Nested Inductive Types

Thomas Lamiaux, Yannick Forster, Matthieu Sozeau, Nicolas Tabareau

Nested Inductive Types

Justified and Usable Nested Inductive Types in Lean and Rocq

THOMAS LAMIAUX, Nantes Université, Inria, France
YANNICK FORSTER, Inria, France
MATTHIEU SOZEAU, Inria, France
NICOLAS TABAREAU, Inria, France

Inductive types are a fundamental abstraction mechanism in type theory and proof assistants, supporting the definition of data structures and rich specifications. *Nested* inductive types extend this mechanism by allowing constructors to use parametric types instantiated with the type being defined (*e.g.,* lists or trees of the type to be defined). They are widely used in large verification projects – including CompCert, Iris, Verinum, and MetaRocq – to express complex, structured specifications. Despite this widespread use, the treatment of nested inductive types in both Lean and Rocq is unsatisfactory. Lean rejects many practical definitions while Rocq accepts definitions for which no usable elimination principle can be defined. Neither system provides reliable automatic generation of elimination principles. As a result, developers must define custom eliminators by hand, leading to fragility, duplication, and significant proof engineering overhead. This paper introduces a novel validity criterion for nested inductive types that guarantees that they can be elaborated into well-formed mutual inductive types. Under this criterion, the elimination principle for the original nested definition is provably equivalent to that of its elaborated mutual form. Our condition strictly generalizes Lean's current check while ruling out exactly the problematic cases accepted in Rocq. Using this foundation, we give a systematic method for automatically generating correct elimination principles for nested inductive types, and we provide an implementation integrated into Rocq, along with an implementation plan for Lean.

1 Introduction

In type theory, inductive types are a fundamental construct playing a role similar to data structures in programming language. They enable the representation of canonical notions of mathematics and computer science such as natural numbers, syntax, and other tree-like structures. However, inductive types go beyond classical data structures by offering a natural and expressive framework for specifying and reasoning about programs and their properties within a rigorous logical foundation, such as typing relations, logical relations, or intrinsically typed syntax.

Inductive Types. To ensure logical consistency in the presence of (recursive) inductive types, not all inductive types can be accepted. A so-called *strict positivity criterion* is required, prohibiting definitions where the inductive type appears in a non-strictly positive position – for example, to avoid the definition of an indudctive type equivalent to its own negation. Concretely, strict positivity requires that in the type of a constructor for an inductive type `I`, all occurrences of `I` appear only on the right-hand side of function arrows in each of the constructor arguments' types. This criterion naturally generalizes to mutually inductive definitions as well, and both Lean and Rocq enforce the same condition.[1]

The word inductive comes from the fact that *all* the terms of an inductive type are combinations of its constructors. This fundamental principle gives rise to elimination principles. Giving an inductive type satisfying the strict positivity criterion, it is possible to define an elimination principle. For the case of `list`, the elimination principle has type:

[1]Rocq and Lean appear to enforce the same condition for mutual but non-nested inductive types, although this is not explicitly documented in Lean's manual, making it difficult to confirm with certainty.

Authors' Contact Information: Thomas Lamiaux, Nantes Université, Inria, France; Yannick Forster, Inria, France; Matthieu Sozeau, Inria, France; Nicolas Tabareau, Inria, France.

```
list_elim : ∀(A : Type) (P : list A → Type),
       P nil → (∀(a : A) (l' : list A), P l' → P (cons a l')) → ∀l : list A, P l.
```

Coarsely, this elimination principle, for a given predicate `P: list A → Type`, is generated by asking the user to provide a proof that the predicate holds for each constructor—here, `P nil` and `P (cons a l')`—assuming that it holds for all the recursive occurrences that have been deemed positive—here `P l'` for the `list A` argument of cons. Elimination principles are also often called induction principles (with a more logical connotation) or recursion principles (with a more computational connotation)—we simply say elimination principle in this paper and mean all variants by this.

Depending on the foundational system and proof assistant, elimination principles are either added as primitives at definition time, or can be defined in terms of other primitives of the proof assistant. In Lean, elimination principles are added as new axioms with their associated computation rules, while in Rocq, they are automatically generated as a combination of fixpoints and pattern matching that are primitives in Rocq. This difference does really not matter for the rest of the paper as our work applies to both design choices, even if it has slightly different technical implications.

Nested Inductive Types. To capture more sophisticated forms of recursion and structure, *nested* inductive types extend the expressive power of inductive types, analogous to nested datatypes in programming languages [11]. Nested inductive types allow constructor arguments to have a recursive reference to the defined types below other type constructors—most commonly below `list` or other parameterized types. This added flexibility allows, for instance, the concise representation of arbitrary n-ary operations or richly structured syntax trees. Such constructs occurs for instance in functional languages such as OCaml, in formalisations of (first-order) logic, n-ary application in λ-calculus, or type theory.

Probably the most well-known example of nested inductive type is the type of trees with arbitrary many subtrees, a.k.a. Rose Trees[2], but such nesting with lists occurs also naturally *e.g.*, when formalising syntax. Let us look at a typical definition of syntax for a λ-calculus with a `switch` case-analysis construct.

```
Inductive term := lam : term → term | ... | switch : term → list term → term.
```

It uses nesting by instantiating the parameter of the inductive type `list` with the type `term` to be defined in order to encode the different branches of a pattern matching, the first argument (of type `term`) being the discriminee. Here, `list` would be called a container type. Note that in this paper, we do not cover so-called truly nested inductive types, which use themselves as container [26].

Using types like `list` and `option` allows the reuse of facilities for the construction of complex data structures and the integration with existing libraries. As a result, nested inductive types are particularly well-suited for defining and reasoning about complex structures or specifications that are compositionally built from simpler container types. This includes not only common containers but also more advanced constructions such as maps [24].

An important example is the typing judgment for our λ-calculus: since terms are nested, the typing judgment will be as well. For a switch to be well-typed, the discriminee has to have finite type, and all branches the return type[3]. This is implemented by nesting with the unary parametricity translation of `list`, called `All : (A → Prop) → list A → Prop`, ensuring that a predicate `P` holds for all elements of a list.

```
Inductive typing : term → ty → Prop := ...
| typing_switch (discr : term) (brs : list term) (n : nat) (return_type : ty) :
          typing discr (finite n) → All (fun a ⇒ typing a return_type) brs →
```

[2]https://en.wikipedia.org/wiki/Rose_tree
[3]A complete definition would have to deal with a context for variables, but this is not necessary to illustrate our point.

```
        typing (switch discr brs) return_type.
```

Nested inductive types are crucially used in large-scale formalization efforts in Rocq: They are used in one of the fundamental type definitions of CompCert; in program verification frameworks such as the Verified Software Toolchain (using CompCert's C) and the Iris project, in the verification of numerical algorithms in Verinum, and are omnipresent in formalisations of type theory e.g. for Martin-Löf type theory in Agda [2] or Rocq [3], and the *MetaRocq* project of Sozeau et al. [28], which amongst other things specifies the type theory of Rocq and implements a verified type checker for it in Rocq. In Lean, lack for more wider support for nested inductive types is wished for by users in software verification, *e.g.,* to be able to nest over maps [18].

Broken Support in Rocq and Lean. Considering the importance of nested inductive types, one would expect Rocq and Lean to offer good support for them, yet neither one does. They both fail to generate usable elimination principles, and surprisingly accept a different set of nested inductive types, Rocq being too permissive, whereas Lean is too restrictive.

Broken Support in Rocq. Rocq is not able to generate a useful elimination principle for *any* nested inductive type. This lacking feature frequently confuses users and results in many questions in online forums, while in large-scale verification projects it leads to manual elimination principle definitions taking hundreds of lines, which have to be maintained by hand.

For the type of terms, it generates an elimination principle which does not contain any induction hypothesis for the branches of the pattern-matching. For containers such as `list` in a simple type fragment, it is folklore the elimination principle will make use of the (unary) parametricity translation for containers, as described in the Coq'Art book in 2004 [9]. For the case of terms, one would want the elimination principle for the constructor `switch` to be:

```
 ∀(discr : term) (brs : list term), P discr → All P brs → P (switch discr brs))
```

While Rocq does not generate this elimination principle, it can be manually proved by users using fixpoints and pattern-matching.

This has been worked out formally in a line of work by Johann et al. [19–21]. Seminal work by Tassi [30] attempted to lift this intuition to dependent type theory. Key to the validity of generating induction principles via parametricity is that one can generate proofs of the fundamental lemma of the parametricity translation. As observed by Tassi, this is in general impossible: For some inductive types, the fundamental lemma only holds externally, and cannot by proved in all generality inside the type theory. Consequently, this approach falls short for some nesting containers, see Section 2.4.

Another issue of Rocq's implementation is that it proves to be excessively permissive. It accepts nested inductive types that lack a mutual encoding and are challenging to analyze or reason about within the meta-theory. Moreover, for types that cannot be encoded as mutual inductives, it seems to be impossible to define the expected elimination principle even manually, making them unusable in practice, in which case rejecting the nested definition is arguably a better behavior.

Broken Support in Lean. Switching to Lean does not resolve the situation either. Lean's approach is to compute an equivalent mutual inductive encoding of a given nested inductive type and then generate directly in the kernel an elimination principle that mirrors the one of this mutual encoding. For the type of terms, it generates a principle

```
term.rec : ∀(P : term → Sort u), ∀(P' : list term → Sort u),
  (∀discr brs, P discr → P' brs → P (switch discr brs)) →
  P' nil → (∀t l, P t → P' l → P' (x :: l)) → ∀t, P t
```

However, in this approach, the mutual encoding leaks into the elimination principle, as the user must explicitly provide an additional predicate `P'` on `list term`. In practice, `P'` can be instantiated with

`All P`, but this workaround highlights the need for a more systematic treatment of the problem—one that avoids such leakage of the internal encoding. More problematic, since a straightforward mutual encoding becomes impractical in more complex scenarios, Lean ultimately rejects some nested definitions that are theoretically valid—and often crucial for advanced specifications. For example, defining `typing` in version 4.24.0 produces the following error message:

```
> (kernel) invalid nested inductive datatype 'All', nested inductive datatypes parameters
cannot contain local variables.
```

Contributions. These implementation shortcomings likely stem from an incomplete foundational understanding of nested inductive types. In this paper, we argue that, just as the strict positivity condition guarantees the existence of an eliminator for mutual inductive types, a similar well-formedness condition must be established for nested inductive types to ensure their eliminators can be derived. However, the nested setting introduces additional complexity: the positivity condition itself must be employed in the very definition of the elimination principle.

In this paper, we *set the theoretical and practical foundations* to make nested inductive types *justified and usable* in both Lean and Rocq. To do so:

- We introduce a *novel positivity condition* for nested inductive types based on *strictly positive parameters*, which guarantees the existence of the elimination principles expected by users.
- In order to generate nested elimination principles, we build on a *sparse* variant of *parametricity*, which provides a lightweight yet powerful tool for ensuring correctness while preserving practical usability. Our novel positivity condition and the use of *sparse* parametricity together ensure that the fundamental lemma can always be proved, resolving the issues observed by Tassi.
- To *justify* our positivity criterion, we show that every nested inductive type can be encoded as a positive, non-nested mutual inductive type: the elimination principle for the original nested type can be systematically defined for its translation, and proven using the mutual elimination principle.

