コンパイラ

Jacques Garrigue, 2016 年 1 月 20 日

1 前回の課題

```
Section Iterator.
(* ... *)

Lemma repeat_n_eq A:Type n f1 f2  (st : A) :
  (forall st, f1 st = f2 st) -> repeat_n n f1 st = repeat_n n f2 st.
Proof.
  elim: n st => //= n IHn st Heq.
  by rewrite Heq IHn.
Qed.

Lemma eval_code_cat k stack (l1 l2 : seq code) :
  balanced l1 ->
  eval_code k stack (l1 ++ l2) =
  eval_code k (eval_code k stack l1) l2.
Proof.
  move=> Hbal.
  elim: Hbal k stack l2.
      by destruct c, k.
    move=> l1l1 l2 Hbal1 IHl1 Hbal2 IHl2 k st l'.
    by rewrite -catA !IHl1 IHl2.
  move=> l1l1 Hbal IHl1 k st l'.
  destruct k, st => /=; rewrite -catA !IHl1 //.
  destruct z => //=.
  rewrite subnn.
  f_equal.
  f_equal.
  apply repeat_n_eq => st'.
  by rewrite !IHl1.
Qed.

End Iterator.
```

2 λ計算

```
Require Import ZArith.
From Ssreflect Require Import ssreflect ssrnat ssrbool seq.

(* Lambda calculator *)

Module Lambda.

Inductive expr : Set :=
  | Var of nat
  | Abs of expr
  | App of expr & expr.
```

```
Fixpoint shift k (e : expr) :=
  match e with
  | Var n     => if leq 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.

Fixpoint open_rec k u (e : expr) :=
  match e with
  | Var n     => if eqn 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.

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.

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.

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

Coercion Var : nat >-> expr.

Fixpoint church (n : nat) :=
  Abs (Abs (repeat_n n (App 1) 0)).

Eval compute in church 3.

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

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

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_shift h k e :
  closed_expr h e -> closed_expr h.+1 (shift k e).
Proof.
  move: h k.
  induction e => //= h k Hc.
  + admit.
  + admit.
  + move/andP: Hc => [Hc1 Hc2].
Admitted.

Lemma closed_expr_open n e1 e2 k e' :
  closed_expr n.+1 e1 -> k <= n -> closed_expr n e2 ->
  open_rec k e2 e1 = e' -> closed_expr n e'.
Proof.
  move: n k e' e2.
  induction e1 => // h k e' e2 He1 Hk He2 <- //=.
  + case: ifP => Hk1 //.
    rewrite /= in He1.
    admit.
  + apply (IHe1 _ (k.+1) _ (shift 0 e2)) => //.
    by apply closed_expr_shift.
  + move: He1 => /= /andP [He11 He12].
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').

Lemma reduce_ok e e' : reduce e = Some e' -> reduces e e'.
Proof.
  move: e'.
  induction e => //= e'.
    admit.
  destruct e1 => //=.
  + case_eq (reduce e2) => [e2'|] He2 // [] <- /=.
    constructor.
    by apply IHe2.
  + move => [] <-.
    by constructor.
  + case_eq (reduce (App e1_1 e1_2)) => [e1'|] He1.
      simpl in He1; rewrite He1 => [] [] <-.
      constructor.
      by apply IHe1.
Admitted.

Lemma closed_inv n e e' :
```

```
  closed_expr n e -> reduce e = Some e' -> closed_expr n e'.
Proof.
  move=> Hc Hr.
  apply reduce_ok in Hr.
  move: n Hc.
Admitted.

Lemma open_rec_inv n u e : closed_expr n e -> open_rec n u e = e.
Admitted.

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

Theorem chadd_ok m n f x :
  closed_expr 0 f ->
  exists h, exists h',
    eval h (App (App (App (App chadd (church m)) (church n)) f) x)
    = eval h' (App (App (church (m+n)) f) x).
Admitted.

End Lambda.
```