Theory of Inductive Definitions

By Sebastian Pack & Max de Boer-Blazdell

Radboud University

Inductive Definitions: the basics

Recall these very basic inductive definitions:

Inductive *list* (A:Set) : Set :=
| *nil* : *list A*
| *cons* : $A \to$ *list* $A \to$ *list* A.

Inductive *tree* : Set :=
| *node* : *forest* $\to$ *tree*
with *forest* : Set :=
| *emptyf* : *forest*
| *consf* : *tree* $\to$ *forest* $\to$ *forest*.

There is a clear structure here: we start with Inductive, followed by the name and type of the of inductive types, followed by the name and types of constructors for the inductive type.

Inductive Definitions: Formalised

Formally, induction types are represented as $Ind[p](\Gamma_I := \Gamma_c)$

- p: number of **parameters** of the inductive type
- Γ_I: name and types of the **inductive type**
- Γ_c: name and types of the **constructors**

Inductive *list* (A:Set) : Set :=
| *nil* : *list A*
| *cons* : $A \to$ *list* $A \to$ *list* A.

Corresponds to:

$$\text{Ind}\ [1]\left([\text{list} : \text{Set} \to \text{Set}]\ := \begin{bmatrix} \text{nil} & : & \forall A : \text{Set},\ \text{list}\ A \\ \text{cons} & : & \forall A : \text{Set},\ A \to \text{list}\ A \to \text{list}\ A \end{bmatrix}\right)$$

Types of inductive definitions

Ind

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad \textsf{Ind}\,[p]\,(\Gamma_I := \Gamma_C) \in E \qquad (a : A) \in \Gamma_I}{E[\Gamma] \vdash a : A}$$

- Well-formed environment E with context Γ
- Inductive definition Γ_I
- a of type A is in Γ_I
- **Conclusion: $a : A$ is well-typed in E**

Types of constructors

Constr

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad \textsf{Ind}\ [p]\ (\Gamma_I := \Gamma_C) \in E \qquad (c : C) \in \Gamma_C}{E[\Gamma] \vdash c : C}$$

- Well-formed environment E with context Γ
- Inductive definition with constructors Γ_C in E
- Constructor c of type C is in Γ_C
- **Conclusion:** $c : C$ **is well-typed in** E

Well-formed Inductive Definitions

- Only **some** inductive definitions should be accepted
- "Bad" definitions lead to an **inconsistent** system:

```
Inductive Bad := bad (_: Bad → Bad)
```

Non-terminating terms when using paradox:

```
Definition paradox (x : Bad) : Bad :=
  match x with
  | bad f ⇒ f x
  end.
```

Well-formed Inductive Definitions

- Only **some** inductive definitions should be accepted
- "Bad" definitions lead to an **inconsistent** system:

Inductive Bad := bad ($\_$: $Bad \rightarrow Bad$)

Non-terminating terms when using paradox:

Definition $paradox$ (x : Bad) : Bad :=
  match x with
  | $bad\ f \Rightarrow f\ x$
  end.

- Coq error: Non strictly positive occurrence of "Bad" in "(Bad -> Bad) -> Bad (Explained later)

Arity of a Given Sort

Arity of sort s: A **Type** that **leads to sort s**.
Two cases:

- $\boldsymbol{T}$ is **already of sort s**. For example: $T = Prop$, because $Prop$ is already a sort
- T is a function type, where the **function** has **arity of sort s**. For example, $A \to Set$ is an arity of Set

A type T is an arity if there is an $s \in Sorts$ such that T is an arity of sort s

- $Sorts = \{Prop, Set, Type\}$

Type of Constructor

We say that T is a type of constructor of inductive type I in the following cases:

- **Direct application:** T is $(I \; t_1, ..., t_q)$, i.e. T itself directly **produces an instance of I**, possibly after applying arguments $t_1, ..., t_q$.

Inductive *bool* : Set :=
| *true* : *bool*
| *false* : *bool*.

- Type T is a constructor of bool if it directly produces an instance of bool
  - So T = true is a valid constructor of bool because it directly produces an instance of bool
  - And T = false is also a valid constructor of bool because it directly produces an instance of bool

Type of Constructor

We say that T is a type of constructor of inductive type I in the following cases:

- **Universally quantified type:** T is $\forall x : U, T'$ where **T' is also a constructor** for I.
  - We introduce a universal quantification over U, but eventually end up with T'
  - And T' is a valid constructor for I

```
Inductive list (A : Type) : Type :=
| nil : ∀ a:A, list A
| cons : ∀ a:A, A → list A → list A.
```

