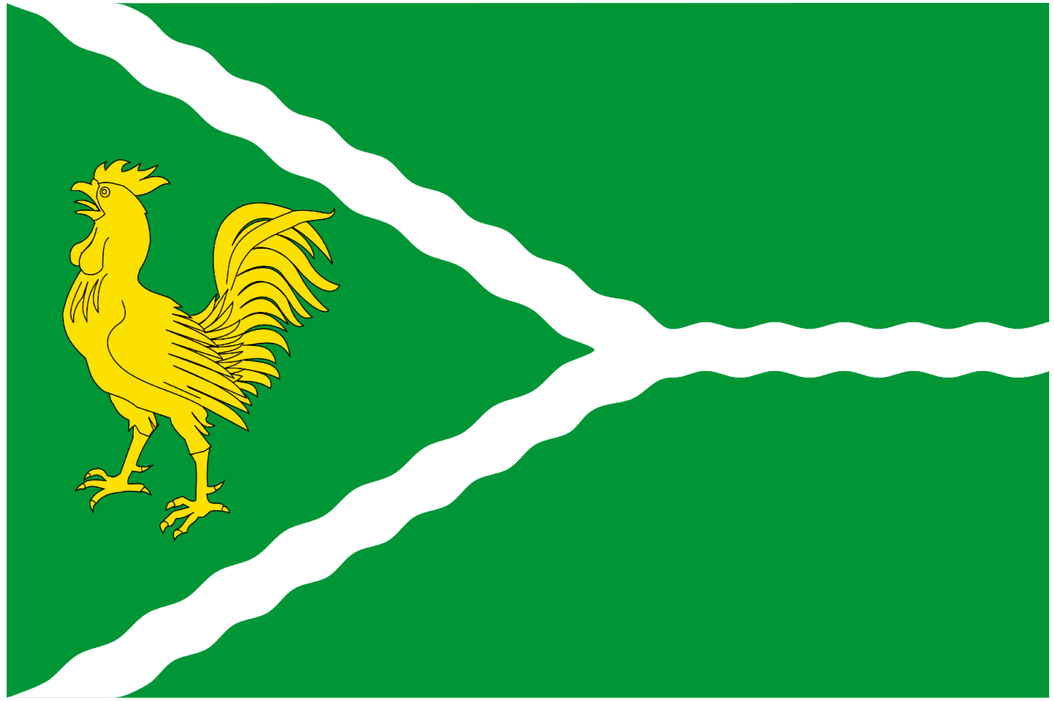
Peano Arithmetic in Untyped Lambda Calculus



Flag of Ripoll, Catalonia

Xavier Ripoll Echeveste

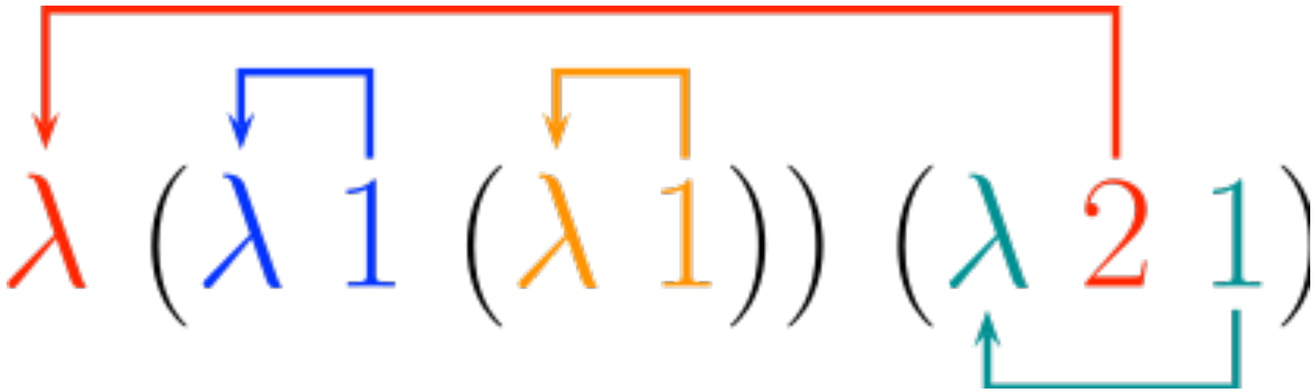
Overview

- Mr. Church
- Mr. De Bruijn

A church in Ripoll, Catalonia

A brown

- a variable, x, is itself a valid lambda term
- if t is a lambda term, and x is a variable, then $(\lambda x.\, t)$ is a lambda term (called an **abstraction**);
- if t and s are lambda terms, then (ts) is a lambda term (called an **application**).



