Towards Formalising the Guard Checker of Coq

Yee-Jian Tan

Masters Year 1 Internship Report
(improved with incoporated feedback)

Master Parisien de Recherche en Informatique (MPRI)
École Polytechnique

Advisor: Yannick Forster

Cambium, Inria Paris

March 09, 2025

Table of Contents

Appendix

Chapter 1

Introduction

Most consistency proofs of type theory crucially rely on the termination of all recursive functions, because non-terminating functions can induce inconsistency in the theory. Therefore, to ensure that all recursive functions terminate, the Calculus of Inductive Constructions (CIC) only allows structurally recursive functions. The Coq proof assistant implements a variant of CIC, therefore, its kernel has a "guard checker" that checks for the termination of recursive functions [1], and the condition it checks for is called the "guard condition".

The guard checker was first implemented in 1994 by Christine Paulin-Mohring at the time of introducing inductive types to Coq's type theory [2]. In the 30 following years, it has been extended, adapted, and fixed by different contributors. Concretely, it has been extended to e.g. allow easier definition of nested fixpoints [2], adapted to maintain strong instead of just weak normalisation [3], extended to allow β-ι commutative cuts [4], and fixed to be compatible with frequently used extensionality axioms [5]. As a result, Coq's guard checker nowadays consists of around 1,000 lines of intricate OCaml code[1] with few comments, meaning it is understood by few. As a result, there is no specification of Coq's guard condition, and proofs of its soundness are out of reach. In contrast, metatheoretical properties about the typing and reduction rules of Coq has already been verified in the MetaCoq project [6] (which we will explain shortly after), except for the guard checker, which is parameterised in MetaCoq. Therefore, understanding the guard checker will be crucial to bridge this gap.

In this internship project, we provide a full implementation of Coq's most recent guard checker in Coq using the MetaCoq project. Additionally, in this report, we provide examples that explain the inner workings of the guard checker, which can serve as documentation.

First and foremost, non-terminating fixpoints can lead to inconsistency, that is, allow proofs of the statement `False` in the empty context. If non-terminating fixpoints were allowed, one could define the function below in Coq:

```
Fixpoint boom (x : nat) : False := boom x.
```

Then, `boom` would be non-terminating for any input, and, for example, `boom 0` has the type exactly as False. This would make Coq inconsistent.

To understand how guard checking in Coq is done, first, we need to understand the "data" recursive functions compute on: inductive types. In general, inductive types encode well-founded mathematical objects, such as natural numbers, lists, vectors, trees, and logical predicates. The terms of an inductive type are created by the inductive type's *constructors* in a recursive manner.

There are two ways to compute on (or eliminate) the terms of an inductive type, namely by fixpoints

[1] https://github.com/coq/coq/blob/master/kernel/inductive.ml#L569

and pattern matching[2], or by the eliminator (a.k.a. recursor) of the inductive type. Therefore, type theories with inductive types could choose between these options. They can either include fixpoints natively and represent eliminators using fixpoints, or vice versa, include eliminators natively and represent fixpoints using eliminators. Using eliminators natively in the theory allows one to specify the type theory more easily, but the extracted code would be less idiomatic. Coq was designed as a system to extract code to programming languages such as OCaml, Haskell, or Scheme [7], [8]. Allowing fixpoints makes the direct extraction to these languages look more like idiomatic code. However, making fixpoints a primitive construct means that the kernel must check for the termination of fixpoints – which is why the guard checker was designed.

Despite its importance to consistency, Coq's guard checker is severely understudied. The technical concepts and data structures in the guard checker, are crucial to understanding it, but are mostly absent in published literature, making the guard checker even more obscure. Therefore, there is a need for recent and detailed documentation for the guard checker.

The guard checker is also a crucial component to formalise in the MetaCoq project. Two central parts of the MetaCoq project are a formalisation of Coq's type-theory in Coq [9] and, on top of that, verified implementations of a type checker [10] and an extraction function to OCaml [11]. The formalisation of type theory faithfully captures the typing rules of Coq in an inductive predicate, and is parameterised in a guard checker function which is required to fulfill some basic properties such as being stable under reduction and substitution. Based on this formalisation, crucial properties such as subject reduction (types are preserved by reduction) and canonicity (normal forms of inductive types start with a constructor) are proved [10].

The verified type checker axiomatically assumes that reduction in the system is strongly normalising. The strong normalisation assumption can also be used to prove consistency, because any proof of `False` would have a normal form using strong normalisation, which would have the same type using subject reduction, and would start with a constructor of the inductive type `False` using canonicity – which is a contradiction, because `False` has no constructors. Having an actual, faithful implementation of Coq's guard checker in MetaCoq will fill the crucial gap by removing the parametrisation by a guard checker, and put a stamp of verification on the guard checker's correctness. especially when the perfect setting of MetaCoq presents itself.

Therefore, this project contributes to the understanding of the guard checker via

- an implementation of Coq's most recent guard checker in Coq, using MetaCoq's representation of Coq terms;
- an exposition of various features implemented in the guard checker by examples; as well as
- an abstract specification of the guard checker on paper.

The rest of the report will be organised as follows: Chapter 2 first gives an overview of the history of the guard checker. Chapter 3 gives incremental examples of guard checking on fixpoints to illustrate the different features of the guard checker, followed by an abstract specification of its implementation in Chapter 4. Finally, Chapter 5 discusses related and future work, including development plans of a mechanised soundness proof in MetaCoq, and how it might lead to a consistency proof of Coq's type theory.

[2]In the rest of this introduction chapter, the term "fixpoint" is used synonymously with "fixpoints and match constructs", as they rely on each other to define recursive functions.

Chapter 2

History of the Guard Checker

In 1989, Frank Pfenning and Christine Paulin-Mohring introduced inductive types in the Calculus of Constructions [12], forming what is now known as the Calculus of Inductive Constructions. In 1992, Thierry Coquand introduced pattern matching in dependent types [13], more precisely in Martin Löf's Logical Framework, which served as a basis for Coq to adopt primitive fixpoints and match constructs instead of eliminators [14].

In 1994, the guard checker was first implemented in Coq v5.10.2 [2, 14]. In the same year, Eduardo Gimenez specified the guard condition implemented in Coq in the paper "Codifying Recursive Definitions with Recursive Schemes", where he gave a translation from guarded fixpoint definitions to "primitive recursive" eliminators [15], although properties such as confluence and normalization remained to be established [1].

In 1996, Paulin-Mohring documented in her habilitation thesis, "Inductive Definitions for Type Theory" a proof that guarded fixpoint operators correspond to *"well-founded recursion with respect to a well-chosen order"* [1]. In the same year, Gimenez proposed annotations in his Ph.D. thesis that, according to Paulin-Mohring in [1], capture the syntactic conditions of "strict subterms" [16], and gave a normalisation proof of the theory. To the best of our knowledge, these annotations are not included in the Coq kernel.

In 2010, Pierre Boutillier relaxed the guard checker by adding the "β-ι commutative cut subterm rule", which allowed more fixpoints on indexed inductive types (such as vectors) to pass through. This change was then written up in 2012 [4]. In 2013, Daniel Schepler found that the guard checker was incompatible with the functional extensionality axiom [17], and subsequently, Maxime Dénès refined Schepler's conterexample and found that even assuming propositional extensionality was sufficient to derive a proof of False. In 2014, Maxime Dénès restricted the extension to the guard checker in [4] to forbid the unwanted proofs, restoring Coq's compatibility with the extensionality axioms [5].

In 2022, Hugo Herbelin restricted the guard checker to ensure strong normalisation rather than just weak normalisation by checking for guardedness in erasable subterms [3]. Even though weak normalisation suffices for consistency, strong normalisation is a behaviour that seems to be more in line with the intuition of users, as well as ensuring termination in extracted code [18, 19].

In 2024, Herbelin introduced a relaxation that allows simpler implementation of nested recursive functions [20], in particular by extruding the uniform parameters of fixpoint definitions. Furthermore, changes to the guard condition keep being proposed, such as relaxing the compatibility fix for Propositional Extensionality in [5] to allow the *"compilation of pattern matching by small inversion"* [21].

Chapter 3

A Taste of the Guard Checker

This section aims to present intuitively the various features of the guard checker, leading to the detailed explanation of the addition of natural numbers in Section 3.3. To achieve this, we first explain the necessary concepts, namely the guard environment and subterm specifications in Section 3.1, and the stack of subterm specifications in Section 3.2, then the example in Section 3.3. The strongly guarded fixpoints (Section 3.4), propositional extensionality fix (Section 3.5), regular trees (Section 3.6), and the extrusion of uniform parameters (Section 3.7) will similarly be explained intuitively in the rest of this chapter. A more detailed specification of all the features mentioned in this section can be found later in Section 4.1.

