More Logic More Types
ML/OCamlの型システムとGADT

Jacques Garrigue (名古屋大学)
協力者　Jacques Le Normand (Google)
Didier Rémy (INRIA)

@garriguejej　　ocamlgadt

最初に論理があった

- MLは定理証明システムEdinburgh LCFのメタ言語である
- 証明の組立と他の処理を同じ言語ですることを目指した
- MLの型システムは証明の正しさを保証してくれる

Curry-Howard 同型

論理	ML
命題	型
証明	プログラム
含意（$\supset$）	関数型（$\rightarrow$）
論理積（$\wedge$）	直積（$\times$）

プログラムは証明である

命題論理

$$(\text{寝坊} \supset \text{遅刻}) \wedge \text{寝坊} \supset \text{遅刻}$$

ML

```
type nebou and chikoku ;;

let mp (args : (nebou -> chikoku) * nebou) =
  let (f, x) = args in f x ;;
val mp : (nebou -> chikoku) * nebou -> chikoku = <fun>
```

型推論でより強力に?!

さっきのプログラムから型情報を消す

```
let mp args =
  let (f, x) = args in f x ;;
val mp : ('a -> 'b) * 'a -> 'b = <fun>
```

勝手に多相型になった

論理では

$$\forall P, Q.(P \supset Q) \wedge P \supset Q$$

いわゆる *modus ponens* である

命題変数つきの命題論理になり，実際の証明が可能

データ型は論理和

Brouwer-Heyting-Kolmogorov 解釈によれば，論理和が直和になる

$$A \vee B \text{の証明} = \langle 1, A \text{の証明} \rangle \text{または} \langle 2, B \text{の証明} \rangle$$

MLでは

```
type ('a,'b) sum = Left of 'a | Right of 'b
```

多相バリアントの極意

データ型は論理和なら，多相バリアントは集合の和である

```
type myint = [‘Int of int]
type mybool = [‘Bool of bool]

type myint_mybool = [myint | mybool]
type myint_mybool = [ ‘Bool of bool | ‘Int of int ]
```

オブジェクトにも論理的解釈

自然言語などで使われるフィーチャー論理に対応する

$$\texttt{< x : int; y : int; color : string >}$$

$$\exists p.\ p\#x : \texttt{int} \wedge p\#y : \texttt{int} \wedge p\#color : \texttt{string}$$

他にも色々

- 多相メソッド ⇔ Girard の System F
- プライベート列型 ⇔ Cardelli等の $F_{\leq}$
- プライベートデータ型・略称型 ⇔ 不変量を持った型
- ラベルつき引数 ⇔ DiCosmo の型同型の実現
- 幽霊型 ⇔ 手作りの型制約・詳細型

次のステップ：GADT（一般化代数的データ型）

- 場合ごとに異なる型パラメータを許す再帰データ型の拡張
- Coqなどの依存性を含む帰納的データ型に似ている

```
type _ expr =
  | Int : int -> int expr
  | Add : (int -> int -> int) expr
  | App : ('a -> 'b) expr * 'a expr -> 'b expr
App (Add, Int 3) : (int -> int) expr
```

- 不変量 と 証明 が表現できる
- おまけに 存在型 も提供される

実装は既にある機能を利用する

- 多くの例では 多相的再帰 が必要
  → OCaml 3.12 で導入，過去にGADTのために作った
- パターンマッチングで型の場合分けも行う → 局所抽象型を利用
- 両方を組み合わせた新しい構文を導入

```
let rec eval : type t. t expr -> t = function
  | Int n -> n                                    (* t = int *)
  | Add -> (+)                          (* t = int -> int -> int *)
  | App (f, x) -> eval f (eval x)                 (* 多相的再帰 *)
val eval : 'a expr -> 'a = <fun>

eval (App (App (Add, Int 3), Int 4));;
- : int = 7
```

型注釈の新しい構文

前のスライドの構文

$$\texttt{let rec}\ f : \texttt{type}\ t_1\ \ldots\ t_n.\tau = body$$

は以下の構文に変換される

$$\texttt{let rec}\ f : \alpha_1\ \ldots\ \alpha_n.[\alpha_1 \ldots \alpha_n / t_1 \ldots t_n]\tau =$$
$$\texttt{fun}\ (\texttt{type}\ t_1)\ \ldots\ (\texttt{type}\ t_n) \rightarrow (body : \tau)$$

