40 years of Guard Conditions

Thomas Lamiaux
Nantes Université, Inria
France

Yann Leray
Nantes Université, Inria
France

Yee Jian Tan
KU Leuven
Belguim

Yannick Forster
Inria
France

Matthieu Sozeau
Inria
France

Nicolas Tabareau
Inria
France

Abstract

The Rocq-prover is a functional programming language and logical system based on fixpoints and pattern-matching. To be consistent, Rocq only accepts functions that terminate. This is enforced by a syntactic check named the "guard condition", that ensures that every recursive call is performed on a smaller term than the original one. Despite its critical role, the guard condition is little understood: there is no mathematical justification for it, nor formal specification of it. Moreover, its implementation consist in about 2000 lines of intricate OCaml code, that even experts struggle to understand. This lack of understanding has led to inconsistency bugs due to both implementation and theoretical issues, and prevents any further modifications.

With this work, we provide the first formal specification of the guard condition as it should be implemented. We further build knowledge on guard conditions by introducing its different features one by one, and discussing their motivations, limitations and related historical bugs. Based on this specification, we provide a reimplementation of the guard condition in MetaRocq.

1 Introduction

Inductive Types in Rocq. Inductive types are one of the most useful features of programming languages and of proof assistants. They are useful both to define data structures and to state specifications. For instance, one can define the usual `list` data type, but also a predicate `even`:

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A → list A → list A.

Inductive even : nat → Prop :=
| even_O  : even 0
| even_SS : ∀n, even n → even (S (S n)).
```

As in functional programming languages, to define functions over inductive types, Rocq has primitive fixpoints and pattern-matching:

```
Fixpoint map {A B} (f : A → B) (l : list A) :=
  match l with
  | []     ⇒ []
  | a :: l' ⇒ f a :: map f l'
  end.
```

Non-Terminating Functions. To be logical systems, proof assistants must be consistent: `False` must not be provable in the empty context. To that end, proof assistants refuse non-terminating functions as they generally enable one to prove `False`:

```
Fixpoint loop (n : nat) : False := loop n.
```

Accepting a non-terminating function does not necessarily imply that a language is inconsistent. Yet, experience with Rocq and Agda has shown that the smallest implementation mistake that lets a non-terminating function slip through can be exploited to prove `False`. So far, this has not proven to be problematic as users do not seem to naturally stumble on these subtle implementation mistakes, and exploit them. However, this is crucial in an adversarial context as an opponent could rely on a bug to prove `False`, and have an incorrect certificate accepted. Even without ill intent, this is worth considering in the age of LLM-generated proofs: LLMs can attack the codebase to try to prove `False` when faced with a statement that is too difficult or impossible for them to prove. At the time of writing, LLMs do not seem to be able to prove `False` on their own, but Claude 4.6 can already find subtle implementation bugs when directed by Rocq developers, who succeeded in exploiting them to prove `False` [1]. Consequently, it is fundamental to have a well-designed and mathematically justified mechanism for rejecting non-terminating functions.

Guard Condition. In Rocq, non-terminating functions are filtered out by a *guard condition*, a syntax check performed when type-checking fixpoints. Conceptually, a guard condition is rather simple: it is a syntax check that goes through terms, locates all recursive calls, and checks that they are performed on smaller arguments than the original one. It gets complicated when trying to decide what a subterm should be, how subterm information should be propagated through constructors and how it interacts with reduction. An overly simple guard condition would be a hassle to users as it would not accept functions and proofs as users naturally write them. For instance, it is natural to write functions as fixpoints and writing proofs with tactics naturally creates redexes; neither would be accepted by a guard condition that only accepts eliminators. Therefore, one would like a guard condition

as expressive and versatile as possible. Yet, in the absence of a mathematical justification, every extension is at the risk of threatening consistency by letting non-terminating functions slip through.

Rocq's Guard Condition. Since its initial implementation, Rocq's guard condition has been modified several times over the years to extend it, and to fix bugs introduced by previous changes. Most of these modifications have been implemented without being supported by a publication or a mathematical artifact, and were only discussed through GitHub pull requests – when they existed. As a result, Rocq's guard condition is not only lacking a mathematical justification, but also a specification.

From a historical point of view, it is difficult to reconstruct the intent and moral justification of each modification. It is also complex to identify the exact content of each modification as some commits both modified the guard condition and partially reimplemented it at once. As far as we understand, excluding implementation bug fixes, the guard condition evolved as follows:

199x The first versions of the guard condition were developed and implemented from 1994 to 1999 [8], and supported:
- Functions on (nested) inductive types
- Checking terms modulo weak-head reduction
- Complex subterm analysis going through fixpoints and pattern-matching

2010 Pierre Boutillier extended the guard condition to propagate subterm information through β-ι cuts [2] – a match returning a function applied to a term. This was reported in 2012 in Boutillier [3].

2013 It was noticed by Daniel Schepler that the support for β-ι cuts introduced in 2012 is incompatible with the axiom of Propositional Extensionality (PropExt) [10]. In 2014, Maxime Dénès restricted the support for β-ι to restore compatibility with PropExt [4].

2022 Checking terms up to weak-head normalization does not ensure strong normalization as non-terminating subterms can be erased [5], even though they would be evaluated with another strategy. To restore strong normalisation, Hugo Herbelin implemented a reduction mechanism to check terms up to reduction without forgetting to check erasable subterms [6].

2024 Hugo Herbelin added hoisting of uniform arguments of fixpoints to enable users to write functions on nested inductive types more modularly [7].

This complex history makes it unrealistic to understand what the guard condition should do through its history. Unfortunately, it is not easier to understand the current implementation as the different features have not been designed and implemented modularly. They are stacked on each other, when not intertwined, making it challenging to understand what the guard condition is supposed to do, let alone verify it. This is not helped by the implementation, which consists of about 2000 lines of complex OCaml code, which has evolved organically over the years, presenting different styles and layers of complexity.

Overall, this makes the current guard condition difficult to understand even for current Rocq developers who inherited it, and regularly confusing to users who fail to understand why some functions are accepted while others are rejected. Unsurprisingly, this lack of understanding and complex implementation have led to many inconsistency bugs due to both implementation and theoretical issues. It is actually remarkable how the tiniest oversight or typo in the implementation can be exploited to prove `False`. Moreover, the absence of a specification further prevents any modifications or even rewriting, as this could introduce new unforeseen bugs.

Contributions. In this work, we set the first stepping stone towards formalizing a realistic guard condition for proof assistants based on fixpoints and pattern-matching. Reconstructing years of unwritten evolutions, we provide the first formal specification of Rocq's guard condition as it is believed to be currently implemented in Rocq's kernel. That is, our specification has been agreed upon by Rocq developers, and in case of a bug, this specification is checked to have been implemented properly. Moreover, as a proof of concept, following our specification, we reimplemented in MetaRocq a guard condition.

We identify that the guard condition can be broken down into four main parts that we introduce and discuss gradually:

1. A minimal guard condition that should only accept functions written directly using the eliminators of nested inductive types
2. An extension to propagate subterm information through β-ι cuts compatible with Propositional Extensionality
3. A reduction mechanism to perform as much reduction as possible without forgetting to check erasable subterms
4. An extended subterm analysis that goes through fixpoints and pattern-matching

We discuss and showcase the motivation of each feature, its importance and limitations, and the main bugs that affected them over the years. In particular, we discuss the famous incompatibility bugs of the full-blown support of β-ι with PropExt and its fix, which remains little understood, despite proposals to extend its support [9]. Doing so, we shed light on a little-understood though crucial piece of software, and a component that can be useful to other proof assistants, either for termination checking or elaboration. In particular, while writing this specification, we have found a few bugs in Rocq's guard condition that we discuss in this paper.

References

[1] 2025. [Zulip Rocq] Proof of false found by Opus 4.6 and mxdys (bbchallenge). https://rocq-prover.zulipchat.com/#narrow/channel/237977-Rocq-users/topic/Proof.20of.20false.20found.20by.20Opus.204.2E6.20and.20mxdys.20.28bbchallenge.29/with/580830257

[2] Pierre Boutillier. 2010. Applicative commutative cuts in Fixpoint guard condition. https://github.com/rocq-prover/rocq/commit/e7c9f5f130

[3] Pierre Boutillier. 2012. A relaxation of Coq's guard condition. In *JFLA - Journées Francophones des langages applicatifs - 2012*. Carnac, France, 1 – 14. https://hal.science/hal-00651780

[4] Maxime Dénès. 2014. Tentative fix for the commutative cut subterm rule. https://github.com/coq/coq/commit/9b272a861bc3263c69b699cd2ac40ab2606543fa. https://github.com/coq/coq/commit/9b272a861bc3263c69b699cd2ac40ab2606543fa

[5] Andres Erbsen. 2017. Fixpoint termination check allows unused recursive calls. https://github.com/rocq-prover/rocq/issues/6487

[6] Hugo Herbelin. 2022. Check guardedness of fixpoints also in erasable subterms. https://github.com/coq/coq/pull/15434

[7] Hugo Herbelin. 2024. Extrude uniform parameters of inner fixpoints in guard condition check. https://github.com/coq/coq/pull/17986

[8] Hugo Herbelin. 2024. Size-preserving dependent elimination. https://pauillac.inria.fr/~herbelin/talks/talk-types24-2.pdf

[9] Hugo Herbelin. 2024. Size-preserving dependent elimination. (2024), 35-37 pages. https://types2024.itu.dk/abstracts.pdf

[10] Daniel Schepler. 2013. [Coq-Club] bijective function implies equal types is provably inconsistent with functional extensionality in Coq. https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00114.html