Manifest Termination[*]

Assia Mahboubi[1] and Guillaume Melquiond[2]

[1] Nantes Université, École Centrale Nantes, CNRS, Inria, LS2N, UMR 6004, 44000 Nantes, France
[2] Université Paris Saclay, CNRS, ENS Paris Saclay, Inria, LMF, 91190 Gif-sur-Yvette, France

In formal systems combining dependent types and inductive types, such as the Coq proof assistant [8], non-terminating programs are frowned upon. They can indeed be made to return impossible results, thus endangering the consistency of the system [7], although the transient usage of a non-terminating Y combinator, typically for searching witnesses, is safe [4]. To avoid this issue, the definition of a recursive function is allowed only if one of its arguments is of an inductive type and any recursive call is performed on a syntactically smaller argument. If there is no such argument, the user has to artificially add one, *e.g.*, an accessibility property. Free monads can still be used to address general recursion [6] and elegant methods make possible to *extract* partial functions from sophisticated recursive schemes [2, 5]. The latter yet rely on an inductive characterization of the domain of a function, and of its computational graph, which in turn might require a substantial effort of specification and proof.

This leads to a rather frustrating situation when computations are involved. Indeed, the user first has to formally prove that the function will terminate, then the computation can be performed, and finally a result is obtained (assuming the user waited long enough). But since the computation did terminate, what was the point of proving that it would terminate? This abstract investigates how users of proof assistants based on variants of the Calculus of Inductive Constructions could benefit from manifestly terminating computations. A companion file showcasing the approach in the Coq proof assistant is available on-line [1].

Iteration. Traditional call-by-value programming languages allow a `fix` operator:

```
let rec fix f x = f (fix f) x
```

As this definition is typically forbidden in our setting, we resort to a more domain theoretic approach, so as to enable reasoning about a term y such that $\mathtt{fix}\,F\,x$ terminates on y for a certain x. Consider two types T and U and a function $F : (T \to U) \to (T \to U)$. Given an integer n, a variable $k : T \to U$, and an input $x : T$, the computation of $F^n\,k\,x = (F \circ \ldots \circ F)\,k\,x$ reduces to a value y of type U. If no occurrence of the variable k appears in y, then y is also the result of $F^m\,k\,x$ for any $m \geq n$, which we denote $F^*\,x \rightsquigarrow y$.

Since Coq can compute the normal form of $F^n\,k\,x$ (*i.e.*, `Nat.iter n F k x`) for some concrete n, the property $\forall k,\ F^n\,k\,x = y$ holds by reflexivity. But this equality is only a means to an end. The next step is to prove some properties about y so that it can be used inside some other proof. For a predicate $P : T \to U \to \mathit{Prop}$ that relates inputs and outputs, and from the fact $F^*\,x \rightsquigarrow y$, we can derive $P\,x\,y$ by applying Lemma 1 (whose proof is actually trivial).

Lemma 1. *If* $\exists f, \forall x, P\,x\,(f\,x)$ *and* $\forall k, (\forall x, P\,x\,(k\,x)) \Rightarrow \forall x, P\,x\,(F\,k\,x)$, *then* $\forall n\,x\,y, (\forall k, F^n\,k\,x = y) \Rightarrow P\,x\,y$.

As an illustration, we define an efficient implementation of factorial over binary relative integers (type `Z` in Coq) and, using Lemma 1, we easily prove that, when it terminates on a non-negative input, it does indeed compute its factorial. We then use this to prove a definitional

[*]This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101001995).

identity whose type-checking would take astronomical time. Note that at no point do we need to formally prove that `factZ` terminates over non-negative integers to use it. It just does.

```
Definition factZ k n := if Z.eqb n 0 then 1 else Z.mul n (k (n - 1)).
Lemma factZ_spec n x y : (forall k, Nat.iter n factZ k x = y) → (0 <= x) →
   Z.of_nat ((Z.to_nat x)!) = y. (* n! defines the reference factorial, on type nat *)
Goal Z.of_nat ((Z.to_nat 15)!) = 1307674368000. Proof. exact: (factZ_spec 20). Qed.
```

Accessibility. The previous approach is effective, but it hinges on the fact that one can exhibit a function f such that $\forall x, P\,x\,(f\,x)$. (Above, we used `fun x => Z.of_nat ((Z.to_nat x)!.`) The companion file illustrates how to compute the function McCarthy91 using its recursive definition, but using the non-recursive equivalent definition as f.

