# Automation of alignments of HOL-Light types and definitions in Rocq

## Or the journey to prove that the length of a list is indeed the length of a list without too much effort

Antoine Gontard, laboratoire méthodes formelles, Inria

# Some context

*hol2dk*

We can translate HOL-Light proofs to Rocq using hol2dk

But to do so, everything (including definitions) is translated to new objects.

# Some context

*HOL Light*

Types :

- ind, bool, $\rightarrow$

- subtypes: given a predicate $P : A \rightarrow \text{bool}$, creates a type B and axioms stating bijection between B and $\{x \in A, P\ x\}$

# Some context

*HOL Light*

Types :

- ind, bool, →
- subtypes: given a predicate P : A → bool, creates a type B and axioms stating bijection between B and $\{x \in A, P\ x\}$

Terms :

- new_definition ‘x = (...)’: adds a new axiom x_def = ‘x = (...)’
- ε P: if P : A → bool is satisfiable then picks an x satisfying it, otherwise a default value.

# Some context

*HOL Light*

Types :

- ind, bool, →

- subtypes: given a predicate P : A → bool, creates a type B and axioms stating bijection between B and{x ∈ A, P x}

Terms :

- new_definition ‘x = (...)’: adds a new axiom x_def = ‘x = (...)’

- ε P: if P : A → bool is satisfiable then picks an x satisfying it, otherwise a default value.

Classical logic: proof irrelevance, funext, propext, EM

# We would like to prove in Rocq that

The length (in HOL-Light) is equal to The length (in Rocq)

# We would like to prove in Rocq that

| | | |
|---|---|---|
| The length (in HOL-Light) | is equal to | The length (in Rocq) |
| of a list (in HOL-Light) | | of a list (in Rocq) |

# We would like to prove in Rocq that

The length (in HOL-Light)

of a list (in HOL-Light)

which is an element of an inductive type (in HOL-Light)

is equal to

The length (in Rocq)

of a list (in Rocq)

which is an element of an inductive type (in Rocq)

# We would like to prove in Rocq that

The length (in HOL-Light)

of a list (in HOL-Light)

which is an element of an inductive type (in HOL-Light)

(defined using inductive propositions)

is equal to

The length (in Rocq)

of a list (in Rocq)

which is an element of an inductive type (in Rocq)

# Plan

- How to align inductive propositions
- How inductive types are defined in HOL-Light
- How to align inductive types
- How to align total recursive functions

# Plan

- How to align inductive propositions
- How inductive types are defined in HOL-Light
- How to align inductive types
- How to align total recursive functions

- How to align partial (sometimes recursive) functions
- What more can be done

# How to align inductive propositions

Example :

```
Inductive finite (A : Type) : set A -> Prop :=
|finite_set0 : finite {}
|finite_setU1 s a : finite s -> finite (s U {a}).
```

# How to align inductive propositions

Example :

```
Inductive finite (A : Type) : set A -> Prop :=
|finite_set0 : finite {}
|finite_setU1 s a : finite s -> finite (s U {a}).
```

In HOL-Light it is equal to its induction principle :

```
Definition FINITE {A : Type} (s : set A) := forall P,
  (forall s', s' = {} \/ (exists x s'0, s' = s'0 U {x} /\ P s'0) -> P s') -> P s.
```

# How to align inductive propositions

Alignment of finite sets :

```
Lemma FINITE_eq_finite (A:Type') (s:A -> Prop) : FINITE s = finite s.
Proof.
  apply prop_ext; intro h.

  apply h. intros P [i|[x [s' [i j]]]]; rewrite i.
  apply finite_EMPTY.
  apply finite_INSERT. exact j.

  induction h; intros P H; apply H.
  left. reflexivity.
  right. exists a. exists s. split. reflexivity. apply IHh. exact H.
Qed.
```

