MPRI 2-7-2: Proof Assistants

Bruno Barras, Matthieu Sozeau

Jan 12, 2017
Simple inductive types (datatypes):

\textbf{Inductive} \texttt{nat} : \texttt{Type} := \texttt{O} : \texttt{nat} | \texttt{S} : \texttt{nat}\rightarrow\texttt{nat}.
\textbf{Inductive} \texttt{bool} := \texttt{true} | \texttt{false}.
\textbf{Inductive} \texttt{list} (\texttt{A:Type}) : \texttt{Type} :=
  \texttt{nil} | \texttt{cons} (\texttt{hd:A}) (\texttt{tl:list A}).
\textbf{Inductive} \texttt{tree} (\texttt{A:Type}) :=
  \texttt{leaf} | \texttt{node} (\texttt{_:A}) (\texttt{_:nat}\rightarrow\texttt{tree A}).

Smallest type closed by introduction rules (constructors)

\textbf{Parameters}: \texttt{cons} : \texttt{forall A:Type, A -> list A -> list A}
\textbf{Coq prelude}: \texttt{cons 0 nil : list nat}
Generated elimination scheme (not primitive):

\[
\text{nat\_rect}
\]

\[
: \forall P : \text{nat} \to \text{Type},
\]

\[
P \ O \to (\forall n, P \ n \to P \ (S \ n)) \to
\forall n, P \ n.
\]

\[
:= \text{fun } P \ h0 \ hS \Rightarrow \text{fix } F \ n \Rightarrow
\]

\[
\text{match } n \ \text{return } P \ n \ \text{with}
\]

\[
| O \Rightarrow h0
\]

\[
| S \ k \Rightarrow hS \ k \ (F \ k)
\]

end

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

Inductive families
  Predicate defined by inference rules
  Definition of equality
  Vectors

Non-uniform parameters

Theory of Inductive types
  Strict Positivity
  Dependent pattern-matching
  Guarded fixpoint
  The guardedness check
Limitations of parameters

Defining a predicate:

```
Inductive even (n:nat) : Prop :=
  even_i (half:nat) (_:half+half=n).
```

Inductive types with parameters are some kind of “template”

```
Inductive listnat :=
  nilnat | consnat (_:nat) (_:listnat).
Inductive listbool :=
  nilbool | consbool (_:bool) (_:listbool).
```

No dependency between both types.

But in the definition of `even:nat->Prop` as an inductive type/set

```
E_0:even 0
```

even \((S\ (S\ 0))\) depends on even 0.
Inductive families

Family = indexed type

\( P : \text{nat} \to \text{Type} \) represents the type family \((P(n))_{n \in \mathbb{N}}\)

Inductive family:

- Constructors do not inhabit uniformly the members of the family
- Recursive arguments can change the value of the index

Even numbers:

\[
\text{Inductive even : nat} \to \text{Prop} := \\
\quad \text{E0 : even } 0 \\
\mid \text{ESS (n:nat) (e:even n) : even (S (S n))}.
\]

Syntax very close to inference rules!
Elimination scheme

Elimination scheme: minimality of predicate, rule-induction

even_ind : forall (P:nat->Prop),
P O -> (forall n, P n -> P (S (S n))) ->
forall n, even n -> P n.

Seems the analogous of nat’s dependent scheme
Elimination scheme: minimality of predicate, rule-induction

even_ind : forall (P:nat->Prop),
  P O -> (forall n, P n -> P (S (S n))) ->
  forall n, even n -> P n.

Seems the analogous of nat’s dependent scheme

Even’s dependent scheme (refers to constructors E0 and ESS):

forall (P : forall n, even n -> Prop),
P 0 E0 ->
(foreall n (e:even n), P n e -> P (S (S n)) (ESS n e)) ->
forall n (e:even n), P n e

Definable in Coq, but not automatically generated (why? wait and see...)
Defining the dependent elimination scheme