Our work lays the theoretical and practical groundwork for robust and automated support for nested inductive types, addressing the core limitations in both Lean and Rocq. In Lean, it extends the mutual encoding approach to arbitrarily complex cases while preventing the encoding from leaking into the generated eliminators. In Rocq, it enables correct and systematic eliminator generation for all valid nested definitions, ruling out ill-formed cases where no eliminator can exist. It bridges the gap between usability and meta-theory, enabling proof assistants to better accommodate large-scale formal developments while reducing the manual effort required from users.

We also contribute actual implementations and formal proofs of our results:

- The reduction from nested inductive to mutual inductive and the proof that it produces a valid mutual inductive has been mechanized in MetaRocq.[4]
- We demonstrate the feasibility and utility of our approach in practice by providing a prototype plugin for Rocq along with our formalization and a corresponding port to OCaml, part of a pull request on Rocq to replace and subsume the actual mechanism for generating elimination principles. We also outline a plan for integration in Lean.

Note that we have *not* mechanized the encoding of the elimination principle as it is currently out of reach: it would require a framework for verified meta-programming capable of building complex type derivations which currently does not exist.

[4]We use MetaRocq as the analogous Lean4Lean formalisation [15] is still work in progress. References to the formalization are made using *[Foo.v]*. All links have been anonymized, but can be retrieved from anonymous supplementary material.

Outline of the Paper. After reviewing the current support for nested inductive types in Section 2, Section 3 introduces our notion of positivity for nested inductive types and proves that every positive nested inductive type corresponds to a strictly positive mutual inductive type. Then, Section 4 presents *sparse parametricity*, which we use to derive the appropriate elimination principles for nested inductive types. After discussing related work in Section 5, we describe our implementation in Section 6 and conclude in Section 7.

2 Current Support for Nested Inductive Types

Different proof assistants based on dependent type theory—such as Rocq and Lean—have adopted divergent design choices regarding nested inductive types, each with its own set of limitations. In this section, we begin by reviewing commonly accepted practices, then examine the specific design decisions made by these systems along with their associated constraints. We conclude by analyzing the recent technique based on parametricity proposed by Johann and Polonsky [21] and Tassi [30], and highlight its advantages and shortcomings.

2.1 Folklore

Nested inductive types enable the definition of complex inductive structures—such as optional or arbitrary branching—by leveraging existing inductive types. A well-known example is the rose tree, a tree structure in which each node may have an arbitrary number of subtrees. As shown in the introduction, rose trees can be naturally encoded using a nested inductive type, typically defined as `rose_tree A`, with a constructor that recursively takes a list of rose trees as input:

```
Inductive rose_tree A : Type :=
| leaf : A → rose_tree A
| node : list (rose_tree A) → rose_tree A.
```

Encoding Nested Inductive Types as Mutual Inductive Types. The folklore foundational justification for nested inductive types is that they can be encoded as mutual inductive types. Consequently, they are not expected to introduce any additional logical power. Their purpose is purely practical: to simplify formalization and improve modularity. The core idea behind this correspondence is to replace each occurrence of the nesting container—*i.e.,* the inductive type used for nesting—with a specialized copy of its definition, instantiated to operate solely on the nested inductive type.

For the case of rose trees, `rose_tree_mut` is mutually defined with `list_rose_tree_mut` as follows

```
Inductive rose_tree_mut A : Type :=
  | leaf_mut (a : A) : rose_tree_mut A
  | node_mut (l : list_rose_tree_mut A) : rose_tree_mut A
with list_rose_tree_mut A : Type :=
  | nil_mut : list_rose_tree_mut A
  | cons_mut : rose_tree_mut A → list_rose_tree_mut A → list_rose_tree_mut A.
```

The need for a dedicated version of `list` in the mutual definition highlights the practical value of nested inductive types: they allow shared use of existing definitions and properties of common container types. Without support for nested inductives, the entire library developed for lists would have to be duplicated for the specialized type `list_rose_tree_mut`, significantly increasing redundancy and maintenance effort.

This encoding naturally raises the question of which nested inductive types are valid through corresponding to a mutual inductive definition. In particular, it is not possible to nest over non-uniform parameters or indices, as no general translation to mutual inductive types then exists. We discuss why this is not possible in Appendix B using power lists as an example.

Elimination principles for Nested Inductive Types. For a non-nested inductive type, the elimination principle closely follows the structure of the inductive type, and generating it and inhabiting it is straightforward. This is much less trivial to derive the elimination principle for nested inductive types, in particular to define appropriate recursion hypotheses.

It is folklore that elimination principles for nested inductive types require the notion of parametricity to define the recursion hypotheses for nested arguments, as introduced by Reynolds [27] and extended to type theory by Bernardy et al. [8], Keller and Lasson [23]. Parametricity is a principle that formalizes the idea that polymorphic functions must behave uniformly across all their instantiations. More precisely, it states that a term of a polymorphic type cannot depend on the specific structure of its type arguments, but only on their abstract behavior. This leads to "free theorems," or properties that can be derived purely from the type of a function, without needing to inspect its definition. The idea is to leverage the parametricity predicate to enforce that the motives we are trying to prove hold on all the subterms. This is possible as in dependently typed settings, parametricity can be expressed via a syntactic translation that associate a term with a logical relation or predicate [8, 23].

The standard example of the literature is to define the elimination principle for `rose_tree` that is nested on `list`, using the unary parametricity `listε` of `list` to generate the recursion hypothesis [9]. In the case of unary parametricity, the predicate associated to `list` is given by the following inductive type, which requires a predicate `Aε : A → Type` on the parameter `A`, and enforces this predicate holds on all subterms of type `A`.

```
Inductive listε A (Aε : A → Type) : list A → Type :=
  | nilε : listε A Aε nil
  | consε : ∀a (aε : Aε a) l (lε : listε A Aε l), listε A Aε (cons a l).
```

Note that `listε` actually corresponds to the inductive predicate `All` presented in the Introduction. Concretely, `listε` is used in the induction hypothesis to ensure that the motive we aim to prove holds for all the subterms, in this case, for all the elements of the list. This gives us the following elimination principle:

```
Definition expected_ty_rose_tree_elim :=
  ∀(A : Type) (P : rose_tree A → Type) (Pleaf: ∀a, P (leaf a)),
  ∀(Pnode : ∀l, listε (rose_tree A) P l → P (node l)) (t:rose_tree A), P t.
```

Inhabiting the elimination principle as in Rocq or via the mutual encoding requires some form of a local version of the fundamental lemma for the parametricity to prove the recursion hypotheses hold. For `list`, it requires that any term of type `list A` gives rise to a witness that `listε A Aε l` if any term of type `A` gives rise to a witness that `Aε a`. The proof that any list is parametric is obtained by doing an induction on the list, so the elimination principle `list_elim` is crucial in the proof of the fundamental lemma for lists.

```
Definition listε^fl A (Aε : A → Type) (aε : ∀a, Aε a) (l : list A) : listε A Aε l
  := list_elim (listε A Aε) nilε (fun r ⇒ consε r (aε r)) l.
```

Generalizing this idea to automatically derive elimination principles is far from straightforward because parametricity, while closely related, is not entirely suited to the task. One of the contributions of this paper is to clarify that this connection, while strong, is not perfect. In particular, when nesting through types like vectors instead of simple lists, blindly applying parametricity results in the generation of unnecessary predicates and overly complex recursion hypotheses. These, in turn, significantly hinder the usability of the resulting elimination principles. As we will see in Section 2.4, this can even lead to elimination principles that are unusable in practice. To solve this problem, we provide a comprehensive description of a modified version of parametricity that properly scales to nested inductive definitions in Section 4.

2.2 Nested Inductive Types in Rocq

Positivity Condition in Rocq. Positivity checking for nested inductive types in Rocq proceeds similarly to the case of non-nested inductive types (see Appendix A for a comprehensive definition), with the key difference being in how nested occurrences are handled.

When the positivity checker encounters a nested occurrence such as `list (rose_tree A)`, Rocq performs the following steps:

(1) It verifies that the type used for nesting (*e.g.,* `list A`) is not a mutual inductive type, and is only nested on uniform parameters: nesting over non-uniform parameters or indices is not possible as discussed in Appendix B.
(2) It substitutes the recursive occurrence (*e.g.,* `rose_tree A`) into the types of the constructors of the nesting type. For instance, it substitutes `rose_tree` into the types of `nil` and `cons` for `list`, and then recursively checks that the resulting types are strictly positive.

While this appears perfectly reasonable it suffers from several issues. First, Rocq currently forbids nesting with mutual inductive types which is fundamentally strange if nested inductive types are justified by having a mutual encoding, as it would allow nesting with `rose_tree` but not with its encoding `rose_tree_mut`. As we will see, this limitation is artificial and likely reflects the broader uncertainty and lack of understanding surrounding nested inductive definitions.

Second, this approach ensures that nested inductive definitions remain well-founded and consistent by reducing the positivity check to a form of "dynamic inlining" of the nesting type's structure. A particularity of this positivity check is that it can accept nested inductives that could not be accepted by a purely syntactic check. For instance, consider the inductive type `depends_on_b` which takes a boolean `b:bool` and a type parameter `A:Type`.

```
Inductive depends_on_b (b : bool) (A : Type) :=
| cst : (if b then A else (A → bool)) → depends_on_b b A.
```

If the boolean `b` is `true`, then nesting on `A` is allowed as it reduces to `A` which is strictly positive, but it is not allowed if `b` is `false` as in this case it reduces to `A → bool` which is not positive.

Moreover, there are no checks to prevent nested inductive types from corresponding to inductive-inductive types. The positivity checker operates on the specialized version but only checks strict positivity, and not well-typedness which ensures inductive types are not inductive-inductive. Thus, because the specialized version is not type-checked independently, certain patterns slip through and Rocq currently accepts a small subset of nested inductive types that, in effect, correspond to inductive-inductive definitions. Consider the type `Fnat`, representing natural numbers where the `Fnat_succ` constructor takes two equal variables rather than one.

```
Inductive Fnat : Type := Fnat_zero : Fnat | Fnat_succ : ∀(n m : Fnat), n = m → Fnat.
```

In this case, the equality `n = m` expands to `eq Fnat n m`, where `eq` is itself an inductive type. The recursive dependency via `eq` introduces an inductive-inductive structure, but Rocq accepts it because the strict positivity checker does not reject such cases. And indeed the corresponding mutual encoding `Fnat_mut` of `Fnat` is strictly positive as can be seen below. Yet, the type of `Fnat_mut_eq` must have an index of type `Fnat` to be instantiated in `n` and `m`, making it inductive-inductive.

```
Fail Inductive Fnat_mut : Type :=
| Fnat_mut_zero : Fnat_mut | Fnat_mut_succ : ∀(n m : Fnat_mut), Fnat_mut_eq m n → Fnat
with Fnat_mut_eq : Fnat_mut → Fnat_mut → Type :=
| eq_refl_mut : ∀(n : Fnat_mut), Fnat_mut_eq n n.
```

All the nested inductive types accepted by Rocq that happen to correspond to inductive-inductive definitions appear to be harmless, as they fall within a very limited and well-behaved fragment of such definitions. In particular, these cases probably do not compromise consistency, although we have no proof of that. However, even though such definitions may be accepted, Rocq lacks proper support for inductive-inductive reasoning, making it difficult to do anything meaningful with them. For instance, the expected elimination principle for `Fnat`—defined using `fix` and `match`—is not accepted by the guardedness condition. As a result, no structurally recursive functions can be written over such types in practice.