# How to align inductive propositions

## Alignment of finite sets :

```
Lemma FINITE_eq_finite (A:Type') (s:A -> Prop) : FINITE s = finite s.
Proof.
  apply prop_ext; intro h.

  apply h. intros P [i|[x [s' [i j]]]]; rewrite i.
  apply finite_EMPTY.
  apply finite_INSERT. exact j.

  induction h; intros P H; apply H.
  left. reflexivity.
  right. exists a. exists s. split. reflexivity. apply IHh. exact H.
Qed.
```

```
Lemma FINITE_def (A : Type') : @finite A = @FINITE A.
Proof.
  ind_align.
Qed.
```

# How to align inductive propositions

The tactic:

```
Ltac ind_align :=
  try ext x ; try ext y ; try ext z ; apply prop_ext ; intro H ;
  (* Prove equality by double implication *)
  [ (* Proving [P_r x -> P_h x] *)
    intros P' H' ; induction H ; apply H' ; try breakgoal
  | (* Proving [P_h x -> P_r x] *)
    apply H ; (* introductions ... *) ;
    full_destruct ; (* Destructing H results in one goal per case, and separates the hypotheses *)
    blindrewrite ; (* not much to do, each clause should be proved with a rule,
                      we just try to rewrite [a = f x1 ... xn] if it exists *)
    try now (constructor ; auto) ].
```

# How to align inductive propositions

Use in practice :

```
Inductive prenex : form -> Prop :=
| prenex_qfree : forall f, qfree f -> prenex f
| prenex_FAll : forall f n, prenex f -> prenex (FAll n f)
| prenex_FEx : forall f n, prenex f -> prenex (FEx n f).

Lemma prenex_def : prenex = PRENEX.
Proof. ind_align. Qed.

Inductive universal : form -> Prop :=
| universal_qfree : forall f, qfree f -> universal f
| universal_FAll : forall f n, universal f -> universal (FAll n f).

Lemma universal_def : universal = UNIVERSAL.
Proof. ind_align. Qed.
```

# How inductive types are defined in HOL-Light

Creates one complex inductive type by hand for any type A which has two constructors :

```
Inductive recspace (A : Type) :=
| BOTTOM : recspace A
| CONSTR : N -> A -> (N -> recspace A) -> recspace A.
```

# How inductive types are defined in HOL-Light

Creates one complex inductive type by hand for any type A which has two constructors :

```
Inductive recspace (A : Type) :=
| BOTTOM : recspace A
| CONSTR : N -> A -> (N -> recspace A) -> recspace A.
```

Number of the constructor

Non-recursive arguments

« list » of recursive arguments

# How inductive types are defined in HOL-Light

## How it is used

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A -> list A -> list A.
```

nil = CONSTR 0 dflt FNIL

cons a l = CONSTR 1 a (FCONS l FNIL)

# How to align inductive types

**mk : recspace A → list A**

**dest : list A → recspace A**

dest nil = CONSTR 0 dflt FNIL

dest (cons a l) = CONSTR 1 a (FCONS (dest l) FNIL)

# How to align inductive types

**mk : recspace A → list A**

We want mk to be
the inverse of dest…

**dest : list A → recspace A**

dest nil = CONSTR 0 dflt FNIL

dest (cons a l) = CONSTR 1 a (FCONS (dest l) FNIL)

# How to align inductive types

**mk : recspace A → list A**

**dest : list A → recspace A**

We want mk to be
the inverse of dest…

dest nil = CONSTR 0 dflt FNIL

So we define it as
the inverse of dest :

dest (cons a l) = CONSTR 1 a (FCONS (dest l) FNIL)

mk r = ε (dest x = r)

# How to align inductive types

Two results to prove :

[$\forall x$, mk (dest x) = x]

[$\forall r$, P r $\leftrightarrow$ dest (mk r) = r] where P defines the correct subset of recspace A

Thanks to the definition of mk, they simplify to injectivity and surjectivity [P r $\leftrightarrow$ $\exists x$, r = dest x] of dest.

# How to align inductive types

