Efficient and automatic recognition of mathematical structures in Coq

Matthias Puech

Laboratoire d’Informatique de l’Ecole Polytechnique, dir. Hugo Herbelin

October 31 2008
My view of Coq

- High-level tactical language ($\mathcal{L}_{tac}$)
- Low-level proof/type language (CIC)

Some tactics rely on mathematical structures
My view of Coq

- High-level tactical language ($\mathcal{L}_{tac}$)
- Low-level proof/type language (CIC)

Some tactics rely on mathematical structures

Define, Declare

- Define the object
- Let the system know they fulfill some properties
Example

Definition R := [...] .
Lemma R_refl : forall A, reflexive R A.
Lemma R_sym : forall A, symmetric R A.
Lemma R_trans : forall A, transitive R A.
Example

Definition R := [...] .
Lemma R_refl : forall A, reflexive R A.
Lemma R_sym : forall A, symmetric R A.
Lemma R_trans : forall A, transitive R A.

Add Parametric Relation x1 x2 : (A x1 x2) R
t reflexivity proved by R_refl
symmetry proved by R_sym
transitivity proved by R_trans
as R_rel.
Example

Definition R := [...].
Lemma R_refl : forall A, reflexive R A.
Lemma R_sym : forall A, symmetric R A.
Lemma R_trans : forall A, transitive R A.

Add Parametric Relation x1 x2 : (A x1 x2) R
  reflexivity proved by R_refl
  symmetry proved by R_sym
  transitivity proved by R_trans
as R_rel.

Lemma foo : [...].
Proof. transitivity y; reflexivity. Qed.
Definition R := [...].
Lemma R_equiv : equivalence R.

Lemma foo : [...].
Proof. transitivity y; reflexivity. Qed.
Questions

▶ Can Coq have a «consciousness» of these structures?
▶ Can they be automatically inferred?

More generally

▶ A step towards inferring “trivial reasoning steps”?
Typeclasses

Ad-hoc polymorphism
Overloading function names, depending on the context.

Example

\[(+) : \text{int} \rightarrow \text{int} \rightarrow \text{int} = \text{plus\_int}\]

\[(+) : \text{float} \rightarrow \text{float} \rightarrow \text{float} = \text{plus\_float}\]

\[(+) : \text{string} \rightarrow \text{string} \rightarrow \text{string} = \text{concat}\]
Typeclasses

Principle
Relation between:

**Classes** (specification of the functions to overload)

**Instances** (actual implementations in different contexts)

- Relation between classes also (inheritance)
- Overloading resolution à la PROLOG
Class Addable (A:Type) :=
  (+) : A -> A -> A.

Instance ex1 : Addable nat :=
  (+) := Peano.plus.

Instance ex2 : Addable Z :=
  (+) := ZArith.Zplus.
Class Monoid (A : Type) :=
  (⋆) : A -> A -> A;
  assoc : forall a b c, (a ⋆ b) ⋆ c = a ⋆ (b ⋆ c);
  e : A;
  ident_l : forall a, a ⋆ e = a;
  ident_r : forall a, e ⋆ a = a.

Class [Monoid A] => Group :=
  inverse : forall x:A, exists y:A, 
x * y = y * x = e.
First class implementation (almost only syntactic sugar)

- Class $\Rightarrow$ Record type
- Instance $\Rightarrow$ (dependent) Record
- Overloaded method $\Rightarrow$ Field
- Parent class $\Rightarrow$ inferred argument
- Resolution $\Rightarrow$ eauto
First class implementation (almost only syntactic sugar)

<table>
<thead>
<tr>
<th>Term</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>⇒ Record type</td>
</tr>
<tr>
<td>Instance</td>
<td>⇒ (dependent) Record</td>
</tr>
<tr>
<td>Overloaded method</td>
<td>⇒ Field</td>
</tr>
<tr>
<td>Parent class</td>
<td>⇒ inferred argument</td>
</tr>
<tr>
<td>Resolution</td>
<td>⇒ eauto</td>
</tr>
</tbody>
</table>

Structure recognition ⇒ Instance search
## Under the hood

**First class implementation (almost only syntactic sugar)**

<table>
<thead>
<tr>
<th>Term</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>$\Rightarrow$ Record type</td>
</tr>
<tr>
<td>Instance</td>
<td>$\Rightarrow$ (dependent) Record</td>
</tr>
<tr>
<td>Overloaded method</td>
<td>$\Rightarrow$ Field</td>
</tr>
<tr>
<td>Parent class</td>
<td>$\Rightarrow$ inferred argument</td>
</tr>
<tr>
<td>Resolution</td>
<td>$\Rightarrow$ eauto</td>
</tr>
</tbody>
</table>

**Structure recognition**

- $\Rightarrow$ Instance search
- $\Rightarrow$ Proof search
Instance search

Given a class $C = \langle x_1 : T_1, \ldots, x_n : T_n \rangle$, we’re searching for $i : C$.

