Jacques Garrigue, 2019 年 6 月 5 日

Curry-Howard 対応

1 プログラムの型付け

型 $\tau, \theta ::= \mathsf{nat} \mid \mathsf{Z} \mid \ldots \mid \theta \to \tau \mid \tau \times \theta$ データ型, 関数型, 直積

型判定 $\Gamma \vdash M : \tau$ $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ という仮定のもとで，M が型 τ をもつ.

型付け規則 Coq の式は以下の型付け規則によって型付けされる.

変数 $\Gamma \vdash x : \tau$ $(x : \tau$はΓに含まれる$)$

定義 $$\frac{\Gamma \vdash M : \theta \quad \Gamma, x : \theta \vdash N : \tau}{\Gamma \vdash \mathtt{let}\ x\ \mathtt{:=}\ M\ \mathtt{in}\ N : \tau}$$

抽象 $$\frac{\Gamma, x : \theta \vdash M : \tau}{\Gamma \vdash \mathtt{fun}\ x{:}\theta \Rightarrow M : \theta \to \tau}$$

不動点 $$\frac{\Gamma, f : \theta \to \tau, x : \theta \vdash M : \tau}{\Gamma \vdash \mathtt{fix}\ f\ (x{:}\theta)\ \mathtt{:=}\ M : \theta \to \tau}$$

適用 $$\frac{\Gamma \vdash M : \theta \to \tau \quad \Gamma \vdash N : \theta}{\Gamma \vdash M\ N : \tau}$$

直積 $$\frac{\Gamma \vdash M : \tau \quad \Gamma \vdash N : \theta}{\Gamma \vdash (M, N) : \tau \times \theta}$$

射影 $\Gamma \vdash \mathsf{fst} : \tau \times \theta \to \tau$ $\Gamma \vdash \mathsf{snd} : \tau \times \theta \to \theta$

実際のプログラムの型付けには単一化に基いた型推論が使われている.

型付けの例

$$\dfrac{\dfrac{\dfrac{\Gamma, x : \mathsf{nat} \vdash \mathsf{S} : \mathsf{nat} \to \mathsf{nat} \quad \Gamma, x : \mathsf{nat} \vdash x : \mathsf{nat}}{\Gamma, x : \mathsf{nat} \vdash \mathsf{S}\ x : \mathsf{nat}}\ \text{適用}}{\Gamma \vdash \mathtt{fun}\ x{:}\mathsf{nat} \Rightarrow \mathsf{S}\ x : \mathsf{nat} \to \mathsf{nat}}\ \text{抽象} \qquad \Gamma \vdash \mathsf{O} : \mathsf{nat}}{\Gamma \vdash (\mathtt{fun}\ x{:}\mathsf{nat} \Rightarrow \mathsf{S}\ x)\ \mathsf{O} : \mathsf{nat}}\ \text{適用}$$

2 命題と型の対応

カリー・ハワード同型により，1 回目に見た命題論理と型理論 (型付 λ 計算) が対応している.

公理 $\Delta \vdash P$ $(P$ は Δに含まれる$)$

$\to$ 導入 $$\frac{\Delta, P \vdash Q}{\Delta \vdash P \to Q}$$

$\to$ 除去 $$\frac{\Delta \vdash P \quad \Delta \vdash P \to Q}{\Delta \vdash Q}$$

矛盾 $$\frac{\Delta \vdash \mathit{False}}{\Delta \vdash P}$$

$\wedge$ 導入 $$\frac{\Delta \vdash P \quad \Delta \vdash Q}{\Delta \vdash P \wedge Q}$$

$\wedge$ 除去 $$\frac{\Delta \vdash P \wedge Q}{\Delta \vdash P} \qquad \frac{\Delta \vdash P \wedge Q}{\Delta \vdash Q}$$

$\vee$ 導入 $$\frac{\Delta \vdash P}{\Delta \vdash P \vee Q} \qquad \frac{\Delta \vdash Q}{\Delta \vdash P \vee Q}$$

$\vee$ 除去 $$\frac{\Delta \vdash P \vee Q \quad \Delta, P \vdash R \quad \Delta, Q \vdash R}{\Delta \vdash R}$$

具体的には，以下のような対応が見られる.