```
Lemma _dest_list_inj :
  forall {A : Type'} (l l' : list A), _dest_list l = _dest_list l' -> l = l'.
Proof.
  induction l; induction l'; simpl; rewrite (@CONSTR_INJ A); intros [e1 [e2 e3]].
  reflexivity. discriminate. discriminate. rewrite e2. rewrite (@IHl l'). reflexivity.
  rewrite <- (FCONS_0 (_dest_list l) ((fun _ : N => BOTTOM))).
  rewrite <- (FCONS_0 (_dest_list l') ((fun _ : N => BOTTOM))).
  rewrite e3. reflexivity.
Qed.

Lemma axiom_15 : forall {A : Type'} (a : list A), (@_mk_list A (@_dest_list A a)) = a.
Proof.
  intros A l. unfold _mk_list.
  match goal with [|- ε ?x = _] => set (L' := x); set (l' := ε L') end.
  assert (i : exists l', L' l'). exists l. reflexivity.
  generalize (ε_spec i). fold l'. unfold L', _mk_list_pred. apply _dest_list_inj.
Qed.

Definition list_pred {A : Type'} (r : recspace A) :=
  forall list'0 : recspace A -> Prop,
  (forall a' : recspace A,
  a' = CONSTR (NUMERAL N0) (ε (fun _ : A => True)) (fun _ : N => BOTTOM) \/
  (exists (a0 : A) (a1 : recspace A), a' = CONSTR (N.succ (NUMERAL N0)) a0 (FCONS a1 (fun _ : N =>
  -> list'0 r.

Inductive list_ind {A : Type'} : recspace A -> Prop :=
| list_ind0 : list_ind (CONSTR (NUMERAL N0) (ε (fun _ : A => True)) (fun _ : N => BOTTOM))
| list_ind1 a'' l'': list_ind (CONSTR (N.succ (NUMERAL N0)) a'' (FCONS (_dest_list l'') (fun _ : N

Lemma list_eq {A : Type'} : @list_pred A = @list_ind A.
Proof.
  ext r. apply prop_ext.
  intro h. apply h. intros r' H. destruct H. rewrite H. exact list_ind0. destruct H. destruct H. d
  assert (_dest_list nil = @CONSTR A (NUMERAL N0) (@ε A (fun v : A => True)) (fun n : N => @BOTTOM
  reflexivity. rewrite <- H0. exact (list_ind1 x nil).
  assert (_dest_list (cons a'' l'') = @CONSTR A (N.succ (NUMERAL N0)) a'' (@FCONS (recspace A) (@_
  reflexivity. rewrite <- H0. exact (list_ind1 x (a'':: l'')).
```

```
  induction 1; unfold list_pred; intros R h; apply h.
  left; reflexivity.
  right. exists a''. exists (_dest_list l''). split. reflexivity. apply h.
  induction l''. auto. right. exists a. exists (_dest_list l''). split. reflexivity.
  apply h. exact IHl''.
Qed.

Lemma axiom_16' : forall {A : Type'} (r : recspace A), (list_pred r) = ((@_dest_list
Proof.
  intros A r. apply prop_ext.

  intro h. apply (@ε_spec _ (_mk_list_pred r)).
  rewrite list_eq in h. induction h.
  exists nil. reflexivity. exists (cons a'' l''). reflexivity.

  intro e. rewrite <- e. intros P h. apply h. destruct (_mk_list r).
  left. reflexivity. right. exists t. exists (_dest_list l). split.
  reflexivity. apply h. generalize l.
  induction l0. left; reflexivity. right. exists a. exists (_dest_list l0). split.
  reflexivity. apply h. exact IHl0.
Qed.