Even more complex return clause: in

\[
\text{Definition even\_ind\_dep } (P:\forall n, \text{even } n \rightarrow \text{Prop}) \\
\text{ (h0:} P \ 0 \ \text{E0)} \\
\text{ (hSS:} \forall n \ e, P \ n \ e \rightarrow P \ (S \ (S \ n)) \ (\text{ESS } n \ e)) \\
: \forall n, \text{even } n \rightarrow P \ n := \\
\text{fix } F \ n \ e := \\
\text{match } e \ \text{as } e' \ in \ \text{even } k \ return \ P \ k \ e' \ with \\
| \ E0 \Rightarrow \ h0 : \ P \ 0 \ \text{E0} \\
| \ \text{ESS } k \ e' \Rightarrow \ \\
\ \text{hSS } k \ e' \ (F \ k \ e') : \ P \ (S \ (S \ k)) \ (\text{ESS } k \ e') \\
\text{end}
\]

\text{Notation as } e' \ in \ \text{even } k \ return \ P \ k \ e' \ is \ just \ a \ way \ to \ write \ the \ term \ \text{fun } k \ e' \Rightarrow P \ k \ e'.

Becomes natural with time...
Equality: the paradigmatic indexed family

Propositional equality is defined as:

\[
\text{Inductive } \text{eq} \ (A : \text{Type}) \ (a : A) : A \to \text{Prop} := \\
\text{eq}\_\text{refl} : \text{eq} \ A \ a \ a.
\]

Notation "\(x = y\)" := (\(\text{eq} \ x \ y\)).

Its dependent elimination principle is of the form:

\[
\begin{align*}
\Gamma \vdash e : \text{eq} \ A \ t \ u & \quad \Gamma, y : A, e' : \text{eq} \ A \ t \ y \vdash C(y, e') : s \\
\Gamma & \vdash t : C(t, \text{eq}\_\text{refl}_{A,t})
\end{align*}
\]

\[
\Gamma \vdash \left( \begin{array}{c}
\text{match } e \text{ as } e' \text{ in } \text{eq\_y} \ \text{return } C(y, e') \text{ with } \\
\text{eq}\_\text{refl} \Rightarrow t \\
\text{end}
\end{array} \right) : C(u, e)
\]
Tactics related to equality

Tactics:

- **f_equal** (congruence) \( \frac{x=y}{f(x) = f(y)} \)
- **discriminate** (constructor discrimination) \( \frac{C(t_1, \ldots, t_n) = D(u_1, \ldots, u_k)}{A} \)
- **injection** (injectivity of constructors) \( \frac{C(t_1, \ldots, t_n) = C(u_1, \ldots, u_n)}{t_1 = u_1 \ldots t_n = u_n} \)
- **inversion** (necessary conditions) \( \frac{\text{even } (S(Sn))}{\text{even } n} \)
- **rewrite** (substitution) \( \frac{x=y}{P(y)} \)
- **symmetry, transitivity**
Inductive types with parameters and index

Example of vectors with size

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:
    Γ ⊢ vect : Type → nat → Type
▶ a set of introduction rules for the types in this family

Γ ⊢ A : Type
    Γ ⊢ niln_A : vect A O

Γ ⊢ A : Type  Γ ⊢ a : A  Γ ⊢ n : nat  Γ ⊢ l : vect A n
Γ ⊢ consn_A a n l : list A (S n)
Inductive types with parameters and index

vectors : elimination

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

\[
\Gamma \vdash v : vect A n \\
\Gamma, m : nat, x : vect A m \vdash C(m, x) : s \\
\Gamma \vdash t_1 : C(O, \text{nil}_n A) \\
\Gamma, a : A, n : nat, l : vect A n \vdash t_2 : C(S n, \text{cons}_n A a n l) \\
\Gamma \vdash \left( \begin{array}{l}
\text{match } v \text{ as } x \text{ in } vect \_ p \text{ return } C(p, x) \text{ with } \\
\text{ niln } \Rightarrow t_1 \mid \text{ consn } a n l \Rightarrow t_2 \\
\text{end}
\end{array} \right) : C(n, v)
\]
Inductive types with parameters and index

