Generating Induction Principles for Nested Inductive Types in MetaCoq

Marcel Ullrich
Advisor - Yannick Forster
Supervisor - Gert Smolka

Saarland University
Programming Systems Lab

3rd July 2020

MetaCoq

- based on Template-Coq[1]
- a wealth of projects[2]
- PCUIC for idealised syntax
- plugins as Coq functions $\mathit{environment} \to \mathtt{term} \to \mathtt{term}$
- commands for quoting and environment manipulation

[1] Gregory Michael Malecha. "Extensible proof engineering in intensional type theory". PhD thesis. Harvard University, 2015.

[2] Matthieu Sozeau et al. "The MetaCoq Project". In: (2019).

PCUIC Terms

```
Inductive term : Type :=
    tRel : nat -> term
  | tVar : ident -> term
  | tEvar : nat -> list term -> term
  | tSort : Universe.t -> term
  | tProd : name -> term -> term -> term
  | tLambda : name -> term -> term -> term
  | tLetIn : name -> term -> term -> term -> term
  | tApp : term -> term -> term
  | tConst : kername -> Instance.t -> term
  | tInd : inductive -> Instance.t -> term
  | tConstruct : inductive -> nat -> Instance.t -> term
  | tCase : inductive * nat ->
            term -> term -> list (nat * term) -> term
  | tProj : projection -> term -> term
  | tFix : mfixpoint term -> nat -> term
  | tCoFix : mfixpoint term -> nat -> term
```

Nested Induction

Nested Induction

$$\mathsf{roseTree} ::= \mathsf{node}\ (\mathsf{xs} : \mathsf{list}\ \mathsf{roseTree})$$

Nested Induction

$$\mathsf{roseTree} ::= \mathsf{node}\ (\mathsf{xs} : \mathsf{list}\ \mathsf{roseTree})$$

$$\begin{aligned}
&\forall(\mathsf{P} : \mathsf{roseTree} \to \mathbb{P}).\\
&(\forall(\mathsf{xs} : \mathsf{list}\ \mathsf{roseTree}).\\
&\quad \mathsf{P}\ (\mathsf{node}\ \mathsf{xs})) \to\\
&\forall(\mathsf{r} : \mathsf{roseTree}).\ \mathsf{P}\ \mathsf{r}
\end{aligned}$$

Nested Induction

$$\mathsf{roseTree} ::= \mathsf{node}\ (\mathsf{xs} : \mathsf{list}\ \mathsf{roseTree})$$

$$\begin{aligned}
&\forall(\mathsf{P} : \mathsf{roseTree} \to \mathbb{P}).\\
&(\forall(\mathsf{xs} : \mathsf{list}\ \mathsf{roseTree}).\\
&\quad \mathsf{All}\ \mathsf{roseTree}\ \mathsf{P}\ \mathsf{xs} \to \mathsf{P}\ (\mathsf{node}\ \mathsf{xs})) \to\\
&\forall(\mathsf{r} : \mathsf{roseTree}).\ \mathsf{P}\ \mathsf{r}
\end{aligned}$$

$$\frac{}{\mathsf{All}\ \mathsf{X}\ \mathsf{P}\ [\,]}\ \mathsf{All}_{\mathsf{nil}} \qquad \frac{\mathsf{P}\ \mathsf{x} \qquad \mathsf{All}\ \mathsf{X}\ \mathsf{P}\ \mathsf{xs}}{\mathsf{All}\ \mathsf{X}\ \mathsf{P}\ (\mathsf{x} :: \mathsf{xs})}\ \mathsf{All}_{\mathsf{cons}}$$

Manual Derivation

```
Definition roseTree_induct :
  forall P : roseTree -> Prop,
  (forall xs : list roseTree,
  All P xs ->
  P (node xs)) ->
  forall r : roseTree, P r :=
fun P H => fix f r :=
match r with
  | node xs => H xs
    ((fix f' xs : All P xs :=
      match xs with
      | nil => All_nil P
      | cons y ys => All_cons P y ys (f y) (f' ys)
      end
    ) xs)
end.
```

Related Work

- Sozeau et al., “The MetaCoq Project”, 2019
- Johann and Polonsky, “Deep Induction: Induction Rules for (Truly) Nested Types”, 2020
- Tassi, “Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq”, 2019

