Introduction to the Calculus of Inductive Constructions

Workshop $\forall X.X\pi$, Vienna Summer of Logic

Christine Paulin-Mohring

Université Paris Sud & INRIA Saclay - Île-de-France

July 18, 2014

Motivations

- Limitations of first-order logic
  - need to work in axiomatic theories
  - consistency issue
  - infinite models of any cardinality
  - equational reasoning but no computation on terms
- Calculus of Inductive Constructions
  - a powerful functional basis
  - types to automatically classify objects
  - a general scheme to declare data-structures and relations
    - rules for recursion / induction
    - logical foundation of type theory (Agda, Coq)

Examples of inductive data-types

Emphasis is on constructors

```
Inductive bool := true | false.
Inductive nat :=  O | S : nat → nat.
Inductive min := At : nat → min | Imp : min → min → min.

Inductive tree A
    := leaf | node : A → tree A → tree A → tree A.

Inductive BDT := T | F | var : nat → (bool → BDT) → BDT.
Inductive W A B := node : ∀(a:A),(B a → W A B) → W A B.
```

Examples of inductive data-types

Emphasis is on constructors

```
Inductive bool := true | false.
Inductive nat :=  O | S : nat → nat.
Inductive min := At : nat → min | Imp : min → min → min.

Inductive tree A
    := leaf | node : A → tree A → tree A → tree A.

Inductive BDT := T | F | var : nat → (bool → BDT) → BDT.
Inductive W A B := node : ∀(a:A),(B a → W A B) → W A B.
```

Examples of inductive data-types

Emphasis is on constructors

```
Inductive bool := true | false.
Inductive nat :=  O | S : nat → nat.
Inductive min := At : nat → min | Imp : min → min → min.

Inductive tree A
    := leaf | node : A → tree A → tree A → tree A.

Inductive BDT := T | F | var : nat → (bool → BDT) → BDT.
Inductive W A B := node : ∀(a:A),(B a → W A B) → W A B.
```

Expected properties

```
Inductive min := At : nat → min | Imp : min → min → min.
```

Initiality – iterator (fold)

```
Variable X : Type.
Variable Xat : nat → X.
Variable Ximp : X → X → X.
Definition It (A : min) : X := ...
Lemma Itat : ∀ n, It (At n) = Xat n.
Lemma Itint : ∀ A B, It (Imp A B) = Ximp (It A) (It B).
```

Non confusion

```
Lemma diff : ∀ n A B, (At n) ≠ (Imp A B).
```

Induction

```
Variable P (A : min) : Prop.
Variable Pat : ∀(n:nat),P (At n).
Variable Pimp : ∀ A B : min,P A → P B → P (Imp A B)
Definition min_ind : ∀ A : min,P A := ...
```

Examples of Inductive Relations

Relations defined by inference rules, clauses

$$\frac{}{A, l \vdash A, r} \qquad \frac{A, l \vdash B, r}{l \vdash A \Rightarrow B, r} \qquad \frac{l \vdash A, r \qquad B, l \vdash r}{A \Rightarrow B, l \vdash r}$$

```
Inductive proof : set min → set min → Prop :=
  ax : ∀(A:min)(l r:set min), proof (A :: l) (A :: r)
| right : ∀(A B:min)(l r:set min)
    proof (A :: l) (B :: r) → proof l (Imp(A,B) :: r)
| left : ∀(A B:min)(l r:set min)
    proof l (A :: r) → proof (B :: l) r
    → proof (Imp(A,B) :: l) r
```

Examples of Inductive Relations

Relations defined by inference rules, clauses

$$\frac{}{A, l \vdash A, r} \qquad \frac{A, l \vdash B, r}{l \vdash A \Rightarrow B, r} \qquad \frac{l \vdash A, r \qquad B, l \vdash r}{A \Rightarrow B, l \vdash r}$$

```
Inductive proof : set min → set min → Prop :=
  ax : ∀(A:min)(l r:set min), proof (A :: l) (A :: r)
| right : ∀(A B:min)(l r:set min)
    proof (A :: l) (B :: r) → proof l (Imp(A,B) :: r)
| left : ∀(A B:min)(l r:set min)
    proof l (A :: r) → proof (B :: l) r
    → proof (Imp(A,B) :: l) r
```

Expected properties

Minimality

```
Variable P (l r : set min) : Prop.
Variable Pax : ∀(A:min)(l r:set min),P (A :: l) (A :: r).
Variable Pright : ∀(A B:min)(l r:set min)
      P (A :: l) (B :: r) → P l (Imp(A,B) :: r).
Variable Pleft : ∀(A B:min)(l r:set min)
      P l (A :: r) → P (B :: l) r → P (Imp(A,B) :: l) r.
Lemma proof_ind : ∀ l r, proof l r → P l r.
```