Elimination Principle through Fixpoints and Pattern-Matching. When Rocq encounters a nested occurrence during the generation of an elimination principle—such as `list term`—it fails to generate any induction hypothesis for the nested components. As a result, the derived elimination principle is incomplete and effectively unusable for reasoning about such structures.

However, the correct principle can be defined manually, using the underlying primitive constructs `fix` and `match`, using the ability to nest fixpoint expressions. Appendix C provides an illustration for rose trees and `typing`.

The lack of automatic support for elimination principles of nested inductive types require the users to do boilerplate work manually. Even though the elimination principle generated by Rocq can be copied and adapted to add the missing induction hypotheses, this process is extremely tedious when the inductive types get more involved. For example, the definition `typing` of MetaRocq/PCUIC is about 100loc, the type of the elimination principle 150loc, and its proof 300loc.

2.3 Nested Inductive Types in Lean

Positivity Condition in Lean. To decide if a nested inductive type is well-formed, Lean translates it to mutually inductive types, and checks that it is well-formed as a mutual inductive type. Checking for well-formedness and not just positivity, prevents Lean from accepting wrongly inductive-inductive types as Rocq does. However, as Rocq, Lean accepts types that can only be accepted dynamically like `depends_on_b`. Moreover, as mentioned in the introduction, Lean relies on too naive mutual translation resulting in ill-typed mutual inductive definitions for more complex situations—even in cases where a more refined encoding would succeed. This forces Lean to prevent users from using previously seen arguments in nested occurrences, and hence to reject definitions such as `typing`, as it cannot handle variables. Yet, as we will demonstrate in Section 3, a more sophisticated translation can accommodate such definitions correctly.

Elimination Principle Based on the Mutual Encoding. In Lean, the elimination principle of the corresponding mutual type is used to implement the elimination principles for the nested inductive. Thus, this elimination principle involves a predicate for `rose_tree_mut` as well as a predicate for `list_rose_tree_mut`, and four conditions mixing those dealing with rose trees and those with lists:

```
rose_tree_mut_elim :
∀(A : Type) (Pr : rose_tree_mut A → Type) (Pl : list_rose_tree_mut A → Type),
       (∀a : A, Pr (leaf_mut a)) →
       (∀l : list_rose_tree_mut A, Pl l → Pr (node_mut l)) →
       Pl nil_mut →
       (∀r : rose_tree_mut A, Pr r → ∀l, Pl l → Pl (cons_mut r l)) →
       ∀r : rose_tree_mut A, Pr r.
```

The implementation of Lean tries to hide this encoding to the user by redefining a dedicated elimination principle generated from this one by refolding the original definitions of `list` and `rose_tree` and adding the corresponding reduction rules.

This approach is rather unsatisfactory, as the resulting elimination principle does not match what users typically expect. In particular, the original mutual elimination principle leaks into the user interface. This is evident from the explicit induction cases for `nil` and `cons`, which prevent the reuse of generic definitions and lemmas over standard lists. Instead, users are forced to manually inline or reimplement these definitions, undermining modularity and compositional reasoning. Moreover, the reduction rule for `cons` exposes the elimination principle for `list_mut`, which itself has been unfolded back into `list`. This leads to unnecessary complexity and verbosity. Another major drawback is the potential blowup in size: mutual encodings generate an inductive block for every distinct nested occurrence. As a result, a motive like `P0` must be added for each such block, along with corresponding assumptions like `Pnil` for each constructor. This makes the resulting elimination principles impractical for complex types such as `typing`, which already involve many nested and non-trivial occurrences. That said, these mutual induction principles are still useful: in practice, they can be employed to define the elimination principles that users expect. However, this translation is neither automatic nor trivial, as it relies on the availability of a fundamental lemma for parametricity.

2.4 Limitations of Using Full Parametricity

The main challenge with nested elimination principles lies in automatically generating and proving suitable recursion hypotheses that ensure the motive holds for all subterms.

To address this generically, Tassi [30] adopts the folklore intuition and generates an intermediate elimination principles using unary parametricity, which is then instantiated to recover the elimination principles expected by users. Independently, Johann and Polonsky [21] tackled this issue using categorical semantic for Algebraic Data Types (ADT), a non-dependent fragment of Rocq or Lean inductive types, but covering truly nested inductive types. Their elimination principles coincide with Tassi [30]'s intermediate ones. Therefore, providing a categorical justification to Tassi [30]'s approach for the Algebraic Data Types fragment. Unfortunately, as we now show, using parametricity does not scale in the presence of type dependency.

Concept. Tassi's [30] approach proceeds in two steps. First, it uses parametricity to generate an intermediate elimination principle, then it instantiates it to obtain a usable elimination principle. To explain how it works, we use rose trees as a running example.

The generation of the type of the intermediate elimination principle of an inductive block is done using parametricity via the following procedure [30, Section 5.4]:

(1) The parameters of the inductive type `A:X` are quantified, and for each of them a new hypothesis is added whose type is the parametricity witness of `A`, that is `PA:`$[\![$`X`$]\!]_\varepsilon$ `A`.
(2) A motive is added for each inductive block following the usual procedure except that a hypothesis using parametricity is added for each index.
(3) To each constructor `c:T` of each inductive block, an hypothesis whose type is the parametricity of `T` is added, except that the inductive blocks are not translated by their parametricity predicates but by their motives, which have the same type. We write it `Pc:`$[\![$`T`$]\!]_{\mathtt{P}}$ `c` for this version of parametricity.
(4) The conclusion states that the motive holds when parametricity of the inductive block holds.

Following this procedure for `rose_tree`, gives us one new hypothesis `PA :` $[\![$`Type`$]\!]_\varepsilon$ `A` as `rose_tree` only has one parameter `A : Type`. Moreover, as `rose_tree` it is not a mutual nor indexed inductive type, only one motive is added which is the usual one `P : rose_tree A →  Type`. This gives us the following intermediate elimination principle of `rose_tree A`, where the hypotheses generated using parametricity are shown after simplification in comments.

```
Definition ty_rose_tree_elim_param :=
  ∀(A : Type) (PA : ⟦Type⟧ε A) (* (PA : A → Type) *) (P : rose_tree A → Type),
  ∀(Pleaf: ⟦A → rose_tree A⟧P leaf), (* (Pleaf: ∀a, PA a → P (leaf a)) *)
  ∀(Pnode : ⟦list (rose_tree A) → rose_tree A⟧P node),
      (* (Pnode : ∀l, listε (rose_tree A) P l → P (node l)) *)
  ∀t, rose_treeε A PA t → P t
```

The type `ty_rose_tree_elim_param` corresponds to the expected elimination principle, up to two key modifications.

First, it introduces an additional predicate `PA : A → Type` along with a proof `PA a` in the type of the `Pleaf` constructor. This extra hypothesis arises from the parametricity translation: being a purely syntactic and uniform transformation, parametricity introduces a predicate for every constructor argument, regardless of its type. Because parametricity is oblivious to semantic intent—such as whether `A` should be treated as a constant type—it generates a predicate `PA` unconditionally. As a result, the elimination principle must include `PA` to provide a definition for the parametricity witness of any term `a : A`.

Second, the elimination principle requires an additional hypothesis of the form `rose_treeε PA t` in order to establish that `P t` holds. This requirement arises as a workaround for the (unnecessary) introduction of the predicate `PA`. Without this assumption, it would generally be impossible to prove `P t` by structural recursion—using `fix` and `match` on `t`—since one cannot assume that `PA a` holds for arbitrary `a : A`. Indeed, in the case where `t := leaf a`, there is no reason to expect `PA a` to hold—`PA` could just as well be defined as the constantly false predicate. Consequently, Tassi [30]'s intermediate elimination principle requires an additional hypothesis `rose_treeε PA t` to ensure that `PA a` holds in the case `t := leaf a`. This intermediate principle can then be inhabited using `fix` and `match` over the parametricity proof, relying on the functoriality of the type used for nesting to bridge the parametricity predicates and the target motives. In the case of `rose_tree`, this translation uses the functoriality of `listε`, namely the existence of a term:

```
listε_funct : ∀A P Q (H : ∀a, P a → Q a), ∀l, listε A P l → listε A Q l
```

This allows one to recursively convert the parametricity witness `listε (rose_tree A) (rose_tree ε A PA) l` into the desired goal `listε (rose_tree A) P l`.

The second step of the construction consists in automatically deriving the expected elimination principle from the intermediate one. This is achieved by instantiating the auxiliary predicates introduced by parametricity with trivial ones—typically predicates that always hold—and then automatically proving a form of local fundamental lemma, that parametricity holds for the considered nested inductive type under this instantiation. For the `rose_tree` example, this amounts to instantiating `PA` with `fun _ ⇒ True`, and automatically constructing a proof of the fundamental lemma for rose trees: `∀t, rose_treeε (fun _ ⇒ True) t`. If this step can be achieved, this provides us with a usable elimination principle very close to the one expected by users, though polluted with trivial hypotheses.

Advantages. The main advantage of the approach proposed by Tassi [30] is that it offers a generic and modular method for generating recursion hypotheses, ensuring that the motive holds on all relevant subterms. Because this approach is driven by parametricity, it automatically handles more complex nesting patterns without requiring any special treatment. Likewise, the method is robust to situations where the inductive type used for nesting is itself nested. For instance, if we define a new inductive type that nests using `rose_tree`, the recursion hypothesis will naturally involve `rose_treeε`, without requiring any additional encoding effort.

In many cases, this allows the automatic generation of both an intermediate elimination principle and a final elimination principle close to what users expect—albeit still cluttered with trivial hypotheses. Compared to the current support in ROCQ and LEAN, which either omits such principles or generates unsatisfactory ones, this represents a significant improvement in automation.

Limitations. The main limitation of Tassi's [30]'s approach is that it relies on two non-trivial ingredients: the functoriality of the inductive types used in nesting, and a local fundamental lemma for parametricity itself. Unfortunately, both are sometimes difficult to establish—even for non-nested inductive types—due to unnecessary hypotheses introduced by the parametricity translation.

The fundamental lemma for parametricity states that if a term `t` is well-typed in a context $\Gamma \vdash t : T$, then its parametricity translation $[\![t]\!]_\varepsilon$ is well-typed in the translated context $[\![\Gamma]\!]_\varepsilon \vdash [\![t]\!]_\varepsilon : [\![T]\!]_\varepsilon\ t$. This means that parametricity must also hold over every constant of the environment used, and over context, and thus must handle open terms correctly—including variable bindings introduced by type constructors.