Generalisation

$$
\begin{aligned}
\mathsf{E}_{\mathtt{roseTree}} : \forall(\mathsf{P} : \mathsf{roseTree} \to \mathbb{P}).\ & \\
& (\forall(\mathsf{xs} : \mathsf{list\ roseTree}). \\
& \quad \mathsf{P}_{\mathtt{list}}\ \mathsf{roseTree}\ \mathsf{P}\ \mathsf{xs} \to \mathsf{P}\ (\mathsf{node\ xs})) \to \\
& \forall(\mathsf{r} : \mathsf{roseTree}).\ \mathsf{P\ r}
\end{aligned}
$$

Generalisation

$$
\begin{aligned}
\mathsf{E}_{\mathsf{roseTree}} : \forall(\mathsf{P} : \mathsf{roseTree} \to \mathbb{P}).\ & \\
& (\forall(\mathsf{xs} : \mathsf{list\ roseTree}). \\
& \quad \mathsf{P}_{\mathsf{list}}\ \mathsf{roseTree\ P\ xs} \to \mathsf{P}\ (\mathsf{node\ xs})) \to \\
& \forall(\mathsf{r} : \mathsf{roseTree}).\ \mathsf{P\ r}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{E}_{\mathsf{roseTree}} := \lambda\ \mathsf{P}\ \mathsf{H}_{\mathsf{node}}.\ & \\
& \mathsf{fix\ f\ r}. \\
& \mathsf{match\ r\ [} \\
& \quad \mathsf{node\ xs} \Rightarrow \mathsf{H}_{\mathsf{node}}\ \mathsf{xs}\ (\mathsf{F}_{\mathsf{list}}\ \mathsf{roseTree\ P\ f\ xs}) \\
& \mathsf{]}
\end{aligned}
$$

Unary Parametricity

$$xs : \mathrm{list}\ (X : \mathbb{T}) := [\,] \mid x :: xs$$

$$\mathrm{list}^{\mathrm{t}} : \forall (X : \mathbb{T}).\ (X \to \mathbb{T}) \to \mathrm{list}\ X \to \mathbb{T}$$

$$\frac{}{\mathrm{list}^{\mathrm{t}}\ X\ P_X\ [\,]}\ \mathrm{nil}^{\mathrm{t}} \qquad \frac{P_X\ x \qquad \mathrm{list}^{\mathrm{t}}\ X\ P_X\ xs}{\mathrm{list}^{\mathrm{t}}\ X\ P_X\ (x :: xs)}\ \mathrm{cons}^{\mathrm{t}}$$

Unary Parametricity

$$xs : \mathrm{list}\ (X : \mathbb{T}) := [\,] \mid x :: xs$$

$$\mathrm{list}^{\mathrm{t}} : \forall (X : \mathbb{T}).\ (X \to \mathbb{T}) \to \mathrm{list}\ X \to \mathbb{T}$$

$$\frac{}{\mathrm{list}^{\mathrm{t}}\ X\ P_X\ [\,]}\ \mathrm{nil}^{\mathrm{t}} \qquad \frac{P_X\ x \qquad \mathrm{list}^{\mathrm{t}}\ X\ P_X\ xs}{\mathrm{list}^{\mathrm{t}}\ X\ P_X\ (x :: xs)}\ \mathrm{cons}^{\mathrm{t}}$$

$$(\forall x.\ P_X\ x) \to (\forall xs.\ \mathrm{list}^{\mathrm{t}}\ X\ P_X\ xs)$$

