型システムとプログラミング言語

Jacques Garrigue
名古屋大学

型とプログラミング言語

- **型**とはデータや関数の種類を区別するもの
- プログラミング言語には型が**必須ではない**
- しかし、**安全性**と**効率**のために、型が有利に働く
- 本日の談話会はプログラミング言語における**型**、特に**代数的データ型**の役割を**実践**と**理論**の観点から紹介する

型のない言語：シェルスクリプト

文字列しかない言語

```
$ print_lines ()
> for i in $1; do
>   echo $i
> done
$ print_lines "a b c"
a
b
c
```

上の関数は文字列を空白のところで区切って、行に分けて印刷する。他にデータ構造がないので、もっと複雑なデータを渡すことができない。たとえば、上の関数で空白を含む行が出力できない。

全てのコマンドが文字列を受け付けるので、ある意味でエラーは存在しない。

動的型の言語：Emacs Lisp

文字列や整数以外に任意の値を含むリストというデータが扱える。

```
(defun print-lines (lines)
  (dolist (str lines) (insert str) (insert "\n")))

(print-lines '("a" "b c"))
a
b c

(print-lines '(("a") ("b" "c")))
error
```

print-linesが文字列のリストをもらっているので、文字列の中に空白が入ってもいい。

しかし、それ以外のデータを渡すとエラーになる。

効率のための型：C言語

型によってメモリの中の表現が決まる。

```
#include<stdio.h>
void print_lines(char *lines[]) {
  while (*lines != NULL) printf("%s\n", *lines++);
}
int main() {
  char *lines[] = { "a", "b c", NULL };
  print_lines(lines);
}
$ ./print_lines
a
b c
```

効率がよく、様々なデータ構造が表現できる。

効率のための型：C言語

型によってメモリの中の表現が決まる。

```
#include<stdio.h>
void print_lines(char *lines[]) {
  while (*lines != NULL) printf("%s\n", *lines++);
}
int main() {
  char *lines[] = { "a", "b c", (char*)3, NULL };
  print_lines(lines);
}
$ ./print_lines
a
b c
Segmentation fault
```

しかし、メモリを読むときに型を信じるので、未定義な動作になることが多い。

C言語が安全でない理由

- あるデータ構造の範囲を越えられる。
  (前スライドの `lines++`)
- 便宜のために型のキャストを許している。
  (前スライドの `(char*)3`)
- データを並べておく構造体は安全だが、中身のいずれかだけをおく共用体の安全な使い方がプログラマの責任に任されている。

```
struct person {
  char name[20];
  int age;
}
```

```
union person_or_id {
  struct person *person;
  int id;
}
```

型安全性

定理 1 もしもプログラムPの型付けが成功していれば、Pの実行時に型の不整合に由来するエラーが起きない。

上記の性質を満すプログラミング言語を型安全また強い型付けを持つという。

λ計算と型理論

強い型付けを持つプログラミング言語の多くが型付きλ計算に基いている。

- λ計算は1930年代にChurchによって提案された計算系。表現力が一般的な計算のモデルであるTuring機械と同等であると証明された。
- 言語の定義が簡潔で、関数の作り方と適用を中心としている。
- 型付λ計算はλ計算に型理論の原理を入れたもので、直観主義論理との対応(Curry-Howard対応)が知られている。

型付λ計算：型と項

型

$$\begin{array}{rcll} \tau & ::= & b & \textbf{基本型} \\ & | & \tau \to \tau & \textbf{関数型} \end{array}$$

項

$$\begin{array}{rcll} t & ::= & x & \textbf{変数} \\ & | & \lambda(x:\tau) \to t & \textbf{関数構成} \\ & | & t(t) & \textbf{関数適用} \end{array}$$