Mathematical justification

Intersection of a non-empty family of sets

$$\bigcap G = \bigcap_{X \in G} X = \{x \in X_0 | \forall X \in G, x \in X\}$$

Fixpoints of monotonic operators

$$\begin{aligned} F(P)(l,r) \stackrel{\text{def}}{=} & \exists A\, l'\, r', l = A :: l' \wedge r = A :: r' \\ & \vee \exists A\, B\, r', r = \mathtt{Imp}(A,B) :: r' \wedge P(A :: l, B :: r') \\ & \vee \exists A\, B\, l', l = \mathtt{Imp}(A,B) :: l' \wedge P(l', A :: r) \wedge P(B :: l', r) \end{aligned}$$

$$\begin{aligned} \mathtt{proof} \stackrel{\text{def}}{=} & \bigcap \{P \in \wp(M \times M) | \forall l\, r, F(P)(l,r) \Rightarrow P(l,r)\} \\ & \Leftrightarrow F(\mathtt{proof}) \end{aligned}$$

Minimality

$$(\forall l\, r, F(P)(l,r) \Rightarrow P(l,r)) \Rightarrow \forall l\, r. \mathtt{proof}(l,r) \Rightarrow P(l,r)$$

Inductive data-types

- start with an infinite set (usually $\mathbb{N}$)
- find a physical representation
  - support type : words, trees, . . . (large enough, infinite branching)
  - map “nodes” to index of constructors
- inductive definition of the set of terms
- (abstract) type of well-formed terms

Calculus of Inductive Constructions

- calculus and logic behind the Coq proof assistant
- can be used as higher-order logic
- propositions as types, proofs as objects
- dependent types
- internal computation $\texttt{fib}(7) \equiv 13$
- constructive, intensional logic
  - $\lambda x \Rightarrow \neg\neg x \neq \lambda x \Rightarrow x$
  - $\texttt{bool} \neq \mathsf{Prop}$
  - $\not\vdash \neg\neg A \to A$

Outline

Logical Rules : functional part

- Same language for terms and types:
  $$t ::= \mathcal{S} \mid \mathcal{V} \mid \forall x : t, t \mid \lambda x : t \Rightarrow t \mid t\ t$$
- Sorts:
  $$\mathcal{S} = \{\mathsf{Prop}, (\mathsf{Type}_i)_{i \in \mathbb{N}}\}$$
- Notation:
  $$t \to u \stackrel{\text{def}}{=} \forall \_ : t, u$$
- Judgements:
  $$\Gamma \vdash t : t'$$
  - every object is typed
  - a type is a term of type a sort
  - everything (signature, hypothesis) is declared and named in the context in a consistent way

$$A : \mathsf{Type}, R : A \to A \to \mathsf{Prop}, x : A, y : A, p : R\,x\,y \vdash$$

(Informal) Rules

- Sorts: $\mathsf{Prop}, \mathsf{Type}_i : \mathsf{Type}_{i+1}$
- Variables: $\dfrac{\Gamma \vdash A : s \quad x \notin \Gamma}{\Gamma, x : A \vdash}$ $\qquad \dfrac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$
- Product: $\dfrac{\Gamma, x : A \vdash B : s \quad \Gamma \vdash A : s'}{\Gamma \vdash \forall x : A, B : s}$ $\quad s = s'$ or $s = \mathsf{Prop}$
- Abstraction/Application:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A \Rightarrow t : \forall x : A, B} \qquad \frac{\Gamma \vdash t : \forall x : A, B \quad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B[x \leftarrow u]}$$

- Computation (β) $\dfrac{\Gamma \vdash t : A \quad \Gamma \vdash B : s \quad A \equiv B}{\Gamma \vdash t : B}$

Examples

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A \Rightarrow t : \forall x : A, B} \qquad \frac{\Gamma \vdash t : \forall x : A, B \quad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B[x \leftarrow u]}$$

- functional application

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A \Rightarrow t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B}$$

- natural deduction for implication

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

- natural deduction for universal quantification

$$\frac{\Gamma \vdash B \quad x \notin \Gamma}{\Gamma \vdash \forall x, B} \qquad \frac{\Gamma \vdash \forall x, B}{\Gamma \vdash B[x \leftarrow u]}$$

Dependent types

type of maps parameterized by a domain

```
map : ∀(A B:Type)(A → Prop) → Type
find : ∀(A B:Type)(d:A → Prop)(m:map A B d)(x:A)d x → B
```

