Spécification et Validation de Programmes 5I554

Lecture 9 : Induction

(How?)

Pierre-Évariste Dagand

pierre-evariste.dagand@lip6.fr

Université Pierre & Marie Curie

Motivation

```
Inductive rosetree :=
| rosenode : nat → list rosetree → rosetree.

Fixpoint rev_rosetree (t: rosetree): rosetree :=
  match t with
  | rosenode n ts ⇒
    rosenode n (fold_left
                  (fun xs t ⇒ rev_rosetree t :: xs) ts [])
  end.

Lemma rev_rev_rosetree:
  ∀ t, rev_rosetree (rev_rosetree t) = t.
Proof. induction t. (* WTF?! *) Abort.
```

Motivation

```
Inductive term :=
| App :  term → term → term
| Abs :  (term → term) → term.

Error: Non strictly positive occurrence
of "term" in "(term  → term)  → term".
```

Motivation

```
Lemma vmap_map {A B}:
  ∀ f n,
  ∀ v : vector A n,
    list_from_vect (vmap f v) = map f (list_from_vect v).
Proof.
  intros f n v.
  induction v.
  (..)
```

Inductive Types

Anatomy of an inductive type

```
Inductive tree : Type :=
| Leaf : tree
| Node : nat → tree → tree → tree.
```

Vocabulary

- (Algebraic) datatype / signature
- Constructors / operations
- Recursive arguments / arity

Fixpoint interpretation

$$
\begin{array}{lcl}
\Sigma_{\mathsf{tree}}\ (X : \textsc{Type}) & : & \textsc{Type} \\
\Sigma_{\mathsf{tree}}\ \ X & \mapsto & \mathsf{unit} + \mathsf{nat} \times X \times X
\end{array}
$$

$$
\begin{array}{lcl}
\mu\ (\Sigma : \textsc{Type} \to \textsc{Type}) & : & \textsc{Type} \\
\mu\ \ \Sigma & \mapsto & \Sigma\ (\mu\ \Sigma)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{tree} & : & \textsc{Type} \\
\mathsf{tree} & \mapsto & \mu\ \Sigma_{\mathsf{tree}}
\end{array}
$$

Vocabulary

- Signature functor
- Built from a fixed grammar of type operators
- Tying the knot

Fixpoint interpretation

Remark

$$\begin{array}{lcl} \Sigma_{\mathsf{tree}}\,(X : \textsc{Type}) & : & \textsc{Type} \\ \Sigma_{\mathsf{tree}}\quad X & \mapsto & \mathsf{unit} + \mathsf{nat} \times X \times X \end{array}$$

is equivalent to

```
Inductive sigma_tree (X: Type): Type :=
| OpLeaf : sigma_tree X
| OpNode : nat → X → X → sigma_tree X.
```

Fixpoint interpretation

Historical origin

"The set $\mathcal{F}$ of propositional formulas over P is the smallest set that

- *contains P*
- *contains $\neg F$, for every F it contains*
- *contains $F \wedge G$, $F \vee G$ and $F \Rightarrow G$, for every F and G it contains"*

Mathematical logic, Cori & Lascar

EXERCISE: implement a datatype Fml of propositional formulas parameterized over P : TYPE.

Fixpoint interpretation

Solution

```
Inductive Fml (P: Type): Type :=
| Atom : P → Fml P
| Neg  : Fml P → Fml P
| Or   : Fml P → Fml P → Fml P
| And  : Fml P → Fml P → Fml P
| Impl : Fml P → Fml P → Fml P.
```

Recursion

Recursion

Over lists

Let $A : \textsc{Type}$ be a type parameter.
Let $X : \textsc{Type}$ and $\alpha : \Sigma_{\mathsf{list}}\ X \to X$.

