MPRI 2-7-2: Proof Assistants

Bruno Barras

Oct 4, 2019

Recap

Simple inductive types (datatypes):

```
Inductive nat : Type := O : nat | S : nat->nat.
Inductive bool := true | false.
Inductive list (A:Type) : Type :=
 nil | cons (hd:A) (tl:list A).
Inductive tree (A:Type) :=
 leaf | node (_:A) (_:nat->tree A).
```

Smallest type closed by introduction rules (constructors)

Parameters: `cons : forall A:Type, A -> list A -> list A`
Coq prelude: `cons 0 nil : list nat`

Recap: Elimination rules

Generated elimination scheme (not primitive):

```
nat_rect
  : forall P:nat->Type,
    P O -> (forall n, P n -> P (S n)) ->
    forall n, P n.
  := fun P h0 hS => fix F n :=
      match n return P n with
      | O => h0
      | S k => hS k (F k)
      end
```

Eliminator of recursive type =
dependent pattern-matching + guarded fixpoint

Recap: Logical connectives

Logical connectives and their non-dependent elimination schemes:

```
Inductive True : Prop := I.
 True_rect : forall P:Type, P -> True -> P.

Inductive False : Prop := .
 False_rect : forall P:Type, False -> P

Inductive and (A B:Prop) : Prop :=
 conj (_:A) (_:B).
 and_rect : forall (A B:Prop) (P:Type), (A->B->P)-> A/\B
      -> P

Inductive or (A B:Prop) : Prop :=
 or_introl (_:A) | or_intror (_:B).
 or_ind : forall (A B P:Prop), (A->P) -> (B->P) -> P.
```

Overview

1. Inductive types
   - Equality
   - Arithmetic
   - Vectors
2. Theory of Inductive types
   - Strict Positivity
   - Dependent pattern-matching
   - Guarded fixpoint
   - The guard condition

Equality as an inductive family

```
Inductive eq (A:Type) (x:A) : A -> Prop :=
| eq_refl : eq A x x.
```

Elimination:

- `eq_rect: ∀ A x (P:A->Type), P x -> ∀ y, x=y -> P y`
- Dependent version (generated by `Scheme`):

```
∀ A x (P:∀ z, x=z ->Type), P x eq_refl ->
  ∀ y (e:x=y) -> P y e
```

Equational theory of `nat`

Dependent elimination needed to prove minimality:

```
match n return n=O \/ ∃ m, n=S m with
| O => inl eq_refl : (O=O \/ ∃ m, O=S m)
| S k => inr (ex_intro k eq_refl)
       : (S k = O \/ ∃ m, S k = S m)
end
```

Equational theory of `nat`

Injectivity of constructors:

```
Definition pred (n:nat) :=
  match n with O => O | S k => k end.

f_equal pred : S n = S m -> n = m
```

Tactic `injection H`:

- applies this construction on hyp `H: C` $t_1 .. t_n$ `= C` $u_1 .. u_n$
- derives proofs of $t_1 = u_1$ .. $t_n = u_n$

Equational theory of `nat`

Discrimination of constructors:

```
Definition P (n:nat) :=
  match n return Prop with O => True | S k => False end.

match (e:0=1) in _=y return P y with
| eq_refl => I : P 0 (* P 0 = True *)
end : P 1 (* P 1 = False *)
```

Tactic `discriminate`:

- solves goals of the form `C` $t_1 \ldots t_n$ `<>` `D` $u_1 \ldots u_k$
- `discriminate H` solves the goal when

  `H : C` $t_1 \ldots t_n$ `= D` $u_1 \ldots u_k$

Vectors (Lists with size)

Inductive type with parameters and index:

```
Inductive vect (A:Type) : nat -> Type :=
| niln : vect A O
| consn :
   A -> forall n:nat, vect A n -> vect A (S n).
```

which defines

- a family of types-predicates:
  $\Gamma \vdash \mathit{vect} : \mathbf{Type} \to \mathit{nat} \to \mathbf{Type}$
- a set of introduction rules for the types in this family

$$\frac{\Gamma \vdash A : \mathbf{Type}}{\Gamma \vdash \mathtt{niln}_A : \mathit{vect}\ A\ O}$$

$$\frac{\Gamma \vdash A : \mathbf{Type} \quad \Gamma \vdash a : A \quad \Gamma \vdash n : \mathit{nat} \quad \Gamma \vdash l : \mathit{vect}\ A\ n}{\Gamma \vdash \mathtt{consn}_A\ a\ n\ l : \mathit{vect}\ A\ (S\ n)}$$

