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, ~ 15 authors, ~ 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** \( \text{Zis\_gcd} \) (a b g:Z) : Prop :=

\( \text{Zis\_gcd\_intro} : (g \mid a) \to (g \mid b) \to (\forall x, (x \mid a) \to (x \mid b) \to (x \mid g)) \to \text{Zis\_gcd} \ a \ b \ g. \)

**Definition** \( \text{rel\_prime} \) (a b:Z) : Prop := \( \text{Zis\_gcd} \ a \ b \ 1. \)

**Inductive** \( \text{prime} \) (p:Z) : Prop :=

\( \text{prime\_intro} : 1 < p \to (\forall n:Z, 1 \leq n < p \to \text{rel\_prime} n p) \to \text{prime} p. \)
Boolean Reflection

Compare:

\[
\text{Inductive } \text{Zis\_gcd}(a \ b \ g:Z) : \text{Prop} := \\
\text{Zis\_gcd\_intro} : \\
(g \mid a) \rightarrow (g \mid b) \rightarrow \\
(\forall x, (x \mid a) \rightarrow (x \mid b) \rightarrow (x \mid g)) \rightarrow \\
\text{Zis\_gcd} \ a \ b \ g.
\]

\[
\text{Definition } \text{rel\_prime}(a \ b:Z) : \text{Prop} := \text{Zis\_gcd} \ a \ b \ 1.
\]

\[
\text{Inductive } \text{prime}(p:Z) : \text{Prop} := \\
\text{prime\_intro} : \\
1 < p \rightarrow (\forall n:Z, 1 \leq n < p \rightarrow \text{rel\_prime} n p) \rightarrow \text{prime} \ p.
\]

Or even:

\[
\text{Definition } \text{prime} \ k : \text{Prop} := \\
k > 1 \land \forall r \ d, 1 < d < k \rightarrow k \neq r \times d.
\]
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 \(\text{le} \ n \ m := \text{match} \ n, \ m \text{ with} \)
| \(0\), \(_\) \(=> \text{true} \)
| \(S\) \(_,\) \(0\) \(=> \text{false} \)
| \(S \ n', \ S \ m' \Rightarrow \text{le} \ n' \ m' \text{ end.} \)
Notation "\(a <= b\)" := (le a b).

(*Free theorems, thanks computation *)

Lemma \(\text{leOn} \ n \ : \ 0 <= n = \text{true.} \)
Proof. reflexivity. Qed.

Lemma \(\text{leSS} \ n \ m \ : \ S \ n <= S \ m = n <= m. \)
Proof. reflexivity. Qed.

(* Almost free theorems *)

Lemma \(\text{lenn} \ n \ : \ n <= n = \text{true.} \)
Proof. by elim: n. Qed.
Free theorems combine well with boolean connectives:

\[
\begin{align*}
&n : \text{nat} \\
m : \text{nat} \\
&=\text{true}
\end{align*}
\]

simpl.
Free theorems combine well with boolean connectives:

\[
\begin{align*}
n : \mathbb{N} \\
m : \mathbb{N} \\
\text{==================}
\end{align*}
\]

\[
1 \leq S m \land (S n \leq 0 \Rightarrow b) \land P = \text{true}
\]

