SAARLAND UNIVERSITY

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

BACHELOR'S THESIS

GENERATING INDUCTION PRINCIPLES FOR NESTED INDUCTIVE TYPES IN METACOQ

Author
Marcel Ullrich

Advisor
Yannick Forster

Reviewers
Prof. Dr. Smolka
Prof. Dr. Hack

Submitted: 23rd June 2020

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Raguhn-Jeßnitz, 23rd June, 2020

Abstract

We implement a plugin to generate induction principles in MetaCoq. Inductive types and their induction principles are essential components in modern dependent type theory and proof assistants like Coq, where most proofs involve induction and a great deal of predicates are written in an inductive fashion.

While Coq automatically generates induction principles, these principles are too weak for nested inductive types. Therefore, one needs to put in additional work to get induction principles that are strong enough. It is folklore that induction principles for nested inductive types require creativity and lots of work. Only recently the underlying principle to generate these induction principles was discovered.

We implement a plugin in MetaCoq to automatically generate such induction principles based on the unary parametricity translation. Furthermore, we verify the correctness of parts of the plugin using MetaCoq.

Acknowledgements

I want to thank my advisor Yannick Forster for the helpful discussions and ideas throughout the thesis. I also appreciate the plentiful feedback on this thesis.

I am grateful to Professor Smolka for the opportunity to write this thesis at his chair and introducing me to formal proofs, type theory and the proof assistants Coq. I am thankful for his advice and support as mentor throughout my Bachelor studies.

I want to thank Professor Smolka and Professor Hack for reviewing this thesis. In addition, I thank my friends and family for the support I received. Especially, I would like to thank Anton, Lennard, and Philippe for proof-reading this thesis.

Contents

Chapter 1

Introduction

Induction is one of the most important techniques in modern dependent type theory. In the proof assistant Coq [34] that is based on the calculus of inductive constructions types, predicates and functions are defined inductively, and nearly all interesting proofs use induction or rely on lemmas that use induction. Therefore, induction principles play an essential role in Coq and modern dependent type theory in general. It is thus all the more important for Coq-like proof assistants, like Adga and Lean, to automatically generate induction principles.

Inductive definitions define how objects of a type can be constructed. The definition is done using constructors for each way instances of the type should be able to be created. An example of an inductive definition are natural numbers:

$$n : \mathbb{N} ::= O \mid S\ n$$

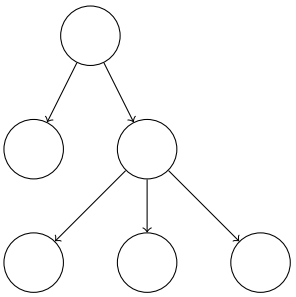
A number is either zero or the successor of another number.

Each inductive type has according induction principles. Induction principles allow to prove statements on all elements of an inductive type. To do so, case analysis is performed with one case for each constructor that allows to generate elements of the type. Induction hypotheses are added for structurally smaller arguments in the constructors of the type. The induction hypotheses state that the predicate holds for the structurally smaller arguments. The induction principle of natural numbers has two cases, one for the zero case and one for the successor case:

$$\forall (P : \mathbb{N} \to \mathbb{T}).\ P\ O \to (\forall (n : \mathbb{N}).\ P\ n \to P\ (S\ n)) \to \forall (n : \mathbb{N}).\ P\ n$$

In the successor case the induction hypothesis states that P holds for n and under this assumption $P\ (S\ n)$ has to be shown.

Coq generates induction principles when a type is defined, but these automatically generated principles are too weak for nested inductive types like rose trees. Rose



$\mathtt{node}\ [\mathtt{node}\ [];\ \mathtt{node}\ [\mathtt{node}[]; \mathtt{node}[]; \mathtt{node}[]]]$

Figure 1.1: *An example rose tree with a size of six and a depth of two. On the left is a drawing of the rose tree and the Coq term is on the right. The root node has two direct sub-trees and the right sub-tree has three direct sub-trees.*

trees are defined by nodes with a list of direct sub-trees:

$$\mathtt{roseTree} ::= \mathtt{node}\ (\mathtt{xs} : \mathtt{list}\ \mathtt{roseTree})$$

The principle generated by Coq is:

$$\forall P.\ (\forall xs.\ P\ (\mathtt{node}\ xs)) \to \forall r.\ P\ r$$

This principle is too weak as it gives no assumption for the direct sub-tree list xs. One would hope to get the assumption that P holds for every tree in xs. A better induction principle would be:

$$\forall P.\ (\forall xs.\ (\forall t.t \in xs \to P\ t) \to P\ (\mathtt{node}\ xs)) \to \forall r.\ P\ r$$

The generation of principles for such types is often needed and is done by hand. There are common techniques to derive the principles if the argument is a list like in rose trees. But for other types it is more complicated and Chlipala for example states that "[...] it takes creativity to figure out good ways to incorporate nested uses of different families" [5]. It was recently discovered [33, 19], that unary parametricity can be used as an underlying theory to generate induction principles for nested inductive types.

Lemma 1.1 (depth bound) *To illustrate why the better principle is needed we prove informally that* $\mathtt{depth}\ t < \mathtt{size}\ t$ *holds. Here, the* depth *of a tree is the length of the longest path from the root to a leaf, and* size *is the count of nodes in the tree (see Figure 1.1).*

$$\begin{aligned} \mathtt{size}\ (\mathtt{node}\ xs) &:= 1 + \mathtt{sum}\ (\mathtt{map}\ \mathtt{size}\ xs) \\ \mathtt{depth}\ (\mathtt{node}\ [\,]) &:= 0 \\ \mathtt{depth}\ (\mathtt{node}\ (x :: xs)) &:= 1 + \mathtt{max}\ (\mathtt{map}\ \mathtt{depth}\ (x :: xs)) \end{aligned}$$

To prove the statement, induction over the tree and then over the direct sub-tree list is performed. The base case for an empty direct sub-tree list simplifies to $0 < 1$. *The goal becomes* $\mathtt{max}\ (1 + \mathtt{depth}\ x)\ (\mathtt{depth}\ (\mathtt{node}\ xs)) < \mathtt{size}\ x + \mathtt{size}\ (\mathtt{node}\ xs)$ *with the induction hypothesis* $IH_t : \forall t.\ t \in x :: xs \to \mathtt{depth}\ t < \mathtt{size}\ t$ *from the better induction principle for rose trees and* $IH_{xs} : \mathtt{depth}\ (\mathtt{node}\ xs) < \mathtt{size}\ xs$.

From IH_t *one can get the assumption that* $\mathtt{depth}\ x < \mathtt{size}\ x$ *and therefore* $1 + \mathtt{depth}\ x \leqslant \mathtt{size}\ x$ *which suffices to conclude the proof. The important assumption over the relation between* depth *and* size *of the direct sub-tree* x *was provided by* IH_t.

This proof shows that the relation is provable with the stronger induction principle for rose trees. Additionally, one can see that it is not possible to prove the statement with the principle provided by Coq.

Parametricity translations [26] are a technique used to express relations over objects of types. The parametricity translations are often used to derive statements purely from the type of functions [37]. The unary parametricity translation generates the unary parametricity relation of a type. The unary parametricity relation of an inductive type can be viewed as a predicate over elements of that type.

We use unary parametricity, previously explored by Tassi [33], to solve this predicament and derive strong induction principles for general uses of nested induction. To implement the generation of strong induction principles one could write a Coq plugin. But plugin development for Coq is complicated as one has has to understand large parts of the 210000 lines of mostly undocumented OCaml source code of Coq. There are multiple questions in the Coq club mailing list[1] that show that it is hard in plugin development to know how to do things properly.

Other theorem provers like Lean [11] use a metaframework to allow users to write plugins using meta-programming in a dedicated framework more suitable than plugins interacting directly with the source code. We use the MetaCoq project [31] to implement a plugin for strong induction principles in Coq. MetaCoq allows to manipulate the environment of Coq directly in Coq and therefore enables the user to write plugins as Coq functions.

Our plugin provides an alternative of Coq's `Scheme` command to derive induction principles. The `Scheme` command allows to create induction principles for inductive types and is called automatically when a new inductive type is defined. For natural numbers the induction principle can be generated with:

```
Scheme ℕ_induct := Induction for ℕ Sort 𝕋
```

1 Talia Ringer, 10.04.2018: Plugins, type-checking, and universe constraints
Talia Ringer, 17.11.2017: Looking up definitions from inside an ml plugin

Our MetaCoq plugin generates the induction principle for rose trees with:

```
MetaCoq Run Scheme rtree_induct := Induction for rtree
```

$$\forall P.\ (\forall xs.\ \mathtt{list}^{t}\ \mathtt{roseTree}\ P\ xs \to P\ (\mathtt{node}\ xs)) \to \forall r.\ P\ r$$

$\mathtt{list}^{t}$ states that the predicate P holds for all elements in xs.

In this thesis we implement the generation of induction principles for nested inductive types as a MetaCoq plugin. To do so we show how to extend the theory of induction principles to nested inductive types based on the unary parametricity translation of inductive types as shown by Tassi [33]. We provide an overview over the landscape including the theory behind case analysis and induction principles for arbitrary types as well as an overview over the MetaCoq project and its capabilities. Lastly, we attempt to verify the correctness of the plugin. The correctness proof is not complete as it contains unfinished lemmas. The remaining unfinished lemmas are technical.

This thesis is based on many areas in the field of type theory [17]. The calculus of construction was presented in [6]. In the first presentation of the calculus of construction inductive types were not implemented. Instead impredicative characterizations with Church encodings were used. As this system is very difficult to work with and is not strong enough to prove facts like $0 \neq 1$, inductive definitions were added to the calculus of construction [7, 25]. Inductive definitions were added to Coq in [22].

Other areas influencing this thesis were parametricity translations [26], MetaCoq [31], and plugin development in Coq.

The accompanying Coq development is available at:

`https://www.ps.uni-saarland.de/~ullrich/bachelor.php`

The thesis introduces definitions and notation (Chapter 2), followed by an introduction to the MetaCoq project (Chapter 3). Afterwards, the theory and implementation of case analysis principles are discussed (Chapter 4) and extended to induction principles (Chapter 5). In Chapter 6 we will discuss how to extend structural induction to cover nested inductive types like rose trees and the implementation in MetaCoq. Lastly, the correctness of the plugin is proven in Chapter 7.

Chapter 2

Preliminaries

In this chapter we will introduce common inductive types used in later chapters.

There is a hierarchy of universes in type theory with an impredicative universe $\mathbb{P}$ for propositions. The predicative universes are written as $\mathbb{T}$. Each universe is typed in a larger universe. Predicates can be stated as inductive predicates. An example is the disjunction predicate $\mathsf{Or} : \mathbb{P} \to \mathbb{P} \to \mathbb{P}$ that takes two propositions and returns a proposition that one of the argument propositions holds.

In this thesis we use the polymorphic calculus of cumulative inductive constructions as implemented by the Coq proof assistant version 8.11 [34]. For the correctness proof we use version 8.9 of the Coq proof assistant. We use the MetaCoq project for version 8.11 of Coq for the plugin and MetaCoq for version 8.9 for the correctness proof.

We will use grammars in Backus-Naur form to describe non dependent types and inference rules to describe dependent types.

$$
\begin{aligned}
\mathbb{B} &::= \mathsf{true} \mid \mathsf{false} \\
n : \mathbb{N} &::= \mathsf{O} \mid \mathsf{S}\ n \\
\mathsf{Or}\ (X : \mathbb{P})\ (Y : \mathbb{P}) &::= \mathsf{L}\ (x : X) \mid \mathsf{R}\ (y : Y) \\
xs : \mathsf{list}\ X &::= [\,] \mid x :: xs \\
\mathsf{roseTree} &::= \mathsf{node}\ (xs : \mathsf{list}\ \mathsf{roseTree})
\end{aligned}
$$

Those non-dependent types are from top to bottom the boolean truth values with true and false, the natural numbers built using zero and successors, the or predicate for disjunctions over propositions X and Y built using a proof of either X or Y, lists over a type X that can be empty $[\,]$ or a concatenation of an element x to a list xs, and lastly rose trees that are built from a list of direct sub-trees.

Vectors $\mathit{vec} : \mathbb{T} \to \mathbb{N} \to \mathbb{T}$ are like lists with an additional index stating how many elements the vector contains.

$$\frac{}{vec\ X\ 0}\ nil \qquad\qquad \frac{vec\ X\ n \qquad x : X}{vec\ X\ (S n)}\ cons$$

Less or equal $le\ n\ m$ is an inductively defined predicate expressing that a number n is less or equal to m. It is defined with two constructors predicating that a number n is less or equal to itself, and if n is less or equal than m, n is also less or equal to the larger number $S\ m$.

$$\frac{}{le\ n\ n}\ le\_n \qquad\qquad \frac{le\ n\ m}{le\ n\ (S m)}\ le\_S$$

Even and odd are defined as mutual inductive predicates with three constructors. Zero is an even number, and if a number is even the successor is odd and conversely.

$$\frac{}{even\ O}\ even_O \qquad \frac{odd\ n}{even\ (S\ n)}\ even_S \qquad \frac{even\ n}{odd\ (S\ n)}\ odd_S$$

The guard predicate $G\ n$ states that a function $f : \mathbb{N} \to \mathbb{B}$ is true for a number larger or equal to n. It is defined using the guard (see Section 5.2.2) $f\ n = false$, rather than two constructors, in order to allow for large elimination.

$$\frac{f\ n = false \to G\ (S\ n)}{G\ n}\ G_I$$

The accessibility predicate is used to define well-foundedness and perform well-founded induction. $Acc\ R\ x$ holds if $Acc\ R\ y$ holds for all smaller elements according to $R : X \to X \to \mathbb{P}$.

$$\frac{\forall y.\ R\ y\ x \to Acc\ R\ y}{Acc\ R\ x}\ Acc_I$$

An instance of an inductive type T allows for **large elimination** if one can perform a case analysis on the instance with a return type of type $\mathbb{T}$. An instance of an inductive type $T : \ldots \to \mathbb{T}$ always allow for large elimination.

For inductive propositions $T : \ldots \to \mathbb{P}$ the elimination over $\mathbb{T}$ has to be restricted in order to preserve consistency with the impredicative universe $\mathbb{P}$ [15]. An inductive instance of a proposition can be eliminated over $\mathbb{T}$ (perform large elimination) if the inductive type for the proposition or predicate has at most one proof constructor and every non-parametric argument of the constructor is a proof itself.

Chapter 3

MetaCoq

In this chapter we will discuss the basics of the MetaCoq project[31]. MetaCoq is composed out of multiple projects like TemplateCoq together with the ability to manipulate terms and environments, PCUIC as the specification of Coq's calculus, and a verified implementation and erasure procedure for Coq. Coq is often used to model system like the calculus of constructions or System F. In the MetaCoq project models Coq itself in Coq.

Therefore, one is able to manipulate Coq terms and environments directly in Coq. Additionally, MetaCoq makes it possible to state and prove lemmas over Coq terms and the theoretic foundation of Coq. In [30] a type checker and the erasure phase of extraction were proven correct for example.

When one models structures in Coq and proves lemmas, one often needs basic lemmas and properties for example of inductive types.

For most properties the fundamental structure is known and it is usually easy to derive and prove these lemmas. But it can be very tedious to do this over and over again and be quite exhausting for large inductive types.

One can automate the repetitive work by writing a plugin. These plugins are mainly written in OCaml code as Coq itself is written in this language.

Programming plugins in OCaml comes with some disadvantages. To write a plugin one first has to understand the implementation of Coq which is mostly undocumented, and therefore, hard to understand. It is hard to get the plugin right without the direct testing environment of Coq. A plugin is a large complicated program manipulating the environments. The terms and environments are given in a low level representation. Additionally, one can test the plugin or perform model checking but not formally prove correctness of the plugin.

Other theorem provers come with a tailored meta-programming language [11] for plugin development, and many re-use their internal language as meta-programming

```
Inductive term : 𝕋 :=
    tApp : term → list term → term
  | tRel : ℕ → term
  | tVar : ident → term
  | tEvar : ℕ → list term → term
  | tSort : Universe.t → term
  | tCast : term → cast_kind → term → term
  | tProd : name → term → term → term
  | tLambda : name → term → term → term
  | tLetIn : name → term → term → term → term
  | tConst : kername → Instance.t → term
  | tInd : inductive → Instance.t → term
  | tConstruct : inductive → ℕ → Instance.t → term
  | tCase : inductive × ℕ → term → term → list (nat ×term) → term
  | tProj : projection → term → term
  | tFix : mfixpoint term → ℕ → term
  | tCoFix : mfixpoint term → ℕ → term
```

Figure 3.1: *This inductive type in TemplateCoq represents Coq terms.*

language. This is also the approach used for meta-programming in the MetaCoq project [31]: Meta-programs can be written in Gallina.[1] The meta-programming concept has also been applied to other languages like Haskell with TemplateHaskell [27] and OCaml with [32].

The MetaCoq project includes many sub-projects such as the TemplateCoq project [21, 2] to represent Coq terms in Coq and PCUIC (Polymorphic Calculus of Cumulative Inductive Constructions) for a idealised syntax.

With MetaCoq it is possible to write the meta-functions like normal Coq functions. The transform of TemplateCoq terms needed for plugins is done as Coq functions. The functions are then used to transform Coq terms.

3.1 TemplateCoq and PCUIC

MetaCoq is combines mutliple projects and is built on top of the TemplateCoq project [20]. TemplateCoq introduces types to represent Coq's syntax. OCaml represents Coq's syntax with types. These types are reimplemented with Coq's inductive types (see Figure 3.1) to transform abstract syntax trees (short ASTs) from OCaml's representations of Coq terms to TemplateCoq representations. MetaCoq extends TemplateCoq to the entire calculus of Coq including declaration structures, inductive types and environments.

1 Gallina is the language used in Coq to write terms.

```
tLambda (nNamed "f")
  (tProd nAnon
     (tInd {| inductive_mind := "Coq.Init.Datatypes.nat";
               inductive_ind := 0 |} [])
     (tSort (Universe.from_kernel_repr (Level.lSet, false) [])))
  (tApp (tRel 0)
     [tConstruct {| inductive_mind := "Coq.Init.Datatypes.nat";
                    inductive_ind := 0 |} 0 []])
```

Figure 3.2: *An example of the quoted represenation in TemplateCoq for the term* `λ (f:ℕ→ Set). f 0`. *The* `tInd` *construct represents the type* ℕ. *The representation is obtained with* `MetaCoq Quote quoted := (λ (f: ℕ→ Set). f 0)`

The TemplateCoq implementation is very close to the OCaml implementation of Coq's syntax. It has not completely the same structure as native arrays and strings from OCaml have currently no direct counterpart in Coq. Therefore, Coq's lists and strings are used. Coq 8.12 might change the implementation when persistent arrays are added.

To use TemplateCoq there are vernaculars to convert Coq terms to the TemplateCoq representation and back to Coq terms again (see Figure 3.2). Vernaculars are commands that operate in the OCaml code of Coq. The procedure to move into TemplateCoq is called reification or quoting and the other direction is called unquoting.

As already mentioned there is a second implementation of the syntax with PCUIC. The PCUIC representation is idealised and more suitable in practice. The PCUIC part of MetaCoq contains more theorems and functions as the proofs are easier in the abstract representation.

The vernacular commands of MetaCoq work on the level of TemplateCoq to ensure the correspondence to the original Coq implementation. The correspondence also makes argumentation and reasoning of the guarantees of MetaCoq easier as the relationship to Coq is directly given.

Although the quoting gives TemplateCoq terms, it is possible to reason in PCUIC with a translation procedure from TemplateCoq terms and environments to the syntax of PCUIC (see Figure 3.3). This translation preserves typing and gives therefore a valid representation of the TemplateCoq terms in PCUIC.

As part of this thesis we also implemented the counterpart which is a translation from the idealised PCUIC representation into the TemplateCoq representation en-

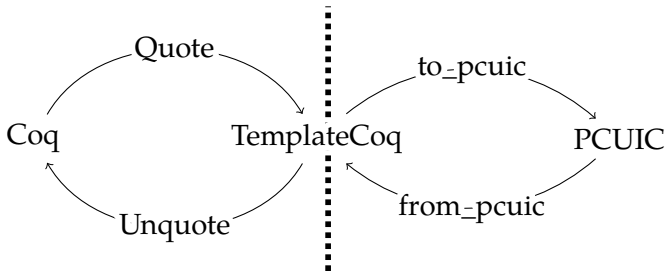


Figure 3.3: *Relations between Coq, TemplateCoq, and PCUIC. The left quoting between Coq and TemplateCoq is done with vernacular commands on the OCaml level whereas the right hand translation is implemented in Coq.*

suring type preservation (see Section A.2)[2].

3.1.1 Typing

MetaCoq further implements inductive types to represent the typing relation on terms which enables the user to prove typing in the abstract context of reified functions written in MetaCoq.

The typing predicate can be used to verify correctness properties of plugins. For the plugin discussed in this thesis the correctness property explained in Chapter 7 states that the derived principles always typecheck with a given type corresponding to the inductive declaration.

So MetaCoq does not only provide a representation of the syntax of Coq objects but also gives semantics to them. This is new as there was no satisfying specification linking Coq's logic and semantics to the underlying type theory foundations before. Some attempts were made such as the proof of [36]. But such proofs over Coq's logic are very large and difficult on paper. The MetaCoq project makes many proofs about the semantics of Coq easier because it allows to reason in Coq using the interactive proof style.

This makes it possible to write a formal specification for Coq's kernel to verify the correctness of its implementation. These proofs are important to be able to trust Coq and therefore be able to reliably use the proof assistant for machine checked proofs. This was done in [30]. MetaCoq first introduced a formal specification of Coq that went beyond inference rules like the ones given in [34, 23].

The typing predicate, written as $\Sigma;\Gamma \vdash \mathtt{t} : \mathtt{T}$, states that in the global environment Σ under the local environment Γ the term `t` has a type `T`. It is mutually defined with the predicate `typing_spine`. `typing_spine` Σ Γ `a xs T` means that a term of type `a` can

2 This does not gives us the complete statement $\mathtt{from\_pcuic}(\mathtt{to\_pcuic}\ \mathtt{t}) = \mathtt{t}$ (see Section 3.5)

```
typing_spine (Σ : global_env_ext) (Γ : context) : term → list term → term → 𝕋
:=
  type_spine_nil : ∀ ty : term. typing_spine Σ Γ ty nil ty
| type_spine_cons : ∀ (hd : term) (tl : list term)
                     (na : name) (A B : term) (s : Universe.t)
                     (T B' : term).
                   Σ; Γ ⊢ tProd na A B : tSort s →
                   Σ; Γ ⊢ T ⩽ tProd na A B →
                   Σ; Γ ⊢ hd : A →
                   typing_spine Σ Γ (B {0 := hd}) tl B' →
                   typing_spine Σ Γ T (hd :: tl) B'
```

Figure 3.4: *The typing spine predicate as part of the typing predicate in TemplateCoq.*

be applied to arguments `xs` and results in a term of type `T`.

For typing we also need substitutions. If we substitute `u` for the variable `x` in a term `t` we write t^x_u. In TemplateCoq the notation `t {n := u}` is used where `n` is the de Bruijn index of `x` in `t`.

Typing spine

The typing spine predicate is implemented using two constructors (Figure 3.4). The base case `typing_spine_nil` is without any arguments, an empty `xs`, where the type `a` of the term is the same as the result type.

The second case `typing_spine_cons` is a bit more complicated: if `xs` has at least one element `hd` which has a type `A`, the applied term should have a type `T` that is convertible to an ∀-quantification (see Section 3.2), a `tProd` in MetaCoq. The argument of this quantification should be the same type as the type of our argument and a type `B` as body. If these conditions are satisfied and the ∀-quantification type is a valid type, the result type can be recursively determined with the remaining argument list and the new starting type `B` where the argument of type `A` is substituted with the first argument `hd`.

This procedure basically substitutes every argument in the corresponding position and checks that each argument is valid for the function type provided. In the base case, without arguments, the type can directly be returned.

The `typing_spine` predicate is not needed in PCUIC as the applications are handled without lists.

The typing predicate has a constructor for each term constructor[3] and an additional constructor for conversion.

3 The term constructors were explained in Section 3.1.2.

Well-formed environments

Most typing constructors assert the environment to be well-formed. This is done with the predicate `wf_local` $\Sigma\ \Gamma =$ `All_local_env` (`lift_typing` `typing` Σ)Γ. The statement means that the type of every assumption in Γ has some universe and for every assumption with a non empty body that the body is typed under the type of the assumption. Therefore, `wf_local` ensures that all assumptions in the environment are well-typed.

3.1.2 Terms and types

We will take a closer look at the term constructors, the syntax, and their typing rules, the semantics. As notation we will use `qX` when we refer to the quoted, i.e. TemplateCoq, representation of the Coq term `X`. If PCUIC differs from TemplateCoq, we will highlight the differences.

The terms use natural numbers to refer to objects in the local environment. This concept is called de Bruijn indexing [8]. The number basically indicates how many binders above the current terms (in the abstract representation of a syntax tree) have to be skipped until the correct identifier is reached. A detailed explanation is found in Section A.1.

For typing rules the inference rule is provided followed by the corresponding constructor of the typing predicate.

Applications

Applications are represented with a body term, to which the arguments are applied, and a list of arguments. This is somewhat unusual as in type theory one commonly sees application as a binary operation taking a body and one single argument returning a term. This representation was chosen in Coq's design to have a direct access to the body and $\mathcal{O}(1)$ access to the arguments with arrays.