命題 (論理式)	型
証明 (導出)	プログラム (証明項)
仮定 Δ	型環境 Γ
$\to$	$\to$
$\wedge$	$\times$
$\vee$	$+$

導出規則と型付け規則も基本的には 1 対 1 で対応している．それぞれの体系を少し修正すると以下の定理がなりたつ．

定理 1 (Curry-Howard 同型) ある同型 $\langle\_\rangle$: 命題 $\to$ 型 が存在し，任意の Δ と P について，導出 Π より $\Delta \vdash P$ が示せるならば，Π からプログラム M が作れ，$\langle\Delta\rangle \vdash M : \langle P\rangle$．また，任意の Γ, M, τ について型理論で $\Gamma \vdash M : \tau$ が導出できれば，命題論理において $\langle\Gamma\rangle^{-1} \vdash \langle\tau\rangle^{-1}$ が導出できる．

ただし，上の**不動点**の規則は気を付けないと矛盾を生んでしまう．具体的には，$\theta = \mathit{True}$ と $\tau = \mathit{False}$ にすると，以下の導出が可能になる．

$$\frac{\Gamma, f : \mathit{True} \to \mathit{False}, x : \mathit{True} \vdash f\ x : \mathit{False}}{\Gamma \vdash \mathtt{fix}\ f\ (x{:}\theta)\ \mathtt{:=}\ f\ x : \mathit{True} \to \mathit{False}}$$

それを考慮した Coq の本当の**不動点**の規則はさらに f が x より小さな引数に適用されることを求めているので，この矛盾が実際には起きない．本当の規則が複雑なのでここには書かない．

この関係は述語論理や高階述語論理に拡張できる．ただし，論理演算子がそのまま型として使われるので，重要なのは証明項の存在である．特に，$\exists x : T, P(x)$ の証明項は型 T の元 t と $P(t)$ の証明項の対になる．

Coq の場合では，Inductive の表現力が高く,「$\to$」以外の各論理演算子が Inductive として定義され，除去規則は `match` 構文を使った証明項に翻訳される．$\wedge$，$\vee$，`False` や $\exists$ の除去がその例．

```
Require Import ssreflect.
Section CurryHoward.
Variables P Q R : Prop.

Goal P -> (P -> Q) -> Q.
  move=> p pq. apply pq. assumption.
  Show Proof.
(fun (p : P) (pq : P -> Q) => pq p)
Abort.

Goal P /\ Q -> Q /\ P.
  move=> pq; case pq. split; assumption.
  Show Proof.
(fun pq : P /\ Q => match pq with
                   | conj H H0 => conj H0 H
                   end)
Abort.

Goal P \/ Q -> (P -> R) -> (Q -> R) -> R.
  move=> pq pr qr. case pq; assumption.
  Show Proof.
(fun (pq : P \/ Q) (pr : P -> R) (qr : Q -> R) =>
 match pq with
 | or_introl x => pr x
```

```
 | or_intror x => qr x
 end)
Abort.

Goal False -> P.
  move=> f; case f.
  Show Proof.
(fun f : False => match f return P with end)
Abort.

Variables (A : Set) (drinks : A -> Prop).
Variable me : A.                                    (* 自分がそこにいる *)
Hypothesis EM : forall P, P \/ ~ P.                          (* 排中律 *)

(* 飲み屋には必ずそいつが飲めば全員は飲むというやつがいる *)
Theorem drinkers_paradox : exists x, drinks x -> forall y, drinks y.
Proof.
  case (EM (exists x, ~drinks x)) => H.
    case H => x Hx.                (* 飲まないやつを見付けたので，そいつにする *)
    exists x => Hnx; case Hx; assumption.
  exists me => _ y.                           (* 皆飲んでいるので，誰でもいい *)
  case (EM (drinks y)) => Hy.
    assumption.
  case H; exists y; assumption.
  Show Proof.
match EM (exists x : A, ~ drinks x) with
| or_introl (ex_intro _ x Hx) =>
    ex_intro (fun x0 : A => drinks x0 -> forall y : A, drinks y) x
      (fun Hnx : drinks x =>                                    (* 矛盾 *)
       match Hx Hnx return (forall y : A, drinks y) with end)
| or_intror H =>
    ex_intro (fun x : A => drinks x -> forall y : A, drinks y) me
      (fun (_ : drinks me) (y : A) =>
       match EM (drinks y) with
       | or_introl Hy => Hy
       | or_intror Hy =>                                        (* 矛盾 *)
           match H (ex_intro (fun x : A => ~ drinks x) y Hy) return (drinks y)
           with end
       end)
end
Qed.
End CurryHoward.
```

3 eqType と seq

Mathcomp ではある型 T における等価性が判定可能ということをソートみたいに表現できる．2 つの T の値が等価かどうかを計算する T -> T -> bool の関数があることを T : eqType と宣言できる．そうすれば，a と b の型が T なら，a == b がその関数を呼ぶ．

実は eqType は Type や Set のようなソートではなく，その関数とその正しさの証明を含むデータ構造である．

```
Record eqType : Type := Pack
  { T : Type; op : T -> T -> bool; _ : forall x y, reflect (x = y) (op x y) }
```

ただし，たとえば nat : eqType と書いても，本当は nat : Set なのでそのまま eqType の値ではないので，あらかじめ宣言された nat_eqType が自動的に使われ，区別する必要があまりない．

