Coq libraries for HoTT/UF

Assia Mahboubi – HOTT/UF workshop 2015

Disclaimer

The title and the content do not match.

In this talk:

- No brand new library;
- No new formalized result;
- No comparative survey.

Only some methodological remarks.

Large Libraries of Formalized Mathematics

Issues:

- Get the definition(s) and notations right
- Get the right corpus of lemmas
- Get the right automation tools
- Maintain a rigorous software engineering discipline
- Write proofs robust to the regular re-factoring

Mathematical Components

- Authors: the Math. Comp. team (led by G. Gonthier);
- Follow up of a Coq proof of the Four Colour Theorem;
- Culminates with a proof of the Odd Order Theorem.
- 6 years, $\sim$ 15 authors, $\sim$ 160 000 l.o.c.

Theorem (Feit-Thompson - 1963)

Every group of odd order is solvable.

Mathematical Components

Features

- Constructive proof;
- Wide variety of algebraic theories;
- Large hierarchy of algebraic structures, with many instances;
- Coherent policies maintained across the libraries;
- Methodology: small scale reflection and type inference;
- Extension of the tactic language (ssreflect).

http://ssr.msr-inria.inria.fr/

Small Scale Reflection

Reflection: Use conversion and definitional equality
to devise automated deduction procedures.

Small scale: Make reflection local and pervasive
to automate bookkeeping.

Boolean Reflection

Compare:

```
Inductive Zis_gcd (a b g:Z) : Prop :=
 Zis_gcd_intro :
  (g | a) -> (g | b) ->
  (forall x, (x | a) -> (x | b) -> (x | g)) ->
  Zis_gcd a b g.

Definition rel_prime (a b:Z) : Prop := Zis_gcd a b 1.

Inductive prime (p:Z) : Prop :=
  prime_intro :
    1 < p -> (forall n:Z, 1 <= n < p -> rel_prime n p) -> prime p.
```

Boolean Reflection

Compare:

```
Inductive Zis_gcd (a b g:Z) : Prop :=
 Zis_gcd_intro :
  (g | a) -> (g | b) ->
  (forall x, (x | a) -> (x | b) -> (x | g)) ->
  Zis_gcd a b g.

Definition rel_prime (a b:Z) : Prop := Zis_gcd a b 1.

Inductive prime (p:Z) : Prop :=
  prime_intro :
    1 < p -> (forall n:Z, 1 <= n < p -> rel_prime n p) -> prime p.
```

Or even:

```
Definition prime k : Prop :=
    k > 1 /\ forall r d, 1 < d < k -> k <> r * d.
```

Boolean Reflection

With:

```
Fixpoint prime_decomp_rec m k a b c e :=
  let p := k.*2.+1 in
  if a is a'.+1 then
    if b - (ifnz e 1 k - c) is b'.+1 then
      [rec m, k, a', b', ifnz c c.-1 (ifnz e p.-2 1), e] else
    if (b == 0) && (c == 0) then
      let b' := k + a' in [rec b'.*2.+3, k, a', b', k.-1, e.+1] else
    let bc' := ifnz e (ifnz b (k, 0) (edivn2 0 c)) (b, c) in
    p ^? e :: ifnz a' [rec m, k.+1, a'.-1, bc'.1 + a', bc'.2, 0] [:: (m, 1)]
  else if (b == 0) && (c == 0) then [:: (p, e.+2)] else p ^? e :: [:: (m, 1)]
where "[ 'rec' m , k , a , b , c , e ]" := (prime_decomp_rec m k a b c e).

Definition prime_decomp n :=
  let: (e2, m2) := elogn2 0 n.-1 n.-1 in
  if m2 < 2 then 2 ^? e2 :: 3 ^? m2 :: [::] else
  let: (a, bc) := edivn m2.-2 3 in
  let: (b, c) := edivn (2 - bc) 2 in
  2 ^? e2 :: [rec m2.*2.+1, 1, a, b, c, 0].

Definition prime p :=
  if prime_decomp p is [:: (_ , 1)] then true else false.
```

