Touring the MetaCoq Project (Invited Paper)

Matthieu Sozeau

Touring the MetaCoq Project (Invited Paper)

Matthieu Sozeau
Inria & LS2N, Université de Nantes
France
matthieu.sozeau@inria.fr

Proof assistants are getting more widespread use in research and industry to provide certified and independently checkable guarantees about theories, designs, systems and implementations. However, proof assistant implementations themselves are seldom verified, although they take a major share of the trusted code base in any such certification effort. In this area, proof assistants based on Higher-Order Logic enjoy stronger guarantees, as self-certified implementations have been available for some years. One cause of this difference is the inherent complexity of dependent type theories together with their extensions with inductive types, universe polymorphism and complex sort systems, and the gap between theory on paper and practical implementations in efficient programming languages. MetaCoq is a collaborative project that aims to tackle these difficulties to provide the first fully-certified realistic implementation of a type checker for the full calculus underlying the Coq proof assistant. To achieve this, we refined the sometimes blurry, if not incorrect, specification and implementation of the system. We show how theoretical tools from this community such as bidirectional type-checking, Tait-Martin-Löf/Takahashi's confluence proof technique and monadic and dependently-typed programming can help construct the following artifacts:

- a specification of Coq's syntax and type theory, the Polymorphic Cumulative Calculus of (Co)-Inductive Constructions (PCUIC);
- a monad for the manipulation of raw syntax and interaction with the Coq system;
- a verification of PCUIC's metatheory, whose main results are the confluence of reduction, type preservation and principality of typing;
- a realistic, correct and complete type-checker for PCUIC;
- a sound type and proof erasure procedure from PCUIC to untyped λ-calculus, i.e., the core of the extraction mechanism of Coq.

1 Introduction

Proof assistants have become invaluable tools in the hands of mathematicians, computer scientists and proof engineers that aim to build certified theories, software, systems and hardware, as evidenced by successful, large formalization projects ranging from famously hard mathematical results ([20],[23]) to realistic compilers ([33], [50]), program logics ([28, 5]), operating systems ([30, 21]) and even hardware design ([2, 36, 14]). Ultimately, however, all these formalizations rely on a Trusted Theory Base (TTB), that consists of the mathematical foundations of the proof-assistant -most often a variant of Higher-Order Logic, Set Theory or Type Theory- and a Trusted Code Base (TCB): its actual implementation in a general purpose programming language. To obtain the highest guarantees on the proof assistants results, one should in principle also verify the consistency of the foundations, usually by building models in a Trusted Theory (Zermelo-Fraenkel Set Theory being the most common one), the adequacy of the implementation with this theory, and the correct compilation of this implementation.

Submitted to:
LFMTP 2021

1.1 A little history

To date, only the HOL family of provers have benefitted from such a guarantee, due to the seminal work of Kummar *et al* [31], who built a self-formalization of Higher-Order Logic modeled by set theory (ZF) in HOL Light (building on Harrison's work [24]), implemented itself in CakeML. In contrast, for dependent type theories at the basis of proof assistants like Coq, Agda, Idris or Lean, self-certification, or more informally type-theory eating itself [13] is a long-standing open problem. A rather large fragment of the theory at the basis of Coq, the Calculus of Constructions with universes, was formalized in Coq by Barras during his PhD [7] and extended in his habilitation thesis [8], culminating in a proof of Strong Normalization (SN) and hence relative consistency with IZF with a number of inaccessible cardinals for this theory. His development includes a proof of subject reduction and a model-theoretic proof using a specification of Intuitionistic Zermelo Fraenkel Set Theory in Type Theory, resulting from a long line of work relating the two [3, 54]. Due to Gödel's second incompleteness theorem, one can only hope to prove the consistency of the theory with n universes in a theory with $n+1$ universes. These results prove that a quite large fragment of Coq can be shown relatively consistent to a standard foundation. Since Barras' work, both pen-and-paper and formalized model-theoretic proofs have been constructed for many variants of dependent type theories, from decidability of type-checking for a type theory with universes [1] or canonicity [26] and normalization [48] for cubical type theories. We hence consider Coq's core type theory to be well-studied, a most recent reference for a consistency proof of the calculus with inductive types, universe polymorphism and cumulativity is [53, 52].

1.2 Goals of the project

The theory behind Coq's calculus, the Polymorphic Cumulative Calculus of (Co-)Inductive Constructions (PCUIC), is rather well-studied and well-tested now: most important fragments have accepted consistency proofs and no inconsistency was found in the rules that have been implemented in the last 30 years. That is, until a new axiom like Univalence breaks implicit assumptions in the calculus as happened recently for the guard checking algorithm. More worryingly, blatant inconsistencies (accepted proofs of False), are reported at least once a year, due to bugs in the implementation[1]. The source of these bugs falls generally under the category of programming errors and unforeseen interactions between subsystems. Coq is indeed a complex piece of software resulting from 37 years of research and development, incorporating many features in a single system. Its architecture was last revised in the V7 series (1999-2004) by Jean-Christophe Filliâtre [17, 16], following the de Bruijn criterion. It means that COQ does have a well-delimited, trustable proof-checking kernel, so these errors are not to be attributed to bad design in general. Rather, the problem is that this "small" kernel already comprises around 20kLoC of complex OCAML code + around 10kLoC of C code implementing a virtual machine for conversion. The system also relies on the whole OCaml compiler when used with the `native_compute` tactic for fast computation/conversion. To be fair (and grateful!), one should note that we never had to blame OCaml for a miscompilation resulting in an inconsistency. In conclusion, to avoid these errors, we should rather apply program verification to COQ's implementation.

This is a slightly different endeavor than the above works. Indeed, mechanized or not, the aforementionned proofs are concerned with idealized versions of the calculus that do not correspond to the actual implementation of PCUIC in OCaml, nor do they have any bearing on its compiled version, *a priori*. The METACOQ[2] project's goal is to bridge the gap between the model theoretic justification of the theory and

[1] https://github.com/coq/coq/blob/master/dev/doc/critical-bugs
[2] https://metacoq.github.io

the actual implementation of the Coq kernel. To do so, we need to answer the following questions in a formal, concrete way:

- What calculus is implemented by Coq exactly?
- Which meta-theoretical properties hold on the implementation?

To answer these questions, we develop a *specification* of Coq's type theory (§2), a Coq definition of a type-checker and conversion procedure that corresponds to the current Coq implementation, and verify both the sanity of the specification and correctness and completeness of the implementation.

