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 relie on mathematical structures

My view of Coq

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

Some tactics relie 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
    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.
```

Exemple

```
Definition R := [...].
Lemma R_equiv : equivalence R.
```

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

Initial goal

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

$$
\begin{aligned}
(+) &: \mathit{int} \rightarrow \mathit{int} \rightarrow \mathit{int} &= \texttt{plus\_int} \\
(+) &: \mathit{float} \rightarrow \mathit{float} \rightarrow \mathit{float} &= \texttt{plus\_float} \\
(+) &: \mathit{string} \rightarrow \mathit{string} \rightarrow \mathit{string} &= \texttt{concat}
\end{aligned}
$$

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

Example

```
Class Addable (A:Type) :=
  (+) : A -> A -> A.

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

Instance ex2 : Addable Z :=
  (+) := ZArith.Zplus.
```

Example

```
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.
```

Under the hood

First class implementation (almost only syntactic sugar)

Class	$\Longrightarrow$	Record type
Instance	$\Longrightarrow$	(dependent) Record
Overloaded method	$\Longrightarrow$	Field
Parent class	$\Longrightarrow$	inferred argument
Resolution	$\Longrightarrow$	`eauto`

Under the hood

First class implementation (almost only syntactic sugar)

Class	$\Longrightarrow$	Record type
Instance	$\Longrightarrow$	(dependent) Record
Overloaded method	$\Longrightarrow$	Field
Parent class	$\Longrightarrow$	inferred argument
Resolution	$\Longrightarrow$	`eauto`
Structure recognition	$\Longrightarrow$	Instance search

Under the hood

First class implementation (almost only syntactic sugar)

Class	$\Longrightarrow$	Record type
Instance	$\Longrightarrow$	(dependent) Record
Overloaded method	$\Longrightarrow$	Field
Parent class	$\Longrightarrow$	inferred argument
Resolution	$\Longrightarrow$	`eauto`

Structure recognition	$\Longrightarrow$	Instance search
	$\Longrightarrow$	Proof search

Instance search

Given a class $\mathcal{C} = \langle x_1 : T_1, \ldots, x_n : T_n \rangle$, we're searching for $i : C$.
Decidable subset of the type system:

$$\text{(var)}\ \frac{}{\Gamma, t : T \ \vdash \ t : T}$$

$$\text{(inst)}\ \frac{\Gamma \ \vdash \ t_1 : T_1 \qquad \cdots \qquad \Gamma \ \vdash \ t_n : T_n}{\Gamma \ \vdash \ \langle t_1 : T_1 \ \ldots \ t_n : T_n \rangle}$$

$$\frac{\Gamma \ \vdash \ t_i : \forall x : T \cdot T_i \qquad \Gamma, x : T \ \vdash \ \langle\, t_1 : T_1 \ldots (t_i\ x) : T_i \ldots t_n : T_n \rangle}{\Gamma \ \vdash \ \forall x : T \cdot \langle\, t_1 : T_1 \ldots (t_i\ x) : T_i \ldots t_n : T_n \rangle}$$

Overview of the algorithm

Combinatorial work

We are looking for all possible proofs of $\mathcal{C}$

Textual recognition

- `∀ x, _ x x` : reflexivity lemma ⇒ instance of

```
Class Reflexive A (R:relation A) :=
  reflexive : forall x, R x x
```

- `∀ a b c, _ (_ a b) c = _ a (_ b c)` : associativity lemmay. Maybe a monoid ?

= Filtering on types

Usage example

```
Welcome to Coq trunk (11262)

Coq <
```

Usage example

```
Welcome to Coq trunk (11262)

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

Coq <
```

Usage example

```
Welcome to Coq trunk (11262)

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

Coq < Lemma R_sym : symmetric R. auto. Qed.
R_refl is defined
new instance Symmetric_1 : Symmetric R

Coq <
```

Usage example

```
Welcome to Coq trunk (11262)

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

Coq < Lemma R_sym : symmetric R. auto. Qed.
R_refl is defined
new instance Symmetric_1 : Symmetric R

Coq < Lemma R_trans : transitive R. auto. Qed.
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

Coq < Lemma R_sym : symmetric R. auto. Qed.
R_refl is defined
new instance Symmetric_1 : Symmetric R

Coq < Lemma R_trans : transitive R. auto. Qed.
R_trans is defined
new instance Transitive_1 : Transitive R
new instance PER_1 : PER R

Coq < Lemma R_refl : reflexive R. auto. Qed.
R_refl is defined
new instance Reflexive_1 : Reflexive R
new instance Equivalence_1 : Equivalence R
new instance Setoid_1 : Setoid A R
```

Discrimination nets

An efficient structure for one-to-many filtering.

The problem

Given an algebra of terms Λ, I have :

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

Which terms in S filter the pattern p ?

Discrimination nets

An efficient structure for one-to-many filtering.

The problem

Given an algebra of terms Λ, 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 *list* of sub-terms.

Example