While these are hard proofs to generate in many cases, in the general case it turns out to be *impossible*, due to parametricity only holding externally. Notably, this happens when types quantify over arbitrary type variables. As a minimal and striking example, consider the inductive type $\texttt{closed}_{id}$, which is non-recursive and has only one constructor, together with is translation:

```
Inductive closed_id := c_id : (∀X, X → X) → closed_id.
Inductive closed_idε : closed_id → Type :=
c_idε:∀(f:∀X,X → X), (∀X (PX:X → Type) (x:X), PX x → PX (f x)) → closed_idε (c_id f).
```

To prove the fundamental lemma—namely, that $\texttt{closed}_{id\varepsilon}$ `t` holds for any `t : ` $\texttt{closed}_{id}$, one most construct a value of type $\texttt{closed}_{id\varepsilon}$ $(\texttt{c}_{id}$ `f)`, and thus prove: `∀X (PX : X → Type) (x : X), PX x → PX (f x)`. This is clearly impossible without further assumptions on `f`, since we cannot derive parametricity internally for such higher-rank polymorphic functions. Although the fundamental lemma of parametricity holds externally (*i.e.,* semantically), it cannot be internalized in intensional type theory without significant extensions, as discussed in [7, 25]. This is especially unfortunate since the standard elimination principle for $\texttt{closed}_{id}$ is trivial—it merely requires a pattern match on a non-recursive type. Thus, the parametricity-based translation ends up overcomplicating a case that should be elementary and cannot be used as a default method as it would not support inductive types that are currently supported.

3 From Nested Inductive Types to Mutual Inductive Types

In this paper, we adapt the approach of Tassi [30], propagating predicates modularly through the nested structure. However, we rely on a sparse variant of parametricity that introduces only the hypotheses that are strictly necessary. To achieve this, we must precisely control which subterms of each constructor argument require recursive application of parametricity. When such recursion is unnecessary, we classify the argument as constant. This is not achievable using the current positivity conditions – which elaborate the nested encoding and only check positivity after reduction – as this allows for dynamic nesting, like `depends_on_b` (Section 2.2), making it impossible to decide positivity purely syntactically.

To overcome this issue, we define a new positivity condition to syntactically decide if a nested inductive type is positive or not, ruling out these ill-behaved nested inductive types. The core idea is to only allow nesting on the parameters that are arities—of the form $\forall\, \texttt{X}_0 \ \ldots\ \texttt{X}_n,\ \texttt{Type}$—and that occur in strictly positive position, with arguments that are themselves strictly positive. We call these parameters strictly positive parameters. This approach ensures, statically, that strict positivity is preserved during elaboration into a mutual inductive definition: substituting a strictly positive position with a strictly positive parameter preserves strict positivity. Moreover, it guarantees that hypotheses are only generated for strictly positive parameters, which enables a sparse form of parametricity. This, in turn, allows us to generate recursion hypotheses selectively and automatically derive the elimination principles users expect.

In this section, we first define the mutual encoding of nested inductive types and identify the necessary conditions for this encoding to be well-defined. We then introduce a new positivity criterion for nested inductive types, based on the notion of strictly positive parameters, and show that this condition is preserved under the mutual encoding.

Technical Preliminaries. From this point on, we assume that terms are represented by the type `term` of MetaRocq [28], representing Rocq's kernel abstract syntax trees and supporting constructs such as products, functions, n-ary applications, and inductive types. To enable reasoning about strictly positive parametricity, we further assume that each inductive type is associated with a function `check_nestable`, which determines whether its parameters are strictly positive. The role of the strict positivity condition is to guarantee the correctness of this function: that is, it defines what it means for parameters to be strictly positive. As a result, `check_nestable` can be computed directly from the requirements imposed by the strict positivity criterion. For readability, we present examples using named variables; the accompanying formalization provides full details, including the handling of De Bruijn indices, encoded using the constructor `tRel` of `term`. Moreover, we do not distinguish indices from non-uniform parameters for inductive types as they behave exactly as indices for the generation of elimination principles.

3.1 Encoding Nested Inductive Types

Intuitively, to encode nested inductive types as mutual ones one has to substitute parameters with their instantiations. However, in a dependently typed setting, this substitution is far from straightforward, as the instantiations may depend on earlier arguments in the constructor. As a result, naively performing substitution can lead to ill-scoped terms—expressions that refer to variables no longer present in the context. Care must therefore be taken when defining the mutual encoding: the transformation must preserve well-scopedness and avoid generating ill-formed inductive definitions.

Case Study. Consider the nested inductive type `typing` of the introduction. Among others, it takes an argument `return_type` which appear in the nested argument using `All`. Substituting the predicate `P` in `All` with this instance would lead to the following signature for `All_cons_mut` which would be ill-scoped as `return_type` is not in its context.

```
∀a l, typing_mut a return_type → All_mut return_type l → All_mut return_type (a::l)
```

It is not possible to prevent such arguments to appear in the instantiation as done in Lean (Section 1) without excluding `typing` which is perfectly valid and actually needed for large scale formalization like MetaRocq. Thus, we need to ensure that any arguments on which the instantiations depend remain in scope. Since the parameters must be shared across all mutual inductive blocks, we must include the relevant arguments as indices to keep them available. However, we cannot simply add all arguments as indices, because some of them may be recursive and including such recursive arguments as indices would result in an inductive-inductive definition, which we want to avoid. For instance, for `typing`, adding all the previously seen arguments to the indices of `All_mut` would give the following signature which is clearly inductive-inductive as its type refers to `typing`.

```
  All_mut : ∀(discr ind : term), term → list term → typing discr ind → Type
```

```
Inductive typing_mut : term → term → Type :=
| typing_mut_switch (discr : term) (branches : list term) (n : nat) (return_type : term)
    :
    typing_mut discr (fintype n) → All_mut return_type branches → typing_mut (switch
    discr branches) return_type
with All_mut : term → list term → Type :=
| All_mut_nil return_type : All_mut return_type []
| All_mut_cons return_type a l : typing_mut a return_type → All_mut return_type l →
    All_mut return_type (a::l).
```

Fig. 1. Encoding of typing.

Consequently, we can only retain non-recursive arguments and must enforce that recursive occurrences do not appear within the instantiation of nested arguments. This restriction also applies to strictly positive parameters: although they are allowed in principle, substituting them further could still lead to inductive-inductive definitions. In the case of typing, optimizing to retain only the necessary arguments leads to the encoding in Figure 1.

Encoding. The encoding proceeds as follows: for each nested argument, the (potentially mutual) inductive types used for nesting are first specialized so that their definitions no longer mention the nested inductive type in their signatures. These specialized inductive blocks are then incorporated into the mutual inductive definition, and the nested arguments are replaced with their corresponding specialized instantiations.

To define the specialization of an inductive block, let us first specify the context of a nesting argument appearing in a given constructor. We note $\texttt{params}_g$ the list of types of parameters of the constructor, $\texttt{args}_g$ the list of types of arguments appearing before the nested occurrence and $\texttt{largs}_g$ the list of types of arguments of the nested occurrence , and σ the instantiation used in the nesting argument, such that $\texttt{params}_g,\ \texttt{args}_g,\ \texttt{largs}_g \vdash \sigma$. Given an inductive block $\texttt{ind}_1$ of the inductive type used for nesting, let $\texttt{params}_1,\ \texttt{indices}_1$ be its parameters and indices, that is, such that $\texttt{ind}_1 : \texttt{params}_1 \to \texttt{indices}_1 \to \texttt{Type}$.

We first need to define the type of the specialization of a block. The parameters must be the same for all the inductive blocks, therefore, the parameters of the specialization of $\texttt{ind}_1$ must be $\texttt{params}_g$. We also need to define the indices. As some parameters can appear in indices, we must substitute σ in $\texttt{indices}_1$. For σ to be well-scoped, we need to add $\texttt{args}_g,\ \texttt{largs}_g$ as indices. Yet, as explained above, doing so directly could create an inductive-inductive type. Consequently, we need to first strengthen $\texttt{args}_g$ to remove recursive occurrences. This obviously includes recursive and nested arguments, but also strictly positive parameters as they could become recursive if nested upon. We write $\uparrow\texttt{args}_g$ as the strengthening of $\texttt{args}_g$. Thus, the signature of the specialized block is:

$$\texttt{ind}_1\texttt{\_mut params}_g : \uparrow\texttt{args}_g \to \texttt{largs}_g \to \sigma(\texttt{indices}_1) \to \texttt{Type}.$$

Given a constructor $\texttt{c} : \forall \texttt{x}_1 \ldots \texttt{x}_n,\ \texttt{ind params}_1\ \texttt{t}_1 \ldots \texttt{t}_n$, we need to substitute σ in the definition. First, for this to be well-scoped, we need to add $\uparrow\texttt{args}_g,\ \texttt{largs}_g$ as arguments. Second, as we have changed the parameters and added new indices, we cannot simply substitute in the definition. Recursive argument instantiated with $\texttt{params}_1\ \texttt{inst\_indices}$ must be translated now instantiated with $\texttt{params}_g\ \texttt{args largs}\ \sigma(\texttt{inst\_indices})$. We still abuse the notation and note $\sigma(\_)$ for this variant of substitution. Put altogether, this gives the following type for the specialization of the constructor.

$$\texttt{c\_mut} : \forall (\texttt{a}{:}\uparrow\texttt{args}_g)\ (\texttt{l}{:}\texttt{largs}_g)\ \sigma(\texttt{x}_1) \ldots \sigma(\texttt{x}_n),\ \texttt{ind}_1\texttt{\_mut params}_g\ \texttt{a l}\ \sigma(\texttt{t}_1) \ldots \sigma(\texttt{t}_n).$$

This procedure describes only a single-layer encoding: if either σ or $\texttt{ind}_1$ is itself nested, then the specialization will also still be nested. As a result, the process must be repeated recursively on the specialized version.

3.2 Positivity Conditions

To formally define the mutual encoding—and more importantly, to specify the sparse parametricity translation and elimination principle in Section 4—we require structural information about the nested inductive type, which is typically entangled with the strict positivity condition.

A key technical contribution of our work is to recognize that this structural information can be decoupled from the positivity condition and described independently via a notion of *view* on the nested inductive type. Strict positivity then guarantees that these views are computed correctly—for instance, that arguments marked as constant (*i.e.,* that does not mention the recursive blocks or the strictly positive parameters) are indeed constant. This separation not only clarifies the structure of nested inductive types, but also provides a principled and modular foundation for presenting the strict positivity condition.

View of a Nested Inductive Type. We define a view `argument` (see *[Positvity_condition.v]*) that decomposes the shape of arguments of constructors of nested inductive types. An argument is either:

(1) A constant `t : term` which is recorded by `arg_is_cst`.
(2) Of the form `∀ largs, X inst_args` where `largs` are constants, and `X` is either (a) a recursive occurrence of the nested inductive type, (b) a strictly positive parameter we can nest on, or (c) a nested argument.