$$\mathrm{list}^{\mathrm{t}}\ X\ P_X\ xs \to \forall x.\ x \in xs \to P_X\ x$$

Implementation

```
Class registered {X:Type} (ty:X) :=
{
  assumptionType: Type;
  assumption: assumptionType;
  proofType: Type;
  proof: proofType
}.
```

```
Instance listInst : registered list := {|
asumptionType := ∀ X, (X → Type) → list X → Type;
assumption    := @listᵗ;
prodType      := ∀ X P, (∀ x, P x) →
                   ∀ xs, listᵗ P xs;
prod          := @listProof |}
```


```
Require Import String.

Inductive rtree := node (xs:list rtree).

Require Import MetaCoq.Induction.MetaCoqInductionPrinciples.

MetaCoq Run Scheme Elimination for nat.
Check nat_case_MC.
MetaCoq Run Scheme Induction for nat.
Check nat_ind_MC.

MetaCoq Run Set Nested Inductives.

MetaCoq Run Scheme rtree_induct' := Induction for rtree.
Check rtree_induct'.

MetaCoq Run Unset Nested Inductives.

MetaCoq Run Scheme rtree_induct := Induction for rtree.
Check rtree_induct.

MetaCoq Run Set Nested Inductives.

From MetaCoq.PCUIC Require Import PCUICAst.
MetaCoq Run Scheme term_induct := Induction for term.
Check term_induct.

Inductive list' X : Type :=
| nil' : list' X
| cons' : X -> list' X -> list' X.
```


```
Require Import String.

Inductive rtree := node (xs:list rtree).

Require Import MetaCoq.Induction.MetaCoqInductionPrinciples.

MetaCoq Run Scheme Elimination for nat.
Check nat_case_MC.
MetaCoq Run Scheme Induction for nat.
Check nat_ind_MC.

MetaCoq Run Set Nested Inductives.

MetaCoq Run Scheme rtree_induct' := Induction for rtree.
Check rtree_induct'.

MetaCoq Run Unset Nested Inductives.

MetaCoq Run Scheme rtree_induct := Induction for rtree.
Check rtree_induct.

MetaCoq Run Set Nested Inductives.

From MetaCoq.PCUIC Require Import PCUICAst.
MetaCoq Run Scheme term_induct := Induction for term.
Check term_induct.

Inductive list' X : Type :=
| nil' : list' X
| cons' : X -> list' X -> list' X.
```


```
Require Import String.

Inductive rtree := node (xs:list rtree).

Require Import MetaCoq.Induction.MetaCoqInductionPrinciples.

MetaCoq Run Scheme Elimination for nat.
Check nat_case_MC.
MetaCoq Run Scheme Induction for nat.
Check nat_ind_MC.

MetaCoq Run Set Nested Inductives.

MetaCoq Run Scheme rtree_induct' := Induction for rtree.
Check rtree_induct'.

MetaCoq Run Unset Nested Inductives.

MetaCoq Run Scheme rtree_induct := Induction for rtree.
Check rtree_induct.

MetaCoq Run Set Nested Inductives.

From MetaCoq.PCUIC Require Import PCUICAst.
MetaCoq Run Scheme term_induct := Induction for term.
Check term_induct.

Inductive list' X : Type :=
| nil' : list' X
| cons' : X -> list' X -> list' X.
```



```
nat_case_MC
     : forall p : nat -> Type,
       p 0 ->
       (forall H : nat, p (S H)) ->
       forall inst : nat, p inst
```


```
Require Import String.

Inductive rtree := node (xs:list rtree).

Require Import MetaCoq.Induction.MetaCoqInductionPrinciples.

MetaCoq Run Scheme Elimination for nat.
Check nat_case_MC.
MetaCoq Run Scheme Induction for nat.
Check nat_ind_MC.

MetaCoq Run Set Nested Inductives.

MetaCoq Run Scheme rtree_induct' := Induction for rtree.
Check rtree_induct'.

MetaCoq Run Unset Nested Inductives.

MetaCoq Run Scheme rtree_induct := Induction for rtree.
Check rtree_induct.

MetaCoq Run Set Nested Inductives.

From MetaCoq.PCUIC Require Import PCUICAst.
MetaCoq Run Scheme term_induct := Induction for term.
Check term_induct.

Inductive list' X : Type :=
| nil' : list' X
| cons' : X -> list' X -> list' X.
```