EXERCISE: define Σ_{list} as an inductive type
EXERCISE: implement a function $\mathsf{fold\_list} : \mathsf{list}\ A \to X$
EXERCISE: implement a function $\mathsf{length} : \mathsf{list}\ A \to \mathsf{nat}$
EXERCISE: what is the relationship with Coq's
`fold_right` : $\forall$ `A X, (A` $\to$ `X` $\to$ `X)` $\to$ `X` $\to$ `list A` $\to$ `X`

Recursion

Solution

```
Variable A X: Type.
Inductive sigma_list X :=
| OpNil : sigma_list X
| OpCons : A → X → sigma_list X.
Variable alpha : sigma_list X → X.

Fixpoint fold_list (l: list A): X :=
  match l with
  | []  ⇒ alpha (OpNil _)
  | a :: xs ⇒ alpha (OpCons _ a (fold_list xs))
  end.

Definition length {A} :=
  fold_list A nat (fun xs ⇒ match xs with
                            | OpNil ⇒ 0
                            | OpCons _ n ⇒ S n
                            end).
```

Recursion

Let $X : \textsc{Type}$ and $\alpha : \Sigma_{\mathsf{tree}}\, X \to X$.

EXERCISE: implement a function $\mathsf{fold\_tree} : \mathsf{tree} \to X$
EXERCISE: implement a function $\mathsf{height} : \mathsf{tree} \to \mathsf{nat}$

Recall:

$$
\begin{array}{lcl}
\Sigma_{\mathsf{tree}}\,(X : \textsc{Type}) & : & \textsc{Type} \\
\Sigma_{\mathsf{tree}} \qquad X & \mapsto & \mathsf{unit} + \mathsf{nat} \times X \times X
\end{array}
$$

Recursion

Solution

```
Variable X: Type.
Variable alpha : sigma_tree X → X.

Fixpoint fold_tree (t: tree): X :=
  match t with
  | Leaf ⇒ alpha (OpLeaf _)
  | Node n l r ⇒ alpha (OpNode _ n (fold_tree l)
                                     (fold_tree r))
  end.

Definition height :=
  fold_tree nat
            (fun xs ⇒ match xs with
                      | OpLeaf ⇒ 0
                      | OpNode _ lh rh ⇒ 1 + max lh rh
                      end).
```

Recursion

In the Real World

```
interface TreeVisitor {
    void visit(Node n);
    void visit(Leaf l);
}

interface TreeElement {
    void accept(TreeVisitor visitor);
}

class Node implements TreeElement {
    private int x;
    print TreeElement l, r;

    public void accept(TreeVisitor visitor) {
        l.accept(visitor); visitor.visit(this);
        r.accept(visitor);
    }
    (...)
}
```

Initial algebra semantics

Σ_{tree} is functorial, *i.e.* we have:

$$\Sigma\text{-}\mathsf{map}_{\mathsf{tree}} : \forall X\, Y : \textsc{Type}.\ (X \to Y) \to \Sigma_{\mathsf{tree}}\ X \to \Sigma_{\mathsf{tree}}\ Y$$

We define (recursively)

$$\begin{tikzcd}
\mathsf{tree} \arrow[rr, "\mathsf{fold_tree}\ \alpha"] \arrow[d, "\mathsf{out_tree}"'] & & X \\
\Sigma_{\mathsf{tree}}\ \mathsf{tree} \arrow[rr, "\Sigma\text{-}\mathsf{map}_{\mathsf{tree}}\ (\mathsf{fold_tree}\ \alpha)"'] & & \Sigma_{\mathsf{tree}}\ X \arrow[u, "\alpha"']
\end{tikzcd}$$

EXERCISE: implement $\mathsf{out\_tree}$, $\Sigma\text{-}\mathsf{map}_{\mathsf{tree}}$, and $\mathsf{fold\_tree}$.

Vocabulary

- Algebra

Initial algebra semantics

Solution

```
Definition out_tree (t: tree): sigma_tree tree :=
  match t with
  | Leaf ⇒ OpLeaf _
  | Node n l r ⇒ OpNode _ n l r
  end.

Definition sigma_tree_map {X Y} (f: X → Y)
           (xs: sigma_tree X): sigma_tree Y :=
  match xs with
  | OpLeaf ⇒ OpLeaf _
  | OpNode n l r ⇒ OpNode _ n (f l) (f r)
  end.
```