Vectors : elimination

- an elimination rule (pattern-matching operator with a result depending on the object which is eliminated)

$$\frac{\begin{array}{c}\Gamma \vdash v : \mathit{vect}\ A\ n \qquad \Gamma, p{:}\mathit{nat}, x{:}\mathit{vect}\ A\ p \vdash C(p,x) : s \\ \Gamma \vdash t_1 : C(O, \mathtt{niln}_A) \\ \Gamma, a{:}A, n{:}\mathit{nat}, l{:}\mathit{vect}\ A\ n \vdash t_2 : C(S\ n, \mathtt{consn}_A\ a\ n\ l)\end{array}}{\Gamma \vdash \left(\begin{array}{l}\mathtt{match}\ v\ \mathtt{as}\ x\ \mathtt{in}\ \mathit{vect}\ \_\ p\ \mathtt{return}\ C(p,x)\ \mathtt{with} \\ \quad \mathtt{niln} \Rightarrow t_1 \mid \mathtt{consn}\ a\ n\ l \Rightarrow t_2 \\ \mathtt{end}\end{array}\right) : C(n,v)}$$

- and the obvious reduction rules (ι-reduction)

Well-formed inductive definitions

Issues

Constructors of the inductive definition I have type:

$$\Gamma \ : \ \forall (z_1 : C_1) \ldots (z_k : C_k).I\, a_1 \ldots a_n$$

where C_i can feature intances of I.
Question: can these instances be arbitrary?

Issues

Constructors of the inductive definition I have type:

$$\Gamma \ : \ \forall (z_1 : C_1) \ldots (z_k : C_k).I\, a_1 \ldots a_n$$

where C_i can feature intances of I.
Question: can these instances be arbitrary? No!
Example:

```
Inductive lambda : Type :=
| Lam : (lambda -> lambda) -> lambda
```

Issues

Constructors of the inductive definition I have type:

$$\Gamma \ : \ \forall (z_1 : C_1) \ldots (z_k : C_k).I\, a_1 \ldots a_n$$

where C_i can feature intances of I.
Question: can these instances be arbitrary? No!
Example:

```
Inductive lambda : Type :=
| Lam : (lambda -> lambda) -> lambda
```

We define:

```
Definition app (x y:lambda)
   := match x with (Lam f) => f y end.
Definition Delta := Lam (fun x => app x x).
Definition Omega := app Delta Delta.
```

and the evaluation of Ω loops.

Necessity of restrictions

Things can even be worse:

```
Inductive lambda : Type :=
| Lam : (lambda -> lambda) -> lambda
```

Now define:

```
Fixpoint lambda_to_nat (t : lambda) : nat :=
  match t with Lam f -> S (lambda_to_nat (f t)) end.
```

Necessity of restrictions

Things can even be worse:

```
Inductive lambda : Type :=
| Lam : (lambda -> lambda) -> lambda
```

Now define:

```
Fixpoint lambda_to_nat (t : lambda) : nat :=
  match t with Lam f -> S (lambda_to_nat (f t)) end.
```

What happens with `(lambda_to_nat (Lam (fun x => x)))`?

The way out: (strict) positivity condition

- An inductive type is defined as the smallest type generated by a set $(\Gamma_i)_{1\leq i\leq n}$ of constructors.
- We can see it as $\mu X, \oplus_{1\leq i\leq n}\Gamma_i(X)$ (with μ a fixpoint operator on types).
  Eg: $\mathbb{N} = \mu X.\, 1 + X$ and so $\mathbb{N} = 1 + \mathbb{N}$
- The existence of this smallest type can be proved at the impredicative level when the operator $\lambda X, \oplus_{1\leq i\leq n}\Gamma_i(X)$ is monotonic.
  $\mu X : \mathbb{P}.(X \to A) \to A$ has a fixpoint...
- In order both to ensure monotonicity and to avoid paradox (predicativity of `Type`), Coq enforces a strict positivity condition: X should never appear on the left of an arrow in the type of its constructors.

The way out: (strict) positivity condition

More precisely, if the type (a.k.a arity) of a constructor is:

```
c : C1 -> ... -> Ck -> I a1 .. ak
```

it is well-formed when:

- `I a1 .. ak` is well-formed w.r.t. the uniformity of parametric arguments and typing constraints;
- `I` does not appear in any of the `a1`, ... `ak`;
- Each `Ci` should either not refer to `I` or be of the form:

  ```
  C'1 -> ... C'm -> I b1 ... bk
  ```

  well typed and with no other occurrence of `I`.

And the rule generalizes as such to dependent products (instead of arrow).

More well-formation conditions...

