

λ計算の評価

Jacques Garrigue, 2022年1月19日

1 λ計算

1.1 De Bruijn 添字

λ項 $M ::= n \mid \lambda M \mid MM$

β簡約 $(\lambda M)N \rightarrow [N/0]M$

代入

$$[N/k]n = \begin{cases} N & n = k \\ n & n < k \\ n - 1 & n > k \end{cases} \quad \begin{aligned} [N/k]\lambda M &= \lambda[\uparrow_0 N/k + 1]M \\ [N/k](M_1 M_2) &= [N/k]M_1 [N/k]M_2 \end{aligned}$$

シフト

$$\uparrow_k n = \begin{cases} n + 1 & k \leq n \\ n & k > n \end{cases} \quad \begin{aligned} \uparrow_k \lambda M &= \lambda \uparrow_{k+1} M \\ \uparrow_k (M_1 M_2) &= \uparrow_k M_1 \uparrow_k M_2 \end{aligned}$$

簡約関係

$$\frac{M \rightarrow M'}{M N \rightarrow M' N} \quad \frac{N \rightarrow N'}{M N \rightarrow M N'} \quad \frac{M \rightarrow M'}{\lambda M \rightarrow \lambda M'} \quad M \Rightarrow M \quad \frac{M \rightarrow M' \quad M' \Rightarrow N}{M \Rightarrow N}$$

1.2 コード

```
From mathcomp Require Import all_ssreflect.
```

```
(* Lambda calculator *)
```

```
Module Lambda.
```

```
Inductive expr : Set :=
```

```
  | Var of nat
```

```
    (* De Bruijn 添字 : 束縛する λ までの入れ子 *)
```

```
  | Abs of expr
```

```
    (* 抽象 : λ[x].M *)
```

```
  | App of expr & expr.
```

```
    (* 適用 *)
```

```
Coercion Var : nat -> expr.
```

```
(* 自然数を変数として直接に使える *)
```

```
Definition church n := Abs (Abs (iter n (App 1) 0)).
```

```
(* λf.λx.(f^n x) *)
```

```
Eval compute in church 3.
```

```
= Abs (Abs (App 1 (App 1 (App 1 0))))
```

```
Definition chadd := Abs (Abs (Abs (Abs (App (App 3 1) (App (App 2 1) 0))))).
```

```
(* λm.λn.λf.λx.(m f (n f x)) *)
```

```
Fixpoint shift k (e : expr) :=
```

```
(* 自由変数をずらす *)
```

```
  match e with
```

```
  | Var n    => if k <= n then Var n.+1 else Var n
```

```
  | Abs e1   => Abs (shift k.+1 e1)
```

```
  | App e1 e2 => App (shift k e1) (shift k e2)
```

```

end.

Eval compute in shift 1 (App 1 0). = App 2 0

Fixpoint open_rec k u (e : expr) := (* 自由変数の代入 *)
  match e with
  | Var n      => if k == n then u else if leq k n then Var n.-1 else e
  | Abs e1     => Abs (open_rec k.+1 (shift 0 u) e1)
  | App e1 e2 => App (open_rec k u e1) (open_rec k u e2)
  end.

Eval compute in open_rec 0 (Abs (App 1 0)) (Abs (App 1 0)).
= Abs (App (Abs (App 2 0)) 0)

Inductive reduces : expr -> expr -> Prop := (* 簡約の導出規則 *)
  | Rbeta : forall e1 e2, reduces (App (Abs e1) e2) (open_rec 0 e2 e1)
  | Rapp1 : forall e1 e2 e1',
    reduces e1 e1' -> reduces (App e1 e2) (App e1' e2)
  | Rapp2 : forall e1 e2 e2',
    reduces e2 e2' -> reduces (App e1 e2) (App e1 e2')
  | Rabs : forall e1 e1',
    reduces e1 e1' -> reduces (Abs e1) (Abs e1').

(* 簡約関係は reduces の反射推移閉包 *)
Inductive RT_closure {A} (R : A -> A -> Prop) : A -> A -> Prop :=
  | RTbase : forall a, RT_closure R a a
  | RTnext : forall a b c, R a b -> RT_closure R b c -> RT_closure R a c.
Hint Constructors reduces RT_closure : core.

Fixpoint reduce (e : expr) : option expr := (* 1ステップ簡約 *)
  match e with
  | App (Abs e1) e2 => Some (open_rec 0 e2 e1)
  | App e1 e2 =>
    match reduce e1, reduce e2 with
    | Some e1', _ => Some (App e1' e2)
    | None, Some e2' => Some (App e1 e2')
    | None, None => None
    end
  | Abs e1 =>
    if reduce e1 is Some e1' then Some (Abs e1') else None
  | _ => None
  end.

Fixpoint eval (n : nat) e := (* nステップ簡約 *)
  if n is k.+1 then
    if reduce e is Some e' then eval k e' else e
  else e.

Eval compute in eval 6 (App (App chadd (church 3)) (church 2)).
= Abs (Abs (App 1 (App 1 (App 1 (App 1 (App 1 0)))))) : expr

Lemma reduce_ok e e' : (* 1-step 簡約の健全性 *)
  reduce e = Some e' -> reduces e e'.
Proof.