Lemma axiom_16 : forall {A : Type'} (r : recspace A), ((fun a : recspace A => forall
Proof. intros A r. apply axiom_16'. Qed.
```

# How to align inductive types

```
Lemma axiom_15 {A : Type'} : forall (a : list A), (@_mk_list A (@_dest_list A a)) = a.
Proof. _mk_dest_inductive. Qed.

Lemma axiom_16 : forall {A : Type'} (r : recspace A), ((fun a : recspace A => forall list
Proof.
  _dest_mk_inductive.
  - now exists nil.
  - exists (cons x0 x2). now rewrite <- H0.
Qed.
```

# How to align inductive types

```
Tactic Notation (at level 0) "breakgoal" "by" tactic(solvetac) :=
  let rec body := match goal with
  | |- _ \/ _ => left + right ; body (* Try both *)
  | |- _ /\ _ => split ; body
  | |- exists _,_ => eexists ; body (* The witness should be obvious *)
  | |- _ = _ => reflexivity
  | |- _ => first [by eauto | by solvetac] end
  in body. (* if solvetac/eauto cannot do the job, it fails to branch back. *)
```

The tactics:

```
Ltac _dest_inj_inductive :=
  induction x ; induction x' ; simpl ; intro e ;
  (* e is of the form "CONSTR n a f = CONSTR n' a' f'", so inversion
     gives hypotheses n=n' , a=a' and f=f'. *)
  inversion e ; auto ; repeat rewrite -> FCONS_inj in * ; f_equal ;
  match goal with IH : _ |- _ => now apply IH end.

Ltac _mk_dest_inductive := finv_inv_l ; try _dest_inj_inductive.

Ltac _dest_mk_inductive :=
  intros ; apply finv_inv_r ;
  [ intro H ; apply H ; full_destruct ; rewrite H ; clear H ; simpl in *
  | (* simply inducting over [x] such that [_dest_ x = r]. *)
    intros (x,<-) P ; induction x ; intros H ; apply H ; try breakgoal ].
```

# How to align inductive types

```
Inductive form :=
| FFalse : form
| Atom : N -> list term -> form
| FImp : form -> form -> form
| FAll : N -> form -> form.
```

```
Lemma _mk_dest_form : forall (a : form), (_mk_form (_dest_form a)) = a.
Proof. _mk_dest_inductive. Qed.

Lemma _dest_mk_form : forall (r : recspace (prod N (list term))), ((fun
Proof.
  intro r. _dest_mk_inductive.
  - now exists FFalse.
  - now exists (Atom x0 x1).
  - exists (FImp x3 x2). unfold _dest_form. now repeat f_equal.
  - exists (FAll x0 x2). unfold _dest_form. now repeat f_equal.
Qed.
```

# How to align total recursive functions

```
Definition LENGTH = ε (forall uv, x uv nil = 0 /\ (forall (a : A) (l : list A),
  x uv (cons a l) = N.succ (x uv l)) (76, (69, (78, (71, (84, 72))))).
```

```
Lemma LENGTH_def : length = LENGTH.
```

# How to align total recursive functions

```
Definition LENGTH = ε (forall uv, x uv nil = 0 /\ (forall (a : A) (l : list A),
  x uv (cons a l) = N.succ (x uv l)) (76, (69, (78, (71, (84, 72))))).

Lemma align_ε : forall P a, P a -> (forall x, P a -> P x -> a = x) -> a = ε P.

Lemma LENGTH_def : length = LENGTH.
```

# How to align total recursive functions

```
Definition LENGTH = ε (forall uv, x uv nil = 0 /\ (forall (a : A) (l : list A),
  x uv (cons a l) = N.succ (x uv l)) (76, (69, (78, (71, (84, 72))))).

