}$$

推論された型　　$(\delta \to \gamma \to \eta) \to \gamma \to \delta \to \eta$

ML多相性（Hindley-Milner多相性）

多相型を定義する $\sigma ::= \tau \mid \forall \bar{\alpha}.\tau$

$$\text{Var}\quad \frac{\Gamma(x) = \forall \bar{\alpha}.\tau}{\Gamma \vdash x : \theta|_{\bar{\alpha}}(\tau)} \qquad \text{Let}\quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \forall (FTV(\tau_1) \setminus FTV(\Gamma)).\tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

これにより，letで定義された変数は複数の型で利用できるようになる

型推論の**一般性**は保たれる

定理(健全性) $\Gamma \vdash e : \tau$ が成り立つならば，任意の $e \xrightarrow{*} e'$ に関して，$\Gamma \vdash e' : \tau$.
かつ，e' が簡約できない場合，e' は関数適用およびlet式を含まない

型付関数型言語(ML,Haskellなど)はML多相性に基いている

その他の多相性(polymorphism)

多相性とは，同じ値が複数の異なる型として利用できることをいう

二つの次元で分類できる

- *parametric* polymorphism vs. *ad hoc* polymorphism
  型がプログラムの意味に影響を与えるかどうか.

- *instanciation* (型変数の代入) vs. *subsumption* (型の包含関係)
  値の本質的な型と利用するときの型の関係の種類

$$\forall\alpha.\tau \succ [\tau'/\alpha]\tau \qquad\qquad t_{sub} \leq t_{sup}$$

Objective Camlではどちらも前者しかないが,
instanciationによってsubsumptionの真似をしている

オブジェクトの導入

Objective Camlはクラスを持っているが，簡単のためにここではオブジェクトのみで説明する．

e	::=	$\ldots$	
	$\mid$	object (x) f $\ldots$ end	**オブジェクト**
	$\mid$	$e\#m$	**メソッド呼び出し**
	$\mid$	$\{< x = e; \ldots >\}$	**フィールド更新**
	$\mid$	$(e : \tau :> \tau')$	**サブタイピング**
f	::=	val $x = e$	**フィールド定義**
	$\mid$	method $m = e$	**メソッド定義**
τ	::=	$\ldots$	
	$\mid$	$< m : \tau; \ldots >$	**オブジェクト型**
	$\mid$	$< m : \tau; \ldots; .. >$	**開いたオブジェクト型**
	$\mid$	τ as α	**再帰型とエイリアス**

オブジェクトの型付け

Object

$$\frac{\Gamma;;\vdash f_i : \tau_i \quad \Gamma;[x_i : \tau_i]_1^n; x : \tau \vdash e_j : \tau'_j \quad \tau = < m_j : \tau'_j >_1^m}{\Gamma \vdash \text{object } (x) \text{ [val } x_i = f_i]_1^n \text{ [method } m_j = e_j]_1^m \text{ end} : \tau}$$

Method

$$\frac{\Gamma \vdash e : < \ldots; m : \tau; \ldots >}{\Gamma \vdash e\#m : \tau}$$

Update

$$\frac{\Gamma; F; x_s : \tau_s \vdash e : \tau \quad x : \tau \in F}{\Gamma; F; x_s : \tau_s \vdash \{< x = e >\} : \tau_s}$$

Subtype

$$\frac{\Gamma \vdash e : \theta(\tau) \quad \vdash \tau <: \tau'}{\Gamma \vdash (e : \tau :> \tau') : \theta(\tau')}$$

構造的サブタイピング

Width

$$\vdash < m_1 : \tau_1; \ldots; m_n : \tau_n > <: < m_1 : \tau_1; \ldots; m_k : \tau_k > \ (k < n)$$

Open

$$\vdash < m_i : \tau_i; .. >_1^n <: < m_i : \tau_i >_1^n$$

Hyp

$$S, \alpha <: \alpha' \vdash \alpha <: \alpha'$$

Depth

$$\frac{S \vdash \tau_i <: \tau_i' \qquad (1 \leq i \leq n)}{S \vdash < m_i : \tau_i >_1^n <: < m_i : \tau_i' >_1^n}$$

Rec

$$\frac{S, \alpha <: \alpha' \vdash \tau <: \tau'}{S \vdash \tau \text{ as } \alpha <: \tau' \text{ as } \alpha'}$$