Initial algebra semantics

Abstract non-sense

PARENTAL ADVISORY EXPLICIT CONTENT

$$
\begin{tikzcd}
\mu\ \Sigma \arrow[rr, dashed, "\mathsf{fold}\ \alpha"] & & X \\
& & \\
\Sigma\ (\mu\ \Sigma) \arrow[uu, "\cong"] \arrow[rr, dashed, "\Sigma\text{-}\mathsf{map}\ (\mathsf{fold}\ \alpha)"'] & & \Sigma\ X \arrow[uu, "\alpha"']
\end{tikzcd}
$$

Induction

Anatomy of a familiar induction principle

Induction over natural numbers

Statement: We show by recurrence the following property:
"for all n, $P(n)$"

Initialization: We show that the property is true for $n = 0$, *i.e.* $P(0)$.

Heredity: Assume that the property is true at m, *i.e.* $P(m)$.
We show that the property is true at $P(\mathsf{S}\ m)$.

Conclusion: By recurrence, the property is true for all n.

Anatomy of a familiar induction principle

Induction over natural numbers

Statement: We show by recurrence the following property: "for all n, $P(n)$"

$$P : \mathsf{nat} \to \textsc{Type}$$

Initialization: We show that the property is true for $n = 0$, *i.e.* $P(0)$.

Heredity: Assume that the property is true at m, *i.e.* $P(m)$. We show that the property is true at $P(\mathsf{S}\ m)$.

Conclusion: By recurrence, the property is true for all n.

Anatomy of a familiar induction principle

Induction over natural numbers

Statement: We show by recurrence the following property: "for all n, $P(n)$"

$$P : \mathsf{nat} \rightarrow \textsc{Type}$$

Initialization: We show that the property is true for $n = 0$, *i.e.* $P(0)$.

$$init : P\ 0$$

Heredity: Assume that the property is true at m, *i.e.* $P(m)$. We show that the property is true at $P(\mathsf{S}\ m)$.

Conclusion: By recurrence, the property is true for all n.

Anatomy of a familiar induction principle

Induction over natural numbers

Statement: We show by recurrence the following property: "for all n, $P(n)$"

$$P : \mathsf{nat} \to \textsc{Type}$$

Initialization: We show that the property is true for $n = 0$, *i.e.* $P(0)$.

$$\mathit{init} : P\ 0$$

Heredity: Assume that the property is true at m, *i.e.* $P(m)$. We show that the property is true at $P(\mathsf{S}\ m)$.

$$\mathit{step} : \forall m : \mathsf{nat}.\ P\ m \to P\ (\mathsf{S}\ m)$$

Conclusion: By recurrence, the property is true for all n.

Anatomy of a familiar induction principle

Induction over natural numbers

Statement: We show by recurrence the following property:
"for all n, $P(n)$"

$$P : \mathsf{nat} \to \textsc{Type}$$

Initialization: We show that the property is true for $n = 0$, *i.e.* $P(0)$.

$$\mathit{init} : P\ 0$$

Heredity: Assume that the property is true at m, *i.e.* $P(m)$.
We show that the property is true at $P(\mathsf{S}\ \ m)$.

$$\mathit{step} : \forall m : \mathsf{nat}.\ P\ m \to P\ (\mathsf{S}\ \ m)$$

Conclusion: By recurrence, the property is true for all n.

$$\mathsf{nat\_rect}\ P\ \mathit{init}\ \mathit{step} : \forall n : \mathsf{nat}.\ P\ n$$

Anatomy of another induction principle

Induction over trees

```
tree_rect :
  ∀ P : tree → Type,
    P Leaf →
    (∀ n l r, P l → P r → P (Node n l r)) →
      ∀ t : tree, P t
```

Remarks

- Induction is not restricted to natural numbers!
- In particular: semantics judgements (lecture 2)

Anatomy of another induction principle

Induction over trees, uniformly

```
Hypothesis P: tree → Type.

Inductive sigma_ind_tree: tree → Type :=
| OpIndLeaf :
    sigma_ind_tree Leaf
| OpIndNode : ∀ n l r,
    P l → P r → sigma_ind_tree (Node n l r).

Definition tree_rect':
 (∀ t, sigma_ind_tree t → P t) → ∀t : tree, P t.
```

EXERCISE: implement uniform induction for natural numbers.