Lemma align_ε : forall P a, P a -> (forall x, P a -> P x -> a = x) -> a = ε P.
```

```
Lemma LENGTH_def : length = LENGTH.
Proof.
  total_align.
Qed.
```

```
Lemma LENGTH_def : length = LENGTH.
Proof.
  generalize (NUMERAL (BIT0 (BIT0 (BIT1 (BIT1 (BIT0 (BIT0 (BIT1 0))))))),
             (NUMERAL (BIT1 (BIT0 (BIT1 (BIT0 (BIT0 (BIT0 (BIT1 0))))))),
               (NUMERAL (BIT0 (BIT1 (BIT1 (BIT1 (BIT0 (BIT0 (BIT1 0))))))),
                 (NUMERAL (BIT1 (BIT1 (BIT1 (BIT0 (BIT0 (BIT0 (BIT1 0))))))),
                   (NUMERAL (BIT0 (BIT0 (BIT1 (BIT0 (BIT1 (BIT0 (BIT1 0))))))),
                     NUMERAL (BIT0 (BIT0 (BIT0 (BIT1 (BIT0 (BIT0 (BIT1 0)))))))))))); intro p.
  apply fun_ext. intro l. simpl.
  match goal with |- _ = ε ?x _ _ => set (Q := x) end.
  assert (i: exists q, Q q). exists (fun _ => @length A). unfold Q. auto.
  generalize (ε_spec i). intro H. symmetry.
  induction l. simpl. apply H.
  simpl. rewrite <- IHl. apply H.
Qed.
```

# How to align total recursive functions

The tactic:

```
Ltac total_align1 :=
  align_ε ;
  [ repeat split ; intros ; auto.
  | intros f' H H' ; ext r ; induction r ;
    try ext a; try ext b ; try ext c ; try ext d ;
    try full_destruct ; (* with the correct induction principle, we have one case per clause,
                           we can replace [f] and [f']'s values with the corresponding
                           clause in [P] (that we have split).
                           By also rewriting all induction hypotheses,
                           the goal should become a reflexive equality. *)
    repeat match goal with
    H : _ |- _ => rewrite H end ; auto ].
```

# How to align total recursive functions

```
Fixpoint functions_form f : (prod N N) -> Prop :=
  match f with
  | FFalse => set0
  | Atom _ l => list_Union (map functions_term l)
  | FImp f f' => (functions_form f) `|` (functions_form f')
  | FAll _ f => functions_form f end.

Lemma functions_form_def : functions_form = (@ε ((prod N (prod N
Proof. total_align. Qed.

Fixpoint predicates_form f : (prod N N) -> Prop :=
  match f with
  | FFalse => set0
  | Atom a l => [set (a , lengthN l)]
  | FImp f f' => (predicates_form f) `|` (predicates_form f')
  | FAll _ f => predicates_form f end.

Lemma predicates_form_def : predicates_form = (@ε ((prod N (proc
Proof.
  total_align. by ssimpl.
Qed.
```

# Partial recursive functions

```
Definition HD := ε (forall uv l a, x uv (cons a l) = a) (72, 68).
```

```
Definition hd := hd (HD nil).
```

# Partial recursive functions

```
Definition HD := ε (forall uv l a, x uv (cons a l) = a) (72, 68).
```

```
Definition hd := hd (HD nil).
```

```
Lemma HD_def {A : Type'} : @hd A = @HD A.
Proof. unfold HD. partial_align (is_nil A). Qed.
```

# Partial functions

```
Definition Prenex_right f f' := if prenex f' then Prenex_right0 f f' else PRENEX_RIGHT f f'.
```

# Partial functions

```
Definition Prenex_right f f' := if prenex f' then Prenex_right0 f f' else PRENEX_RIGHT f f'.
```

```
Lemma align_ε_if1 {A B : Type'}
  (Q : A -> Prop) (f : A -> B) (P : (A -> B) -> Prop) :
  P f ->
  (forall f', P f -> P f' -> forall x, Q x -> f x = f' x) ->
  forall x, (if Q x then f x else ε P x) = ε P x.
```

# We wanted to prove in Rocq that

The length (in HOL-Light)

of a list (in HOL-Light)

which is an element of an inductive type (in HOL-Light)

(defined using inductive propositions)

is equal to

The length (in Rocq)

of a list (in Rocq)

which is an element of an inductive type (in Rocq)

And we can also align partial functions

# Appendix 1: more complex type