例

$$\lambda(plus : nat \to nat \to nat) \to \lambda(y : nat) \to plus(y)(y)$$

型付λ計算：型判定

型環境

$$\Gamma ::= \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$$

型判定：Γ の元で、項 t が型 τ を持つ

$$\Gamma \vdash t : \tau$$

型導出：以下の規則を組合せて作る

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash t_1 : \tau_1}{\Gamma \vdash \lambda(x : \tau) \to t_1 : \tau \to \tau_1}$$

$$\frac{\Gamma \vdash t_1 : \tau_2 \to \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1(t_2) : \tau}$$

型付λ計算：評価

計算結果と文脈

$$\begin{array}{rcll} v & ::= & c & \textbf{定数} \\ & | & (\eta, \lambda(x:\tau) \to t) & \textbf{関数閉包} \\ \eta & ::= & \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} & \textbf{文脈} \end{array}$$

自然意味論 $\eta \models t \downarrow v$ **文脈**η**の中で**t**を**v**に評価する部分関数**

$$\frac{x \mapsto v \in \eta}{\eta \models x \downarrow v} \qquad \eta \models \lambda(x:\tau) \to t \downarrow (\eta, \lambda(x:\tau) \to t)$$

$$\frac{\eta \models t_1 \downarrow (\eta', \lambda(x:\tau) \to t') \quad \eta \models t_2 \downarrow v_2 \quad \eta', x \mapsto v_2 \models t' \downarrow v_3}{\eta \models t_1(t_2) \downarrow v_3}$$

型付λ計算：型安全性

定理 2 もしもプログラムtについて$\emptyset \vdash t : \tau$ ならば、tの評価が必ず成功し、ある結果vを返す: $\emptyset \models t \downarrow v$

ここに定義した核のλ計算にはループがなく、型付けされたプログラムが全て止まる。型安全性のもっと一般的な定義では、プログラムが止まらない場合を考慮する必要がある。

代数的データ型

前述の型付きλ計算では実際のプログラミングができない。しかし、代数的データ型を追加するだけで可能になる。

定義

$$\begin{aligned}\texttt{type}\ a\ = \ & C_1\ \texttt{of}\ \tau_{11} \times \ldots \times \tau_{1m_1} \\ \mid\ & C_2\ \texttt{of}\ \tau_{21} \times \ldots \times \tau_{2m_2} \\ & \vdots \\ \mid\ & C_n\ \texttt{of}\ \tau_{n1} \times \ldots \times \tau_{nm_n}\end{aligned}$$

各C_iが型aの構成子になるように次の規則が型体系に追加される。

$$\frac{\Gamma \vdash t_1 : \tau_{i1} \quad \ldots \quad \Gamma \vdash t_{m_i} : \tau_{im_i}}{\Gamma \vdash C_i(t_1, \ldots, t_{m_i}) : a}$$

($m_i = 0$のとき、$\Gamma \vdash C_i : a$とする)

代数的データ型の例

型 a と b の対

```
type pair_a_b = Pair_a_b of a * b
```

型 a の値があるかないか

```
type option_a = Some_a of a | None
```

型 a のリスト

```
type list_a = Nil | Cons of a * list_a
[a1; a2; a3] = Cons (a1, Cons (a2, Cons (a3, Nil)))
```

ペアノ式自然数

```
type nat = Zero | Succ of nat
3 = Succ (Succ (Succ (Zero)))
```

パターン・マッチング

代数的データ型の場合分けを行うために、新しい`match`構文を導入する。

$$\begin{array}{l}\texttt{match } t \texttt{ with} \\ \mid C_1(x_1, \ldots, x_{m_1}) \to t_1 \\ \mid \quad \vdots \\ \mid C_n(x_n, \ldots, x_{m_n}) \to t_n\end{array}$$

型付け

$$\frac{\Gamma \vdash t : a \qquad \Gamma, x_1 : \tau_{i1}, \ldots, x_{m_i} : \tau_{im_i} \vdash t_i : \tau \ (1 \le i \le n)}{\Gamma \vdash \texttt{match } t \texttt{ with } C_1(x_1, \ldots, x_{m_1}) \to t_1 \mid \ldots \mid C_n(x_n, \ldots, x_{m_n}) \to t_n : \tau}$$