- For nil (similar approach for cons)
  - T = $\forall a : A$, list A
  - T' = list A, which is indeed a valid constructor

Positivity Condition

By enforcing that parameters should only occur in positive positions, we ensure that:

- Recursive functions terminate
- The type does not allow paradoxical or circular definitions

This means that parameters should:

- Be in the result of a constructor or function

```
Inductive Tree (A : Type) : Type :=
| Leaf : A → Tree A
| Node : Tree A → Tree A → Tree A.
```

- Not present on the left side of an arrow in a function

```
Inductive BadTree (A : Type) : Type :=
| bad : (BadTree A → A) → BadTree A.
```

Positivity Condition

The type of constructor T **satisfies the positivity constraints** for a set of constants $X_1, \ldots, X_k$ in the following cases:

- $\mathrm{T} = (X_j t_1, \ldots, t_q)$ for some j and **no** $X_1, \ldots, X_k$ **occur free** in any term t_i.
- $\mathrm{T} = \forall x : U, V$ and $X_1, \ldots, X_k$ only occur **strictly positively** in U and the type V **satisfies the positivity condition** for $X_1, \ldots, X_k$

```
Inductive list_with_length (A : Type) : nat → Type :=
| nil : list_with_length A 0
| cons : ∀ n : nat, A → list_with_length A n → list_with_length A (S n).
```

Positivity Condition Example

- $T = (X_j t_1, ..., t_q)$ for some j and **no** $X_1, ..., X_k$ **occur free** in any t_i.
  - T := nil
  - X_1 := list_with_length
  - t_1 := A, t_2 := 0
- $T = \forall x : U, V$ and $X_1, ..., X_k$ only occur **strictly positively** in U and the type V **satisfies the positivity condition** for $X_1, ..., X_k$
  - T := cons
  - X_1 := list_with_length
  - U := nat
  - V := A → list_with_length A n → list_with_length A (S n)

```
Inductive list_with_length (A : Type) : nat → Type :=
| nil : list_with_length A 0
| cons : ∀ n : nat, A → list_with_length A n → list_with_length A (S n).
```

Strict Positivity

The constants $X_1, ..., X_k$ occur strictly positively in T in the following cases:

- **No** $X_1, ..., X_k$ occur in T
- T converts to $(X_j t_1 ... t_q)$ for some j and **no** $X_1, ..., X_k$ **occur** in any t_i
  For example, if T is $X_i(A)$, and A does **not involve** X_1, then X_1 is strictly positive in T
- T converts to $\forall x : U, V$, and $X_1, ..., X_k$ **occur strictly positively** in type V but **none of them occur** in U
  So $X_1, ..., X_k$ can occur in the co-domain of the quantified type (V), but not in the domain of the quantified type (U)

`Inductive` $\mathit{Bad} := \mathit{bad}\ (\_ : \mathit{Bad} \to \mathit{Bad})$

Recursively (non-)Uniform Parameters

- **Recursively uniform** parameters that remain unchanged between recursive calls (A)
- **Recursively non-uniform** parameters that change between recursive calls (nat)
- $p =$ the total number of **parameters** (2)
- $m =$ the number of **recursively uniform parameters** (1)
- $p - m =$ the number of **recursively non-uniform** parameters (1)
- We can partially assign the recursively uniform parameters $q1, ..., q_r$ with $0 <= r <= m$

$\texttt{Inductive}\ \mathit{list\_with\_length}\ (A : \texttt{Type}) :\ \mathit{nat} \rightarrow \texttt{Type} :=$
$|\ \mathit{nil} :\ \mathit{list\_with\_length}\ A\ 0$
$|\ \mathit{cons} : \forall\ n :\ \mathit{nat},\ A \rightarrow \mathit{list\_with\_length}\ A\ n \rightarrow \mathit{list\_with\_length}\ A\ (S\ n).$

Strict Positivity for Inductive Types

- T converts to $(I\ a_1 \ldots a_r\ t_1 \ldots t_s)$ where I is the name of an inductive definition, $a_1 \ldots a_r$ are **parameters** (non-recursive part) and $t_1 \ldots t_s$ are **indices** (recursive part)
- I is defined as Ind [r] $(I : A := c_1 : P_1, \ldots, c_n : P_n)$ where c_i is a **constructor**, and P_i is the **type of the constructor**

```
Inductive vec (A : Type) : nat -> Type :=
| vnil : vec A 0
| vcons : A -> vec A n -> vec A (S n).
```

- Here, A is a parameter, and nat is an index
- The instantiated types of the constructors should also satisfy the nested positivity condition for $X_1, \ldots, X_k$

Strict Positivity for Inductive Types Continued

For the inductive definition of I, the constants $X_1, ... X_k$ are strictly positive in T if the following rules are satisfied:

- No $X_1, ..., X_k$ in **terms**

Inductive *list* (A : Type) : Type :=
| *nil* : *list* A
| *cons* : $A \rightarrow$ *list* (*list* A) $\rightarrow$ *list* A.