再帰的多相関数を定義する．局所型が$body$で利用できる．

多重型関数への応用

以下の関数は正しい

```
let rec neg : 'a. 'a -> 'a = function
  | (n : int)  -> -n
  | (b : bool) -> not b
  | (a, b)     -> (neg a, neg b)
  | x          -> x
val neg : 'a -> 'a
```

型を前提とする言語では問題がある．(関数論理型言語はOK)

- 実行時型情報 が必要になる
- 網羅性 のためにデフォルトが必要

GADT では解決できる

タグ付き実装

普通の代数的データ型を使いながら，パラメータにより中身の種類を見せる

```
type _ data =
  | Int  : int -> int data
  | Bool : bool -> bool data
  | Pair : 'a data * 'b data -> ('a * 'b) data

let rec neg : type a. a data -> a data = function
  | Int n  -> Int (-n)
  | Bool b -> Bool (not b)
  | Pair (a, b) -> Pair (neg a, neg b)
```

入力と結果の型が一致することを保証

タグなし実装

タグをデータに含めなくても実装できる

```
type _ ty =
  | Tint : int ty
  | Tbool : bool ty
  | Tpair : 'a ty * 'b ty -> ('a * 'b) ty

let rec print : type a. a ty -> a -> string = fun t d ->
  match t, d with
  | Tint, n -> string_of_int n
  | Tbool, b -> if b then "true" else "false"
  | Tpair (ta, tb), (a, b) ->
      Printf.sprintf "(%s, %s)" (print ta a) (print tb b)
```

依存性がパターンの左から右に流れなければならない

GADTと等価性

```
type zero                                              (* 型レベルの自然数 *)
type _ succ
type _ nat =                                           (* 自然数の証人 *)
  | NZ : zero nat                                      (* 実際の自然数でもある *)
  | NS : 'a nat -> 'a succ nat

type (_,_) equal = Eq : ('a,'a) equal                  (* 等価性の証人 *)
let convert : type a b. (a,b) equal -> a -> b = fun Eq x -> x

let rec sameNat :
  type a b. a nat -> b nat -> (a,b) equal option =
  fun a b -> match a, b with                           (* 等価性の判定 *)
  | NZ, NZ -> Some Eq                                  (* a = b = zero *)
  | NS a', NS b' ->                                    (* a = a' succ, b = b' succ *)
      begin match sameNat a' b' with
      | Some Eq -> Some Eq                             (* a' = b' => a = b *)
      | None -> None
      end
  | _ -> None                                          (* 異なる場合 *)
```

GADTで型レベル関数

OCamlでは型レベル関数がないが，GADTで関係が定義できる

```
type (_,_,_) plus =
  | PlusZ : 'a nat -> (zero, 'a, 'a) plus
  | PlusS : ('a,'b,'c) plus -> ('a succ, 'b, 'c succ) plus
```

関数であることは手で証明しなければならないが，自明である

```
let rec plus_func : type a b m n.
  (a,b,m) plus -> (a,b,n) plus -> (m,n) equal =
  fun p1 p2 ->
  match p1, p2 with
  | PlusZ _, PlusZ _ -> Eq              (* a = zero => m = n = b *)
  | PlusS p1', PlusS p2' -> (* a = a' succ, m = m' succ, ... *)
      let Eq = plus_func p1' p2' in Eq    (* m' = n' => m = n *)
```

GADTで証明

これだけあれば普通に証明できる

```
let rec plus_assoc :                                    (* 足し算の結合律 *)
  type a b c ab bc m n.
    (a,b,ab) plus -> (ab,c,m) plus ->
    (b,c,bc) plus -> (a,bc,n) plus -> (m,n) equal =
  fun p1 p2 p3 p4 ->
  match p1, p4 with
  | PlusZ b, PlusZ bc ->                        (* a = zero => bc = n *)
      let Eq = plus_func p2 p3 in Eq              (* m = bc => m = n *)
  | PlusS p1', PlusS p4' -> (* a = a' succ, n = n' succ, ... *)
      let PlusS p2' = p2 in                          (* m = m' succ *)
      let Eq = plus_assoc p1' p2' p3 p4' in Eq
```