MathComp のリストの関数はモジュール seq で定義されている．因に，その中で Notation seq := list と宣言されているので，MathComp を使うと全てのリストの型が seq と表示される．

リストに関する関数は等価性の判定が必要なもの以外は普通に一般的な Type に対して定義されているが，各関数の性質を表した述語などがほとんどの場合 eqType を仮定している．特に，ある元 t : T : eqType がリスト l : seq T に含まれることを t \in l と書く．

また，ビュー機構を使い，bool 上の述語と量化子を使った表現を結びつけている．たとえば，全称は，

```
Variables (T : eqType) (a : T -> bool).
Fixpoint all s := if s is x :: s' then a x && all s' else true.
Lemma allP s : reflect (forall x : T, x \in s -> a x) (all a s).
```

4 整列の証明, 再び

```
Section sort.
Variable A : eqType.
Variable le : A -> A -> bool.
Variable le_trans: forall x y z, le x y -> le y z -> le x z.
Variable le_total: forall x y, ~~ le x y -> le y x.

Fixpoint insert a l := match l with
  | nil     => (a :: nil)
  | b :: l' => if le a b then a :: l else b :: insert a l'
  end.

Fixpoint isort l :=
  if l is a :: l' then insert a (isort l') else nil.

Fixpoint sorted l :=                    (* all を使って bool 上の述語を定義する *)
  if l is a :: l' then all (le a) l' && sorted l' else true.

Lemma le_seq_insert a b l :
  le a b -> all (le a) l -> all (le a) (insert b l).
Proof.
  elim: l => /= [-> // | c l IH].
  move=> leab /andP [leac leal].
  case: ifPn => lebc /=.
  - by rewrite leab leac.
  - by rewrite leac IH.
Qed.

Lemma le_seq_trans a b l :
  le a b -> all (le b) l -> all (le a) l.
Proof.
  move=> leab /allP lebl.
  apply/allP => x Hx.
  by apply/le_trans/lebl.
Qed.

Theorem insert_ok a l : sorted l -> sorted (insert a l). Admitted.
Theorem isort_ok l : sorted (isort l). Admitted.
```

```
(* perm_eq が seq で定義されているが補題だけを使う *)
Theorem insert_perm l a : perm_eq (a :: l) (insert a l).
Proof.
  elim: l => //= b l pal.
  case: ifPn => //= leab.
  by rewrite (perm_catCA [:: a] [:: b]) perm_cons.
Qed.

perm_eq_trans : forall (T : eqType), transitive (seq T) perm_eq
Theorem isort_perm l : perm_eq l (isort l). Admitted.
End sort.

Check isort.
Definition isortn : seq nat -> seq nat := isort _ leq.
Definition sortedn := sorted _ leq.
Lemma leq_total a b : ~~ (a <= b) -> b <= a. Admitted.

Theorem isortn_ok l : sortedn (isortn l) && perm_eq l (isortn l). Admitted.

Require Import Extraction.
Extraction "isort.ml" isortn.                    (* コードが分かりにくい *)
```

OCamlで正しく動く.

```
% ocamlc -c isort.mli isort.ml
% ocaml
        OCaml version 4.07.1
# #load"isort.cmo";;
# open Isort;;
# isortn (Cons (S O, Cons (O, Cons (S (S O), Nil))));;
- : Isort.nat Isort.list = Cons (O, Cons (S O, Cons (S (S O), Nil)))
```

練習問題 4.1 Admitted を Proof に変え，証明を完成させよ.

5 偶奇，mathcomp式

Propでの述語を使ったとき，述語の導出に対する帰納法の重要性を見た．しかし，mathcomp式に述語をboolで定義すると，通常の帰納法で足りることが多い.

oddはssrnatでboolの述語とした定義されている．evenをその否定として定義すると様々なことが簡単になる.

```
Section even_odd.
Notation even n := (~~ odd n).            (* 単なる表記なので，展開が要らない *)

Theorem even_double n : even (n + n).
Proof. elim: n => // n. by rewrite addnS /= negbK. Qed.

(* 等式を使ってnに対する通常の帰納法を可能にする *)
Theorem even_plus m n : even m -> even n = even (m + n).
Proof.
  elim: n => /= [|n IH] Hm.
  - by rewrite addn0.
  - by rewrite addnS IH.
```

```
Qed.

Theorem one_not_even : ~~ even 1.
Proof. reflexivity. Qed.

Theorem even_not_odd n : even n -> ~~ odd n.
Proof. done. Qed.

Theorem even_odd n : even n -> odd n.+1. Admitted.
Theorem odd_even n : odd n -> even n.+1. Admitted.
Theorem even_or_odd n : even n || odd n. Admitted.
Theorem odd_odd_even m n : odd m -> odd n = even (m+n). Admitted.
End even_odd.
```

練習問題 5.1 Admitted を Proof に変え，証明を完成させよ．