- They do not appear in any of the **non-recursively uniform parameters**
- Recursive arguments should satisfy **nested positivity**

Nested Positivity

If I is a **non-mutual inductive type** with r parameters, then the type of constructor T for I satisfies **nested positivity** in the following cases:

- $\mathrm{T} = (I\ a_1 ... a_r\ t_1 ... t_s)$ and no $X_1, ..., X_k$ occur in any t_i or a_j where $m \leq j \leq r$
- $\mathrm{T} = \forall x : U, V$ and $X_1, ..., X_k$ occur **only strictly positively** in U and the type V satisfies the **nested positivity condition** for $X_1, ... X_k$

Positivity vs. Strict Positivity vs. Nested Positivity

- **Positivity**
  - Scope: general recursive types
  - Goal: prevent negative occurrences
  - Common problems: recursive type in negative position
- **Strict Positivity**
  - Scope: recursive types, focus on valid positions
  - Goal: ensure recursive calls are strictly positive
  - Common problems: recursive type in non-positive context
- **Nested Positivity**
  - Scope: inductive families with parameters and indices
  - Goal: enforce valid positions in parameterised types
  - Common problems: recursive type in non-recursive parameter/index

Correctness Rules

W-Ind

$$\frac{\mathcal{WF}(E)[\Gamma_P] \qquad (E[\Gamma_I;\Gamma_P] \vdash C_i : s_{q_i})_{i=1\ldots n}}{\mathcal{WF}(E;\ \textsf{Ind}\ [l]\ (\Gamma_I\ :=\ \Gamma_C))[]}$$

- E is a global environment
- $\Gamma_p, \Gamma_I, \Gamma_C$ are contexts such that:
  - Γ_I is $[I_1 : \forall\Gamma_p, A_1; \ldots; I_k : \forall\Gamma_p, A_k]$
  - Γ_C is $[c_1 : \forall\Gamma_p, C_1; \ldots; c_n : \forall\Gamma_p, C_n]$
- With the following constraints:
  - $k > 0$ and all of I_j and c_i are **distinctive names** for $j = 1 \ldots k$ and $i = 1 \ldots n$
  - l is the size of Γ_p which is called the **context of parameters**
  - For $j = 1 \ldots k$ we have that A_j is an arity of sort s_j and $I_j \notin E$
  - For $i = 1 \ldots n$ we have that C_i is a type of constructor of I_{qi} which **satisfies the positivity condition** for $I_1, \ldots, I_k$ and $c_i \notin E$

Replacing sorts in arities

- Assume that A is an arity of some sort, and s is a sort
- Then, we can write $A_{/s}$ for the arity obtained by **replacing the sort of A with s**
- If A is well-typed in a global environment and local context, then $A_{/s}$ is also typable

$$A = \forall x : nat, x = x \rightarrow Type$$

- Here, A is a function that takes a proof that x = x, and produces a Type (which is its sort).

Then, we can replace A with Prop as follows:

$$A_{/Prop} = \forall x : nat, x = x \rightarrow Prop$$

Ind-family Typing Rule

Ind-Family

$$\frac{\begin{cases} \mathsf{Ind}\,[p]\,(\Gamma_I := \Gamma_C) \in E \\ (E[] \vdash q_l : P'_l)_{l=1\ldots r} \\ (E[] \vdash P'_l \leq_{\beta\delta\iota\zeta\eta} P_l\{p_u/q_u\}_{u=1\ldots l-1})_{l=1\ldots r} \\ 1 \leq j \leq k \end{cases}}{E[] \vdash I_j\, q_1 \ldots q_r : \forall [p_{r+1} : P_{r+1}; \; \ldots; \; p_p : P_p], (A_j)_{/s_j}}$$

- Conclusion: the j-th constructor I_j is **well-typed** under the inductive family type, taking all arguments $q1 \ldots q_r$ into account, producing a result type A_j, possibly modified by the sort s
- Where $\Gamma_P = [p_1 : P_1; \ldots; p_p : P_p]$ is the context of parameters
- Provided that 4 conditions are met

