λ計算

Jacques Garrigue, 2016 年 1 月 27 日

```
Module Lambda.
(* ... *)

Fixpoint repeat_n A : Type (n : nat) (f : A -> A) (x : A) :=
  if n is S n' then f (repeat_n n' f x) else x.

Coercion Var : nat >-> expr.

Definition church (n : nat) := Abs (Abs (repeat_n n (App 1) 0)).
Definition chadd := Abs (Abs (Abs (Abs (App (App 3 1) (App (App 2 1) 0))))).

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

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').

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.
      simpl in He1; rewrite He1 => [] [] <-.
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 open_rec_inv 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.
```

```
      rewrite eqnE in Hk1.
      by rewrite (eqnP Hk1) ltnn in Hc.
    by rewrite leqNgt Hc.
Admitted.

Parameter shift_inv n e : closed_expr n e -> shift n e = e.

Lemma closed_repeat_app n k e1 e2 :
  closed_expr k e1 -> closed_expr k e2 ->
  closed_expr k (repeat_n n (App e1) e2).
Admitted.

Parameter closed_church n : closed_expr 0 (church n).
Parameter closed_expr_S n e : closed_expr n e -> closed_expr n.+1 e.

Hint Resolve closed_repeat_app closed_church closed_expr_S.

Parameter open_repeat_app k n u e1 e2 :
  open_rec k u (repeat_n n (App e1) e2) =
  repeat_n n (App (open_rec k u e1)) (open_rec k u e2).
Parameter eval_add m n e : eval (m+n) e = eval m (eval n e).
Parameter repeat_n_add A m n (f : A -> A) x :
  repeat_n (m+n) f x = repeat_n m f (repeat_n n f x).
Parameter reduce_repeat_app n (k : nat) x :
  reduce (repeat_n n (App k) x) =
    if reduce x is Some x' then Some (repeat_n n (App k) x') else None.

Theorem chadd_ok m n :
  exists h, exists h',
    eval h (Abs (Abs (App (App (App (App chadd (church m)) (church n)) 0) 1)))
    = eval h' (Abs (Abs (App (App (church (m+n)) 0) 1))).
Proof.
  elim: m n => [|m IHm] n.
    rewrite add0n.
    move: (church n) (closed_church n) => cn Hcn.
    exists 6; exists 0 => /=.
    by rewrite !open_rec_inv !shift_inv; auto.
  move: IHm(IHm n.+1) => [h [h' IHm]].
  exists (8+h); exists (8+h').
  rewrite (addSnnS m) -(addn1 m).
  move: (f_equal (eval 8) IHm).
  rewrite -!eval_add => <-.
  simpl.
  rewrite !shift_inv /=; auto.
  rewrite !open_repeat_app /=.
  rewrite !reduce_repeat_app /=.
  rewrite !reduce_repeat_app /=.
  by rewrite repeat_n_add.
Qed.

End Lambda.
```