Boolean Reflection: Free Theorems

```
(* Order relation on nat *)
Fixpoint le n m := match n, m with
  | 0    , _    => true
  | S _  , 0    => false
  | S n', S m' => le n' m' end.
Notation "a <= b" := (le a b).
```

Boolean Reflection: Free Theorems

```
(* Order relation on nat *)
Fixpoint le n m := match n, m with
  | 0    , _    => true
  | S _  , 0    => false
  | S n', S m' => le n' m' end.
Notation "a <= b" := (le a b).

(*Free theorems, thanks computation *)
Lemma le0n n   : 0 <= n = true.
Proof. reflexivity. Qed.

Lemma leSS n m : S n <= S m = n <= m.
Proof. reflexivity. Qed.
```

Boolean Reflection: Free Theorems

```
(* Order relation on nat *)
Fixpoint le n m := match n, m with
  | 0    , _     => true
  | S _ , 0      => false
  | S n', S m' => le n' m' end.
Notation "a <= b" := (le a b).

(*Free theorems, thanks computation *)
Lemma le0n n   : 0 <= n = true.
Proof. reflexivity. Qed.

Lemma leSS n m : S n <= S m  =  n <= m.
Proof. reflexivity. Qed.

(* Almost free theorems *)
Lemma lenn n   : n <= n = true.
Proof. by elim: n. Qed.
```

Boolean Reflection and Deduction

Free theorems combine well with boolean connectives:

```
  n : nat
  m : nat
  ==================
   1 <= S m && (S n <= 0   ==>  b) && P = true

simpl.
```

Boolean Reflection and Deduction

Free theorems combine well with boolean connectives:

```
n : nat
m : nat
===================
 1 <= S m && (S n <= 0   ==>  b) && P = true
```

```
n : nat
m : nat
===================
 P = true
```

```
simpl.
```

Boolean vs Prop Definitions

Whereas using the relation defined in the standard library:

```
Inductive le (n : nat) : nat -> Prop :=
    le_n : le n n
  | le_S : forall m : nat, le n m -> le n (S m)
```

- The proof of `n <= m` chains `m - n + 1` constructors;
- Local simplifications are (much) less easy.

The Rewrite Swiss Knife: Examples

Chaining: `rewrite foo bar` rewrites with `foo`, then `bar`.

Repeating, repeating if possible: `rewrite !foo`, `rewrite ?bar`

Simpl: `rewrite /=` but also `rewrite foo /= bar`

Trivial: `rewrite //` but also `rewrite foo // bar`

Unfold: `rewrite /blah`

Change for convertible: `rewrite -[foo]/blah`

Exact Patterns: `rewrite [X in _ <= X]foo`, `rewrite [LHS]foo`, `rewrite [X in X + _ = _]/=`

Context Patterns: `rewrite [in X in _ <= X]foo`, `rewrite [in LHS]foo`

Boolean and Prop Definitions

- Nested binary `Prop` conjunctions and unary, boolean, triple-conjunction:

```
Lemma and3P : [/\ b1, b2 & b3] <-> [&& b1, b2 & b3] = true.
```

- Back to the definition of primality:

```
Lemma primeP p :
  reflect (p > 1 /\ forall d, d %| p -> d == 1 || d == p) (prime p).
```

From Bool to Prop and Back

`move/eqP: h => h` : transforms hypothesis `h : n == m` in the context into `h : n = m`

`apply/eqP` : transforms a goal `n == m` into `n = m`.

`case/orP: h => h` : when `h : p || q`, performs a case analysis: `h : p` in one branch, `h : q` in the other.

`case/andP: h => h1 h2` : when `h : p && q`, introduces both `h1 : p` and `h2 : q`.

`rewrite (negPf h):= when h : ~~ p` : rewrites occurrences of p to `false` in the goal.