Ind-family Type Rule Conditions

Ind-Family

$$\frac{\left\{\begin{array}{l} \mathsf{Ind}\,[p]\,(\Gamma_I := \Gamma_C) \in E \\ (E[] \vdash q_l : P'_l)_{l=1\ldots r} \\ (E[] \vdash P'_l \leq_{\beta\delta\iota\zeta\eta} P_l\{p_u/q_u\}_{u=1\ldots l-1})_{l=1\ldots r} \\ 1 \leq j \leq k \end{array}\right.}{E[] \vdash I_j\, q_1 \ldots q_r : \forall [p_{r+1} : P_{r+1};\ \ldots ;\, p_p : P_p], (A_j)_{/s_j}}$$

- We have an inductive type that **exists** within the environment
- Each argument q_l matches the **expected type** P'_l of those arguments
- The argument P'_l is a **valid instance** of the inductive family after substituting q_u into the appropriate spots, which follows from $\beta, \delta, \iota, \zeta, \eta$ reductions
- This rule applies to all constructors of the inductive family

match ... with ... end

- Inductive object $m = (c_i x_1 ... x_{p_i})$
- Goal: Prove/Define property P on m
- Method: Consider each constructor c_i of m

match m as x in I_a return P with $(c_1 x_{11} \ldots x_{1p_1})$ $\Rightarrow f_1 \mid \ldots \mid (c_n x_{n1} \ldots x_{np_n}) \Rightarrow f_n$ end

```
Fixpoint example (m : I) :=
  match m with
  | c1 x11 ... x1p1 ⇒ f1
  ...
  | cn xn1 ... xnpn ⇒ fn
  end.
```

"case" Shorthand

```
Fixpoint example (m : I) :=
  match m with
  | c1 x11 ... x1p1 => f1
  ...
  | cn xn1 ... xnpn => fn
  end.
```

match m as x in I_a return P with $(c_1 x_{11} \ldots x_{1p_1})$
$\Rightarrow f_1 \mid \ldots \mid (c_n x_{n1} \ldots x_{np_n}) \Rightarrow f_n$ end

$\text{case}(m, (\lambda ax.P), \lambda x_{11} \ldots x_{1p_1} \cdot f_1 \mid \ldots \mid \lambda x_{n1} \ldots x_{np_n} \cdot f_n)$

More notation...

$\text{case}(m, (\lambda ax.P), \lambda x_{11} \ldots x_{1p_1} \cdot f_1 \mid \ldots \mid \lambda x_{n1} \ldots x_{np_n} \cdot f_n)$

Let:

- $m : I$
- $I : A$
- $\lambda ax.P : B$

The notation $[I : A \mid B]$ or just [I | B] means we are allowed to use $\lambda ax.P$ with m in the *match* statement.

Rules for [A|B]

Prod

$$\frac{[(I\ x) : A\prime | B\prime]}{[I : \forall x : A,\ A\prime | \forall x : A,\ B\prime]}$$

Set & Type

$$\frac{s_1 \in \{\text{Set}, \text{Type}(j)\} \qquad s_2 \in \mathcal{S}}{[I : s_1 | I \rightarrow s_2]}$$

- $\mathcal{S}$: The set of sorts
- Sort of *I* is **Set** or **Type** $\Rightarrow$ No restrictions

Rules for [A|B]

Prop

$$\frac{s \in \{\mathsf{SProp}, \mathsf{Prop}\}}{[I : \mathsf{Prop} | I \to s]}$$

- Exception when Sort of I is **Prop**
- Propositions are **not included** in extracted programs
- $\Rightarrow$ Extracted predicate would be defined over a **non-existent** object

Exception of the exception

Prop-extended

$$\frac{I \text{ is an empty or singleton definition} \qquad s \in \mathcal{S}}{[I : \text{Prop} | I \to s]}$$

We have no restrictions when:

- *I* is empty: trivial
- Singleton: **one** constructor, only **Prop** arguments
- Intuition: You can use an equality (**Prop**) to rewrite an object in **Set** (or **Type**)

Typing rule for *match*

- P: property over I
- c_{p_i}: ith constructor of I
- We write $\{c_{p_i}\}^P$ for the **type** of the **branch** that P gives for c_{p_i}

match

$$\frac{\begin{array}{l} E[\Gamma] \vdash c : (I\ q_1 \dots q_r\ t_1 \dots t_s) \\ E[\Gamma] \vdash P : B \\ [(I\ q_1 \dots q_r)|B] \\ (E[\Gamma] \vdash f_i : \{(c_{p_i}\ q_1 \dots q_r)\}^P)_{i=1\dots l} \end{array}}{E[\Gamma] \vdash \mathsf{case}(c, P, f_1| \dots |f_l) : (P\ t_1 \dots t_s\ c)}$$

Example: typing *plus*

```
Fixpoint plus (n m:nat) {struct n} : nat :=
match n with
| O ⇒ m
| S p ⇒ S (plus p m)
end.
```

