MPRI 2-7-2: Proof Assistants

Bruno Barras, Matthieu Sozeau

Jan 12, 2017

Recap: Inductive Types and Elimination Rules

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

Logical connectives

Logical connecctives 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

$$\frac{}{\texttt{E}_0\texttt{:even 0}} \qquad \frac{\texttt{e:even n}}{\texttt{E}_{SS}\texttt{(e):even (S (S n))}}$$

`even (S (S O))` depends on `even O`.

Inductive families

Family = indexed type

`P : nat -> 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:

```
Inductive even : nat -> Prop :=
  E0 : even O
| 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

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 ->
 (forall 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

```
Definition even_ind_dep (P:forall n , even n -> Prop)
  (h0:P 0 E0)
  (hSS:forall n e, P n e -> P (S (S n)) (ESS n e))
  : forall n, even n -> P n :=
  fix F n e :=
  match e as e' in even k return P k e' with
  | E0 => h0 :       P 0 E0
  | ESS k e' =>
     hSS k e' (F k e') : P (S (S k)) (ESS k e')
  end
```

Notation `as e' in even k return P k e'` is just a way to write the term `fun k e' => P k e'`.
Becomes natural with time...

Equality: the paradigmatic indexed family

Propositional equality is defined as:

```
Inductive eq (A : Type) (a : A) : A -> Prop :=
  eq_refl : eq A a a.
Notation "x = y" := (eq x y).
```

Its dependent elimination principle is of the form:

$$\frac{\Gamma \vdash e : eq\ A\ t\ u \qquad \Gamma, y : A, e' : eq\ A\ t\ y \vdash C(y, e') : s \qquad \Gamma \vdash t : C(t, \mathrm{eq\_refl}_{A,t})}{\Gamma \vdash \left(\begin{array}{l}\texttt{match}\ e\ \texttt{as}\ e'\ \texttt{in}\ eq\ \_\ y\ \texttt{return}\ C(y, e')\ \texttt{with} \\ \quad \mathrm{eq\_refl} \Rightarrow t \\ \texttt{end}\end{array}\right) : C(u, e)}$$

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) $\frac{C(t_1,\ldots,t_n)=C(u_1,\ldots,u_n)}{t_1=u_1 \quad \ldots \quad t_n=u_n}$
- `inversion` (necessary conditions) $\frac{\text{even } (S(Sn))}{\text{even } n}$
- `rewrite` (substitution) $\frac{x=y \quad P(y)}{P(x)}$
- `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:
  $\Gamma \vdash \mathit{vect} : \mathbf{Type} \to \mathit{nat} \to \mathbf{Type}$
- a set of introduction rules for the types in this family

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

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

Inductive types with parameters and index

vectors : elimination

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

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

Inductive types with parameters and index

- reduction rules preserve typing (ι-reduction)

$$\left(\begin{array}{l}\texttt{match}\ \texttt{niln}_A\ \texttt{as}\ x\ \texttt{in}\ \textit{vect}\_p\ \texttt{return}\ C(x,p)\ \texttt{with} \\ \quad \texttt{niln} \Rightarrow t_1 \mid \texttt{consn}\ a\ n\ l \Rightarrow t_2 \\ \texttt{end}\end{array}\right)$$

$$\rightarrow_\iota \quad t_1$$

$$\left(\begin{array}{l}\texttt{match}\ \texttt{consn}_A\ a'\ n'\ l'\ \texttt{as}\ x\ \texttt{in}\ \textit{vect}\_p\ \texttt{return}\ C(x,p)\ \texttt{with} \\ \quad \texttt{niln} \Rightarrow t_1 \mid \texttt{consn}\ a\ n\ l \Rightarrow t_2 \\ \texttt{end}\end{array}\right.$$

$$\rightarrow_\iota \quad t_2[a', n', l' / a, n, l]$$

Non-uniform parameters

Non-uniform parameter:

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

```
Inductive tuple (A:Type) :=
| H0 (_:A)
| HS (_:tuple (A*A)).

Definition t4 : tuple nat :=
  HS nat (HS (nat*nat) (H0 _ ((1,2),(3,4)))).
```

Elimination rules

Pattern-matching:

$$\frac{\Gamma \vdash e : \mathit{tuple}\ A \quad \Gamma, h : \mathit{tuple}\ A \vdash P(h) : s \qquad \Gamma, x : A \vdash t_0 : P(H0\ A\ x) \quad \Gamma, h : \mathit{tuple}(A * A) \vdash t_S : P(HS\ A\ h)}{\Gamma \vdash \left(\begin{array}{l}\texttt{match}\ e\ \texttt{as}\ h\ \texttt{return}\ P(h)\ \texttt{with} \\ \qquad H0\ x \Rightarrow t_0 \\ \quad \mid \ HS\ h \Rightarrow t_S \\ \texttt{end}\end{array}\right) : P(e)}$$

Elimination:

```
tuple_rect :
  forall (P:forall A, tuple A -> Type),
  (forall A x, P A (H0 A x)) ->
  (forall A h, P (A*A) h -> P A (HS A h)) ->
  forall A (h:tuple A), P A h.
```

Non-uniform parameters:

- In pattern-matching, behaves like a parameter
- In recursive principles, behaves like an index

Encoding inductive families

Non-uniform parameters can encode inductive families:

```
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

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?
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?
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).
- 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.
- 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 : 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

Dependent pattern-matching

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

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

Typing conditions:

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

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

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

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

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

Fixpoint operator : typing rules

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

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

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

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

$$\frac{y \in \rho}{f\ y|_f^{\rho} <_I x} \qquad \frac{f \notin t}{t|_f^{\rho} <_I x}$$

+ 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 immediate subterms.

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

- has type

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

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:

```
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=O)) => ...
  | (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)

```
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:

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

Fixpoint expansion

We would expect the usual expansion rule for fixpoints:

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

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

Fixpoint expansion

We would expect the usual expansion rule for fixpoints:

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

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

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