Messages ↗ Errors ↗ Jobs ↗

```
nat_ind_MC
     : forall p : nat -> Type,
       p 0 ->
       (forall H : nat, p H -> p (S H)) ->
       forall inst : nat, p inst
```


```
Require Import String.

Inductive rtree := node (xs:list rtree).

Require Import MetaCoq.Induction.MetaCoqInductionPrinciples.

MetaCoq Run Scheme Elimination for nat.
Check nat_case_MC.
MetaCoq Run Scheme Induction for nat.
Check nat_ind_MC.

MetaCoq Run Set Nested Inductives.

MetaCoq Run Scheme rtree_induct' := Induction for rtree.
Check rtree_induct'.

MetaCoq Run Unset Nested Inductives.

MetaCoq Run Scheme rtree_induct := Induction for rtree.
Check rtree_induct.

MetaCoq Run Set Nested Inductives.

From MetaCoq.PCUIC Require Import PCUICAst.
MetaCoq Run Scheme term_induct := Induction for term.
Check term_induct.

Inductive list' X : Type :=
| nil' : list' X
| cons' : X -> list' X -> list' X.
```



```
rtree_induct'
     : forall p : rtree -> Type,
       (forall xs : list rtree,
        standardNested.is_list rtree
          (fun H : rtree => p H) xs ->
        p (node xs)) -> forall inst : rtree, p inst
```


```
Require Import String.

Inductive rtree := node (xs:list rtree).

Require Import MetaCoq.Induction.MetaCoqInductionPrinciples.

MetaCoq Run Scheme Elimination for nat.
Check nat_case_MC.
MetaCoq Run Scheme Induction for nat.
Check nat_ind_MC.

MetaCoq Run Set Nested Inductives.

MetaCoq Run Scheme rtree_induct' := Induction for rtree.
Check rtree_induct'.

MetaCoq Run Unset Nested Inductives.

MetaCoq Run Scheme rtree_induct := Induction for rtree.
Check rtree_induct.

MetaCoq Run Set Nested Inductives.

From MetaCoq.PCUIC Require Import PCUICAst.
MetaCoq Run Scheme term_induct := Induction for term.
Check term_induct.

Inductive list' X : Type :=
| nil' : list' X
| cons' : X -> list' X -> list' X.
```



```
Nested_Inductives0 is defined
The mode Nested_Inductives was unset
```


```
Require Import String.

Inductive rtree := node (xs:list rtree).

Require Import MetaCoq.Induction.MetaCoqInductionPrinciples.

MetaCoq Run Scheme Elimination for nat.
Check nat_case_MC.
MetaCoq Run Scheme Induction for nat.
Check nat_ind_MC.

MetaCoq Run Set Nested Inductives.

MetaCoq Run Scheme rtree_induct' := Induction for rtree.
Check rtree_induct'.

MetaCoq Run Unset Nested Inductives.

MetaCoq Run Scheme rtree_induct := Induction for rtree.
Check rtree_induct.

MetaCoq Run Set Nested Inductives.

From MetaCoq.PCUIC Require Import PCUICAst.
MetaCoq Run Scheme term_induct := Induction for term.
Check term_induct.

Inductive list' X : Type :=
| nil' : list' X
| cons' : X -> list' X -> list' X.
```