3.1 Subterm Specification

The core idea of the guard checker is simple: it checks for structural recursion. One should remember the slogan: for a fixpoint,

Guarded means that recursive calls are made on (strict) subterms.

Therefore, "what constitutes a subterm" is the first question the guard checker needs to answer.

Let us start with the example of the addition fixpoint on (unary) natural numbers:

```
1  Fixpoint add (m n : nat) {struct m} :=
2    match m with
3    | O => n
4    | S m' => add m' (S n)
5    end.
```

Every fixpoint must have a **recursive parameter**, and it is `m` in the example above. The user can specify explicitly in curly braces `{struct m}`, otherwise it will be assigned during elaboration. To ensure that `add` is guarded, all recursive calls of `add` in the body must be supplied with a *strict subterm* of `m`. In other words, the **recursive argument** in every recursive call must be a strict subterm of the recursive parameter.

This property of being a "strict subterm" (or otherwise) is called the **subterm specification** of a term, *relative* to the recursive parameter `m`. A subterm specification can be assigned to any term in the body. It is either a[3]

- strict subterm (of `m`), or
- large subterm (of `m`, meaning equals `m`), or
- not subterm (unrelated to `m`).

In the body of `add`, since `m'` is obtained from `m` by pattern matching, it is a strict subterm of `m`, and so will be all subterms of `m'`. This transitivity of "strict subterm" -ness depends on previously specified terms,

[3]We omit the subterm specifications of *dead code* and *internally bound subterm* for now, see Section 4.1 for an explantion.

and is taken care of by a **guard environment**, which associates every entry in the local context with its subterm information. During the initialisation of the guard environment, `m` was specified to be a large subterm (of itself).

To specify the subterm property of a term, it will first be reduced to weak-head normal form. A term is said to be in weak-head normal form if

1. the term has a constructed form (e.g. lambda-expression, constructor, cofixpoint, inductive type, product type or sort) and is thus irreducible, or
2. the leftmost-outermost redex (the *head* redex) is "blocked" by a variable or a an opaque constant, say `x`:
   - a blocked β-redex: `x u ...`
   - a blocked δ-redex: `x`
   - a blocked ι-redex: `match x with ...` or `fix f ... x {struct x} := u`.

For example, for any term `u`, the term `id u` (applying the identity function to `u`) has the weak-head normal form of `u`. Therefore, any term `u` will have the subterm specification as `id u`.

Having seen how terms are given their subterm specification, we will now see, as a first pass, how the guard checker checks for the termination of `add`. This algorithm will be expanded in Section 3.2. The guard checker will check that `add` is guarded by recursing on the body of the fixpoint, in which only the cases of match constructs, function applications, and lambda expressions are relevant to our example. In particular:

- A match construct is guarded if
  - the discriminant is guarded, and
  - the return type is guarded, and
  - every branch is guarded. While checking the branch, branches are treated as lambda abstractions of the parameters over the body of the branch. The parameters will be instantiated to have their respective subterm specifications, according to the subterm specification of the discriminant.
- A function application is guarded if
  - the arguments are all guarded, and
  - if the head of the application is a recursive call of the fixpoint in question, then, it must have a strict subterm in the recursive argument. Otherwise, the head must be recursively guarded.

The case for a lambda expression is just checking its subexpressions recursively:

- A lambda abstraction is guarded if
  - the type of the binder is guarded, and
  - its body is guarded.

Now, we can already see why the guard checker decides that the fixpoint above is guarded: the only recursive call of add in line 4 is called on a strict subterm of `m`, the recursive parameter.

3.2 The `stack` and β-ι commutative cuts

The **stack** of subterm specifications (or *stack* in short) corresponding to the arguments of a β-redex, is implemented by Pierre Boutillier in 2010 [4]. It is motivated by the following situation: when the head of an application is a match, the subterm specifications in the stack are used when checking the branches.

The stack contains the subterm specifications of arguments applied to a head. For example, the *stack* of the following term

```
f a b c d e
```

has length 5 and contains the subterm specifications of `a,b,c,d,e` respectively, with the subterm specification of `a` on top of the stack.

Allowing the subterm information on the stack to enter the branch in the match simulates what is known as a β-ι cut[4]. This is especially useful when the branches contain lambda abstractions, for example in the following definition of `map2` over lists:

```
1  Fixpoint map2 {A B C : Set} (f:A->B->C) (l1 : list A) (l2 : list B) {struct l1} : list C :=
2    match l1 with
3    | nil => nil
4    | cons h1 t1 => (match l2 with
5      | nil => fun _ => nil
6      | cons h2 t2 => fun t1' => cons (f h1 h2) (map2 f t1' t2)
7      end) t1
8    end.
```

The second branch of the outer match in lines 4-7 is of interest: the body of the branch is an application between an inner match and the term `l1`. Here, there stack has length 1 and contains the subterm information of `t1`, and can enter the said match. That means, the subterm information of `t1` is given to the lambda binder `t1'` in the inner match, eventually reaching the recursive call of `map2` in line 5. Since `map2` has `t1'`, which has the subterm specification of `t1`, it is treated as a strict subterm of `l1`, therefore this instance of recursive call is guarded. Since this is the only recursive call in this fixpoint definition, the fixpoint definition is guarded.

With the addition of the stack, the guard checking for `add` is now updated to the following, changes highlighted:

- A match construct is guarded if
  - the discriminant is guarded, and
  - the return type is guarded, and
  - every branch is guarded. While checking the branch, branches are treated as lambda abstractions of the parameters over the body of the branch. The parameters will be instantiated to have their respective subterm specifications, according to the subterm specification of the discriminant.
- A function application is guarded if
  - the arguments are all guarded, and
  - if the head of the application is a recursive call of the fixpoint in question, then ~~it must have a strict subterm in the recursive argument~~ the stack element corresponding to the recursive parameter must be a strict subterm. Otherwise, the head must be recursively guarded.

The case for a lambda expression is just checking its subexpressions recursively:

- A lambda abstraction is guarded if
  - the type of the binder is guarded, and
  - its body is guarded, assuming that the binder now has the subterm specification of the top of the stack.

3.3 Example: `add`

Now we have the necessary concepts ready to understand the guard checking of the `add` function, namely the guard environment, subterm specification, and the stack.

Here is a breakdown of the steps taken by the guard checker while checking `add`:

[4] β-ι commutative cut is not part of Coq's reduction since it breaks typing. See Appendix D for a counterexample.

Target for checking	Data

`Fixpoint add (m n : nat):`
`match m with`
`| O => n`
`| S m' => add m' (S n)`
`end.`
	`call stack : []`
	`loc env : []`
	`guard env : []`
	`stack : []`

Initial state. All parameters after the recursive argument are part of the fixpoint body. The recursive argument is a large subterm of itself.	

`fun (n : nat) =>`
`match m with`
`| O => n`
`| S m' => add m' (S n)`
`end.`
	`call stack : [tLambda]`
	`loc env : [m, add]`
	`guard env : [Large]`
	`stack : []`

The fixpoint body is a lambda abstraction, which is guarded if its binder's type (`nat`) and body are guarded. The return type contains no recursive call: therefore `nat` is guarded. The next step is to push the binder, `n` and its subterm specification into the guard environment. Here, `n` is given the subterm specification of `Bound{1}`, which means it is bound by a lambda and potentially needs reduction to determine its subterm specification. This information is unimportant in this example and we leave the explanation to Section 3.4. Once the binder is in the guard environment, we now check the body, a match construct.	

`match m with`
`| O => n`
`| S m' => add m' (S n)`
`end.`
	`call stack : [tCase, tLambda]`
	`loc env : [n, m, add]`
	`guard env : [Bound{1}, Large]`
	`stack : []`

A match construct is guarded if its discriminant, return type, and every branch is guarded, which we now check in order.	

`m (* discriminant *)`	`call stack : [tRel, tCase, tLambda]`
	`loc env : [n, m, add]`
	`guard env : [Bound{1}, Large]`
	`stack : []`

The discriminant contains no recursive call: `m` is guarded. The return type `nat` also contains no recursive call. The branches will be checked as lambda abstractions, bounded by the parameters in the corresponding branches, and supplied with a stack populated with relevant subterm information due to pattern matching.	

`n (* 0-th branch *)`	`call stack : [tRel, tCase, tLambda]`
	`loc env : [n, m, add]`
	`guard env : [Bound{1}, Large]`
	`stack : []`

The 0-th branch has no parameter. Since it contains no recursive calls, the 0-th branch is bounded.	

`fun m':nat => add m' (S n)`
`(* 1-st branch *)`
	`call stack : [tLambda, tCase, tLambda]`
	`loc env : [n, m, add]`
	`guard env : [Bound{1}, Large]`
	`stack : [Strict]`

The 1-st branch (corresponding to `S`) has one parameter and is thus a lambda abstraction, so we first check its binder's type, then its body. The type of the binder, `nat`, is guarded. Before checking the body, we push the bound variable `m'` into the guard environment with the subterm specification from the the top of the stack, which is already populated with the specification of a strict subterm, since `m'` is obtained by pattern matching on `m`.	