In the latter case, the view on the terms `X` and `inst_args` is further decomposed:

(a) `X` it is an inductive block at position `pos_block` of the mutual inductive type. In that case, `inst_args` is decomposed between the instantiations of the parameters which are not recorded as they can be inferred and the instantiations of the indices `inst_indices` which are supposed to be constants. Thus, the view on arguments is recorded as `arg_is_ind largs pos_block inst_indices`.
(b) `X` is a strictly positive parameter $\texttt{A}_k$ and the argument is recorded through its position `k` as `arg_is_sup largs k inst_args`.
(c) `X` is an nesting inductive type `ind` and then `inst_args` is decomposed between the instantiations of the parameters `inst_params` and the instantiations of the indices `inst_indices` which are supposed to be constants. As the parameters can be nested on—provided they are arities of the form `∀ llargs, Type`—we further decompose each instantiation as functions `fun llargs ⇒ arg` to recover `arg : term` of the nested term. `llargs : list term` are potentially empty, and considered as constants. This gives us an instantation of parameters of type `list (list term * term)`. The argument is then recorded as `arg_is_nested largs ind inst_params inst_indices`.

Put altogether, this leads us to the inductive definition of `argument` given in Appendix D representing our view on a nested inductive type. We can define a function `arg_of_term : term → argument` that unfold and accumulates products in `largs`, until reaching the head of the product. Once the head is reached, if it is a strictly positive parameter, or an inductive or nested argument, it is decomposed as explained above, otherwise the original term is considered as a constant. Conversely, there is a function `term_of_arg : argument → term`, that keeps track of the context, refolds the products and applications, and insert parameters for recursive arguments (see Appendix E for an illustration of the view on `All_cons`).

Positivity condition for Nested Inductive Types. Our strict positivity condition extends the standard strict positivity condition for mutual inductive types by incorporating both strictly positive parameters and nested arguments. The extension is minimal: except for the handling of arguments—where two new cases are introduced—our condition aligns closely with the traditional one. The key distinctions are that:

(1) strictly positive parameters are not permitted to appear in positions that are not strictly positive
(2) recursive occurrences are not permitted to appear in nested arguments, nor in strictly positive parameters as they could be replaced by nested arguments if substituted.

A mutual inductive type is strictly positive if all of its inductive blocks are positive. An inductive block `ind params : indices → Type` is positive provided that all its constructors are positive, and that the `indices` do not refer to the inductive blocks nor the strictly positive parameters. A constructor `c : x`$_1$ `... x`$_n$`, ind params t`$_1$ `... t`$_n$ is positive provided that all its arguments are positive, and that the instantiation of the indices `t`$_1$ `... t`$_n$ does not refer to the inductive blocks nor to the strictly positive parameters.

To present the strict positivity condition for arguments clearly, we rely on the `argument` view. Since our strict positivity condition is a conservative extension of the standard one, the first two cases remain unchanged—except for the additional treatment of strictly positive parameters that should not appear in constants. To handle nested arguments, we introduce two additional cases: one for the nested arguments themselves, and another for strictly positive parameters, which are now treated as a primitive notion via the constructor `arg_is_sup`.

(1) a constant `arg_is_cst t` then `t` must not refer to the inductive blocks or strictly positive parameters
(2) a recursive argument `arg_is_ind largs pos_block inst_indices` then: `largs`, `inst_indices` must nor refer to the inductive blocks and the strictly positive parameters; `pos_block` must be strictly inferior to the total number of inductive block.
(3) a strictly positive parameter `arg_is_sup largs k inst_args`, then: `largs, inst_args` must nor refer to the inductive blocks and `k`-th parameter does correspond to a strictly uniform parameter. To ensure the encoding is not inductive-inductive, we also need that recursive occurrences do not appear in in `largs, inst_indices`.
(4) a nested argument `largs ind inst_params inst_indices`, then `largs, inst_indices` must nor refer to the inductive blocks and the strictly positive parameters. Moreover, for each instantiation `(largs, arg) : list term * term` of a parameter, `llargs` must nor refer to the inductive blocks or the strictly positive parameter; and if the parameter is strictly positive then `arg` must be positive, otherwise it must not refer to the inductive blocks or the strictly positive parameters. To ensure the encoding is not inductive-inductive, we also need that recursive occurrences do not appear in `largs`, nor in `llargs` and `args`.

The complete formal definition of `positive_argument` can be found in *[Positvity_condition.v]*.

3.3 Preservation of Positivity Through the Encoding

To prove that the encoding preserves positivity, it suffices to prove that the specialization of the nesting containers preserve positivity, as the only modifications in the original inductive blocks are switching from nested arguments to recursive arguments which is trivial.

The encoding have been fully formalized using the MetaRocq project in *[Nested_to_Mutual.v]*. The lemma stating that the mutual encoding is strictlcy positive is proven in *[Nested_to_Mutual_proof.v]* as `pos_nested_to_mutual`. While conceptually close to the informal presentation, the formalization is significantly more involved due to the use of De Bruijn indices for variables and various low-level issues, which we briefly review here.

The main overhead comes from De Bruijn indices: contexts must be explicitly tracked to verify whether parameters appear in terms due to a lack of a verified meta-programming framework, adding substantial complexity to the formalization part. This is especially cumbersome due to the many interacting contexts involved in the encoding and in the specialization of inductive blocks (*e.g.*, `params, args_g, largs_g, ...`). Another source of complexity lies in our use of (a variant of) the view `argument`'s. While essential to the formalization to separate definitions from proof, it forces us to port many definitions and properties from `term` to `argument`. Finally, Rocq supports non-uniform parameters, which behave like indices. We refer to the formalization for their precise treatment. Despite those overheads, nothing is conceptually difficult, and we believe this formalization offers an excellent benchmark for verified meta-programming frameworks.

4 Eliminators For Nested Inductive Types

The main advantage of our new positivity condition is that it precisely identifies the parameters that can be nested on—namely, the strictly positive ones. As a result, we only need to generate predicates and recursion hypotheses for these parameters. This enables the definition of a *sparse* version of parametricity, which introduces only the necessary recursion hypotheses. Strict positivity now exactly characterizes which subterms can be treated as constants, avoiding unnecessary overhead. Furthermore, sparse parametricity is always provable, provided the associated predicates hold. This is guaranteed by the fact that all uses of the predicates are confined to strictly positive positions. This property is crucial to derive recursion hypotheses for nested arguments defined via sparse parametricity *without requiring additional preconditions*, in contrast to the approach of Tassi [30].

In the following, we use the Rocq, *i.e.*, fixpoints and pattern-matching in definitions. However, our approach can be adapted to the Lean setting by relying solely on elimination principles and reduction rules. All the code presented has been formally defined within Rocq, together with the corresponding proofs.

To understand how it solves our problem, consider the type `All` that is used to nest in `typing`, introduced in Section 1 (this example is treated in more details in Appendix E). We do not want to generate predicates for `A` as it appears in the indices, and hence cannot be nested on. We only want to add a predicate for the predicate `P` that we can actually nest on. As a consequence, `A` and `List A` are constants, and there is no need to generate additional hypotheses for them. We only need to add hypotheses for `P a`, and `All P l`. This gives us the following sparse parametricity for `All` (noted with suffix $_s$), which is much simpler than the usual parametricity (noted with suffix $_\epsilon$).

```
Inductive Allₛ {A} {P : A → Type} (Pₚ : ∀a, P a → Type): ∀{l}, All P l → Type :=
| Allₛ_nil : Allₛ Pₚ All_nil
| Allₛ_cons a l : ∀(ra : P a), Pₚ a ra → ∀(x : All P l), Allₛ Pₚ x →
                  Allₛ Pₚ (All_cons a l ra x).
```

Supposing that all the added predicates hold, in this case that P_P holds, the sparse parametricity holds. Indeed, the only hypotheses added outside recursive ones are predicate in strictly positive positions, which all hold by hypotheses. For instance, we can easily prove the local fundamental lemma for `All` using that P_P was only added in strictly positive positions.

```
Fixpoint lfl_Allₛ {A} {P : A → Type} {Pₚ : ∀a, P a → Type} (HPₚ : ∀a pa, Pₚ a pa)
    : ∀l (t : All P l), Allₛ Pₚ t := fun l t ⇒ match t with
  | All_nil ⇒ Allₛ_nil
  | All_cons a l ra x ⇒ Allₛ_cons a l ra (HPₚ a ra) x (lfl_Allₛ HPₚ l x) end.
```

This local fundamental lemma is crucial to define eliminators as it enables to prove the recursion hypotheses defined using sparse parametricity for nested arguments. For instance, it is possible to prove the elimination principle of `typing` by `fix` and `match` on the argument of type `typing t T` using the local fundamental lemma to prove `Allₛ Pₚ x`. Therefore, the approach provides a general method without the intermediate eliminator needed by Tassi [30]. We now make these ideas formal.

We define sparse parametricity, nested eliminators, and show how they can be encoded with eliminators of their mutual encoding.

4.1 Sparse Parametricity for Nested Inductive Types

To define sparse parametricity as an inductive predicate on the original inductive type, we need to quantify all the parameters and indices of the original inductive type for it to be well-defined. In addition, for each strictly positive parameter `A: ∀ llargs, Type`, we add a predicate `PA : ∀ llargs , A llargs → Type`. In contrast to standard parametricity, we do not introduce hypotheses for `llargs`, as they are assumed to be constant. Similarly, we do not add hypotheses for non-strictly positive parameters or indices, as it is impossible to nest on them. This can be summarized as follows, where $\texttt{A}_1 \; \ldots \; \texttt{A}_n$ are the parameters, and $\texttt{PA}_1 \; \ldots \; \texttt{PA}_n$ are the predicates that are added for the strictly positive parameters. We underline the arguments added by sparse parametricity and present the original term alongside its parametricity translation, separated by a dashed line.

$$\texttt{ind A}_1 \; \ldots \; \texttt{A}_n \; \texttt{: indices} \rightarrow \texttt{Type}$$

```
------------------------------------------------------------------------
```

$$\texttt{ind}_s \; \texttt{A}_1 \; \underline{\texttt{PA}_1} \; \ldots \; \texttt{A}_n \; \underline{\texttt{PA}_n} \; \texttt{:} \; \forall \texttt{(ins : indices), ind A}_1 \; \ldots \; \texttt{A}_n \; \texttt{ins} \rightarrow \texttt{Type}$$

Given a constructor of an inductive block, sparse parametricity consists in adding a hypotheses or not for each argument depending on its shape using the function $[\![\texttt{x:T}]\!]_s$ defined below.

$$\texttt{| c :} \; \forall \texttt{(x}_1 \; \texttt{: T}_1\texttt{)} \; \ldots \; \texttt{(x}_n \; \texttt{: T}_n\texttt{), ind params t1} \; \ldots \; \texttt{tn}$$

```
--------------------------------------------------------------------------
```

$$\texttt{| c}_s \; \texttt{:} \; \forall \texttt{(x}_1 \; \texttt{: T}_1\texttt{)} \; \underline{\texttt{(x}_{1s} \; \texttt{:} \; [\![\texttt{x}_1 \; \texttt{: T}_1]\!]_s\texttt{)}} \; \ldots \; \texttt{(x}_n \; \texttt{: T}_n\texttt{)} \; \underline{\texttt{(x}_{ns} \; \texttt{:} \; [\![\texttt{x}_n \; \texttt{: T}_n]\!]_s\texttt{)}}\texttt{,}$$
$$\texttt{ind}_s \; \texttt{params t1} \; \ldots \; \texttt{tn (c x}_1 \; \ldots \; \texttt{x}_n\texttt{)}$$

For each argument `x : T`, we add a hypothesis or not depending on the shape of `T`. Thanks to the view `argument`, we can add hypotheses only for non-constant variables as constant variables have already been preprocessed, that is `t, largs, inst_indices, inst_args`, and hence can be ignored. More specifically, given an argument:

(1) If it is a constant, we do not need to generate an extra hypothesis.
(2) If it is a strictly positive parameter $\texttt{A}_k$, we generate a proof of the associated predicate $\texttt{PA}_k$.
(3) If it is an inductive block `block pos_block`, we generate a recursive block of the sparse parametricity $\texttt{block}_s \; \texttt{i}$.
(4) If it is a nested argument, for each instantiation of a nestable parameter, we recursively compute the recursion hypothesis. If it is `Some P`, we return `P`. If it is `None` which means it is not nested, then we return a trivial predicate `fun _ ⇒ True`.

