Well-founded recursion done right

(Coq programming pearl)

Xavier Leroy
Collège de France, PSL University
Paris, France
xavier.leroy@college-de-france.fr

Abstract

Several Coq libraries and extensions support the definition of non-structural recursive functions using well-founded orderings for termination. After pointing out some drawbacks of these existing approaches, we advocate going back to the basics and defining recursive functions by explicit structural induction on a proof of accessibility of the principal recursive argument.

1 Introduction

Coq and other proof assistants featuring inductive predicates provide excellent support for proofs by well-founded induction: to prove $\forall x, P(x)$, just show $P(x)$ assuming the induction hypothesis $\forall y < x, P(y)$, where $<$ is a well-founded ordering. The proof is constructive and proceeds by structural induction on a proof term for the "accessibility" of x, namely the inductive fact that all descending chains originating from x are finite.

In this note, we focus on well-founded *recursion*: the definition of recursive functions that terminate not because recursive calls are made on structural subterms of the principal argument, but because they are made on principal arguments that are strictly smaller in a well-founded ordering. Examples of such functions include computing greatest common denominators by Euclid's algorithm (shown below), computing least fixed points by Tarski iteration (appendix A), and DFS traversal of a DAG (appendix B).

2 Evaluating the existing approaches

When developing verified software in Coq, such as CompCert [4], function definitions are held to more stringent legibility requirements than proof terms. I found the following properties to be highly desirable:

1. *The function definition should be easy to read for a functional programmer and make it obvious which algorithm is used.* This rules out most definitions by tactics. Even the use of fixed-point combinators such as the `Fix` combinator from Coq's standard library hurts legibility.
2. *It should be easy to prove properties of the function after its definition.* We should not be forced to prove all needed properties during the definition. This is a known problem with the `Program` mechanism.
3. *The function definition and the proofs of its properties should not involve axioms.* This is a known problem for curried multiple-argument functions defined with the `Fix` combinator: the `Fix_eq` proof rule has an extensionality hypothesis that cannot be proved without the function extensionality axiom.
4. *Extraction should generate OCaml code that contains no extraneous computations and looks natural to an OCaml programmer.* Tactics tend to generate awful code. `Program Fixpoint` and `Equations` leave unnecessary and noisy `let` bindings in extracted code.

Table 1 evaluates five known ways to implement well-founded recursion in Coq with respect to these properties: tactics in proof mode [2, §16.4] [3, §7.1], the `Fix` combinator from Coq's standard library [1], the `Function` mechanism [6, §4.1.4], the `Program Fixpoint` mechanism [6, §2.2.9], and the `Equations` package [5]. Sadly, the only way that ticks all the box is `Function`, which is documented as "legacy functionality" [6, p. 585], and its claimed successor `Equations` is not quite as good extraction-wise yet.

3 Proposed approach

After 20 years of using and teaching a mixture of `Fix`, `Function` and `Program Fixpoint`, I realized that there is a simpler approach to well-founded recursion that meets requirements 1–4 above and requires no extension to Coq: just go back to the basics, namely structural recursion over accessibility proofs.

Here is an example for the GCD function. Start with the naive recursive definition that is rejected because it is not structurally recursive:

```
Fail Fixpoint gcd_rec (a b: nat): nat :=
  if Nat.eq_dec b 0 then a
                    else gcd_rec b (a mod b).
Fail Definition gcd (a b: nat): nat :=
  if b <=? a then gcd_rec a b else gcd_rec b a.
```

Then, add an argument that is a proof of accessibility of `b`, the argument that decreases at recursive calls. The function becomes structurally recursive on this proof.

```
Fixpoint gcd_rec (a b: nat) (ACC: Acc lt b)
                 {struct ACC}: nat :=
  if Nat.eq_dec b 0 then a
                    else gcd_rec b (a mod b) _ .
```

	Tactics	Fix	Function	Program	Equations	This paper
(1) Legibility of definitions	✘	≈	✔	✔	✔	✔
(2) Ease of proving properties	≈	≈	✔	✘	✔	✔
(3) No axioms used	✔	✘	✔	✘	✔	✔
(4) Legibility of extracted code	✘	✔	✔	✘	✘	✔

Table 1. Comparing various approaches to well-founded recursive definitions. ✔: good, ≈: passable, ✘: bad.

```
Definition gcd (a b: nat): nat :=
  if b <=? a then gcd_rec a b (lt_wf b)
             else gcd_rec b a (lt_wf a).
```

Now it remains to fill the hole _ with a proof of accessibility of `a mod b`. The proof must be transparent and return a structural subterm of `ACC`, otherwise Coq rejects the recursion as not structural. A reliable way to ensure this is to explicitly use the `Acc_inv` lemma from the Coq standard library:

```
Fixpoint gcd_rec (a b: nat) (ACC: Acc lt b)
                 {struct ACC}: nat :=
  if Nat.eq_dec b 0 then a else
    gcd_rec b (a mod b) (Acc_inv ACC _).
```

For reference, `Acc_inv` is just the inversion principle for the `Acc` inductive predicate:

```
Acc_inv: forall (A: Type) (R: A -> A -> Prop) (x: A),
         Acc R x -> forall y: A, R y x -> Acc R y
```

Now, the hole in `gcd_rec` is any proof of `a mod b < b`, and it can be opaque. I often use `Program` to fill these holes, but an explicit proof works well too, especially for teaching purposes.

```
Remark gcd_oblig:
  forall (a b: nat) (NE: b <> 0), a mod b < b.
Proof.
  intros. apply Nat.mod_bound_pos; lia.
Qed.
Fixpoint gcd_rec (a b: nat) (ACC: Acc lt b)
                 {struct ACC}: nat :=
  match Nat.eq_dec b 0 with
  | left EQ => a
  | right NE => gcd_rec b (a mod b)
                 (Acc_inv ACC (gcd_oblig a b NE))
  end.
```

