MPRI 2-7-2: Proof Assistants

Bruno Barras

Jan 10th, 2022
Recap

Simple inductive types (datatypes):

```coq
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):

\[
\text{nat} \_rect \\
: \forall P : \text{nat} \to \text{Type}, \\
P \text{ O} \to (\forall n, P n \to P (S n)) \to \\
\forall n, P n. \\
:= \text{fun } P \text{ h0 hS } \to \text{fix } F n := \\
\quad \text{match } n \text{ return } P n \text{ with} \\
\quad \mid \text{ O } \Rightarrow \text{h0} \\
\quad \mid \text{ S } k \Rightarrow \text{hS } k \ (F \ k) \\
\text{end}
\]

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
Dependent elimination needed to prove minimality:

```haskell
match n return n=O \/ \ \exists m, n=S m with
| O => inl eq_refl : (O=O \/ \ \exists m, O=S m)
| S k => inr (ex_intro k eq_refl)
          : (S k = O \/ \ \exists m, S k = S m)
end
```
Equational theory of \texttt{nat}

Injectivity of constructors:

\textbf{Definition} \texttt{pred} (n: \texttt{nat}) :=
\begin{verbatim}
m\texttt{match n with} O \Rightarrow O \mid S k \Rightarrow k \end{verbatim}
\texttt{end.}

\texttt{f\_equal pred : S n = S m \Rightarrow n = m}

\textbf{Tactic} \texttt{injection H}:

- applies this construction on hyp \texttt{H : C t1..t_n = C u1..u_n}
- derives proofs of \texttt{t1=u_1 .. t_n=u_n}
Equational theory of \texttt{nat}

Discrimination of constructors:

\begin{verbatim}
Definition \texttt{P} (n:nat) :=
    match n return Prop with
    O => True | S k => False end.

match (e:0=1) in _=_y return \texttt{P} y with
    | eq_refl => I : \texttt{P} 0 (* \texttt{P} 0 = True *)
end : \texttt{P} 1 (* \texttt{P} 1 = False *)
\end{verbatim}

Tactic \texttt{discriminate}:

- solves goals of the form \texttt{C \, t_1...t_n <> D \, u_1...u_k}
- \texttt{discriminate H} solves the goal when

\begin{verbatim}
H : \texttt{C \, t_1...t_n = D \, u_1...u_k}
\end{verbatim}
Vectors (Lists with size)

Inductive type with parameters and index:

\[
\text{Inductive } \text{vect} \ (A: \text{Type}) \ : \ \text{nat} \to \ \text{Type} \ :=
\begin{array}{l}
\mid \text{niln} : \text{vect} \ A \ O \\
\mid \text{consn} : \\
\quad A \to \forall \ n: \text{nat}, \ \text{vect} \ A \ n \to \ \text{vect} \ A \ (S \ n).
\end{array}
\]

\textit{which defines}

- a family of types-predicates:
  \[
  \Gamma \vdash \text{vect} : \text{Type} \to \text{nat} \to \text{Type}
  \]
- a set of introduction rules for the types in this family

\[
\Gamma \vdash A : \text{Type} \\
\hline
\Gamma \vdash \text{niln}_A : \text{vect} \ A \ O
\]

\[
\Gamma \vdash A : \text{Type} \quad \Gamma \vdash a : A \quad \Gamma \vdash n : \text{nat} \quad \Gamma \vdash l : \text{vect} \ A \ n \\
\hline
\Gamma \vdash \text{consn}_A \ a \ n \ l : \text{vect} \ A \ (S \ n)
\]
Vectors : elimination

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

\[
\Gamma \vdash v : \text{vect } A \ n \quad \Gamma, p : \text{nat}, x : \text{vect } A \ p \vdash C(p, x) : s \\
\Gamma \vdash t_1 : C(O, \text{nil}_A) \\
\Gamma, a : A, n : \text{nat}, l : \text{vect } A \ n \vdash t_2 : C(S \ n, \text{cons}_A \ a \ n \ l)
\]

\[
\Gamma \vdash \left(\begin{array}{l}
\text{match } v \text{ as } x \text{ in } \text{vect } _{-} \ p \text{ return } C(p, x) \text{ with }
\text{niln } \Rightarrow t_1 \mid \text{consn } a \ n \ l \Rightarrow t_2 \\
\text{end}
\end{array}\right) : C(n, v)
\]

- and the obvious reduction rules (\(\iota\)-reduction)
Well-formed inductive definitions
Constructors of the inductive definition $I$ have type:

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

where $C_i$ can feature instances of $I$. Question: can these instances be arbitrary?
Constructors of the inductive definition \( I \) have type:

\[
\kappa : \forall (z_1 : C_1) \ldots (z_k : C_k). I \ a_1 \ldots a_n
\]

where \( C_i \) can feature instances of \( I \).

Question: can these instances be arbitrary? No!

Example:

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

Constructors of the inductive definition \( I \) have type:

\[
\kappa : \forall (z_1 : C_1) \ldots (z_k : C_k) . I \ a_1 \ldots a_n
\]

where \( C_i \) can feature instances 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 \( \Omega \) loops.
Things can even be worse:

\[
\text{Inductive } \lambda \text{ : Type := } \\
| \text{Lam : (lambda } \rightarrow \text{ lambda) } \rightarrow \text{ lambda} \\
\]

Now define:

\[
\text{Fixpoint } \lambda\text{to}_\text{nat} \ (t \ : \ \lambda\text{ :nat} : \\
\text{match } t \ \text{with Lam } f \rightarrow S \ (\lambda\text{to}_\text{nat} (f \ t)) \ \text{end.}
\]
Things can even be worse:

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

Now define:

```latex
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 \((\kappa_i)_{1 \leq i \leq n}\) of constructors.
- We can see it as \(\mu X, \bigoplus_{1 \leq i \leq n} \kappa_i(X)\) (with \(\mu\) 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 \(\lambda X, \bigoplus_{1 \leq i \leq n} \kappa_i(X)\) is monotonic.
  
  \(\mu X : \mathcal{P}.(X \rightarrow A) \rightarrow A\) has a fixpoint...

- In order both to ensure monotonicity and to avoid paradox (predicativity of \(\text{Type}\)), 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:

\[ c : C_1 \rightarrow \ldots \rightarrow C_k \rightarrow I \ a_1 \ldots a_k \]

it is well-formed when:

- \( I \ a_1 \ldots a_k \) is well-formed w.r.t. the uniformity of parametric arguments and typing constraints;
- \( I \) does not appear in any of the \( a_1, \ldots, a_k \);
- Each \( C_i \) should either not refer to \( I \) or be of the form:

\[ C'_1 \rightarrow \ldots \rightarrow C'_m \rightarrow I \ b_1 \ldots b_k \]

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 $X : \text{Type}$ with an embedding $\text{Type} \to X$

\[
\text{Inductive } e \ (A:s1) : s2 := C \ (_:A).
\]

- $C : A \to e(A)$
- 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)
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:

- ⊢ t : I q b
- a : A[q/p], h : I q a ⊢ P : s'
- x₁ : C₁[q/p], ..., xₙ : Cₙ[q/p] ⊢ e : P(u[q/p], κ q x₁...xₙ)

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 choosing 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

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

- Typing

$$f : (\forall (x : A), B(x)), x : A \vdash t : B(x)$$

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

\[
\begin{align*}
\lambda P : \text{nat} & \rightarrow s, \\
\lambda H_O : P(O), \\
\lambda H_S : \forall m : \text{nat}, P(m) & \rightarrow P(S\ m), \\
\text{fix } f (n : \text{nat}) : P(n) & := \\
\text{match } n \text{ as } y & \text{ return } P(y) \text{ with} \\
O & \Rightarrow H_O \mid S\ m & \Rightarrow H_S\ m\ (f\ m) \\
\text{end}
\end{align*}
\]

is correct with respect to CCI: recursive call on \( m \) which is structurally smaller than \( n \) in the inductive \( \text{nat} \).
Fixpoint operator : typing rules

\[
\Gamma \vdash l : s \quad \Gamma, x : l \vdash C : s' \quad \Gamma, x : l, f : (\forall x : l, C) \vdash t : C \\
\quad \Gamma \vdash (\text{fix } f (x : l) : C := t) : \forall x : l, C
\]

the main rules for \( t|_f^\rho <_l x \) are:

\[
\begin{align*}
& z \in \rho \cup \{x\} \quad (u_i|_f^\rho <_l x)_{i=1...n} \quad A|_f^\rho <_l x \quad (t_i|_f^\rho \cup \{x \in \bar{x}|x:U/1\bar{u}\} <_l x)_i \\
& \text{match } z \ u_1 \ldots u_n \text{ return } A \text{ with } (c_i \ x_i \Rightarrow t_i)_i \text{ end}|_f^\rho <_l x
\end{align*}
\]