Decidable subset of the type system:

$$
\begin{align*}
&\frac{(\text{var})}{\Gamma, t : T \vdash t : T} \\
&\frac{(\text{inst})}{\Gamma \vdash t_1 : T_1 \quad \cdots \quad \Gamma \vdash t_n : T_n} \\
&\qquad \frac{}{\Gamma \vdash \langle t_1 : T_1 \quad \cdots \quad t_n : T_n \rangle} \\
&\frac{}{\Gamma \vdash \forall x : T \cdot \langle t_1 : T_1 \quad \cdots \quad (t_i x) : T_i \quad \cdots \quad t_n : T_n \rangle} \\
&\frac{}{\Gamma \vdash \forall x : T \cdot \langle t_1 : T_1 \quad \cdots \quad (t_i x) : T_i \quad \cdots \quad t_n : T_n \rangle}
\end{align*}
$$
Overview of the algorithm

Combinatorial work
We are looking for all possible proofs of \( C \)

Textual recognition

- \( \forall \; x, \; \_ \; x \; x : \text{reflexivity lemma} \Rightarrow \text{instance of} \)
  
  Class Reflexive A (R:relation A) :=
  reflexive : forall x, R x x

- \( \forall \; a \; b \; c, \; \_ \; (\_ \; a \; b) \; c = \_ \; a \; (\_ \; b \; c) : \text{associativity lemma?} \). Maybe a monoid?

= Filtering on types
Welcome to Coq trunk (11262)

Coq <
Welcome to Coq trunk (11262)

Coq < Definition R := [...].
R is defined

Coq <
Welcome to Coq trunk (11262)

Coq < Definition R := [...].
R is defined

R_refl is defined
new instance Symmetric_1 : Symmetric R

Coq <
Welcome to Coq trunk (11262)

Coq < Definition R := [...].
R is defined

R_refl is defined
new instance Symmetric_1 : Symmetric R

R_trans is defined
new instance Transitive_1 : Transitive R
new instance PER_1 : PER R

Coq <
Usage example

Welcome to Coq trunk (11262)

Coq < Definition R := [...].
R is defined

R_refl is defined
new instance Symmetric_1 : Symmetric R

R_trans is defined
new instance Transitive_1 : Transitive R
new instance PER_1 : PER R

R_refl is defined
new instance Reflexive_1 : Reflexive R
new instance Equivalence_1 : Equivalence R
new instance Setoid_1 : Setoid A R
An efficient structure for one-to-many filtering.

The problem
Given an algebra of terms $\Lambda$, I have:

- A pattern $p$
- A (big) set of terms $S$

Which terms in $S$ filter the pattern $p$?
An efficient structure for one-to-many filtering.

The problem
Given an algebra of terms $\Lambda$, I have:

- A pattern $p$
- A (big) set of terms $S$

Which terms in $S$ filter the pattern $p$?
Discrimination nets

= A collection datastructure, with pattern searching.

▶ To a datatype, we associate a structure where each-node represents a list of sub-terms.
Discrimination nets

= A collection datastructure, with pattern searching.

▶ To a datatype, we associate a structure where each-node
represents a \textit{list} of sub-terms.

\textbf{Example}

\begin{verbatim}
type t =
  | Var of int
  | Lam of t
  | App of t * t
\end{verbatim}
= A collection datastructure, with pattern searching.

- To a datatype, we associate a structure where each-node represents a list of sub-terms.

Example

type t =
  | Var of int
  | Lam of t list
  | App of t list * t list
Discrimination nets

= A collection datastructure, with pattern searching.

- To a datatype, we associate a structure where each-node represents a *list* of sub-terms.
- At the leaf of the structure, we store unique identifiers

**Example**

type t =
   | Var of int
   | Lam of t list
   | App of t list * t list
Discrimination nets

= A collection datastructure, with pattern searching.