`add m' (S n)`	`call stack : [tApp, tLambda, tCase, tLambda]`
	`loc env : [m', n, m, add]`
	`guard env : [Strict, Bound{1}, Large]`
	`stack : []`

For a function application to be guarded, all arguments and the head must be guarded. If the head is the recursive function being checked (which is the case here), its recursive argument must be a strict subterm, determined from the stack, which will be populated when checking each argument	

This concludes the guard checking of `add`, which is guarded.

The guard environment is used crucially in specifying the term `m'`, but the stack was not critically used. Appendix A gives a similar breakdown on the example of `map2` on vectors, in which the stack is critically used. An example of a non-terminating fixpoint of `add_typo` is also given in Appendix B.

3.4 Strongly Guarded Fixpoints

This subsection explains the mechanism that maintains strong normalisation by guard-checking in erasable subterms, compared to weak normalisation before the fix. Hugo Herbelin made the change in 2022 [3] to accept only strongly guarded fixpoints, since it corresponds better with the intuition of users, as well as ensures termination in extracted code [18, 19].

It is possible to define fixpoints that are guarded under only certain reduction strategies, but not all. We say that this fixpoint is only **weakly guarded** instead of **strongly guarded**, corresponding to weak and strong normalisation of terms, respectively.

For example, the fixpoint defined below

```
Fixpoint erasable_zeta (x : bool) : bool :=
  let _ := erasable_zeta x in
  true.
```

is the constant `true` function if one does not eagerly evaluate the body of the let-in binding: therefore the subterm in the body is called **erasable**. However, when evaluated eagerly on any input (for example using `cbv`), this fixpoint does not terminate, because the let-in binding is evaluated first, which contains a non-terminating recursive call. If it is evaluated using the `lazy` strategy, for example, this fixpoint terminates on any boolean input since the binding was never used in the body.

The change that only allows strongly guarded fixpoints is as follows: previously, there are two places where weak-head normalisation is used: when checking for the subterm specification, and when checking for guardedness. After the change, only the former remains, and the latter is removed. Instead, the need for reduction is manually recorded in a **redex stack** whenever a redex is encountered. The size of the redex stack measures the "depth" of the current term being checked in terms of the number of redexes it is wrapped in. A term can now have a new subterm specification of an **internally bound subterm** if it is bound within a redex, because reducing the redex away could possibly generate useful subterm information for it.

For example, in our running example of `add`, which is seen internally as below:

```
Fixpoint add (m : nat) {struct m} :=
  fun n : nat =>
    match m with
    | O => n
    | S m' => add m' (S n)
    end.
```

Occurrences of the variable `n` in the body thus have the subterm specification of *internally bound subterm*. An internally bound subterm carries a set of natural numbers, which represents the redexes it is wrapped in. This set is a singleton if the term in question is a binder, since every binder can only be bound once. The generalisation of a set of redexes is due to combining subterm specifications from the branches of a match construct, whose details we leave to Section 4.1.

[5]Not to be confused with the OCaml lazy feature used in the implementation of the guard checker in the Coq kernel; here it purely refers to the deferred specification of closure terms on the stack.

3.5 Restoring Compatibility with Propositional Extensionality

In 2013, the guard checker (with the then newly implemented β-ι cuts) was found to be incompatible with the Propositional Extensionality axiom [17]. Shortly after, the compatibility is restored by the fix of Maxime Dénès in 2014 [5].

Here we take the example from Herbelin's slides from TYPES 2024 [2], which demonstrated how one could prove `False` before the fix by simply assuming the Propositional Extensionality axiom.

```
1  Inductive True2 : Prop := C2 : (False -> True2) -> True2.
2  Axiom prop_ext: forall {P Q}, (P <-> Q) -> P = Q.
3
4  Theorem T2T : True2 = True.
5  Proof. exact (prop_ext (conj (fun _ => I) (fun _ => C2 (False_rect True2)))). Qed.
6
7  Theorem T2F_FT2F : (True2 -> False) = ((False -> True2) -> False).
8  Proof. rewrite T2T; apply prop_ext; split; auto. Qed.
9
10 Fail Fixpoint loop (x : True2) : False :=
11   match x with
12   | C2 f => match T2F_FT2F in _=T return T with
13       eq_refl => fun g => loop g
14     end f
15   end.
```

The problem here lies in dependent pattern matching on equality: the inner match in lines 12-14 is applied to `f`, `f` being a subterm of `x` is propagated into the dependent match on `T2F_FT2F`, an equality, causing the guard checker to determine it as guarded. The fix is to restrict all subterm information on the stack, for example that of `f`, from entering the branches of the match expression, **unless** the return type T fullfills the following conditions:

- T has the form

  $$T = \text{fun } (x_1 : A_1)(x_2 : A_2)... \Rightarrow B$$

  where B does not depend on any parameter x_i, AND
- B has the form

  $$B = \text{forall } (y_1 : T_1)(y_2 : T_2)..., C$$

  AND
- every T_i is a (possibly qualified) inductive type, meaning having the shape

  $$T_i = \text{forall } (z_1 : U_1)(z_2 : U_2)..., I\ t_1\ t_2...$$

  where I is an inductive type.

and we forbid the propagation of subterm information from the stack by setting every entry to *Not Subterm* otherwise. Intuitively, the x_i represent the parameters of the inductive type, the y_j represent the indices of the inductive type, which

In the example above, the return type of the inner match in lines 12-14 has the return type of

```
fun (T : Prop) (fun x0 : eq Prop (True2 → False) T) => T
```

Since the body of the return type, `T` depends on the first argument, it violates the first condition above. Therefore the restriction applies: when the match is applied to `f` in line 14, its subterm information on the stack can no longer reach the branch to give `g` the subterm specification of "strict subterm".

This restriction on the β-ι cuts has, unfortunately, made one of the motivating examples, namely `map2` on vectors of equal lengths fail guard checking. Appendix A contains the breakdown of an example of this restriction in action.

3.6 Representing Inductive Types with Regular Trees

Determining what is a subterm is straightforward for terms of simple inductive types (not mutual/nested, such as `nat`), since only the constructors with recursive parameters (such as `S`) can result in strict subterms. However, inductive types could also be mutually defined (eg. even-odd predicates) or nested other inductive types (eg. rose trees). In these cases, what constitutes a subterm can no longer be inferred by simply looking only at the constructors of the inductive type in the definition, since the strict subterms could be implicit. For example, consider the mutual inductive type of even-odd and the nested inductive type of rose trees:

```
Inductive even : nat -> Type :=
| even0 : even 0
| evenS (n : nat) : odd n -> even (S n)
with odd : nat -> Type :=
| oddS (n : nat) : even n -> odd (S n).

Inductive rtree := rnode : list rtree -> rtree.
```

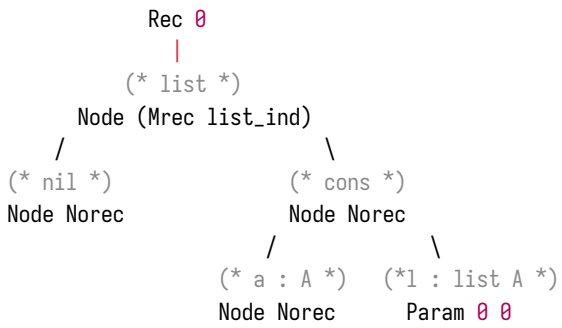
In the example of even-odd predicates, just by looking at the constructors of `even`, there does not seem to be recursive parameters to `even` at all, since the only parameters in `evenS` have types `nat` and `odd`. However, by the mutual definition of `odd`, the recursive parameter of type `even` is hidden in the constructor of `oddS`.

For the nested inductive type of `rtree`, in the only constructor `rnode`, there seems to be no recursive parameters of type `rtree` at first glance. However, every element in the list is also a strict subterm, but only transitively. As the nesting and mutuality of inductive types gets more complicated, **regular trees** provide a natural way to represent these inductive types, as well as inferring subterm specifications. In this section, we only show the regular tree for `rtree`, but the construction is similar for the other inductive types.