simpl.
Boolean vs Prop Definitions

Whereas using the relation defined in the standard library:

```coq
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 \( n \leq m \) chains \( m - n + 1 \) constructors;
• Local simplifications are (much) less easy.
The Rewrite Swiss Knife: Examples

Chaining: \texttt{rewrite foo bar} rewrites with \texttt{foo}, then \texttt{bar}.

Repeating, repeating if possible: \texttt{rewrite !foo, rewrite ?bar}

Simpl: \texttt{rewrite /=} but also \texttt{rewrite foo /= bar}

Trivial: \texttt{rewrite //} but also \texttt{rewrite foo // bar}

Unfold: \texttt{rewrite /blah}

Change for convertible: \texttt{rewrite -[foo]/blah}

Exact Patterns: \texttt{rewrite [X in _ <= X]foo, rewrite [LHS]foo, rewrite [X in X + _ = _] /=}

Context Patterns: \texttt{rewrite [in X in _ <= X]foo, rewrite [in LHS]foo}
Boolean and Prop Definitions

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

Lemma \texttt{and3P} \ : \ [\land b_1, b_2 \land b_3] \iff [\& b_1, b_2 \land b_3] = \text{true}.

• Back to the definition of primality:

Lemma \texttt{primeP} p :
\text{reflect} (p > 1 \land \forall d, d \mid p \implies d = 1 \lor d = p) (\text{prime p}).
From Bool to Prop and Back

**move/eqP**: \( h \Rightarrow h : \text{transforms hypothesis } h : n = m \text{ in the context into } h : n = m \)

**apply/eqP**: transforms a goal \( n = m \) into \( n = m \).

**case/orP**: \( h \Rightarrow h : \text{when } h : p \lor q, \text{performs a case analysis: } h : p \text{ in one branch, } h : q \text{ in the other.} \)

**case/andP**: \( h \Rightarrow h_1 \ h_2 : \text{when } h : p \land q, \text{introduces both } h_1 : p \text{ and } h_2 : q. \)

**rewrite** \((\text{negPf } h) := \text{when } h : \neg p : \text{rewrites occurrences of } p \text{ to false in the goal.} \)
**Boolean Reflection & Classical Logic**

Excluded middle is just case analysis:

(* Boolean Excluded Middle, never used as such. *)

Lemma **EMb** \( (b : \text{bool}) : b \lor \lnot b = \text{true} \).

Proof. by case \( b \). Qed.
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.
The classical monad is convenient to use:

Definition **classically** P b := (P \to b = true) \to b = true.

Lemma **classic_EM** : forall P, classically (decidable P).
The classical monad is convenient to use:

Definition **classically** \( P \rightarrow b := (P \rightarrow b = true) \rightarrow b = true. \)

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

Lemma **classic_pick** \((T : Type) (P : T \rightarrow Prop) : \)
\[
\text{classically } \{ x : T \mid P \ x \} + (\text{forall } x, \sim P \ x).\]
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 $\text{eqType}$ are:

- The infix $\equiv$ notation
- Hedberg’s theorem:

  $$\text{Theorem } \text{eq_irrelevance } (T : \text{eqType}) \ x \ y :$$
  $$\forall e_1 \ e_2 : x = y :> T, \ e_1 = e_2.$$  

- Canonical preservation of the $\text{eqType}$ 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 :: eqType :: eq_sort ?1 :: eq_sort ?1
Inference of an eqType Structure

We input an incomplete term:

\[ \text{@eq\_op ?1 \ [:: 9] \ [:: 3, 6]} \]

with the expected types:

\[ \text{@eq\_op \ ?1 \ [:: 9] \ eq\_Type \ eq\_sort \ ?1 \ [:: 3, 6]} \]

And we should therefore solve the unification equation:

\[ \text{list \ nat = eq\_sort \ ?1} \]
Inference of an eqType Structure

We want to solve \( \text{list nat} = \text{eq_sort } ?1 \)
Inference of an eqType Structure

We want to solve \( \text{list nat} = \text{eq_sort} \ ?1 \)

Theorem \text{list_eqType} provides a canonical op on lists:

\[
T : \text{eqType} \\
\text{list (eq_sort} \ T) \equiv \text{eq_sort (list_eqType} \ T)
\]
Inference of an eqType Structure

We want to solve \( \text{list nat} = \text{eq_sort } ?1 \)

Theorem \( \text{list_eqType} \) provides a canonical op on lists:

\[
T : \text{eqType} \\
\text{list (eq_sort } T) \equiv \text{eq_sort (list_eqType } T)
\]

We can look for a solution of the shape:

\( ?1 = \text{list_eqType } ?2 \)
Inference of an eqType Structure

We want to solve \( \text{list}\n\text{nat} = \text{eq}\_\text{sort} \ ?1 \)

Theorem \text{list\_eqType} provides a canonical op on lists:

\[
T : \text{eqType} \\
\text{list} \ (\text{eq\_sort} \ T) \equiv \text{eq\_sort} \ (\text{list\_eqType} \ T)
\]

We can look for a solution of the shape:

\(?1 = \text{list\_eqType} \ ?2\)

With the new constraint:

\(\text{nat} = \text{eq\_sort} \ ?2\)
Inference of an eqType Structure

We want to solve $\text{nat} = \text{eq\_sort} \ ?2$

Theorem nat_eqType provides a canonical op on lists:

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

which concludes the search with the solution:

@\text{eq\_op} \ ?1 \ [:: 9] \ [:: 3, 6] \\
\text{list\_nat} = \text{eq\_sort} \ ?1 \\
?1 = \text{list\_eqType} \ ?2 \\
?2 = \text{nat\_eqType}$
Inference of an eqType Structure

We want to solve \( \text{nat} = \text{eq\_sort} \ ?2 \)

Theorem \text{nat\_eqType} provides a canonical operation on lists:

\[
\text{nat} \equiv \text{eq\_sort} \ \text{nat\_eqType}
\]
Inference of an eqType Structure

We want to solve $\text{nat} = \text{eq\_sort } ?2$

Theorem $\text{nat\_eqType}$ provides a canonical op on lists:

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

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 \text{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:
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 $\{x : T \mid P \, x = true\}$
- 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

\[(s : \text{subType } T P) \text{ is isomorphic to } \{x : T \mid Px = \text{true}\}\].

This infrastructure provides:

- A generic construction for natural subTypes;
- Canonical instances of transferred \[\text{[eq|fin|count|choice]}\]Type;
- A proof that \(\text{val} : s \to T\) is injective;
- A generic partial projection \(T \to \text{option } s\).
Populating the Hierarchy

More generally new instances of \(\textbf{eq}_{\textbf{fin}}\textbf{count}_{\textbf{choice}}\) 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 \( \text{Prop} \);
- 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?