Let $P := \lambda n : nat.\ nat$

$$E[\Gamma] \vdash n : nat$$
$$E[\Gamma] \vdash P : nat \to Set$$
$$[nat \mid nat \to Set]$$
$$E[\Gamma] \vdash m : \{O\}^{P} \equiv nat$$
$$\frac{E[\Gamma] \vdash S\ (plus\ p\ m) : \{S\ p\}^{P} \equiv nat}{E[\Gamma] \vdash case\,(n, P, m \mid \lambda p : nat.\ S\ (plus\ p\ m)) : P\ n \equiv nat}$$

ι-reduction for *match/case*

- No surprises here

$$\mathsf{case}((c_{p_i}\ q_1 \ldots q_r\ a_1 \ldots a_m), P, f_1 | \ldots | f_l) \triangleright_\iota (f_i\ a_1 \ldots a_m)$$

- Just keep *i*th branch for *i*th constructor

$$\begin{aligned} \mathsf{plus}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{O}))\ (\mathsf{S}\ \mathsf{O}) &\triangleright_\iota \mathsf{S}\ (\mathsf{plus}\ (\mathsf{S}\ \mathsf{O})\ (\mathsf{S}\ \mathsf{O})) \\ &\triangleright_\iota \mathsf{S}\ (\mathsf{S}\ (\mathsf{plus}\ \mathsf{O}\ (\mathsf{S}\ \mathsf{O}))) \\ &\triangleright_\iota \mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{O})) \end{aligned}$$

Fixpoint

- n mutually recursive definitions:

$$\text{fix } f_1(\Gamma_1) : A_1 := t_1 \text{ with} \ldots \text{with } f_n(\Gamma_n) : A_n := t_n \text{ for } f_i$$

- "for f_i" projects the ith term
- Shorthand where contexts are abstracted out:

$$\text{Fix } f_i\{f_1 : A'_1 := t'_1 \ldots f_n : A'_n := t'_n\}$$

- Formally: $t'_i = \lambda\Gamma_i.t_i$, similarly $A'_i = \forall\Gamma_i, A_i$

Typing rule for *Fixpoint*

Fix

$$\frac{(E[\Gamma] \vdash A_i : s_i)_{i=1\ldots n} \qquad (E[\Gamma;\ f_1 : A_1;\ \ldots;\ f_n : A_n] \vdash t_i : A_i)_{i=1\ldots n}}{E[\Gamma] \vdash \mathsf{Fix}\ f_i\{f_1 : A_1 := t_1 \ldots f_n : A_n := t_n\} : A_i}$$

- Give type judgement for every A_i
- Give type judgement that $f_i : A_i$
- Type judgements **transfer** over to Fixpoint with branches $f_i : A_i$

Guarded Fixpoints

$$\text{Fix } f_i\{f_1/k_1 : A_1 := t_1 \ldots f_n/k_n : A_n := t_n\}$$

- k_i: Integer pointing to the argument of f_i that gets **structurally smaller** (recall *struct*)

```
Fixpoint plus (n m:nat) {struct n} : nat :=
match n with
| O => m
| S p => S (plus p m)
end.
```

$$\text{Fix } plus\{plus/1 : nat \to nat \to nat$$
$$:= \lambda n, m : nat.\ case\,(n, P, m \mid \lambda p : nat.\ S(plus\ p\ m))\}$$

With $P := \lambda n : nat.\ nat$

Guarded Fixpoints (rules)

$$\text{Fix } f_i\{f_1/k_1 : A_1 := t_1 \dots f_n/k_n : A_n := t_n\}$$

- Each A_i starts with $\geq k_i$ products $\forall y_1 : B_1, \dots \forall y_{k_i} : B_{k_i}$
- B_{k_i} is an inductive type

If f_j occurs in t_i:

- $\geq k_j$ arguments
- k_jth argument is **structurally smaller** than y_{k_i}

Structurally smaller

Suppose we have

- $y : Ind[r](\Gamma_i := \Gamma_C)$
- $\Gamma_I := [I_1 : A_1; \ldots; I_k : A_k]$
- $\Gamma_C := [c_1 : C_1; \ldots; c_n : C_n]$

Structurally smaller than y are:

- Variables corresponding to recursive arguments
  - e.g.: In branch $S\ p \Rightarrow (S(plus\ p\ m))$, p is smaller as it is a recursive argument
- $(t\ u)$ and $\lambda x.t$ when t is structurally smaller
- case$(m, P, f_1 \ldots f_n)$, if:
  - $m : I_p$ for some p
  - Each f_i is structurally smaller