Therefore, it can be difficult to work with applications as one cannot perform simple recursion on the inner body but rather has to use list procedures like folding. Additionally, the proofs become harder as better induction principles are needed and one often has to find non trivial generalisations of the statement to be able to prove them.

```
t a1 ... an =                    λ (f:ℕ → ℕ→ ℕ)
((t a1) ...) an                    (g:ℕ).
                                   f 1 g
```

```
tApp t [a1; ...; an]             tLambda (nNamed "f")
                                    (tProd nAnon qNat
                                       (tProd nAnon qNat qNat))
                                    (tLambda (nNamed "g") qNat
                                       (tApp (tRel 1)
                                          [tApp qS [q0]; tRel 0]))
```

An application of `t` applied with arguments `l` has a type `t'` if the application is well-formed, that is `t` is not an application itself and `l` is not empty. Additionally, `t` has to be well-typed with a type `t_ty` and be able to be applied to the arguments `l`. This is handled by the `typing_spine` predicate mentioned in Section 3.1.1.

```
type_App (t : term) (l : list term) (t_ty t' : term):
  Σ; Γ ⊢ t : t_ty →
  isApp t = false →
  l ≠ [] →
  typing_spine Σ Γ t_ty l t' →
  Σ; Γ ⊢ tApp t l : t'
```

Applications are presented in a more intuitive way in PCUIC with unary arguments:

```
tLambda (nNamed "f")
   (tProd nAnon qNat
      (tProd nAnon qNat qNat))
   (tLambda (nNamed "g") qNat
      (tApp
        (tApp (tRel 1) (tApp qS q0))
        (tRel 0))))
```

$$\frac{\Sigma;\Gamma \vdash t : \forall x : A.B \qquad \Sigma;\Gamma \vdash u : A}{\Sigma;\Gamma \vdash (t\ u) : B^x_u}$$

In PCUIC the typing rule is simpler as the constructor is more straightforward:

```
type_App (t : term) (na : name) (A B u : term):
  Σ; Γ ⊢ t : tProd na A B →
  Σ; Γ ⊢ u : A →
  Σ; Γ ⊢ tApp t u : B {0 := u}
```

An application `t u` is well-typed if the term `t` has the type of a quantification ∀na : A, B and the argument `u` has the type `A`. The resulting type is `B` where the argument `na` is substituted with `u`. This way we also avoid the invariants of TemplateCoq while proving lemmas.

References

References refer to variables bound in the term by quantifications, let-in-expressions, or lambda-expressions. The references are constructed with `tRel n` where `n`: ℕ is the de Bruijn index representing the identifier.

Most mappings over terms like substitution and liftings manipulate the indices or replace the whole `tRel n` term.

```
λ (x:ℕ) (f:ℕ→ nat). f x      (tLambda (nNamed "x") qNat
                                (tLambda (nNamed "f")
                                   (tProd nAnon qNat qNat)
                                   (tApp (tRel 0) [tRel 1])))
```

To type a term `tRel n` the environment has to be well-formed and the definition corresponding to `n` has to be found in the local environment Γ. This means that it is the type of the nth binder introduced before. The type is the looked up type lifted to the current position to accommodate for the `n` declarations afterwards. Visually speaking, the environment Γ is $\Gamma', d_n, \ldots d_1, d_0$ and `tRel n` corresponds to d_n. Because the type d_n is typed in the environment Γ′, it has to be lifted over $d_0, \ldots d_n$ to be typed in Γ.

$$\frac{wf\ \Sigma\ \Gamma \qquad (x : T) \in \Gamma}{\Sigma; \Gamma \vdash x : T}$$

```
type_Rel (n : ℕ) (decl : context_decl):
  wf_local Σ Γ →
  nth_error Γ n = Some decl →
  Σ; Γ ⊢ tRel n : lift0 (S n) (decl_type decl)
```

Variables and existentials

The `tVar` and `tEvar` constructors are skipped as we do not use these and both constructs currently have no semantics in MetaCoq.

Sorts

Sorts like ℙ, `Set`, and `Type` are represented using the `tSort u` constructor. Hereby, `u` represents the level of the universe.

$\mathbb{P}$	`tSort (Universe.from_kernel_repr (Level.lℙ, false) [])`
`Set`	`tSort (Universe.from_kernel_repr (Level.lSet, false) [])`
$\mathbb{T} \to \mathbb{T}$	`tProd nAnon (tSort (Universe.from_kernel_repr (Level.Level "Top.65", false) [])) (tSort (Universe.from_kernel_repr (Level.Level "Top.66", false) []))`

One can observe that the universe sort is encoded in the level type.

A sort[4] constructed by `tSort` is typed in a higher sort if the sort level `l` is valid in the global environment Σ.

$$\frac{wf\ \Sigma\ \Gamma}{\Sigma;\Gamma \vdash \mathbb{P} : \mathrm{Type}(1)} \qquad \frac{wf\ \Sigma\ \Gamma}{\Sigma;\Gamma \vdash \mathrm{Type}(i) : \mathrm{Type}(i+1)}$$

```
type_Sort (l : LevelSet.elt):
  wf_local Σ Γ →
  LevelSet.In l (global_ext_levels Σ) →
  Σ; Γ ⊢ tSort (Universe.make l) : tSort (Universe.super l)
```

Casting

The `tCast` constructor takes three arguments: the term, a cast kind, and the expected type. The kind indicates the algorithm used to verify the type conversion in the original Coq source code. The kind is currently ignored in MetaCoq.

`t: A`	`nat : Set`
`tCast t k A`	`tCast qNat Cast qSet`

A type cast is well-typed if the type `t` is well-typed with some universe type `s` and the term `c` has the type `t`. The unification strategy `k` is ignored.

```
type_Cast (c : term) (k : cast_kind) (t : term) (s : Universe.t):
  Σ; Γ ⊢ t : tSort s →
  Σ; Γ ⊢ c : t →
  Σ; Γ ⊢ tCast c k t : t
```

4 This typing rule only applies to non-algebraic universes.

Casts only exist in TemplateCoq. Therefore, PCUIC has no cast term or typing constructor.

Quantification

In MetaCoq ∀-quantifications and implications, also called dependent products, are introduced with the `tProd na t1 t2` constructor. `t1` is the type of the argument/premise and `t2` is the term of the body/conclusion.

An implication $A \to B$ is semantically the same as a ∀-quantification $\forall a : A.B$ where one ignores the name `a` of the argument by not using it in the conclusion `B`. The first argument is a name which can either be unnamed (`nAnon`) for implications or a name (`nNamed id`) where the identifier `id` is a string. Names have no semantic meaning and are only annotations for printing.

```
∀ x: A.  B                    ⊥ → ∀ (P: ℙ). P

tProd x A B                   tProd nAnon q⊥
                                 (tProd (nNamed "P")
                                    qℙ
                                    (tRel 0))
```

A quantification $\forall n : t.b$ is well-typed in the larger[5] universe of `s1` and `s2` if the type `t` is typed in `s1` and the body `b` is well-typed under the environment extended by the assumption named `n` of type `t` with a type in `s2`.

$$\frac{\Sigma;\Gamma \vdash t : s_1 \qquad \Sigma;\Gamma,(x:t) \vdash b : \mathbb{P}}{\Sigma;\Gamma \vdash \forall x : t.b : \mathbb{P}} \qquad \frac{\Sigma;\Gamma \vdash t : \mathrm{Type}(i) \qquad \Sigma;\Gamma,(x:t) \vdash b : \mathrm{Type}(i+1)}{\Sigma;\Gamma \vdash \forall x : t.b : \mathrm{Type}(i+1)}$$

```
type_Prod (n : name) (t b : term) (s1 s2 : Universe.t):
  Σ; Γ ⊢ t : tSort s1 →
  Σ; Γ, vass n t ⊢ b : tSort s2 →
  Σ; Γ ⊢ tProd n t b
    : tSort (Universe.sort_of_product s1 s2)
```

λ-expressions

Abstractions can be constructed with `tLambda na t1 t2`. The structure is basically the same as the one of quantifications: `t1` is the argument type and `t2` is the body term.

5 This is only the case if `s2` does not correspond to ℙ due to impredicativity.

```
λ x:A. b
```

```
λ (X:𝕋) (f:X→ ℙ).
  ∀ (x: X). f x
```

```
tLambda x A b
```

```
tLambda (nNamed "X")
   (tSort (Universe.from_kernel_repr
             (Level.Level "Top.244", false) []))
   (tLambda (nNamed "f")
      (tProd nAnon (tRel 0) qℙ)
      (tProd (nNamed "x") (tRel 1)
        (tApp (tRel 1) [tRel 0]))))
```

An abstraction `λ (n:t). b` is typed with the corresponding type `∀ (n: t). bty` if the argument type `t` is a valid type and the body `b` is typed with `bty` under the environment extended by the assumption `n: t`.

$$\frac{\Sigma;\Gamma \vdash t : s \qquad \Sigma;\Gamma, n : t \vdash b : bty}{\Sigma;\Gamma \vdash \lambda n : t.b : \forall n : t.bty}$$

```
type_Lambda (n : name) (t b : term)
      (s1 : Universe.t) (bty : term):
  Σ; Γ ⊢ t : tSort s1 →
  Σ; Γ, vass n t ⊢ b : bty →
  Σ; Γ ⊢ tLambda n t b : tProd n t bty
```

Let-in expressions

Let-in expressions take a name, a term for the argument, the type of the argument, and lastly a term for the body.

```
let x := t : A in b
```

```
let x := 0 : ℕ in
  x
```

```
tLetIn x t A b
```

```
tLetIn (nNamed "x") q0 qNat
   (tRel 0)
```

The typing of a let-in expressions `let n := b : b_ty in b'` is nearly the same as for λ-abstractions: the type `b_ty` has to be well-typed, the let-in body `b` has to be typed with `b_ty`, and the inner body `b'` has to have a type `b'_ty`. If the constraints are satisfied the let-in expression has the type `let n := b : b_ty in b'_ty`.

$$\frac{\Sigma;\Gamma \vdash b_{ty} : s \qquad \Sigma;\Gamma \vdash b : b_{ty} \qquad \Sigma;\Gamma, (x := b : b_{ty}) \vdash b' : b'_{ty}}{\Sigma;\Gamma \vdash \text{let } n := b : b_{ty} \text{ in } b' \ : \ \text{let } n := b : b_{ty} \text{ in } b'_{ty}}$$

```
type_LetIn (n : name) (b b_ty b' : term)
       (s1 : Universe.t) (b'_ty : term):
  Σ; Γ ⊢ b_ty : tSort s1 →
  Σ; Γ ⊢ b : b_ty →
  Σ; Γ, vdef n b b_ty ⊢ b' : b'_ty →
  Σ; Γ ⊢ tLetIn n b b_ty b' : tLetIn n b b_ty b'_ty
```

Constants

The `tConst k l` constructor for constants takes a kernel name of the constant and a list of universes to account for the possibility of universe polymorphism. For details on universes and universe polymorphism see [35].

`plus`	`tConst "Coq.Init.Nat.add" []`
0+0	`tApp (tConst "Coq.Init.Nat.add" [])` `  [q0; q0]`

Constants `tConst cst u`, definition, lemmas and axioms in Coq, are well-typed if they refer to a global declaration `decl`, determined by `declared_constant`. The predicate `declared_constant` executed the `lookup_env` function and expects as result a constant `Some (ConstantDecl cst decl)`. Lastly, it is checked that the universe constraints are satisfied and the universe instance is substituted for the universes in the type of the declaration.

```
type_Const (cst : ident) (u : Instance.t)
    (decl : constant_body):
  wf_local Σ Γ →
  declared_constant Σ.1 cst decl →
  consistent_instance_ext Σ (cst_universes decl) u →
  Σ; Γ ⊢ tConst cst u
    : subst_instance_constr u (cst_type decl)
```

Inductive types

Inductive types are formed with the `tInd ind u` constructor where `ind: inductive` identifies the inductive type. Like constants, inductive types also have an argument for universes. The type `inductive` is a record containing the kernel name `inductive_mind` and an index for mutual inductive declarations `inductive_ind`.

```
nat                    tInd {| inductive_mind :=
                                   "Coq.Init.Datatypes.nat";
                               inductive_ind := 0 |} []

⊥                      tInd {| inductive_mind :=
                                   "Coq.Init.Logic.⊥";
                               inductive_ind := 0 |} []

odd                    tInd {| inductive_mind :=
                                   "Coq.Arith.Even.even";
                               inductive_ind := 1 |} []
```

Similar to constants, the inductive type is looked up in the global context and checked with `declared_inductive`. Hereby, `mdecl` is the mutual inductive declaration and `idecl` is the inductive body. Lastly, the universes are again checked and substituted.

```
type_Ind (ind : inductive) (u : Instance.t)
      (mdecl : mutual_inductive_body) (idecl : one_inductive_body):
  wf_local Σ Γ →
  declared_inductive Σ.1 mdecl ind idecl →
  consistent_instance_ext Σ (ind_universes mdecl) u →
  Σ; Γ ⊢ tInd ind u
    : subst_instance_constr u (ind_type idecl)
```

Constructors

The constructor terms take the same arguments as the inductive type and an additional number `n` stating which constructor should be used. They are introduced with `tConstruct ind n u`.

```
0                      tConstruct
                          {| inductive_mind :=
                               "Coq.Init.Datatypes.nat";
                             inductive_ind := 0 |} 0 []

S                      tConstruct
                          {| inductive_mind :=
                               "Coq.Init.Datatypes.nat";
                             inductive_ind := 0 |} 1 []
```

To type a constructor the inductive type has to be defined in the global environment and a constructor corresponding to the number `i` has to be found. This is done

with `declared_constructor` which itself is a conjunction of `declared_inductive` and a constructor lookup with `nth_error`. The type is then constructed by substituting the inductive type into the type of the constructor. This is necessary as constructors refer to the inductive type with `tRel 0`.

```
type_Construct (ind : inductive) (i : ℕ) (u : Instance.t)
    (mdecl : TemplateEnvironment.mutual_inductive_body)
    (idecl : TemplateEnvironment.one_inductive_body)
    (cdecl : (ident × term) × ℕ):
  wf_local Σ Γ →
  declared_constructor Σ.1 mdecl idecl (ind, i) cdecl →
  consistent_instance_ext Σ (ind_universes mdecl) u →
  Σ; Γ ⊢ tConstruct ind i u
   : type_of_constructor mdecl cdecl (ind, i) u
```

Matches

Pattern matching is introduced with `tCase (ind,npar) p c brs`.

Here, `ind`: `inductive` is the inductive type, `npars`:ℕ the number of parameter, `p`: `term` the return type which takes all indices and the instance of type `ind` with λ-abstractions, `c`: `term` is the object over which the match is performed and lastly `brs`: `list (ℕ×term)` are the branches. Each branch is a pair with the number of arguments and the term used in the match.

```
λ (P:ℕ→ ℙ)
    (H0: P 0)
    (HS: ∀ n, P (S n))
    (n:ℕ).
    match n as m
      return P m with
    | 0 ⇒ H0
    | S k ⇒ HS k
    end
```

```
... (* the lambda abstractions *)
tCase
 (* match type *)
 ({| inductive_mind := "Coq.Init.Datatypes.nat";
  inductive_ind := 0 |}, 0)

 (* return type *)
 (tLambda (nNamed "m") qNat
     (tApp (tRel 4) [tRel 0]))

 (* match object *)
 (tRel 0)

 [
  (* 0 case without arguments *)
  (0, tRel 2);
  (* S case with one argument *)
  (1, tLambda (nNamed "k") qNat
   (tApp (tRel 2) [tRel 0]))
  ]
```

```
type_Case (indnpar : inductive × ℕ)
          (u : Instance.t) (p c : term) (brs : list (nat × term))
          (args : list term)
          (mdecl : TemplateEnvironment.mutual_inductive_body)
          (idecl : TemplateEnvironment.one_inductive_body)
          (ps : Universe.t) (pty : term)
          (btys : list (nat × term)):
let ind := indnpar.1 in
let npar := indnpar.2 in
declared_inductive Σ.1 mdecl ind idecl →
ind_npars mdecl = npar →
Σ; Γ ⊢ c : mkApps (tInd ind u) args →
let params := firstn npar args in
build_case_predicate_type ind mdecl idecl params u ps = Some pty →
Σ; Γ ⊢ p : pty →
leb_sort_family (universe_family ps) (ind_kelim idecl) →
  map_option_out
    (build_branches_type ind mdecl idecl params u p) = Some btys →
  All2
    (λ br bty : ℕ × term.
     (br.1 = bty.1 × Σ; Γ ⊢ br.2 : bty.2)
     × Σ; Γ ⊢ bty.2 : tSort ps) brs btys →
  Σ; Γ ⊢ tCase indnpar p c brs
    : mkApps p (skipn npar args ++ [c])
```

Pattern matching with `tCase` is the most complicated typing rule. The typing constructor is only given for completeness.

First the inductive type and the parameter count is extracted from the first argument `indnpar`. It is then checked that the inductive type is declared in the environment with a mutual inductive declaration `mdecl` and an inductive body `idcel`, and that the number of parameters of `mdecl` correspond to the count `npar` given by the `tCase` constructor arguments.

The arguments `args` are used to verify that the match object `c` is an instance of the inductive type fully applied with parameters, the first `npar` arguments of `args`, and indices.

The return type is checked to type with a type `pty` computated by the function `build_case_predicate_type`. This function ensures that the return type takes all indices and the match instance and returns terms in the `ps` universe space. This universe is checked to be a valid elimination with `leb_sort_family`. Therefore, large elimination is only possible if this not prohibited by the type.

Similar to the type of the return type a list of types `btys` for the branches is calculated using the `build_branches_type` function. The types are checked with the

`All2` predicate that performs a pointwise check that every branch types with the corresponding branch type and the branch type is a valid type.

If all requirements are met, the case construct types with the return type `p` applied to the arguments `args` and match object `c`.

Projections

Primitive projections are introduced when a type is declared with `Record`.

Records can be declared with a declaration `Record T := con { p1 : t1; p2: t2 }` where the record is called `T`, the constructor to build is `con` and it has two fields, `p1` of type `t1` and `p2` of type `t2`. These arguments can be dependent. Therefore, `t2` can mention `p1`.

The record `T` is basically the same as if it was declared with an inductive type `Inductive T := con (p1 : t1) (p2 : t2)`. Hence, a record is semantically an inductive type with one constructor without indices. The only difference is that Coq automatically defines the projections for the arguments. These projections are functions that extract an argument from an instance of the record type.

The MetaCoq projections refer to primitive projections [34] Coq declares for record types when the `Set Primitive Projections` option is set. Primitive projections work the same way as normal projection functions. In Coq they are written as `t.( p )` or `p t` where `t` is an instance of the record and `p` is the projection.

One can, for example, define rational numbers[6] with a record.

```
Record Rat : Set := mkRat {
  top : ℕ;
  bottom : ℕ;
  Rat_cond : bottom > 0 }.
```

The number $\frac{1}{2}$ is then represented as follows:

```
Definition one_half : Rat.
  refine (mkRat 1 2 _).
  repeat constructor.
Defined.
```

In MetaCoq projections are used with the `tProj p t` constructor where `p`: `projection` and `t`: `term`. The type `projection` is a tuple of `inductive` and two natural numbers representing the number of parameters of the record and the index of the argument that is projected.

6 This representation does not admit a unique representation for numbers.

```
one_half.(bottom)        tProj ({| inductive_mind := "Top.Rat";
                                   inductive_ind := 0 |}, 0, 1)
                           (tConst "Top.one_half" [])
```

First the projection p: `inductive` $\times \mathbb{N} \times \mathbb{N}$ and the inductive declaration is looked up in the global environment with `declared_projection`. Then the instance `c` is checked to be a correctly applied instance of the inductive type where all parameters are instantiated with `args`, ensured by `#| args | = ind_npars mdecl`. To determine the resulting type, the arguments, `c`, and the universe are substituted for the local references in the record type `ty`.

```
type_Proj (p : projection) (c : term)
    (u : Instance.t)
    (mdecl : TemplateEnvironment.mutual_inductive_body)
    (idecl : TemplateEnvironment.one_inductive_body)
    (pdecl : ident × term)
    (args : list term):
  declared_projection Σ.1 mdecl idecl p pdecl →
  Σ; Γ ⊢ c : mkApps (tInd p.1.1 u) args →
  #| args | = ind_npars mdecl →
  let ty := pdecl.2 in
  Σ; Γ ⊢ tProj p c
    : subst0 (c :: rev args) (subst_instance_constr u ty)
```

Fixed points

Fixed points allow for structural recursive function declarations in Coq and so form a basic principle needed, for example, to perform induction. One usually writes fixpoint definitions with the vernacular command `Fixpoint` but it is also possible to write terms involving inlined fixed points. To allow for mutual recursion it is possible to intertwine multiple fixed-point functions.

```
fix f1 (x11:X11) ... (xn1:Xn1) {struct xk1}: A1 := t1 with
... with
fix fm (x1m:X1m) ... (xnm:Xnm) {struct xkm}: Am := tm
for fi
```

Each function f_l in the mutual fixpoint takes arguments x_{il} of some type X_{il}, has an argument that has to decrease in every recursive call x_{kl} and returns something of type A_l in the environment with all fixpoint declarations.

In MetaCoq the fixed points are represented using `tFix mfix i` with `mfix:mfixpoint` and `i`:$\mathbb{N}$ as the index of the fixpoint function used, the f_i in the example above. The

type `mfixpoint` is `list (def term)` where `def` is a record with a type as parameter. The `def` record has four fields:

- `dname` is the name of the definition, in this case the name of the fixpoint,
- `dtype` is the type, A_i in the example above,
- `dbody` is the body, t_i, with λ-abstractions for the arguments,
- and lastly `rarg` is the index of the recursive argument.

Coq:

```
fix add (n m:ℕ) {struct m} : ℕ :=
match m with
| O ⇒ n
| S m' ⇒ S (add n m')
end
```

TemplateCoq[7]:

```
tFix [{|
  dname := nNamed "add";
  dtype := tProd (nNamed "n") qNat
             (tProd (nNamed "m") qNat qNat);
  dbody := tLambda (nNamed "n") qNat
             (tLambda (nNamed "m") qNat
                (tCase (qNat',0 )
                   (tLambda (nNamed "m") qNat qNat)
                   (tRel 0)
                   [(0, tRel 1);
                   (1, tLambda (nNamed "m'") qNat
                     (tApp (tConstruct qNat' 1 [])
                        [tApp (tRel 3) [tRel 2; tRel 0]])])]));
  rarg := 1 |}] 0
```

The fixed point should be structurally recursive, stated by `fix_guard`, but this condition is currently not implemented. Next, the selected function has to be in the mutual fixpoint construction, checked with `nth_error`. Afterwards, the type context with all fixpoints and assumptions is generated and checked for well-formedness. The main statement checks that the body of every fixpoint is a function and has a type `dtype d` that is valid in the environment Γ extended with the fixpoint declarations `types`.

```
type_Fix (mfix : mfixpoint term) (n : ℕ) (decl : def term):
   fix_guard mfix →
   nth_error mfix n = Some decl →
```

7 `qNat'` is the inductive part (without universe list) of `qNat`

```
let types := fix_context mfix in
wf_local Σ (Γ, types) →
All
  (λ d : def term.
   Σ; Γ, types ⊢ dbody d : lift0 #|types| (dtype d)
   × isLambda (dbody d) = true) mfix →
Σ; Γ ⊢ tFix mfix n : dtype decl
```

Co-fixed points

Co-fixed points are defined analogously to fixed points with the `tCoFix` constructor taking the same arguments. Co-fixed points are typed nearly the same as fixed points except that the `fix_guard` is left out. The productivity criterion is currently not checked.

Conversion

To prove that a term `t` has a type `B` one can always use conversion (see Section 3.2) to transform the type `B` into a convertible type `A` and prove that `t` has type `A` instead. Additionally, `B` has to be a valid type in the environment and needs to be well-formed.

```
type_Conv t A B :
  Σ; Γ ⊢ t : A →
  isWfArity typing Σ Γ B +
  (Σ s : Universe.t, Σ; Γ ⊢ B : tSort s) →
  Σ; Γ ⊢ A ⩽ B →
  Σ; Γ ⊢ t : B
```

3.2 Conversion

Two terms `t` and `u` are convertible if the `cumul` predicate, written as written as $\Sigma; \Gamma \vdash t \leqslant u$, holds both ways. The relation `cumul Σ Γ t u` holds if the terms are the same under the `leq_term` predicate or one of the terms can be reduced to a term `v` using the `red1` predicate, and this term is checked recursively with the other term. The `leq_term` predicate checks that two terms are congruent up to universes. If two terms are convertible, they can be reduced to α-convertible terms.

The one step reduction `red1` has constructors for congruence rules, that is reduction in subterms, for each constructor. Quantification, for example, has two rules:

```
prod_red_l (na : name) (M1 M2 N1 : term):
  red1 Σ Γ M1 N1 →
  red1 Σ Γ (tProd na M1 M2) (tProd na N1 M2)
prod_red_r (na : name) (M2 N2 M1 : term):
  red1 Σ (Γ, vass na M1) M2 N2 →
  red1 Σ Γ (tProd na M1 M2) (tProd na M1 N2)
```

The first rule allows for reduction in the argument, and the second rule allows to perform reduction under the quantification.

The remaining rules are the usual conversion rules known from Coq's type theory.

β-reduction

```
red_beta (na : name) (t b a : term) (l : list term):
   red1 Σ Γ (tApp (tLambda na t b) (a :: l))
     (mkApps (b {0 := a}) l)
```

β-reduction is the evaluation of an application to a function expression, an expression with a λ-abstraction at its head. The first applied argument is consumed and substituted for the argument `na`.

$(\lambda(x : t).b)a \rightsquigarrow b^x_a$

ζ-reduction

```
red_zeta (na : name) (b t b' : term):
  red1 Σ Γ (tLetIn na b t b') (b' {0 := b})
```

The reduction of let-in expressions is called ζ reduction and allows to insert the body `b` for `na` in `b'`.[8]

$\texttt{let x := b : t in b'} \rightsquigarrow \texttt{b'}^{\texttt{x}}_{\texttt{b}}$

Unfold

```
red_rel (i : ℕ) (body : term):
  option_map decl_body (nth_error Γ i) = Some (Some body) →
  red1 Σ Γ (tRel i) (lift0 (S i) body)
```

The unfold reduction allows to lookup a relation in the local environment and use the body instead of the relation if one is found. This corresponds to ζ-reduction when the let-in expression was already introduced in the local environment Γ. The lifting is necessary to compensate for the new assumptions added to Γ after the let-in.