評価

$$\frac{\eta \models t \downarrow C_i(v_1, \ldots, v_{m_i}) \qquad \eta, x_1 \mapsto v_1, \ldots, x_{m_i} \mapsto v_{m_i} \models t_i \downarrow v}{\eta \models \texttt{match } t \texttt{ with } C_1(x_1, \ldots, x_{m_1}) \to t_1 \mid \ldots \mid C_n(x_n, \ldots, x_{m_n}) \to t_n \downarrow v}$$

パターン・マッチングの例

ここから、関数型プログラミング言語OCamlの構文を使う。

対からの抽出

```
let fst_a_b (p : pair_a_b) = match p with Pair_a_b (a,b) -> a
val fst_a_b : pair_a_b -> a
let snd_a_b (p : pair_a_b) = match p with Pair_a_b (a,b) -> b
val snd_a_b : pair_a_b -> b
```

自然数の足し算

```
let rec add (m : nat) (n : nat) =
  match m with
  | Zero -> n
  | Succ m' -> Succ (add (m') (n))
val add : nat -> nat -> nat
```

多相型

対やリストの定義が任意の型に対してできる。

```
type ('a,'b) pair = Pair of 'a * 'b
type 'a option = Some of 'a | None
type 'a list = Nil | Cons of 'a * 'a list
```

代数的データ型だけでなく、それを使った関数も多相型になる。

```
let fst (p : ('a,'b) pair) = match p with Pair (a,b) -> a
val fst : ('a,'b) pair -> 'a
let snd (p : ('a,'b) pair) = match p with Pair (a,b) -> b
val snd : ('a,'b) pair -> 'b
let rec append (l1 : 'a list) (l2 : 'a list) =
  match l1 with
  | None -> l2
  | Cons (a, l1') -> Cons (a, append (l1') (l2))
val append : 'a list -> 'a list -> 'a list
```

再帰関数・不動点

前出の`let rec`は再帰的な定義に使われる。一見体系の拡張に見えるが、実は代数的データ型を入れた時点で再帰関数が定義できるようになる。

```
type 'a recc = In of ('a recc -> 'a)
let out (In x) = x
val out : 'a recc -> 'a recc -> 'a
let y (f : ('a -> 'b) -> ('a -> 'b)) =
  (fun x a -> f (out x x) a) (In (fun x a -> f (out x x) a))
val y : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
```

上記のyは型なしλ計算では再帰関数を定義するために使う不動点演算子である。これがあれば任意の再帰的関数が定義できる。

型推論

ここまでで、代数的データ型が型安全性を保ちながら、C言語の型定義より強力であることが伝わったはずである。
しかし、多相性を利用してパラメータ付きの型を増やすと、段々型を書くのが億劫になる。特にOCamlやHaskellなど関数型プログラミング言語ではなるべく変数を共有せずに、全ての状態を関数の入出力に表そうとすると、型が複雑になりがちである。

そのために、型推論が強い型付けの言語に不可欠な機能である。全ての型付け規則から項の中の型を消すと、以下の重要な定理がDamasとMilnerによって証明された。

定理 3 (型付けの主要性) もしもプログラムPと環境Γについて$\Gamma \vdash P : \tau$があるτについて導出できるなら、型推論アルゴリズムがτより一般的な型τ'を返す。

型推論の原理

型導出を再構築しようとして、等式を抽出する

$$\dfrac{\dfrac{\Gamma \vdash f : \alpha_f \quad \Gamma \vdash x : \alpha_x}{\Gamma \vdash f(x) : \alpha_2} \quad \dfrac{\Gamma \vdash g : \alpha_g \quad \Gamma \vdash x : \alpha_x}{\Gamma \vdash g(x) : \alpha_3}}{\dfrac{\Gamma \vdash f(x)(g(x)) : \alpha_1}{\emptyset \vdash \lambda f \to \lambda g \to \lambda x \to f(x)(g(x)) : \alpha}}$$