GADTの証明能力

- 述語や関係をGADTとして定義することで**一階述語論理**の証明ができる

- しかし，無限ループが定義できるので，**論理的な健全性が保証されていない**

```
type (_,_) rel =
  | R1 : (int, bool) rel
  | R2 : ('a,'b) rel -> ('b,'a) rel

let rec any = R2 (R2 any) ;;
val any : ('a, 'b) rel = R2 ...
```

様々な応用

GADTを利用するアルゴリズムが既に多く知られている

- データ構造の不変量を保つアルゴリズム
  例 均衡木構造 (Tim Sheard など)

- 型付構文木
  例 型付λ計算と評価機の実装 (同上)

- パーザ
  Menhirが生成するパーザがGADTで型付け可能

- 多くのDSL: GUI, データベースなど

OCamlのGADTの特徴　その1：網羅性

不可能な場合を排除したい

```
let rec equal : type a. a data -> a data -> bool = fun a b ->
  match a, b with
  | Int m, Int n ->
      m - n = 0
  | Bool b, Bool c ->
      if b then c else not c
  | Pair(a1,a2), Pair(b1,b2) ->
      equal a1 b1 && equal a2 b2
```

型情報から，a と b が同じ種類であることが分かるので，以上の場合で足りる．

網羅性の検証は他の場合を生成し，不可能であることを証明している．アルゴリズムは健全であるが，完全ではない．

網羅性の難しさ

- 通常の型推論は特定の文脈で単一化可能かどうかを調べる
- 網羅性では他の文脈も考えなければならない

```
type (_,_) eq = Eq : ('a,'a) eq

module M : sig
  type t and u
  val eq : (t,u) eq
end = struct
  type t = int and u = int
  let eq = Eq
end

match M.eq with Eq -> "ここでは t = u !"
```

矛盾関係

単一化とは別に，矛盾関係を定義することで問題を解決.

２つの型が矛盾しているとは

- 構造的に異なる（例 関数と組)
- データ型な，表現が異なれば矛盾する（プライベート型も同様)
- 抽象型は，どちらも初期定義（Pervasives）または定義中のモジュールにあるとき

初期定義を含めないと `int` と `bool` すら区別できない!

パターンマッチングでは，型が矛盾しない限り単一化が失敗しないように改造する．ただし，細分化できるのは局所抽象型のみである.

OCamlのGADTの特徴　その２：曖昧性の検証

GADTの型推論は難しい．パターンマッチングの前後の型が分からないと曖昧性が生じてしまう．

```
type _ t = Int : int t

let f (type a) (x : a t) =
  match x with Int -> 1                    (* a = int *)
```

fの結果の型はintかaで定まらないと解釈できる．

しかしfの型は本当に曖昧と思うべきか?

曖昧性の定義

```
let f' (type a) (x : a t) =
  match x with Int -> true                    (* a = int *)

let g (type a) (x : a t) (y : a) =
  match x with Int -> (y > 0)
```

f'やgには曖昧性がない.
fとf'は等式を使っていない.
gは等式を使っているが，yの型が指定されている.

曖昧性のより良い定義を考えなければならない

能動的な曖昧性

– 等式を活用している

– かつその活用が型注釈により隠蔽されていない

ときにのみ曖昧性があるという．たとえば

```
let g' (type a) (x : a t) (y : a) =
  match x with Int -> if y > 0 then y else 0
Error: This instance of int is ambiguous
```

単一化の対象となった全ての型を記憶することでそれを簡単に調べることができる．

Haskellとの比較

	OCaml	Haskell
GADT	4.00から	2005年頃から
網羅性チェック	○	×
型推論	曖昧性判定	OutsideIn
型レベル関数	×	最近

OCaml 4.00

- GADTの導入 (詳しくは ocamlgadt で検索)
- 第一級モジュールの型推論
- バイナリアノテーションの出力
- エラーメッセージとワーニングの大幅な改良
- などなど

間も無くβ版をリリース