Messages ↗ Errors ↗ Jobs ↗

```
rtree_induct
     : forall p : rtree -> Type,
       (forall xs : list rtree, p (node xs)) ->
       forall inst : rtree, p inst
```


```
MetaCoq Run Unset Nested Inductives.

MetaCoq Run Scheme rtree_induct := Induction for rtree.
Check rtree_induct.

MetaCoq Run Set Nested Inductives.

From MetaCoq.PCUIC Require Import PCUICAst.
MetaCoq Run Scheme term_induct := Induction for term.
Check term_induct.

Inductive list' X : Type :=
| nil' : list' X
| cons' : X -> list' X -> list' X.

Inductive rtree' : Type :=
| Node' (l : list' rtree') : rtree'.

MetaCoq Run Scheme rtree'_induct := Induction for rtree'.
Check rtree'_induct.

MetaCoq Run Derive Container for list'.

MetaCoq Run Scheme rtree'_induct' := Induction for rtree'.
Check rtree'_induct'.
```

```
term_induct
     : forall p : term -> Type,
       (forall n : nat, p (tRel n)) ->
       (forall i : string, p (tVar i)) ->
       (forall (n : nat) (l : list term),
        standardNested.is_list term (fun H : term => p H)
          l -> p (tEvar n l)) ->
       (forall u : Universe.t, p (tSort u)) ->
       (forall (na : name) (A : term),
        p A -> forall B : term, p B -> p (tProd na A B)) ->
       (forall (na : name) (A : term),
        p A -> forall t : term, p t -> p (tLambda na A t)) ->
       (forall (na : name) (b : term),
        p b ->
        forall B : term,
        p B -> forall t : term, p t -> p (tLetIn na b B t)) ->
       (forall u : term,
        p u -> forall v : term, p v -> p (tApp u v)) ->
       (forall (k : string) (ui : list Level.t_),
        p (tConst k ui)) ->
       (forall (ind : inductive) (ui : list Level.t_),
        p (tInd ind ui)) ->
       (forall (ind : inductive) (n : nat)
          (ui : list Level.t_), p (tConstruct ind n ui)) ->
       (forall (indn : inductive * nat) (p0 : term),
        p p0 ->
        forall c : term,
        p c ->
```


```
MetaCoq Run Unset Nested Inductives.

MetaCoq Run Scheme rtree_induct := Induction for rtree.
Check rtree_induct.

MetaCoq Run Set Nested Inductives.

From MetaCoq.PCUIC Require Import PCUICAst.
MetaCoq Run Scheme term_induct := Induction for term.
Check term_induct.

Inductive list' X : Type :=
| nil' : list' X
| cons' : X -> list' X -> list' X.

Inductive rtree' : Type :=
| Node' (l : list' rtree') : rtree'.

MetaCoq Run Scheme rtree'_induct := Induction for rtree'.
Check rtree'_induct.

MetaCoq Run Derive Container for list'.

MetaCoq Run Scheme rtree'_induct' := Induction for rtree'.
Check rtree'_induct'.
```



```
(forall (ind : inductive) (ui : list Level.t_),
 p (tInd ind ui)) ->
(forall (ind : inductive) (n : nat)
   (ui : list Level.t_), p (tConstruct ind n ui)) ->
(forall (indn : inductive * nat) (p0 : term),
 p p0 ->
 forall c : term,
 p c ->
 forall brs : list (nat * term),
 standardNested.is_list (nat * term)
   (fun H : nat * term =>
    let (_, y) := H in (True * p y)%type) brs ->
 p (tCase indn p0 c brs)) ->
(forall (p0 : inductive * nat * nat) (c : term),
 p c -> p (tProj p0 c)) ->
(forall mfix : list (def term),
 standardNested.is_list (def term)
   (fun x : def term =>
    (p (let (_, dtype, _, _) := x in dtype) *
     p (let (_, _, dbody, _) := x in dbody))%type)
   mfix -> forall idx : nat, p (tFix mfix idx)) ->
(forall mfix : list (def term),
 standardNested.is_list (def term)
   (fun x : def term =>
    (p (let (_, dtype, _, _) := x in dtype) *
     p (let (_, _, dbody, _) := x in dbody))%type)
   mfix -> forall idx : nat, p (tCoFix mfix idx)) ->
forall inst : term, p inst
```


```
MetaCoq Run Unset Nested Inductives.

MetaCoq Run Scheme rtree_induct := Induction for rtree.
Check rtree_induct.

MetaCoq Run Set Nested Inductives.

From MetaCoq.PCUIC Require Import PCUICAst.
MetaCoq Run Scheme term_induct := Induction for term.
Check term_induct.

Inductive list' X : Type :=
| nil' : list' X
| cons' : X -> list' X -> list' X.

Inductive rtree' : Type :=
| Node' (l : list' rtree') : rtree'.

MetaCoq Run Scheme rtree'_induct := Induction for rtree'.
Check rtree'_induct.

MetaCoq Run Derive Container for list'.

MetaCoq Run Scheme rtree'_induct' := Induction for rtree'.
Check rtree'_induct'.
```