Plan of the article. To verify the sanity of the specification we develop the meta-theory (§3) of the PCUIC calculus and show that it enjoys type preservation (§3.2.2) and principality (§3.2.4) of types, along with the expected confluence of reduction (§3.2.3) and proof that conversion is a congruence. We can then construct a corresponding type checker (§4) that is shown correct and complete with respect to the specification. Finally, to be able to execute this type-checker on large examples, we can extract it to OCaml and compare it to the type-checker implemented in Coq. Ideally, this last step should also be verified: we present a verified erasure procedure (§5) that takes a Coq environment and definition and produces a program in an (untyped) weak call-by-value λ-calculus extended with a dummy $\Box$ constructor ($\lambda_\Box$). We prove that erasure preserves observations of values, starting from a well-typed closed term. In combination with the CertiCoq compiler [4] from $\lambda_\Box$ to `C-light` and the CompCert [18] compiler from `C-light` to assembly, this will provide an end-to-end verified implementation of Coq's kernel. We discuss the remaining hurdles to realize this and further extensions in section 6.

Attribution The results surveyed in this article are due to MetaCoq team as a whole: Abhishek Anand, Dannil Annenkov, Simon Boulier, Cyril Cohen, Yannick Forster, Meven Lennon-Bertrand, Gregory Malecha, Jakob Botsch Nielsen, Matthieu Sozeau, Nicolas Tabareau and Théo Winterhalter.

Link to the formal development This article is best read online, as links in the text point to the formal development definition, which are generally too large to include in the presentation.

2 Syntax and Semantics

The MetaCoq project initially started as an extension of Gregory Malecha's Template-Coq plugin, developed during his PhD on efficient and extensible reflection in dependent type theory [37]. Template-Coq provided a *reification* of Coq's term and environment structures in Coq itself, i.e. Coq `Inductive` datatype declarations that model the syntax of terms and global declarations of definitions and inductive types, as close as possible to the OCaml definitions used in Coq's kernel. In addition, it implemented in an OCaml Coq plugin the meta-definitions of quoting and unquoting of terms between the two representations, similarly to Agda's reflection mechanism. The syntax of terms is given in figure 1. It corresponds closely to Coq's internal syntax, handling local variables (tRel, with a de Bruijn index), free named variables (tVar), existential variables (tEvar), sorts (tSort), the type-cast construct (tCast), dependent products, lambda-abstraction, let-ins and n-ary application (tApp), application of global references: constants, inductives, and constructors, a dependent case analysis construct (tCase, see 3.5 for more details) and primitive projections (tProj), fixpoints and co-fixpoints and finally, primitive integers and floating-point values (a recent [11] addition to Coq).

```
Inductive term : Type :=
| tRel (n : nat)
| tVar (id : ident) (* For free variables (e.g. in a goal) *)
| tEvar (ev : nat) (args : list term)
| tSort (s : Universe.t)
| tCast (t : term) (kind : cast_kind) (v : term)
| tProd (na : aname) (ty : term) (body : term)
| tLambda (na : aname) (ty : term) (body : term)
| tLetIn (na : aname) (def : term) (def_ty : term) (body : term)
| tApp (f : term) (args : list term)
| tConst (c : kername) (u : Instance.t)
| tInd (ind : inductive) (u : Instance.t)
| tConstruct (ind : inductive) (idx : nat) (u : Instance.t)
| tCase (ci : case_info) (type_info:predicate term)
        (discr:term) (branches : list (branch term))
| tProj (proj : projection) (t : term)
| tFix (mfix : mfixpoint term) (idx : nat)
| tCoFix (mfix : mfixpoint term) (idx : nat)
| tInt (i : Int63.int)
| tFloat (f : PrimFloat.float).
```

Figure 1: Term syntax

On top of the term syntax, the MetaCoq.Template library also defines the type of local contexts which are lists of typing assumptions or local definitions, and global contexts: associative lists of global reference names to constant/axiom or inductive and constructors declarations.

2.1 The Template Monad

On top of these syntactic objects, one can define an API much like Coq's OCaml API to interact with the kernel of Coq: adding and looking up global definitions, calling the type-checker or higher-level primitives like proof-search for specific type-class instances. In [44], we show how this can be organized by defining a general TemplateMonad type-class that describes the structure of programs interacting with Coq's kernel. We need a monadic abstraction to encapsulate the handling of the global state of the Coq system that is being manipulated: updates to the global environment but also the obligation handling machinery of Program [43]. An interpreter for actions in this monad (e.g. adding a new definition with a given name and proof term) is meta-programmed in OCaml, using continuations to handle the interactive aspect of execution and maintaining coherence of the system state, similarly to the proof engine monad of the MTac2 [29] tactic language.

Using this monad, one can meta-program plugins in Coq that take input from the user, ask for user to provide definitions or proofs of particular types and update the environment from computed definitions. This can be used to define user-extensible translations [44, §4], or to derive lenses for a record definition [44, §5]. The TemplateMonad has a variant that also allows *extraction* of the monadic programs so they can be run efficiently in OCaml, putting it on par with the development of plugins for Coq directly in OCaml. MetaCoq plugins deriving parametricity theorems [44, §4.3.1] and induction principles and subterm relations from inductive definitions [12] can be defined this way, opening the possibility to verify their implementations. For parametricity for example, one can show that there is a procedure to construct from any well-typed term from Coq a proof that the parametricity predicate derived from its type holds on the term. As shown by Pédrot and Tabareau in [40], this can in turn be used to build internal syntactic models of richer, effectful type theories.

In the rest of this article, we will review how we built certified foundations needed for such efforts,

that is the typing specification and the type inference algorithm used to check well-typedness.

2.2 Typing, Reduction and Conversion

The calculus at the basis of Coq is the Polymorphic Cumulative Calculus of (Co-)Inductive Constructions (PCUIC). PCUIC is a general dependently-typed programming language, with pattern-matching and (co-)recursive definitions, universe polymorphic global declarations (constants or inductive types). Its origin is the Calculus of Constructions of Coquand and Huet [15] with $\beta\eta$-conversion, extended by the (Co-)Inductive type definition scheme of Paulin-Mohring [38], guarded (co-)fixpoint definitions [19], universe polymorphism [46] and cumulative inductive types [53]. The latest additions to the core calculus are a definitionally proof-irrelevant sort `SProp` and the addition of primitive types [6]. While they are supported in the syntax, they are not yet supported by our specification.

The sort system includes an infinite hierarchy of predicative universes `Type@{i}` ($i \in \mathbb{N}$) and an impredicative sort `Prop`. We consider `Set` to be a synonym for `Type@{0}`, hence its interpretation is always predicative[3]. A specificity of Coq is the treatment of `Prop` and in particular the singleton elimination rule that allows to eliminate propositional content into computational content, if it is trivial (a proof of absurdity, an equality, conjunction or accessibility proof): we will see in the section on erasure (5) how that is justified from the computational standpoint.