ι-reduction

```
red_iota (ind : inductive) (pars c : ℕ)
     (u : Instance.t) (args : list term)
     (p : term) (brs : list (nat × term)):
  red1 Σ Γ
    (tCase (ind, pars) p (mkApps (tConstruct ind c u) args) brs)
    (iota_red pars c args brs)
```

8 This coincides with the understanding that `let x := b : t in b'` is semantically the same as `( λ x:t. b' ) b`.

```
iota_red (npar c : ℕ) (args : list term) (brs : list (nat × term)) =
  mkApps (nth c brs (0, tDummy)).2 (skipn npar args)
```

When a match is performed over a constructor of the inductive type, ι-reduction can be used to select the corresponding branch. This selection is done with the `iota_red` function that chooses the corresponding branch and forwards the arguments without parameters.

```
match C a b with
| ... ⇒ ...
| C x y ⇒ H x y    ⇝  H a b
| ... ⇒ ...
end
```

Fixed point unfolding

```
red_fix (mfix : mfixpoint term) (idx : ℕ)
    (args : list term) (narg : ℕ) (fn : term):
  unfold_fix mfix idx = Some (narg, fn) →
  is_constructor narg args = true →
  red1 Σ Γ (tApp (tFix mfix idx) args) (tApp fn args)
```

Fixed point functions also can be unfolded by reduction like λ-abstractions. However, there is an additional constraint that the recursive argument in position `narg` has to be a constructor to prevent infinite unfolding.

Co-fixed point unfolding

```
red_cofix_case (ip : inductive × ℕ)
       (p : term) (mfix : mfixpoint term)
       (idx : ℕ) (args : list term)
       (narg : ℕ) (fn : term) (brs : list (nat × term)),
   unfold_cofix mfix idx = Some (narg, fn) →
   red1 Σ Γ
     (tCase ip p (mkApps (tCoFix mfix idx) args) brs)
     (tCase ip p (mkApps fn args) brs)

red_cofix_proj (p : projection) (mfix : mfixpoint term)
       (idx : ℕ) (args : list term)
       (narg : ℕ) (fn : term),
   unfold_cofix mfix idx = Some (narg, fn) →
   red1 Σ Γ (tProj p (mkApps (tCoFix mfix idx) args))
     (tProj p (mkApps fn args))
```

Co-fixed points can be unfolded if they appear in a match or projection. These cases

are handled by `red_cofix_case` and `red_cofix_proj`.

δ-reduction

```
red_delta (c : ident) (decl : TemplateEnvironment.constant_body)
          (body : term) (u: Instance.t):
  declared_constant Σ c decl →
  cst_body decl = Some body →
  red1 Σ Γ (tConst c u) (subst_instance_constr u body)
```

δ-reduction is unfolding of constants. A constant can be looked up in the environment and substituted for the body with instantiated universes.

Projection unfolding

```
red_proj (i : inductive) (pars narg : ℕ)
         (args : list term) (k : ℕ) (u : Instance.t)
         (arg : term):
  nth_error args (pars + narg) = Some arg →
  red1 Σ Γ
    (tProj (i, pars, narg) (mkApps (tConstruct i k u) args)) arg
```

A projection of an applied constructor can be unfolded to the corresponding argument.

```
(C a b).(first)  ⇝  a
```

Conversion in PCUIC

The `cumul` predicate in PCUIC has two additional constructors for η-conversion. These constructors allow to perform η-expansion on both terms. The η-expansion of a term `t: A → B` is `λ (x:A). t x`.

3.3 Inductive types

Every inductive declaration is represented by a `mutual_inductive_body` construct (Figure 3.5). This construct represents the whole mutual inductive definition with the individual inductive types as a list in `ind_bodies`.

As the parameters are constant across all inductive types in the mutual inductive declaration they are stored in the field `ind_params` of `mutual_inductive_body` in reverse order and their number is stored in `ind_npars` (see Figure 3.6). The parameters have to be uniform in the conclusion of every constructor but can vary in the arguments mentioning one of the types. In the case that they are the same in every instance of the type they are called uniform parameters and non uniform otherwise, like the last parameter called `x` in Figure 3.6[9].

9 There is currently no indication which parameters are uniform. Therefore, we implemented a syntactic check to determine this property which we need later on.

```
Record mutual_inductive_body : 𝕋 := Build_mutual_inductive_body
  { ind_finite : recursivity_kind;
    ind_npars : ℕ;
    ind_params : context;
    ind_bodies : list one_inductive_body;
    ind_universes : universes_decl;
    ind_variance : option (list Variance.t) }

Record one_inductive_body : 𝕋 := Build_one_inductive_body
  { ind_name : ident;
    ind_type : term;
    ind_kelim : sort_family;
    ind_ctors : list ((ident × term) × ℕ);
    ind_projs : list (ident × term) }
```

Figure 3.5: *Mutual and one inductive body declarations from MetaCoq.*

Other than that, the `mutual_inductive_body` also has a field for the universe constraints `ind_universes` and `ind_variance` to handle variances for polymorphic cumulative inductive types.

Every single inductive type is represented by an object of type `one_inductive_body`. It has a name in `ind_name` and a type `ind_type` which quantifies all parameters and indices. The possible elimination is stored in `ind_kelim` and all constructors are represented in `ind_ctors`. `ind_ctors` is a list with the name, type and number of arguments. Lastly there is the field `ind_projs` which contains a list of possible projections (see Section 3.1.2) of the inductive type.

3.4 Execute MetaCoq commands

Vernacular commands are instructions on the toplevel of Coq that invoke OCaml code. Some examples are `Print`, `Definition`, `Fixpoint`, or `Fail`. MetaCoq implements a set of vernacular commands and a set of monadic functions which allow for easy manipulation of the environments of Coq (Figure 3.7). These commands allow to implement general plugins in Coq as plugins are transformations on the syntax and environments of Coq terms. A program consists of individual commands chained with monadic operations to build a complex algorithm. The commands are monadic operations, reification commands to quote and unquote objects, and useful general purpose commands.

The monadic aspect of the `TemplateMonad` programs mean that it is possible to combine the commands into larger programs. To do so one can use the `tmBind` constructor with a convenient notation common for monadic programs: if one wants to execute a command `A` and then another command `B` using the result `a` of `A`, one

```
Inductive Acc (A : 𝕋) (R: A → A → ℙ)(x : A) : ℙ :=
    Acc_I : (∀ y : A, R y x → Acc R y) → Acc R x

{|
ind_finite := Finite;
ind_npars := 3;
ind_params := [{| decl_name := nNamed "x";
                  decl_body := None;
                  decl_type := tRel 1 |};
                {| decl_name := nNamed "R";
                  decl_body := None;
                  decl_type := tProd nAnon (tRel 0)
                      (tProd nAnon (tRel 1) qℙ) |};
                {| decl_name := nNamed "A";
                  decl_body := None;
                  decl_type := qType |}];
ind_bodies := [{|
               ind_name := "Acc";
               ind_type := tProd (nNamed "A") qType
                             (tProd (nNamed "R")
                                (tProd nAnon (tRel 0)
                                   (tProd nAnon (tRel 1) qℙ))
                                (tProd (nNamed "x") (tRel 1) qℙ));
               ind_kelim := InType;
               ind_ctors := [
    ("Acc_I",
    tProd (nNamed "A") qType
      (tProd (nNamed "R")
         (tProd nAnon (tRel 0)
            (tProd nAnon (tRel 1) qℙ))
         (tProd (nNamed "x")
            (tRel 1)
            (tProd nAnon
               (tProd (nNamed "y")
                  (tRel 2)
                  (tProd nAnon
                     (tApp (tRel 2) [tRel 0; tRel 1])
                     (tApp (tRel 5) [tRel 4; tRel 3; tRel 1])))
               (tApp (tRel 4) [tRel 3; tRel 2; tRel 1])))),
    1)];
               ind_projs := [] |}];
ind_universes := Monomorphic_ctx (of_list [], ConstraintSet.empty);
ind_variance := None |}
```

Figure 3.6: *The type* `Acc` *and its quoted TemplateCoq representation below.*

can simply write `a ← A; B` or in a short notation `A >>=( λ a. B )`. This notation is short for `tmBind A ( λ a. B )`. The other monadic constructor is `tmReturn` which takes any object and transports it into the `TemplateMonad` space.

A program can then be executed with the `MetaCoq Run` vernacular functioning as an interpreter for programs using the `TemplateMonad`. These commands and the vernacular are implemented in OCaml as a traditional Coq plugin where the underlying functions are mostly simple transformations from the OCaml types to TemplateCoq. Hereby, one usually executes programs of the type `TemplateMonad unit` and uses the side effects of the commands. Side effects are modification to the environment of Coq and include the addition of declarations and definitions or printing messages to the output.

Most of the `TemplateMonad` commands also have a vernacular command associated with them:

command[10]	**Vernacular**	**meaning**
tmQuote t	MetaCoq Quote	Returns the representation of t in TemplateCoq
tmQuoteRec t	MetaCoq Quote	Returns the representation of t and all declarations needed for t
tmQuoteInductive kn	MetaCoq Quote	Returns the declaration of the inductive type kn
tmQuoteUniverses		Returns a quoted representation of the universes in the current environment
tmQuoteConstant kn b	MetaCoq Quote	Returns the declaration of the constant kn. The boolean b indicates whether the opacity should be ignored to get the body
tmMkInductive d	MetaCoq Unquote	Declares an inductive represented by d
tmUnquote tm	MetaCoq Unquote	Returns a type A and a term with the type A and syntax tm
tmUnquoteTyped A tm	MetaCoq Unquote	Returns a term with the syntax tm and checks the type to be A
tmPrint x	Print	Prints an object x
tmMsg msg	Print	Prints a string message

tmFail msg	Fail	Fails with an error message msg
tmEval red t	Eval	Evaluates t with the strategy red [11]
tmLemma id A	Lemma	Creates an obligation of type A for the user and returns the constant id afterwards
tmDefinitionRed_ b ident red x	Definition	Generates a definition named ident with the body x. Before the definition is created the reduction strategy red is performed (if one is given). The boolean b determines whether the definition should be transparent (false) or opaque (true).
tmAxiomRed ident red X	Axiom	Reduces X with red and creates an axiom named ident
tmFreshName ident		Generates a fresh name from ident which does not occur in the environment. Hereby ident already has to be in the correct format of identifiers.
tmAbout id	About	If id is a constant in the environment a global reference is returned
tmCurrentModPath ()		Returns the current prefix of modules. Therefore, this command in a module A would return "Top.A".
tmExistingInstance ident	Existing Instance	Adds the object identified by ident to the type class.
tmInferInstance red X		Performs reduction with red and tries to infer an instance of type X.

As the quoting function is used very often there are some other variants to call

10 See Figure 3.7 for types.

11 Possible strategies are cbn, cbv, hnf, all, unfold or lazy

```
Inductive TemplateMonad@{t u} : 𝕋 → ℙ:=
  (* monadic commands *)
    tmReturn : ∀ A : 𝕋. A → TemplateMonad A
  | tmBind : ∀ A B : 𝕋.
             TemplateMonad A → (A → TemplateMonad B ) → TemplateMonad B

  (* Reification Commands *)
  | tmQuote : ∀ A : 𝕋. A → TemplateMonad Ast.term
  | tmQuoteRec : ∀ A : 𝕋. A → TemplateMonad Ast.program
  | tmQuoteInductive : qualid → TemplateMonad Ast.mutual_inductive_body
  | tmQuoteUniverses : TemplateMonad ConstraintSet.t
  | tmQuoteConstant : qualid → bool → TemplateMonad Ast.constant_body
  | tmMkInductive : Ast.mutual_inductive_entry → TemplateMonad unit
  | tmUnquote : Ast.term → TemplateMonad typed_term
  | tmUnquoteTyped : ∀ A : 𝕋. Ast.term → TemplateMonad A

  (* informational commands *)
  | tmPrint : ∀ A : 𝕋. A → TemplateMonad unit
  | tmMsg : string → TemplateMonad unit
  | tmFail : ∀ A : 𝕋. string → TemplateMonad A
  | tmEval : reductionStrategy → ∀A : 𝕋. A → TemplateMonad A
  | tmLemma : ident → ∀A : 𝕋. TemplateMonad A
  | tmDefinitionRed_ : bool → ident → option reductionStrategy →
                       ∀ A : 𝕋. A → TemplateMonad A
  | tmAxiomRed : ident → option reductionStrategy →
                       ∀ A : 𝕋. TemplateMonad A
  | tmFreshName : ident → TemplateMonad ident
  | tmAbout : qualid → TemplateMonad (option global_reference )
  | tmCurrentModPath : unit → TemplateMonad string
  | tmExistingInstance : qualid → TemplateMonad unit
  | tmInferInstance : option reductionStrategy →
                      ∀ A : 𝕋. TemplateMonad (option A )
```

Figure 3.7: *The inductive types containing all MetaCoq commands that can be chained in a monadic fashion.*

it. It is possible to write <% X %> to quote X inline. Additionally, there is the quote_term t (λ x. T) tactic which quotes the object t and then performs the tactic T.

Supplementary to the commands mentioned in Figure 3.7 the MetaCoq project also has other commands built on top of the basic TemplateMonad ones for easier use. An example is the tmDefinition command that creates a transparent definition without reduction and therefore simply calls tmDefinitionRed_ with the first and third argument already specified. Another example is tmMkDefinition which takes a term, quotes it, evaluates the result, and then creates a definition.

Let us look at two small mini-plugins to get familiar with the TemplateMonad programs.

Contraposition

The first plugin takes an implication P → Q or a definition of an implication and adds the contraposition ¬ Q → ¬P as an axiom to the environment.

We start with the main part of our mini-plugin:

1. check if the argument is an implication,
2. extract the assumption and conclusion,
3. unquote both,
4. construct the contraposition,
5. add the axiom.

Theoretically the steps three and four could be exchanged so that the contraposition is constructed in the quoted representation and then is unquoted. We choose this order to have a better understandable function and better failure messages if the user inputs an implication where one side is not a proposition. We explicitly require propositions to use the propositional not function of Coq.[12]

```
Definition computeContrapos (t:term) (name: ident) : TemplateMonad unit :=
  match t with
  | tProd nAnon t1 t2 ⇒
    na ← tmFreshName name;
    q1 ← tmUnquoteTyped ℙ t1;
    q2 ← tmUnquoteTyped ℙ t2;
    let q := ¬q2 → ¬q1 in
    (
      tmAxiomRed na None q;
      tmPrint q;
```

12 It would easily be possible to generalize this function to arbitrary implications in Type.

```
      tmMsg (append "was added to the environment as axiom " na)
    )
| tProd _ _ _ ⇒ tmFail "A non dependent implication is expected"
| _ ⇒ tmFail "Invalid argument: Implication expected."
end.
```

We first match the term to check whether it is an implication, that is `tProd` where the assumption is unnamed (`nAnon`), in other cases the function fails with an informative message. In the `tProd nAnon` case we generate a fresh name for the axiom from the given name and unquote the assumption and conclusion with $\mathbb{P}$ as expected type. When we have both propositions as Coq term we can construct the contraposition `q`. Lastly, the axiom is constructed with `tmAxiomRed` and a success message is displayed.

Because it is inconvenient to give the implication explicitly, we extend our plugin to recognize if a definition is given and call the `computeContrapos` function accordingly.

```
Definition addContrapos (H: ℙ) (name: ident) : TemplateMonad unit :=
  p ← tmQuote H;
  match p with
  | tConst qual _ ⇒
    q ← tmQuoteConstant qual false;
    match Ast.cst_body q with
    | Some t ⇒ computeContrapos t name
    | None ⇒ tmFail "a constant with an empty body is not a valid argument"
    end
  | _ ⇒ computeContrapos p name
  end.
```

The first step of the main function is to quote the argument. Next, we compare the argument and unfold the constant in case of a definition. This is simply done with `tmQuoteConstant`.

The plugin then can be called with `MetaCoq Run (addContrapos (⊥ → ⊤)"Contra")`, or `MetaCoq Run (addContrapos H "Contra")` if `H` is a definition containing the implication, creating the axiom `Contra : ¬ ⊤ → ¬⊥`.

Constructor list

As inductive types are a large focus in later chapters we also take a look at another mini-plugin creating a list with the types of the constructors of an inductive type. For example the plugin should return `[nat;nat → ℕ]` for the type `nat`.

To do this we have to get the abstract representation of the type, extract the constructors, and then unquote all constructor types and write them in a list.

We first quote the type with `tmQuote` to extract the kernel name and quote the inductive type with `tmQuoteInductive`. Afterwards, we can inspect the inductive body and operate on the constructor list `ind_ctors`. The details of the implementation are found in Section A.4.

The plugin can be executed with `MetaCoq Run (getCtors or)` for the type `or`, and `MetaCoq Run (getCtors sig)` for the type `sig`. The results, with the addition of the constructors, are:

```
or_ctors =
  [(∀ A B : ℙ, A → A ∨ B; or_introl);
  (∀ A B : ℙ, B → A ∨ B; or_intror)]

sig_ctors =
  [(∀ (A : 𝕋) (P : A → ℙ)(x : A),
    P x → {x : A | P x}; exist)]
```

Interactive commands

Two commands useful for interactive plugins are the `tmLemma` and `tmInferInstance` commands.

With `tmInferInstance` it is possible to automatically fill holes by inference of missing instances of types.

In Coq one can declare objects as an instance of a class and these instances then can be automatically inferred in definitions and proofs.

```
Existing Class ⊤.
Existing Instance I.

(* for definitions *)
Definition inferTest : ⊤ := _.

(* in a proof *)
Goal ⊤.
Proof.
  refine (_).
Qed.
```

The same concept is available in MetaCoq with the TemplateMonad commands:

```
MetaCoq Run (tmExistingInstance "I").
MetaCoq Run (tmInferInstance None ⊤ >>=tmPrint).
```

The second command prints `Some I` as result which means that the instance `I` of type ⊤ was found.

The lemma command `tmLemma id T` asks the user for an object of type `T` and creates an object `id` with that instance.

```
MetaCoq Run (n ← tmLemma "nat_inst" ℕ;
             e ← tmEval all n;
             tmPrint e ).
Next Obligation.
  exact (40+2 ).
Defined.
```

This program creates an obligation [28] for the user to provide a natural number and after the instance is provided, creates `nat_inst`, evaluate it and prints the number. An obligation is a goal that has to be closed for the definition. After the corresponding obligation for a `tmLemma` commands is closed the execution of the program continues.

3.5 Remarks

To conclude this introduction to MetaCoq we want to reflect if MetaCoq meets the ideals of the project. Therefore, we take a close look at the guarantees of MetaCoq and what a user has to trust.

Translation guarantees

Due to ongoing development and changes in TemplateCoq and PCUIC, some lemmas, for example, the main statement of the correctness of the two translations between the syntaxes, are admitted. Nevertheless, one can manually look at the transformations and proofs and see that the transformations are correct and only mostly intuitive statements are admitted. But formally the correctness is not fully proven yet.

The correctness of the translation is stated as typing preservation instead of a bijection. This is due to the two factors that, first, the second half of the translation, from PCUIC back to TemplateCoq, was only recently added. And, second, there are some problems with the statement as PCUIC does not contain casts, and therefore, there is no need to create TemplateCoq casts which leads to being non-surjective.

A possible solution to mitigate this would be to invent an equality relation and prove that every term converted to PCUIC and back satisfies this relation. But in the end, this equality is morally the same as to say that types behave the same in possible contexts which itself is already covered by the typing preservation.

MetaCoq guarantees

The trust in MetaCoq relies in part on the trust in Coq. The underlying trust is strengthened by the papers [31, 30] that use MetaCoq itself to create a formal specification of the type theory used by Coq and implement a trusted type checker for the kernel of Coq. The other part is the modelling of the syntax and semantic.

One has to make the same assumptions one usually makes when working with Coq, that is, that the kernel works correctly, Coq parses and executes the proofs in a proper way and Coq is a faithful implementation of the underlying type theory. To work with MetaCoq one has to also trust the conversion from OCaml to TemplateCoq, that the translations produce the right terms, and that the type system implemented in MetaCoq is the type system that was meant in the implementation of Coq's kernel. The TemplateCoq syntax implementation is trusted because there are rather simple functions from the OCaml types into TemplateCoq.

In the current state of the development one might have to check used typing lemmas by hand to be safe as the complete code has as of today over 75 admitted lemmas and over 140 todo comments.

A big advantage is that the source code is under active development and improvement. But this also means that many aspects are expected to change.

A big step to improve the trust would be an extensive well commented test suite and a comprehensive documentation. These tests would show that the functions and commands work in practice and how to use them. The tests should include use cases of the vernacular commands, example lemmas for typing and automation tactics, and some mini-plugins.

Chapter 4

Case analysis

Every inductive type has a case analysis principle that scrutinises an instance of the inductive type and allows proving statements on all elements by looking at each way to construct an instance instead of a general unknown instance.

A case analysis principle is used to perform case analysis on elements of inductive types. This corresponds to the behaviour of the destruct tactic in a proof.[1] If we want to prove a statement of the form $\forall(n : \mathbb{N}).\ P\ n$ over all natural numbers n, we can perform case analysis and prove the statement for 0 and for $S\ m$ with $m : \mathbb{N}$. The same strategy applies for inductively defined proposition like $\wedge$, $\vee$ and $\leqslant$.

An example is the case analysis principle for natural numbers (see Example 4.1.1):

$$E_{\mathbb{N}} :\ \forall(p : \mathbb{N} \to \mathbb{T}).\ p\ O \to (\forall m.\ p(S\ m)) \to \forall x.\ p\ x$$

The proof of a case analysis principle or induction principle is called an **eliminator**. The eliminators for case analysis principles is non recursive. Later on, we will see that the eliminators for induction principles use recursion.

Every inductive type has a case analysis principle similar to the induction principle without induction hypotheses and therefore without recursion in the eliminator.

A way to generate the case analysis principle in Coq is the `Scheme` command which generates a proof for the case analysis principle of an inductive type `T`.

```
Scheme T_case := Elimination for T Sort U.
```

We want to generate eliminators for the case analysis principles given a representation of an inductive type. For the generation, we use the MetaCoq project. Our main goal is to replicate the case analysis scheme command:

```
MetaCoq Run Scheme T_case := Elimination for T.
```

1 The destruct tactic generated the case analysis principle on the fly when applied.

The case analysis principle has to handle the replacement and instantiation of indices, quantification of variables like the m above and additional hypotheses as would be the case for less or equal (see Example 4.1.3).

4.1 Application and examples

In the following examples we show the constructed lemmas and use cases.

4.1.1 Natural numbers

$$\mathbb{N} : \mathbb{T}$$
$$n : \mathbb{N} ::= O \mid S\ n$$

$$E_{\mathbb{N}} :\ \forall(p : \mathbb{N} \to \mathbb{T}).\ p\ O \to (\forall m.\ p(S\ m)) \to \forall x.\ p\ x$$

For natural numbers the case analysis principle, also called non recursive eliminator, is quite simple: There is one case without arguments and a second case with a natural number as argument. There are no parameters or indices.

4.1.2 Disjunction

$$\vee : \mathbb{P} \to \mathbb{P} \to \mathbb{P}$$
$$L : \forall(A : \mathbb{P})\ (B : \mathbb{P}).\ A \to A \vee B$$
$$R : \forall(A : \mathbb{P})\ (B : \mathbb{P}).\ B \to A \vee B$$

Disjunction is an inductive type with two parameters A and B.

$$\frac{A}{A \vee B}\,L \qquad\qquad \frac{B}{A \vee B}\,R$$

$$\begin{aligned} E_{\vee} : \forall A\ B.\ \forall(p : A \vee B \to \mathbb{P}).&\\ (\forall(a : A).\ p\ (L\ A\ B\ a)) \to&\\ (\forall(b : B).\ p\ (R\ A\ B\ b)) \to&\\ \forall(x : A \vee B).\ p\ x& \end{aligned}$$

The two parameters A and B are quantified at the beginning. There are two cases, one for each way the disjunction can be proven. In both cases the proof of the proposition, either A or B, is quantified as argument and used to construct the instance of the disjunction for the predicate p.

4.1.3 Less or equal

$$le : \forall(n : \mathbb{N}).\ \mathbb{N} \to \mathbb{P}$$

Less or equal has one parameter, the first argument, and one index, the second argument.

$$\frac{}{\mathsf{le}\ n\ n}\ \mathsf{le\_n} \qquad\qquad \frac{\mathsf{le}\ n\ m}{\mathsf{le}\ n\ (\mathsf{S}\,m)}\ \mathsf{le\_S}$$

$$\begin{aligned}\mathsf{E}_{\mathsf{le}} : \forall(n : \mathbb{N}).\ \forall(p : \forall m.\ \mathsf{le}\ n\ m \to \mathbb{P}).\\ p\ n\ (\mathsf{le\_n}\ n) \to\\ (\forall m\ (h : \mathsf{le}\ n\ m).\ p\ (\mathsf{S}\ m)\ (\mathsf{le\_S}\ n\ m\ h)) \to\\ \forall m\ (x : \mathsf{le}\ n\ m).\ p\ m\ x\end{aligned}$$

In the first case the index is instantiated with the parameter n and the constructor has no additional arguments. In the second case the constructor takes a natural number m and the statement $\mathsf{le}\ n\ m$ as arguments. The index is instantiated with $\mathsf{S}\ m$ and all arguments are provided to the constructor. The indices are taken directly from the constructor.

4.2 Theory

Let T be an inductive Type. In general the case analysis principle of T applies to a statement $\forall(x : \mathsf{T}).\ p\ x$ where x is an instance of type T. By applying the case analysis principle x gets instantiated with a concrete application of the constructors. This generates one case for each constructor.

T can have arguments like disjunction (see Example 4.1.2) or less or equal (see Example 4.1.3). An argument is called a **parameter** if the instantiation is the same across all calls to T in the constructors and an **index** if the instantiation varies.

Parameters are quantified in front of the principle (see Example 4.2.2). In order to deal with the indices the predicate p does not only take the instance but also the indices (see Example 4.2.4) as arguments.

4.2.1 Parameter-free types

$$\begin{aligned}\mathsf{T} &: \mathbb{T}\\ \mathsf{E}_{\mathsf{T}} &: \forall(p : \mathsf{T} \to \mathbb{P}).\ p\ c_0 \to \ldots \to (\forall a_0 \ldots a_n.\ p\ (c_m\ a_0 \ldots a_n)) \to \forall(x : \mathsf{T}).\ p\ x\end{aligned}$$

An example of a parameter-free type is the type of natural numbers $\mathbb{N}$ (see Example 4.1.1).