オブジェクトの型付けの例

大堀流 kind 付き型と再帰型の組み合わせ

```
# let moi = object
    val age = 33
    method name = "Jacques"
    method age = age
    method birthday = {< age = age+1 >}
  end;;
val moi : < age : int; birthday : 'a; name : string > as 'a
# let next_age pers = pers#birthday#age;;
val next_age : < birthday : < age : 'a; .. >; .. > -> 'a
# next_age moi;;
- : int = 34
# (moi : <age: _; birthday: 'a; ..> as 'a
       :> <age: _; birthday: 'b> as 'b);;
- : < age : int; birthday : 'a > as 'a = <obj>
```

オブジェクトのための新概念

再帰型 selfやフィールド更新で再帰型が現われる．それ以外にもバイナリメソッドも再帰型を要する．

仇名型 型を読みやすくするため必要不可欠な機能．クラスを定義するとその仇名型が定義され，型推論で伝播される．再帰の場合，定義が正則(regular)でなければならない．

型引数の制約 開いたオブジェクト型は隠れた型変数をもっている．その変数に名前を付けることができないので，そのオブジェクト型全体を変数のように扱う必要がある．

仇名型の正則な展開

```
type abbrev_memo =
    Mnil
  | Mcons of Path.t * type_expr * type_expr * abbrev_memo
  | Mlink of abbrev_memo ref
```

{mutable desc: type_desc; mutable level: int}

```
type type_desc = ...
  | Tconstr of Path.t * type_expr list * abbrev_memo ref
```

Tconstr(型の一意的な名前 p，型引数 tl，展開された仇名 a)

- pの展開がaに含まれていなければ，pの定義の新しいコピーを作る．そのとき，コピーされる全てのTconstrの中に新しく展開された仇名とその型引数が記録される．

- pの展開がaに含まれていれば，それを使う．

単一化が仇名を残す

```
let rec unify env t1 t2 =
  (* First step: special cases (optimizations) *)
  ...
and unify2 env t1 t2 =
  (* Second step: expansion of abbreviations *)
  let t1' = expand_head env t1 and t2' = expand_head env t2 in
  if t1' == t2' then () else
  if (t1 == t1') || (t2 != t2') then unify3 env t1 t1' t2 t2'
  else unify3 env t2 t2' t1 t1'

and unify3 env t1 t1' t2 t2' =
  (* Third step: truly unification *)
  (* Assumes either [t1 == t1'] or [t2 != t2'] *)
  let d1 = t1'.desc and d2 = t2'.desc in
  occur env t1' t2;
  update_level env t1'.level t2;
  link_type t1' t2;
  match (d1, d2) with
  ...
```

バグ1 引数のない仇名型

型推論は単一化を多用するが，単一化の度に展開を行うとコストが高い.
最適化のためにこんなコードが入っていた.

```
let rec unify env t1 t2 =
  let t1 = repr t1 in
  let t2 = repr t2 in
  if t1 == t2 then () else
  match (t1.desc, t2.desc) with
  | (Tconstr (p1, [], a1), Tconstr (p2, [], a2)) when Path.same p1 p2 ->
      update_level env t1.level t2;
      link_type t1 t2
  ...
  | _ -> unify2 env t1 t2
```

バグ1 引数のない仇名型

もしも，過去に`t2`が展開されたことがあり，しかもその展開されたものが`t1`と単一化されたことがあれば，`a2`の中の`p2`の展開が`t1`になっているかも知れない．

とても珍しい場合だが，このコードを実行すると，`t1`の展開が`t1` 自身になることがありうる．

故に，現在はコードが修正されている．

```
| (Tconstr (p1, [], a1), Tconstr (p2, [], a2))
      when Path.same p1 p2
        && not (has_cached_expansion p1 !a1
              || has_cached_expansion p2 !a2) ->
    update_level env t1.level t2;
    link_type t1 t2
```

バグ２ 計算時間の爆発(無駄働き編)

報告 こんなプログラムがコンパイルできなくなった

```
let f = format (lit "Up for " ^^
                int ^^ (lit " day") ^^ str ^^ (lit ", ") ^^
                int ^^ (lit " hour") ^^ str ^^ (lit ", ") ^^
                int ^^ (lit " minute") ^^ str ^^ (lit ", ") ^^
                (lit "and ") ^^
                int ^^ (lit " second") ^^ str ^^ (lit ".  ") ^^
                (lit "I have used ") ^^ flt ^^ (lit " seconds of CPU time.")
                ^^ (lit "  I have allocated ") ^^ flt ^^
                (lit " bytes since I was started.  ") ^^
                (lit "I currently contain ") ^^ int ^^
                (lit " bytes on my heap."))
```

原因 各引数の型付けを２回やっていた.