There are more constraints, that will be explained later:

1. predicativity/impredicativity
   An inductive is predicative when all constructor argument types live in a sort not bigger than the declared sort for the inductive
2. restriction on eliminations
   when the predicativity condition is not satisfied

Size paradoxes

Girard's paradox:

- **Type** : **Type**
- Generalizes to X : **Type** with an embedding **Type** $\to X$

```
Inductive e (A:s1) : s2 := C (_:A).
```

- $C : A \to e(A)$
- pattern-matching: $e(A) \to A$
- reduction: C and pattern-matching are inverses

If $s_2 : s_1$, the paradox applies...

Conclusion: inductive definitions must be predicative, otherwise eliminations must be restricted (see Paulin's Habilitation thesis)

Dependent pattern-matching

```
Inductive I (p:Par) : A -> s :=
| κ (x1:C1)...(xn:Cn) : I p u
| ...

match t as h in I _ a return P(a,h) with
| κ x1 ... xn => e
...
end
```

Typing conditions:

- $\vdash t : I\ q\ b$
- $a : A[q/p], h : I\ q\ a \vdash P : s'$
- $x_1 : C_1[q/p], ..., x_n : C_n[q/p] \vdash e : P(u[q/p], \kappa\ q\ x_1 ... x_n)$

Then the match has type $P(b, t)$

Tactics for case analysis

- `case t` is the most primitive. It:
  - generates a (proof) term of the form `match t with ...`;
  - guesses the return type from the goal (under the line);
  - does not introduce/name the arguments of the constructor by default, but there is a syntax for chosing names.
- The `case_eq` variant modifies the guessing of the return type so that equalities are generated.
- The `destruct` variant modifies the guessing of the return type so that it generalizes the hypotheses depending on `t`.

The fixpoint operator (reduction)

Fixpoint expression with dependent result

$$(\texttt{fix}\ f\ (x : A) : B(x) := t(f, x))$$

- Typing

$$\frac{f : (\forall (x : A), B(x)), x : A \vdash t : B(x)}{\vdash (\texttt{fix}\ f\ (x : A) : B(x) := t(f, x)) : \forall (x : A), B(x)}$$

Fixpoint operator : well-foundness

Requirement of the Calculus of Inductive Constructions :

- the argument of the fixpoint has type an inductive definition
- recursive calls are on arguments which are *structurally* smaller

Example of recursor on natural numbers

$$
\begin{array}{l}
\lambda P : \texttt{nat} \to s, \\
\lambda H_O : P(O), \\
\lambda H_S : \forall m : \texttt{nat}, P(m) \to P(S\ m), \\
\texttt{fix}\ f\ (n : \texttt{nat})\ :\ P(n)\ := \\
\quad \texttt{match}\ n\ \texttt{as}\ y\ \texttt{return}\ P(y)\ \texttt{with} \\
\qquad O \Rightarrow H_O \mid S\ m \Rightarrow H_S\ m\ (f\ m) \\
\quad \texttt{end}
\end{array}
$$

is correct with respect to CCI : recursive call on m which is structurally smaller than n in the inductive `nat`.

Fixpoint operator : typing rules

$$\frac{\Gamma \vdash I : s \quad \Gamma, x : A \vdash C : s' \quad \Gamma, x : I, f : (\forall x : I, C) \vdash t : C \quad t|_f^{\emptyset} <_I x}{\Gamma \vdash (\texttt{fix}\ f\ (x : I) : C := t) : \forall x : I, C}$$

the main definition of $t|_f^{\rho} <_I x$ are:

$$\frac{z \in \rho \cup \{x\} \quad (u_i|_f^{\rho} <_I x)_{i=1\ldots n} \quad A|_f^{\rho} <_I x \quad (t_i|_f^{\rho \cup \{x \in \vec{x_i} | x : \forall y : \vec{U}. I\, \vec{u}\}} <_I x)_i}{\texttt{match}\ z\ u_1 \ldots u_n\ \texttt{return}\ A\ \texttt{with}\ (c_i\ \vec{x_i} \Rightarrow t_i)_i\ \texttt{end}|_f^{\rho} <_I x}$$

$$\frac{t \neq (z\ \vec{u}) \text{ for } z \in \rho \cup \{x\} \quad t|_f^{\rho} <_I x \quad A|_f^{\rho} <_I x \quad (t_i|_f^{\rho} <_I x)_i}{\texttt{match}\ t\ \texttt{return}\ A\ \texttt{with}\ (c_i\ \vec{x_i} \Rightarrow t_i)_i\ \texttt{end}|_f^{\rho} <_I x}$$

$$\frac{y \in \rho}{f\ (y\ u_1 \ldots u_n)|_f^{\rho} <_I x} \qquad \frac{f \notin FV(t)}{t|_f^{\rho} <_I x}$$

+ contextual rules ...