2.2.1 Conversion, Cumulativity

In dependent type theory, conversion and typing are usually intertwined, through the conversion rule which states that a term of type T can be seen as a term of type U for any (well-formed) typed U convertible to T. PCUIC is presented in the style of Pure Type Systems, where this conversion relation is untyped and can be defined independently on raw terms as the reflexive, symmetric and transitive closure of one-step reduction. We hence first define a reduction relation as an inductive predicate that includes all reduction rules of PCUIC: β for application, ι for cases, ζ for let-ins, `fix` and `cofix`, `delta` for constants and congruence rules allowing to apply reductions under any context (under lambdas, dependent products, etc). We take its closure with an additional twist: rather than a strict reflexivity rule, we define an α-equivalence relation that ignores the name annotations of binders (they are included in the core syntax for printing purposes only). Moreover, this relation is parameterized by a relation on universes to implement syntactic cumulativity of universes and inductive types. Indeed in Coq's theory we have:

$$\frac{i \leq j}{\texttt{Type@\{}i\texttt{\}} \leq \texttt{Type@\{}j\texttt{\}}}$$

A similar rule applies to cumulative inductive types. We call this relation α-cumulativity when instantiated with the large inequality of universes, in which case it is only a preorder. In case we instantiate it with equality of universes, we recover the usual α-conversion relation, which is an equivalence.

Two terms are hence in the cumulativity relation if they can be linked by reductions *or expansions* up-to α-cumulativity.

2.2.2 Typing

The typing relation of PCUIC is a fairly standard inductively defined relation that mostly corresponds to usual "on paper" treatments (e.g. Coq's reference manual [51]):

[3] Coq supports an `-impredicative-set` flag to switch to an impredicative interpretation, but it is seldom used today

```
Inductive typing (Σ : global_env_ext) (Γ : context) : term → term → Type
" Σ ; Γ ⊢ t : T " := (typing Σ Γ t T).
```

Figure 2: Type signature and notation for typing

The typing judgement is a predicate taking as parameters the global context extended with a universe declaration, a local context of assumptions and two terms corresponding to the subject and type of the typing rules. Derivations are defined in the Type sort to allow for easy inductions on the size of derivations. The typing rules are explained in detail in [44, §2] and [45][§2.2]. The typing rules are syntax-directed, i.e. there is one rule per head constructor of `term`, *except* for the cumulativity rule which can apply anywhere in a derivation. Note that we use a standard de Bruijn encoding for local variables, along with lifting and parallel substitution operations. As can be seen from the definition of figure 1, we used a nested datatype definition (`list term`, `list (branch term)`), hence some care must be taken to define recursive definitions and user-friendly induction principles on terms and derivations, lifting a predicate on terms to lists in the appropriate way. This is done by defining first a size measure on terms and then using well-founded induction on sizes of terms and derivations to derive easy to use induction principles.

Global environments Typing extends straightforwardly to local and global contexts. We formalize in particular which invariants should hold on the definition of inductive types, including strict positivity and the invariants enjoyed by cumulative inductive types. This is one point where we depart from the pen-and-paper treatments: indeed in [53], the theory that is studied is an idealization of Coq's implementation where equality is a judgement and inductive declarations do not carry parameters. In contrast, the implementation cannot rely on typing conditions to decide the cumulativity judgement and subtyping is rather defined on two different instances of the *same* inductive type, e..g `list@{Set} nat` and `list@{i} nat`. We hence had to retro-engineer, from Coq's OCaml code, a proper treatment of cumulative inductive types. We'll see in section 3.5 that this was a non-trivial endeavor.

2.3 Translation from Template to PCUIC

The tApp constructor represents n-ary application of a term f to a list of arguments *args*. This follows rather closely Coq's implementation, where the application node takes an array of arguments, for an even more compact representation of applications. Immediate access to the head of applications is an important optimization in practice, but this representation, while memory-efficient, imposes a hidden invariant on the term representation: the term f should not itself be an application, and the list of arguments should also always be non-empty. The application typing rule is likewise complicated as we have to consider application to a spine of arguments, rather than a single argument.

In Coq's kernel, this is handled by making the term type abstract and using smart constructors to enforce these invariants. Replicating this in Coq is tedious, as we have to either:

- work everywhere with an abstract/subset type of terms, precluding the use of primitive fixpoint and case-analysis
- or work with the raw syntax and add well-formedness preconditions everywhere

Our solution is to interface with Coq using the raw Template term syntax, keeping close to the implementation. To avoid dealing with well-formedness assumptions, we define a translation from this

syntax to the PCUIC term syntax where application is a binary constructor. We define similar reduction and typing judgments on the simplified syntax and show the equivalence of the two systems Template and PCUIC. This crucially relies on well-founded induction on the size of derivations to "reassociate" between binary applications and n-ary ones. The metatheory hereafter is developed on the more proof-friendly PCUIC syntax, but its theorems also apply to the original system. We simplify the PCUIC syntax further by removing the tCast constructor and translating it by an application of the identity function: this is observationally equivalent. The cast syntax in COQ is solely used to mark a place where a specific conversion algorithm should be used to convert the inferred type of a term with a specified expected type. This is used to call `vm_compute` or `native_compute` to perform the cumulativity check, rather than COQ's standard "call-by-need" conversion algorithm. As we do not formalize these fast conversion checks, this is not a loss. Note also that this has no bearing on the typeability of the term, *in theory*. Only *in practice* performing conversion with the default algorithm might be infeasible.

3 Metatheory

Now armed with the definition of typing and reduction in PCUIC, we can start proving the usual metatheoretical results of dependent type theories. We first derive the theory of our binding structures: the usual lemmas about de Bruijn operations of lifting and substitution are proven easily.

3.1 Digression on binding

Unfortunately, at the time we started the project (circa 2017), the Autosubst framework [41] could not be used to automatically derive this theory for us, due to the use of nested lists. We however learned from their work [47] and developed the more expressive σ-calculus, defined from operations of renaming (for a renaming $\mathbb{N} \to \mathbb{N}$) and instantiation (for a function $\mathbb{N} \to$ term), which provide a more proof-friendly interface to reason on the de Bruijn representation. We show that COQ's kernel functions of lifting and substitution (carrying just the number of crossed binders) are simulated with specific renaming and instantiation operations. Both variants are still of interest: it would be out of the question to use the σ-calculus operations which build closures while traversing terms in COQ's implementation. However the structured nature of the σ-calculus and its amenability to automation, having a decidable equational theory [47], is a clear advantage in practice.