Before we see how the regular tree of rose trees are constructed, we first look at the nested inductive type within rose trees, namely `list`. The type of list has the following definition:

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

And the corresponding regular tree:

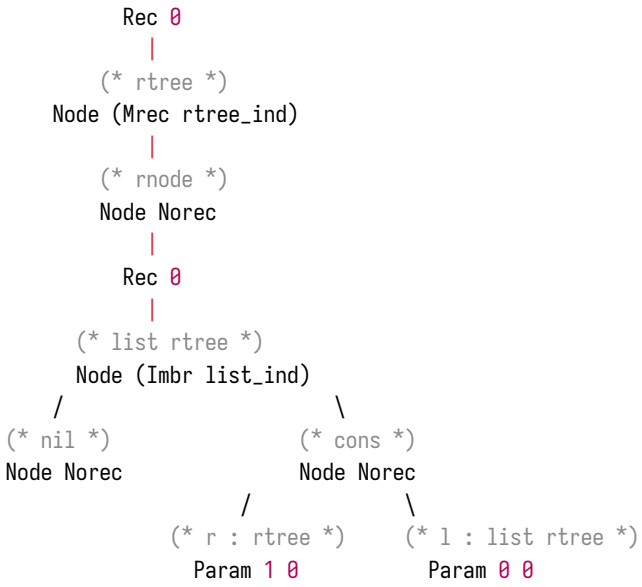


The construction of regular trees from inductive definitions is as follows:

- The root is a binder `Rec 0` for recursive references (from constructors). Every inductive type in a mutually defined collection of inductive types will have a separate tree, with the root nodes named `Rec 0`, `Rec 1`, ... and so on. Since `list` is not mutually defined, there is only one tree.
- The child of `Rec 0` represents the inductive type itself. In turn, it will have as many children as the number of constructors, each representing one constructor.
- If a constructor has no parameter, it has no children and thus a leaf node; otherwise, it will have as many children as its parameters, and all of them are leaf nodes. For example, the node representing `nil` has no children, while the node representing `cons` has two children, representing the head and tail of the list, respectively.
- What separates a regular tree and typical trees is the following: recursive parameters, such as the second parameter of `cons`, has an edge to the node binding that inductive type. In this case, the edge is represented by a `Param 0 0` node, a reference to the root of the tree, i.e. a recursion. The numbers in `Param 0 0` will be explained in the rose tree example below.

Nodes that have an edge to one of their ancestors are the recursive parameters in a constructor; to the guard checker, all such nodes represent strict subterms that can be used in recursion.[6]

Now we are ready to see the regular tree of rose trees:



At first glance, the regular tree looks different from its definition, which has only one constructor, and the constructor only has one parameter of `list rtree`, but its regular tree is not linear. This is because the node representing `list rtree`, at the 5-th level from the root, is *substituted* by the regular tree of `list`, which we have seen before, but now instantiated to refer to `rtree` whenever the type parameter `A` is mentioned. There are two recurrences at the leaves: `Param 1 0` refers to the root, while `Param 0 0` refers to the ancestor two levels up. A rough analogy can be drawn between these two leaf nodes and the de-Bruijn representation of

$$\lambda.\lambda.(1, 0).$$

The second `0` in `Param _ 0` can be omitted as they refer to *mutually* defined inductive types. In that case `n` mutual inductive types will result in `n` trees with inter-tree references.

As an example that illustrates the power of regular trees in determining subterms, we can look at the eliminator for rose trees:

[6]Not all regular trees represent valid inductive types: inductive types have to be *strictly positive* to avoid contradictions. Interested readers can refer to the Coq Reference Manual for more details.

```
1  Definition rtree_rec {A} (P : rtree A -> Set) (Q : list (rtree A) -> Set)
2    (Qnil : Q nil)
3    (Qcons : forall (t : rtree A) (l : list (rtree A)), P t -> Q l -> Q (cons t l))
4    (PNode: forall (children : list (rtree A)), Q children -> P (Node children)) :=
5    let fix prec (r : rtree A) : P r :=
6      let fix qrec (l : list (rtree A)) : Q l :=
7        match l with
8        | nil => Qnil
9        | cons r l => Qcons r l (prec r) (qrec l)
10       end
11     in match r with
12     | Node l => PNode l (qrec l)
13     end
14   in prec.
```

In the body, the fixpoint returned is `prec` (line 14). The only recursive occurrence of `prec` is in line 9, and via the regular tree structure, the guard checker correctly determines the recursive call `prec r` is calling on a strict subterm, which one can easily deduce from the regular tree structure, but not obvious in the definition of rose trees.

3.7 Extrusion of Uniform Parameters

This subsection introduces the quality-of-life improvement that allows the automatic extrusion of uniform parameters by Hugo Herbelin [20], which is particularly useful when writing fixpoints over nested inductive types.

Before this fix, the below definition of `map` over rose trees fails:

```
1  Inductive tree (A : Type) := Node : A -> list (tree A) -> tree A.
2
3  Fixpoint lmap {A B} (f : A -> B) (l : list A) : list B :=
4  match l with
5  | nil => nil
6  | cons x l => cons (f x) (lmap f l)
7  end.
8
9  (* Used to fail with:
10   Recursive definition of map is ill-formed.
11   In environment
12   map : forall A B : Type, (A -> B) -> tree A -> tree B
13   A : Type
14   B : Type
15   f : A -> B
16   t : tree A
17   x : A
18   l : list (tree A)
19   Recursive call to map has not enough arguments.
20   Recursive definition is:
21   "fun (A B : Type) (f : A -> B) (t : tree A) =>
22    match t with
23    | Node _ x l => Node B (f x) (lmap (map A B f) l)
24    end".
25 *)
26 Fixpoint map {A B} (f : A -> B) (t : tree A) {struct t} : tree B :=
27 match t with
28 | Node _ x l => Node _ (f x) (lmap (map f) l)
29 end.
```

The parameter `f` in `lmap` is a **uniform parameter**; that means it stays the same at every point of the recursion. The same can be said for the parameter `f` in map. The extrusion of such uniform parameters is needed for the following reasons.

When the guard checker checks the definition of `map`, the only recursive call occurs at line 28, where it is given only the argument `f`. Since `map` has its recursive parameter in the second and only receives one argument, there are insufficient arguments to deduce the subterm information. Therefore, to determine guardedness of the definition, the guard checker reduces the term by unfolding the definition of `lmap`, hoping to obtain the recursive argument. The expression in which the recursive call occurs thus becomes the following:

```
(lmap (map f) l) ⟶ match l with ... | cons x l ⇒ cons ((map f) x) (lmap (map f) l) end
```

Two recursive calls of `map` are resulted in the unfolding of definition. The first recursive call, `((map f) x)` has two arguments. Since the recursive parameter is the second, it is sufficient to deduce guardedness for this recursive call. However, the second recursive call, namely in `(lmap (map f) l)` only has one argument, and we face the same problem again. Further expansion will not help in this situation exactly because of the uniformity of the first parameter of `lmap`, unfolding `lmap` will always result an occurrence of `map f` that never obtains the recursive argument, in particular in the `cons` case.

In contrast, if the uniform variable, `f`, is "extracted" outside the fixpoint in `lmap`, which is possible due to uniformity, and `lmap` is equivalently defined as below instead:

```
1  Definition lmap {A B} (f : A -> B) :=
2    fix aux (l : list A) : list B :=
3      match l with
4      | nil => nil
5      | cons x l => cons (f x) (aux l)
6      end.
```

Then, after unfolding the definition of `lmap`, the following expression will form:

```
(lmap (map f) l) ⟶ match l with ... | cons x l ⇒ cons ((map f) x) (aux l) end
```

Now, the recursive call without a recursive argument no longer occurs.

After the implementation of automatic extrusion of uniform parameters, the guard checker will do an extra check to determine which parameters are uniform, then automatically exclude them in guardedness checks, achieving the same effect as our alternative definition above. This allows easier definition of nested fixpoints, such as in our case, `lmap` nested in `map`.

Chapter 4

Implementation of the Guard Checker

The full implementation described in this chapter can be found on https://github.com/inria-cambium/m1-tan/tree/v1.0.0.

We identify at least 4 dimensions of complexity of the guard checker and here explain why they are needed:

1. Ensuring strong normalisation contributes to new data structures (such as a redex stack and a new subterm specification), as well as changes in the control flow of guard checking (redexes are manually reduced in `check_rec_call_state`).
2. The regular tree representation of inductive types (`wf_paths`) contributes a new auxiliary data structure that allows the determination of subterms in mutual and nested inductive types.
3. The stack of subterm specifications (`stack_element`) contributes another new data structure that has to be maintained, to allow β-ι commutative cuts of subterm specifications.
4. In the OCaml implementation, the extensive use of OCaml's lazy feature causes some verbosity in implementation for increased efficiency.

The final point is irrelevant in our Coq implementation since one can set the evaluation strategy to `lazy` in Coq to achieve the same. The complexity of the first 3 dimensions will be explained in the rest of this chapter.

4.1 High Level Description

The intuitive idea of the guard checker is simple: recurse until a recursive call is found (in the form of a de Bruijn index), and specify its recursive argument. For this, auxiliary data structures, such as the guard environment, the stack, and the redex stack have to be maintained when a subexpression is checked.