Messages ↗ Errors ↗ Jobs ↗

```
list' is not a registered container and won't generate nested
inductive hypothesis.
Use `MetaCoq Run Derive Container for list'` to register list'
as container.
created rtree'_induct
rtree'_induct is defined
```


```
MetaCoq Run Unset Nested Inductives.

MetaCoq Run Scheme rtree_induct := Induction for rtree.
Check rtree_induct.

MetaCoq Run Set Nested Inductives.

From MetaCoq.PCUIC Require Import PCUICAst.
MetaCoq Run Scheme term_induct := Induction for term.
Check term_induct.

Inductive list' X : Type :=
| nil' : list' X
| cons' : X -> list' X -> list' X.

Inductive rtree' : Type :=
| Node' (l : list' rtree') : rtree'.

MetaCoq Run Scheme rtree'_induct := Induction for rtree'.
Check rtree'_induct.

MetaCoq Run Derive Container for list'.

MetaCoq Run Scheme rtree'_induct' := Induction for rtree'.
Check rtree'_induct'.
```



```
rtree'_induct
     : forall p : rtree' -> Type,
       (forall l : list' rtree', p (Node' l)) ->
       forall inst : rtree', p inst
```


```
MetaCoq Run Unset Nested Inductives.

MetaCoq Run Scheme rtree_induct := Induction for rtree.
Check rtree_induct.

MetaCoq Run Set Nested Inductives.

From MetaCoq.PCUIC Require Import PCUICAst.
MetaCoq Run Scheme term_induct := Induction for term.
Check term_induct.

Inductive list' X : Type :=
| nil' : list' X
| cons' : X -> list' X -> list' X.

Inductive rtree' : Type :=
| Node' (l : list' rtree') : rtree'.

MetaCoq Run Scheme rtree'_induct := Induction for rtree'.
Check rtree'_induct.

MetaCoq Run Derive Container for list'.

MetaCoq Run Scheme rtree'_induct' := Induction for rtree'.
Check rtree'_induct'.
```

Messages ↗ Errors ↗ Jobs ↗

```
"~~~~~~~~~~~~~~~~~~~"%string
"Translating test.list'"%string
list'ᵗ is defined
list'ᵗ_rect is defined
list'ᵗ_ind is defined
list'ᵗ_rec is defined
list'ᵗ_sind is defined
"test.list' has been translated."%string
list'ᵗ0 is defined
list'ᵗ0_rect is defined
list'ᵗ0_ind is defined
list'ᵗ0_rec is defined
list'ᵗ0_sind is defined
Fresh universe MetaCoq.Induction.addContainer.173 was added to
the context.
Fresh universe MetaCoq.Induction.addContainer.170 was added to
the context.
Please provide a proof for the predicate.
(forall (X : Type) (XP : X -> Type),
 (forall x : X, XP x) ->
 forall H : list' X, list'ᵗ0 X XP H)
list'_proof has type-checked, generating 1 obligation
Solving obligations automatically...
list'_proof_obligation_1 is defined
No more obligations remaining
list'_proof is defined
list'_inst is defined
New instance list'_inst was created
```

test.v

```
| cons' : X -> list' X -> list' X.

Inductive rtree' : Type :=
| Node' (l : list' rtree') : rtree'.

MetaCoq Run Scheme rtree'_induct := Induction for rtree'.
Check rtree'_induct.

MetaCoq Run Derive Container for list'.

MetaCoq Run Scheme rtree'_induct' := Induction for rtree'.
Check rtree'_induct'.
```

Messages Errors Jobs

```
rtree'_induct'
     : forall p : rtree' -> Type,
       (forall l : list' rtree',
        list'_0 rtree' (fun H : rtree' => p H) l ->
        p (Node' l)) -> forall inst : rtree', p inst
```

Comparison

Tassi in Elpi

- Embeddable λProlog interpreter
- syntax manipulation with named variables

$$\forall(A : \mathbb{T})\ (P_A : A \to \mathbb{T})\ (P : \mathrm{list}\ A \to \mathbb{T}).$$
$$P\ [\,] \to$$
$$(\forall(a : A)\ (l : \mathrm{list}\ A).\ P_A\ a \to P\ l \to P\ (a :: l)) \to$$
$$\forall(l : \mathrm{list}\ A).\ \mathrm{list}^t\ A\ P_A\ l \to P\ l$$

$$\forall(P : \mathrm{roseTree} \to \mathbb{T}).$$
$$(\forall xs.\ \mathrm{list}^t\ \mathrm{roseTree}\ P\ xs \to P\ (\mathrm{node}\ xs)) \to$$
$$\forall(r : \mathrm{roseTree}).\ \mathrm{roseTree}^t\ r \to P\ r$$