Properties of the recursive function `gcd_rec` are easily proved by well-founded induction. It is tempting to do an induction on the `ACC` accessibility proof argument, to mimick the structure of the function definition, but this quickly runs into the usual problems with dependent induction. Instead, we just do well-founded induction over the decreasing argument `b`, followed by a destruction of the `ACC` argument and simplification, which unrolls the recursive definition — just like the `Fix_eq` lemma from the Coq standard library but without the unprovable-without-axioms extensionality hypothesis.

```
Lemma gcd_rec_correct: forall b a ACC,
  b <= a -> is_gcd (gcd_rec a b ACC) a b.
Proof.
  induction b using (well_founded_induction lt_wf);
  intros; destruct ACC; simpl.
  destruct (Nat.eq_dec b 0).
- (* base case *)
- (* recursive case *)
Qed.
```

Finally, extraction produces optimal OCaml code, just by erasing the `ACC` argument because it has sort `Prop`.

```
let rec gcd_rec a b =
  if Nat.eq_dec b O then a else
    gcd_rec b (Nat.modulo a b)
let gcd a b =
  if Nat.leb b a then gcd_rec a b else gcd_rec b a
```

4 Conclusions

That's all there is to it! I believe this simple, back-to-the-basics approach makes non-structural recursive functions in Coq less mysterious and more accessible, both in the classroom and for developing verified software in Coq.

References

[1] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In *Theorem Proving in Higher Order Logics, TPHOLs 2000*, volume 1869 of *LNCS*, pages 1–16. Springer, 2000.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[3] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2022.

[4] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[5] Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in Coq. *Proc. ACM Program. Lang.*, 3(ICFP), 2019.

[6] The Coq Development Team. The Coq reference manual – release 8.18.0. https://coq.inria.fr/doc/V8.18.0/refman, 2023.

A Additional example: Computing least fixed points by Tarski iteration

```
(* A semi-lattice *)
Variable A: Type.
Variable bot: A.
Variable le: A -> A -> Prop.
Hypothesis le_trans: forall x y z, le x y -> le y z -> le x z.
Hypothesis le_bot: forall x, le bot x.
Hypothesis eq_dec: forall (x y: A), {x=y} + {x<>y}.

(* There are no infinite ascending chains *)
Let gt (x y: A): Prop := x <> y /\ le y x.
Hypothesis wf: well_founded gt.

(* A monotonic operator *)
Variable F: A -> A.
Hypothesis F_mono: forall x y, le x y -> le (F x) (F y).

(* Tarski iteration from a pre-fixed point [x] *)
Program Fixpoint iter (x: A) (PRE: le x (F x)) (ACC: Acc gt x) {struct ACC}: A :=
  let x' := F x in
  match eq_dec x x' return _ with
  | left EQ  => x
  | right NE => iter x' _ (Acc_inv ACC _)
  end.
Next Obligation. split; auto. Qed.

Program Definition lfp: A := iter bot _ (wf bot).

(* The result is a fixed point. *)
Theorem lfp_eq: F lfp = lfp.
Proof.
  unfold lfp. generalize bot lfp_obligation_1 (wf bot).
  induction bot0 using (well_founded_induction wf).
  rename bot0 into x. intros PRE ACC. destruct ACC; simpl.
  destruct (eq_dec x (F x)).
- auto.
- apply H. split; auto.
Qed.

(* The result is the smallest fixed point. *)
Theorem lfp_least: forall z, F z = z -> le lfp z.
Proof.
  (* exercise *)
Qed.
```

B Additional example: DFS traversal of an acyclic graph

```
(* A directed graph *)
Variable node: Type.
Variable successors: node -> list node.
Variable eq_node: forall (x y: node), {x=y} + {x<>y}.

(* The 'x is a successor of y' relation *)
```

```
Definition succ (x y: node): Prop := In x (successors y).

(* The graph is acyclic iff there are no infinite chains of successors *)
Hypothesis acyclic: well_founded succ.

(* Preorder enumeration of the nodes, using DFS traversal *)
(* Note that the recursive call to [dfs] must have its accessibility argument
   of the form [Acc_inv A _] so that it is recognized as a structural subterm of [A].
   This leads to using [incl l (successors x)] as the precondition for
   the iteration over the list of successors of [x]. *)

Program Fixpoint dfs (x: node) (accu: list node) (A: Acc succ x) {struct A}: list node :=
  if In_dec eq_node x accu then accu else
    let fix dfs_list (l: list node) (accu: list node) {struct l} :
                    incl l (successors x) -> list node :=
      match l return _ with
      | nil => fun INCL => accu
      | y :: l => fun INCL => dfs_list l (dfs y accu (Acc_inv A _)) _
      end in
    x :: dfs_list (successors x) accu _ .
Next Obligation. apply INCL. simpl; auto. Qed.
Next Obligation. eapply incl_cons_inv; eauto. Qed.
Next Obligation. apply incl_refl. Qed.

(* A key property of [dfs]. *)

Definition succ_closed (l: list node): Prop :=
  forall x y, In x l -> succ y x -> In y l.

Lemma dfs_correct: forall x l A,
  succ_closed l ->
  forall y, In y (dfs x l A) <-> In y l \/ clos_refl_trans _ succ y x.
Proof.
  (* exercise *)
Qed.
```