Sparse parametricity is implemented in *[sparse_parametricity.v]*. The key advantage of it over the standard one is that it allows us to prove that the translation holds locally, provided the added predicates hold—since no unnecessary hypotheses are introduced. This gives rise to a local fundamental lemma for sparse parametricity (see *[local_fundamental_lemma.v]* for an implementation).

Theorem 4.1 (local fundamental lemma). *Given an inductive block of an inductive type* $\texttt{ind A}_1 \; \ldots \; \texttt{A}_n \; \texttt{: indices} \rightarrow \texttt{Type}$*, then assuming that the added predicates* $\texttt{PA}_k$ *hold with witness* $\texttt{HPA}_k$*, then the sparse parametricity holds for each inductive block. Namely, we have a term* $\texttt{lft\_ind}_s$ *inhabiting the following type:*

$$\texttt{lft\_ind}_s \; \texttt{:} \; \forall \texttt{A}_1 \; \underline{\texttt{PA}_1}\texttt{,} \; \underline{\texttt{HPA}_1} \; \ldots \; \texttt{A}_n \; \underline{\texttt{PA}_n}\texttt{,} \; \underline{\texttt{HPA}_n} \; \texttt{(ins : indices),}$$
$$\forall \texttt{(t : ind pos\_block A}_1 \; \ldots \; \texttt{A}_n \; \texttt{ins), ind}_s \; \texttt{pos\_block A}_1 \; \texttt{PA}_1 \; \ldots \; \texttt{A}_n \; \texttt{PA}_n \; \texttt{ins t}$$

Proof. The proof is done very directly by `fix` and `match` on `t`, as it suffices to inhabit the added hypotheses. Given an argument:

(1) The constant case is trivial as no hypothesis is added
(2) The strictly positive parameter case holds by hypothesis.
(3) The recursive case is inhabited by recursion.
(4) The nested case holds recursively as the theorem hold for the inductive type used for nesting, and the predicates added for nesting hold recursively. □

4.2 Eliminators for Nested Inductive Types

We have now defined sparse parametricity and proven the local fundamental lemma. The elimination principles are generated following the usual procedure, except that we use sparse parametricity to define the recursion hypotheses of nested arguments, and the local fundamental lemma to prove them. In particular, this means the elimination principles we obtain are the same as usual one for non-nested inductive types, making our approach a conservative extension of the non-nested case. This was demonstrated through a backward-compatible pull request to Rocq, as discussed in Section 6.

Elimination Principles. Given an inductive block with its indices:

(1) Each parameter of the inductive type is quantified `par : params`
(2) Each inductive block `ind par : indices → Type` corresponds to a predicate $\texttt{P}_i$ `: ∀(ins : indices),` $\texttt{ind}_i$ `par ins → Type`
(3) For each constructor of each inductive block `c`, an assumption `Pc` is added whose definition is stated below
(4) We conclude the predicate of the ith blocks holds `∀ins (t :` $\texttt{ind}_i$ `par ins),` $\texttt{P}_i$ `ins t`

Given a constructor `c : ∀`$\texttt{x}_1 \dots \texttt{x}_n$`, block pos_block inst_indices`, for each argument, we add a recursion hypothesis or not depending on its type, that is `Pc : ∀`$(\texttt{x}_1 : \texttt{T}_1)$ $\underline{(\texttt{IHx}_1 : \texttt{IH}_s\ \texttt{x}_1\ \texttt{T}_1)}$ $\dots$ $(\texttt{x}_n : \texttt{T}_n)$ $\underline{(\texttt{IHx}_n : \texttt{IH}_s\ \texttt{x}_n : \texttt{T}_n)}$ `, P inst_indices (c` $\texttt{x}_1 \dots \texttt{x}_n$`)` where $\texttt{IH}_s$ `: term → term → option term` computes the recursion hypothesis. This function is easy to define using `argument`. For constants and strictly positive parameters we do not return a recursion hypotheses as there is no recursion happening. For inductive blocks, we return a proof of the motive. For nested arguments, we use sparse parametricity that we instantiate by recursively computing the recursion hypotheses for the instantiation of the parameters.

Inhabiting Elimination Principles. Thanks to the local fundamental lemma for sparse parametricity, nested elimination principles can easily be inhabited using `fix` and `match`. Indeed, we need to inhabit each added recursion hypotheses. The ones added for recursive arguments hold trivially by recursion as usual. The ones added for nested arguments hold using the local fundamental lemma for sparse parametricity, and that the recursion hypotheses used for nesting hold by recursion. The formal definition of this procedure can be found in *[eliminators.v]*. For example, for `typing` nested with `All`, this gives the definition of `typing_elim` given in Appendix C.

In Rocq, this guarantees that elimination principles can be automatically generated, provided the guard condition accepts the nested fixpoints. As long as the proof of the local fundamental lemma is transparent and can be unfolded, recursion proceeds only on strictly smaller subterms, thereby ensuring termination.

In Lean, elimination principles can be postulated directly, with their reduction rules defined using the pattern-matching branches from the corresponding fixpoint definition. In this setting, the local fundamental lemma appears on the right-hand side of the reduction rules, as for instance `lfl_All_s` in the reduction of `typing_elim` in Appendix C.

To justify that we have generated the correct eliminators for nested inductive types, we show in Appendix F that they can also be defined and proven for their mutual encoding. The resulting elimination principles satisfy the reduction rules only up to propositional equality, making them useful primarily as a sanity check rather than as first-class definitions.

5 Related work

In Section 2.4, we discuss the works of Johann and Polonsky [21], Tassi [30], which are the closest to our own, and highlight their limitations that we address in the remainder of the paper. We now discuss work in settings fundamentally different to ours.

Impredicative Encoding. Regarding other variants of dependent type theory, Cedille [29] aims at being foundationally more sparse than systems like Rocq or Lean, not implementing any inductive types and instead implementing them through impredicativity and Church encodings. Nested induction is supported via course-of-values induction schemes [17].

Indutive-Inductive Encoding. Considering other major proof assistants, Agda supports several variants of inductive types (via compiler flags), including nested and inductive–inductive types. In principle, our proposed positivity criterion could extend to Agda 's setting, where direct mutual encoding may yield inductive–inductive definitions. However, since Agda lacks automatic generation and systematic use of elimination principles, evaluating the practical usability of nested inductive types in this context remains difficult. In particular, the standard approach to defining functions on nested inductive types in Agda relies on sized types, which would require significant modifications to generate the elimination principle this way.

Categorical Perspective. Regarding a categorical view, Abbott et al. [1] treat a translation of a categorical version of nested inductive "types" to a categorical version of W-"types". Johann and Ghiorzi [19], Johann et al. [20] also considered parametricity categorically for System F extended with "true" nested inductive types, that is including the inductive type `Bush`.

Direct Encoding. The most complete treatment of encodings for inductive types is carried out in Isabelle/HOL, where (co)datatypes are not native outside of a few primitive ones, like `bool`, `nat` or `sum`. Instead, inductive types are explicitly encoded using Bounded Natural Functors (BNFs) that are closed by design under the primitive datatypes, composition, and initial algebra and final coalgebra [10, 31]. This provides Isabelle/HOL with a modular and powerful support for datatypes, including mixed inductive coinductive datatypes like infinite finitely branching trees without extending the theory. It was further used and developed to provide a better support to corecursion [14], non-uniform parameters [13], or binders [12].

The concept of Quotient Polynomial Functors (QPFs) was developed in Lean as an alternative to BNF [6]. The authors leverage Lean's native support for quotients to extend the class of polynomial functors to quotient polynomial functors, and prove that like BNFs, QPFs are closed under composition, initial algebra and final coalgebra. They further show that QPFs are more expressive than BNFs [6, Section 5], while being more natural in type theory, e.g. by avoiding cardinality conditions or preservation of weak pullback which is required for BNF to be expressible in Isabelle/HOL [31, Section 3.C]. However, compared to Isabelle/HOL, the automatic generation of QPFs and of the encoding on top of it is still very limited, though recent progresses have been made [22]. A categorical encoding of inductive types was also achieved in Rocq's Unimath library [4, 5] using univalent categories, ω-cocontinuous functors, and algebra, only assuming basic inductive types like `nat`. Compared to our work, neither of this approach support dependent types. This greatly limits their use for nesting and specifications in dependent type theory, as they are not expressive enough to define inductive types like `All` or `Typing`. In the non-dependent case, the eliminator we generate coincide with theirs as sparse parametricity applied to constant predicate amounts to functoriality. For instance, when defining a function `f` by recursion, the expression $\texttt{All}_s$ `(fun _ ⇒ B) l` is the equivalent to the pair `(l, map f l)`. However, BNF and QPF have the advantage to support mixes of inductive and coinductive type, which elaborating nested inductive types to mutual inductive cannot justify. Yet, to make QPF scale to dependent types, one would most likely need to adopt a variant of our sparse parametricity to deal with dependencies.

Induction Principle Generation. Previous tools have been provided to generate induction principles for nested inductive types and predicates. Notably, Tassi's study came with a usable tool implemented in Rocq Elpi [30], which however does not treat mutual nested types, a requirement e.g. for MetaRocq. Ullrich [32] has used MetaRocq's meta-programming support to implement a similar approach to Tassi's, similarly not covering mutual inductives, and also experiencing problems with the fundamental lemma for the non-sparse parametricity translation.

6 Implementation

MetaRocq plugin. We prototyped our approach in MetaRocq, the metaprogramming framework for Rocq, by formalizing the full elaboration pipeline from nested to mutual inductive types together with automatic generation of sparse-parametric elimination principles. The implementation relies critically on the view mechanism introduced in MetaRocq, which provides a stable, syntax-directed interface to the internal representation of inductive types. This design isolates our transformations from low-level kernel details and enables robust rewriting even as the interactive theorem prover evolves. The plugin has successfully generated the appropriate elimination principle for the real world example of `typing` of MetaRocq[5] which specifies the complete typing rules for Rocq's kernel and contains a dozen of nested occurrences with complex containers.