If the inductive type T does not have any parameters and indices, then the case analysis principle E_{T} has one case for each constructor with quantification over all arguments of the constructor. In this example the constructor c_0 takes no argument, like O for natural numbers, and c_m takes n arguments called a_0 to a_n.

The result of E_{T} is the statement that p holds for every instance x of T if it holds in each possible way to construct x. The result type of p depends on T and p can be of type $\mathsf{T} \to \mathbb{T}$ if T allows for large elimination (see Section 2).

4.2.2 Index-free types

$$
\begin{aligned}
&T : T_{P_0} \to \ldots \to T_{P_k} \to \mathbb{T} \\
&E_T : \forall P_0 \ldots P_k.\ \forall (p : T\ P_0 \ldots P_k \to \mathbb{P}). \\
&\quad p\ (c_0\ P_0 \ldots P_k) \to \ldots \to (\forall a_0 \ldots a_n.\ p\ (c_m\ P_0 \ldots P_k\ a_0 \ldots a_n)) \to \\
&\quad \forall (x : T\ P_0 \ldots P_k).\ p\ x
\end{aligned}
$$

T_{P_i} is the type of the i-th parameter.

An example of an index-free type is the disjunction (see Example 4.1.2).

If the inductive type T has parameters $P_0, \ldots P_k$ but no indices, quantifications for the parameters have to be added. As the parameters are the same across the whole lemma, they are quantified first. For typing the parameters need to be applied to the constructors.

4.2.3 Non-uniform parameter types

Non-uniform parameters are parameters which can have different instantiations in recursive occurrences of T. They can be handled like normal parameters for case analysis principles but are seen more like indices. This also means that they are quantified in the cases and instantiated in the proof. This special treatment is to make the case analysis principles compatible to the induction principles. In Section 5.2.1 we will see in detail why this special treatment is necessary.

4.2.4 Indexed types

$$
\begin{aligned}
&T : T_{P_0} \to \ldots \to T_{P_k} \to T_{I_0} \to \ldots \to T_{I_l} \to \mathbb{T} \\
&E_T : \forall P_0 \ldots P_k.\forall (p : \forall I_0 \ldots I_l.T\ P_0 \ldots P_k\ I_0 \ldots I_l \to \mathbb{P}). \\
&\quad p\ i_0 \ldots i_l\ (c_0\ P_0 \ldots P_k) \to \ldots \to \\
&\quad (\forall a_0 \ldots a_n.p\ i_0 \ldots i_l\ (c_m\ P_0 \ldots P_k\ a_0 \ldots a_n)) \to \\
&\quad \forall I_0 \ldots I_l.\forall (x : T\ P_0 \ldots P_k\ I_0 \ldots I_l).p\ I_0 \ldots I_l\ x
\end{aligned}
$$

T_{I_i} is the type of the ith index.

An example of an indexed type is less or equal (see Example 4.1.3).

In the most general case T can have parameters $P_0, \ldots P_k$ and indices $I_0, \ldots I_l$ as well. As the instantiation of the indices can change in each constructor, the indices need to be provided to p and therefore p quantifies over indices and the instance of the inductive type T.

In each case the indices are instantiated with $i_0 \ldots i_l$ as they were in the constructor case. Here $i_0 \ldots i_l$ can be some terms possibly involving the parameters and arguments $a_0 \ldots a_n$ of the case.

```
E_⩽ := λ (n : ℕ)
(p : ∀ m : ℕ. n ⩽ m → ℙ)
(H_le_n : p n (le_n n))
(H_le_S : ∀ (m : ℕ) (H : n ⩽ m). p (S m) (le_S n m H)) ⇒
fix f (m : ℕ) (x : n ⩽ m) {struct x} : p m x :=
match x as y in (_ ⩽ m) return (p m y) with
| @le_n _ ⇒ H_le_n
| @le_S _ m x ⇒ H_le_S m x
end
```

Listing 4.1: *fully annotated case analysis principle proof term for* $\leqslant$

The conclusion of the principle is that p holds for every instance x of T with arbitrary indices I_0 to I_l if it holds for every way to construct instances of T.

4.3 Eliminator

$$
\begin{aligned}
E_T := {} & \lambda P_0 \ldots P_k. && (4.1)\\
& \lambda(p : \forall I_0 \ldots I_l.T\ P_0 \ldots P_k\ I_0 \ldots I_l \to \mathbb{P}). && (4.2)\\
& \lambda H_0 \ldots H_m. && (4.3)\\
& \text{fix } f\ I_0 \ldots I_l\ (x : T\ P_0 \ldots P_k\ I_0 \ldots I_l). && (4.4)\\
& \text{match } x \text{ return } p\ I_0 \ldots I_l\ x \text{ with} && (4.5)\\
& \quad c_i\ a_0 \ldots a_n \Rightarrow H_i\ a_0 \ldots a_n && (4.6)\\
& \quad \ldots && (4.7)
\end{aligned}
$$

The proof of the case analysis principles follows the type of the case analysis principle closely (compare Listing 4.1). To construct the case analysis principle, it is enough to construct the proof term and infer the type.

Parameters, the principle p and the branches for the constructors are dealt with λ-abstractions (4.1-4.3), an intro call in proof scripts.

The quantification over the indices and the instance is handled by a fixpoint declaration (4.4) as this allows us to use recursion later on. Currently, this is not necessary because case analysis does not use induction hypotheses.

The main proof is done using a match on the instance x (4.5) with application of the corresponding case in each constructor case (4.6). The arguments are forwarded from the constructor to the case.

Using type inference it suffices to provide the type of p, x and the return type of the match.

```
fold ( λ t param ⇒ lambda param.name param.type t ) params
```

Listing 4.2: *Pseudocode of parameter quantification*

4.4 Implementation

Our implementation follows the strategy outlined in Section 4.3 except the type inference of arguments cannot be used.

First, the `mutual inductive body` and the `one inductive body` of the specific type `T` are extracted. The `mutual inductive body` contains the parameters and the inductive types, which can be more than one in case of a mutual inductive definition. The `one inductive body` contains the name, type, elimination possibilities and constructors of `T`. Each constructor is represented by a number of arguments, a name and a type.

Afterwards, information for the later implementation is gathered like the types of the indices, the number of constructors and indices and some preparation on the type is done.

Step (4.1): The parameters are taken using `fold` over the parameter list. For each parameter a λ abstraction is nested around the proof term starting with the remaining proof (see Listing 4.2). The parameter list needs to be reversed as it is stored in reverse order in the inductive body.

Step (4.2): The next step is to take `p`. We can directly adopt the index quantification inside the type of the predicate from the inductive type without parameter quantification.

Step (4.3): Next, the cases for the constructors are taken. The main problem in this step is to construct the type according to the constructor type, remove parameters and construct the corresponding call to `p` with the constructor at the end. The instantiation of the indices is acquired from the original term by replacing `T` with `p`, removing the parameters in the application and appending a call of the constructor with all arguments to construct an element of `T` for `p`.

Step (4.4): The main step happens in the fixpoint declaration. For the type of the fixed point all indices and the instance are quantified as they are in the type of `p`. The quantification is changed to λ abstractions for the body of the fixed point to take each argument. Here we need to lift each parameter access additionally by one in comparison to the type as we have the recursive binder for `f` in front.

Step (4.5): The same λ abstractions are needed inside the return type of the match for the indices and match instance. Our result is again the application to `p` as it was

in the result of our fixed point.

Step (4.6): Finally, the match cases are generated where each constructor c_i calls the corresponding case hypothesis H_i and provides the arguments a from the constructor. This is done with a mapping on the constructor types and is nearly the same as in the cases above with different lifting for the parameters and application of the case instead of p.

4.5 Usage example

The program to generate a case analysis principle for type T can be called using the command `MetaCoq Run Scheme T_case := Elimination for T.`

To derive the case analysis principle for $\mathbb{B}$ one can execute the command:

```
MetaCoq Run Scheme bool_case := Elimination for bool Sort 𝕋.
```

The resulting case analysis principle is

$$\forall (p : \mathbb{B} \to \mathbb{T}).\ p\ true \to p\ false \to \forall (b : \mathbb{B}).\ p\ b$$

Similarly `MetaCoq Run Scheme ℕ_case := Elimination for ℕSort ℙ.` generates the case analysis principle for natural numbers:

$$\forall (p : \mathbb{N} \to \mathbb{P}).\ p\ O \to (\forall (m : \mathbb{N}).\ p\ (S\ m)) \to \forall (n : \mathbb{N}).\ p\ n$$

For less or equal the case analysis principle is a bit more complicated due to the parameter and index:

```
MetaCoq Run Scheme le_case := Elimination for le Sort ℙ.
```

$$\begin{aligned}
&\forall (n : \mathbb{N})\ (p : \forall m.\ n \leqslant m \to \mathbb{P}).\\
&\ p\ n\ (le_n\ n) \to\\
&(\forall (m : \mathbb{N})\ (h : n \leqslant m).\ p\ (S\ m)\ (le_S\ n\ m\ h)) \to\\
&\forall (m : \mathbb{N})\ (h : n \leqslant m).\ p\ m\ h
\end{aligned}$$

Chapter 5

Induction

In Chapter 4 the case analysis principles for inductive types were shown. Those principles allow to inspect how an instance of an inductive type is constructed. But if one wants to prove statements over all elements of an inductive type, case analysis principles are often not strong enough. In each constructor case one can assume that the predicate already holds for the structurally smaller arguments of inductive type in the constructor.

The knowledge that the predicate holds for the structurally smaller instances is expressed by the fact that additional assumptions are added in the cases of the principles. The assumptions are introduced for every recursive argument of the inductive type for which we construct the principle. One usually calls these new assumptions induction hypotheses.

Structural induction principles generate induction hypotheses for directly structurally smaller instances. These instances are given as argument to the constructor. For the constructor S of natural numbers the argument m is such an argument for which an induction hypothesis is added. Besides structural induction, there is also complete induction. In the complete induction principle for a type T, the assumption is that the principle holds for all smaller instances of the type T. In this thesis we will only look at structural induction.

The hypotheses get more involved when the type of the argument is not the inductive type but rather a construct involving the inductive type. We will look at these cases in Section 5.2.2 and Chapter 6.

5.1 Application and examples

One normally defines functions with recursive specification directly with recursion using fixed points. Alternatively, one could also use recursive eliminators as the cases of the predicate are a case analysis and provide the recursive result as induction hypotheses. Additionally, the eliminators can be used in a logical con-

text to prove statements by induction. Recall that an eliminator E_T for a type T is the proof of an induction principle and as such describes a function that takes a predicate, case assumptions, an instance of T and returns a proof of the predicate applied with the instance.

We look again at the examples from Chapter 4 and see how the inductive hypotheses change the principles. We will use blue colour to indicate differences compared to case analysis. From this chapter on, we will identify the recursive eliminator for the induction principle of a type T with E_T.

5.1.1 Natural numbers

$$\mathbb{N} : \mathbb{T}$$
$$n : \mathbb{N} ::= O \mid S\ n$$

$$\mathsf{E}_\mathbb{N} : \forall(p : \mathbb{N} \to \mathbb{T}).p\ O \to (\forall m.p\ m \to p(S\ m)) \to \forall x.p\ x$$

The base case O has no recursive argument of type $\mathbb{N}$ and therefore stays the same compared to the case analysis principle. As the successor has a recursive argument $m : \mathbb{N}$ we gain an additional assumption $p\ m$ stating that p holds for m. This usually makes it much easier to prove $p\ (S\ m)$ or in some cases is what makes it even possible.

Lemma 5.1 (Addition with zero) *We are now able to prove lemmas like* $\forall n.\ n+0 = n$. *If we apply the induction principle, we get two cases.*

The base case $0 + 0 = 0$ *is trivially provable with conversion.*

In the successor case $S\ m + 0 = 0$ *we get the additional assumption* $IH : p\ m$ *which is* $m + 0 = m$ *for this proof. With conversion the goal is transformed into a state where we have to solve the original problem* $m+0 = m$ *but now on the instance* m *which is structurally smaller than* n (*because* $n = S\ m$). *This can be solved with the induction hypothesis* IH *which directly gives us the needed statement for* m.

Additionally, we can use the eliminator to write functions for natural numbers: If we want to write the $\mathtt{even} : \mathbb{N} \to \mathbb{B}$ function determining whether a number is even, we have the base case that zero is even and the successor case where we negate the result of the even check for the number below. The boolean negation is handled with the function $\mathtt{negb} : \mathbb{B} \to \mathbb{B}$.

$$even\ O := true$$
$$even\ (S\ m) := negb(even\ m)$$

We can transform the basic equations for each case into an application of the eliminator.

$$even\ n := \mathsf{E}_{\mathbb{N}}\ (\lambda\_.\ \mathbb{B})\ true\ (\lambda m\ h.\ negb\ h)\ n$$

5.1.2 Less or equal

Less or equal is a predicate with a parameter and an index. Recall that predicates can only be eliminated over propositions (see Section 2).

$$\frac{}{le\ n\ n}\ le\_n \qquad\qquad \frac{le\ n\ m}{le\ n\ (S m)}\ le\_S$$

$$\begin{aligned}
\mathsf{E}_{le} : \forall (n : \mathbb{N}).\forall (p : \forall m.\ le\ n\ m \to \mathbb{P}).\ & \\
p\ n\ (le_n\ n) \to\ & \\
(\forall m\ (h : le\ n\ m).\ p\ m\ h \to p\ (S\ m)\ (le_S\ n\ m\ h)) \to\ & \\
\forall m\ (x : le\ n\ m).\ p\ m\ x &
\end{aligned}$$

As with natural numbers we have a recursive occurrence in the argument h in the le_S constructor. The statement $p\ m\ h$ takes the argument h as instance and extracts the indices from the type of h. This is necessary as the indices can be different in recursive calls and therefore need to be specified for p.

5.1.3 Accessibility

Accessibility is an inductive predicate with two uniform parameters A and R and one non-uniform parameter a.

$$\frac{\forall y.\ R\ y\ x \to Acc\ R\ y}{Acc\ R\ x}\ Acc_I$$

$$\begin{aligned}
\mathsf{E}_{Acc} : \forall (A : \mathbb{T})\ (R : A \to A \to \mathbb{P})\ (p : \forall (a : A).\ Acc\ R\ a \to \mathbb{T}) & \\
(\forall (a : A)\ (h : \forall (y : A).\ R\ y\ a \to Acc\ R\ y). & \\
\quad (\forall (y : A)\ (g : R\ y\ a).\ p\ y\ (h\ y\ g)) \to & \\
p\ a\ (Acc_I\ a\ h)) \to & \\
\forall (a : A)\ (x : Acc\ R\ a).\ p\ a\ x &
\end{aligned}$$

The non-uniform parameter a is treated as an index in the induction principle. The case for the constructor Acc_I has a recursive argument h for which an induction hypothesis is added. As the argument h is guarded recursive, additional quantification have to be added (see Section 5.2.2). From h a structurally smaller instance is generated with the application $h\ y\ g$ and used in the induction hypothesis as instance of Acc for the predicate p.

$$C : (h : T\ a\ i_0) \to T\ a\ i_1$$
$$H_C : (h : T\ a\ i_0) \to (IH_h : P\ i_0\ h) \to P\ a\ i_1\ (C\ h)$$

Figure 5.1: *The first line shows the constructor* C *of the Type* T *with one parameter and one index. In the second line we have the case* H_C *taking the same arguments. If an argument* h *is recursive, mentions the type* T*, an induction hypothesis* IH_h *is added. The type of* IH_h *is the type of* h *where the type application is replaced with a call to the predicate* P*. The instance of the type* T *is* h *itself for the predicate call in* IH_h *and the index instantiation is taken from* h*.*

$$\frac{G\ 42}{G\ n}\ G_S$$

Figure 5.2: *An inductive type* G *with a non-uniform parameter* n*.* n *is the same in all conclusions but is instantiated with* 42 *in* G_S*. Therefore, the predicate should be* $p : \forall n : \mathbb{N}.\ G\ n \to \mathbb{T}$ *and the induction hypothesis* $p\ 42\ h$ *when the argument is named* h*.*

If the second argument of p is ignored, one can see the structural induction principle for Acc as well-founded induction principle over elements of type A with R as well-founded relation: $p\ a$ holds if $p\ y$ holds for all elements that are smaller than a in the relation R.

5.2 Theory

The differences to case analysis principles are that we have to augment the cases with the induction hypotheses and generate their proofs in the eliminator. To generate the induction hypotheses we iterate over all arguments and if an argument mentions the inductive type, we copy it and replace the type with the predicate of the principle. The call to the principle uses the same index instantiation as the type occurrence in the argument and the argument is the instance (see Figure 5.1).

5.2.1 Non-uniform parameters

Recall that a non-uniform parameter is the same across all conclusions of the constructors but can vary in the arguments (see Figure 5.2, Section 4.2.3). The separate handling of non-uniform parameters becomes important for induction principles. The recursive calls to the predicate are for instances with other instantiations of the non-uniform parameter and have a different type. Therefore, we have to treat the non-uniform parameters like we treat indices and quantify the non-uniform parameters in the predicate type. This also means that we have to extract the parameter instantiations from calls to the inductive type and apply them to the predicate in induction hypothesis.

$$\frac{f\ n = F \to G\ (S\ n)}{G\ n}\ G_I$$

$E_G : \forall (f : \mathbb{N} \to \mathbb{B})(p : \forall n.\ G\ f\ n \to \mathbb{T}).$
$(\forall n\ (h : f\ n = F \to G\ f\ (S\ n)).\ (\forall (g : f\ n = F).\ p\ (S\ n)\ (h\ g)) \to p\ n\ (G_I\ f\ n\ h)) \to$
$\forall n\ (x : G\ f\ n).\ \to p\ n\ x$
$E_G\ f\ p\ H_{G_I}\ n\ (G_I\ \_\ \_\ h) := H_{G_I}\ n\ h\ (\lambda g.\ E_G\ p\ f\ H_{G_I}\ (S\ n)\ (h\ g))$

Figure 5.3: *G is an inductive type with a uniform parameter $f : \mathbb{N} \to \mathbb{B}$, a non-uniform parameter $n : \mathbb{N}$, and guarded recursion in G_I. We can observe that we treat n like an index for the most parts as it is a non-uniform parameter. The guarded recursion is in the argument h of G_I. We only get the structurally smaller instance $G\ f\ (S\ n)$ if $f\ n$ is false. Therefore, the induction hypothesis only gives a proof of p if $f\ n$ is false. Structurally the induction hypothesis is the same as the argument h where G was replaced by p and the instance derived from h with $h\ g$ was added to p. The proof works similar: we take $g : f\ n = F$ and use it to derive an instance with h which can be used to recursively call E_G.*

5.2.2 Guarded recursion

With guarded recursion we mean an argument where the recursion is "guarded" under quantifications. An example is the constructor $C : (f\ n = F \to T) \to T$. The first argument of type $f\ n = F \to T$ employs guarded recursion with the guard $f\ n = F$. This means that the first argument is a function returning a structurally smaller instance of T when supplied with an argument $f\ n = F$.

The use cases of guarded recursion are limited to strictly positive occurrences [34]. An occurrence of a type T in an argument is strictly positive only if it appears in the conclusion of that argument. Therefore, arguments with guarded recursion always take the form $A_0 \to \ldots \to A_n \to T$ where $A_0, \ldots A_n$ do not mention T.

If one would allow negative recursion, the type $D : \mathbb{T}$ with the constructor $DI : (D \to \bot) \to D$ would be valid. One then could write the function

```
Definition g (d: D) : ⊥ := match d with DI f ⇒ f d end
```

The function call $g\ (DI\ g) : \bot$ results in infinite recursion without termination. This, in turn, would diminish the trust in Coq as non-termination can be used to prove anything.

If we generate the induction principle for a type T and have an argument $H : nat \to T$, we get an instance of T if it is applied to a natural number. And, as we have a proof of the predicate for structurally smaller instances, the induction hypothesis

reflects this knowledge. The hypothesis $IH : \forall(n : nat), p(Hn)$ states that for every natural number the predicate holds for the instance constructed by H with n.

In general, the induction hypothesis for an argument H with guarded recursion can be obtained by changing the conclusion of H with an application to p with H applied to all arguments of H as instance for the inductive type. This procedure can be viewed as an iteration over the type of H replacing the occurrence of the inductive type with an application of the predicate (see Figure 5.3). The view of the iteration unifies guarded recursion with normal recursive arguments and the construction of the case type.

The proofs of guarded induction hypotheses are analogous to their type where the call to the predicate p is replaced with a call to the recursive function f.

5.3 Eliminator

For the eliminator we have to generate a proof of the new assumptions in each case. For example we can look at the proof of $E_{\mathbb{N}}$:

$$E_{\mathbb{N}}\ p\ (H_O : p\ O)\ (H_S : \forall m.\ p\ m \to p\ (S\ m))\ O := H_O$$
$$E_{\mathbb{N}}\ p\ (H_O : p\ O)\ (H_S : \forall m.\ p\ m \to p\ (S\ m))\ (S\ n) := H_S\ n\ (E_{\mathbb{N}}\ H_0\ H_S\ n)$$

We can observe that the proofs of induction hypotheses are recursive calls to the eliminator. This corresponds to the intuition that the induction hypothesis states that our predicate holds for the structurally smaller instance.

Similar to the construction of the hypothesis itself the proof transfers the indices from the argument type to the recursive function of our principle and applies the argument as instance of our type.

The proof is valid as the fixed point of our proof is only applied to structurally smaller arguments. Therefore, the eliminator terminates and returns a proof for the given instance.

Termination

To create the eliminator we apply a fixed point function recursively to generate the proofs for the induction hypotheses. Therefore, we need to make sure that the eliminator terminates in order to derive a valid function. Basically, Coq only supports structural recursion. This means that a fixed point can only be applied to arguments that are structurally directly smaller than the given ones.

This condition is satisfied for the eliminators E_T because we only call the recursive function on arguments of the constructors of the type T. Such constructor arguments are by definition structurally smaller than the instance derived from the constructor because the constructor took the argument to create the instance [7].

$$C : \ldots \to T\ p_0\ \ldots\ p_k\ p_{k+1}\ \ldots\ p_n\ i_0\ \ldots\ i_m \Rightarrow p\ p_{k+1}\ \ldots\ p_n\ i_0\ \ldots\ i_m\ (C\ \ldots)$$

Figure 5.4: *An example constructor* C *for a type* T *with* k *uniform parameter* $p_0, \ldots p_k$ *and some non-uniform parameter* $p_{k+1}, \ldots p_n$ *as well as with indices* $i_0, \ldots i_m$ *is transformed into the type of the case assumption for the induction principle.*

Therefore, the termination criterion for fixed points is trivially satisfied by the eliminators.

5.4 Implementation

The eliminators are implemented much the same way as the ones for case analysis. We only extend the function for the cases 4.3 and proofs 4.6.

We have seen that the generation of the type for the constructor cases and the proof term is similar in structure. Therefore, we write one function mapping a constructor to first generate the case type and secondly create the proof term in the match.

The function needs to know whether induction hypotheses should be added, the case type or proof term is generated, the de Bruijn index of the predicate `p`, the recursive function `f` (only for the proof term), and the de Bruijn index to identify the recursive call to the inductive type. Additionally, a body, an application term and an application list is needed as we will see.

We will take a closer look how to construct the assumptions and proofs for different terms by case analysis on the type of the constructor. The important cases are references `tRel n` and $\forall$-quantifications `tProd x A b`. Recall that the constructor `S` of natural numbers has the quoted type `tProd nAnon (tRel 0) (tRel 1)`. Here, both `tRel` terms point to the inductive type $\mathbb{N}$ itself, and therefore, indicate recursion.

tRel n

Constructor branches: The base case is a term of the form `tRel n` or `tApp (tRel n) ...`[1] which simply is transformed to a call of `p` if `n` is equal to the recursive position of the inductive type (see Figure 5.4). The predicate has to be applied with the non-uniform parameters and indices and lastly the application object. The application object is the constructor and the application list are the arguments encountered in the constructor.

Proof terms: For the proof term we need a case distinction whether we generate the proof term for an argument or for the whole constructor.

1 With this we mean an application with a `tRel` body. In PCUIC the body might be hidden under additional application terms.

$$\begin{aligned}\text{Constructor } S &: \mathbb{N} \to \mathbb{N}\\ \text{Case Assumption } H_S &: \forall(n : \mathbb{N}).p\ n \to p\ (S\ n)\\ \text{Proof } P_S &: \lambda(n : \mathbb{N}).H_S\ n\ (f\ n)\end{aligned}$$

Figure 5.5: *The case assumption and proof term for the successor constructor of natural numbers. For the main proof of* $p\ (S\ n)$ *(after we took all arguments), we can apply the case assumption* H_S *and supply all arguments. For each argument mentioning* $\mathbb{N}$ *we have to supply a proof for the newly added induction hypothesis. This is done with the recursive proof* f*.*

In the first case, we can directly adapt the procedure for the branch type by simply substituting f for p where f is the recursive function for the proof.

The second case is the main case where we want to generate the proof term that the predicate holds for the current constructor. Here we cannot apply f because f is only applicable to structurally smaller instances. Instead we call the corresponding case H, given as application object, and supply the constructor arguments which were collected in the application list during construction (see Figure 5.5).

tProd x A b

The other case of our function are terms of the form `tProd x A b` which correspond to $\forall(x : A).b$. We first try to generate an induction hypothesis / proof term for $x : A$.

Constructor branches: If A is suitable to be augmented into an induction hypothesis we add the hypothesis after the quantification of $x : A$ and lift the remaining term accordingly. Otherwise, we only add the quantification for $x : A$. Afterwards, we continue with b.