Remarks on the criteria

- It covers simply the schema of primitive recursive definitions and proofs by induction which have recursive calls on all subterms.

$$
\begin{array}{l}
\lambda P : \mathtt{list}\, A \to s, \\
\lambda f_1 : P\, \mathtt{nil}, \\
\lambda f_2 : \forall (a : A)(l : \mathtt{list}\, A), P\, l \to P\, (\mathtt{cons}\, a\, l), \\
\mathtt{fix}\ \mathit{Rec}\ (x : \mathtt{list}\, A)\ :\ P\, x\ := \\
\quad \mathtt{match}\ x\ \mathtt{return}\ P\, x\ \mathtt{with} \\
\qquad \mathtt{nil} \Rightarrow f_1 \mid (\mathtt{cons}\, a\, l) \Rightarrow f_2\, a\, l\, (\mathit{Rec}\, l) \\
\quad \mathtt{end}
\end{array}
$$

- has type

$$
\begin{array}{l}
\forall P : \mathtt{list}\, A \to s, \\
P\, \mathtt{nil}, \to \\
(\forall (a : A)(l : \mathtt{list}\, A), P\, l \to P\, (\mathtt{cons}\, a\, l)) \to \\
\forall (x : \mathtt{list}\, A), P\, x
\end{array}
$$

Remarks on the criteria

Possibility of recursive call on deep subterms

```
Fixpoint mod2 (n:nat) : nat :=
  match n with O => O | S O => S O
           | S (S x) => mod2 x
  end
```

Possibility of recursive call on terms build by case analysis if each branch is a strict subterm (actual rule very complex!):

```
Definition pred (n:nat) : n<>0->nat:=
 match n return n<>0->nat with
    S p => (fun (h:S p<>0) => p)
  | O => (fun (h:0<>0) =>
          match h (refl_equal 0) return nat with end)
 end
Fixpoint F (n:nat) : C :=
 match iszero n with
   (left H (*H:n=O*)) => ...
 | (right H (*H:n<>0*)) => F (pred n H)
 end
```

Remarks on the criteria

Note : only the recursive arguments with the *same* type are considered recursive (otherwise paradox related to impredicativity)

```
Inductive Singl (A:Prop) : Prop := c : A -> Singl A.
Definition ID : Prop := forall (A:Prop), A -> A.
Definition id : ID := fun A x => x.
Fixpoint f (x : Singl ID) : bool :=
    match x with (c a) => f (a (Singl ID) (c ID id)) end.
```

f (c ID id) $\longrightarrow$ f (id (Singl ID)(c ID id)) $\longrightarrow$ f (c ID id)

Tactics for induction

`fix <n>`, where `<n>` is a numeral is the most primitive. It:

- generates a (proof) term of the form:

```
fun g1 g2 => fix f h1 h2 t h3 {struct t} := ?F h1 h2 t
```

  where:

- `g1`, `g2` are the objects in the context (above the line);
- `h1`, `h2`, `t`, `h3` are the objects quantified in the goal (under the line);
- `?F` can call `f` (= recursive calls);
- the termination of `f` is should eventually be guaranteed by structural recursion on `t`;

`Qed` checks the well-formedness, which was not guaranteed so far: error messages come late and may be difficult to interpret.

Tactics for induction

`elim t` applies an induction scheme, i.e. a lemma of the form:

```
forall P : T -> Type, .... -> forall t' : T, P t'
```

- It guesses argument `P` from the goal (under the line), abstracting all the occurrences of `t`.
- It guesses the elimination scheme to be used (`T_ind`, `T_rect`,...) from the sort of the goal and the type of `t`.
- The `elim t using S` variant allows to provide a custom elimination scheme (or lemma!) `S`, with the same unification heuristic.
- The `induction t` tactic guesses argument `P` taking into account the possible hypotheses depending on `t` present in the context (above the line). Plus it can introduce and name things automatically.

Remark: the `rewrite` tactic does a similar guessing job...

Fixpoint expansion

We would expect the usual expansion rule for fixpoints:

```
(fix f (x:A): B x := t(f,x))e
⟶ t(fix f (x:A): B x := t(f,x), e)
```

Fixpoint expansion

We would expect the usual expansion rule for fixpoints:

```
(fix f (x:A): B x := t(f,x))e
```
$\longrightarrow$ `t(fix f (x:A): B x := t(f,x), e)`

... but this leads to infinite unfolding (SN broken)

Fixpoint expansion

We would expect the usual expansion rule for fixpoints:

```
(fix f (x:A): B x := t(f,x))e
⟶ t(fix f (x:A): B x := t(f,x), e)
```

... but this leads to infinite unfolding (SN broken)

Solution: allow this reduction only when *e* is a constructor

Beware:

- Guard condition ensures consistency (meaningful definition)
- Expansion restriction imposes a strategy

Next week...

Advanced features of inductive types

- Prop vs Type
- Impredicative inductive definitions