ここで$\Gamma = f : \alpha_f, g : \alpha_g, x : \alpha_x$
各推論規則の条件をまとめると以下の等式が得られる

$$\{\alpha = \alpha_f \to \alpha_g \to \alpha_x \to \alpha_1, \alpha_2 = \alpha_3 \to \alpha_1,\ \alpha_f = \alpha_x \to \alpha_2, \alpha_g = \alpha_x \to \alpha_3\}$$

単一化とは、変数を含む方程式の集合を木構造領域の上で解く方法である。元々論理学でレゾリューションという手続きを行うために考えられたアルゴリズムだが、ここでも使える。特に、単一化の *mgu*(最も一般的な単一化代入) の存在が前述の主要性定理に直接的につながる。ここで単一化代入σを計算すると

$$\sigma(\alpha) = (\alpha_x \to \alpha_3 \to \alpha_1) \to (\alpha_x \to \alpha_3) \to \alpha_x \to \alpha_1$$

主要性の重要さ

- **型推論の結果が理解しやすい**
  型導出を一つ見付けらば、推論が成功る

- **モジュラーな型推論を可能にする**
  各部分項に対する推論の結果を後から組み合わせる

- **アルゴリズムを改良したときの頑丈さ**
  より一般的な解を返すように推論を改良した場合、今までのプログラムが型付けできる

- **妥当性の基準**
  良い条件を満しているので、安心して使える

型推論の応用

上記の定理によれば、型を全く書かなくて良い！
ついでに引数の括弧も省略する。
黄色はアルゴリズムが推論した型である。

```
let fst p = match p with Pair (a,b) -> a
val fst : ('a,'b) pair -> 'a
let snd p = match p with Pair (a,b) -> b
val snd : ('a,'b) pair -> 'b
let rec append l1 l2 =
  match l1 with
  | None -> l2
  | Cons (a, l1') -> Cons (a, append l1' l2)
val append : 'a list -> 'a list -> 'a list
```

柔軟性

汎用性と安全性の両面で代数的データ型が関数型プログラミング言語で中心的な役割を果してる。典型的なプログラム開発では、まず問題に合ったデータ型を定義し、それに合わせてアルゴリズムを書いていく。

しかし、代数的データ型の多相性が足りない場合もある。同じ構成子を異なるデータ型で使いたいときである。

部分型 ある型の構成子の一部しか含まない型を定義したいとき

類似型 一部の構成子の引数が異なる型

自立型 関係のない型に同じ名前の構成子が使いたい

部分型と類似型は言語処理系などでよく現れる。自立型は元々型のない世界で開発されたライブラリを利用しようとするときに起こりがちである。

多相ヴァリアント

多相ヴァリアントは部分型・類似型・自立型に対応できる方法である。アイデアは、型定義自体をなくすことである。

```
let l = [ `Feet 3.0; `Mm 33.2 ]
val l : [> `Feet of float | `Mm of float ] list
let normalize len =
  match len with
    `Feet n -> `Inch (n *. 12.)
  | `Inch n -> `Inch n
  | `Cm n   -> `Mm (n *. 10.)
  | `Mm n   -> `Mm n
val normalize :
  [< `Cm of float | `Feet of float | `Inch of float
  | `Mm of float ] -> [> `Inch of float | `Mm of float ]
```

再帰型の推論

通常の単一化は再帰型 (無限な正則木) を排除する。例えば、代数的データ型の定義では、(’a list as ’a) は意味をなさない。しかし、多相ヴァリアントでは再帰型も意味を持つ。

```
let rec length l =
  match l with
    ‘Nil -> 0
  | ‘Cons (a, t) -> 1 + length t
val length : ([< ‘Cons of ’b * ’a | ‘Nil ] as ’a) -> int
```