Original Goals

- **Minimal goal**: formalize lambda calculus and encode Peano arithmetic
- **Realistic goal**: minimal goal + construct integers and rationals
- **Stretch goal**: realistic goal + construct computable real numbers

Revised Goals

- **Minimal goal**: ~~open Coq without crying~~ formalize lambda calculus
- **Realistic goal**: formalize lambda calculus *correctly* (see following slides)
- **Stretch goal**: formalize lambda calculus and encode Peano arithmetic

Sketch

1. Formalize a lambda term `term` as an inductive
2. Implement beta reduction
3. Define `encode`, `succ`, `plus`, `mult`, etc.
   and prove Peano's axioms.

1. Lambda Terms

```
Inductive term :=
| Var: nat -> term
| Abs: term -> term
| App: term -> term -> term
.
```

```
Notation "# n" := (Var n) (at level 10).
Notation "\ p" := (Abs p) (at level 20).
Notation "$ p q" := (App p q) (at level 0
```

2. Beta Reduction

Too imperative

- Church, Rosser, and me

```
Definition beta_equiv (p p': term) :=
  apply 100 leftmost_outermost_step p = apply 100 leftmost_outermost_step p.
```

2. Beta Reduction

Too typed

- Beta reduction requires finding redexes.
- This is *the* correct way to index subtrees:

```
Fixpoint subterm (p: term) (i: index p): term :=
  match i with
  | Root p => p
  | SubAbs q j => subterm q j
  | SubAppLeft q r j => subterm q j
  | SubAppRight q r j => subterm q j
  end
.
```

```
Inductive index: term -> Type :=
| Root: forall t, index t
| SubAbs: forall t, index t -> index (\t)
| SubAppLeft: forall t t', index t -> index ($ t t')
| SubAppRight: forall t t', index t -> index ($ t' t)
.
```

- … but it doesn’t always work as we want it to (dependent matching).

2. Beta Reduction

Too exhaustive

```
(* List of all possible 1-step beta reducts of p *)
Fixpoint beta_reducts (p: term): list term :=
  let others := (match p with
  | #n => []
  | \q => map (fun (t: term) => \ t) (beta_reducts q)
  | $ q r => map (fun (t: term) => $ t r) (beta_reducts q) ++ map (fun (t: term) => $ q t) (beta_reducts r)
  end) in
    if term_eq_dec (beta p) p then others else beta p :: others
.

(* p ->_\beta p' *)
Definition beta_step (p p': term): Prop :=
  In p' (beta_reducts p)
.
```

2. Beta Reduction

Just enough?

- The literal textbook definition.
- `beta` returns `options` now.

```
Definition beta_step (p p': term): Prop :=
  exists (rt: path), beta p rt = Some p'.
```

```
Definition beta_equiv: term -> term -> Prop := closure beta_step.

Notation "p == q" := (beta_equiv p q) (at level 70, no associativity).

Lemma BetaRefl: forall (p: term), p == p.
Proof. apply CRefl. Qed.

Lemma BetaSym: forall (p q: term), p == q -> q == p.
Proof. apply CSym. Qed.

Lemma BetaTrans: forall (p q r: term), p == q -> q == r -> p == r.
Proof. apply CTrans. Qed.

Add Parametric Relation: term beta_equiv
  reflexivity proved by BetaRefl
  symmetry proved by BetaSym
  transitivity proved by BetaTrans
  as beta_eq.
```

```
  end
| $ _ _, _ => None (* Invalid path *)
```

"This should never happen" —Famous last words

2.5. A Tactical Detour

```
Ltac reduct rt := match goal with
| |- (?a == ?b) =>
  match eval compute in (beta a rt) with
  | Some ?a_reduct => transitivity a_reduct; [ apply CStep; exists rt; reflexivity | ]
  | None => fail
  end
end.

Goal ($ \ #0 #100) == #100.
  reduct PRoot.
  reflexivity.
Qed.
```

```
Definition COMMON_REDUCT_MAX_DEPTH := 10.
Ltac common_reduct := match goal with
  | |- (?a == ?b) =>
    unfold "==";
    match eval compute in (common_reducts (acc_step a COMMON_REDUCT_MAX_DEPTH) (acc_step b COMMON_REDUCT_MAX_DEPTH)) with
    | (?k, ?l) => let k' := eval compute in (tl (rev k)) in (common_reduct_aux k');
                  apply CSym;
                  let l' := eval compute in (tl (rev l)) in (common_reduct_aux l');
            apply CRefl
    (* TODO add second matching case | None => ... and make it work *)
    end
end.
```

```
Ltac common_reduct := match goal with
| |- (?a == ?b) =>
  match eval compute in (leftmost_outermost_step a) with
  | Some (?a_reduct, ?a_path) => transitivity a_reduct; [ apply CStep; exists a_path; reflexivity | ]
  | None => idtac
  end;
  symmetry;
  match eval compute in (leftmost_outermost_step b) with
  | Some (?b_reduct, ?b_path) => transitivity b_reduct; [ apply CStep; exists b_path; reflexivity | ]
  | None => idtac
  end;
  symmetry;
  try reflexivity;
  try common_reduct
end.
```