▶ To a datatype, we associate a structure where each-node represents a list of sub-terms.

▶ At the leaf of the structure, we store unique identifiers

Example

type t =
| Var of int * ident list
| Lam of t list
| App of t list * t list
Term Search

- To search for a term is to search for a path in the discrimination net.
Term Search

- To search for a term is to search for a path in the discrimination net.

Example

This net:

contains terms:
To search for a term is to search for a path in the discrimination net.

Example

This net:

contains terms:
Term Search

- To search for a term is to search for a path in the discrimination net.

Example

This net:

\[
\begin{array}{c}
\text{[Lam; App]} \\
\text{[Var 1; Lam]} & \text{[Lam]} & \text{[Lam]} \\
1 & 2 & 3 & 3 \\
\text{[Var 1]} & \text{[Var 1] [Var 1]} \\
2 & 3 \\
\end{array}
\]

contains terms:

1 Lam(Var 1)
2 Lam(Lam(Var 1))
To search for a term is to search for a path in the discrimination net.

**Example**

This net:

contains terms:

- 1 Lam(Var 1)
- 2 Lam(Lam(Var 1))
- 3 App(Lam(Var 1), Lam(Var 1))
To search for a term is to search for a path in the discrimination net.

Example

This net:

\[
\begin{array}{c}
\text{[Lam; App]} \\
\text{[Var 1; Lam]} & \text{[Lam]} & \text{[Lam]} \\
\text{[Var 1]} & \text{[Var 1]} & \text{[Var 1]} \\
\text{2} & \text{3} & \text{3} \\
\end{array}
\]

contains terms:

1. \(\text{Lam(Var 1)}\)
2. \(\text{Lam(Lam(Var 1))}\)
3. \(\text{App(Lam(Var 1), Lam(Var 1))}\)

\[\Rightarrow O(|T|)\]
To search for a pattern is to search for a term, stop at the holes and enumerate terms underneath.

Example

```
[Var 1; Lam] [Var 1] [Var 1] [Var 1]
1 2 3 4

[Lam] [Lam; Var 1]

[Lam; App]

```

App(X, Lam(Y)) = \bigcap\{3, 4\} = 3
To search for a pattern is to search for a term, stop at the holes and enumerate terms underneith.

Example

\[
\begin{array}{c}
\text{App}(X, \text{Lam}(Y)) ?
\end{array}
\]
To search for a pattern is to search for a term, stop at the holes and enumerate terms underneith.

Example

\[
\text{App}(X, \text{Lam}(Y)) \land \bigcap \{3, 4; 3\} = 3
\]
Current Implementation

Functor:

\[
\begin{align*}
\text{type } & t \\
\text{val } & \text{map} \\
\text{val } & \text{fold} \\
\text{val } & \text{compare}
\end{align*}
\rightarrow
\begin{align*}
\text{type } & \text{dn} \\
\text{val } & \text{add} \\
\text{val } & \text{find\_all} \\
\text{val } & \text{fold\_pattern}
\end{align*}
\]

Applied to Coq’s constr
Current Implementation

Functor:

- type t
- val map
- val fold
- val compare

\[ \rightarrow \]

- type dn
- val add
- val find_all
- val fold_pattern

Applied to Coq’s constr

Primitives:

- add
- find_all
- fold_pattern
Current Implementation

Functor:

```plaintext
type t
val map
val fold
val compare

→

type dn
val add
val find_all
val fold_pattern
```

Applied to Coq’s constr

Primitives:

- add
- find_all
- fold_pattern

Allow to code many typical search problems...
Current Implementation

Head search

let search_concl pat = 
    possibly_under prod_pat 
    (search_pat pat) all_types

Search for equalities

let search_eq_concl pat = 
    possibly_under prod_pat 
    (under (eq_pat) (search_pat pat) 
    ) all_types
Multiple variations

- Term/Pattern or Pattern/Term
- Full unification
- Filtering modulo $\delta, \beta \ldots$

Numerous applications

- Rewriting systems
- Efficient proof search (Hints)
- Interactive search tools
To go further

- Use discrimination nets pervasively
- Relax the textual recognition (isomorphisms of types)
- Unify with all the other proof search frameworks
To go further

- Use discrimination nets pervasively
- Relax the textual recognition (isomorphisms of types)
- Unify with all the other proof search frameworks

But also,

- Typeclasses were just a pretext, reify all meta-objects to gain control.