Fundamental (non-)inductive definitions

```
Inductive zero : Type := .
Inductive unit : Type := tt : one.
Inductive sum (A B:Type) : Type :=
   inl : A → sum A B | inr : B → sum A B.
Inductive sig (A:Type) (B:A → Type) : Type :=
   pair : ∀ a:A, B a → sig A B.
```

propositions as types $\rightsquigarrow$ constructor $=$ introduction rule

$$\texttt{zero} \equiv \bot \quad \texttt{unit} \equiv \top \quad \texttt{sum}\, A\, B \equiv A \vee B$$

$$\texttt{sig}\, A\, (\lambda x \Rightarrow B) \equiv \exists x : A, B \quad \texttt{sig}\, A\, (\lambda\_ \Rightarrow B) \equiv A \wedge B$$

Elimination by pattern-matching

- Any value in an inductive type I starts with a constructor $c_1 | \ldots | c_p$
- Initiality in the non-recursive case :
  $X : \mathrm{Type} \quad f_1, \ldots, f_p : \ldots \quad \mathtt{match}_I(f_1, \ldots, f_p) : I \to X$
- Dependent pattern-matching (proof by case):
  $P : I \to \mathrm{Type} \quad f_1, \ldots, f_p : \ldots \quad \mathtt{match}_I(f_1, \ldots, f_p) : \forall i : I, P\, i$
- Exactly one branch f_i for each constructor c_i
- when $c_i : \forall (x_1 : A_1) \ldots (x_n : A_n) I$ we require
  $f_i : \forall (x_1 : A_1) \ldots (x_n : A_n) P\,(c_i\, x_1 \ldots x_n)$
- Computation : $\mathtt{match}_I(f_1, \ldots, f_p)(c_i\, t_1 \ldots t_n) \longrightarrow f_i\, t_1 \ldots t_n$

Pattern-matching in Coq

- Coq fully expanded notation :

```
match p as i in I return P i with
 c1 x1...xn1 ⇒ t1
|...
|cp x1...xnp ⇒ tp
end
: P p
```

- works for any (co-)inductive definition
- more complex patterns are compiled : always complete

Records

Dependent records as sigma types

```
Variable A : Type.
Variable B : A → Type.
Record dep := mkd {p1 : A; p2 : B p1}.
```

Compiled to

```
Inductive dep := mkd : ∀(p1:A),(B p1) → dep.
Definition p1 (p:dep) : A
  := match p with mkd a _ ⇒ a end.
Definition p2 (p:dep) : B (p1 p)
  := match p return B (p1 p) with mkd a b ⇒ b end.
```

Dependent types

- Types (Propositions) can depend on type variables, object variables, proof variables
- Atomic predicates $P : A \to \mathsf{Type}$ (variables, inductive definitions)
- Predicates defined by case analysis (non-canonical form)

Example : finite map

```
Variables A B : Type.
Inductive optB := Def : optB | Val : B → optB.

Definition isB (o:optB) : Prop :=
match o with Def ⇒ False | Val _ ⇒ True end.

Record map (d:A → Prop) :=
  mkmap { acc :> A → optB ;
          indom : ∀ (x:A), d x → isB (acc x)}.

Definition find (d:A → Prop) (m:map d) (x:A) (p:d x) : B
   := match m x as o return isB o → B with
         Def ⇒ (λ (h:False) ⇒ match h return B with end)
       | Val b ⇒ (λ _ ⇒ b)
      end (indom _ M x p).
```

Recursive constructions

- Possible recursive arguments in a type of constructor
- Normalisation implies decidability of type-checking and consistency
- Strict positivity condition

```
Inductive L := lam : (L → L) → L.
Definition app (l : L) : L → L
    := λ m ⇒ match l with (lam f) ⇒ f m end.
```

- Monotonicity is not enough at the predicative level

```
Inductive R : Type := Ri : ((R → Prop) → Prop) → R.
```

- Nested definitions are accepted, as well as mutually inductive definitions

```
Inductive S := Si : list S → S.
Inductive S := Si : LS → S
with LS := nil : LS | cons : S → LS → LS.
```

Fixpoints

```
Fixpoint size (s: S) : nat
  := match s with Si l ⇒ S (sizel l) end
with sizel (ls : LS) : nat
  := match ls with nil ⇒ 0
        | cons s' ls' ⇒ size s' + sizel ls' end.
```

- structural recursion on one argument in an inductive type
- a syntactic condition limits recursive calls to “subterms”
- fixpoint reduction is restricted when the recursive argument starts with a constructor
- more advanced recursive schemes are encoded