In general, the function f is recursive, and thus one has to prove its termination, which might in some cases be just as hard as proving the termination of F^* itself. Those cases thus require a different approach. Consider the following Coq function, inspired from Charguéraud [3]. (See the companion file for the exact but less readable term.)

```
Definition fixacc (dummy : U) F (R : T → T → bool) (x : T) : Acc R x → U :=
  Acc_rect (fun x k ⇒ F (fun y ⇒ if R y x then k y else dummy) x).
```

In addition to the already known arguments F and x, this function takes a Boolean relation R and a proof that x is accessible using this relation (*i.e.*, no infinite decreasing chain from x). This accessibility proof is used to fuel the recursive calls. If at some point on input u, F tries to perform a recursive call with input v, the inner function will check that v is indeed smaller (*i.e.*, $R\,v\,u = \mathit{true}$). If so, it allows the recursive call $k\,v$. Otherwise, it returns a dummy value from U. Note that in practice U will be non-empty, for there is a y such that $F^*\,x \rightsquigarrow y$.

The relation R represents the call graph of the computation $F^*\,x \rightsquigarrow y$. More precisely, $R\,v\,u = \mathit{false}$ if and only if F was invoked on u but it did not perform a direct recursive call with input v. In other words, the above function behaves the same as `Acc_rect (fun u k => F k u)`, which is just F^*. As for the accessibility of x by R, it is a consequence of the fact that the computation actually terminated and thus that the call graph is finite and acyclic. This gives rise to the following consistent axiom, where `fixrel F x y` stands for $F^*\,x \rightsquigarrow y$:

```
Parameter fixrel : forall {T U}, ((T → U) → T → U) → T → U → Prop.
Axiom fixrel_spec : forall {T U} F x y, fixrel F x y → exists R,
  (forall v k1 k2, (forall u, R u v = true → k1 u = k2 u) → F k1 v = F k2 v) /\
  exists W, forall u, y = fixacc u F R x W.
```

Using this axiom, it becomes possible to prove a lemma such as “`fixrel factZ x y -> 0 < y`” without first exhibiting a terminating function or proving termination of `factZ`. Instead, proofs go by unfolding `fixacc`, and by exhibiting two functions k_1 and k_2 that are equal for every input v except when $R\,v\,u = \mathit{false}$. This way, we construct an *ad hoc* proof for every sensible F, *i.e.*, for functionals that do not perform any irrelevant recursive call (*e.g.*, $k\,x - k\,x$).

Conclusion. Checking the antecedent `(fixrel F x y)` of the axiom `fixrel_spec` is easy to do in the kernel of Coq, *e.g.*, in the bytecode interpreter. This could obviously lead to non-terminating computations when checking proofs, but from the user's viewpoint, there is not much difference between computations that cannot terminate and computations that take too long to terminate.

At this point, the open questions are whether there exists a simpler version of this axiom, and whether one can deduce from it a generic variant of Lemma 1.

References

[1] https://fresco.gitlabpages.inria.fr/divers/fix.v.

[2] Ana Bove and Venanzio Capretta. A type of partial recursive functions. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *21st International Conference in Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 102–117. Springer, August 2008.

[3] Arthur Charguéraud. Proof pearl: A constructive fixed point combinator for recursive functions, March 2009.

[4] Herman Geuvers, Erik Poll, and Jan Zwanenburg. Safe proof checking in type theory with Y. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, volume 1683 of *Lecture Notes in Computer Science*, pages 439–452. Springer, 1999.

[5] Dominique Larchey-Wendling and Jean-François Monin. The Braga method: Extracting certified algorithms from complex recursive schemes in Coq. In *Proof and Computation II*, pages 305–386. World Scientific, August 2021.

[6] Conor McBride. Turing-completeness totally free. In Ralf Hinze and Janis Voigtländer, editors, *12th International Conference on Mathematics of Program Construction*, volume 9129 of *Lecture Notes in Computer Science*, pages 257–275. Springer, June 2015.

[7] Pierre-Marie Pédrot and Nicolas Tabareau. The fire triangle: how to mix substitution, dependent elimination, and effects. *Proc. ACM Program. Lang.*, 4(POPL):58:1–58:28, 2020.

[8] The Coq Development Team. The Coq proof assistant, September 2022.