Vocabulary

- Predicate lifting
- Inductive step

Anatomy of another induction principle

Solution

```
Hypothesis P: nat → Type.

Inductive sigma_ind_nat: nat → Type :=
| OpIndZ :
    sigma_ind_nat 0
| OpIndS: ∀ n,
    P n → sigma_ind_nat (S n).

Definition nat_rect':
 (∀ t, sigma_ind_nat t → P t) → ∀t : nat, P t.
```

Recursion *v.s.* Induction

Recursion

```
Inductive sigma_tree (X: Type): Type.
Definition fold_tree {X}:
 (sigma_tree X → X) → tree → X.
```

Induction

```
Inductive sigma_ind_tree
                 (P: tree → Type): tree → Type.
Definition tree_rect' {P}:
 (∀ t, sigma_ind_tree P t → P t) → ∀t : tree, P t.
```

From induction to recursion

Let $X : \textsc{Type}$.

$$
\begin{aligned}
\mathsf{tree\_rect}\ (\lambda t.\, X) : X &\to (\forall n : \mathsf{nat}.\ \forall l, r : \mathsf{tree}.\ X \to X \to X) \to \forall t : \mathsf{tree}.\ X \\
&\cong X \to (\mathsf{nat} \to X \to X \to X) \to X \\
&\cong (\mathsf{unit} \to X) \to (\mathsf{nat} \to X \to X \to X) \to X \\
&\cong (\mathsf{unit} \to X) \times (\mathsf{nat} \to X \to X \to X) \to X \\
&\cong (\mathsf{unit} \to X) \times (\mathsf{nat} \times X \times X \to X) \to X \\
&\cong ((\mathsf{unit} + \mathsf{nat} \times X \times X) \to X) \to X
\end{aligned}
$$

EXERCISE: implement yet another fold_tree from tree_rect
EXERCISE: implement height using this fold_tree

From recursion to induction?

"Induction is not derivable in λP2"

Geuvers (2001)

From recursion to induction?

A failed but informative attempt

```
Hypothesis P: tree → Type.
Hypothesis init: P Leaf.
Hypothesis step: ∀n l r, P l → P r → P (Node n l r).

Definition tree_ind' (t: tree): { t : tree & P t } :=
  fold_tree { t : tree & P t } alg t.

Lemma tree_ind'_correct: ∀t, projT1 (tree_ind' t) = t.
```

Initial algebra semantics of induction

Computational content

```
Inductive sigma_ind_nat: nat → Type :=
| OpIndZ :
    sigma_ind_nat 0
| OpIndS: ∀ n,
    P n → sigma_ind_nat (S n).

Fixpoint nat_rect'
         (IH: ∀ n, sigma_ind_nat n → P n)
         (n: nat): P n :=
  match n with
  | 0 ⇒ IH 0 OpIndZ
  | S n ⇒ IH (S n) (OpIndS n (nat_rect' IH n))
  end.
```

EXERCISE: implement the induction principle for trees

Initial algebra semantics of induction

Solution

```
Fixpoint tree_rect'
        (IH: ∀ t, sigma_ind_tree t → P t)
        (t : tree): P t :=
  match t with
  | Leaf ⇒ IH Leaf OpIndLeaf
  | Node n l r ⇒ IH (Node n l r)
                    (OpIndNode n l r
                               (tree_rect' IH l)
                               (tree_rect' IH r))
  end.
```

Inductive Families

Anatomy of an inductive family

```
Inductive color := red | black.

Inductive rbt: color → nat → Type :=
| bleaf:
    nat → rbt black 0
| rnode: ∀ n,
    rbt black n → rbt black n → rbt red n
| bnode: ∀ c1 c2 n,
    rbt c1 n → rbt c2 n → rbt black (S n).
```