この型は非自明な部分型を無数に持つ。例えば、長さが偶数のリスト

```
([‘Cons of ’b * [‘Cons of ’b * ’a] | ‘Nil ] as ’a)
```

型の解釈

多相ヴァリアントの型は型変数に関する条件として解釈される。

$$(U, L, T) \in (\mathsf{Fin}(\mathcal{L}) \cup \mathcal{L}) \times \mathsf{Fin}(\mathcal{L}) \times \mathsf{Fin}(\mathcal{L} \times \mathcal{T})$$

具体的には、パターンはUに、値はLに含まれる。

$$\texttt{[> `Feet of float | `Mm of float] list} =$$
$$\alpha :: (\mathcal{L}, \{\texttt{Feet}, \texttt{Mm}\}, \{\texttt{Feet} \mapsto \texttt{float}, \texttt{Mm} \mapsto \texttt{float}\}) \triangleright \alpha\ \texttt{list}$$

$$\texttt{[<`C of f |`F of f |`I of f |`M of f] -> [>`I of f |`M of f]} =$$
$$\alpha_1 :: (\{\texttt{C}, \texttt{F}, \texttt{I}, \texttt{M}\}, \emptyset, \{\texttt{C} \mapsto \texttt{f}, \texttt{F} \mapsto \texttt{f}, \texttt{I} \mapsto \texttt{f}, \texttt{M} \mapsto \texttt{f}\}),$$
$$\alpha_2 :: (\mathcal{L}, \{\texttt{I}, \texttt{M}\}, \{\texttt{I} \mapsto \texttt{f}, \texttt{M} \mapsto \texttt{f}\}) \triangleright \alpha_1 \to \alpha_2$$

型推論の主要性

主要性への近道はmguのある単一化である。しかし、気を付けないとmguが失われる。

$$\{\alpha = [< \text{`A of int} \mid \text{`B}], \alpha = [< \text{`A of } \alpha_1 \mid \text{`B}]\}$$

ここですなおに、$\sigma(\alpha_1) = \texttt{int}$ とすると、次の等式を追加したときに単一化が失敗する。

$$\{\alpha = [\text{`B}], \alpha_1 = \texttt{bool}\}$$

しかし、元々$\sigma = \{\alpha \mapsto [\text{`}B]\}$ があり、そちらが前述の解と比較できなくて、最終的に解につながる。

この問題を解決するために、新しい型を導入しなければならない。

$$\sigma = \{\alpha \mapsto [< \text{`A of int \& } \alpha_1 \mid \text{`B}]\}$$

意味は、`Aの値がαに含まれるときだけ、$\alpha_1 = \texttt{int}$ でなければならない。

多相バリアントの利用例

LablGL OpenGL という3Dグラフィックスのライブラリのインターフェイス

部分型・自立型

Cでありながら、Lispに似たデザイン：多くのシンボルが全体で定義され、関数ごとに受け付けものが違う。

LablTk Tcl/Tk というGUIライブラリとのインターフェイス

類似型

関数の呼び方が似ていながら、少し違う。

LablGTK Gtk+ というGUIライブラリとのインターフェイス

独立型

同じ名前を使うCのEnumが多く定義されている

一般化代数的データ型（GADTs）

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

```
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
```

- 様々な不変量や証明を表現できる
- おまけに存在型も使える

GADTのパターン・マッチング

- 各場合で型パラメータが構成子の定義と単一化される
- 新しい等式が環境に追加される
- 複雑な場合には多相的再起も使われる

```
let rec eval : type a. a expr -> a = function
  | Int n -> n                                    (* a = int *)
  | Add -> (+)                          (* a = int -> int -> int *)
  | App (f, x) -> eval f (eval x)                 (* 多相的再起呼出 *)
val eval : 'a expr -> 'a = <fun>

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

GADTの型推論

- パターン・マッチングで得た等式は局所的にしか使えない
- 外に見せる型をどう選べばいいか分からない