- Guard environment: When checking the body of a binder (let-in, lambda, nested fixes), the guard environment needs to be updated with subterm specifications, possibly from the stack.
- Stack: When checking an application term, update the stack with subterm specification of the arguments before checking the guardedness of the head.
- Redex Stack: incremented before checking the branches of a match, and the body of a nested fixpoint.

Now we define the terms.

Recursive Parameter. A fixpoint must be defined with a **recursive parameter**, which is a binder to an inductive type.

Regular Tree. Every inductive type has an associated **regular tree**, which one can construct by looking at its constructors. Regular trees help determine the subterm specification of a term or binder in the fixpoint body.

Subterm Specification. A subterm specification is always relative to the fixpoint's recursive parameter. There are five kinds of subterm specification, which form a total order, written as $\leq$, defined below:

$$\text{Dead Code} \leq \text{Internally Bound Subterm} \leq \text{Strict Subterm} \leq \text{Large Subterm} \leq \text{Not Subterm}$$

This partial order is useful because of pattern matching: the subterm specification of a match construct is the maximum of the branches – more precisely, the least upper bound of the subterm specification of its branches.[7]

Dead code represents terms obtained by eliminating terms of empty inductive types (such as `False`), which can be an arbitrarily strict subterm of any term, therefore it is minimal. When a variable is bound in a binder, its subterm specification might need further reduction to determine, and therefore has a "temporary" subterm specification of Internally Bound Subterm. Pattern matching on a large subterm (i.e. the recursive parameter) results in strict subterms, so do pattern matching on strict subterms.

Stack. The stack of subterm specifications is pushed after checking arguments in an application term, or given automatically in branches of a match expression. Its elements are popped into the guard environment when lambda binders are added to the context.

Redex Stack. A redex stack can contain either a `NeedReduce` or `NoNeedReduce`. A `NoNeedReduce` is added when checking the bodies of a nested fix and the branches of a match expression. When exiting the checking of these expressions, the top of the redex stack is popped and the guard checker determines if a reduction is needed. When checking a recursive call, if reduction is needed (i.e. not enough arguments for the recursive argument), `NoNeedReduce` on top of the stack will be set as `NeedReduce`. More details are provided in Section 4.3.

4.2 Algorithm: Determining the Subterm Specification of a Term

The subterm specification of a term can be inferred from information in two sources. The first is the subterm specification of already-seen terms in the context. The second is the subterm specification of the arguments applied to it: for example, $(\lambda x.x)t$ has the subterm information of t. This is the motivation for the two auxiliary data structures for subterm information: the **guard environment** and the **stack** of subterm specifications of arguments (or stack in short). The guard environment assigns entries in the local context a subterm specification, and the stack stores subterm specifications for arguments applied to a term.

To obtain the subterm specification of a term, it is first reduced to weak-head normal form, then pattern matched in the head. An interesting case is the subterm specification of a match construct: it is the least upper bound of the subterm information of its branches, with respect to the $\leq$ partial order defined above. The reason is that a match construct can return either of its branches, and the least upper bound is a natural approximation.

Furthermore, the subterm specification of a match expression is restricted for compatibility with extensional axioms. That is if the return type of the match

- is of the form `fun (x1:A1) (x2:A2) ... ⇒ B` where `B` does not depend on any `x`is, AND
- `B = forall (y1:T1) (y2:T2) ..., C` AND
- every `Ti` is a (possibly qualified) inductive type, meaning having the shape `Ti = forall (z1:C1) ..., IND t1 t2 t3`

then the subterm specification remains as-is, otherwise is deemed as Not Subterm.

[7] In the actual implementation, this order is not only implicit but also reversed, and the subterm specification of a match construct is obtained by getting its greatest lower bound (`subterm_spec_glb`) instead. However, the notion described here and implemented are isomorphic.

```
subterm_specif guard_env t stack
whd_all t → hd, args.
match hd with
 | tRel p:
 |  | guard_env[p]
 | tLambda x ty body:
 |  | args must be empty because weak head reduction
 |  | subterm_specif body (x:NotSubterm :: guard_env)
 | tCase discr ty branches:
 |  | glb (subterm_specif branches) → res
 |  | if ty = forall … . IND … then res else Not Subterm
 | tFix mutual_fixes fix_index:
 |  | mutual_fixes[fix_index] → current_fix
 |  | push all mutual_fixes into guard_env as NotSubterm
 |  |  | except for current_fix: Strict Subterm
 |  | push all parameters of current_fix up till recursive parameter to guard_env as Not Subterm
 |  | push all args onto stack as closure
 |  | subterm_specif current_fix.body
 | tProj p t:
 |  | subterm_specif t stack → t_spec
 |  | if t_spec = Large or Strict Subterm then Strict Subterm
 |  |  | else t_spec
 | tEvar: Dead Code (smallest possible subterm)
 | tConst, tApp, tLetIn x ty c b:
 |  | impossible
 | else:
 |  | Not Subterm
```

4.3 Algorithm: Checking guardedness of a fixpoint

At the initialisation of a guard environment, only the recursive argument will be given a subterm specification. We use the term "context" or "local context" to refer to the typical context in dependent types, and "guard environment" to refer to the context of subterm specifications. For any fixpoint `f` defined as

```
Fixpoint f p0 ... pk pk+1 ... pn {struct pk} : ret_type_f := f_body
with     g q0 ... ql ql+1 ... qm {struct ql} : ret_type_g := g_body.
```

will be treated as

```
Fixpoint f p0 ... pk {struct pk} : ret_type_f' := fun pk+1 => ... fun pn => f_body.
Fixpoint g q0 ... ql {struct ql} : ret_type_g' := fun ql+1 => ... fun qm => g_body.
```

where `ret_type_f'` is `ret_type_f` generalised over the types of `pk+1` to `pn`, similarly for `ret_type_g'`.

The initial guard environment while checking `f` will be

```
local context: [pk   , ..., p0, g, f]
guard env    : [Large]
```

and similarly for `g`:

```
local context: [ql   , ..., q0, g, f]
guard env    : [Large]
```

The general idea is to check subexpressions recursively, paying special attention in key cases like match expressions, recursive calls, and nested fixpoints. In implementation, there are a few variants to these recursive checks. The most common way of checking an expression is the `check_rec_call_stack` function, which checks a term using the stack. However, there are situations where the stack should not be used. The first is when checking the guardedness of subexpressions such as the return type of a match, the stack should not be used, and there is no reduction needed since it is "inert" and does not affect the subterm specification of other subexpressions. Such instances are called `check_inert_subterm_rec_call`. The second is when checking a term and the stack should not be used, but the term is not inert and might prompt a need for reduce. Then, the `check_rec_call` function is used exactly so. Finally, to trigger a reduction on the current term, `check_rec_call_state` is used for contracting a fix, reducing a match if its discriminant is a constructor of the same type, or substituting a let-in binder to its body.

These four functions above and a fifth, `check_nested_fix_body` (which does as the name suggests), forms five mutually defined fixpoints spanning hundreds of lines. We attempt to give an idea of it in the pseudocode description below, in the hope that it will be useful to future readers interested in working on the guard checker.

```
check_rec_call_stack guard_env redex_stack stack t:
 tCase discriminant return_type branches:
  expand branches (make them into lambda form)
  check_rec_call discriminant rs → needred_discriminant, rs
  check_inert_subterm_rec_call return_type rs → rs
  specify_branches → branches_specif (set_iota_specif disc_spec default:IB(|rs|))
  filter_stack_domain stack → stack
  rs := NoNeed:rs
  for each branch 0..n
   check_rec_call_stack branches[i] branches_specif[i] rs → rs
  needred := needred_discriminant or rs[top] check_rec_call_state needred rs.tail (
   whd_all discriminant → hd, args
   hd := if tCoFix then whd_all (contract_cofix hd) else hd
   if hd = tConstruct then apply_branch hd args else whd_handle_rest)
 tFix mutual_fixes fix_inductive:
  rs := NoNeed::rs
  for each fix in mutual_fixes
   check_inert_subterm_rec_call fix.return_type rs → rs
  push all mutual fixes into the context, as NotSubterm
  drop_uniform_parameters bodies → bodies
  filter_fix_stack_domain |rs| stack num_unif_params → fix_stack
  firstn (decr_arg + 1) fix_stack → fix_stack
  fix_stack → stack_this (gets the whole stack)
  firstn num_unif_params fix_stack → stack_others (mutual branches get uniform params)
  for b in bodies:
   check_nested_fix_body (if correct fix then stack_this else stack_others) b → rs
  check_rec_call_state rs[0] rs.tail (skipn num_unif_params stack) (
   if stack[recarg] = SArg then
    raise NoReductionPossible
   else whd_all recarg → recarg
   recarg → (hd, args)
   if hd = tConstruct then
    contract_fix fix hd args
   else whd_handle_rest)
 tCofix mutual_cofixes cofix_coind:
  for every mutual cofix, check_inert_subterm_rec_call return_type rs → rs
  push all mutual cofixes into guard_env as NotSubterm
  for every mutual cofix,
   check_rec_call rs body → needreduce, rs
   check_rec_call_state needreduce rs (raise NoRedPossible) → rs
 tLambda x ty body:
  check_rec_call ty
  if stack is not empty then
   pop the stack, push binder into env with the specs from the stack, check_rec_call body
  else push binder into env as internally bound subterm { |rs| }, or NotSubterm if rs is empty
  check_rec_call_stack body
 tProd x ty body: Same as tLambda, but stack is always irrelevant.
  check_rec_call ty
  push binder into env as internally bound subterm { |rs| }, or NotSubterm if rs is empty
 tLetIn x ty bound_term body
  check_rec_call rs bound_term → needred_c, rs
  t[illegible] (i.e. we are
    making a recursive call then
```

```
check_inert_subterm_rec_call guard_env redex_stack t:
check_rec_call guard_env stack=[] redex_stack t → (rs_top, rs)
check_rec_call_state guard_env stack=[] rs_top rs t (fun _ ⇒ NoReductionPossible)
```

```
check_rec_call guard_env redex_stack t:
NoNeed::redex_stack → redex_stack
check_rec_call_stack guard_env stack=[] redex_stack t → rs_top::rs
(rs_top, rs)
```

```
check_rec_call_state guard_env stack needreduce redex_stack t expand_head:
if needreduce or redex_stack needs to reduce then
  expand_head tt → t', stack'
  if NoReductionPossible then
   set redex_stack.top as NeedReduce
  else if other error then raise
  else check_rec_call_stack (stack' ++ stack) redex_stack t'
```