One example where this shines is the treatment of (dependent) let-bindings in the calculus. Dependent let-bindings are an entirely different beast than ML-like let-bindings which can be simulated with abstraction and application. In particular, three rules of reduction can apply on local definitions:

$$\begin{array}{llll}
\Gamma \vdash \texttt{let}\ x := t\ \texttt{in}\ b & \rightsquigarrow & b[t/x] & \zeta \\
\Gamma \vdash \texttt{let}\ x := t\ \texttt{in}\ b & \rightsquigarrow & \texttt{let}\ x := t\ \texttt{in}\ b' \quad \texttt{when}\ \Gamma, x := t \vdash b \rightsquigarrow b' & \texttt{cong-let-body} \\
\Gamma, x := t, \Delta \vdash x & \rightsquigarrow & \uparrow^{|\Delta|+1}(t) & \delta
\end{array}$$

Here $\uparrow^n (t)$ represents the shifting of indices of the free variables of t by n, and $b[t/x]$ the usual capture-avoiding substitution. The first rule is usual let-reduction, the second is a congruence rule allowing to reduce under a `let` and the last allows to expand a local definition from the context. In the course of the metatheoretical development we must prove lemmas that allow to "squeeze" or smash the let-bindings in a context. This results in a reduced context with no let-bindings anymore and a corresponding substitution that mixes the properly substituted bodies corresponding to the removed let-bindings and regular variables, to apply to terms in the original context. This involves interchanging δ,

zeta and `cong-let-body` rules combined with the proper liftings and substitutions. This kind of reasoning appears in particular as soon as we want to invert an application judgment as let-bindings can appear anywhere in the type of the functional argument. Using σ-calculus reasoning and building the right abstractions tremendously helped simplify the proofs which would otherwise easily become indecipherable algebraic rewritings with the low level indices in liftings and substitutions.

3.2 Properties

3.2.1 Structural Properties

The usual Weakening and Substitution theorems can be proven straightforwardly by induction on the derivations. We use a beefed-up eliminator that automatically lifts the typing property we want to prove to well-formedness of the global environment, which also contains typing derivations. Likewise, we always prove properties simulataneously on well-formed local contexts and type derivations, so our theorems provide a conjunction of properties. When we moved to the σ-calculus represenation, we factorized these proofs by first stating renaming and instantiation lemmas, from which weakening and substitution follow as corrolaries. We also verify that typing is invariant by alpha-conversion, so name annotations on binders are indeed irrelevant.

3.2.2 Type Preservation

Proving subject reduction (a.k.a. type preservation) for dependent type theories can be rather difficult in a setting where definitional equality is typed, as it usually requires a logical relation argument/model construction, see e.g. [1]. However, the syntactic theory is relatively well-understood for PTS: one can first independently prove context conversion/cumulativity and injectivity of Π-types (i.e. $\Pi x : A.B \equiv \Pi x : A'.B' \rightarrow A \equiv A' \wedge B \equiv B'$), to prove type preservation in the application case. Similarly, we have injectivity of inductive type applications, up-to the cumulativity relation on universes.

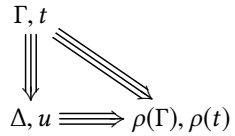
However, two other difficulties arise for PCUIC. First, we are considering a realistic type theory, will full-blown inductive family declarations, with a distinction between parameters and indices (that can contain let-bindings), and cumulative, universe polymorphic inductive types. To our knowledge, nobody ever attempted to formalize the proof at this level of detail before, even on paper. There is a good reason for that: the level of complexity is very high. Showing that the dependent case analysis reduction rule is sound mixes features such as let-bindings, the de Bruijn representation of binding adding various liftings and substitutions and the usual indigestible nesting of quantifications on indices in the typing rules of inductive types, constructors and branches. This is also ample reason to verify the code: many critical bugs a propos let-binding and inductive types were reported over the years. To tame this complexity, we tried to modularize the proof and provide the most abstract inversion lemmas on typing of inductive values and `match` constructs.

Second, COQ's theory is known to be broken regarding positive co-inductive types and subject reduction. We hence parameterize the proof: subject reduction holds only on judgments where no dependent case analysis is performed on co-inductive types. Negative co-inductive types implemented with primitive projections however can be show to enjoy subject reduction without restriction.

3.2.3 Confluence

To support inversion lemmas such as Π-type injectivity, we need to show that reduction is confluent. From this proof, it follows that the abstract, undirected conversion relation $T \equiv U$ is equivalent to reduc-

tion of the two sides to terms T' and U' that are in the syntactic α-cumulativity relation. We extend here Takahashi's refinement [49] of Tait/Martin-Löf's seminal proof of confluence for λ-calculus. The basic idea of the confluence proof is to consider parallel reduction instead of one-step reduction and prove the following triangle lemma:



The ρ function here is an "optimal" reduction function that reduces simultaneously all parallel redexes in a term or context. The fact that we need to consider contexts is due to let-bindings again: in one step of ρ, we might reduce a local definition to an abstraction, expand it in function position of an application and reduce the produced beta-redex. Using the triangle property twice, it is trivial to derive the diamond lemma and hence confluence for parallel reduction. By inclusion of one step reduction in parallel reduction, and parallel reduction in the transitive closure of one-step reduction (the "squashing" lemma), reduction is also confluent. This last part of reasoning is done abstractly, but accounting for the generalization to reduction in pairs of a context and a term. It then suffices to show commutation lemmas for reduction and α-cumulativity to show the equivalence of reduction up-to α-cumulativity and the undirected cumulativity relation. Confluence is crucial to show transitivity of the directed version.

Using this characterization of cumulativity, we can easily show that it is a congruence and that it enjoys expected inversion lemmas: if two Π-types are convertible, then they both reduce to Π-types that are in the α-cumulativity relation, so their domains are convertible and their codomains are in the cumulativity relation. Similarly, Π-types cannot be convertible to sorts or inductive types: there is no confusion between distinct type constructors.

3.2.4 Principality

As PCUIC is based on the notion of cumulativity rather than mere conversion, type uniqueness does not hold in the system. However, the refined property of principality does: for any context Γ and well-typed term t, there is a unique type T such that any other typing of t, $\Gamma \vdash t : U$ we have $T \leq U$. This property is essential for deciding type-checking: to check $\Gamma \vdash t : U$, it suffices to infer the principal type of t and check cumulativity. Principal typing is also used by the erasure procedure to take sound decisions based on the principal type of a given term.