このプログラムでは引数が入れ子になっているので，計算時間が指数的に増える.

バグ２ 計算時間の爆発(相互再帰編)

報告 こんなプログラムがコンパイルできない

```
let somefun ... =
  ...
  let module G = Gdome in
  ...
```

原因 上のGdomeが複雑な相互再帰で定義されるオブジェクト型を含んでいる．中では全ての型がGのものになっているが，関数が値を返すので，スコープを出るときにGへの参照を全て解消しなければならない．それは仇名の展開によって行われる．
仇名型の展開は異る枝の間では共有されないので，生成される型の大きさが指数的になってしまう．

解決 引数のない仇名型の展開を共有するようにした．引数がないので，共有されても影響はない．ただし，引数のある仇名型の間の相互再帰ならばお手あげ．

バグ２ 計算時間の爆発(サブタイピング編)

報告こんなプログラムがコンパイルできない

```
...
(x : c :> c')
...
```

原因 当初は，サブタイピングで既に辿ったノード対をリストとしてもっていた．型の展開が大きいと，そのリストへのアクセスが$O(N)$になり，アルゴリズムの複雑さが上がる．普通の例でも何分もかかったりした．

解決 現在では`type_expr`が整数のidを持っている．

```
type type_expr =
  {mutable desc: type_desc; mutable level: int; id: int}
```

サブタイピングなどではそのidに基いたハッシュテーブルを使っている．

型引数の共変性・反変性

直感的には $\tau <: \tau'$ ならば，τ list $<: \tau'$ list も正しいはず．

こういうデータ型を含むサブタイピングを可能にするために，型引数の共変性・反変性は型定義時に推論される．相互再帰もありうるので，不動点によって計算される．

```
type ('a,'b) a = A of 'a | B of ('b -> 'a, 'a -> 'b) b
and ('a,'b) b = A' of ('a,'b) a | B' of ('b -> unit)
```

この定義では，'aは共変で，'bは反変である．以下の定義で確認できる．

```
type (+'a,-'b) u = ('a,'b) a
```

バグ3 型引数の共変性

報告 このプログラムでは共変性の推論がおかしい.

```
        Objective Caml version 3.08.2

# type 'a expr = Var of 'var | App of 'app | Abs of 'abs
    constraint 'a = <var:'var; app:'app; abs:'abs; ..>;;
# (Var 3 :> <var:bool; ..> expr);;
- : < abs : 'a; app : 'b; var : bool; .. > expr = Var <unknown constructor>
```

原因 型引数が展開された型の中に現われないと，共変でも反変でもない完全に自由なものになる.

解決 Objective Caml 3.08.3 以降では，制約のある型引数の共変性は推論されない．明示的に与えられないとサブタイピングが使えない．与えられれば，中に出現される型変数の共変性もチェックされる.

多相メソッド

多相型はMLの最大の利点．しかし，元のObjective Camlには多相メソッドがなかった．

メソッドの型がオブジェクト型に含まれるので，単純に多相メソッドを許すと任意のランクの多相型を推論しなければならない．この問題は判定不能と証明されている．

Objective Camlでは多相メソッドの型が推論されない．そのメソッドを呼ぶ前に型を教えなければならない．ただし，型推論の主要性を保証するために，必要な情報が正しく与えられたかどうかを推論している．

第一級多相型の型推論を行わないので，型の等価性だけを確認すればいい．これはスコーレム化によって簡単に行える．

$$\forall\alpha\beta.\alpha \times \beta \to \beta \times \alpha = \forall\alpha'\beta'.\beta' \times \alpha' \to \alpha' \times \beta'$$

構造照合で$\alpha \leftrightarrow \beta'$と$\beta \leftrightarrow \alpha'$を発見し，その対応の元では両方の等価性が分かる．

バグ4 第一級多相型と参照型

報告 こんな不健全なプログラムが通る

```
        Objective Caml version 3.05

# type lop = {list: 'a. 'a -> 'a list};;
# let lop =
    {list = let r = ref [] in fun x -> let l = !r in r := [x]; l};;
val lop : lop = {list = <fun>}
# lop.list 3;;
- : int list = []
# lop.list true;;
- : bool list = [<unknown constructor>]
```

原因 多相性をチェックする前にvalue restructionをチェックするのを忘れた.

影響 一ヶ月後に 3.06 を出した.