3. Peano Arithmetic

- $\forall x \, (0 \neq S(x))$
- $\forall x, y \, (S(x) = S(y) \Rightarrow x = y)$
- $\forall x \, (x + 0 = x)$
- $\forall x, y \, (x + S(y) = S(x + y))$
- $\forall x \, (x \cdot 0 = 0)$
- $\forall x, y \, (x \cdot S(y) = x \cdot y + x)$

> formalize the calculus and then encode Peano arithmetic
>
> That sounds nice, even easy (I might regret this judgment later on).

Excerpt of letter from myself to Dr. Kappé

```
Fixpoint encode_aux (n: nat): term :=
  match n with
  | 0 => #0
  | S n' => $ (#1) (encode_aux n')
  end
.

(* Church encoding of a given natural *)
Definition encode (n: nat): term :=
  \\(encode_aux n)
.
```

```
Definition succ: term :=
  \\\ $ (#1) ($ ($ (#2) (#1)) (#0))
.
```

3. Peano Arithmetic

- Foundational crisis!
  a.k.a. verify what you can, test what you can’t

```
Fixpoint increase_free_variables (p: term) (increment level: nat): term :=
  match p with
  | #n => if (lt_dec n level) (* if n < level *)
    then #n (* #n was bound, leave it as it is *)
    else #(n + increment) (* #n was free, must increase it *)
  | \q => \(increase_free_variables q increment (S level))
  | $ q r => $ (increase_free_variables q increment level) (increase_free_variables r increment level)
  end
.
```

3. Peano Arithmetic

```
Lemma plus_succ (n m: nat): $ $ plus (encode n) ($ succ (encode m)) == $ succ $ $ plus (encode n) (encode m).
Proof.
  induction m.
  - rewrite zero_neutral.
    unfold plus.
    step ($ ((\ $ $ (encode n) succ (# 0))) $ succ (encode 0)) (PLeft PRoot).
    step (($ $ (encode n) succ ($ succ (encode 0)))) (PRoot).
    induction n.
    + common_reduct.
    + rewrite <- succ_encode.
      unfold succ at 1.
      step ($ $ ((\ (\ $ (# 1) $ $ (encode n) (# 1) (# 0))) succ $ succ (encode 0)) (PLeft (PLeft PRoot)).
      step ($ ((\ $ succ $ $ (encode n) succ (# 0))) $ succ (encode 0)) (PLeft PRoot).
      step (($ succ $ $ (encode n) succ $ succ (encode 0))) PRoot.
      f_equiv.
      apply IHn.
  - rewrite <- succ_encode.
    unfold plus at 1.
    step ($ ((\ $ $ (encode n) succ (# 0))) $ succ $ succ (encode m)) (PLeft PRoot).
    step (($ $ (encode n) succ ($ succ $ succ (encode m)))) (PRoot).
    rewrite IHm.
    clear IHm.
    induction n.
    + unfold encode at 1.
      unfold encode_aux at 1.
      step ($ (\ # 0) $ succ $ succ (encode m)) (PLeft PRoot).
      step ($ succ $ succ (encode m)) (PRoot).
      repeat f_equiv.
      unfold plus, encode at 2.
      symmetry.
      step ($ ((\ $ $ (\ (\ encode_aux 0)) succ (# 0))) (encode m)) (PLeft PRoot).
      step ((($ $ (\ (\ encode_aux 0)) succ (encode m))) ) (PRoot).
      unfold encode_aux.
      step ($ ((\ # 0)) (encode m)) (PLeft PRoot).
      step (encode m) (PRoot).
      reflexivity.
    + rewrite <- succ_encode in *.
      unfold succ at 1.
      step ($ $ ((\ (\ $ (# 1) $ $ (encode n) (# 1) (# 0))) succ $ succ $ succ (encode m)) (PLeft (PLeft PRoot)).
      step ($ ((\ $ succ $ $ (encode n) succ (# 0))) $ succ $ succ (encode m)) (PLeft PRoot).
      step (($ succ $ $ (encode n) succ ($ succ $ succ (encode m)))) (PRoot).
      f_equiv.
      transitivity ($ succ $ succ $ $ plus (encode n) (encode m)).
      * apply IHn.
      * f_equiv.
        symmetry.
        unfold plus at 1.
        step ($ ((\ $ $ $ succ (encode n) succ (# 0))) (encode m)) (PLeft PRoot).
        step (($ $ $ succ (encode n) succ (encode m))) PRoot.
        unfold succ at 1.
        step ($ $ ((\ (\ $ (# 1) $ $ ((encode n)) (# 1) (# 0))) succ (encode m)) (PLeft (PLeft PRoot)).
        step ($ ((\ $ succ $ $ (encode n) succ (# 0))) (encode m)) (PLeft PRoot).
        step ($ succ $ $ (encode n) succ (encode m)) PRoot.
        f_equiv.
        symmetry.
        unfold plus.
        step ($ ((\ $ $ (encode n) succ (# 0))) (encode m)) (PLeft PRoot).
        step ($ $ (encode n) succ (encode m)) PRoot.
        reflexivity.
Qed.
```