```
type _ t = Int : int t

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

解決方法

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

- 投げやりな解：適当に選ぶ
  当然、主要性が失われる
- 保守的な解：完全な型注釈を求める
  中と外の接点はパターンの変数と結果だけではない。他の変数にも注釈を求めなければならない。
- 一方的な解：外の制約を先に計算する
  Haskell(GHC) で使われる方法。主要な型に限定できるが、主要性自体は失われる。
- 曖昧性の再定義：等式が使われなければ OK
  OCaml で使われる方法。主要性が保たれるが定義が若干複雑。

GADTと述語論理

```
exception Axiom
type falso and 'p not = 'p -> falso
let ex_falso (f : falso) = raise Axiom
type ('a,'b) ior = Inl of 'a | Inr of 'b
let classic () : ('a, 'a not) ior = raise Axiom
(* Drinkers Paradox: ∃x, Drinks(x) → ∀y, Drinks(y) *)
type _ drinks
type all_drink = {all: 'y. unit -> 'y drinks}
type non_drinker = Nd : 'x drinks not -> non_drinker
type drinkers_paradox =
  Drinker : ('x drinks -> all_drink) -> drinkers_paradox
let proof () : drinkers_paradox =
  match classic () with
  | Inl (Nd nd) -> Drinker (fun d -> ex_falso (nd d))
  | Inr nnd -> Drinker (fun d ->
      {all = fun () -> match classic () with
      | Inr nd -> ex_falso (nnd (Nd nd))
      | Inl d -> d})
```

網羅性

パターン・マッチングではパターンは自由に書ける。

```
let third l =
  match l with
  | Cons (a, Cons (b, (Cons (c, l)))) -> c
  | Cons (a, Cons (b, Nil)) -> b
  | Nil -> raise Not_found
```

網羅性のチェックが自動的に足りない場合を発見する。

```
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Cons (_, Nil)
```

特に型の定義を拡張したときに重宝する機能である。

多相ヴァリアントの網羅性

多相ヴァリアントの網羅性は代数的データ型と大きく異ならないが、推論される型に影響を与える。

```
let f = function
  | `A -> 1
  | `B -> 2
val f : [< `A | `B ] -> int
let g = function
  | `A -> 1
  | `B -> 2
  | _  -> 3
val g : [> `A | `B ] -> int
let h = function
  | `A, `A -> 1
  | `A, `B -> 2
  | `B, _  -> 3
val h : [< `A | `B ] * [< `A | `B ] -> int
```

GADTの網羅性

GADTの場合、型パラメータが可能な値の集合を変えるので、正確な網羅性が難しくなる。

```
type _ t =
  | Int : int t
  | Bool : bool t

let g : int t -> int = function
  | Int -> 1                                    (* boolは不可能 *)

let h : type a. a t -> a t -> bool =
  fun x y -> match x, y with
  | Int, Int -> true
  | Bool, Bool -> true                  (* 型が共有されるので網羅的 *)
```

網羅性が決定不能問題

任意のGADT `t`について、次の関数を考える

```
let f : (char t) option -> int = function
  | None -> 0
```

`f`が網羅的かどうかは、`char t`の値が作れるかどうかと同値である。前のページの例では`int t`と`bool t`しか作れないので、`f`が網羅的である。

一般的には、`t`の各場合を**述語論理のホーン節**として見ることができる。

$$C_i : \forall \bar{\alpha}, (\exists \bar{\beta}, \tau_{i1} \wedge \ldots \wedge \tau_{im_i}) \rightarrow \tau\, t$$

逆に任意のホーン節がGADTとして書くことができる。

木構造領域の上で、ホーン節の集合における**推論可能性が決定不能問題**なので、**網羅性も決定不能**。

ありがとうございました

- **OCamlについて**

  http://www.ocaml.org/

- **今回のスライドと後半の拡張に関する論文**

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