```
Inductive term : Set := V (_ : N) | Fn (_ : N) (_ : list term).
```

```
Lemma _dest_term_tl_inj : (forall t t', _dest_term t = _dest_term t' -> t = t')
  /\ (forall l l', _dest_list_204637 l = _dest_list_204637 l' -> l = l').
```

# Appendix 2: more complex functions

```
Fixpoint functions_term t : (prod N N) -> Prop :=
  match t with
  | V _ => set0
  | Fn n l => (n , lengthN l) |` (list_Union (map (functions_term) l)) end.

Lemma functions_term_def : functions_term = (@ε ((prod N (prod N (prod N (pro
Proof. term_align. Qed.
```

# Appendix 3: the same but better

```
Inductive form :=
| FFalse : form
| Atom : N -> list term -> form
| FImp : form -> form -> form
| FAll : N -> form -> form.
```

```
Lemma _mk_dest_form : forall (a : form), (_mk_form (_dest_form a)) = a.
Proof. _mk_dest_rec. Qed.

Lemma _dest_mk_form : forall (r : recspace (prod N (list term))), ((fun a : recspace (p
Proof.
  intro r. _dest_mk_rec.
  - now exists FFalse.
  - now exists (Atom x0 x1).
  - exists (FImp x3 x2). unfold _dest_form. now repeat f_equal.
  - exists (FAll x0 x2). unfold _dest_form. now repeat f_equal.
  - do 2 right. left. exists (_dest_form x0_1). exists (_dest_form x0_2).
    repeat split;auto. now apply IHx0_1. now apply IHx0_2.
  - do 3 right. exists n. exists (_dest_form x0). split. reflexivity. now apply IHx0.
Qed.
```

```
Fixpoint functions_form f : (prod N N) -> Prop :=
  match f with
  | FFalse => set0
  | Atom _ l => list_Union (map functions_term l)
  | FImp f f' => (functions_form f) `|` (functions_form f')
  | FAll _ f => functions_form f end.

Lemma functions_form_def : functions_form = (@ε ((prod N (prod N
Proof. total_align. Qed.
```

# Appendix 4: generalized

```
Class list_Class := {
  list : Type' -> Type';
  _mk_list : forall (A : Type'), (recspace A) -> list A;
  _dest_list : forall (A : Type'), (list A) -> recspace A;
  axiom_15 : forall (A : Type') (a : list A), (_mk_list A (_dest_list A a)) = a;
  axiom_16 : forall (A : Type') (r : recspace A), ((fun a : recspace A => forall list' :
Context {list_var : list_Class}.
```

```
Class LENGTH_Class := {
  LENGTH (A : Type') : (list A) -> num;
  LENGTH_def (A : Type') : (LENGTH A) = (@ε ((prod num (prod num (prod num (p
Context {LENGTH_var : LENGTH_Class}.
```

```
Axiom thm_LENGTH : forall (A : Type'), ((LENGTH A (NIL A)) = (NUMERAL _0)) /\
  (forall h : A, forall t : list A, (LENGTH A (CONS A h t)) = (SUC (LENGTH A t))).
```

```
Instance N_list : list_Class := {|
  axiom_15 := @Naxiom_15 ;
  axiom_16 := @Naxiom_16 |}.

Canonical N_list.
```

```
Instance N_LENGTH : LENGTH_Class := {| LENGTH_def := @NLENGTH_def |}.

Canonical N_LENGTH.
```

# Thank you for listening

The tactics are all located in https://github.com/Deducteam/coq-hol-light-real-with-N/blob/main/mappings.v#L267

(l.267-1072)

Attempts and ideas for typeclasses are located in https://github.com/agontard/rocq-hol-light-experimental