バグ4 第一級多相型と再帰型

再帰型があると，型のコピーが難しくなる.

```
# fun (x : <m:'b. 'b * <p:'d. 'd * 'c * 'a> as 'c> as 'a) -> x#m;;
- : (< m : 'b. 'b * < p : 'd. 'd * 'c * 'a > as 'c > as 'a) ->
    ('f * < p : 'g. 'g * 'e * 'a > as 'e)
```

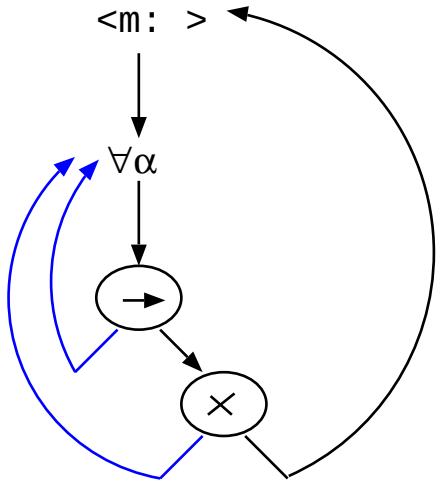
「単一化については特別なことをする必要はない」Didier Rémy

事実 [ICFP04] でPottierとGauthierがFμの型の等価性を判定するアルゴリズムを与えた.

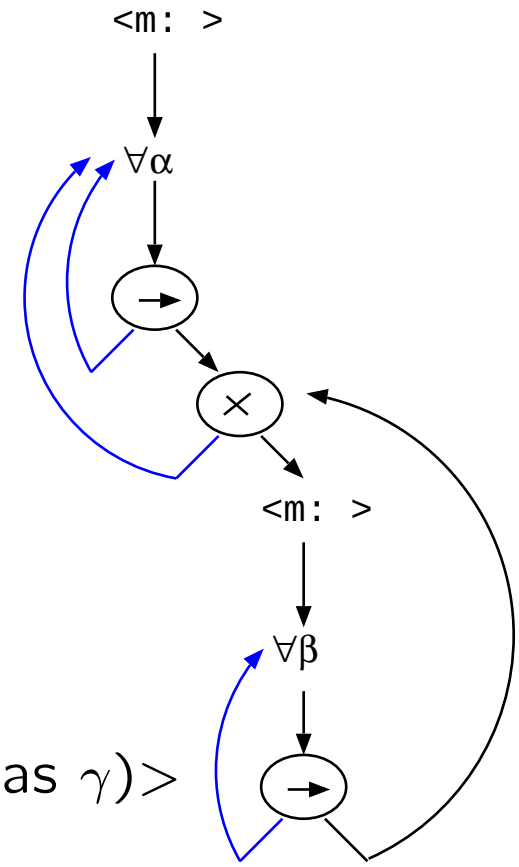
事実 Objective CamlではもともとFμがエンコードできる.

再帰型があると話はそう簡単ではなかった…

バグ4 第一級多相型と再帰型



$<m : \forall\alpha.\alpha \to (\alpha \times \gamma)>$ as γ



$<m : \forall\alpha.\alpha \to (\alpha \times <m : \forall\beta.\beta \to \gamma>$ as $\gamma)>$

バグ4 第一級多相型と再帰型

```
        Objective Caml version 3.08.2

# let f x =
    (x : <m : 'a. 'a -> ('a * <m:'c. 'c -> 'bar> as 'bar)>
       :> <m : 'a. 'a -> ('a * 'foo)> as 'foo);;
# let o =
  object method m : 'a. 'a -> ('a * <m:'c. 'c -> 'bar> as 'bar) =
    fun x ->
      (x, object (self) method m : 'c. 'c -> _ = fun y -> (x, self) end)
  end;;
val o : < m : 'a. 'a -> ('a * < m : 'c. 'c -> 'b > as 'b) > = <obj>
# let (x, o') = (f o)#m 3;;
val x : int = 3
val o' : < m : 'b. 'b -> 'b * 'a > as 'a = <obj>
# let x' = fst (o'#m true);;
val x' : bool = <unknown constructor>
```

まとめ

Objective Caml は多くの機能を持っている

- 参照型 (relaxed value restriction)
- モジュールとファンクター
- オブジェクトと再帰型
- サブタイピング，仇名型，制約付き型引数
- 第一級多相型
- 多相ヴァリアント
- 再帰モジュール
- ラベル付引数と省略可能引数

それぞれの機能が互いに影響しあうので，特別な場合は見落しやすい．

それでもgccよりずっと安全と思われる