- reduction rules preserve typing ($\iota$-reduction)

\[
\begin{align*}
\text{match } \text{niln}_A \text{ as } x \text{ in } \text{vect}_p & \text{ return } C(x, p) \text{ with} \\
\text{niln} & \Rightarrow t_1 | \text{consn } a \ n \ l & \Rightarrow t_2 \\
\text{end} \\
\rightarrow_\iota & t_1
\end{align*}
\]

\[
\begin{align*}
\text{match } \text{consn}_A a' \ n' \ l' \text{ as } x \text{ in } \text{vect}_p & \text{ return } C(x, p) \text{ with} \\
\text{niln} & \Rightarrow t_1 | \text{consn } a \ n \ l & \Rightarrow t_2 \\
\text{end} \\
\rightarrow_\iota & t_2[a', n', l'/a, n, l]
\end{align*}
\]
Non-uniform parameters

Non-uniform parameter:

- Like parameters: uniform conclusion
- Like indices: value can change in recursive subterms

\[\text{Inductive } \text{tuple } (A:Type) := \]
\[| \text{H0} (_:A) \]
\[| \text{HS} (_:\text{tuple} (A*A)). \]

\[\text{Definition } t4 : \text{tuple} \text{ nat} := \]
\[\text{HS nat} (\text{HS} (\text{nat*nat}) (\text{H0} _ ((1,2),(3,4)))).\]
Elimination rules

Pattern-matching:

\[
\begin{align*}
\Gamma & \vdash e : \text{tuple } A & \Gamma, h : \text{tuple } A & \vdash P(h) : s \\
\Gamma, x : A & \vdash t_0 : P(H0 A x) & \Gamma, h : \text{tuple}(A \times A) & \vdash t_S : P(HS A h)
\end{align*}
\]

\[
\Gamma \vdash \left( \begin{array}{l}
\text{match } e \text{ as } h \text{ return } P(h) \text{ with } \\
\quad H0 x \Rightarrow t_0 \\
\quad HS h \Rightarrow t_S \\
\text{end}
\end{array} \right) : P(e)
\]

Elimination:

tuple_rect :
\[
\text{forall } (P : \text{forall } A, \text{ tuple } A \rightarrow \text{ Type}), \\
\quad (\text{forall } A x, P A (H0 A x)) \rightarrow \\
\quad (\text{forall } A h, P (A \times A) h \rightarrow P A (HS A h)) \rightarrow \\
\quad \text{forall } A (h : \text{tuple } A), P A h.
\]

Non-uniform parameters:

- In pattern-matching, behaves like a parameter
- In recursive principles, behaves like an index
Non-uniform parameters can encode inductive families:

```plaintext
Inductive even (n:nat) : Prop :=
  E0' (n:=0)
| ESS' (k:nat) (e:even k) (n:=S (S k)).
Definition E0 : even 0 := E0' 0 eq_refl.
Definition ESS n e : even (S (S n)) :=
  ESS' (S (S n)) n e eq_refl.
```
Well-formed inductive definitions
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 instances of $I$.
Question: can these instances be arbitrary?
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 instances of $I$.

Question: can these instances be arbitrary?

Example:

```hs
Inductive lambda : Type :=
| Lam : (lambda -> lambda) -> lambda
```
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 instances of $I$.
Question: can these instances be arbitrary?
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.
Necessity of restrictions