Proof term: For the proof term we also take a look at $x : A$ with a recursive call. If a proof term is generated, we add the proof term as additional application term into the application list for the call to b together with a call to the current argument x. Then, we compute the proof term for b and return a λ-abstraction taking the argument $x : A$ and returning the recursively generated proof term (see Figure 5.6)

5.5 Usage example

To generate the induction principle for a type `T` the plugin can be invoked with `MetaCoq Run Scheme T_induct := Induction for T.`

For natural numbers the induction principle is generated with

```
MetaCoq Run Scheme ℕ_induct := Induction for ℕ.
```

$$\begin{aligned}\text{Constructor } C &: \forall (x : T).\ \ldots \\ \text{Case Assumption } H_C &: \forall (x : T).\ IH_x \to \ldots \\ \text{Proof } P_C &: \lambda (x : T).\ H_C\ x\ P_x\ \ldots\end{aligned}$$

Figure 5.6: *For the case assumption of* C *we add the recursively generated induction hypothesis* IH_x*. For the proof term we don't change the* λ*-expression taking the arguments but only add a recursively generated proof term* P_x *to the call of* H_C*.*

$$\forall (p : \mathbb{N} \to \mathbb{T}).\ p\ O \to (\forall (n : \mathbb{N}).\ p\ n \to p\ (S\ n)) \to \forall n.\ p\ n$$

For less or equal the index instantiation for the induction hypothesis is taken from the recursive argument.

```
MetaCoq Run Scheme le_induct := Induction for le.
```

$$\begin{aligned}&\forall (n : \mathbb{N})\ (p : \forall m.\ n \leqslant m \to \mathbb{P}).\\ &\ p\ n\ (le_n\ n) \to\\ &(\forall (m : \mathbb{N})\ (h : n \leqslant m).\ p\ m\ h \to p\ (S\ m)\ (le_S\ n\ m\ h)) \to\\ &\forall (m : \mathbb{N})\ (h : n \leqslant m).\ p\ m\ h\end{aligned}$$

The induction principle of `Acc` is more complicated as it involved guarded recursion and a non uniform parameter `x`: `X`. The recursive call is generated by the instantiation $H\ y\ g$ of the argument that involves guarded recursion.

```
MetaCoq Run Scheme Acc_induct := Induction for Acc.
```

$$\begin{aligned}&\forall (X : \mathbb{T})\ (R : X \to X \to \mathbb{P})\ (p : \forall (x : X).\ Acc\ R\ x \to \mathbb{T}).\\ &(\forall (x : X)\ (h : \forall (y : X).\ R\ y\ x \to Acc\ R\ y).\\ &\quad (\forall (y : X)\ (g : R\ y\ x).\ p\ y\ (H\ y\ g)) \to\\ &\quad p\ x\ (Acc_{intro}\ x\ H)\\ &) \to\\ &\forall (x : X)\ (h : Acc\ R\ x).\ p\ x\ h\end{aligned}$$

5.6 Remarks

An interesting development was that the function started very involved and difficult. We first generated the case types and proof terms separately with nested

folding and mapping where some parts of the terms were replaced manually. The induction hypotheses, for example, were first constructed by an indexed filter and mapping operation over the collected arguments which was hard to understand and debug, and not at all extensible to guarded recursion.

As we added guarded induction, nested induction, and moved to PCUIC, our induction principles were not only more general and useful but the functions became simpler and more structured over time. We were able to unify the different aspects of induction, develop a general mapping over the constructors and combine the type function with the proof term function into one function.

At first, the induction hypotheses were added after all arguments as this leads to overall easier lifting behaviour and was implemented using mapping. But as the function developed, we switched to a recursive approach with much easier subtasks which also caused the induction hypotheses to be directly after the arguments as it is usual in Coq.

In Section 5.3 we mentioned that the recursion has to be structural in order to guarantee termination. This is checked when the eliminator is unquoted into Coq by the Coq kernel but cannot be checked by the typing predicate in MetaCoq as this condition is currently not implemented in MetaCoq.

Chapter 6

Nested induction

Induction principles enable proving statements on all elements of inductive types. To achieve this, the principle provides additional assumptions, the induction hypotheses, for recursive occurrences of the type in the constructors. Section 5.2.2 showed that the occurrence can be guarded in function calls. Moreover, the occurrence can also be nested in other inductive types like lists.

In the following chapter we will discuss how to derive stronger induction principles and eliminators for such inductive types. We will first see a generalization of the hypotheses followed by the mathematical theory and implementation in MetaCoq.

We call types like lists that depend on other types **container types**, because instances of $\mathsf{list}\ X$, for example, contain elements of the argument type X. An argument H of a constructor C of the type T is called **nested recursive** [5] if the recursive occurrence of T in H is an argument of a container type. An example is the argument H in this constructor $C : (H : \mathsf{list}\ T) \to T$.

6.1 Application and examples

Nested inductive types often are used when complex structures are needed. Such cases are rose trees, trees with arbitrary many direct sub-trees, or types representing the syntax of languages.

6.1.1 Rose trees

A nested recursive argument is used for $\mathsf{roseTree}$:

$$\mathsf{roseTree} ::= \mathsf{node}\ (xs : \mathsf{list}\ \mathsf{roseTree})$$

The argument of node is a list of sub-trees. Coq generates the following principle that only performs a case analysis and provides no further information about the list of trees xs:

$$\forall P.\ (\forall xs.\ P\ (\mathsf{node}\ xs)) \to \forall r.\ P\ r$$

But one can write a stronger principle that provides P for all trees in the list xs [5].

$$\forall P.\ (\forall xs.\ (\forall x.\ x \in xs \to P\ x) \to P\ (node\ xs)) \to \forall r.\ P\ r$$

The eliminator of the stronger principle is the same as the eliminator of Coq's principle except that one has to provide a proof for the new assumption.

$$\begin{aligned}
&\lambda P\ H.\ fixf\ (r : roseTree) : P\ r :=\\
&match\ r\ with\\
&\quad node\ xs \Rightarrow H\ xs\ (F^{\in}_{list}\ roseTree\ P\ f\ xs)\\
&end
\end{aligned}$$

The function $F^{\in}_{list}$ generates the proof of the new assumption for the list xs. We call such a function **proof function**.

This principle is useful for most applications in proofs. For example, relations over the depth and size of rose trees are provable. But this principle cannot be used computationally.

The stronger principle for roseTree is not strong enough to define a size function on roseTree. The size function $size : roseTree \to \mathbb{N}$ maps rose trees to the number of nodes in the tree. The size of a rose tree is one more than the sum of the sizes of the direct sub-trees. Therefore, the size is computed using the sizes of all direct sub-trees. With the eliminator generated by Coq it is not possible to compute such a function because no results for the direct sub-trees are calculated.

Even with the stronger principle provided here one cannot define the size function. On the one hand, the eliminator does no allow for large elimination due to the definition of $\in$. On the other hand, the structure of the list xs is lost in the predicate and therefore, one cannot sum up the sizes of the direct sub-trees in xs.

$$\frac{}{All\ X\ P\ []}\ All_{nil} \qquad \frac{P\ x \qquad All\ X\ P\ xs}{All\ X\ P\ (x :: xs)}\ All_{cons}$$

The construction of All follows the structure of list and the All_{cons} constructor guarantees that every element $x : X$ in the list satisfies P. Intuitively All X P xs is equivalent to the existence of a list $ys : list\ (\Sigma_x\ P\ x)$ such that $map\ \pi_1\ ys = xs$. In Section 6.2.1 we will discuss how this constructor can be generalized for arbitrary container types. With the new predicate the assumption becomes All roseTree P xs and hence the eliminator is strong enough to define a size function with induction over the All predicate.

$$\forall P.\ (\forall xs.\ All\ roseTree\ P\ xs \to P\ (node\ xs)) \to \forall r.\ P\ r$$

This principle is strong enough even for computation of functions. But this approach does not generalize to arbitrary container types other than list.

A general approach is to use unary parametricity for the new assumptions. We will discuss the general approach with unary parametricity in Section 6.2.1. The principle with the unary parametricity translation as assumption is

$$\forall P.\ (\forall xs.\ \mathrm{list}^t\ \mathrm{roseTree}\ P\ xs \to P\ (\mathrm{node}\ xs)) \to \forall r.\ P\ r$$

Here, list^t is basically the same as All.

6.1.2 Binary trees

Another example for nested induction are binary trees defined with pairs:

$$\mathrm{bTree} ::= \mathrm{leaf} \mid \mathrm{bnode}\ (h : \mathrm{bTree} \times \mathrm{bTree})$$

For this definition of bTree the bnode constructor takes an argument with nested recursive occurrences in the pair container type. Like for $\mathrm{roseTree}$ the induction principle generated by Coq gives no additional assumptions in the case of bnode:

$$\forall (P : \mathrm{bTree} \to \mathbb{P}), P\ \mathrm{leaf} \to (\forall (h : \mathrm{bTree} \times \mathrm{bTree}),\ P\ (\mathrm{bnode}\ h)) \to \forall b,\ P\ b$$

Like with lists, an assumption with an element predicate $\in_\times$ can be added:

$$x \in_\times (p_1, p_2) := (x = p_1 \vee x = p_2)$$

$$\begin{aligned}
&\forall (P : \mathrm{bTree} \to \mathbb{P}).\ P\ \mathrm{leaf} \to \\
&(\forall (h : \mathrm{bTree} \times \mathrm{bTree}).\ (\forall x.x \in_\times h \to P\ x) \to P\ (\mathrm{bnode}\ h)) \to \\
&\forall b.\ P\ b
\end{aligned}$$

For an even stronger induction principle one can add the assumption $P\ x \times P\ y$ for the argument $(x, y) : \mathrm{bTree} \times \mathrm{bTree}$:

$$\begin{aligned}
&\forall (P : \mathrm{bTree} \to \mathbb{P}).\ P\ \mathrm{leaf} \to \\
&(\forall (h : \mathrm{bTree} \times \mathrm{bTree}).\ p\ (\pi_1\ h)\ \times\ p\ (\pi_2\ h) \to P\ (\mathrm{bnode}\ h)) \to \\
&\forall b.\ P\ b
\end{aligned}$$

It is not obvious how to generalize the principle, but again it will turn out that this assumption is basically the same as the unary parametricity translation of products.

6.1.3 First-order terms

First-order terms over signatures can be variables identified by natural numbers and function applications. For the function applications the correct numbers of first-order terms have to be applied to the function. Therefore, the constructor takes as argument a vector.

$$\mathrm{foterm} := \mathrm{var}\ (n : \mathbb{N}) \mid \mathrm{func}\ (f : X)\ (a : \mathrm{vec}\ \mathrm{foterm}\ (\mathrm{arity}\ f))$$

The type X is the spaces of functions. The principle generated by Coq does not provide any assumptions about the terms that are applied to the function in the func constructor.

$$
\begin{aligned}
&\forall(P : \mathrm{foterm} \to \mathbb{T}).\\
&(\forall(n : \mathbb{N}).\ P\ (\mathrm{var}\ n)) \to\\
&(\forall(f : X)\ (a : \mathrm{vec}\ \mathrm{foterm}\ (\mathrm{arity}\ f)).\\
&\ P\ (\mathrm{func}\ f\ a)) \to\\
&\forall(f : \mathrm{foterm}).\ P\ f
\end{aligned}
$$

With the correct function for the induction hypothesis assumptions one gets the principle

$$
\begin{aligned}
&\forall(P : \mathrm{foterm} \to \mathbb{T}).\\
&(\forall(n : \mathbb{N}).\ P\ (\mathrm{var}\ n)) \to\\
&(\forall(f : X)\ (a : \mathrm{vec}\ \mathrm{foterm}\ (\mathrm{arity}\ f)).\\
&\quad \mathrm{vec}^{t}\ \mathrm{foterm}\ P\ (\mathrm{arity}\ f)\ a \to\\
&\ P\ (\mathrm{func}\ f\ a)) \to\\
&\forall(f : \mathrm{foterm}).\ P\ f
\end{aligned}
$$

In this principle, $\mathrm{vec}^{t}\ T\ P\ n\ t$ states that P holds for all elements of type T in $t : \mathrm{vec}\ T\ n$.

6.1.4 TemplateCoq terms

The term type from TemplateCoq (see Section 3.1) is also a type with many nested recursive arguments. Therefore, the induction principle that is generated by Coq is too weak for many applications as important assumptions are missing.

The constructor `tApp` takes a list of terms as argument, the `tCase` constructor has an argument of type `list (nat × term)`, and `tFix` and `tCoFix` take an argument of type `mfixpoint term`. Therefore, the types `list`, `prod`, and the type definition `mfixpoint` are used as containers. The function `mfixpoint X = list (def X)` is a nesting of two container types. With the generalization of container types to arbitrary container definitions, it is possible to add a induction hypotheses for all of the container definitions in the `term` type.

The term type also shows, that our plugin is able to derive induction principles for types with nested containers, arguments where the type is a container inside another container like `list (nat × term)` and `list (def term)`.

Because the principle is quite long, we will only look at the important changes:

$$
\begin{aligned}
&\ldots \\
&(\forall (f : \mathsf{term})\ (\mathsf{args} : \mathsf{list}\ \mathsf{term}).\ P\ f \to P\ (\mathsf{tApp}\ f\ \mathsf{args})) \to \\
&(\forall\ ip\ (t : \mathsf{term})\ (d : \mathsf{term})\ (b : \mathsf{list}\ (\mathbb{N} \times \mathsf{term})). \\
&\quad P\ t \to P\ d \to \\
&\ P\ (\mathsf{tCase}\ ip\ t\ d\ b)) \to \\
&(\forall (\mathsf{mfix} : \mathsf{mfixpoint}\ \mathsf{term})\ (\mathsf{idx} : \mathbb{N}).\ P\ (\mathsf{tFix}\ \mathsf{mfix}\ \mathsf{idx})) \to \\
&\ldots
\end{aligned}
$$

In the strong principle induction hypotheses are added for all recursive arguments:

$$
\begin{aligned}
&\ldots \\
&(\forall (f : \mathsf{term})\ (\mathsf{args} : \mathsf{list}\ \mathsf{term}).\ P\ f \to \\
&\quad \mathsf{list}^{\mathsf{t}}\ \mathsf{term}\ P\ \mathsf{args} \to \\
&\ P\ (\mathsf{tApp}\ f\ \mathsf{args})) \to \\
&(\forall\ ip\ (t : \mathsf{term})\ (d : \mathsf{term})\ (b : \mathsf{list}\ (\mathbb{N} \times \mathsf{term})). \\
&\quad P\ t \to P\ d \to \\
&\quad \mathsf{list}^{\mathsf{t}}\ (\mathbb{N} \times \mathsf{term})\ (\lambda (h : \mathbb{N} \times \mathsf{term}).\ \top \times p\ (\pi_2\ h))\ b \to \\
&\ P\ (\mathsf{tCase}\ ip\ t\ d\ b)) \to \\
&(\forall (\mathsf{mfix} : \mathsf{mfixpoint}\ \mathsf{term})\ (\mathsf{idx} : \mathbb{N}). \\
&\quad \mathsf{list}^{\mathsf{t}}\ (\mathsf{def}\ \mathsf{term})\ (\lambda (x : \mathsf{def}\ \mathsf{term}).\ p\ (\mathsf{dtype}\ x) \times p\ (\mathsf{dbody}\ x))\ \mathsf{mfix} \to \\
&\ P\ (\mathsf{tFix}\ \mathsf{mfix}\ \mathsf{idx})) \to \\
&\ldots
\end{aligned}
$$

Before, the induction principles for TemplateCoq terms were written and proven by hand. As the term type is large, it is tedious to even state the principle. it is tedious to even state the principle. For the proof of the principle good bookkeeping is required and such it takes a long time to do the proof by hand.

6.2 Theory

One can observe that the structure of the induction hypotheses for nested recursive occurrences depends on the container type used. We call the predicates used to generate the induction hypotheses of nested recursive arguments **assumption functions**, the applications of these functions are marked blue above. We call the corresponding functions providing the proof terms **proof functions**.

The proof function is used in the eliminator to derive the proof term for the new induction hypotheses involving the assumption functions. In the following part of

the eliminator $E_{roseTree}$ for roseTree the recursive function is f and F_{list} is the proof function:

$$\ldots \quad node\ xs \Rightarrow H_{node}\ xs\ (F_{list}\ roseTree\ P\ f\ xs) \ldots$$

For roseTree an assumption function for list is needed. In general these assumption functions P_{list} for list have the type $\forall X\ (P : X \to \mathbb{T}).\ list\ X \to \mathbb{T}$. The proof functions then are required to have the type $\forall X\ (P : X \to \mathbb{T})\ (f : \forall x.\ P\ x)\ (xs : list\ X).\ P_{list}\ X\ P\ xs$, where f is the recursive function used to generate proof for the smaller instances in the container type.

A straightforward assumption function is $P_{list}\ X\ P\ xs := \forall (x : X).\ x \in xs \to P\ x$. This function simply states that every element in the list satisfies the predicate.[1] The generated induction principle is the first one from above.

6.2.1 Parametricity

To generate induction hypotheses for arbitrary container types a general concept is needed. Tassi shows that the unary parametricity translation of a container type can be used to create the induction hypotheses [33].

Parametricity translations [26] are a technique used to express relations over objects of types. The parametricity translation is commonly used to derive statements from the type of a function alone [37]. The unary parametricity translation of an inductive type can be viewed as a predicate over elements of that type.

Following the idea from Tassi [33] we use the unary parametricity translation of the container types to generate assumption functions. We show how the parametricity translation fits the definition of assumption functions. The connection between induction principles and unary parametricity is also seen in [19]. The parametricity translation is implemented in MetaCoq [1].

In the following T^t is the unary parametricity translation of T.

We will first look at the translation for the container type list.

$$\frac{}{list^t\ X\ P_X\ []}\ nil^t \qquad\qquad \frac{P_X\ x \qquad list^t\ X\ P_X\ xs}{list\ X\ P_X\ (x :: xs)}\ cons^t$$

The translation adds a second parameter P_X for the type parameter X. The unary parametricity translation of list can be seen as a predicate over lists which states that all elements in the list satisfy P_X. Therefore, the predicate $list^t\ X\ P_X\ xs$ over the list xs states that every element in xs satisfies P_X and also preserves the structure of lists in its own structure.

1 This is also how Isabelle [24] generates nested induction principles (see Section 8).

The specification for $\mathtt{list}^t$ states that if P_X holds for every element x, then $\mathtt{list}^t\ X\ P_X\ xs$ for the list xs can be constructed with proofs of P_X for every element in the list:

$$(\forall x.\ P_X\ x) \to (\forall xs.\mathtt{list}^t\ X\ P_X\ xs)$$

This implication is also the type of the proof function $F_{\mathtt{list}}$. On the other hand, $\mathtt{list}^t\ X\ P_X\ xs$ also ensures that P_X holds for all elements in xs.

$$\mathtt{list}^t\ X\ P_X\ xs \to \forall x.\ x \in xs \to P_X\ x$$

The translation $\mathtt{list}^t : \forall(X : \mathbb{T})\ (P_X : X \to \mathbb{T}).\ \mathtt{list}\ X \to \mathbb{T}$ already has the correct type for an assumption function. One also can observe that All and $\mathtt{list}^t$ are equal up to α-renaming. Therefore, $\mathtt{list}^t$ can directly be used as an assumption function in induction principles for nested inductive types.

One can also derive the unary parametricity translation for non-container types like $\mathbb{N}$. But we will not need the parametricity translation of these types.

$$\frac{}{\mathbb{N}^t\ O}\ O^t \qquad\qquad \frac{\mathbb{N}^t\ n}{\mathbb{N}^t\ (S\ n)}\ S^t$$

The translation can be seen as a predicate over natural numbers that is satisfied for every number.

$$\forall(n : \mathbb{N}).\ \mathbb{N}^t\ n$$

Hereby, $\mathbb{N}^t$ follows exactly the structure of $\mathbb{N}$ itself.

Some problems emerge for container types with additional arguments with a type that is not $\mathbb{T}$ like $vec : \forall(A : \mathbb{T})\ (n : \mathbb{N}).\ \mathbb{T}$.

$$\frac{}{vec^t\ X\ P_X\ O\ O^t\ (nil\ X)}\ nil^t \qquad \frac{P_X\ x \qquad vec^t\ X\ P_X\ n\ n^t\ xs}{vec\ X\ P_X\ (S\ n)\ (S^t\ n\ n^t)\ (cons\ X\ x\ n\ xs)}\ cons^t$$

The translation vec^t adds an additional argument of type $\mathbb{N}^t$ that does not fit in our general concept of assumption functions. These arguments cause problems as described in Section 6.4 and 6.6.

In general, the unary parametricity adds an additional argument for every argument. For a type argument $A : \mathbb{T}$ the argument $A^t : A \to \mathbb{T}$ is added. For every argument t with type T other than $\mathbb{T}$, a new argument with the unary parametricity of T applied to t is created. As example, $a : A$ is translated to $h : A^t\ a$, $n : \mathbb{N}$ to $h : \mathbb{N}^t\ n$, and $xs : list\ A$ to $h : \mathtt{list}^t\ A\ A^t\ xs$.

6.2.2 Auxiliary definitions

The assumption function P_{T_C} for the container type T_C needs to know the argument type X and the corresponding predicate p. In general, the assumption function for a type with n type arguments and m other parameters or indices has the type

$$\forall X_1\ (P_{X_1} : X_1 \to \mathbb{T})\ \ldots\ X_n\ (P_{X_n} : X_n \to \mathbb{T})\ p_1\ \ldots\ p_m.\ T_C\ X_1\ \ldots\ X_n\ p_1\ \ldots\ p_m$$

The structure becomes clear if one looks at pairs $\mathtt{prod} : \forall(A : \mathbb{T})\ (B : \mathbb{T}).\ \mathbb{T}$ that have two type arguments, and vectors $\mathit{vec} : \forall(A : \mathbb{T})\ (n : \mathbb{N}).\ \mathbb{T}$ which have an index that is not a type. The types of the assumption functions will be $P_{\mathtt{prod}} : \forall(A : \mathbb{T})\ (P_A : A \to \mathbb{T})\ (B : \mathbb{T})\ (P_B : B \to \mathbb{T}).\ A \times B \to \mathbb{T}$ and $P_{\mathit{vec}} : \forall(X : \mathbb{T})\ (P_X : X \to \mathbb{T})\ (n : \mathbb{N}).\ \mathit{vec}\ X\ n \to \mathbb{T}$ respectively. This style makes it easier to use the assumption functions in the induction principle because it is possible to go through all arguments and add the generated principle if one is generated.

In contrast to implications of arguments, it is not possible to ignore assumptions that do not generate induction hypotheses because the assumption function has to be instantiated. If an argument in the constructor C of the type T has the type $T \times \mathbb{N}$, only an induction hypothesis for the first argument T can be generated but not for $\mathbb{N}$.

One possibility would be to have assumption functions depending on how many recursive arguments are provided but this procedure leads to 2^n different assumption functions for a type with n type arguments.

The induction hypothesis for $h : T \times \mathbb{N}$ is $IH_h : P_{\mathtt{prod}}\ T\ p\ \mathbb{N}\ ?\ h$, where the question mark has to be instantiated with a predicate over natural numbers. But there is no meaningful predicate for the induction hypotheses. Therefore, a **dummy predicate** $P_D : \forall(X : \mathbb{T})\ (x : X).\ \top$ is used and thus the induction hypothesis becomes $IH_h : P_{\mathtt{prod}}\ T\ p\ \mathbb{N}\ (P_D\ \mathbb{N})\ h$.

With the assumption function $P_{\mathtt{prod}}\ X\ P_X\ Y\ P_Y\ (x, y) := P_X\ x \times P_Y\ y$ the hypothesis is $P\ (\pi_1 h) \times \top$.

6.2.3 Guarded recursion

Nested recursion guarded in arguments like $H : X \to \mathtt{list}\ T$ in a constructor for a type T can be handled as described in Section 5.2.2. This can be done by deriving an instance $\mathtt{list}\ T$ from H and generating the induction hypothesis with $P_{\mathtt{list}}$ for that instance.

But if the argument H has the type $\mathtt{list}\ (X \to T)$, a predicate of type $(X \to T) \to \mathbb{T}$ for the type $X \to T$ is needed for $P_{\mathtt{list}}$. The induction hypothesis can be generated as $P_{\mathtt{list}}\ (X \to T)\ (\lambda(h : X \to T)\ (g : X).\ p\ (h\ g))\ H$ by using η-conversion. With η-conversion P_X can be transformed to $\lambda x.P_X\ x$; and from there the instance x can

```
E_roseTree := λ(P : roseTree → 𝕋)
        (H_node : ∀(xs : list roseTree).P_list roseTree P xs → P (node xs))
        fix f (x : roseTree) : P x.
        match x [
          node xs ⇒ H_node xs (F_list roseTree P f xs)
        ]
```

Figure 6.1: *The eliminator for* roseTree *with the assumption function* P_{list} *and proof function* F_{list}. P_{list} *takes the same arguments as* list *but for the type argument an additional predicate* P *and lastly the instance* xs. F_{list} *takes the same arguments as* list *with an additional argument* f *for the type argument.* f *is used by* F_{list} *to generate the proof terms for the elements in* xs.

be replaced by functions calls. In general, the predicate is generated by taking all guard arguments of H with λ-abstraction and using them to generate an instantiation for P.

6.3 Eliminator

As already mentioned, the proof function takes for each type argument X not only the predicate $P_X : X \to \mathbb{T}$ but also a function $F_X : \forall(x : X).\ P_X\ x$ and returns a proof of the corresponding assumption function P_X.

The proof function is instantiated like the assumption function to generate the proof terms for the nested induction hypotheses. For the functions F_X the proof term is generated recursively. Recall that the proof term for an argument H : T in the eliminator of T is a function call to the recursive function f.

For rose trees the eliminator needs the assumption function P_{list} and the proof function F_{list}.

```
E_roseTree := λ(P : roseTree → 𝕋)
        (H_node : ∀(xs : list roseTree).P_list roseTree P xs → P (node xs))
        fix f (x : roseTree) : P x.
        match x [
          node xs ⇒ H_node xs (F_list roseTree P f xs)
        ]
```

P_{list} takes the same arguments as list but for the type argument an additional predicate P and lastly the instance xs. F_{list} takes the same arguments as list with an ad-

```
induction xs.
- constructor.
- constructor.
  + apply F_X.
  + apply IH_xs.
```

Figure 6.2: *The proof of* $F_{list} : \forall(X : \mathbb{T})\ (P_X : X \to \mathbb{T})\ (F_X : \forall(x : X).\ P_X\ x)\ (xs : list\ X).\ list^t\ X\ P_X\ xs$. *After induction over* xs *one has to prove* $list^t\ X\ P_X\ [\,]$ *and* $list^t\ X\ P_X\ (x :: xs)$. *The first case is handled by the constructor* nil^t. *In the second case the* $cons^t$ *constructor needs a proof of* $P_X\ x$ *which is generated by* F_X *and a proof for* $list^t\ X\ P_X\ xs$ *which is exactly the induction hypothesis* IH_{xs}.

ditional argument f for the type argument. In the application of F_{list} the recursive function f is used to generate the proof terms for the elements in xs.