3.3 Strengthening

The last expected structural property of the system is strengthening (a.k.a. thinning), which can be stated as:

$$\Gamma, x : A, \Delta \vdash t : T \rightarrow x \notin \mathtt{FV}(\Delta) \cup \mathtt{FV}(t) \cup \mathtt{FV}(T) \rightarrow \Gamma, \Delta \vdash t : T$$

This property ensures that typing is not influenced by unused variables, and is at the basis of the `clear` tactic of COQ, a quite essential building block in proof scripts. Unfortunately, this property *cannot* be proven by a simple induction on the typing derivation: the free-floating conversion rule allows to go through types mentioning the variables to clear, even if they do not appear in the term and type in the conclusion of the whole derivation.

3.4 Bidirectional Type Checking To The Rescue

This unfortunate situation can be resolved by reifying the principality property and type checking strategy as a bidirectional typing inductive definition. This variant of typing explicitly keeps track of the information flow in typing rules. In practice it separates the syntax-directed rules in a synthesis (a.k.a. inference) judgment (the term position is an input, while the type is an output) from the non-directed ones as checking rules (both positions are input). In [32], Meven Lennon-Bertrand develops a bidirectional variant for PCUIC, show equivalent to the orignal PCUIC, in which strengthening and principality become trivial to derive.

The crux of these argument is that bidirectional typing derivations are "canonical" for a given term, and move uses of the conversion rule to the top of the derivation, where they are strictly necessary. In addition, multiple cumulativity rules get "compressed" into a single change-of-phase rule, relying on transitivity of cumulativity. In a bidirectional synthesis derivation, if a variable does not appear in the term position, then it cannot appear in the inferred type. Simultaneously, in a checking derivation, if a variable does not appear in the term and type, then it cannot appear in these positions in any of the subderivations.

3.5 Case In Point

This detour through bidirectional typechecking is not accidental. In [45], we only proved the soundness of a typechecking algorithm for PCUIC (§4). It is in the course of formalizing the completeness of the type-checker (§4) that we discovered a problem in the typing rules of COQ. The problem appears in the dependent case analysis construct `match`. The gist of the typing rule was to typecheck the scrutinee at some unspecified inductive type $\mathsf{X}@\{\vec{u}\}\ \vec{p}\ \vec{i}$, where $\vec{u}$ is a universe instance,$\vec{p}$ the parameters and $\vec{i}$ the indices of the inductive family. The `match` construct also takes an elimination predicate, expected to be of type:

$$\Pi(\overrightarrow{x : I@[\vec{v}]}), \mathsf{X}@\{\vec{v}\}\ \vec{p'}\ \vec{x} \to \mathtt{Type}$$

Looking at this type, we would extract the universe instance $\vec{v}$ and parameters p' of the inductive X assumption. The typing rule checked that the universe instance of the scrutinee $\vec{u}$ was convertible to $\vec{v}$, rather than only in the cumulativity relation according to the subtyping rules for cumulative inductive types. It also compared the parameters $\vec{p}$ and $\vec{p'}$ for convertibility, by first lifting $\vec{p}$ in a context extended with the $\overrightarrow{x : I@[\vec{v}]}$ bindings, but these two instances did not necessarily live in the same type!

These mistakes lead to a loss of subject reduction, if cumulativity is used to lower the universes of the scrutinee, making the whole pattern-matching untypeable[4]. The problem appeared immediately while trying to prove completeness of type-checking, at the stage of designing the bidirectional typing rules: the flow of information was unclear and led us to the bug. We also realized that, to justify the comparison of the parameters, we would need to verify that $\vec{x} \notin \mathtt{FV}(\vec{p'})$ and apply strengthening, which as we have just seen is not directly provable on undirected typing rules. This motivated us to push for a change in the term representation of `match` in COQ[5] that solves both problems at once, by storing at the `match` node the universe instance and parameters that define the eliminator, and doing a sound cumulativity test of the infered type of the scrutinee and the (reconstructed) expected type of the eliminator. We are currently finishing to update the whole METACOQ development to handle this change of representation[6].

[4] https://github.com/coq/coq/issues/13495

[5] CEP 34 by Hugo Herbelin, Coq PR 13563 by Pierre-Marie Pédrot, integrated in Coq 8.14

[6] https://github.com/MetaCoq/metacoq/pull/534

4 A Type-Checker for PCUIC

4.1 Cumulativity

In [45, §3], we present a sound type-checker for PCUIC. To implement type-checking, we had to first develop a reasonably efficient reduction machine and algorithms to decide cumulativity. There are three separate algorithms at play to implement the cumulativity test.

Universes in COQ are floating variables subject to constraints [25], not directly natural numbers. To ensure consistency, one hence needs to verify that the constraints always have a valuation in the natural numbers. This boils down to deciding the (in-)equational theory of the tropical algebra $(\mathbb{N}, \mathtt{max}, +k, \leq)$. We develop a longest-simple-paths algorithm to check consistency of universe constraints: the valuation of each variable can be read off as the weight of its longest simple path from the source (`Type@{0}`). This is a naïve model and implementation of the state-of-the-art algorithm implemented in COQ, which derives from an incremental cycle detection algorithm [10] and whose formal verification is a work-in-progress [22]. Our specification is more expressive than COQ's current implementation, as it is able to handle arbitrary $\ell + k \leq \ell' + k'$ constraints between universe expressions, avoiding to distinguish so-called *algebraic* universes and implement costly universe refreshing operations when building terms. We hope to integrate this generalization back in COQ's implementation. Using this consistency check, it is easy to define α-cumulativity by structural recursion on terms.

Reduction We implement a weak-head reduction stack machine that can efficiently find the weak-head normal form of a term. To define this function, we must assume an axiom of strong normalization, which implies that reduction is well-founded on well-typed terms. This is the only axiom used in the development.

Conversion COQ uses a smart, mostly call-by-name, conversion algorithm, that uses performance-critical heuristics to decide which constants to unfold and when. COQ's algorithm does not naïvely reduce both terms to normal form to compare them, but rather incrementally reduces them to whnfs (without δ-reduction), compare their heads and recursively calls itself. When faced with the same defined constant on both sides, it first tries to unify their arguments before resorting to unfolding, resulting in faster successes but also potentially costly backtracking.

The main difficulty in the development of the conversion algorithm is that its termination and correctness are intertwined, so it is developed as a dependently-typed program that takes well-typed terms as arguments (ensuring termination of recursive calls assuming SN) and returns a proof of their convertibility (or non-convertibility). In other words it is proven sound and complete by construction. The design of the termination measure also involves a delicate construction of a dependent lexicographic ordering on terms in a stack due to Théo Winterhalter [55].