?

```
Lemma zero_neutral: forall (n: nat), $ $ plus (encode n) (encode 0) == encode n.
Proof.
  intros.
  induction n.
  - common_reduct.
  - rewrite <- succ_encode.
    rewrite <- IHn at 2.
    unfold plus.
    transitivity ($ (\ $ ($ ($ succ (encode n)) succ) (# 0)) (encode 0));
    [ apply CStep;
      exists (PLeft (PRoot));
      simpl;
      repeat f_equiv;
      unfold encode;
      repeat f_equal;
      apply aux' | ].
    transitivity ($ $ $ succ (encode n) succ (encode 0));
    [ apply CStep;
      exists PRoot;
      simpl;
      repeat f_equal;
      unfold encode at 2;
      repeat f_equal;
      apply repl_lemma | ].
    symmetry.
    transitivity ($ succ $ (\ $ $ (encode n) succ (# 0)) (encode 0));
    [ apply CStep;
      exists (PRight (PLeft PRoot));
      simpl;
      repeat f_equal;
      unfold encode;
      repeat f_equal;
      apply aux' | ].
    transitivity ($ succ $ $ (encode n) succ (encode 0));
    [ apply CStep;
      exists (PRight PRoot);
      simpl;
      repeat f_equal;
      unfold encode;
      repeat f_equal;
      apply repl_lemma | ].
    unfold succ.
    transitivity (\\ $ (# 1) $ $ ($ $ (encode n) (\ (\ (\ $ (# 1) $ $ (# 2) (# 1) (# 0)))) (encode 0)) (# 1) (# 0));
    [ apply CStep;
      exists PRoot;
      simpl;
      repeat f_equal;
      unfold encode;
      repeat f_equal;
      apply aux | ].
    symmetry.
    transitivity ($ $ (\ (\ $ (# 1) $ $ (encode n) (# 1) (# 0))) (\ (\ (\ $ (# 1) $ $ (# 2) (# 1) (# 0)))) (encode 0));
    [ apply CStep;
      exists (PLeft (PLeft PRoot));
      simpl;
      repeat f_equal;
      unfold encode;
      repeat f_equal;
      apply aux | ].
    transitivity ($ ((\ $ ((\ (\ (\ $ (# 1) $ $ (# 2) (# 1) (# 0))))) $ $ (encode n) ((\ (\ (\ $ (# 1) $ $ (# 2) (# 1) (# 0))))) (# 0))) (encode 0));
    [ apply CStep;
      exists (PLeft PRoot);
      simpl;
      repeat f_equal;
      unfold encode;
      repeat f_equal;
      apply repl_lemma' | ].
    transitivity (($ (\ (\ (\ $ (# 1) $ $ (# 2) (# 1) (# 0)))) $ $ (encode n) (\ (\ (\ $ (# 1) $ $ (# 2) (# 1) (# 0)))) (encode 0));
```

???

3. Peano Arithmetic

- *There has to be a better way!*
- … probably?

```
Ltac step p rt := transitivity p;
  [ apply CStep;
    exists rt;
    simpl;
    repeat f_equal;
    unfold encode;
    repeat f_equal;
    try apply repl_lemma;
    try apply repl_lemma';
    try apply aux;
    try apply aux';
    try apply aux'' | ].
```

Achievements

- **Minimal goal**: formalize lambda calculus and encode Peano arithmetic ✳️
- **Realistic goal**: minimal goal + construct integers and rationals ❌
- **Stretch goal**: realistic goal + construct computable real numbers ❌

Pros and coqs

✅ Chicken related imagery and nomenclature ❤️🐓

✅ Can’t go wrong (literally)

✅ Tactics

❌ Lacking documentation

❌ Ungoogleable issues

❌ Tactics

Thanks!