Finally, `check_rec_call` is directly called by `check_one_fix`, which is in turn called by the entry point function `check_fix` on every mutual definition given a block of mutual fixpoints.

4.4 The Implementation in Coq

Our implementation of the guard checker in Coq is a faithful reimplementation of the guard checker in Coq's kernel[8]. It maintains parity with the OCaml implementation as closely as possible, extending the previous attempt by Lennard Gäher [22] in 2021. Currently, our guard checker is in full agreement on all examples in the test suite of Coq for the guard checker, other than a few which did not terminate within the time limit. However, an experimental extraction of our guard checker to OCaml code terminated on these said examples.

The implementation is organised as follows. The entry point to the guard checker as a TemplateCoq plugin is found in `plugin.v`, and demonstrated in `example.v`. The heavy-lifting is done in `guardchecker.v`, which contains all the logic written in this report. The implementation handles exceptions via a trace monad defined in `Trace.v`, and instantiated in `Inductives.v`, a file that contains helper functions related to inductive types and MetaCoq terms. In parallel, `positivitychecker.v` contains the positivity checker for inductive types, and `MCRTree.v` implements the regular tree library, both inherited from the previous project.

The `guardchecker.v` file is of particular importance. Its contents can be viewed as the following collection of sections:

1. Definition of subterm specification and utility functions.
2. Definition of guard environment and utility functions.
3. Definition of redex stack and utility functions.
4. Functions related to Propositional Extensionality compatibility.
5. Utility functions to build regular trees for inductive types.
6. Functions to determine subterm specifications (as described in Section 4.2).
7. Functions to determine guardedness of fixpoints (as described in Section 4.3).

An outline of `guardchecker.v` including concrete function names and signatures is provided in Appendix C.

[8]Reference commit hash: d6550d16f01d39dee97f7e645e415de51725fd2e

Chapter 5

Future and Related Work

5.1 Integrating into MetaCoq

MetaCoq provides the perfect background for formalising the guard checker. Formalising and showing the correctness of a guard checker is twofold: the guard condition has to be sound (implies termination), and the guard checker respects the said guard condition. Implementing in MetaCoq can benefit from the already-defined typing predicate and its verified typing function, as well as the reduction relation and its verified reduction function [23]. Together with the guard condition and its guard checker, the three of them are interdependent and thus provide a perfect setup to verify all of them at once.

Having implemented the guard checker in this project, the next step is to synthesize a guard condition that the guard checker checks for. Once this guard condition is formulated in MetaCoq, its soundness can be verified (i.e. fulfilling guard condition implies termination). Then, the implementation of the guard checker can be checked against this condition. However, a direct verification of the current implementation of guard checker will still be far from reach, due to its sheer complexity. A possible way forward is by relative consistency proofs: first reducing the guard checker to a simpler yet stronger one (i.e. a new guard checker that accepts all points that the current guard checker accepts), then, verify that the simpler guard checker satisfies the guard condition. In that case, the correctness of the simpler guard checker will imply the correctness of the original guard checker. Similarly, for the soundness of the guard condition, a relative consistency proof can be done.

With the recent progress in verified extraction of Coq to OCaml [11], the Coq implementation of the guard checker could already be run as an OCaml program. In the future, if the impediment from the efficiency of the extracted OCaml program is sufficiently small, there might be a possibility to have a verified, correct guard checker in the kernel.

5.2 Well-Founded Recursion

Other than defining fixpoints that are structurally recursive, which Coq's guard checker automatically checks for, some terminating fixpoint might not be structurally recursive. An example is the greatest common divisor function between two natural numbers:

```
Require Import PeanoNat. (* for mod *)
Fail Fixpoint gcd (x y : nat) {struct x} :=
  match x with
  | 0 => y
  | _ => gcd (y mod x) x
  end.
```

This definition is not guarded even after expanding the definition of `mod`.[9] However, this function is clearly terminating since the first argument `x` is decreasing (with respect to $<$), and that natural numbers have a minimal element, 0. Such cases where recursion can be defined using a decreasing measure is known as **well-founded recursion**.

[9] Refer to https://coq.inria.fr/doc/V8.19.2/stdlib/Coq.Init.Nat.html#gcd for an equivalent, structural recursive definition of `gcd` which the guard checker accepts.

In general, for any type `A` one can define a binary relation `R : A -> A -> Prop` for it and prove that all its elements fulfill the accessibility predicate:

```
Inductive Acc (x: A) : Prop :=
  Acc_intro : (forall y:A, R y x -> Acc y) -> Acc x.
```

then `R` is said to be a well-founded relation. In other words, a term `x` of type `A` is accessible if every term related to it by `R` (i.e. all `y` of type `A` such that `R y x` in prefix notation) is also accessible. One can do well-founded recursion on any well-founded relation. For any inductive type I, structural recursion is a particular case of well-founded relation by defining the binary relation R as exactly the subterm relation: `R y x` if and only if `y` is a subterm of `x`. Therefore, well-founded recursion is more general than structural recursion.

In our `gcd` example, we can let the relation R be exactly the $<$ relation in natural numbers. In Coq, one can define fixpoints with well-founded recursion instead of the default structural recursion so using the `Equations` package [24], or its predecessor, the `Program` fixpoint function [25]. In both situations, fixpoints can be defined with a custom well-founded relation, and proof obligations for well-foundedness will be generated for the user. In the case of `gcd`, the remaining proof obligation can be proved by simply using the `lia` tactic:

```
From Equations Require Import Equations.
Require Import Lia. (* Linear Integer Arithmetic: for the lia tactic *)

Equations gcd (x y : nat) : nat by wf y:=
gcd 0 y := y ;
gcd x 0 := x ;
gcd x y := gcd y (x mod y).
Next Obligation.
Proof. lia. Defined.
```

Interested readers can refer to https://coq.inria.fr/platform-docs/ for a tutorial on using the Equations plugin.

5.3 Agda and Sized Types

Coq uses what is known as the syntactic condition for its guard condition, avoiding reduction as much as possible and inferring information from just the syntax. For example, termination checking in Agda is done *semantically* via Sized Types: a special, implicit type is used by the type checker to determine if the recursion is structural, i.e. the recursion is done on a strictly smaller argument [26], [27]. In principle, syntactic checks are faster because they use a minimal amount of reduction and evaluation (hence the name), but are less accurate in comparison (rejected fixpoints that, in reality, terminate). On the other hand, since semantic checks allow the kernel to get closer to the actual "meaning" of programs by carrying out type checking and reduction, they can provide more accurate termination checking (i.e. reject fewer terminating fixpoints). However, semantic criteria risk being less efficient due to the extra reductions and computations compared to syntactic checks. For example, an attempt in 2019 by Chan, Li, and Bowman [28] to add Sized Typing to Coq without explicit annotations slowed down the compilation of the standard library significantly, costing 5.5× the original compile time for `Coq.Msets.MSetList`, and 15× in `Coq.setoid_ring.Field_theory`. New progress on Agda's termination checking by Nisht and Abel [29] has resulted in an algorithm that has linear time which was proven to be sound but not yet proven complete. Well-founded recursion is included in Agda's standard library in `Induction.WellFounded`.