Isabelle - old-style

$$\left(\bigwedge xs.\ P_2\ xs \Rightarrow P_1\ (node\ xs)\right) \Rightarrow$$
$$P_2\ [\,] \Rightarrow \left(\bigwedge y\ ys.\ P_1\ y \Rightarrow P_2\ ys \Rightarrow P_2\ (y\#ys)\right) \Rightarrow$$
$$P_1\ ?tree$$

$$x : roseTree := node\ ys$$
$$xs, ys : listRoseTree := [\,] \mid cons\ x\ xs$$

Isabelle - new-style

$$\Big(\bigwedge xs.\ P_2\ xs \Rightarrow P_1\ (node\ xs)\Big) \Rightarrow$$
$$P_2\ [\,] \Rightarrow \Big(\bigwedge y\ ys.\ P_1\ y \Rightarrow P_2\ ys \Rightarrow P_2\ (y\#ys)\Big) \Rightarrow$$
$$P_1\ ?tree$$

$$x : roseTree := node\ ys$$
$$xs, ys : listRoseTree := [\,] \mid cons\ x\ xs$$

$$\Big(\bigwedge xs.\ \Big(\bigwedge t.\ t \in set\ xs \Rightarrow P\ t\Big) \Rightarrow P\ (node\ xs)\Big) \Rightarrow P\ ?tree$$

Lean

$$x : \mathrm{roseTree} := \mathrm{node}\ ys$$
$$xs, ys : \mathrm{listRoseTree} := \mathrm{nil} \mid \mathrm{cons}\ x\ xs$$

$$f\ \mathrm{nil} := [\,]$$
$$f\ (\mathrm{cons}\ x\ xs) := x :: f\ xs$$

$$\mathrm{def}\ \mathrm{roseTree.node} : \mathrm{list}\ \mathrm{roseTree} \to \mathrm{roseTree} :=$$
$$\lambda(a : \mathrm{list}\ \mathrm{roseTree}).\ \mathrm{nested.roseTree.node}\ (f^{-1}\ a)$$

Conclusion

Verification

$$wf\ \Sigma\ \Gamma \to wf\_inductive\ ind \to$$
$$\Sigma; \Gamma \vdash createElim\ ind : createElimType\ ind$$

- verification theoretically possible
- complicated in practice

Future Work

- abstraction layers

Future Work

- abstraction layers
- mutual induction

Future Work

- abstraction layers
- mutual induction
- more control for principles

Future Work

- abstraction layers
- mutual induction
- more control for principles
- tests with QuickChick[3]

[3] Maxime Dénès et al. "QuickChick: Property-based testing for Coq". In: *The Coq Workshop*. 2014.

Future Work

- abstraction layers
- mutual induction
- more control for principles
- tests with QuickChick[3]
- more plugins like countability and finiteness

[3] Dénès et al., "QuickChick: Property-based testing for Coq".

Conclusion

Thank you! Please ask questions.

```
Require Import MetaCoq.Induction.MetaCoqInductionPrinciples.
MetaCoq Run Set Nested Inductives.

MetaCoq Run Scheme rtree_induct := Induction for rtree.
Check rtree_induct.
```

Contributions:

- command to generate induction principles
- support for nested inductive types
- foundation for other plugins
- unfinished correctness proof

Github: `https://github.com/uds-psl/metacoq-nested-induction`

References I

- Anand, Abhishek et al. “Towards certified meta-programming with typed Template-Coq”. In: *International Conference on Interactive Theorem Proving*. Springer. 2018.
- Dénès, Maxime et al. “QuickChick: Property-based testing for Coq”. In: *The Coq Workshop*. 2014.
- Johann, Patricia and Andrew Polonsky. “Deep Induction: Induction Rules for (Truly) Nested Types”. In: *International Conference on Foundations of Software Science and Computation Structures*. Springer, Cham. 2020, pp. 339–358.
- Malecha, Gregory Michael. “Extensible proof engineering in intensional type theory”. PhD thesis. Harvard University, 2015.
- Sozeau, Matthieu et al. “The MetaCoq Project”. In: (2019).
- Tassi, Enrico. “Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq”. In: *ITP 2019 - 10th International Conference on Interactive Theorem Proving*. Portland, United States, Sept. 2019. DOI: 10.4230/LIPIcs.CVIT.2016.23

Lines of Code