4.2 Type Checking

On top of conversion, implementing a type inference algorithm is straightforward: it is a simple structurally recursive function that takes well-formed global and local contexts and a raw term. It simply checks if the rule determined by the head of the term can apply. Figure 3 shows the type and beginning of the inference algorithm.

```
Program Fixpoint infer (Γ : context) (HΓ : ∥ wf_local Σ Γ ∥) (t : term) {struct t}
  : typing_result ({ A : term & ∥ Σ ;;; Γ |- t : A ∥ }) :=
  match t with
  | tRel n ⇒
        match nth_error Γ n with
        | Some c ⇒ ret ((lift0 (S n)) (decl_type c); _)
        | None ⇒ raise (UnboundRel n)
        end
```

Figure 3: Type inference function excerpt

Again, the function is strongly typed: its result lives in the typing_result monad, which is an exception monad, returning (ret) a sigma-type of an inferred type A and a "squashed" proof that the term has this type or failing with type error (raise). As all our derivations are in Type by default, we use explicit squashing into Prop when writing programs manipulating terms:

```
Record squash (A : Type) : Prop := sq { _ : A }.

Notation "∥ T ∥" := (squash T) (at level 10).
```

The elimination rules for propositional inductives ensures that our programs cannot make computational choices based on the shape of the squashed derivations, and that extraction will remove these arguments. The extracted version of infer hence only takes a context (assumed to be well-formed) and a term and returns an error or an infered type, just like in Coq's implementation.

Using the bidirectional presentation of the system, we can simplify the correctness and completeness proof in [45] as the algorithm really follows the same structure as bidirectional derivations. Typechecking is simply defined as type inference followed by a conversion test, as usual.

4.3 Verifying Global Environments

Once the type-checker for terms is defined, we can lift it to verify global environment declarations. For constants and axioms, this is straightforward. However, declarations of inductive types are more complex and require to first define a sound context cumulativity test, a strict positivity check and to turn the universe constraints into a graph structure. This is again done using a monad EnvCheck:

```
Program Fixpoint check_wf_env (Σ : global_env)
  : EnvCheck (∑ G, (is_graph_of_uctx G (global_uctx Σ) ∧ ∥ wf Σ ∥))
```

Given a global environment Σ, this produces either an error or a pair of a graph and a proof that the universe graph models the constraints in Σ and a (squashed) proof that the environment is well-formed.

5 Erasure from PCUIC to λ-calculus

The type-checker can be extracted to OCaml and run on reasonably large programs. For example it can be used to successfully check the prelude of the HoTT library [9], including a large proof that adjoint equivalences can be promoted to homotopy equivalences. However, our first attempt to extraction was unsuccessful: we had to change the Coq definitions so that OCaml could typecheck the generated code, as we hit a limitation of the extraction mechanism in presence of dependently-typed variadic functions. The obvious next step was hence to verify the erasure procedure itself!

In [45, §4], we present a sound erasure procedure from PCUIC to untyped, call-by-value λ-calculus. This corresponds to the first part of COQ's extraction mechanism [34], which additionally tries to maintain simple types corresponding to the original program. Erausre is performed by a single traversal of the term, expected to be well-typed. It checks if the given subterm is a type (its type is a sort `Prop` or `Type`) or if it is a proof of a proposition (its type P has sort `Prop`), in which case it returns a dummy $\Box$ term, and otherwise proceeds recursively, copying the structure of the original term. The result of erasure hence contains no type information anymore, and all propositional content is replaced with $\Box$.

We can prove the following correctness statement on this procedure:

```
Lemma erases_correct Σ t T t' v Σ' :
  extraction_pre Σ →
  Σ;;; [] |- t : T →
  Σ;;; [] |- t ⇝ℰ t' →
   erases_global Σ Σ' →
  Σ;;; [] |- t ▷ v →
  ∃ v', Σ;;; [] |- v ⇝ℰ v' ∧ Σ' ⊢ t' ▷ v'.
```

Our correctness theorem shows that if we have a well-typed term t and t erases to t', then if t reduces to a *value* v using weak-cbv reduction, then the erased term t' also reduces to an observationally equivalent value v'. The `extraction_pre` precondition enforces that the environment is well-formed. The proof follows Letouzey's original pen-and-paper proof closely [35]. Since [45], we proved two additional verified passes of optimization on the erased term and environment:

- We remove from Σ' the definitions that are not used for evaluation, pruning the environment from useless declarations that are no longer needed for the computation.
- We remove dummy pattern-matchings on $\Box$ terms, that should always trivially reduce to their single branch.

The end result of erasure is an untyped term that contains only the raw computational content of the original definition. It can be further compiled with the CERTICOQ compiler and COMPCERT to produce certified assembly code from COQ definitions.

6 Going further

We have presented the whole METACOQ project, which spans from the reification of COQ terms down to the erasure of well-typed terms to untyped λ-calculs. The whole project weights about 100kLoC of OCAML and COQ code, and is still under active development. We think this is a convincing example that we can move from a Trusted Code Base consisting of COQ's unverified kernel down to a Trusted Theory Base that consists of the formalized typing rules of PCUIC and its axiom of Strong Normalization.

The METACOQ (and CERTICOQ) projects are both ongoing work subject to limitations, we summarize here the current state of affairs.

6.1 Limitations

While PCUIC models a large part of COQ's implementation, it still misses a few impotant features of the theory:

- The η-conversion rule is not supported in our formalization, preventing us to check most of the standard library. Dealing with η rules in an untyped conversion setting is a notoriously hard

issue. We are however hopeful that we found a solution to this problem by quotienting definitional equality with η-reduction, and hope to present this result soon.

- Similarly, we do not handle the new `SProp` sort of Coq. Our methodology for η-conversion should however also apply for this case.

- We do not formalize yet the guard-checking of fixpoint and co-fixpoint definitions, relying instead on oracles. Our strong normalization assumption hence includes an assumption of correctness of the guard checkers. We are currently working on integrating a definition of the guard checking algorithm and verifying its basic metatheory (invariance by renaming, substitution, etc.).

- We did not consider the module system of Coq, which is mostly orthogonal to the core typing algorithm but represents a significant share of Coq's kernel implementation, we leave this to future work.

- We leave out the so-called "template"-polymorphism feature of Coq, which is a somewhat fragile (i.e. prone to bugs), non-modular alternative to cumulative inductive types. This prevents us from checking most of the Coq standard library today as it makes intensive use of this feature. We are working with the Coq development team to move the standard library to universe polymorphism to sidestep this issue.