The proof function of a type T is constructed by induction over the instance of type T followed by calls to the constructor (see Figure 6.2). Afterwards, the goals remaining are the recursive calls which are exactly the induction hypotheses as well as the statements that the predicates P_Y holds for every element in the container type. The proofs of P_Y can be generated by the provided functions stating $\forall(y : Y).\ P_Y\ y$.

Termination

For Coq to accept the eliminator it has to recognize structural recursion in order to make sure the eliminator always terminates.

The termination of a function f is checked with a structural guard performing a check that every application of f is in a match of the recursive argument and is applied to an argument of the constructor [16]. The check is recursive which allows to nest multiple matches and even functions.

The proof functions F_C are terminating as they are defined as standalone functions. Therefore, we only have to make sure that the applications of the recursive function f that is provided to the proof function are allowed by the termination check of Coq (see Figure 6.3). It is required for the proof functions to be defined transparently for Coq to perform the termination check.

For the first assumption function for $list$, namely $P_{list}\ X\ P\ xs = \forall(x : X).\ x \in xs \to P\ x$, a valid proof function can be constructed in the following manner: First induction on xs is performed, deriving a contradiction with $x \in [\,]$ in the base case. For the successor case a case analysis with subsequent rewriting is performed on $x \in xs$, followed by an application of $f : \forall(x : X).\ P\ x$. This is a valid proof function because f is only applied to arguments in the list xs and thus smaller arguments than $node\ xs$.

```
E_roseTree := ...
        fix f (r : roseTree) : P r.
        match x [
        node xs ⇒ H_node xs
          (fix F (xs' : list roseTree).
            match xs' [
            ...
            | y :: ys ⇒ cons^t ... (f y) ys (F ys)
          ]) xs
        ]
```

Figure 6.3: *The eliminator for* roseTree *with* F_{list} *unfolded to highlight the recursion. The function* f *is terminating because it is only applied to smaller arguments. As* xs : list roseTree *is a constructor argument it is smaller than the argument* r*. In* F *a match over the argument* xs *is performed leading to the smaller argument* y : roseTree *to which* f *is applied. Therefore,* f *is only applied to arguments that are smaller than the argument* r*.*

In contrast, a naive approach might be to ignore $x \in xs$ and directly apply f_X to P x. This is a valid proof for the statement but does not show at all why f_X is only applied to smaller instances because in principle the x might be completely unrelated to node xs. Therefore, this proof function cannot be used to generate the eliminator for roseTree.

A wrong application of the argument f is avoided if the parametricity translation is used as assumption function. The translation encases the proofs for the contained elements in the original structure of the container type and therefore only exposes proof goals on obviously smaller instances. Hence, a valid proof providing a proof term for the assumption function is always an accepted proof function.

Therefore, the parametricity translation for assumption functions ensures that Coq accepts the use of the proof functions in the eliminator because all function calls are on smaller arguments.

6.4 Implementation

For the implementation of the nested induction it is important to have access to the assumption and proof function for all container types in the inductive type. The functions generating the eliminator have a **nesting function** as argument that looks up the container types and returns the assumption and proof function. The

nesting function needs to be constructed for each inductive type.

6.4.1 Database

The nesting function is constructed by iteration over all arguments of all constructors filtering the types suited for induction used in these arguments.

A type class `registered` is introduced to manage all possible container types that have a corresponding assumption and proof function.

From the list of container types the nesting function is generated by a `TemplateMonad` program that tries to get a `registered` instance for each one. If no entry is found, it is tested whether the type is even a container type or if it should be discarded. This distinction is necessary as for an argument of type $\mathsf{T} \times \mathbb{N}$ both `prod` and $\mathbb{N}$ are added, but $\mathbb{N}$ obviously is not a container type. If the type is a container type but not found, a human-readable warning with instructions how to add the container to the database is displayed.

If an entry in the `registered` type class is found, the assumption and proof function are quoted and added to the nesting function.

The registered database is implemented using a type class [29] named `registered`.

```
Class registered {X:𝕋} (ty:X ) :=
{
  assumptionType: 𝕋;
  assumption: assumptionType;
  proofType: 𝕋;
  proof: proofType
}.
```

The argument `ty` identifies the container type. The implicit type `X` would be for example $\mathbb{T} \to \mathbb{T}$ for `list`. The field `assumption` and `proof` are the assumption and proof functions with the corresponding types `assumptionType` and `proofType`. For lists, an instance might be {| `assumption` := `list`$^{\mathsf{t}}$; `proof` := ... |} : `registered list`.

6.4.2 Eliminator generation

The main algorithm stays the same except that every function needs the nesting function to pass it along. Additionally, the generation function (see Sections 5.44.4) for the proofs and assumptions gets a Boolean parameter stating whether dummy predicates and their proof terms should be generated. The dummy predicates are needed to guarantee a full application of the assumption and proof functions. If no induction hypothesis can be generated, the dummy predicate simply is $\lambda(X : \mathbb{T})\ (x : X).\ \top$ and the proof term is $\lambda(X : \mathbb{T})\ (x : X).\ \mathrm{I}$.

The changes to the generation function can be viewed in two cases:

Constants and inductive types are looked up in the nesting function. If the hypothesis is generated, the assumption function is returned and for the eliminator the proof function is returned. If the type is not a container type, a dummy result is returned if required.

For applications to constants and inductive types it is tested if a recursive occurrence is in the arguments. Otherwise, a dummy result is returned if needed because no induction hypothesis can be generated.

For the term $b\ x$ the hypothesis / proof for b is generated first. The result is then applied to the argument x, the recursively generated predicate for x, and the proof term of x if the eliminator is generated.

For the outermost application the recursively generated hypothesis / proof term, that is an application of the assumption function or proof function, also needs to be applied with the argument of the container type on which this hypothesis is based.

For example, an application of $F_{list}\ X\ P_X\ F_X\ xs$ takes the type X, the predicate or induction hypothesis P_X for X, the proof term for elements of X F_X, and lastly the argument of type $list\ X\ xs$.

6.4.3 Adding containers

The dummy proofs and arguments are only generated for the type arguments of the container type. For $vec\ X\ n$ only an hypothesis for $X : \mathbb{T}$ but not for $n : \mathbb{N}$ is generated. The lack of predicates for other arguments is the reason why the parametricity translation cannot be used directly for the assumption functions.

Although $\mathbb{N}^t$ is true for every number and a proof of $\mathbb{N}^t\ n$ could theoretically be given to vec^t, such additional arguments are not feasible in practice. For one, the generation of the proof functions becomes much harder as involved dependent elimination is needed for the proof function. And secondly, the type of the term $\mathbb{N}^t\ n$ depends on the type of the argument of the container type in contrast to the type arguments like X that always have a type $\mathbb{T}$. This type is not annotated in the term representation of the argument; and thus, the generation would be very difficult to implement.[2] For $vec\ T\ n$ the hypothesis would be $vec^t\ T\ P\ n\ ?$ where the question mark is a proof of $\mathbb{N}^t\ n$ dependent on n.

To generate the assumptions needed for non type arguments, knowledge about all types in the arguments of the constructors is required. Therefore, these types would need to be collected and quoted with MetaCoq Programs. Additionally, the bookkeeping would be tedious because the position of the arguments would need to be associated with their types and additional predicates. Moreover, it would no longer be possible to automatically generate the proof functions.

2 The type inference function [30] might be able to infer the type.

$$
\begin{aligned}
\mathrm{vec}^{\mathrm{t}} &: \forall (X : \mathbb{T})\ (P_X : X \to \mathbb{T})\ (n : \mathbb{N})\ (P_n : \mathbb{N}^{\mathrm{t}}\ n).\ \mathrm{vec}\ X\ n \to \mathbb{T} \\
\mathrm{nil}_{\mathrm{vec}} &: \mathrm{vec}^{\mathrm{t}}\ X\ P_X\ 0\ 0^{\mathrm{t}}\ (\mathrm{nil}_{\mathrm{vec}}\ X) \\
\mathrm{cons}_{\mathrm{vec}} &: \forall (x : X)\ (H_x : P_X\ x)\ (n : \mathbb{N})\ (H_n : \mathbb{N}^{\mathrm{t}}\ n)\ (H : \mathrm{vec}\ X\ n). \\
&\qquad \mathrm{vec}^{\mathrm{t}}\ X\ P_X\ n\ H_n\ H \to \mathrm{vec}^{\mathrm{t}}\ X\ P_X\ (S\ n)\ (S^{\mathrm{t}}\ n\ n^{\mathrm{t}})\ (\mathrm{cons}_{\mathrm{vec}}\ X\ x\ n\ H)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{vec}_2^{\mathrm{t}} &: \forall (X : \mathbb{T})\ (P_X : X \to \mathbb{T})\ (n : \mathbb{N}).\ \mathrm{vec}\ X\ n \to \mathbb{T} \\
\mathrm{nil}_{\mathrm{vec}}^{\mathrm{t}} &: \mathrm{vec}_2^{\mathrm{t}}\ X\ P_X\ 0\ (\mathrm{nil}_{\mathrm{vec}}\ X) \\
\mathrm{cons}_{\mathrm{vec}}^{\mathrm{t}} &: \forall (x : X)\ (H_x : P_X\ x)\ (n : \mathbb{N})\ (H : \mathrm{vec}\ X\ n). \\
&\qquad \mathrm{vec}_2^{\mathrm{t}}\ X\ P_X\ n\ H \to \mathrm{vec}_2^{\mathrm{t}}\ X\ P_X\ (S\ n)\ (\mathrm{cons}_{\mathrm{vec}}\ X\ x\ n\ H)
\end{aligned}
$$

Figure 6.4: *The parametricity translation* $\mathrm{vec}^{\mathrm{t}}$ *is used to derive the type* $\mathrm{vec}_2^{\mathrm{t}}$ *that can be used as assumption function. All additional arguments, added by the parametricity translations, that do not correspond to type arguments are removed. Therefore, the arguments* P_n *in* $\mathrm{vec}^{\mathrm{t}}$ *and* H_n *in* $\mathrm{cons}_{\mathrm{vec}}$ *are removed together with the instantiations of* P_n*. This procedure makes the usage of* $\mathrm{vec}_2^{\mathrm{t}}$ *much easier as one does not need to care about the proofs of* $\mathbb{N}^{\mathrm{t}}$ *involving the constructors* O^{t} *and* S^{t}*.*

Although the assumption function cannot be the unary parametricity translation directly, the translation can be used to derive the assumption function.

For the container type X the parametricity translation X^{t} is derived first. Then all freshly added parameters that do not correspond to a type argument are removed as these can not contribute to elements in the container type and therefore are not important for induction hypotheses. The procedure is repeated for all inductive bodies in the mutual inductive definition and for all of their constructors. Afterwards, a new inductive type is generated (see Figure 6.4).

Lastly, the proof function type is computed from the type of X and an obligation [28] prompting the user to provide the proof is opened. After the obligation is solved, the container together with the assumption and proof function is added in the registered database.

As already mentioned in Section 6.3, the proof function is usually proved by induction on the instance followed by constructor calls and application of assumptions. Therefore, an ltac tactic [9] is provided performing induction on the innermost argument of quantifications. This tactic is set as the obligation tactic of Coq to automatically solve the obligation for the proof function.

6.4.4 Flags

To toggle the generation of nested induction hypotheses we implemented a flag system similar to the Coq flags with `Set` and `Unset` in MetaCoq.

A type class `Class mode (s: string) := state: bool` that takes a string flag as argument is used to represent the flags. To set or unset a flag `f` one can simply add an instance of `mode f` with the desired value for state. The flag can be changed because new instances overwrite old ones and therefore only the newest instance is used. Accompanying the type class the functions and notations to get, set, and unset flags were also implemented.

With the flag system it is possible to switch the generation of induction hypotheses for nested argument on and off with `MetaCoq Run Set Nested Inductives` and `MetaCoq Run Unset Nested Inductives`.

6.5 Usage example

In this section we will discuss the application of the plugin from the view of an end-user.

The plugin can be included with a single import. The obligation tactic is overwritten to generate proof functions automatically.

```
Require Import MetaCoq.Induction.MetaCoqInductionPrinciples.
```

The plugin can be invoked similar to the `Scheme` command in Coq with `MetaCoq Run` as prefix. It subsumes the functionality of the `Scheme` command and therefore can also derive case analysis and induction principles for types without nested recursion.

```
MetaCoq Run Scheme Elimination for ℕ.
Check ℕ_case_MC.
(* ∀ P, P 0 → (∀ n, p (S n)) → ∀ x, p x *)
MetaCoq Run Scheme Induction for ℕ.
Check ℕ_ind_MC.
(* ∀ P, P 0 → (∀ n, p n → p (S n)) → ∀ x, p x *)

MetaCoq Run Scheme list_induct := Induction for list.
Check list_induct.
(* ∀ X p, p [] → (∀ x xs, p xs → p (x::xs)) → ∀ inst, p inst *)

MetaCoq Run Scheme vec_induct := Induction for VectorDef.t.
Check vec_induct.
(* ∀ A (p : ∀ H : ℕ, vec A H → 𝕋),
   p 0 (nil A) →
   (∀ (h : A) (n : ℕ) (H : vec A n),
    p n H → p (S n) (cons A h n H)) →
```

```
   ∀ (H : ℕ) (inst : vec A H), p H inst *)
```

With the `Set` and `Unset` command the user can toggle whether induction hypotheses for nested recursive arguments should be generated. With nested induction disabled the generated principle is the same as the one from Coq.

```
MetaCoq Run Unset Nested Inductives.

Inductive rtree A : 𝕋 :=
| Leaf' (a : A)
| Node' (l : list (rtree A)).

MetaCoq Run Scheme rtree_induct := Induction for rtree.
Check rtree_induct.
(* prints the induction lemma Coq would generate for rtee' *)
(* ∀ (A : 𝕋) (p : rtree A → 𝕋),
   (∀ a : A, p (Leaf' A a)) →
   (∀ l : list (rtree A), p (Node' A l)) →
   ∀ inst : rtree A, p inst
*)
```

If the nested induction hypotheses are enabled, the generated principle includes a hypothesis for the direct-subtree list of rose trees.

```
MetaCoq Run Set Nested Inductives.
(* activate generation of induction hypotheses for nested types *)

MetaCoq Run Scheme rtree_induct' := Induction for rtree.
Check rtree_induct'.
(* prints the right induction principle for rtree *)
(* ∀ (A : 𝕋) (p : rtree A → 𝕋),
   (∀ a : A, p (Leaf' A a)) →
   (∀ l : list (rtree A),
    list^t (rtree A) p l →
   p (Node' A l)) → ∀ inst : rtree A, p inst
*)
```

The plugin can even be used to generate an induction principle for PCUIC terms (see Section 3.1). The induction hypotheses for nested recursion are generated for the types `list term`, `list (nat×term)` and `mfixpoint term` for the term type.

```
From MetaCoq.PCUIC Require Import PCUICAst.
MetaCoq Run Scheme term_induct := Induction for term.
```

For new container types like `list'` in the following example, the assumption and proof functions are not in the database. Therefore, no induction hypothesis for the

nested recursive argument is generated and the user is alerted that `list'` is not a registered container and informed how to add `list'` to the container database.

```
Inductive list' X : 𝕋 :=
| nil' : list' X
| cons' : X → list' X → list' X.

Inductive rtree' A : ℕ → 𝕋 :=
| Leaf'' (a : A) : rtree' A 0
| Node'' (n : ℕ) (l : list' (rtree' A n)) : rtree' A (S n).

MetaCoq Run Scheme rtree'_induct := Induction for rtree'.
Check rtree'_induct.
(* prints the induction principle Coq would generate for rtree' *)
```

After the registration of `list'` to the database with the `Derive Container` command, the correct induction principle for `rtree'` is generated.

```
MetaCoq Run Derive Container for list'.
(* adds list' to the database for container types *)

MetaCoq Run Scheme rtree'_induct' := Induction for rtree'.
Check rtree'_induct'.
(* ∀ (A : 𝕋) (p : rtree' A → 𝕋),
   (∀ a : A, p (Leaf'' A a)) →
   (∀ l : list' (rtree' A),
    list't (rtree' A) p l →
   p (Node'' A l)) → ∀ inst : rtree' A, p inst
*)
```

6.6 Remarks

With nested induction we extended the plugin beyond the capabilities of Coq's automatically derived induction schemes. The question how to derive induction principles for nested inductive types has been often asked both in published work [5, 14] and on the Coq club mailing list [3] with the answer that the induction principles have to be constructed by hand and require creativity for other containers than `list`. Only recently was the unary parametricity translation as underlying principle discovered [33, 19]. The generation of these induction principles can be done with our plugin.

One can see induction hypotheses with guarded and nested induction as generalized induction on all functors, container types, as well as quantifications.

3 Benoît Viguier, 07.02.2016: How to prove an inductive property on trees
Ethan Aubin, 18.09.2007: rose tree equality

The reason why also definitions are allowed in nested induction is because they behave similarly to inductive types and can also be polymorphic and therefore act as container types. An example seen in Section 3.1 is `mfixpoint` in the `tFix` constructor of `term` which is defined as `mfixpoint X = list (def X)`. The assumption and proof functions are simply the composition of the ones from `def` and `list`.

In the current state of MetaCoq it is necessary to compute the type of the assumption function separately from the parametricity translation as the unquoting command has difficulties adding new universes to the environment. Moreover, the practical application for complicated types is inhibited by problems with the parametricity translation which cannot unquote types involving container types. Therefore, one is not able to automatically derive the assumption function for `roseTree` or `All` and has to define it manually if these types should be used as a container.

Chapter 7

Correctness

Besides modelling the syntax of Coq terms, MetaCoq also provides conversion and typing relations, as explained in Section 3.1.1. Recall that MetaCoq plugins are Coq functions of type $\mathtt{term} \to \mathtt{term}$. Therefore, one is able to state and prove properties of plugins.

The correctness of plugins is proven to ensure that the plugin behaves as expected. The correctness result might be that wanted outcomes are produced or simply that the plugin never fails to generate an output.

The system of Coq consists of the kernel and the remaining code. The kernel is relatively small and needs to be trusted. It is the core of Coq's implementation of dependent type theory and performs all type checks. Therefore, the kernel enforces guarantees that no ill-typed terms are allowed for example. One can add wrong assumptions with axioms to Coq but these axioms are transparent to the users and can be inspected using commands.

The majority of Coq's code, including plugins, is untrusted and the results of plugins for example are checked by the kernel. Therefore, a wrong plugin cannot introduce inconsistencies to Coq. But nevertheless the correctness of plugins strengthens the trust in the plugin and is a bonus.

There are multiple ways to define correctness for the case analysis and induction plugin.

The simplest specification is that the generated eliminator has a type. For natural numbers with the generated term of the eliminator $\hat{\mathsf{E}}_{\mathbb{N}}$ this specification is

$$\exists(\mathsf{T} : \mathtt{term}).\ \Sigma; \Gamma \vdash \hat{\mathsf{E}}_{\mathbb{N}} : \mathsf{T}$$

The specification with typing guarantees that the output of the plugin is well-typed and therefore it never fails to generate a valid principle. One needs to only trust MetaCoq to accept the guarantees of this correctness statement. But on the other

hand the statement is very weak as it does not provide guarantees about what the generated term means.

A more sophisticated specification is that the type is the expected induction principle for the given type. For natural numbers this judgement is

$$\Sigma; \Gamma \vdash \hat{E}_{\mathbb{N}} : \langle \forall (P : \mathbb{N} \to \mathbb{T}).\ P\ O \to (\forall (m : \mathbb{N}).\ P\ m \to P\ (S\ m)) \to \forall (n : \mathbb{N}).\ P\ n \rangle$$

The brackets $\langle t \rangle$ are used to indicate that the quoted representation of t is used. This specification is more complicated than the previous one. The statement guarantees that the computed eliminator really performs a case analysis and provides a valid eliminator. Therefore, this specification proves soundness of the plugin. To get the guarantees, one has to trust the function computing the resulting type to be correct. As this function is relatively short it is doable to read and understand the function. Additionally, this specification also has the guarantees of the previous one.

The most complicated but also strongest specification is the functional correctness with respect to reductions. This correctness statement can be expressed using defining equations. For the natural numbers the equations are the following:

$$\begin{aligned} E_{\mathbb{N}}\ P\ H_O\ H_S\ O &:= H_O \\ E_{\mathbb{N}}\ P\ H_O\ H_S\ (S\ m) &:= H_S\ m\ (E_{\mathbb{N}}\ P\ H_O\ H_S\ m) \end{aligned}$$

This specification is more complicated and therefore difficult to state formally. The statement guarantees that the eliminator behaves correctly and does exactly what a user expects the eliminator to do. Therefore, the specification is nearly the same as the actual implementation. The specification is as complicated as the plugin itself. Thus, there is no gain in proving the plugin correct under this specification as the specification is as complicated as the plugin to prove correct. Therefore, this specification has the strongest guarantees but also needs much trust and is very complicated to state.

In this thesis we chose the second specification as correctness statement for the plugin as it gives strong enough guarantees for most use cases and needs only little trust. Additionally, the second specification is even simpler to prove than the first one because one does not need to invent structure during the proof as would be the case for the existence of a type.

The current development of the correctness statement contains unfinished lemmas. The unfinished lemmas are on a technical side. The proof was very difficult due to numerous reasons.

The underlying type theory of Coq has complicated typing rules as can be seen in the rule for dependent matches (see Section 3.1.2). Even on paper and in a non-abstract setting like the concrete typing judgement for $E_l e$ the proof is complicated.

One needs the same hierarchy of lemmas and it is infeasible to even prove that the eliminator of less or equal is correct.

The MetaCoq implementation of Coq's semantics is not ideal for proofs that terms type-check. For example the typing rules are defined using functions instead of predicates and involve tail-recursive functions like `rev` and `fold_left` which make proofs difficult.

Lastly, the typing proofs have to be done on a low level as no abstraction layers have been found so far for the semantics provided by MetaCoq.

7.1 Correctness statement

On a high level view the correctness statement says that the generated eliminator type-checks with a type computed for the induction principle. Therefore, the plugin is guaranteed to produce well-typed terms for well-formed inductive types.

To state the correctness in MetaCoq we use the `createElim` function that is the main part of the plugin and computes the eliminator, as well as a function `createElimType` that computes the type of the induction principle for a given inductive type. The requirement for correct operation is a well-formed inductive type `ind` in a mutual inductive declaration.

The concrete correctness statement is the following:

```
wf Σ Γ →
on_ind_body mind mind_body ind ind_body →
declared_inductive Σ mind_body ind ind_body →
  ∀ T t name,
  createElim ind = Some (t, name) →
  createElimType ind = Some T →
  Σ ; Γ ⊢ t : T.
```

This statement means that under a well-formed local environment Γ in the global environment Σ the computed eliminator `t` is typed with the type `T` for a well-formed inductive type with a mutual inductive declaration `mind_body` and single inductive declaration `ind_body`.

The indirection with `Some` is needed to account for the extraction of the inner body in the mutual inductive declaration. It is proven that the functions always return `Some` for a well-formed inductive type. Therefore, the direct formulation would be possible to state but is much longer and less readable because the functions creating the eliminator and it's type would need to be unfolded.

The `createElimType` function has the same structure as `createElim` up to the fixed point where all λ-abstractions are replaced by $\forall$-quantification (see Figure 7.1). Instead of the fixed point function a quantification over the indices and instance of

```
E⩽ := λ (n:ℕ)
(p: ∀ (m:ℕ). n ⩽ m → ℙ)
(H_leB: p n (leB n))
(H_leS: ∀ (m:ℕ) (h:n ⩽ m).
    p (S m) (leS n m h)).
fix f (m:ℕ) (x:n ⩽ m) : p m x :=
match x as y return p m y with
| leB ⇒ H_leB
| leS m h ⇒ H_leS m h
end
```

```
E⩽ : ∀ (n:ℕ)
(p: ∀ m:ℕ. n ⩽ m → ℙ)
(H_leB: p n (leB n))
(H_leS: ∀ (m:ℕ) (h:n ⩽ m).
    p (S m) (leS n m h)).
∀ (m:ℕ) (x:n ⩽ m). p m x
```

Figure 7.1: *On the left is the eliminator for less or equal with the type on the right. The type is exactly the principle where the λ-abstractions are replaced by ∀-quantifications up to the fixed point. The whole fixed point is replaced by quantifications over the arguments with the predicate as conclusion.*

the inductive type is constructed with the applied predicate as body. Therefore, the `createElimType` function exactly generates the term representing the induction principle type whereas `createElim` generates the eliminator. Due to the simpler structure `createElimType` is much shorter.

7.2 Proof structure

The proof of the correctness statement first unfolds the functions and resolves the option results from `createElim` and `createElimType`. This is possible because both functions are guaranteed to return `Some t` for a term `t`. Afterwards, the proof aligns with the structure of the eliminator. The steps are as follows:

1. introduction of parameters and the predicate `P`,
2. typing of the predicate,
3. introduction of all cases,
4. resolution of the fixed point,
5. proof that the environment is well-formed,
6. simplification of the term and type,
7. introduction of the indices and instance,
8. typing of the instance,
9. simplification steps for the case analysis,
10. η-Conversion,[1]
11. case analysis typing

The first three steps introduce the λ-abstractions from the term to the context. Those steps are mostly straightforward as the computed type is exactly the term of the

1 The conversion rules were taken from PCUIC.

eliminator where the λ-abstractions are replaced by ∀-quantifications. The predicate type has to be proven to be a valid type as it is introduced directly with the typing rule of λ-abstractions (see Section 3.1.2) which requires that the type of an argument is a valid type itself.

Step four to six are concerned with the well-formedness of the environments up to and including the fixed point. Afterwards, the arguments of the fixed point are introduced. Lastly, in steps nine to eleven the term and type are transformed to fit the typing rule for case analysis (see Section 3.1.2).