5.4 Lean and Native Eliminators

Contrary to Coq and Agda which includes recursive functions primitively, recursive definitions in Lean only exist in the surface syntax, and is internally represented by eliminators [30]. More precisely, in Lean, when users define recursive functions, they write definitions in equational form:

```
def add (n m : Nat) :=
  match n with
  | 0 ⇒ m
  | n + 1 ⇒ (add n m) + 1
```

while in Lean's kernel, it is directly translated into a recursor representation, thus no explicit recursion remains:

```
#print add
/-
def add : Nat → Nat → Nat :=
fun n m ⇒
  Nat.brecOn (motive := fun n ⇒ Nat → Nat) n
    (fun n f m ⇒
      (match (motive := (n : Nat) → Nat.below (motive := fun n ⇒ Nat → Nat) n → Nat) n with
        | 0 ⇒ fun x ⇒ m
        | n.succ ⇒ fun x ⇒ x.1 m + 1)
        f)
    m
-/
```

`add` is structural recursive thus translated into an encoding by a so-called "`brecOn`" recursor [31], more precisely known as the course-of-values recursor [32], which is in turn defined by the eliminator (`Nat.rec`) of the inductive type.

```
#print Nat.brecOn
/-
@[reducible] protected def Nat.brecOn.{u} : {motive : Nat → Sort u} →
  (t : Nat) → ((t : Nat) → Nat.below t → motive t) → motive t :=
fun {motive} t F_1 ⇒ (Nat.rec ⟨F_1 Nat.zero PUnit.unit, PUnit.unit⟩ (fun n n_ih ⇒ ⟨F_1 n.succ
n_ih, n_ih⟩) t).1
-/
#print Nat.rec
/-
recursor Nat.rec.{u} : {motive : Nat → Sort u} →
  motive Nat.zero → ((n : Nat) → motive n → motive n.succ) → (t : Nat) → motive t
-/
```

Since Lean only has eliminators in its kernel, the kernel does not need termination checking, resulting in a simpler theory. Lean also supports well-founded recursion using keywords such as `termination_by <term>` and `decreasing_by <proof>` [33].

However, if a recursive function in Lean is defined in equational form, its compiled or generated code (equivalent to extracted code for Coq) in C will have the same equational form as in the syntax, it is just that the kernel uses a eliminator-encoding and performs type-checking on it [30]. This allows the compile code to be more idiomatic, however, due to the mismatch in representation, verifying the compilation to C will be a challenge for Lean.

Bibliography

[1] C. Paulin-Mohring, *Définitions Inductives en Théorie des Types. (Inductive Definitions in Type Theory)*. 1996. [Online]. Available: https://tel.archives-ouvertes.fr/tel-00431817

[2] H. Herbelin, "Size-preserving dependent elimination." [Online]. Available: https://pauillac.inria.fr/~herbelin/talks/talk-types24-2.pdf

[3] H. Herbelin, "Check guardedness of fixpoints also in erasable subterms." [Online]. Available: https://github.com/coq/coq/pull/15434

[4] P. Boutillier, "A relaxation of Coq's guard condition," in *JFLA - Journées Francophones des langages applicatifs - 2012*, Carnac, France, Feb. 2012, pp. 1–14. [Online]. Available: https://hal.science/hal-00651780

[5] M. Dénès, "Tentative fix for the commutative cut subterm rule." [Online]. Available: https://github.com/coq/coq/commit/9b272a861bc3263c69b699cd2ac40ab2606543fa

[6] M. Sozeau *et al.*, "The MetaCoq Project," *J. Autom. Reason.*, vol. 64, no. 5, pp. 947–999, 2020, doi: 10.1007/S10817-019-09540-0.

[7] C. Paulin-Mohring, "Extraction de programmes dans le Calcul des Constructions," 1989. [Online]. Available: https://theses.hal.science/tel-00431825

[8] P. Letouzey, "A New Extraction for Coq.," in *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, H. Geuvers and F. Wiedijk, Eds., in Lecture Notes in Computer Science, vol. 2646. Springer, 2002, pp. 200–219. doi: 10.1007/3-540-39185-1_12.

[9] M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, and T. Winterhalter, "Coq Coq correct! verification of type checking and erasure for Coq, in Coq," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 1–28, 2020, doi: 10.1145/3371076.

[10] M. Sozeau, Y. Forster, M. Lennon-Bertrand, J. B. Nielsen, N. Tabareau, and T. Winterhalter, "Correct and Complete Type Checking and Certified Erasure for Coq, in Coq," Apr. 2023. [Online]. Available: https://inria.hal.science/hal-04077552

[11] Y. Forster, M. Sozeau, and N. Tabareau, "Verified Extraction from Coq to OCaml," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Jun. 2024, doi: 10.1145/3656379.

[12] F. Pfenning and C. Paulin-Mohring, "Inductively Defined Types in the Calculus of Constructions," in *Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29 - April 1, 1989, Proceedings*, M. G. Main, A. Melton, M. W. Mislove, and D. A. Schmidt, Eds., in Lecture Notes in Computer Science, vol. 442. Springer, 1989, pp. 209–228. doi: 10.1007/BFB0040259.

[13] T. Coquand, "Pattern matching with dependent types," in *Informal proceedings of Logical Frameworks*, 1992, pp. 66–79.

[14] C. Cornes *et al.*, "The Coq Proof Assistant-Reference Manual," *INRIA Rocquencourt and ENS Lyon, version*, vol. 5, 1996.

[15] E. Giménez, "Codifying Guarded Definitions with Recursive Schemes," in *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers*, P. Dybjer, B. Nordström, and J. M. Smith, Eds., in Lecture Notes in Computer Science, vol. 996. Springer, 1994, pp. 39–59. doi: 10.1007/3-540-60579-7_3.

[16] E. Giménez, "Un Calcul de Constructions Infinies et son application a la vérification de systemes communicants," 1996.

[17] D. Schepler, "[Coq-Club] bijective function implies equal types is provably inconsistent with

[18]

[19]

[20] doi: 10.1007/978-3-030-79876-5_37.

Appendix

Appendix A: Using the stack: `map2`

Target for checking	Data

`Fixpoint map2 {A B C} (f:A->B->C) (n : nat)`
`  (v1 : vec A n) (v2 : vec B n)`
`  {struct v1} : vec C n :=`
`  match v1 with`
`  | nil => (fun _ => nil C)`
`  | cons h1 k t1 => (fun v2' : vec B (S k) =>`
`  match v2' with`
`  | cons h2 m t2 => (fun t1' =>`
`  cons C (f h1 h2) m (map2 f m t1' t2))`
`  end t1)`
`  end v2.`
	call stack : []
	loc env : []
	guard env : []
	stack : []
	redex stack: []

`fun (v2 : vec B n) =>`
`  match v1 with`
`  | nil => (fun _ => nil C)`
`  | cons h1 k t1 => (fun v2' : vec B (S k) =>`
`  match v2' with`
`  | cons h2 m t2 => (fun t1' =>`
`  cons C (f h1 h2) m (map2 f m t1' t2))`
`  end t1)`
`  end v2.`
	call stack : [tLambda]
	loc env : [v1|n|f|C|B|A|map2]
	guard env : [Large]
	stack : []
	redex stack: [NoNeed]

`match v1 with`
`| nil => (fun _ => nil C)`
`| cons h1 k t1 => (fun v2' : vec B (S k) =>`
`  match v2' with`
`  | cons h2 m t2 => (fun t1' =>`
`  cons C (f h1 h2) m (map2 f m t1' t2))`
`  end t1)`
`end v2.`
	call stack : [tApp|tLambda]
	loc env : [v2|v1|n|f|C|B|A|map2]
	guard env : [Bound{1}|Large]
	stack : []
	redex stack: [NoNeed]

v2	call stack : [tRel|tApp|tLambda] loc env : [v2|v1|n|f|C|B|A|map2] guard env : [Bound{1}|Large] stack : [] redex stack: [NoNeed]

`match v1 with`
`| nil => (fun _ => nil C)`
`| cons h1 k t1 =>`
`  (fun v2' : vec B (S k) =>`
`  match v2' with`
`  | cons h2 m t2 => (fun t1' =>`
`  cons C (f h1 h2) m (map2 f m t1' t2))`
`  end t1)`
`end`
	call stack : [tCase|tApp|tLambda]
	loc env : [v2|v1|n|f|C|B|A|map2]
	guard env : [Bound{1}|Large]
	stack : [Closure v2]
	redex stack: [NoNeed]

`fun n0 : nat ⇒`
`  fun v10 : t A n0 ⇒`
`  ∀ x : vec b n0, vec C n`
	call stack : [tLambda|tCase|tApp|tLambda]
	loc env : [v2|v1|n|f|C|B|A|map2]
	guard env : [Bound{1}|Large]
	stack : [Closure v2]
	redex stack: [NoNeed]

