Coq: the reflection principle and encoding of mathematical hierarchies with canonical instances

I.Zhirkov

2016

What is Coq?

- Functional programming language with dependent types

What is Coq?

- Functional programming language with dependent types
  - one type can depend from others;
  - function types can have dependencies;
  - function can accept types.

What is Coq?

- Functional programming language with dependent types
  - one type can depend from others;
  - function types can have dependencies;
  - function can accept types.
- Logical system (intuitionistic logic)
  - types and programs can be used to encode propositions and proofs.

As a programming language

- A variation of a typed lambda calculus
- Usual stuff:
  - Functions;
  - Types and pattern matching;
  - Limited recursion (can’t encode infinite computations).

As a logical system

- Intuitionistic $\sim$ Constructive
  - No $\top$ or $\perp$ values, assigned to each formula;
  - We infer proofs instead.

Correspondance

- **Axiom** (provable apriori) = Axioms, Selected type inhabitants

```
Inductive nat :=
| Zero: nat (* ← witness *)
| Succ: nat → nat.

Axiom em: forall A, A ∨ not A.
```

- **Theorem with premises** = encode a proof algorithm. Proof of $A \wedge B \rightarrow C$: "Given a proof for (A and B) build one for C"
- *Modus ponens* is an application.

Example of a dependent type

Encode the existence quantifier:

```
Inductive sig (A : Type) (P : forall _ : A, Prop) : Type :=
   exist : forall (x : A) (_ : P x), sig P
```

A little less formal:

$$\left(a : A, (\mathit{fun}\ x : A => \_ : \mathit{Prop})\ a \right)$$

Data is decomposed via induction (relevant axioms are generated automatically).

```
Inductive nat :=
| Zero: nat (* ← witness *)
| Succ: nat → nat.

Check nat_ind.
(* :> *)
nat_ind: forall P : nat → Prop,
        P Zero
        → (forall n : nat, P n → P (Succ n))
        → forall n : nat, P n
```

Equality in Coq

- Definitional (judgemental)
- Propositional

```
Inductive eq (A : Type) (x : A) := eq_refl : x = x.
```

- Computable – if we define it (not part of the system)

Examples

“Data” lives in `Set`, Propositions live in `Prop`.

```
Inductive True : Prop := T.
Inductive False : Prop := .

Inductive And (A B: Prop) :=
| join : A → B → And A B.

Inductive Or (A B: Prop) :=
| left : A→  Or A B
| right: B → Or A B.
```

Examples

```
Definition t1 : (A → B → C) → A → B → C :=
  fun f: A→ B→ C ⇒
    fun a: A ⇒
      fun b: B ⇒
        f a b.
```

Examples: tactics

Big proof terms are hard to write at once.
Adapt an iterative approach of their construction.

```
Definition t1 : (A → B → C) → A → B → C :=
  fun f: A→ B→ C ⇒
    fun a: A ⇒
      fun b: B ⇒
        f a b.
```

=>

```
Parameter A B C: Prop.
Definition t1 : (A → B → C) → A → B → C.
  intros f a b.
  apply f.
  exact a.
  exact b.
Qed.
```

Examples: tactics

```
(* This definition is not complete!*)
Definition t1 : (A → B → C) → A → B → C.
```

Goal (shows current step in proof construction: what we want to obtain and what we have in context):

1 `subgoal`, `subgoal` 1 (`ID` 1)

```
(A → B → C) → A → B → C
```

Examples: tactics

```
Definition t1 : (A → B → C) → A → B → C.
  intros f a b.
```

Goal:

```
1 subgoal, subgoal 1 (ID 4)

  f : A → B → C
  a : A
  b : B
  ———
  C
```

Examples: tactics

```
Definition t1 : (A → B → C) → A → B → C.
  intros f a b.
  apply f.
```

Goal (2 subgoals because *f* has two arguments of type *A* and *B*).

```
2 subgoals, subgoal 1 (ID 5)

  f : A → B → C
  a : A
  b : B
  ---
  A

subgoal 2 (ID 6) is:
  B
```

Examples: tactics

```
Definition t1 : (A → B → C) → A → B → C.
  intros f a b.
  apply f.
  exact a.
```

Goal (one subgoal out):

```
1 subgoal, subgoal 1 (ID 6)

  f : A → B → C
  a : A
  b : B
 ---
  B
```

Examples: tactics

```
Definition t1 : (A → B → C) → A → B → C.
  intros f a b.
  apply f.
  exact a.
  exact b.
```

We are done here.

- Some things are not decidable, so in general we can’t autocompute proofs.
  Explicitly provide proof of primarity
- But some things are decidable and we want to use it.

Unleash bruteforce computing to spare human time.

Reflection

is a mean to switch between constructive reasoning and a computable (via beta reductions) form of a proposition.
Entirely inside Coq's theory.
`reflect1.v`
It is a fondament of an **ssreflect** library

ssreflect

- Part of Mathematical Components library (used to formalize Odd Order and Four color theorems)
- Extends Coq with a minimalistic proof language (enough for most proofs)

ssreflect

- *move* manipulates context and goal;
- *case* performs case analysis on a term;
- *elim* performs induction;
- *rewrite* performs a rewrite using a hypothesis $A = B$ (augmented);
- *apply* applies a function;
- *done* tries to finish proof automatically.

ssreflect tactics

`move=> h.`

Γ |- $H \to X$

===

$\Gamma \cup \{h : H\}$ |- X

ssreflect tactics

`move: h.`

$\Gamma \cup \{h : H\}$ |- X

===

Γ |- $H \rightarrow X$

ssreflect tactics

`move /h.`

$\Gamma \cup \{h : H\}$ |- $a- > X$

===

$\Gamma \cup \{h : H\}$ |- $h\ a \rightarrow X$

Many, many ways to combine

```
move=> H [Hl|[Hrl|Hrr]].
elim: x y => //=.
move /eqP => -> [] [] => /=.

(* Real world proof *)
Lemma streeR_total: total streeR.
by move: HRtotal;
rewrite /total /streeR /treeR => ?[[??]?][[??]?] => //=.
Qed.
```

What does ssreflect reflect?

- Equality;
- Has (a list has an item...)
- Logical or/and/not/implication...
- Many more decidable things

`reflect1.v`

An understanding of canonical structures is required to understand how to produce “general equalities” etc.

Canonical instances

- A way to encode proof search;
- A database of hints for type solver;
- Currenty only for specific cases of dependent records.

`canonical1.v`

Canonical instances

- Encode mathematical structures' hierarchy (like semigroups, monoids etc).
- No strict inheritance
- Dualism between coercions and CI (we can always coerce to a simpler form, but how to select the correct rich form?)
- Example: model general equality and ordering separately and make them usable for a type.

`eq_leq.v`

Similar mechanisms

- Type Classes
- Canonical Structures (a bit more general)
- Unification Hints (more general)

Readings on ssreflect, reflection and canonical structures

- ssreflect docs
- ssreflect tutorial by G. Gontier et al
- Ilya Sergey “Programs and proofs” (best ssreflect book available)
- Assia Mahboubi, Enrico Tassi “Canonical Structures for the working Coq user”

Thank you.
Question time.