```
type t =
  | Var of int
  | Lam of t
  | App of t       * t
```

Discrimination nets

= 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

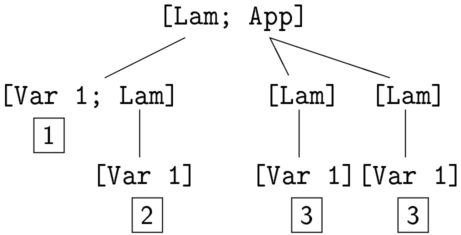
- To search for a term is to search for a path in the discrimination net.

Term Search

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

Example

This net :



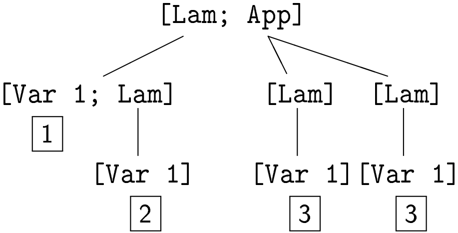
contains terms :

Term Search

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

Example

This net :



1 `Lam(Var 1)`

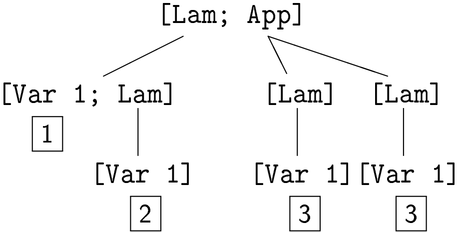
contains terms :

Term Search

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

Example

This net :



contains terms :

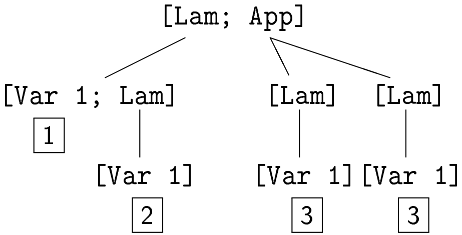
1	`Lam(Var 1)`
2	`Lam(Lam(Var 1))`

Term Search

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

Example

This net :



contains terms :

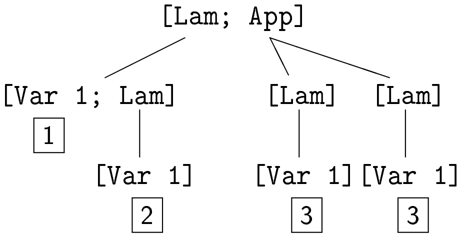
1	`Lam(Var 1)`
2	`Lam(Lam(Var 1))`
3	`App(Lam(Var 1),Lam(Var 1))`

Term Search

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

Example

This net :



contains terms :

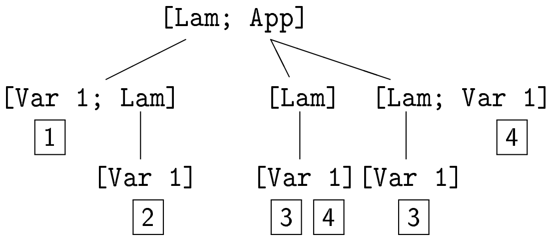
1	`Lam(Var 1)`
2	`Lam(Lam(Var 1))`
3	`App(Lam(Var 1),Lam(Var 1))`

$\Longrightarrow O(|T|)$

Filtering

- To search for a pattern is to search for a term, stop at the holes and enumerate terms underneith.

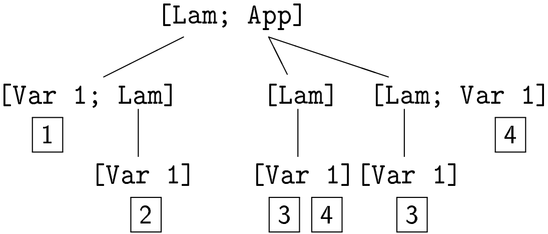
Example



Filtering

- To search for a pattern is to search for a term, stop at the holes and enumerate terms underneith.

Example

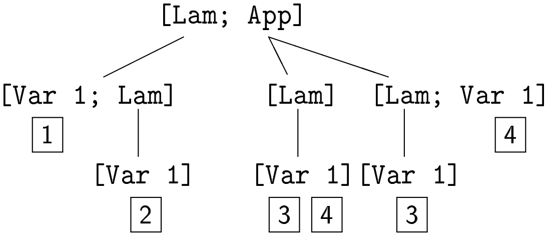


`App(X,Lam(Y)) ?`

Filtering

- To search for a pattern is to search for a term, stop at the holes and enumerate terms underneith.

Example



App(X,Lam(Y)) ?

$\Longrightarrow \bigcap \{ \boxed{3}\,\boxed{4} \,;\, \boxed{3} \} = \boxed{3}$

Current Implementation

$$\text{Functor}: \left.\begin{array}{l}\texttt{type t}\\ \texttt{val map}\\ \texttt{val fold}\\ \texttt{val compare}\end{array}\right\} \longrightarrow \left\{\begin{array}{l}\texttt{type dn}\\ \texttt{val add}\\ \texttt{val find\_all}\\ \texttt{val fold\_pattern}\end{array}\right.$$

Applied to Coq's `constr`

Current Implementation

$$\text{Functor :} \left.\begin{array}{l}\texttt{type t}\\ \texttt{val map}\\ \texttt{val fold}\\ \texttt{val compare}\end{array}\right\} \longrightarrow \left\{\begin{array}{l}\texttt{type dn}\\ \texttt{val add}\\ \texttt{val find\_all}\\ \texttt{val fold\_pattern}\end{array}\right.$$

Applied to Coq's `constr`

Primitives :

- `add`
- `find_all`
- `fold_pattern`

Current Implementation

Functor :

`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
```

Discrimination nets

Multiple variations

- Term/Pattern or Pattern/Term
- Full unification
- Filtering modulo δ, β...

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.