Vocabulary

- Sort

Anatomy of an indexed induction principle

```
rbt_rect:
  ∀ P : ∀ c n, rbt c n → Type,
    (∀ n : nat, P black 0 (bleaf n)) →
    (∀ n l r,
         P black n l → P black n r
       → P red n (rnode n l r)) →
    (∀ c1 c2 n l r,
        P c1 n l → P c2 n r
      → P black (S n) (bnode c1 c2 n l r)) →
    ∀ c n t, P c n t
```

Indexed induction

Uniformly

```
Hypothesis P: ∀ c n, rbt c n → Type.

Inductive rbt_sigma_ind : ∀c n, rbt c n → Type :=
| OpBleaf: ∀ v, rbt_sigma_ind black 0 (bleaf v)
| OpRnode: ∀ n l r,
    P black n l → P black n r →
    rbt_sigma_ind red n (rnode n l r)
| OpBnode: ∀ c1 c2 n l r,
    P c1 n l → P c2 n r →
    rbt_sigma_ind black (S n) (bnode c1 c2 n l r).

Definition rbt_rect':
  (∀ c n t, rbt_sigma_ind c n t → P c n t) →
  ∀ c n (t: rbt c n), P c n t.
```

Indexed induction

Computational content

```
Fixpoint rbt_rect'
      (IH: ∀ c n t, rbt_sigma_ind c n t → P c n t)
      {c n} (t: rbt c n): P c n t
  := match t with
    | bleaf v ⇒ IH black 0 (bleaf v) (OpBleaf v)
    | rnode _ l r ⇒ IH _ _ _
                        (OpRnode _ _ _
                                 (rbt_rect' IH l)
                                 (rbt_rect' IH r))
    | bnode _ _ _ l r ⇒ IH _ _ _
                          (OpBnode _ _ _ _ _
                                   (rbt_rect' IH l)
                                   (rbt_rect' IH r))
    end.
```

Indexed induction

Application

EXERCISE: implement an indexed family of vectors

$$\mathsf{vec} : \mathsf{nat} \to \textsc{Type}$$

EXERCISE: implement its predicate lifting
EXERCISE: implement its uniform induction principle
EXERCISE: deduce its specialized induction principle

Extra

Mutual induction

```
Inductive rosetree :=
| rosenode : nat → list rosetree → rosetree.
```

EXERCISE: Any comment concerning `rosetree_rect`?

Mutual induction

Another attempt

```
Inductive rosetree :=
| rosenode:
    nat → list_rosetree → rosetree
with list_rosetree :=
| rosenil:
    list_rosetree
| rosecons:
    rosetree → list_rosetree → list_rosetree.
```

EXERCISE: More luck?

Mutual induction

Final attempt

```
Inductive rosetreeG: bool → Type :=
| rosenode:
    nat → rosetreeG false → rosetreeG true
| rosenil :
    rosetreeG false
| rosecons :
    rosetreeG true → rosetreeG false → rosetreeG false.

Definition rosetree := rosetreeG true.
Definition list_rosetree := rosetreeG false.
```

EXERCISE: More luck?

Mutual induction

Conceptually: reduces to an indexed family.

Alternatives

- Learn about `Combined Scheme`
- Manually construct the induction principles

EXERCISE: implement induction for the rebellious rose trees

Strict positivity

Non-example

```
Inductive T (A: Type) :=
| c : (T → A) → T.

Definition funny (t: T)(x: T): A :=
  match t with
  | c f ⇒ f x
  end.

Definition haha (t: T): A := funny t t.

Definition bottom: A := haha (c haha).
```

Strict positivity

Definition

- Strictly positive: no recursion to the left of an arrow
- Intuition: Liar's paradox

Implementation in Coq

- Syntactic check
- Necessarily conservative
- May require massaging the definitions

Conclusion

Lessons learned

Inductive definitions

- Non-indexed $\subseteq$ indexed
- Mutual $\approx$ indexed
- Positivity criteria

Induction

- Recursion + proofs
- Mechanically derived from signature
- Not always fully supported by Coq…

Take-away

Recursion

For any inductive type, you are able to

- define its signature functor
- switch between uniform and specialized recursion
- implement a uniform recursion operator

Induction

For any inductive type or inductive family, you are able to

- define its predicate lifting
- switch between uniform and specialized induction
- implement a uniform induction operator