Introduction to Coq
Part 3: Some libraries

Yves Bertot

September 2023

General recursion

- Need to go beyond structural recursion
- Preserve guarantees of termination, but free from structure constraints
- In essence, separate the proof of termination from the algorithm description

Fueled recursion

- The easy trick: count the number of recursive calls
  - Use an extra natural number argument
  - Return a default value upon exhaustion
- Easy to program, but inconvenient
  - Need to figure out how much fuel is enough
  - Any gross over-estimate of fuel slows down the code
- Fuel also clutters the proofs
  - Need to prove that the case of fuel exhaustion is never reached
  - Tantamounts to proving that the intended algorithm was terminating

Example fuel argument

```
Fixpoint fact_fuel (x : Z) (fuel : nat) :=
match fuel with
| O => 0
| S p => if x <=? 0 then x * fact_fuel (x - 1) p
end

Definition Zfact (x : Z) := fact_fuel x (S (Z.to_nat x)).
```

Principled separation of termination proofs

- A generic notion of *well-founded* relations
- Show that recursive calls follow such a well-founded relation
- Proofs can be moved away from algorithmic content
- Minimal clutter to ensure important tests are remembered

The Equations plugin

```
From Equations Require Import Equations.
Require Import Wellfounded.

#[local]
Instance zltwf x :
   WellFounded (fun n m => x <= n < m) := (Z.lt_wf x).

Equations Zfact'(x : Z) : Z
     by wf x (fun n m => 0 <= n < m) :=
  Zfact' x with (Z_le_dec x 0) := {
  | left _ => 1
  | right xnle0 => x * Zfact' (x - 1)
  }.
Next Obligation.
lia.
Qed.
```

Comments on Equations

- Oriented towards frequent use of dependent types
- For instance, use of `Z_le_dec` of type:
  `forall x y :  Z, {x <= y}+{~ x <= y}`
- Rely on an inductive with two constructors, where the first one contains a proof of $\mathtt{x} \leq \mathtt{y}$
- This proof must be constructed at definition time
- The proof is provided at use time and can be used in proofs

Generic use of boolean test capture

```
Definition inspect {A} (a : A) : {b | a = b} :=
  exist _ a eq_refl.

Notation "x 'eqn:' p" := (exist _ x p)
   (only parsing, at level 20).

Equations Zfact2 (x : Z) : Z
     by wf x (fun n m => 0 <= n < m) :=
  Zfact2 x with inspect (x <=? 0) := {
  | true eqn: xle0 => 1
  | false eqn: xnotle0 => x * Zfact2 (x - 1)
  }.
Next Obligation.
lia.
Qed.
```

Advantage of the second approach

- The boolean algorithm can be programmed as usual
- Theorems are required to interpret the result
  - In the example `lia` has the knowledge that `x <=?  0 = false` means `0 < x`

Using functions defined by `Equations`

- Reliance on proofs makes that computation is rarely possible
- In proofs: `Equations` provides lemma to be used for writing
- In computations: No computation inside Coq, but extraction makes it possible to generate OCaml code that performs the same

Example usage in proofs

```
Check (Zfact'_equation_1
     : forall x : Z, Zfact' x =
       Zfact'_unfold_clause_1 x (Z_le_dec x 0)).

Check (Zfact'_unfold_clause_1 =
fun (x : Z) (refine : {x <= 0} + {~ x <= 0}) =>
if refine then 1 else x * Zfact' (x - 1)
     : forall x : Z, {x <= 0} + {~ x <= 0} -> Z).

Lemma Zfact2_main (x : Z) :
  Zfact2 x = if x <=? 0 then 1 else x * Zfact2 (x - 1).
Proof.
rewrite Zfact2_equation_1; simpl.
destruct (x <=? 0); auto.
Qed.
```

Example extraction

```
Extraction Zfact2.
```

```
let rec zfact2 x =
  match inspect (Z.leb x Z0) with
  | True -> Zpos XH
  | False -> Z.mul x (let y = Z.sub x (Zpos XH) in
                      zfact2 y)
```

Real Numbers

Examples using real numbers

- In type theory, only pure lambda-calculus and inductive types have computation constant
- Reasoning modulo axioms is possible, but the axioms come without computation constant
- Justifying the existence of classical real numbers relies on two axioms
- As a result, we can reason about real number computations, but not perform them in the same way