```

```

move: e'; induction e => // = e'.
  case He: (reduce e) => [e1|] // [] <-.
  admit.
destruct e1 => //.
- rewrite /=. admit.
- case => <-.
  by constructor.
- case He1: (reduce (App _ _)) => [e1'|].
  case => <-.
Admitted.

```

(* n -step 簡約の健全性 *)

Theorem eval_ok n e e' : eval n e = e' -> RT_closure reduces e e'. Admitted.

```

Fixpoint closed_expr n e :=
  match e with
  | Var k => k < n
  | App e1 e2 => closed_expr n e1 && closed_expr n e2
  | Abs e1 => closed_expr n.+1 e1
  end.

```

(* 変数が n 個以下の項 *)

Lemma shift_closed n e : closed_expr n e -> shift n e = e. Admitted.

Lemma open_rec_closed n u e : (* $n+1$ 個目の変数を代入しても変わらない *)
 closed_expr n e -> open_rec n u e = e.

Proof.

```

move: n u.
induction e => // = k u Hc.
- case: ifP => Hk1.
  by rewrite (eqP Hk1) lttn in Hc.
  by rewrite leqNgt Hc.
Admitted.

```

Lemma closed_iter_app n k e1 e2 :
 closed_expr k e1 -> closed_expr k e2 -> closed_expr k (iter n (App e1) e2).
 Admitted.

Lemma closed_church n : closed_expr 0 (church n). Admitted.

Lemma closed_expr_S n e : closed_expr n e -> closed_expr n.+1 e. Admitted.

Hint Resolve closed_iter_app closed_church closed_expr_S.

```

Lemma open_iter_app k n u e1 e2 :
  open_rec k u (iter n (App e1) e2) =
  iter n (App (open_rec k u e1)) (open_rec k u e2).
Admitted.

```

Lemma reduces_iter n e1 e2 e2' :
 reduces e2 e2' -> reduces (iter n (App e1) e2) (iter n (App e1) e2').
 Admitted.

Theorem chadd_ok' m n : (* Church 数の足し算が正しい *)
 RT_closure reduces (App (App chadd (church m)) (church n)) (church (m+n)).

Proof.

```

eapply RTnext; repeat constructor.
rewrite /= !shift_closed; auto.
Admitted.

```

```

Lemma eval_add m n e : eval (m+n) e = eval m (eval n e). Admitted.
Lemma reduce_iter_app n (k : nat) x :
  reduce (iter n (App k) x) =
    if reduce x is Some x' then Some (iter n (App k) x') else None.
Admitted.

```

```

Theorem chadd_ok m n :
  exists h, exists h',
    eval h (App (App chadd (church m)) (church n)) = eval h' (church (m+n)).
Proof.

```

```

elim: m n => [|m IHm] n.
  rewrite add0n.
  exists 6; exists 0 => /=.
  rewrite !shift_closed; auto.
  by rewrite !open_iter_app /=.
move: {IHm}(IHm n.+1) => [h [h' IHm]].
exists (6+h); exists (6+h').
rewrite (addSnnS m) -(addn1 m).
move: (f_equal (eval 6) IHm).
rewrite -!eval_add => <- /=.
rewrite !shift_closed /=: auto.
rewrite !open_iter_app /=.
do! rewrite !reduce_iter_app /=:
  by rewrite iter_add.

```

Qed.

End Lambda.

練習問題 1.1 上記の証明の admit と Admitted をなくせ.