Boolean Reflection & Classical Logic

Excluded middle is just case analysis:

```
(* Boolean Excluded Middle, never used as such. *)
Lemma EMb (b : bool) : b || ~~b = true.
Proof. by case b. Qed.
```

Boolean Reflection & Classical Logic

Contraposition is provable:

```
Lemma contra (c b : bool) :
  (c = true -> b = true) -> ~~ b = true -> ~~ c = true.

Lemma contraL (c b : bool) :
(c = true -> ~~ b = true ) -> b = true -> ~~ c = true.
```

Boolean Reflection & Classical Logic

The classical monad is convenient to use:

```
Definition classically P b := (P -> b = true) -> b = true.

Lemma classic_EM : forall P, classically (decidable P).
```

Boolean Reflection & Classical Logic

The classical monad is convenient to use:

```
Definition classically P b := (P -> b = true) -> b = true.

Lemma classic_EM : forall P, classically (decidable P).

Lemma classic_pick (T : Type) (P : T -> Prop) :
  classically ({x : T | P x} + (forall x, ~ P x)).
```

Numbers in the MathComp Libraries

Instances of numbers with boolean comparisons:

- Natural numbers, integers,
- Rational numbers, modular arithmetic,
- Algebraic real and complex numbers,...

With:

- Elementary arithmetic (binomials, primality, logs,...)
- Group, ring, field, ordered structures theories
- ...

Equality Types

The fundamental structure to the library is (unfolds to):

```
Structure eqType := Pack {
eq_sort : Type;
eq_op : eq_sort -> eq_sort -> bool;
eq_opP : forall x y : eq_sort, (op x y = true) <-> (x = y)}.

Notation "x == y" := (@eq_op _ x y).
```

Equality Types

The main properties shared by instances of `eqType` are:

- The infix `==` notation
- Hedberg's theorem:

```
Theorem eq_irrelevance (T : eqType) x y :
  forall e1 e2 : x = y :> T, e1 = e2.
```

- Canonical preservation of the `eqType` structure through pair, list, option, ...

Instances are the expected ones:

unit, booleans, numbers, finite types,...

Inference of an eqType Structure

```
Structure eqType := Pack {
eq_sort : Type;
eq_op : eq_sort -> eq_sort -> bool;
eq_opP : forall x y : eq_sort, (op x y = true) <-> (x = y)}.

Notation "x == y" := (@eq_op _ x y).
```

(Demo)

Inference of an eqType Structure

We input an incomplete term:

```
@eq_op ?1 [:: 9] [:: 3, 6]
```

Inference of an eqType Structure

We input an incomplete term:

```
@eq_op ?1 [:: 9] [:: 3, 6]
```

with the expected types:

```
@eq_op      ?1          [:: 9]            [:: 3, 6]
          eqType      eq_sort ?1          eq_sort ?1
```

Inference of an eqType Structure

We input an incomplete term:

```
@eq_op ?1 [:: 9] [:: 3, 6]
```

with the expected types:

```
@eq_op       ?1          [:: 9]          [:: 3, 6]
          eqType      eq_sort ?1       eq_sort ?1
```

And we should therefore solve the unification equation:

```
list nat = eq_sort ?1
```

Inference of an eqType Structure

We want to solve `list nat = eq_sort ?1`

Inference of an eqType Structure

We want to solve `list nat = eq_sort ?1`

Theorem list_eqType provides a canonical op on lists:

$$\frac{T : \text{eqType}}{\text{list } (\text{eq\_sort } T) \equiv \text{eq\_sort } (\text{list\_eqType } T)}$$

Inference of an eqType Structure

We want to solve `list nat = eq_sort ?1`

Theorem list_eqType provides a canonical op on lists:

$$\frac{T : \text{eqType}}{\text{list } (\text{eq\_sort } T) \equiv \text{eq\_sort } (\text{list\_eqType } T)}$$

