

λ計算の評価

Jacques Garrigue, 2018 年 1 月 17 日

1 前回の課題

Section Iterator2.

(* ... *)

Inductive code : Set :=

```
| Cimm of Z
| Cget of nat
| Cadd
| Cmin
| Cmul
| Cset of nat
| Crep of nat (* ループの長さ (Cnext までの命令数) *)
| Cnext.
```

Fixpoint eval_code (stack : list Z) (l : list code) :=

```
match l with
| nil => stack
| c :: l' =>
  let stack' :=
    match c, stack with
    | Cimm x, _ => x :: stack
    | Cget n, _ => nth 0 stack n :: stack
    | Cadd, x :: y :: st => x+y :: st
    | Cmin, x :: st => 0-x :: st
    | Cmul, x :: y :: st => x*y :: st
    | Cset n, x :: st => set_nth 0%Z st n x
    | Crep _, Zpos n :: st => (* seq の iter を使う *)
      iter (Pos.to_nat n) (fun st => eval_code st l') st
    | Crep _, _ :: st => st
    | Cnext, _ => stack
    | _, _ => nil
    end%Z
  in
  match c with
  | Crep n => eval_drop n stack' l'
  | Cnext => stack'
  | _ => eval_code stack' l'
  end
end
```

```
with eval_drop n st (l : list code) := (* 相互再帰 *)
  match l, n with
  | _ :: l', 0 => eval_code st l'
  | _ :: l', S n' => eval_drop n' st l'
  | [::], _ => st
  end.
```

End Iterator2.

2 λ計算

```
From mathcomp Require Import all_ssreflect.

(* Lambda calculator *)

Module Lambda.

Inductive expr : Set :=
  | Var of nat
  | Abs of expr
  | App of expr & expr.
  (* De Bruijn 添字 *)

Fixpoint shift k (e : expr) :=
  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.
  (* 自由変数をずらす *)

Fixpoint open_rec k u (e : expr) :=
  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.
  (* 自由変数の代入 *)

Fixpoint reduce (e : expr) : option expr :=
  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.
  (* 1 ステップ簡約 *)

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

Coercion Var : nat >-> expr.

Fixpoint church (n : nat) := Abs (Abs (iter n (App 1) 0)). (* λf.λx.(f^n x) *)
Eval compute in church 3.

Definition chadd := Abs (Abs (Abs (Abs (App (App 3 1) (App (App 2 1) 0))))).
  (* λm.λn.λf.λx.(m f (n f x)) *)
```

Eval compute in eval 6 (App (App chadd (church 3)) (church 2)).

Lemma eval_add m n e : eval (m+n) e = eval m (eval n e). Admitted.

```
Inductive reduces : expr -> expr -> Prop :=
| 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').
```

Hint Constructors reduces.

Lemma reduce_ok e e' : reduce e = Some e' -> reduces e e'.

Proof.

```
move: e'.
induction e => // = e'.
  case_eq (reduce e) => [e1|] He // [] <-.
  admit.
destruct e1 => //.
+ admit.
+ move => [] <-.
  by constructor.
+ case_eq (reduce (App e1_1 e1_2)) => [e1'|] He1.
  move => [] <-.
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.
```

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

Lemma open_rec_closed n u e : 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 shift_closed n e : closed_expr n e -> shift n e = e. 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.

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 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 /=.
repeat rewrite !reduce_iter_app /=.
by rewrite iter_add.

Qed.

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.

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 :
 RT_closure reduces (App (App chadd (church m)) (church n)) (church (m+n)).

Proof.

eapply RTnext; repeat constructor.
simpl.
rewrite !shift_closed; auto.

Admitted.

End Lambda.