Integration in Rocq. As further evidence of the usefulness of the view mechanism, we have submitted a pull request—not linked to preserve anonymity—to replace Rocq's old generation of elimination principles, including mutual ones. Although sparse parametricity is not yet included, the PR shows that all existing Rocq developments compile without modification, providing strong empirical confirmation of the conservativity of our approach.

Integration in Lean. Our approach could be integrated into Lean in two senses: First, our translation of nested inductive types to mutual inductive types properly treating indices could be integrated in Lean's kernel, extending the nested inductives it accepts. Secondly, our generation of induction principles via sparse parametricity could be included, allowing Lean's `induction` tactic to work for nested inductive types. This change could be done outside of the kernel, making working with the already accepted nested inductive types easier for users.

7 Conclusion

In this work, we have set a theoretical foundation to add nested inductive types to Rocq and Lean, which makes them practical to use, while maintaining consistency. To do so, we have introduced a new strict positivity condition for nested inductive types that ensures statically that a nested inductive has a mutual encoding. Using this new positivity condition, we have defined a sparse parametricity for nested types and proved a local fundamental theorem for it, which enabled us to automatically generate useful eliminators for nested inductive types as showcased by the highly nested `typing` definition of MetaRocq. We have justified this preserves the consistency of the type system by encoding the nested elimination principles with the elimination principles of their mutual encoding. We have not formalized the encoding of eliminators for these types, as it remains out of reach: no verified meta-programming framework currently provides the infrastructure needed to manage the intricate typing derivations involved. We believe that supporting such encodings would represent a realistic and meaningful challenge for advances in verified meta-programming. An interesting direction for future work is the extension of our approach to truly nested inductive

[5]https://metarocq.github.io/html/MetaRocq.Template.Typing.html#typing

types, *i.e.,* types that use themselves as containers. Such types are currently rejected by all existing systems.

References

[1] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. 2004. Representing Nested Inductive Types Using W-Types. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings (Lecture Notes in Computer Science, Vol. 3142)*, Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella (Eds.). Springer, 59–71. doi:10.1007/978-3-540-27836-8_8

[2] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2018. Decidability of Conversion for Type Theory in Type Theory. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 23 (Jan. 2018), 29 pages. doi:10.1145/3158111

[3] Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. 2024. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, London, UK, January 15-16, 2024*, Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy (Eds.). ACM, 230–245. doi:10.1145/3636501.3636951

[4] Benedikt Ahrens, Ralph Matthes, and Anders Mörtberg. 2019. From Signatures to Monads in UniMath. *J. Autom. Reason.* 63, 2 (2019), 285–318. doi:10.1007/S10817-018-9474-4

[5] Benedikt Ahrens, Ralph Matthes, and Anders Mörtberg. 2022. Implementing a category-theoretic framework for typed abstract syntax. In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, Andrei Popescu and Steve Zdancewic (Eds.). ACM, 307–323. doi:10.1145/3497775.3503678

[6] Jeremy Avigad, Mario Carneiro, and Simon Hudon. 2019. Data Types as Quotients of Polynomial Functors. In *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA (LIPIcs, Vol. 141)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:19. doi:10.4230/LIPICS.ITP.2019.6

[7] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. 2015. A Presheaf Model of Parametric Type Theory. *Electronic Notes in Theoretical Computer Science* 319 (2015), 67–82.

[8] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. 2012. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming* 22, 2 (March 2012), 107–152.

[9] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. doi:10.1007/978-3-662-07964-5

[10] Julian Biendarra, Jasmin Christian Blanchette, Aymeric Bouzy, Martin Desharnais, Mathias Fleury, Johannes Hölzl, Ondrej Kuncar, Andreas Lochbihler, Fabian Meier, Lorenz Panny, Andrei Popescu, Christian Sternagel, René Thiemann, and Dmitriy Traytel. 2017. Foundational (Co)datatypes and (Co)recursion for Higher-Order Logic. In *Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Brasília, Brazil, September 27-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10483)*, Clare Dixon and Marcelo Finger (Eds.). Springer, 3–21. doi:10.1007/978-3-319-66167-4_1

[11] Richard S. Bird and Lambert G. L. T. Meertens. 1998. Nested Datatypes. In *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1422)*, Johan Jeuring (Ed.). Springer, 52–67. doi:10.1007/BFB0054285

[12] Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. 2019. Bindings as bounded natural functors. *Proc. ACM Program. Lang.* 3, POPL (2019), 22:1–22:34. doi:10.1145/3290335

[13] Jasmin Christian Blanchette, Fabian Meier, Andrei Popescu, and Dmitriy Traytel. 2017. Foundational nonuniform (Co)datatypes for higher-order logic. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 1–12. doi:10.1109/LICS.2017.8005071

[14] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. 2015. Foundational extensible corecursion: a proof assistant perspective. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 192–204. doi:10.1145/2784731.2784732

[15] Mario Carneiro. 2024. Lean4Lean: Towards a Verified Typechecker for Lean, in Lean. arXiv:2403.14064 [cs.PL] https://arxiv.org/abs/2403.14064

[16] Thierry Coquand and Christine Paulin. 1988. Inductively Defined Types. In *Proceedings of the International Conference on Computer Logic (COLOG '88)*. Springer-Verlag, Berlin, Heidelberg, 50–66.

[17] Denis Firsov, Larry Diehl, Christopher Jenkins, and Aaron Stump. 2018. Course-of-Value Induction in Cedille. *CoRR* abs/1811.11961 (2018). arXiv:1811.11961 http://arxiv.org/abs/1811.11961

[18] Kesha Hietala and Emina Torlak. [n. d.]. Lean Into Verified Software Development. https://aws.amazon.com/blogs/opensource/lean-into-verified-software-development/

[19] Patricia Johann and Enrico Ghiorzi. 2021. Parametricity for Nested Types and GADTs. *Log. Methods Comput. Sci.* 17, 4 (2021). doi:10.46298/LMCS-17(4:23)2021

[20] Patricia Johann, Enrico Ghiorzi, and Daniel Jeffries. 2021. Parametricity for Primitive Nested Types. In *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12650)*, Stefan Kiefer and Christine Tasson (Eds.). Springer, 324–343. doi:10.1007/978-3-030-71995-1_17

[21] Patricia Johann and Andrew Polonsky. 2020. Deep Induction: Induction Rules for (Truly) Nested Types. In *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12077)*, Jean Goubault-Larrecq and Barbara König (Eds.). Springer, 339–358. doi:10.1007/978-3-030-45231-5_18

[22] Alex C. Keizer. 2023. *Implementing a definitional (co)datatype package in Lean 4, based on quotients of polynomial functors.* Master's thesis. Universiteit van Amsterdam.

[23] Chantal Keller and Marc Lasson. 2012. Parametricity in an Impredicative Sort. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France (LIPIcs, Vol. 16)*, Patrick Cégielski and Arnaud Durand (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 381–395. doi:10.4230/LIPICS.CSL.2012.381

[24] Robbert Krebbers. 2023. Efficient, Extensional, and Generic Finite Maps in Coq-std++. In *The Coq Workshop.* https://coq-workshop.gitlab.io/2023/abstracts/coq2023_finmap-stdpp.pdf

[25] Neelakantan R. Krishnaswami and Derek Dreyer. 2013. Internalizing Relational Parametricity in the Extensional Calculus of Constructions. In *Proceedings of the Conference for Computer Science Logic (CSL 2013).* 432–451.

[26] Ralph Matthes. 2009. An induction principle for nested datatypes in intensional type theory. *J. Funct. Program.* 19, 3-4 (2009), 439–468. doi:10.1017/S095679680900731X

[27] John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress.* 513–523.

[28] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Botsch Nielsen, Nicolas Tabareau, and Théo Winterhalter. 2024. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *Journal of the ACM (JACM)* (Nov. 2024), 1–76. doi:10.1145/3706056

[29] Aaron Stump. 2018. Syntax and Semantics of Cedille. *CoRR* abs/1806.04709 (2018). arXiv:1806.04709 http://arxiv.org/abs/1806.04709

[30] Enrico Tassi. 2019. Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq. In *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA (LIPIcs, Vol. 141)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:18. doi:10.4230/LIPICS.ITP.2019.29

[31] Dmitriy Traytel, Andrei Popescu, and Jasmin Christian Blanchette. 2012. Foundational, Compositional (Co)datatypes for Higher-Order Logic: Category Theory Applied to Theorem Proving. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012.* IEEE Computer Society, 596–605. doi:10.1109/LICS.2012.75

[32] Marcel Ullrich. 2020. Generating induction principles in MetaCoq. (2020). Bachelor's Thesis, Saarland University, https://www.ps.uni-saarland.de/~ullrich/bachelor.php.

[33] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics.* Institute for Advanced Study. https://homotopytypetheory.org/book/

A Strict Positivity for (Non-Nested) inductive Types

Let us consider an inductive type definition of the form:

```
Inductive I (x₁ : A₁) ... (x_n : A_n) : Type := c₁ : T₁ | ... | cₙ : Tₙ.
```

The type constructor I is said to satisfy the *strict positivity condition* if, in each constructor type T_i, any recursive occurrence of I appears only in strictly positive positions. Formally:

Definition A.1 (Strict Positivity for Non-Nested Inductive Types). An inductive type I occurs *strictly positively* in a type T if it satisfies the following inductive rules:

(1) **Base cases:**
- I does not occur in T.
- I occurs as I t_1 ... t_n and I does not occur in any of the t_i.

(2) **Function types:** If T converts to ∀a:A, B, then I is strictly positive in T if:
- I does **not** occur in A (the domain), and
- I is strictly positive in B (the codomain).

Note that it is not possible in general to rely on a weaker notion known simply as the positivity criterion, which permits recursive occurrences on the left-hand side of an arrow, provided they are under an even number of arrows (*e.g.*, `(I → False) → False` is considered positive). However, this weaker criterion has been shown to be too permissive in the presence of impredicativity, as demonstrated by Coquand and Paulin [16]. Their argument, based on a variant of Cantor's diagonalization, constructs a contradiction using an inductive type in `Prop` with a seemingly positive recursive occurrence of the form `(I → Prop) → Prop`.[6]

B Impossibility of Nesting on Non-Uniform Parameters and Indices

It is not possible to nest over non-uniform parameters or indices, as no general translation to mutual inductive types exists. To illustrate this, consider attempting to nest over the non-uniform parameter of the power list type `plist`, where an attempt to define a type `PowerTree A` is rejected in Rocq for positivity issues:

```
Inductive plist (A : Type) : Type :=
| pnil  : plist A | pcons : A → plist (A * A) → plist A.


Fail Inductive PowerTree A : Type :=
| leaf (a : A) : PowerTree A | node (l : plist (PowerTree A)) : PowerTree A.
```

This is because the mutual inductive encoding used for rose trees does not lift to `PowerTree A`. Indeed, applying the same technique would give rise to a local copy `plist_mut A` representing `plist (powerTree A)`. However, ther recursive argument of `pcons_mut`, which should correspond to `plist (powerTree A * powerTree A)`, cannot be expressed using `plist_mut A` alone. In contrast, for rose trees using `list`, it is essential that the parameter `A` is uniform—meaning that all recursive occurrences of the container are also applied to the same `A`. As a result, each instance of `list_rose_tree_mut A` directly corresponds to the encoding of `list (rose_tree_mut A)` in the mutual definition. What happens for `plist` is that its parameter is non-uniform: the recursive occurrence appears on `A * A` instead of `A`. This non-uniformity prevents us from inlining `PowerTree A` directly into a mutual inductive definition.