6.2 Conclusion and Perspectives

There are many directions in which we consider to extend the project:

- On the specification side we would like to link our typing judgment to the "Coq en Coq" formalization of Barras [8], which provides the Strong Normalization proof we are assuming, for a slightly different variant of the calculus. This formalization is based on a sized-typing discipling for inductive types, which will require to show an equivalence with Pcuic's guardness checker, or an update of Pcuic to handle sized typing altogether.

- Proving that the theory is equivalent to a variant where conversion is typed, i.e. definitional equality is a judgment would also make our theory closer to categorical models of type theory, e.g., Categories with Families. This can build on recent results in this direction by Siles and Herbelin [42], updating them to handle cumulativity.

- In addition to the parametricity translation that we would like to prove correct, many syntactic models of type theory, extending it with side-effects [39] or forcing [27] have recently been developed. MetaCoq is the right setting to mechanize these results.

- We have concentrated our verification efforts on the core type-checking algorithm of Coq, but higher-level components like unification, elaboration and the proof engine would also benefit from formal treatment. We hope to tackle these components in the future.

- Finally, on the user side, we are still at the beginning of the exploration of the meta-programming features of MetaCoq. It could be used to justify for example the foundations of the Mtac 2 language [29], to turn the typed tactic language into a definitional extension of Coq's theory.

7 Bibliography

References

[1] Andreas Abel, Joakim Öhman & Andrea Vezzosi (2018): *Decidability of conversion for type theory in type theory*. *PACMPL* 2(POPL), pp. 23:1–23:29, doi:10.1145/3158111.

[2] Behzad Akbarpour, Amr T. Abdel-Hamid, Sofiène Tahar & John Harrison (2010): *Verifying a Synthesized Implementation of IEEE-754 Floating-Point Exponential Function using HOL*. *Comput. J.* 53(4), pp. 465–488, doi:10.1093/comjnl/bxp023.

[3] Thorsten Altenkirch (1993): *Constructions, Inductive Types and Strong Normalization*. Ph.D. thesis, University of Edinburgh.

[4] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau & Matthew Weaver (2017): *CertiCoq: A verified compiler for Coq*. In: *CoqPL*, Paris, France.

[5] Andrew W. Appel (2014): *Program Logics - for Certified Compilers*. Cambridge University Press, doi:10.1017/CBO9781107256552.

[6] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack & Laurent Théry (2010): *Extending Coq with Imperative Features and Its Application to SAT Verification*. In Matt Kaufmann & Lawrence C. Paulson, editors: *Interactive Theorem Proving*, Springer, pp. 83–98, doi:10.1016/j.jal.2007.07.003.

[7] Bruno Barras (1999): *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7.

[8] Bruno Barras (2012): *Semantical Investigations in Intuitionistic Set Theory and Type Theories with Inductive Families*. Unpublished.

[9] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau & Bas Spitters (2017): *The HoTT library: a formalization of homotopy type theory in Coq*. In Yves Bertot & Viktor Vafeiadis, editors: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, ACM, pp. 164–172, doi:10.1145/3018610.3018615.

[10] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert & Robert E. Tarjan (2016): *A New Approach to Incremental Cycle Detection and Related Problems*. *ACM Trans. Algorithms* 12(2), pp. 14:1–14:22, doi:10.1145/2756553.

[11] Guillaume Bertholon, Érik Martin-Dorel & Pierre Roux (2019): *Primitive Floats in Coq*. In John Harrison, John O'Leary & Andrew Tolmach, editors: *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, *LIPIcs* 141, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 7:1–7:20, doi:10.4230/LIPIcs.ITP.2019.7.

[12] Marcel Ullrich Bohdan Liesnikov & Yannick Forster (2020): *Generating induction principles and subterm relations for inductive types using MetaCoq*. *The Coq Workshop 2020*.

[13] James Chapman (2009): *Type Theory Should Eat Itself*. *Electron. Notes Theor. Comput. Sci.* 228, pp. 21–36, doi:10.1016/j.entcs.2008.12.114.

[14] Adam Chlipala (2020): *Proof assistants at the hardware-software interface (invited talk)*. In Jasmin Blanchette & Catalin Hritcu, editors: *CPP 2020*, ACM, p. 2, doi:10.1145/3372885.3378575.

[15] Thierry Coquand & Gérard Huet (1988): *The Calculus of Constructions*. *Information and Computation* 76(2–3), pp. 95–120, doi:10.1016/0890-5401(88)90005-3.

[16] Jean-Christophe Filliâtre (2000): *Design of a proof assistant: Coq version 7*. Research Report, Université Paris-Sud.

[17] Jean-Christophe Filliâtre (2020): *A Coq retrospective, at the heart of Coq architecture, the genesis of version 7.0*. Invited talk at the Coq Workshop 2020.

[18] Gallium, Marelle, CEDRIC & PPS (2008): *The CompCert project*. Compilers You Can *Formally* Trust.

[19] Eduardo Giménez (1998): *Structural Recursive Definitions in Type Theory*. In Kim Guldstrand Larsen, Sven Skyum & Glynn Winskel, editors: *ICALP*, *LNCS* 1443, Springer, pp. 397–408.

[20] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi & Laurent Théry (2013): *A Machine-Checked Proof of the Odd Order Theorem*. In Sandrine Blazy, Christine Paulin-Mohring & David Pichardie, editors: *ITP 2013*, *LNCS* 7998, Springer, pp. 163–179, doi:10.1007/978-3-642-39634-2_14.

[21] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg & David Costanzo (2016): *CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels*. In Kimberly Keeton & Timothy Roscoe, editors: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, USENIX Association, pp. 653–669, doi:10.5555/3026877.3026928.

[22] Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud & François Pottier (2019): *Formal Proof and Analysis of an Incremental Cycle Detection Algorithm*. In: *ITP 2019 - 10th Conference on Interactive Theorem Proving*, Portland, United States.

[23] Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason M. Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu & Roland Zumkeller (2015): *A formal proof of the Kepler conjecture*. *CoRR* abs/1501.02155.

[24] John Harrison (2006): *Towards self-verification of HOL Light*. In Ulrich Furbach & Natarajan Shankar, editors: *Proceedings of the third International Joint Conference, IJCAR 2006*, *LNCS* 4130, Springer-Verlag, Seattle, WA, pp. 177–191.

[25] Hugo Herbelin (2005): *Type Inference with Algebraic Universes in the Calculus of Inductive Constructions*. Manuscript.

[26] Simon Huber (2019): *Canonicity for Cubical Type Theory*. *Journal of Automated Reasoning* 63(2), pp. 173–210, doi:10.1007/s10817-018-9469-1.