Things can even be worse:

```coq
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:

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

Now define:

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

What happens with \((\text{lambda\_to\_nat}\ (\text{Lam}\ (\text{fun}\ x\ \Rightarrow\ x)))\)?
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 \(\mu X, \bigoplus_{1 \leq i \leq n} \Gamma_i(X)\) (with \(\mu\) a fixpoint operator on types).
- The existence of this smallest type can be proved at the impredicative level when the operator \(\lambda X, \bigoplus_{1 \leq i \leq n} \Gamma_i(X)\) is monotonic.
- In order both to ensure monotonicity and to avoid paradox, 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:

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

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:

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

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
Dependent pattern-matching

\[\text{Inductive } I \ (p: \text{Par}) : A \rightarrow s := \]
\[\ldots \mid \Gamma \ (x_1:C_1)\ldots(x_n:C_n) : I \ p \ u \]
\[\mid \ldots\]

\[\text{match } t \text{ as } h \text{ in } I \ a \ \text{return } P(a,h) \text{ with }\]
\[\ldots\]
\[\mid \Gamma \ x_1 \ \ldots \ x_n \Rightarrow e \]
\[\ldots\]
\[\text{end}\]

Typing conditions:

\[\because \quad \vdash t : I q a\]

\[\because \quad a : A[q/p], \ h : I q a \vdash P : s'\]

\[\because \quad x_1 : C_1[q/p], \ldots, x_n : C_n[q/p] \vdash e : P(u[q/p], \Gamma q x_1\ldots x_n)\]

Then the match has type \( P(a,t) \)
Tactics for case analysis

- \texttt{case t} is the most primitive. It:
  - generates a (proof) term of the form \texttt{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 \texttt{case_eq} variant modifies the guessing of the return type so that equalities are generated.

- The \texttt{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

\[
\frac{f : (\forall (x : A), B(x)), x : A \vdash t : B(x)}{\vdash (\text{fix } f (x : A) : B(x) := t(f, x)) : \forall (x : A), B(x)}
\]
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 \textit{structurally} smaller

Example of recursor on natural numbers

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

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

\[ \Gamma \vdash l : s \quad \Gamma, x : A \vdash C : s' \quad \Gamma, x : l, f : (\forall x : l, C) \vdash t : C \quad t|_f^0 <_l x \]

\[ \Gamma \vdash (\text{fix } f (x : l) : C := t) : \forall x : l, C \]

the main definition of \( t|_f^\rho <_l x \) are:

\[ z \in \rho \cup \{ x \} \quad (u_i|_f^\rho <_l x)_{i=1\ldots n} \quad A|_f^\rho <_l x \quad (t_i|_f^\rho \{ x \in \bar{x}_i | x : \forall y : U. \bar{u} \} <_l x)_i \]

match \( z u_1 \ldots u_n \) return \( A \) with \( \ldots c_i \bar{x}_i \Rightarrow t_i \ldots \) end \( |_f^\rho <_l x \)

\[ t \neq (z \bar{u}) \text{ for } z \in \rho \cup \{ x \} \quad t|_f^\rho <_l x \quad A|_f^\rho <_l x \quad \ldots t_i|_f^\rho <_l x \quad \ldots \]

match \( t \) return \( A \) with \( \ldots c_i \bar{x}_i \Rightarrow t_i \ldots \) end \( |_f^\rho <_l x \)

\[ y \in \rho \quad f \not\in t \]

\[ f y|_f^\rho <_l x \quad t|_f^\rho <_l x \]

+ 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 immediate subterms.

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

- has type

\[
\forall P : \text{list } A \rightarrow s, \\
P \text{ nil,} \rightarrow \\
(\forall (a : A)(l : \text{list } A), P l \rightarrow P (\text{cons } a l)) \rightarrow \\
\forall (x : \text{list } A), P x
\]
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:

```
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:n=0)) => ...
    | (right (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)

\[
\text{Inductive Singl} \ (A:\text{Prop}) : \text{Prop} := c : A \to \text{Singl} \ A.
\]
\[
\text{Definition ID} : \text{Prop} := \forall (A:\text{Prop}), A \to A.
\]
\[
\text{Definition id} : ID := \text{fun} \ A \ x \ => \ x.
\]
\[
\text{Fixpoint} \ f \ (x : \text{Singl ID}) : \text{bool} :=
\]
\[
\text{match} \ x \ \text{with} \ (c \ a) \ => \ f \ (a \ (\text{Singl ID} \ (c \ ID \ id))) \ \text{end}.
\]

\[
f (c \ ID \ id) \rightarrow f (id \ (\text{Singl ID} \ (c \ ID \ id))) \rightarrow f (c \ ID \ id)
\]
Tactics for induction

\texttt{fix \(<n>\), where \(<n>\) is a numeral is the most primitive. It:}

- generates a (proof) term of the form:
  \[
  \text{fun } g1 \ g2 \Rightarrow \text{fix } f \ h1 \ h2 \ t \ h3 \ \{\text{struct } t\} := \ ?F \ h1 \ h2 \ t
  \]

  \text{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 \(t\);

\texttt{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...
We would expect the usual expansion rule for fixpoints:

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

We would expect the usual expansion rule for fixpoints:

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

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

We would expect the usual expansion rule for fixpoints:

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

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

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