We can look for a solution of the shape:

```
?1 = list_eqType ?2
```

Inference of an eqType Structure

We want to solve `list nat = eq_sort ?1`

Theorem list_eqType provides a canonical op on lists:

$$\frac{T : \text{eqType}}{\text{list } (\text{eq\_sort } T) \equiv \text{eq\_sort } (\text{list\_eqType } T)}$$

We can look for a solution of the shape:

```
?1 = list_eqType ?2
```

With the new constraint:

```
nat = eq_sort ?2
```

Inference of an eqType Structure

We want to solve `nat = eq_sort ?2`

Inference of an eqType Structure

We want to solve `nat = eq_sort ?2`

Theorem nat_eqType provides a canonical op on lists:

$$\frac{}{\mathsf{nat} \equiv \mathsf{eq\_sort}\ \mathsf{nat\_eqType}}$$

Inference of an eqType Structure

We want to solve `nat = eq_sort ?2`

Theorem nat_eqType provides a canonical op on lists:

$$\frac{}{\text{nat} \equiv \text{eq\_sort nat\_eqType}}$$

which concludes the search with the solution:

```
@eq_op ?1 [:: 9] [:: 3, 6]

list nat = eq_sort ?1
?1 = list_eqType ?2
nat = eq_sort ?2
?2 = nat_eqType


--> ?1 = list_eqType nat_eqType
```

Canonical Structures

- A type inference mechanism via unification hints;
- Based on projections of records;
- Implemented in Coq by A. Saïbi (circa 1997);
- Similar to (but subtly different from)
  N. Oury and M. Sozeau's Type Classes.

Bibliography:

- Typing algorithm in type theory with inheritance,
  A. Saïbi, Proceedings of POPL 1997, ACM Press.
- Canonical Structures for the working Coq user,
  A. Mahboubi, E. Tassi, Proceedings of ITP 2013, Springer.

A Hierarchy of Interfaces

Populating the Hierarchy: subTypes

The root of the hierarchy comprises interfaces for:

- eqType, finType, countType, choiceType

Interestingly enough:

- if P is a decidable (boolean) predicate
- if T is an [eq|fin|count|choice]Type
- then so is $\{x : T \mid P\ x = \mathit{true}\}$
- and its isomorphic copies.

Populating the Hierarchy: subTypes

`(s : subType T P)` is isomorphic to `{x : T | P x = true}`.

```
Structure subType (T : Type) (P : pred T) : Type := SubType {
  sub_sort :> Type;
  val : sub_sort -> T;
  Sub : forall x, P x -> sub_sort;
  _ : forall K (_ : forall x Px, K (@Sub x Px)) u, K u;
  _ : forall x Px, val (@Sub x Px) = x
}.
```

- `sub_sort` is its carrier type;
- `val` injects `s` into `T`
- `Sub` is the pseudo constructor of the subType.

Populating the Hierarchy: subTypes

`(s : subType T P)` is isomorphic to `{x : T | Px = true}`.

This infrastructure provides:

- A generic construction for natural subTypes;
- Canonical instances of transferred [eq|fin|count|choice]Type;
- A proof that `val : s -> T` is injective;
- A generic partial projection `T -> option s`.

Populating the Hierarchy

More generally new instances of [eq|fin|count|choice] structures can be formed canonically for:

- Isomorphic types (via a bijection) or subtypes;
- Quotients by a boolean relation;
- Types isomorphic to an instance of a generic variable-arity labeled tree type.

(Demo)

Features

Features:

- A uniform set of formalized content;
- Reusable design patterns
- A careful management of computational behaviors;
- Several representations for a same object;
- Tatics.

But:

- Based on logic in `Prop`;
- Limited support of the tatics for HoTT;
- Almost no analysis, no category theory.

HoTT/UF Libraries are Young

Impressive and elegant experiments but:

- Large parts of other existing libraries cannot be combined;
- Complementary contents, with incompatible styles;
- Mexican hat syndromes;
- Management of computational behavior;
- Lack of dedicated proof commands.

Some Consolidation Perspectives

- More documentation of the road-map;
- More constructions, and more about their specific theory;
- A better tactic language?