`(fun x : vec B 0 => nil C)` `(*0-th branch*)`	call stack : [tLambda|tCase|tApp|tLambda] loc env : [v2|v1|n|f|C|B|A|map2] guard env : [Bound{1}|Large] stack : [IB{1}] redex stack: [NoNeed|NoNeed]
`nil C`	call stack : [tApp|tLambda|tCase|tApp|tLambda] loc env : [x|v2|v1|n|f|C|B|A|map2] guard env : [IB{1}|Bound{2}|Large] stack : [] redex stack: [NoNeed|NoNeed]
`fun (h1:A) (k:nat) (t1:vec A k) =>` `  (fun v2' : vec B (S k) =>`	call stack : [tLambda|tCase|tApp|tLambda] loc env : [v2|v1|n|f|C|B|A|map2]

Appendix B: Negative Example: `add_typo`

`Fixpoint add_typo (m n : nat) :=`
`match m with`
`| O ⇒ n`
`| S unused ⇒ add_typo m (S n)`
`end.`
	call stack : []
	stack : []
	redex stack : []
	guard env : []
	loc env : []

`fun (n : nat) ⇒`
`match m with`
`| O ⇒ n`
`| S unused ⇒ add m (S n)`
`end.`
	call stack : [tLambda]
	stack : []
	redex stack : [NoNeed]
	guard env : [Large]
	loc env : [m, add]

`match m with`
`| O ⇒ n`
`| S unused ⇒ add m (S n)`
`end.`
	call stack : [tCase, tLambda]
	stack : []
	redex stack : [NoNeed]
	guard env : [Bound{1}, Large]
	loc env : [n, m, add]

`m (* discriminant *)`	call stack : [tRel, tCase, tLambda] stack : [] redex stack : [NoNeed, NoNeed] guard env : [Bound{1}, Large] loc env : [n, m, add]
`n (* 0-th branch *)`	call stack : [tRel, tCase, tLambda] stack : [] redex stack : [NoNeed, NoNeed] guard env : [Bound{1}, Large] loc env : [n, m, add]
`fun unused : nat ⇒ add m (S n)` `(* 1-st branch *)`	call stack : [tLambda, tCase, tLambda] stack : [] redex stack : [NoNeed, NoNeed] guard env : [Bound{1}, Large] loc env : [n, m, add]
`add m (S n)`	call stack : [tApp, tCase, tLambda] stack : [] redex stack : [NoNeed, NoNeed] guard env : [Strict, Bound{1}, Large] loc env : [unused, n, m, add]
`S n`	call stack : [tApp, tApp, tCase, tLambda] stack : [] redex stack : [NoNeed, NoNeed, NoNeed] guard env : [Strict, Bound{1}, Large] loc env : [unused, n, m, add]
`m`	call stack : [tRel, tApp, tCase, tLambda] stack : [] 29 redex stack : [NoNeed, NoNeed, NoNeed] guard env : [Strict, Bound{1}, Large]

Appendix C: Outline of `guardchecker.v`

```
From MetaCoq.Utils Require Import utils MCMSets.
From MetaCoq.Common Require Import BasicAst Universes Environment Reflect.
From MetaCoq.Template Require Import Ast AstUtils LiftSubst Pretty Checker.

From MetaCoq.Guarded Require Import MCRTree Inductives.

Section SubtermDef.
  Inductive size := Large | Strict.
  Definition size_eqb (s1 s2 : size) : bool.
  Definition size_glb s1 s2 : size.
  Module Natset := MSetAVL.Make Nat.
  Inductive subterm_spec :=
    | Subterm (l : Natset.t) (s : size) (r : wf_paths)
    | Dead_code
    | Not_subterm
    | Internally_bound_subterm (l : Natset.t).
  Definition subterm_spec_eqb (s1 s2 : subterm_spec) : bool.
  Definition merge_internal_subterms (l1 l2 : Natset) : Natset.
  Definition spec_of_tree (t : wf_paths) : exc subterm_spec.
  Definition inter_spec (s1 s2 : subterm_spec) : exc subterm_spec.
  Definition subterm_spec_glb (sl : list subterm_spec) : exc subterm_spec.
End SubtermDef.

Section GuardEnvDef.
  Record guard_env := { loc_env : context; rel_min_fix : nat; guarded_env : list
subterm_spec; }.
  Implicit Type (G : guard_env).
  Definition init_guard_env Γ (recarg : nat) (tree : wf_paths) : guard_env.
  Definition push_var G '(na, type, spec) : guard_env.
  Definition push_let G '(na, c, type, spec) : guard_env.
  Definition push_var_nonrec G '(na, type) : guard_env.
  Definition update_guard_spec G (i : nat) new_spec : guard_env.
  Definition push_var_guard_env G (n : nat) na ty : guard_env.
  Definition lookup_subterm G p : subterm_spec.
  Definition push_context_guard_env G Γ : guard_env.
  Definition push_fix_guard_env G (mfix : mfixpoint term) : guard_env:=
End GuardEnvDef.

Section RSDef.
  Inductive fix_check_result := (* or redex_stack_element *)
    | NeedReduce (Γ : context) (e : fix_guard_error)
    | NoNeedReduce.
  Definition set_need_reduce_one Γ nr err rs : list fix_check_result.
  Definition set_need_reduce Γ l err rs : list fix_check_result.
  Definition set_need_reduce_top Γ err rs : list fix_check_result.
End RSDef.

Section StackDef.
  Inductive stack_element :=
    | SClosure (r : fix_check_result) G (nbinders : nat) (t : term)
    | SArg (s : subterm_spec).
  Definition fix_check_result_or (x y : fix_check_result) : fix_check_result.
  Notation "x ||| y" := (fix_check_result_or x y).
  Fixpoint needreduce_of_stack (stack : list stack_element) : fix_check_result.
  Definition redex_level := List.length.
  Definition push_stack_closure G needreduce t stack : list stack_element.
  Definition push_stack_closures G l stack : list stack_element.
  Definition push_stack_args l stack : list stack_element.
  Definition lift_stack_element (k : nat) (elt : stack_element) : stack_element.
  Definition lift_stack (k : nat) : list stack_element.
End StackDef.

Section PropExt.
  Definition restrict_spec_for_match Σ ρ Γ spec (rtf : term) : exc subterm_spec.
  Definition filter_stack_domain Σ ρ Γ nr (rtf : term) (stack : list stack_element)
    : exc (list stack_element).
  Definition filter_fix_stack_domain (nr decrarg : nat) stack nuniformparams
    : list stack_element.
End PropExt.

Section RegularTree.
  Definition match_recarg_inductive (ind : inductive) (r : recarg) : bool.
```

Appendix D: β-ι commutative cut breaks typing

A β-ι cut (applying a β-redex over an ι redex) has the following shape:

```
(match d with                          →        match d with
| ... => (fun a b c => p)                       | ... => p [a/aa, b/bb, c/cc]
| ... => (fun a b c => q)                       | ... => q [a/aa, b/bb, c/cc]
end) aa bb cc                                  end
```

where there ι-redex is in the head. The ι-redex could also be in the argument:

```
f (match d with ... => p | ... => q end)     →     match d with ... => f p | ... => f q end
```

It is worth noting that β-ι cuts do not preserve subject reduction, and thus is not a reduction done in Coq. For example, in the following definition of `map2` on vectors,

```
Require Import Vector.

Fixpoint map2 {A B C : Type} (f : A -> B -> C) (n : nat) (v1 : t A n) (v2 : t B n) : t C n :=
  match v1 with
  | nil _ => fun (_ : t B 0) => nil C
  | cons _ h1 n' t1 => fun v2' : t B (S n') =>
      match v2' with
      | cons _ h2 _ t2 => fun R => cons _ (f h1 h2) _ (R t2)
      end (map2 f _ t1)
  end v2.
```

If we focus on the outermost match in the fixpoint body:

```
  match v1 with
  | nil _           => fun _   : t B 0      => nil C
  | cons _ h1 n' t1 => fun v2' : t B (S n') => ...
  end v2.
```

This term is well-typed. However, the β-ι cut is not, because `v2` cannot have type `t B 0` and `t B (S n')` at the same time; in other words, the two types `t B 0` and `t B (S n')` are non-convertible.

Appendix E: Terminology Reference

There is no uniform terminology for concepts in this paper so far. For reference, these are the possible alternatives:

Terminology	**Possible Alternatives**
Guard Environment (`guard_env`, `G`)	Recursive Environment (`renv`)
Subterm Specification (`subterm_spec`, `spec`)	Size, Smallerness, Subterm Information
Redex Stack (`rs`)	`fix_check_result` stack
Regular Trees	Well-founded paths (`wf_paths`), recarg tree
Recursive Position (`recpos`)	Recursive Argument (`recarg` or `rarg`, abuse of notation), Decreasing Argument