[27] Guilhem Jaber, Gabriel Lewertowski, Pierre-Marie Pédrot, Matthieu Sozeau & Nicolas Tabareau (2016): *The Definitional Side of the Forcing*. In Martin Grohe, Eric Koskinen & Natarajan Shankar, editors: *LICS '16*, ACM, pp. 367–376, doi:10.1145/2933575.2935320.

[28] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers & Derek Dreyer (2021): *Safe systems programming in Rust*. *Commun. ACM* 64(4), pp. 144–152, doi:10.1145/3418295.

[29] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas & Derek Dreyer (2018): *Mtac2: typed tactics for backward reasoning in Coq*. *PACMPL* 2(ICFP), pp. 78:1–78:31, doi:10.1145/3236773.

[30] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch & Simon Winwood (2009): *seL4: formal verification of an OS kernel*. In Jeanna Neefe Matthews & Thomas E. Anderson, editors: *SOSP*, ACM, pp. 207–220, doi:10.1145/1629575.1629596.

[31] Ramana Kumar, Rob Arthan, Magnus O. Myreen & Scott Owens (2016): *Self-Formalisation of Higher-Order Logic - Semantics, Soundness, and a Verified Implementation*. *J. Autom. Reason.* 56(3), pp. 221–259, doi:10.1007/s10817-015-9357-x.

[32] Meven Lennon-Bertrand (2021): *Complete Bidirectional Typing for the Calculus of Inductive Constructions*. In Liron Cohen & Cezary Kaliszyk, editors: *ITP 2021*, *LIPIcs* 193, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 24:1–24:19, doi:10.4230/LIPIcs.ITP.2021.24.

[33] Xavier Leroy (2006): *Formal certification of a compiler back-end, or: programming a compiler with a proof assistant*. In: *33rd symposium Principles of Programming Languages*, ACM Press, pp. 42–54.

[34] Pierre Letouzey (2002): *A New Extraction for Coq*. In Herman Geuvers & Freek Wiedijk, editors: *TYPES'02*, *LNCS* 2646, Springer, pp. 200–219.

[35] Pierre Letouzey (2004): *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq*. Thèse de doctorat, Université Paris-Sud.

[36] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson & Anthony Fox (2019): *Verified Compilation on a Verified Processor*. In: *PLDI 2019*, ACM, New York, NY, USA, pp. 1041–1053, doi:10.1145/3314221.3314622.

[37] Gregory Michael Malecha (2014): *Extensible Proof Engineering in Intensional Type Theory*. Ph.D. thesis, Harvard University.

[38] Christine Paulin-Mohring (1993): *Inductive Definitions in the System Coq - Rules and Properties*. In Marc Bezem & Jan Friso Groote, editors: *Typed Lambda Calculi and Applications*, doi:10.1007/BFb0037116.

[39] Pierre-Marie Pédrot & Nicolas Tabareau (2017): *An effectful way to eliminate addiction to dependence*. In: *LICS 2017*, IEEE Computer Society, pp. 1–12, doi:10.1109/LICS.2017.8005113.

[40] Pierre-Marie Pédrot & Nicolas Tabareau (2020): *The fire triangle: how to mix substitution, dependent elimination, and effects*. *Proc. ACM Program. Lang.* 4(POPL), pp. 58:1–58:28, doi:10.1145/3371126.

[41] Steven Schäfer, Tobias Tebbi & Gert Smolka (2015): *Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions*. In Christian Urban & Xingyuan Zhang, editors: *ITP 2015*, *LNCS* 9236, Springer, pp. 359–374, doi:10.1007/978-3-319-22102-1_24.

[42] Vincent Siles & Hugo Herbelin (2012): *Pure Type System conversion is always typable*. *J. Funct. Program.* 22(2), pp. 153–180, doi:10.1017/S0956796812000044.

[43] Matthieu Sozeau (2007): *Subset Coercions in Coq*. In Thorsten Altenkirch & Conor McBride, editors: *TYPES'06*, *LNCS* 4502, Springer, pp. 237–252, doi:10.1007/978-3-540-74464-1_16.

[44] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau & Théo Winterhalter (2020): *The MetaCoq Project*. *Journal of Automated Reasoning* 64(5), pp. 947–999, doi:10.1007/s10817-019-09540-0.

[45] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau & Théo Winterhalter (2020): *Coq Coq Correct! Verifying Typechecking and Erasure for Coq, in Coq*. *Proceedings of the ACM on Programming Languages* 4(POPL), doi:10.1145/3371076.

[46] Matthieu Sozeau & Nicolas Tabareau (2014): *Universe Polymorphism in Coq*. In Gerwin Klein & Ruben Gamboa, editors: *ITP 2014*, *LNCS* 8558, Springer, pp. 499–514, doi:10.1007/978-3-319-08970-6_32.

[47] Kathrin Stark, Steven Schäfer & Jonas Kaiser (2019): *Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions*. In Assia Mahboubi & Magnus O. Myreen, editors: *CPP 2019*, ACM, pp. 166–180, doi:10.1145/3293880.3294101.

[48] Jonathan Sterling & Carlo Angiuli (2021): *Normalization for Cubical Type Theory*. *CoRR* abs/2101.11479.

[49] Masako Takahashi (1995): *Parallel Reductions in lambda-Calculus*. *Inf. Comput.* 118(1), pp. 120–127, doi:10.1006/inco.1995.1057.

[50] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens & Michael Norrish (2019): *The verified CakeML compiler backend*. *J. Funct. Program.* 29, p. e2, doi:10.1017/S0956796818000229.

[51] The Coq Development Team (2021): *The Coq Proof Assistant*, doi:10.5281/zenodo.4501022.

[52] Amin Timany & Matthieu Sozeau (2017): *Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC)*. Research Report RR-9105, KU Leuven, Belgium ; Inria Paris.

[53] Amin Timany & Matthieu Sozeau (2018): *Cumulative Inductive Types In Coq*. In Hélène Kirchner, editor: *FSCD*, *LIPIcs* 108, pp. 29:1–29:16, doi:10.4230/LIPIcs.FSCD.2018.29.

[54] Benjamin Werner (1997): *Sets in types, types in sets*. In Martín Abadi & Takayasu Ito, editors: *Theoretical Aspects of Computer Software*, Springer, pp. 530–546, doi:10.1007/BFb0014566.

[55] Théo Winterhalter (2020): *Formalisation and Meta-Theory of Type Theory*. Ph.D. thesis, Université de Nantes. 2020NANT4012.