The most complicated steps are the last three steps. The other steps also take a few hundred lines in the proof but could be shortened with a rewriting of the plugin or are sub-tasks of the last step. Therefore, one can see that the typing overall is difficult but most problems are posed by dependent matches.

7.2.1 Auxiliary lemmas

As most steps are complicated and need a hierarchy of auxiliary lemmas, the steps are relocated into separate lemmas that are applied in a chain in the correctness proof. Especially the last step for case analysis typing is complicated. Therefore, six auxiliary lemmas for the assumptions of the typing rule are needed.

To make the individual lemmas a bit easier and more modular, more auxiliary lemmas are introduced:

- function specification and usages of functions,
- lemmas about the MetaCoq framework and constructions like inductive types and definitions,
- typing derivations of parts of the eliminator construction and the types used in the other lemmas,
- proofs that the environments are well-formed and objects are well-formed in relation to the environments,
- and equations for simplification.

During the proof, it is required at many positions to prove that some terms have a type, the environment is well-formed, and new assumptions are well-formed in the previous environment. Those lemmas are built on top of the previous statements and are extended piece by piece for new environments. The lemmas also mutually depend on each other because proofs for well-formed environments need the typing derivation of the terms in the environment and typing derivations need the proofs that the environment is well-formed.

7.2.2 Lemma hierarchy

The structure of the groups can be seen with the well-formedness lemmas for environments. Every time the environment is updated the well-formedness needs to be

extended to accommodate the new assumptions. Additionally, the well-formedness proofs depend on the well-formedness of sub-environments and therefore build a hierarchy.

The well-formedness of environments is stated with the `wf_local` predicate(see Section 3.1.1). An environment is a list of assumptions. The predicate `wf_local Σ Γ` states that every assumption in the environment Γ has a valid universe and the body of the assumption is typed with the type of the assumption. Therefore, `wf_local` expresses that all assumptions in the environment are well-typed.

```
wfLocal: wf_local Σ Γ
wfLocalIndicesMin: wf_local Σ (params,indices )
wfLocalRelIndicesMin: wf_local_rel Σ (Γ, params ) indices
wfParams : wf_local_rel Σ Γ params
wfLocalParamsMinMin: wf_local Σ params
wfLocalParamsMin: wf_local Σ (Γ, params )
wfLocalParams: wf_local Σ (Γ, uniformParams )
wfLocalRelIndices2: wf_local_rel Σ (Γ, uniformParams )
                                        (nonUniformParams, indices )

wfLocalParamsP: wf_local Σ (Γ, params, p )
wfLocalRelCshapeArgs newCase: wf_local_rel Σ (Γ, params, p, cases ) newCase
wfLocalRelCases: wf_local_rel Σ (Γ, params, p ) cases
wfLocalQuantifyCases: wf_local Σ (Γ, params, p, cases )
wfLocalRelIndices: wf_local_rel Σ (Γ, params, p, cases ) (lifted indices )
wfLocalFixEnv: wf_local Σ (Γ, params, p, cases, lifted indices, inst )
wfLocalRelCasesF: wf_local_rel Σ (Γ, params, p, cases ) [f]
wfLocalCaseEnvEnv: wf_local Σ (Γ, params, p, cases, f, lifted indices )
wfLocalMatchObjEnv: wf_local Σ (Γ, params, p, cases, f, lifted indices, inst )

wfLocalMatchTypeEnv: wf_local Σ (Γ, params, p, cases, f,
                          lifted indices, inst, lifted indices )
wfLocalIndInst: wf_local Σ (Γ, params, p, cases, f,
                          lifted indices, inst, lifted indices, inst )
```

The first lemma group establishes that the parameters and indices are well-formed under any local environment Γ and global environment Σ. Afterwards, the lemmas are extended according to the terms introduced during the proof. An assumption here is that only uniform parameters occur.

The lemma `wfLocalRelCshapeArgs` is an auxiliary lemma used in an induction step to prove that all cases can be introduced and lead to a well-formed environment.

The last two lemmas are needed to prove that the type of the inner case analysis is well-typed and has a well-formed environment.

There are similar lemma groups for the `typing` and `typing_spine` predicate.

7.3 Assumptions

We use the axiom that the fixed point used in the plugin always terminates, because the fixpoint guard predicate is not implemented in MetaCoq and was therefore assumed. This assumption is valid as the fixed point basically behaves like a λ-abstraction in the case analysis part of the plugin, because it is never called. With the fixed point formulation of the plugin, this axiom is needed as the typing only could be proven if the complete semantic of fixed point guards would be implemented in MetaCoq.

We use a fixed point in the plugin as this way the already written plugin can be used and no new plugin function has to be written just for the correctness proof. The eliminator plugin is called in a way such that we restrict it to case analysis. If the primary goal of the thesis would be the correctness proof, the procedure could be improved. The plugin could be rewritten for the correctness proof. For example the fixed point could be replaced with a λ-abstraction and the terms could be constructed to inline more with the type constructors. We conjecture that the rewriting would ease some technical parts but overall the correctness proof would still remain very large and difficult.

Properties of inductive definitions that are not stated explicitly in MetaCoq were assumed. Such properties include that the assumptions in parameters and indices have empty bodies. Also, lemmas ranging over universe sorts and η-conversion are admitted due to ongoing changes in MetaCoq.

Some of the basic lemmas and function specifications are admitted due to time constraints.

The assumption was made that all parameters are uniform. This assumption is valid for case analysis principles as all non-uniform parameters can be treated as uniform parameters.

7.4 Difficulties

There were many difficulties both in the ability of MetaCoq to prove typing derivations for large scale terms as well as in the proof details specific to the case analysis plugin.

7.4.1 MetaCoq specific

The typing rules (see Section 3.1.2) use functions like `fold_left`, `mapi` or `rev` which makes proofs, especially inductive proofs, quite difficult. For example the function `build_case_predicate_type ind mdecl idecl params u ps` that is used in the typing rule of `tCase` to compute the type of the return type for dependent matches is unfolded into:

```
(X ←
```

```
match
  instantiate_params_subst
     ( rev ( map ( map_decl (subst_instance_constr u )) (ind_params mdecl ) ) )
    params []
    (subst_instance_constr u (ind_type idecl ))
with
| Some (s, ty ) ⇒ Some ((subst0 s ) ty )
| None ⇒ None
end;
X0 ← destArity [] X;
ret
 ( fold_left
     ( λ (acc : TemplateTerm.term ) (d : context_decl ) ⇒
         mkProd_or_LetIn d acc )
     (X0.1,
      {|
      decl_name := nNamed (ind_name idecl );
      decl_body := None;
      decl_type := mkApps (tInd ind u ) (map (lift0 #| X0.1 | ) params
                          ++to_extended_list X0.1 ) |} )
     (tSort ps ) ) )
```

One can observe that the function is defined with many mappings and reversed arguments.

MetaCoq has many intricate definitions and notations. For example, `cons` and `append` are redefined using definitions that are used in notations. These definitions and notations make it more difficult to apply lemmas and keep an overview over the lemmas and goals.

The ongoing development of MetaCoq refines some lemmas and typing rules and makes some proofs easier on the one hand. But on the other hand, the changes also mean that entire parts of the proof like the case analysis typing and η-conversion are subject to change.

Additionally, there is currently no documentation outlining what functions and lemmas exist and how to use them. The missing documentation also made it quite hard to find the implicitly stated properties of inductive types. For example, one would need a statement that guarantees cetain invariants for every inductive type which is unquoted with the command `tmQuoteInductive`. Such invariants should be that the number of parameters are the same as the parameter-count in the inductive body or that the inductive type can be applied to the parameters and indices resulting in a well-typed term.

Some lemmas are only available in PCUIC but not in TemplateCoq. These lemmas were copied and admitted if they were needed.

In theory the typing proofs should be quite schematic. Therefore, one would hope for good automation tactics to assist in the proofs. Currently, there is a tactic for inference of closed terms but this is not applicable in the context of this plugin as the proofs are on abstract terms and involve variables bound in the environment.

7.4.2 Project specific

The bookkeeping of liftings of de Bruijn indices, especially for the indices of the inductive type, was quite difficult to get correct. We will denote the lifting of a term t by n for de Bruijn indices larger than m by $\uparrow^n_m t$.

The application of lemmas was difficult due to intricate definitions, notations, and the failure of unification.

As an example one can take the following code section:

$$
\begin{aligned}
&\Sigma; \Gamma, \texttt{params}, \texttt{p}, \texttt{cases}, \uparrow^{1+|\text{ctors}|} \texttt{indices} \vdash \\
&\texttt{ind} \\
&(\uparrow^{1+|\text{ctors}|}_{|\text{indices}|} \uparrow^{|\text{indices}|} \texttt{mkRel params++} \\
&\uparrow^{1+|\text{ctors}|}_{|\text{indices}|} \texttt{mkRel indices}) : \\
&\texttt{tSort u}
\end{aligned}
$$

The assumption that is already proven is that the inductive type applied with parameters and indices has a type `u`, stated with the `typing_spine` predicate.

$$
\begin{aligned}
&\Sigma; \Gamma, \texttt{params}, \texttt{p}, \texttt{cases}, \texttt{f}, \uparrow^{2+|\text{ctors}|} \texttt{indices} \vdash \\
&\uparrow \texttt{ind} \\
&(\uparrow^{2+|\text{ctors}|+|\text{indices}|}_{|\text{indices}|} \texttt{mkRel params++} \\
&\uparrow^{1+|\text{ctors}|}_{|\text{indices}|} \texttt{mkRel indices}) : \\
&\texttt{tSort ?s}
\end{aligned}
$$

The goal is to prove that the inductive type applied with parameters and indices still has a type if an argument f is added to the environment and everything is lifted accordingly.

$$
\begin{aligned}
&\Sigma; \Gamma, xs \vdash t\ x : T \rightarrow \\
&\Sigma; \Gamma, ys, \uparrow^{|ys|} xs \vdash t\ \left(\uparrow^{|ys|}_{|xs|} x\right) : \uparrow^{|ys|}_{|xs|} T
\end{aligned}
$$

```
apply typing_spine_lifting.
apply H.
```

The idea would be to state a lemma `typing_spine_lifting` stating that one can add new assumptions to the environment and the typing stays the same except for liftings.

In practice it is more difficult because one first has to make the two terms syntactically equal by moving liftings to the top level, unfolding notations, simplifying of the terms, and transforming applications of functions. Therefore, the application of the lemma is 80 lines long.

Another difficulty is that the goals change frequently in appearance such that it is not possible to hide parts in definitions effectively. Therefore, the goals often were quite large ranging from 150 to 500 lines in length. The long goals as well as the many intricate definitions prevent the `Search` vernacular of Coq to work properly. Altogether, this dysfunction led to more work when lemmas are applied and sometimes caused duplicate proofs of two statements that were basically the same but with different notations.

7.4.3 Tactics

New tactics are added to make some parts of the proofs easier. The tactics mainly group tactics that are often used.

The most important new tactics help to prove the equalities of typing statements of the form $\Sigma_1;\Gamma_1 \vdash t_1 : T_1 = \Sigma_2;\Gamma_2 \vdash t_2 : T_2$ and equalities of well-formedness of environments like $wf\_local\ \Sigma_1\ \Gamma_1 = wf\_local\ \Sigma_2\ \Gamma_2$. These two tactics are specific applications of the `f_equal` tactic in cases where `f_equal` fails. Another tactic helps to prove $A \to B$ where A and B are the same and only rewriting is required.

Other tactics perform reordering of lists and case analysis with subsequent congruence and lemma calls.

7.5 Remarks

The approach of using a specific function to compute the type instead of only stating that there is a type for the generated eliminator was originally not motivated by the stronger guarantees. Instead this approach was chosen as it is much easier to prove the concrete typing derivation without the need to instantiate existentials or infer the instantiation.

The difficulties of the proof seem to not only lie in the abstract context for all inductive types with arbitrary parameters, non-uniform parameters, and indices, but also in the complexity of the Coq's type system. We showed this with an attempt to prove typing for a concrete eliminator like the one of less or equal without the

type inference tactic resulting in several hundred lines of code with similar auxiliary lemma structures as the abstract proof. Therefore, the correctness proofs are even on paper infeasible.

To be able to prove large scale typing derivations it is necessary for MetaCoq to have good automation tactics and hint databases to unify and apply typing constructors. Additionally, the terms need to be constructed with knowledge of how the semantics of MetaCoq is constructed in order to take advantage of the typing system. Therefore, an extensive documentation is needed.

Chapter 8

Related work

Induction principles for nested inductive types are often used but are commonly derived manually using known tricks [5]. Often special principles only suitable for lists are written or the whole container type is inlined to perform mutual induction.

Only recently attempts were made to generalize induction to arbitrary container types [18, 19] and develop plugins for the generation of induction principles for nested inductive types, for example by Tassi in Elpi [33]. Johann and Polonsky introduce a formal method to generate induction principles for nested inductive types [18]. They show how all structured data in constructors can be used for induction hypotheses by using the semantic of nested types as fixed points of accessible functors on locally presentable categories. We will look at the other implementations and realizations of induction principles in other proof assistants.

8.1 Coq implementation

The original design of Coq did not support inductive definitions. Instead inductive types were handled by impredicative characterizations which involve higher-order quantifications. These characterization are very complicated to work with and it is not possible to prove simple facts like $0 \neq 1$. Therefore, inductive definitions were added in 1993 [22].

The original implementation of the generation of induction principles is written in OCaml [1].

The problem with this implementation is that it generates too weak induction principles and has no guarantees about correctness. Especially, the generation could diverge without generating a principle. The MetaCoq plugin is a Coq function, and therefore, is guaranteed to terminate because all Coq functions are total.

In our view the OCaml code is more complicated than the MetaCoq plugin code.

1 `https://github.com/coq/coq/blob/fb7292570380e0490d7c74db1718725149996ffd/pretyping/indrec.ml`

With around 600 lines the OCaml code is also longer and has nearly no comments. Our plugin is documented and is 317 lines long.

8.2 Elpi

Elpi is the embeddable λProlog interpreter. Therefore, it is an implementation of a dialect of λProlog, a language that can be used to manipulate syntax trees. Prolog is a logic programming language where programs are expressed as constraints and relations. λProlog is an extension to Prolog with higher-order quantification, polymorphic types, higher-order unification, and simple typed λ-terms. There is an interface to use Elpi to implement commands and tactics.

Tassi implements induction principles for nested inductive types in Elpi [33] using the unary parametricity translation. Our work is influenced by Tassi and we also use the idea of parametricity for assumption functions in induction principles. Tassi implemented induction principles by an extension of the induction principles to include predicates for elements of other types in the principle. For lists, for example, a predicate for the type A is added:

$$
\begin{aligned}
&\forall (A : \mathbb{T})\ (P_A : A \to \mathbb{T})\ (P : \mathsf{list}\ A \to \mathbb{T}).\\
&P\ [\,] \to\\
&(\forall (a : A)\ (l : \mathsf{list}\ A).\ P_A\ a \to P\ l \to P\ (a :: l)) \to\\
&\forall (l : \mathsf{list}\ A).\ \mathsf{is\_list}\ A\ P_A\ l \to P\ l
\end{aligned}
$$

In the case for cons, an additional assumption $P_A\ a$ is added. Additionally, the predicate $\mathsf{is\_list}\ A\ P_A\ l$ has to be fulfilled in order to provide the new assumptions. Here, $\mathsf{is\_list}$ is the unary parametricity translation of the list type.

The unary parametricity translation of container types is used for assumptions of nested recursive arguments as can be seen for rose trees with labeled leaves:

$$\mathsf{rtree}\ (A : \mathbb{T}) := \mathsf{Leaf}\ (a : A) \mid \mathsf{Node}\ (l : \mathsf{list}\ (\mathsf{rtree}\ A))$$

$$
\begin{aligned}
&\forall (A : \mathbb{T})\ (P_A : A \to \mathbb{T})\ (P : \mathsf{rtree}\ A \to \mathbb{T}).\\
&(\forall (a : A).\ P_A\ a \to P\ (\mathsf{Leaf}\ A\ a)) \to\\
&(\forall (l : \mathsf{list}\ (\mathsf{rtree}\ A)).\ \mathsf{is\_list}\ (\mathsf{rtree}\ A)\ P\ l \to P\ (\mathsf{Node}\ A\ l)) \to\\
&\forall (t : \mathsf{rtree}\ A).\ \mathsf{is\_rtree}\ A\ P_A\ t \to P\ t
\end{aligned}
$$

Again, a predicate P_A for the leaf elements of type A is added, and the $\mathsf{is\_rtree}$ predicate in the conclusion. The induction hypothesis for l is stated using the unary parametricity translation of list, and therefore, coincides with the induction hypothesis generated by our plugin.

In our tests we found some types where the plugin from Tassi could not generated the correct principles. For each problem we will provide a problematic example together with an explanation.

$$\mathsf{DN} ::= \mathsf{C}(\mathsf{xs} : \mathsf{list}(\mathsf{list}(\mathsf{list}\ \mathsf{DN})))$$

When deriving the principle for deeply nested arguments like the one in DN the Elpi plugin takes several minutes[2] to compute the principle.

$$\frac{}{\mathsf{Zero}\ 0}\ \mathsf{C_Z}$$

For types with indices like Zero the derivation fails to generate some auxiliary functions, namely Zero_is_Zero_full and Zero_is_Zero_trivial, resulting in unreadable inductive principle although they seem to be correct. For Zero the generated induction principle is

$$\begin{aligned}
&\forall \mathsf{P} : \forall _\mathsf{elpi_ctx_entry_1_} : \mathbb{N},\\
&\quad \mathsf{nat_is_nat}\ _\mathsf{elpi_ctx_entry_1_} \to \mathsf{Zero}\ _\mathsf{elpi_ctx_entry_1_} \to \mathbb{T},\\
&\mathsf{P}\ 0\ \mathsf{nat_is_O}\ \mathsf{C_Z} \to\\
&\forall (_\mathsf{elpi_ctx_entry_1_} : \mathbb{N})(\mathsf{P_} : \mathsf{nat_is_nat}\ _\mathsf{elpi_ctx_entry_1_})\\
&\quad (s_1 : \mathsf{Zero}\ _\mathsf{elpi_ctx_entry_1_}),\\
&\mathsf{Zero_is_Zero}\ _\mathsf{elpi_ctx_entry_1_}\mathsf{P_}\ s_1 \to \mathsf{P}\ _\mathsf{elpi_ctx_entry_1_}\ \mathsf{P_}\ s_1
\end{aligned}$$

The principle is basically the same as the one generated by Coq with additional assumptions for n:

$$\forall (\mathsf{P} : \forall n.\ \mathsf{Zero}\ n \to \mathbb{T}).\ \mathsf{P}\ 0\ \mathsf{C_Z} \to \forall n\ (z : \mathsf{Zero}\ n).\ \mathsf{P}\ n\ z$$

$$\mathsf{typeTree} ::= \mathsf{Ctt}\ (\mathsf{X} : \mathbb{T})\ (\mathit{vec}\ \mathsf{X}\ 0 \times \mathsf{typeTree})$$

For types that quantify over types in the constructors like typeTree the plugin fails to generate the induction principle.

$$\mathsf{nestGuard} ::= \mathsf{Cng}\ (\mathsf{list}\ (\mathbb{N} \to \mathsf{nestGuard}))$$

While outer guarded induction poses no problems, the plugin cannot generate induction principles when the guarded recursion is the type argument in nested induction.

2 Around six minutes in our test.

For inductive predicates, types of type $\ldots \to \mathbb{P}$, the plugin fails to generate the induction principle and sometimes even terminates the derivation process.

Lastly, the implementation in Elpi uses the unary parametricity translation directly and therefore adds additional predicates for all parameters as well as assumptions for every argument. For non-container types like $\mathbb{N}$ these arguments are equivalent to $\top$ and therefore not useful. The additional predicates prevent a direct application with the `elim` or `induction` tactic.

In comparison to our MetaCoq implementation a reader needs knowledge of Elpi to understand the code[3]. For our MetaCoq plugin this is less an issue as the plugin itself is a function in Coq. Therefore, a Coq user is already familiar with the syntax. The code is similar in length but has the advantage that no calculations of de Bruijn indices are necessary. On the other hand, one cannot prove correctness.

8.3 Other proof assistants

As already mentioned, weak induction principles are a common problem. Therefore, other proof assistants implement other solutions to generate induction principles for nested types.

8.3.1 Isabelle

Isabelle [24] has two possibilities to generate induction principles for nested inductive types [4].

The legacy version can be invoked by registering a datatype as an old-style datatype with `datatype_compat`. The principles for old-style datatypes inline the container type and build a mutual inductive principle. For the principle the nested type is viewed as a mutual inductive type. For rose trees the mutual type for the principle would be

$$\begin{aligned} x &: \mathsf{roseTree} := \mathsf{node}\ ys \\ xs, ys &: \mathsf{listRoseTree} := [\,] \mid \mathsf{cons}\ x\ xs \end{aligned}$$

The generated principle is

$$\begin{aligned} &\forall (P_1 : \mathsf{roseTree} \to \mathbb{T})\ (P_2 : \mathsf{listRoseTree} \to \mathbb{T}). \\ &(\forall xs : \mathsf{listRoseTree}.\ P_2\ xs \to P_1\ (\mathsf{node}\ xs)) \to \\ &P_2\ [\,] \to (\forall x : \mathsf{roseTree}. P_1\ x \to \forall xs : \mathsf{listRoseTree}.\ P_2\ xs \to P_2\ (\mathsf{cons}\ x\ xs)) \to \\ &\forall r : \mathsf{roseTree}.\ P_1\ r \end{aligned}$$

The principle takes a predicate P_1 for the rose trees and a predicate P_2 for lists of rose trees stating that P_1 holds for every tree in the list.

3 `https://github.com/LPCIC/coq-elpi/blob/master/derive/induction.elpi`

The generated principle `compat_roseTree.induct` in Isabelle syntax is

$$\begin{aligned}&\left(\bigwedge xs.\ P_2\ xs \Rightarrow P_1\ (node\ xs)\right) \Rightarrow\\ &P_2\ [\,] \Rightarrow \left(\bigwedge y\ ys.\ P_1\ y \Rightarrow P_2\ ys \Rightarrow P_2\ (y\#ys)\right) \Rightarrow\\ &P_1\ ?tree\end{aligned}$$

The induction predicate of new-style datatypes is simpler but does not preserve the structure of the container type.

$$\left(\bigwedge xs.\ \left(\bigwedge t.\ t \in set\ xs \Rightarrow P\ t\right) \Rightarrow P\ (node xs)\right) \Rightarrow P\ ?tree$$

For new-style datatypes Isabelle generates a set function for the container types collecting all elements and states that every element in the set satisfies the predicate P. Therefore, the induction principles for new-style datatypes are a special case of the principles generated by our plugin where the assumption function forgets the structure of the container type. The weak assumption functions with set are not problematic in Isabelle because the induction principles are only used logically and not computationally in Isabelle.

8.3.2 Lean

The Lean proof assistant handles nested inductive type in a similar fashion as the old-style datatypes in Isabelle. But in contrast to Isabelle, where the mutual inductive type is only generated to create the induction principle, Lean's kernel does not support nested inductive types. Whenever a nested type is defined in Lean, it is compiled into a mutual inductive type together with the container types. Therefore, Lean directly converts the definition of rose trees into the following mutual inductive type:

$$\begin{aligned}x : roseTree &:= node\ ys\\ xs, ys : listRoseTree &:= nil \mid cons\ x\ xs\end{aligned}$$

Afterwards, isomorphisms between the included container types and their counterparts are created and used to define the constructors [3]. Therefore, the mutual type is not visible to the user and the constructors as well as the induction principle appear to be the ones of the nested type. For rose trees the isomorphism $f : listRoseTree \to list\ roseTree$ is between the inlined version of lists $listRoseTree$ and $list\ roseTree$:

$$\begin{aligned}f\ nil &:= [\,]\\ f\ (cons\ x\ xs) &:= x :: f\ xs\end{aligned}$$

The constructor node is defined using the inverse isomorphism f^{-1} and an internal constructor $nested.roseTree.node$:

$$\begin{aligned}&\mathsf{def}\ roseTree.node : list\ roseTree \to roseTree :=\\ &\quad \lambda(a : list\ roseTree).\ nested.roseTree.node\ (f^{-1}\ a)\end{aligned}$$

8.3.3 Agda

For Agda there is, as for Coq, no default mechanism to generate induction principles for nested inductive types. But there is a general procedure to derive induction principle for arbitrary types with implementation in Agda [18]. A semantically justification for the induction principles is given using category theory. In this paper Johann and Polonsky even state "[...] the phenomenon of deep induction has not previously even been identified, let alone studied."

The principles involve additional predicates for involved container types like the principles by Tassi [33]. A justification for the added predicates is not only given by the derivation but also in the fact that these predicates allow to prove statements over special instances of the type. In the case of lists this allows to prove statements over lists only containing prime numbers for example. In the following principle Q is the predicate satisfied by all element of the list:

$$\begin{aligned}&\forall(A : \mathbb{T})\ (P : list\ A \to \mathbb{P})\ (Q : A \to \mathbb{P}).\\ &P\ nil \to (\forall(y : A)\ (ys : list\ A).\ Q\ y \to P\ ys \to P\ (cons\ y\ ys)) \to\\ &\forall(xs : list\ A).\ list^t\ A\ Q\ xs \to P\ xs\end{aligned}$$

It seems like the principles of our plugin are able to replicate this use case. For example for the induction principle for lists, one can choose P to be $\Sigma_a\ Q\ a$. With this instantiation, each y in the constructor carries the proof $Q\ x$ and xs carries the proof of $list^t$.

$$\begin{aligned}&\forall(A : \mathbb{T})\ (P : list\ (\Sigma_a\ Q\ a) \to \mathbb{P}).\\ &P\ nil \to (\forall(y : \Sigma_a\ Q\ a)\ (ys : list\ (\Sigma_a\ Q\ a)).\ P\ ys \to P\ (cons\ y\ ys)) \to\\ &\forall(xs : list\ (\Sigma_a\ Q\ a)).\ P\ xs\end{aligned}$$

Currently the theory of Johann and Polonsky does not include dependent types and indexed containers. But the procedure can be applied to non-strictly-positive types that cannot be defined in Coq like this bush type:

$$Bush\ (A : \mathbb{T}) := BNil \mid BCons\ (a : A)\ (h : Bush\ (Bush\ A))$$

8.3.4 More proof assistants

Most other proof assistants have similar methods as the ones discussed here.