[6]It remains an open question whether the positivity criterion is sufficient in a purely predicative setting.

C Examples of Elimination Principles for Nested Inductive Types

The elimination principle expected for rose trees can be defined as (note the need for $\texttt{list}_\varepsilon{}^{fl}$, the fundamental lemma of parametricity for lists, to inhabit the induction hypothesis for `Pnode`):

```
Fixpoint rose_tree_elim A P Pleaf Pnode (t : rose_tree A) {struct t} : P t :=
   match t with
   | leaf a ⇒ Pleaf a
   | node l ⇒ Pnode l (list_ε^fl (rose_tree A) P (rose_tree_elim A P Pleaf Pnode) l)
   end.
```

Note that in the definition, we can directly use $\texttt{list}_\varepsilon{}^{fl}$ as parametricity and sparse parametricity coincide for the lists.

Similarly, the elimination principle for `typing` is:

```
Fixpoint typing_elim (P : ∀t T, typing t T → Type)
  (Pswitch : ∀discr branches n return_type,
           ∀(ty_discr : typing discr (fintype n)), P discr (fintype n) ty_discr →
           ∀(ty_br : All (fun a ⇒ typing a return_type) branches),
             All_s (fun a ty_a ⇒ P a return_type ty_a) ty_br →
           P _ _ (typing_switch discr branches n return_type ty_discr ty_br)) :
    ∀t T ty_tT, P t T ty_tT :=
  fun t T ty_tT ⇒ match ty_tT with
  | typing_switch discr br n return_type ty_discr ty_br ⇒
      Pswitch discr br n return_type ty_discr (typing_elim P Pswitch _ _ ty_discr)
        ty_br (lfl_All_s (fun t ⇒ typing_elim P Pswitch t return_type) _ ty_br)
  end.
```

In Lean, the elimination principle can simply be postulated, with the corresponding rewrite rules. For instance, in the case of `typing_elim` the rewrite rule is

```
typing_elim P Pcse t T (typing_switch discr ind br return_type ty_discr ty_br) ⇝
  Pswitch discr ind br return_type ty_discr (typing_elim P Pswitch _  _ ty_discr)
    ty_br (lfl_All_s (fun t ⇒ typing_elim P Pswitch t return_type) _ ty_br)
```

D View and Positivity

The inductive type `argument` is used to define the view on the type of constructors of inductive definitions.

```
Inductive argument : Type :=
| arg_is_cst    (t : term)
| arg_is_ind    (largs : list term) (pos_block : nat) (inst_indices : list term)
| arg_is_sup    (largs : list term) (k : nat) (inst_args : list term)
| arg_is_nested (largs : list term) (ind : inductive)
                (inst_params : list (list term * term)) (inst_indices : list term).
```

The file *[Positvity_condition.v]* contains this definition together the formal definition of sparse parametricity.

E Example of Sparse Paramericity

For example, for `All` we add a hypothesis for `P`, but not for `A` as it is not positive as it appears in indices. After simplification, this gives us the following type for the sparse parametricity of `All`:

```
  Inductive All {A : Type} (P : A → Type) : list A → Type
  ------------------------------------------------------------------------------
  Inductive All_s {A} {P : A → Type} (P_P : ∀a, P a → Type) : ∀{l}, All P l → Type
```

For the constructor `All_cons` of type

```
∀(a : A) (l : list A), P a → All P l → All P (a::l)}
```

which is only nestable on the parameter `P` the computation of the view tells us that:

- `A` and `list A` are constants as `A` cannot be nested on, and are hence recorded with `var_is_cst`, that is `var_is_cst A` and `var_is_cst (list A)`.
- `P a` is a strictly positive parameter `P`, the first parameter starting from 0, with arguments `a`. As it is a trivial product, `largs := []`, and it is recorded by `arg_is_sup [] 1 [a,b]`.
- Similarly, `All R l` is a recursive argument recorded by `arg_is_ind [] 0 [l]`.

This gives us the following constructor `All_cons_s` which simplifies after computing the sparse parametricity as many arguments are constant, and do not create additional hypotheses.

```
| All_cons : ∀(a : A) (l : list A) (r : P a) (x : All P l), All P (a::lA)
---------------------------------------------------------------------------------
| All_cons_s : ∀a (a_s : ⟦a : A⟧_s) l (l_s : ⟦l : list A⟧_s) r (r_s : ⟦r : P a⟧_s)
                x (x_s : ⟦x : All P l⟧_s), All_s P_P (All_cons a l r x)
```

F Encoding Nested Eliminators with Their Mutual Encoding

To justify that we have generated the correct eliminators for nested inductive types, we show that they can also be defined and proven for their mutual encoding, as described in Section 3.1. The resulting elimination principles satisfy the reduction rules only up to propositional equality, making them useful primarily as a sanity check rather than as first-class definitions.

Case Study. First, to be able to define the nested elimination principle for its mutual encoding `typing_mut`, we need it to have the same type as `typing`. This is the case as the types of the original inductive blocks are preserves by the encoding. Second, we need to define the constructors of `typing` over `typing_mut`. In our case, it means defining `typing_switch'` using `typing_mut_switch` whose only difference is using a nested argument versus a recursive argument, as showed underlined below:

```
  typing_mut_switch discr ind branches return_type :
    typing_mut discr ind → All_mut return_type branches →
    typing_mut (switch discr branches) return_type
  ----------------------------------------------------------------------------
  typing_switch' discr ind branches return_type :
    typing_mut discr ind → All (fun a => typing_mut a return_type) branches →
    typing_mut (switch discr branches) return_type
```

To do so, we need to prove the nested argument can be proven out of their mutual encoding, and vice versa. That is given the context of the nested argument `return_type branches`, we need to define a type equivalence in the sense of Homotopy Type Theory (HoTT) and Univalent Foundations Program [33].

```
  All_mut return_type branches ≅ All (fun a : term ⇒ typing' a return_type) branches.
```

The construction of the two transport functions (`All_to_All_mut` and `All_mut_to_All`) is done by induction and a direct application of each constructor as by definition of the encoding this amounts to relabeling. The proof that they form an equivalence involves basic HoTT machinery. It is then possible to define `typing_switch'` as

```
typing_switch' discr ind branches return_type ty_discr ty_branches :=
typing_mut_switch discr ind branches return_type ty_discr
  (All_to_All_mut ty_branches).
```

We can now state the nested elimination principle instantiated for `typing_mut`, and try to prove it using the mutual eliminator of `typing_mut` which requires one predicate `P` on `typing_mut` and one predicate `P0` on `All_mut`. The main difficulty is to deal with homogeneity, and switching between `All` and `All_mut`. Indeed, we need to instantiate `P0` with the sparse parametricity. Yet, as it expects a predicate on `All_mut`, we need to transport sparse parametricity, as:

```
P0 := fun return_type branches ty_branches ⇒
      All_s (fun a ty_a ⇒ P a return_type ty_a) (All_mut_to_All ty_branches)
```

However, combined with the encoding of `typing_switch'`, this naturally leads to a back and forth between the use of `All_to_All_mut` and `All_mut_to_All`. Consequently, we need to rewrite hypothesis (using the # notation for transport) with the equality `All_mut_eq` coming from the equivalence between `All` and `All_mut`. The goal then follows from the hypotheses as the remaining `All_mut_to_All` as already been added to the hypotheses through the instantiation of `P0`. Third, we have to prove that `P0All_mut_nil` and `P0All_mut_cons`. As transports computes on constructors, this simplifies and amounts to inlining a proof of the local fundamental lemma. Put altogether, this gives us the following application where we underline transports.

```
Definition typing_elim2
  (P : ∀t T, typing_mut t T → Type)
  (Pswitch' : ∀discr ind branches return_type,
      ∀(ty_discr : typing_mut discr ind), P discr ind ty_discr →
      ∀(ty_branches : All (fun a ⇒ typing_mut a return_type) branches),
        All_s (fun a ty_a ⇒ P a return_type ty_a) ty_branches →
        P _ _ (typing_switch' discr ind branches return_type ty_discr ty_branches)) :
 ∀t T ty_tT, P t T ty_tT :=
typing_mut_All_mut_rec
  (P := fun t T ty_tT ⇒ P t T ty_tT)
  (P0 := fun return_type branches ty_branches ⇒
        All_s (fun a ty_a ⇒ P a return_type ty_a) (All_mut_to_All ty_branches)).
  (fun discr ind branches return_type ty_discr Pty_discr ty_branches Pty_branches ⇒
   (All_mut_eq ty_branches) # (Pswitch' discr ind branches return_type
       ty_discr Pty_discr (All_mut_to_All ty_branches) Pty_branches))
  (fun return_type ⇒ All_s_nil)
  (fun return_type a l ty_a Pty_a ty_l Pty_l ⇒ All_s_cons a l ty_a Pty_a
  (All_mut_to_All ty_l) Pty_l)
```

To get a full-encoding, we also need to show that the definitional equality of the elimination principle are preserved. This is *not* the case here due to back and forth transport between `All` and `All_mut` that are inverse only propositionally, and only definitionally on closed terms. Therefore, the encoding of the nested eliminator verify the computation rules only propositionally, or definitionally on closed terms. The proof is non-trivial and requires us to simplify and to cancel transports, followed by showing that the two proofs of the fundamental lemma are equal—something that can be established by induction. We refer the interested reader to the companion Rocq formalization for full details of the proof.

As a consequence, this encoding justifies only the consistency of the nested inductive types and nested eliminators, and not that termination of the type checking is preserved. However, this not an issue as type checking is not decidable in Lean, and it is ensured in Rocq by the guard checking which is simple for nested eliminators as it reduces on strict subterms.

General Case. This method seem to generalize directly to arbitrary complex nested inductive types. For each nested argument, we define transport back and forth between the nested instantiation and its specialization, prove they are inverses to define the encoding of the eliminators, and proven to satisfy a coherence law to prove the propositional equality. These are direct applications of the elimination principles as it consists in renaming, modulo a small simplification of transports for the coherence law. The nested constructors can then be defined by appropriately inserting transport, which succeeds as by the positivity condition nested arguments can not depend on previous nested arguments. Finally, the encoding of the eliminators can be proven using the mutual encoding instantiated using sparse parametricity up to transport, by rewriting with the equality as for `typing`. The propositional rule then follows by simplifying the transports, and proving both proofs of the fundamental lemma are the same which is trivial.

Although we are fairly confident this method succeeds, we emphasize that, due to the complexity of inductive types, without a verified meta-programming framework to build complex typing derivations and certify this proof, *absolute* certainty remains unattainable. In particular, the absence of definitional computational rules for the encoding of the nested eliminators might force us to move to an extensional type theory for a correctness proof.