Towards a formalization of the guard condition

A translation from guarded to well-founded recursion

CYPRIEN MANGIN, Inria Paris & IRIF - Université Paris 7 Diderot, France
MATTHIEU SOZEAU, Inria Paris & IRIF - Université Paris 7 Diderot, France

We present a translation of guarded recursive functions in Coq to well-founded recursive functions using only case analysis eliminators and the eliminator of the accessibility predicate. This work-in-progress is a possible first step towards a formalization of Coq's guard condition. We also present an idea to extend the guard recursion to handle inductive-inductive definitions.

1 INTRODUCTION

As a programmer, Coq's guard condition[1] is a nice feature that allows to naturally write recursive definitions on arbitrarily deep terms, resulting in readable and easy to write functions. As a proof theorist, it is a regular source of distrust or even unsoundness, as it has changed a lot since its original formulation and justification in [3].

We propose a way to alleviate such concerns by systematically translating guarded definitions to well-founded recursive ones, reducing the trust one needs to place into the guard condition. In short, to each variable deemed a subterm by the guard condition checker, we adjoin a proof that it is indeed a subterm suitable for a recursive call. If this work is properly formalized, the user would only need to trust well-founded recursion to be able to trust guarded definitions. At the same time, it would not increase the complexity of learning Coq, as the user can still lean on the same guard definition they are accustomed to.

2 AN EXAMPLE

To illustrate our intent, we will show how to translate the example originally used by [3]. We start by defining the predicates of evenness and oddness of natural numbers used in this example:

```
Inductive Even : nat → Prop := zeven : Even O | seven {n : nat} : Even n → Even (S (S n)).
Inductive Odd : nat → Prop := oneodd : (Odd (S O)) | sodd {n : nat} : Odd n → Odd (S (S n)).
```

This is the predicate we want to prove for any natural number, separating even and odd ones:

```
Definition EO (n : nat) := {Even n} + {Odd n}.

Definition aux {n : nat} (H : EO n) : EO (S (S n)) := match H with
    left p ⇒ left (seven p) | right p ⇒ right (sodd p)
  end.
```

Using the above auxiliary definition, we can define the following recursive function, which is guarded in Coq even though the recursion is 2 levels deep.

```
Fixpoint evod (n : nat) : EO n := match n with
  | O ⇒ left zeven
  | S m ⇒ match m with
          | O ⇒ right oneodd
          | S p ⇒ aux (evod p)
          end end.
```

We will now show how our translation would operate on such a definition. First we need a subterm relation on the recursive argument of the function, allowing recursion on any structurally smaller subterm. We can derive it automatically with a tool like Equations [4] using: `Derive Subterm for nat.` The subterm relation is a strict one, we will also need one that allows equality to talk about the recursive argument of the function.

```
Definition nat_subterm_eq : relation nat := fun x y ⇒ nat_subterm x y ∨ x = y.
```

Finally, for any inductive type we can define a specialized eliminator that will take a – possibly equal – subterm, and produce constructor arguments that will be considered strict subterms. The first argument of this eliminator, N, is the original recursive argument.

[1] The reference manual is available for a more precise definition: https://coq.inria.fr/refman/language/cic.html#id27

```
Definition nat_case (N : nat) (x : nat) (Hsub : nat_subterm_eq x N)
  (P : nat → Type) (fO : P O) (fS : ∀ (y : nat), nat_subterm y N → P (S y)) : P x.
```

With these tools, we can build our translation in a way which we believe is mechanical enough to be automated. Each case on a recursive argument is replaced by a call to nat_case, allowing to thread the proof that it is a subterm of the original recursive argument.

```
Definition evod_body (n : nat) (F : ∀ (m : nat), nat_subterm m n → EO m) : EO n :=
  nat_case n n (or_intror eq_refl) EO
    (left zeven) (fun m Hsub ⇒ nat_case n m (or_introl Hsub) (fun m ⇒ EO (S m))
      (right oneodd) (fun p Hsub ⇒ aux (F p Hsub))).
```

The function evod_body takes as an argument a prototype for itself. We can then use the Fix combinator provided by Coq to tie the knot and build the actual function, extensionally equal to the first one.

```
Definition evod' := Fix well_founded_nat_subterm EO evod_body.

Theorem evod_evod' (n : nat) : evod n = evod' n.
```

We conjecture that it will also be intensionally equal, in the sense that the subterm well-foundedness proof is always "as defined" as the natural number argument itself.

3 EVOLUTION OF THE GUARD CONDITION

Coq's guard condition has changed over the years, diverging from [3]'s presentation and sometimes introducing soundness bugs. For instance, the most recent fix in the guard condition was related to an extension that allows a match to be considered a subterm if all its branches are subterms. Formulated in such a lax way, it allows for definitions which contradict propositional extensionality. The chosen correction was to forbid this extension, unless the type returned by the case was mutually inductive with the inductive type of the recursion.

The translation we propose would have supported this fix – and disallowed the first extension. Indeed, we adjoin a proof of being a subterm to each subterm. In this case the proof would have to be threaded through the type returned by the case, which is not possible unless this type is already mutually inductive.

A formalization of this translation and the guard condition would help in case we wish to extend it again. We are working towards such a mechanization as part of Template-Coq's [1] formalization of Coq's type system.

4 OPENING TO INDUCTIVE-INDUCTIVE TYPES

We also wish to present our current work on extending the guard condition to support inductive-inductive types. The work done in [2] already provides a theoretical basis to define them and their eliminators, but in the case of Coq we would like to be as liberal as with inductive types and allow recursive calls on arbitrarily deep subterms.

In a nutshell, we simply propose to consider the index of an inductive-inductive value to be considered a subterm of said value. The set-theoretical model of inductive-inductive types in [2] gives a justification for this choice and our experiments show that it would allow to properly define eliminators for inductive-inductive types.

5 CURRENT AND FUTURE WORK

This work is still in an experimental state. As such, the formalization of the guard condition is in its early stages, starting with rewriting and verifying in Coq some OCaml libraries that are used by the positivity checker and the guard checker in the kernel, notably a complex regular tree datastructure.

For the theoretical aspects, we need to make sure that our translation will extend properly to mutual inductive types – which we did not mention here – as well as inductive-inductive types.

REFERENCES

[1] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Towards Certified Meta Programming with Typed Template-Coq. In *Interactive Theorem Proving, ITP'18*.

[2] Fredrik Nordvall Forsberg and Anton Setzer. 2010. Inductive-inductive definitions. In *International Workshop on Computer Science Logic*. Springer, 454–468.

[3] Eduarde Giménez. 1994. Codifying guarded definitions with recursive schemes. In *International Workshop on Types for Proofs and Programs*. Springer, 39–59.

[4] Matthieu Sozeau and Cyprien Mangin. 2017. Equations 1.0 for Coq 8.7. (Dec. 2017). https://doi.org/10.5281/zenodo.1117298