In **F*** the induction is handled with the transitive closure of the subterm relation. Therefore, it is possible to recurse under nested occurrences.

Abella performs induction over inductive predicates instead of the inductive type itself. Recursion is permitted on smaller derivations of the predicate. Therefore, one can use nested induction with strong enough predicates.

In **Cedille** types have to be defined with their Church encoding and induction principles are stated by hand. Therefore, no induction principles are derived automatically.

Twelf also uses the subterm relation for termination check and therefore nested recursion can be used.

In **Beluga** induction is also handled by recursive calls instead of induction hypotheses and these recursive calls are checked to obey the subterm relation.

8.4 MetaCoq plugins

MetaCoq has been used before to implement plugins. Example plugins are given in [31] like a plugin to add constructors to an inductive type and a plugin for syntactic translations. The translation plugin is used to define the parametricity translation [1].

Furthermore, MetaCoq was used to define an extraction from Coq terms to weak-call-by-value λ-calculus [12]. In [30] a safe type-checker for Coq was generated using verified extraction.

Chapter 9

Conclusion

We have implemented a plugin in MetaCoq that generates strong induction principles for nested inductive types. To do so, a technique to enhance induction principles with additional assumptions using unary parametricity was introduced.

The generation function for simple induction principles for types where recursive calls are neither guarded nor nested can be called recursively to generate the induction hypotheses for more complex arguments. Therefore, the simple generation of induction principles is extended to arbitrary functors including the special case of implications for guarded induction and the case of container types for nested induction. Only slight modifications are then needed to construct the correct hypotheses from the recursive results.

We also made an attempt to prove the correctness of the plugin. We managed to verify parts of the generation of case analysis principles.

9.1 Plugins in MetaCoq

As we have seen, the MetaCoq project is suited to develop plugins for Coq. The feature to write plugins directly in Coq makes testing easier and simplifies the introduction to plugin development.

The development is on a low-level representation and the functions become more complicated for complex plugins as the programmer has to keep an overview over the de Bruijn indices of the variables. The development of these functions is also difficult due to the lack of documentation of MetaCoq.

Thus, one needs to become familiar with MetaCoq and organize the development, but overall it has become much easier to write new plugins for Coq with MetaCoq compared to OCaml plugins.

9.2 Verification in MetaCoq

Verification with type check of plugins, on the other hand, is not feasible in the current state of MetaCoq. This is rooted in many different factors from which the most prominent are the following: there is little to no automation to prove essential statements like typing in an abstract context. Basic lemmas of core functions are not available or at least not available in TemplateCoq. Therefore, correctness proofs should be done in PCUIC.

We did the proof in TemplateCoq as it is closer to Coq and the quoting and unquoting moves terms between Coq and TemplateCoq. Additionally, the translation from PCUIC to TemplateCoq was only developed in this thesis and therefore was not available at the time when we did the verification.

The problems are amplified by the missing documentation on which functions are already available in MetaCoq, what they mean, and how to use them. Currently, the typing statement is on a very low level with much work left to the user. Therefore, it would be interesting to search for suitable abstraction layers.

Additionally, an explicit statement on guarantees and well-formedness properties, for example of inductive types created by the quoting mechanism, are hard to find. One would expect MetaCoq to give guarantees, for example what statements are satisfied by a well-typed inductive type or term, what properties a quoted term satisfies, and what guarantees hold if typed term is unquoted.

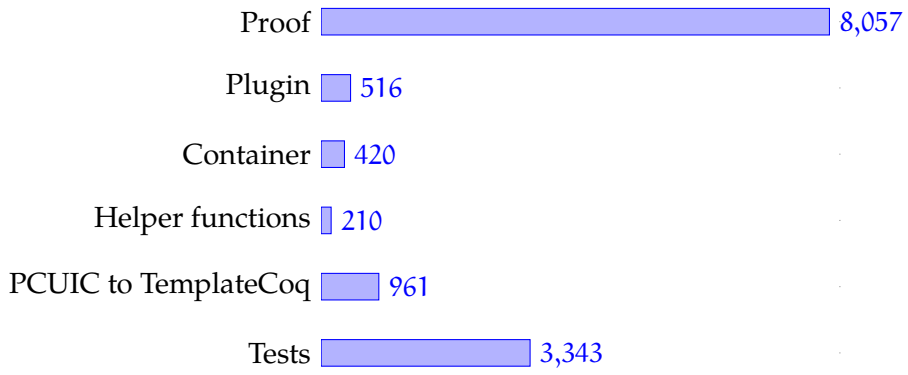
As some parts of the MetaCoq project were adapted from the OCaml implementation, they are designed with a programmer's intent in mind rather than the one of a logician who wants to prove statements. This leads to intricate definitions with unpleasant functions like `rev`, `mapi` and `fold_left` or functional statements instead of inductive predicates. Also, some functions used in the typing statement generate an option result despite being used as a predicate.

In conclusion, verification is very difficult and the model of MetaCoq is not useable for verification in the current state. The difficulty perhaps lies in the complexity of Coq's logic itself.

9.3 Expenses

In this table the length of the development grouped by topics is listed.[1]

1 The line count was estimated using cloc(`https://github.com/AlDanial/cloc`).



In total the development consists of 13507 lines of code. The most time was spent on the correctness proof.

9.4 Future work

The generation of assumption functions and proof functions for container types depend on the parametricity translations. We also needed functions to transform the parametricity translation into a form that is applicable for our use cases.

Currently, there are problems in the parametricity translation of nested inductive types as well as in the unquoting of universes. Therefore, the assumption function for nested inductive container types can not be generated automatically in practice. In theory, the generation could be handled the same way as for simpler container types.

For the proof functions of nested inductive container types the stronger induction principle is needed. The generation of the proof functions therefore needs a database of the induction principles generated by the plugin. To implement such a database, the problem of universe unquoting needs to be solved.

Another line of work is to extend the plugin to mutual inductive types and to add more options, for example using the presented MetaCoq flag mechanism, to control exactly what principles should be generated. An option could be used to control whether the generated principles are dependent.

The proof shown in Chapter 7 states the correctness for the case analysis part of the plugin and contains unfinished lemmas. In the end, one would hope for a complete correctness proof of the plugin. To do so, the proof probably needs to be moved from TemplateCoq to PCUIC to make the reasoning easier. Automation and mathematical abstraction is required to make reasoning about the typing judgement and therefore the mentioned proofs feasible. An option to increase trust in the plugin is to use QuickChick [10] for property-based testing on the eliminator generation

function.

The generation of proof functions uses the unary parametricity translation and syntactically manipulated the resulting type to fit the needs. For a cleaner approach, a dedicated version of unary parametricity should be derived from translation already implemented in MetaCoq.

Finally, the work of this thesis provides the means to write more plugins working on inductive types. Plugins like `show` and `size` functions can use the strong induction principles and extend from there on as they map elements of inductive types to either strings or numbers. Other plugins like equality deciders, finiteness and countability predicates and selectors also benefit from the induction principle and can use the structure of a general mapping of inductive types that was used to write the induction principle plugin.

Appendix A

Appendix

A.1 De Bruijn indices

Terms like $\lambda x.\ \lambda y.\ y\ x$ are easily readable for humans, but are difficult to work with formally. Formally, the equivalence of terms with named variables always needs the notion of α-equivalence. Two terms are α-equivalent if they are the same up to renaming. For example, $\lambda x\ y.\ f\ x\ y$ is the same as $\lambda y\ x.\ f\ y\ x$ up to renaming.

A represention of terms that works around these problems are de Bruijn terms[8]. De Bruijn terms do not use named variables. Instead references are constructed with indices using natural numbers. An index $n : \mathbb{N}$ represents the binder that is n binders above in the abstract syntax tree. Therefore, the index indicates how many binders have to be skipped to reach the corresponding binder.

Examples for the de Bruijn representation of terms are:

$$\lambda x.\ \lambda y.\ x\ y \rightsquigarrow \lambda\ \lambda\ 0\ 1$$
$$\lambda f.\ f\ (\lambda x.\ f\ x\ (\lambda y.\ f\ y\ x)) \rightsquigarrow \lambda\ 0\ (\lambda\ 1\ 0\ (\lambda\ 2\ 0\ 1))$$
$$\lambda f.\ \lambda g.\ g\ (\lambda x.\ f\ f\ g\ x) \rightsquigarrow \lambda\ \lambda\ 0\ (\lambda\ 2\ 2\ 1\ 0)$$

As it can happen easily in an abstract context that one computes wrong de Bruijn indices, we implemented a printing function that generates strings of terms in a humanly-readable format. On the one hand, the resulting string is more readable than the large terms of the syntax in PCUIC as the string is closer to the common notation in math. On the other hand, the printing functions preserves the de Bruijn indices and therefore helps debugging wrong indices when the unquoting commands fail due to invalid indices. To inspect the names of inductive types and their constructors, the printing function is written as TemplateMonad program. Therefore, quoting and unquoting can be performed when such terms occur.

A.2 PCUIC to TemplateCoq

The translation from PCUIC to TemplateCoq is for most parts straightforward as all terms except applications and casts have a direct counter part in both syntaxes. In contrast to TemplateCoq, PCUIC does not have terms for casting. Therefore, this case poses no problems. For applications `tApp u v` the TemplateCoq function `mkApp` that generates an application with a single argument or extend an already existing application can be used. Additionally, translation functions for environments and definitions like inductive bodies have to be defined.

The correctness proof of the translation is more complicated. The main statement is that for well-formed global environments Σ the typing judgement can be transferred through the translation:

$$
\begin{aligned}
&wf\ \Sigma \rightarrow \\
&\Sigma; \Gamma \vdash t : T \\
&(trans\ \Sigma); (trans\ \Gamma) \vdash (trans\ t) : (trans\ T)
\end{aligned}
$$

The proof is performed by induction on the typing predicate. Here, a handcrafted induction principle with assumptions for the nested inductive assumptions is needed. Most cases are proven using auxiliary lemmas for translation under functions. The cases for matches, fixed points, and co-fixed points need more work due to the complexity of the typing rule. Additionally, the typing rule of matches also mentions applications, and therefore, needs the equations for the conversion of applications.

The main difference, the typing rule of applications, is solved with auxiliary lemmas relating the typing of applications to the `typing_spine` predicate. As the proof involved auxiliary lemmas for most functions used in the typing predicate, one directly gets correctness proofs of the translation for most use cases.

A.3 Notation tricks

Coq allows to define notations for convenience. For example, the function creating the eliminators is `runElim` with Boolean arguments to determine whether an induction principle or case analysis principle should be generated. Additional arguments determine the elimination type, the type of which the eliminator is generated, and the name. Therefore the induction principle for natural numbers would be generated with the command

```
MetaCoq Run (runElim ℕ true true None None )
```

But this command is not very user friendly because one does not know what the individual arguments do and also the notation with parentheses is inconvenient. The follow notation defined a short command for induction principles such that the user only has to specify the type.

```
Notation "'Scheme' 'Induction' 'for' T" := (runElim T true true None None)

MetaCoq Run Scheme Induction for ℕ
```

But with Coq's `Scheme` command it is also possible for the user to provide names for the principles that are generated. The name argument has to be a string argument but one does not want to always have to give a string with quotes when a new principle should be generated.

A clever trick [13] is to use MetaCoq to extract names from notations and convert the names into strings. Given an identifier n in a notation, the same identifier can be used to construct a λ-abstraction $\lambda(n : \mathbb{N}).\, n$. This abstraction then can be quoted with MetaCoq commands and from the quoted term the name can be extracted as string.

```
Definition getName (x : ℕ → ℕ):=
x ← tmEval cbv x;;
t ← tmQuote x;;
match t with
| Ast.tLambda (nNamed na) _ _ ⇒ tmReturn na
| _ ⇒ tmReturn ""
end.
```

The `getName` function takes a function $\mathbb{N} \to \mathbb{N}$ and, if the provided function is a λ-abstraction, extract the name of the argument. This trick is used to define the `MetaCoq Run Scheme` commands such that they can be used analogously to Coq's `Scheme` command.

A.4 Constructor list plugin

We have described a plugin to list all constructor types of a type. For natural numbers and disjunction the result should be:

```
nat_ctors =
  [ ℕ;
    ℕ→ ℕ]

or_ctors =
  [∀ A B : ℙ, A → A ∨ B;
   ∀ A B : ℙ, B → A ∨ B]
```

To get get constructors one first has to quote the given type and get the inductive definition with `tmQuoteInductive`. Afterwards, the constructors can be extracted from the body and mapped with the `monad_map` function because the unquoting needed for each constructor is a TemplateMonad command.

```
Definition getCtors {A} (ind: A) : TemplateMonad unit :=
```

```
tm ← tmQuote ind;
match tm with
| tInd (mkInd kername idx as induct ) _ ⇒
    mbody ← tmQuoteInductive kername;
    match nth_error mbody.( ind_bodies ) idx with
    | Some ibody ⇒
        ctors ← monad_map
          ( λ '(na,t, count ) ⇒
            tmUnquoteTyped 𝕋 (subst10 tm t )
          )
          ibody.( ind_ctors );
        tmPrint ctors
    | None ⇒ tmFail "the mutual inductive index was wrong"
        (* this cannot happen when a correct inductive is provided *)
    end
| _ ⇒ tmFail "argument has to be an inductive type"
end.
```

It is easy to implement these steps in MetaCoq. [1] We first quote the argument and check that it is an inductive type which we then quote with `tmQuoteInductive`. Afterwards, we extract the correct inductive type from the mutual inductive definition `mbody` and name it `ibody`.

The main step is the extraction of the constructors and their unquoting. This is done with a monadic mapping using `monad_map` over the constructor list `ibody.( ind_ctors )`. The mapping allows us to execute monadic operation to the whole list.

Before we unquote the constructor type we first have to substitute the inductive type for the `tRel 0` in the constructors as the self-reference is handled by `tRel` references. This of course has to change as the type stands by itself in our list. The `subst10 tm t` call substitutes every occurrence of the reference zero, and the lifted ones under quantifications, let-ins and abstractions, in a term `t` by `tm`.

Lastly, we print the resulting list.

We can easily extend this plugin to also contain the constructor itself when we change the mapping. The idea is to generate a list of type `list (Σ (T:𝕋), T )` which contains dependent pairs of the constructor type and the constructor with this type. For natural numbers this would be the list `[( nat; 0 ); (nat → ℕ; S )]`.

This is the new code for the mapping:

```
ctors ← monad_map_i
  ( λ i '(na, t, count ).
    ctorType ← tmUnquoteTyped 𝕋 (subst10 tm t );
```

1 For general use this code has to be changed to handle the universe constraints for the list properly.

```
      ctor ← tmUnquoteTyped ctorType (tConstruct induct i [] );
      tmEval lazy ((ctorType;ctor):Σ T : 𝕋, T)
    )
    ibody.( ind_ctors );
  tmDefinition (append ibody.( ind_name ) "_ctors" ) ctors;
  tmPrint ctors
```

The difference is that we now use an indexed mapping function and unquote a term representing the ith constructor for each constructor type. We also perform a cast to the dependent pair type to indicate which type our resulting list has. Additionally to the printing, we also create a definition for our constructor list using the name of our inductive type extended with the suffix "_ctors".

We now can run the plugin and look at the result for `or` and `sig`.

```
or_ctors =
  [(∀ A B : ℙ, A → A ∨ B; or_introl );
  (∀ A B : ℙ, B → A ∨ B; or_intror )]

sig_ctors =
  [(∀ (A : 𝕋) (P : A → ℙ)(x : A),
    P x → {x : A | P x}; exist )]
```

A.5 Typing rules

In the following, the typing rules are presented for quick reference.

Applications in TemplateCoq

```
type_App (t : term) (l : list term) (t_ty t' : term):
  Σ; Γ ⊢ t : t_ty →
  isApp t = false →
  l ≠ [] →
  typing_spine Σ Γ t_ty l t' →
  Σ; Γ ⊢ tApp t l : t'
```

Applications in PCUIC

```
type_App (t : term) (na : name) (A B u : term):
  Σ; Γ ⊢ t : tProd na A B →
  Σ; Γ ⊢ u : A →
  Σ; Γ ⊢ tApp t u : B {0 := u}
```

References

```
type_Rel (n : ℕ) (decl : context_decl):
  wf_local Σ Γ →
  nth_error Γ n = Some decl →
  Σ; Γ ⊢ tRel n : lift0 (S n) (decl_type decl)
```

Sorts

```
type_Sort (l : LevelSet.elt):
  wf_local Σ Γ →
  LevelSet.In l (global_ext_levels Σ) →
  Σ; Γ ⊢ tSort (Universe.make l) : tSort (Universe.super l)
```

Casting (only TemplateCoq)

```
type_Cast (c : term) (k : cast_kind) (t : term) (s : Universe.t):
  Σ; Γ ⊢ t : tSort s →
  Σ; Γ ⊢ c : t →
  Σ; Γ ⊢ tCast c k t : t
```

Quantification

```
type_Prod (n : name) (t b : term) (s1 s2 : Universe.t):
  Σ; Γ ⊢ t : tSort s1 →
  Σ; Γ, vass n t ⊢ b : tSort s2 →
  Σ; Γ ⊢ tProd n t b
    : tSort (Universe.sort_of_product s1 s2)
```

λ-expression

```
type_Lambda (n : name) (t b : term)
      (s1 : Universe.t) (bty : term):
  Σ; Γ ⊢ t : tSort s1 →
  Σ; Γ, vass n t ⊢ b : bty →
  Σ; Γ ⊢ tLambda n t b : tProd n t bty
```

Let-in epxression

```
type_LetIn (n : name) (b b_ty b' : term)
       (s1 : Universe.t) (b'_ty : term):
  Σ; Γ ⊢ b_ty : tSort s1 →
  Σ; Γ ⊢ b : b_ty →
  Σ; Γ, vdef n b b_ty ⊢ b' : b'_ty →
  Σ; Γ ⊢ tLetIn n b b_ty b' : tLetIn n b b_ty b'_ty
```

Constant

```
type_Const (cst : ident) (u : Instance.t)
    (decl : constant_body):
  wf_local Σ Γ →
  declared_constant Σ.1 cst decl →
  consistent_instance_ext Σ (cst_universes decl) u →
  Σ; Γ ⊢ tConst cst u
    : subst_instance_constr u (cst_type decl)
```

Inductive types

```
type_Ind (ind : inductive) (u : Instance.t)
      (mdecl : mutual_inductive_body) (idecl : one_inductive_body):
  wf_local Σ Γ →
  declared_inductive Σ.1 mdecl ind idecl →
  consistent_instance_ext Σ (ind_universes mdecl) u →
  Σ; Γ ⊢ tInd ind u
    : subst_instance_constr u (ind_type idecl)
```

Constructors

```
type_Construct (ind : inductive) (i : ℕ) (u: Instance.t)
     (mdecl : TemplateEnvironment.mutual_inductive_body)
     (idecl : TemplateEnvironment.one_inductive_body)
     (cdecl : (ident × term) × ℕ):
   wf_local Σ Γ →
   declared_constructor Σ.1 mdecl idecl (ind, i) cdecl →
   consistent_instance_ext Σ (ind_universes mdecl) u →
   Σ; Γ ⊢ tConstruct ind i u
    : type_of_constructor mdecl cdecl (ind, i) u
```

Matches

```
type_Case (indnpar : inductive × ℕ)
          (u : Instance.t) (p c : term) (brs : list (nat × term))
          (args : list term)
          (mdecl : TemplateEnvironment.mutual_inductive_body)
          (idecl : TemplateEnvironment.one_inductive_body)
          (ps : Universe.t) (pty : term)
          (btys : list (nat × term)):
let ind := indnpar.1 in
let npar := indnpar.2 in
declared_inductive Σ.1 mdecl ind idecl →
ind_npars mdecl = npar →
Σ; Γ ⊢ c : mkApps (tInd ind u) args →
let params := firstn npar args in
build_case_predicate_type ind mdecl idecl params u ps = Some pty →
Σ; Γ ⊢ p : pty →
leb_sort_family (universe_family ps) (ind_kelim idecl) →
  map_option_out
    (build_branches_type ind mdecl idecl params u p) = Some btys →
  All2
    (λ br bty : ℕ × term.
     (br.1 = bty.1 × Σ; Γ ⊢ br.2 : bty.2)
     × Σ; Γ ⊢ bty.2 : tSort ps) brs btys →
  Σ; Γ ⊢ tCase indnpar p c brs
    : mkApps p (skipn npar args ++ [c])
```

Projections

```
type_Proj (p : projection) (c : term)
     (u : Instance.t)
     (mdecl : TemplateEnvironment.mutual_inductive_body)
     (idecl : TemplateEnvironment.one_inductive_body)
     (pdecl : ident × term)
     (args : list term):
  declared_projection Σ.1 mdecl idecl p pdecl →
  Σ; Γ ⊢ c : mkApps (tInd p.1.1  u) args →
  #| args |  = ind_npars mdecl →
  let ty := pdecl.2 in
  Σ; Γ ⊢ tProj p c
    : subst0 (c ::  rev args) (subst_instance_constr u ty)
```

Fixed points

```
type_Fix (mfix : mfixpoint term) (n : ℕ) (decl : def term):
   fix_guard mfix →
   nth_error mfix n = Some decl →
   let types := fix_context mfix in
   wf_local Σ (Γ, types) →
   All
     (λ d : def term.
      Σ; Γ, types ⊢ dbody d : lift0 #| types |  (dtype d)
      × isLambda (dbody d) = true) mfix →
   Σ; Γ ⊢ tFix mfix n : dtype decl
```

Conversion

```
type_Conv t A B :
  Σ; Γ ⊢ t : A →
  isWfArity typing Σ Γ B +
  (Σ s : Universe.t, Σ; Γ ⊢ B : tSort s) →
  Σ; Γ ⊢ A ⩽ B →
  Σ; Γ ⊢ t : B
```

Bibliography

[1] Abhishek Anand and Greg Morrisett. Revisiting parametricity: inductives and uniformity of propositions. *arXiv preprint arXiv:1705.01163*, 2017.

[2] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards certified meta-programming with typed Template-Coq. In *International Conference on Interactive Theorem Proving*, pages 20–39. Springer, 2018.

[3] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. Theorem proving in lean, 2015.

[4] Julian Biendarra, Jasmin Christian Blanchette, Martin Desharnais, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Defining (Co) datatypes and Primitively (Co) recursive Functions in Isabelle/HOL. URL `http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/datatypes.pdf`.

[5] Adam Chlipala. Certified programming with dependent types, 2019.

[6] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988.

[7] Thierry Coquand and Christine Paulin. Inductively defined types. P. Martin-Löf and G. Mints, editors, COLOG-88, LNCS 417, 1990.

[8] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. North-Holland, 1972.

[9] David Delahaye. A tactic language for the system coq. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 85–95. Springer, 2000.

[10] Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. Quickchick: Property-based testing for coq. In *The Coq Workshop*, 2014.

[11] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–29, 2017.

[12] Yannick Forster and Fabian Kunze. A certifying extraction with time bounds from coq to call-by-value lambda calculus. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[13] Yannick Forster and Kathrin Stark. Coq à la carte: a practical approach to modular syntax with binders. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 186–200, 2020.

[14] Yannick Forster, Dominik Kirst, and Dominik Wehr. Completeness theorems for first-order logic analysed in constructive type theory. In *International Symposium on Logical Foundations of Computer Science*, pages 47–74. Springer, 2020.

[15] Herman Geuvers. Inconsistency of classical logic in type theory. *Unpublished notes*, 2001.

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

[17] WA Howard. Per Martin-Löf. Intuitionistic type theory. Studies in proof theory. Bibliopolis, Naples1984, ix+ 91 pp. 1986.

[18] Patricia Johann and Andrew Polonsky. Deep induction: Induction rules for (truly) nested types. In *International Conference on Foundations of Software Science and Computation Structures*, pages 339–358. Springer, Cham, 2020.

[19] Ambrus Kaposi and András Kovács. A syntax for higher inductive-inductive types. In *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[20] Gregory Malecha and Jesper Bengtson. Extensible and efficient automation through reflective tactics. In *European Symposium on Programming*, pages 532–559. Springer, 2016.

[21] Gregory Michael Malecha. *Extensible proof engineering in intensional type theory*. PhD thesis, Harvard University, 2015.

[22] Christine Paulin-Mohring. Inductive definitions in the system coq rules and properties. In *International Conference on Typed Lambda Calculi and Applications*, pages 328–345. Springer, 1993.

[23] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic* (*Mathematical logic and foundations*). College Publications, January 2015.

[24] Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[25] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *International Conference on Mathematical Foundations of Programming Semantics*, pages 209–228. Springer, 1989.

[26] John C Reynolds. Types, abstraction and parametric polymorphism. In *IFIP congress*, volume 83, 1983.

[27] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 Haskell Workshop, Pittsburgh*, pages 1–16, October 2002.

[28] Matthieu Sozeau. Subset coercions in coq. In *International Workshop on Types for Proofs and Programs*, pages 237–252. Springer, 2006.

[29] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *International Conference on Theorem Proving in Higher Order Logics*, pages 278–293. Springer, 2008.

[30] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi: 10.1145/3371076.

[31] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, pages 1–53, 2020.

[32] Walid Taha, Cristiano Calcagno, Liwen Huang, and Xavier Leroy. Metaocaml: A compiled, type-safe multi-stage programming language. *Available online from* `http://www.cs.rice.edu/taha/MetaOCaml`, 2001.

[33] Enrico Tassi. Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq. In *ITP 2019 - 10th International Conference on Interactive Theorem Proving*, Portland, United States, September 2019. doi: 10.4230/LIPIcs.CVIT.2016.23.

[34] The Coq Development Team. The Coq Proof Assistant, version 8.11.0, January 2020. URL `https://doi.org/10.5281/zenodo.3744225`.

[35] Amin Timany and Bart Jacobs. First steps towards cumulative inductive types in cic. In *International Colloquium on Theoretical Aspects of Computing*, pages 608–617. Springer, 2015.

[36] Amin Timany and Matthieu Sozeau. Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC). *CoRR*, abs/1710.03912, 2017. URL `http://arxiv.org/abs/1710.03912`.

[37] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, 1989.