Induction/Recursion

The same scheme for programming and proving

```
Variable Q : nat → Type.
Variable q0 : Q 0.
Variable qS : ∀ i, Q i → Q (S i).
Fixpoint rec (n:nat) : Q n :=
   match n with O ⇒ q0 | S i ⇒ qS i (rec i) end
```

Imperative view

```
int rec (int n) {
  res = q0;
  /*@ loop invariant res : Q i */
  for (int i = 0; i < n; i++) {
    res = qs(i,res); }
  return res;
}
```

General recursion

Well-founded relations

```
Variable A : Type.
Variable R : A → A → Prop.
Inductive Acc (x: A) : Prop :=
     Acc_intro : (∀ y:A, R y x → Acc y) → Acc x.
Definition well_founded := ∀ a:A, Acc a.
```

Well-founded fixpoint

```
Lemma Acc_inv : ∀ x:A, Acc x → ∀ y:A, R y x → Acc y.

Variable P : A → Type.
Variable F : ∀ x:A, (∀ y:A, R y x → P y) → P x.
Fixpoint Fix_F (x:A) (a:Acc x) : P x :=
    F (λ (y:A) (h:R y x) ⇒ Fix_F (Acc_inv a h)).
```

Inductive families

The essential case is equality

```
Inductive eq A (x:A) : A → Prop := refl : eq A x x.
```

$$t \equiv u \quad \Leftrightarrow \quad \texttt{refl}\, t : t = u$$

Indexed types : vectors carrying their size

```
Inductive vec A : nat → Type
  := nil : vec A 0 | add : ∀ n, A → vec A n → vec A (S n)
```

Logical relations as indexed types

```
Inductive proof : set min → set min → Prop :=
  ax : ∀(A:min)(l r:set min), proof (A :: l) (A :: r)
| right : ∀(A B:min)(l r:set min)
    proof (A :: l) (B :: r) → proof l (Imp(A,B) :: r)
| left : ∀(A B:min)(l r:set min)
    proof l (A :: r) → proof (B :: l) r
    → proof (Imp(A,B) :: l) r
```

- term p : `proof` l r concretely represents a proof-tree and can be transformed.
- with `proof` l r : Prop, `ax ___` $\neq$ `right ___` cannot be proven.

Outline

Summary on the language

- Two fundamental mechanisms
  - higher-order functions
  - inductive definitions
- Functional programming languages + induction principles
- Indexed families integrates logic in types (proof by type-checking)
- Different ways to represent the same notion
  - recursive/recursive ($\neq$ algorithms)
  - inductive/inductive ($\neq$ definitions)
  - inductive/recursive (declarative versus computable)
  - constructive/classical ($\vee$, $\exists$ versus $\neg$)
- Intentional theory : limited equality

Dependent pattern-matching

use indexed types to specialise the pattern-matching scheme

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

Definition hd A n (p:vec A (S n)) : A :=
  match p with (add _ a q) ⇒ a end.

Definition hd A n (p:vec A (S n)) : A :=
  match p in vec _ k
  return (match k with O ⇒ unit | S _ ⇒ A end)
  with nil ⇒ tt
     | (add _ a q) ⇒ a end.
```

Equality

- equality in type theory is tricky
- $t \equiv u \Rightarrow t = u \Rightarrow t =_S u$
- add axioms to simplify proofs
  - extensionality
  - proof irrelevance
  - K-axiom
- Homotopy Type Theory
  - better understanding of the structure of equality proofs
  - logical caracterisation of what is a proposition, a set
  - inductive types with equality (quotients)

Proof automation

- basic steps : introduce hypothesis, apply lemmas
- apply the constructors of the inductive definition for resolution
- generate ad-hoc induction principles
- uniform schemes to deal with all inductive definitions
  - intros with patterns to destruct hypothesis
    (constructor names are irrelevant)

Proof by reflection

Use the Coq language to internally program proof strategies

```
Definition build (l r : set min) : bool :=
  if r=(Imp a b)::r' then build (a::l) (b::r')
  else if l=(Imp a b)::l'
       then build (b::l') r && build l' (a::r)
       else (inter l r) <> empty
Lemma correct : ∀ l r, build l r = true → proof l r.
Definition pr : proof l0 r0 := correct l0 r0 (refl true).
```

- A proof of `proof` *l r* is done by computing `build`
- Computation can be large but proof-term stays small
- This technique is both used on large examples and for systematic small steps (Ssreflect)

Conclusion

- both a logical and a programming platform
- strong logical framework
  - gives you several way to represent a notion
  - consistency sensible to minor changes
- (advanced) computation is part of the trusted kernel
- do-it yourself approach
- encourages abstract reasoning but also proofs hacking
- type-checking in the kernel is your safeguard