\[
\begin{align*}
& t \neq (z \ \bar{u}) \text{ for } z \in \rho \cup \{x\} \quad t|_f^\rho <_l x \quad A|_f^\rho <_l x \quad (t_i|_f^\rho <_l x)_i \\
& \text{match } t \text{ return } A \text{ with } (c_i \ x_i \Rightarrow t_i)_i \text{ end}|_f^\rho <_l x
\end{align*}
\]

\[
\begin{align*}
& y \in \rho \\
& f (y \ u_1 \ldots u_n)|_f^\rho <_l x \\
& f \not\in \text{FV}(t) \quad t|_f^\rho <_l x
\end{align*}
\]

+ 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.

\[
\begin{align*}
\lambda P : \text{list } A &\to s, \\
\lambda f_1 : P \text{nil}, \\
\lambda f_2 : \forall (a : A)(l : \text{list } A), P \, l &\to P (\text{cons } a \, l), \\
\text{fix } Rec (x : \text{list } A) : P \, x &:= \\
&\text{match } x \text{ return } P \, x \text{ with} \\
&\quad \text{nil } \Rightarrow f_1 | (\text{cons } a \, l) \Rightarrow f_2 a \, l (Rec \, l) \\
&\text{end}
\end{align*}
\]

- has type

\[
\begin{align*}
\forall P : \text{list } A &\to s, \\
P \text{nil}, &\to \\
(\forall (a : A)(l : \text{list } A), P \, l &\to P (\text{cons } a \, l)) \to \\
\forall (x : \text{list } A), P \, x
\end{align*}
\]
Remarks on the criteria

Possibility of recursive call on deep subterms

\[
\text{Fixpoint } \text{mod2} \ (n : \text{nat}) \ : \ \text{nat} \ := \\
\quad \text{match } n \ \text{with} \ O \Rightarrow O \ | \ S \ O \Rightarrow S \ O \\
\quad \quad \mid S \ (S \ x) \Rightarrow \text{mod2} \ x
\]

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

\[
\text{Definition } \text{pred} \ (n : \text{nat}) \ : \ n \nless \ 0 \Rightarrow \text{nat} := \\
\quad \text{match } n \ \text{return} \ n \nless \ 0 \Rightarrow \text{nat} \ with \\
\quad \quad \quad \mid S \ p \Rightarrow (\text{fun } (h : S \ p \nless \ 0) \Rightarrow p) \\
\quad \quad \quad \mid O \Rightarrow (\text{fun } (h : 0 \nless \ 0) \Rightarrow \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{match } h \ (\text{refl_equal} \ 0) \ \text{return} \ \text{nat} \ with \ \text{end}) \\
\quad \text{end}
\]

\[
\text{Fixpoint } F \ (n : \text{nat}) \ : \ C \ := \\
\quad \text{match } \text{iszero} \ n \ \text{with} \\
\quad \quad \mid \text{left} \ H \ (\text{H:n=O}) \Rightarrow \ldots \\
\quad \quad \mid \text{right} \ H \ (\text{H:n\nless\ 0}) \Rightarrow F \ (\text{pred} \ n \ H)
\quad \text{end}
\]
Remarks on the criteria

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

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

\[
  f (c \text{id}) \xrightarrow{\lambda} f (\text{id Singl (c \text{id})}) \xrightarrow{\beta} f (c \text{id})
\]
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:

\[ \forall P : T \to \text{Type}, \ldots \to \forall t' : T, P t' \]

- 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_{\text{ind}}, T_{\text{rect}}, \ldots`) 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:

\[
\text{(fix } f \ (x:A): B \ x := t(f,x)\text{)} e \\
\rightarrow t(\text{fix } f \ (x:A): B \ x := t(f,x), \ e)
\]
We would expect the usual expansion rule for fixpoints:

\[
(fix \ f \ (x:A): \ B \ x := t(f,x))e \\
\rightarrow t(fix \ f \ (x:A): \ B \ x := t(f,x), \ e)
\]

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

We would expect the usual expansion rule for fixpoints:

\[
(fix \ f \ (x:A): B \ x := t(f,x)) \ e \\
\rightarrow t(fix \ f \ (x:A): B \ x := t(f,x), \ e)
\]

... 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