Example : computation with the number PI

```
Require Import Reals Lra.

Compute PI.
(* R1 + R1 * (let (x, _) := PI_2_aux in x) *)

Print PI.
(* PI = 2 * PI2 *)

Check PI_2_aux.
(* PI_2_aux :
   {z | R | 7 / 8 <= z <= 7 / 4 /\ - cos z = 0} *)

Lemma example_formula_with_pi_and_sin : 1 + sin PI = 1.
Proof.
assert (tmp := sin_PI).
lra.
Qed.
```

How do I compute as with a pocket calculator

- Pocket calculator return approximations
- With minimal guarantees
  - The quality degrades with the number of operations involved
  - Hard to track by users
  - Not satisfactory for proofs
- A proof approach relies on proving equalities or comparisons
  - The previous example was an equality
  - Equalities between real numbers and rational numbers are rare
  - Comparisons are often good enough
  - Even better: intervals

Mathematical Components

The Mathematical Components Library

- Library initiated by G. Gonthier in the proof of the 4 color theorem
- Extended for the proof of the odd-order theorem
- Comes with its own tactic language
- Contents covering finite types, group theory, finite dimension linear algebra, elementary number theory, ponymials, etc.
- A principle use of boolean predicates and reflexion

A hierarchy of structures

- Common theorems should be written (and proved) only once
- There should be a mechanism to inherit theorems for types that respect the right structure
  - Type classes
  - Canonical structures

Example canonical structure

```
Require Import Arith ZArith List Bool.

Structure eqtype :=
  { sort : Type;  eq_op : sort -> sort -> bool;
    eq_prop : forall x y, eq_op x y = true <-> x = y}.

Definition count (T : eqtype) (v : sort T):
  list (sort T) -> nat :=
  fold_right
    (fun x r => if eq_op T x v then 1 + r else r) 0.

Fail Check count _ 2 (2 :: 4 :: 5 :: 2 :: nil).

Canonical nat_eqtype := Build_eqtype nat Nat.eqb Nat.eqb_ec

Check count _ 2 (2 :: 4 :: 5 :: 2 :: nil).
```

Example continued

```
Fail Check count _ 2%Z (2 :: 4 :: 5 :: 2 :: nil)%Z.

Canonical Z_eqtype := Build_eqtype Z Z.eqb Z.eqb_eq.

Check count _ 2%Z (2 :: 4 :: 5 :: 2 :: nil)%Z.
```

Characteristic of Mathematical Components

- Exploit proof irrelevance where it can be proved
  - Types with decidable equality
  - Finite types, etc
- Proposes its own set of tactics
  - Intensive use of rewriting, unfolding
  - Make it easy to exploit changes of point of view

Example

Example with matrices

- Computing the determinant of a matrix

Mathematical idea

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

- The determinant is $(-1)^{n+1}$
- the proof relies on expansion on the first column

Defining the matrix

```
Definition rotmx n : 'M[K]_n :=
  \matrix_(i < n, j < n) (((i.+1 %% n) == j)%N)%:R.
```

- `i` and `j` are bound in the `\matrix` notation
- `i` and `j` are bounded natural numbers
- Coerced silently into natural numbers for the modulo operation `%%`
- comparison with `j` is at `natural number level`
- The boolean is silently coerced to 1 or 0
- Then coerced explicitly into the field `K` using the `%:R` notation
  - The latter will be silent in the future

Demo on a fixed dimension

- computing the determinant for the matrix of size 2
- Use of `expand_det_col` giving a natural number to choose the column
- Use of `mxE` to view a matrix as a function of the two indices
- Use of `big_ord_recr` to remove elements of the sum one by one
- Use of theorems for ring structures, inherited by the field `K`
- Use of `\=` to cleanup computations and notations

Demo on an arbirary dimension

- No use of induction
- After expandin on the first column, use a theorem that distinguishes a given term of the sum
- Use of a generic lemma for a big iteration on the neutral element
- Need to show that all terms are 0
- use the fact that a bounded integer is smaller than the bound
- Need to show that the last cofactor is a multiple of the identity matrix
- Computation on a submatrix, reasoning on index shifts