Aspects of a machine-checked intermediate language for extraction from Coq, in MetaCoq

Yannick Forster and Matthieu Sozeau

Inria, Gallinette Project-Team, Nantes, France

Abstract

We discuss our progress on establishing $\lambda_\Box$, an untyped version of the PCUIC calculus used in MetaCoq to model the type theory of the Coq proof assistant, as intermediate language for various verified extraction and compilation projects. We aim at establishing and unifying correctness properties for erasure from PCUIC to $\lambda_\Box$ and at providing both translation-validated and verified syntax transformations on PCUIC and $\lambda_\Box$.

In previous work, we have given a machine-checked correctness proof for type and proof erasure [10] based on MetaCoq [9]. The central theorem is due to Letouzey [8] and states that if a term t of PCUIC (the name of the calculus used in the MetaCoq formalisation of the type theory underlying Coq) weak call-by-value evaluates to a value v, then the application of the erasure function to t yields a term in the untyped calculus $\lambda_\Box$ which weak call-by-value evaluates to a value v' which is in the erasure relation of v. Crucially, the theorem mentions both an erasure function and a (non-functional) erasure relation, which only agree on values of first-order inductive type but not in general (see [8, 10] for details).

Currently, $\lambda_\Box$ is used as intermediate language for extraction from Coq to other targets, e.g. in the CertiCoq compiler from Coq to C code [1, 3], in the ConCert project [2] extracting from Coq to (ML-like) blockchain languages, and in our own project on verifying Coq's extraction to OCaml.[1] In the latter, we are working with respect to a formal semantics of the Malfunction language, the (untyped) intermediate language of the OCaml compiler [5].

The challenges arising are manifold: They are of mathematical nature, often due to the intricacy of PCUIC regarding universes and cases, proof-engineering challenges, often due to the large size of terms (with 15 constructors) and predicates (the cumulativity relation has 26 rules), and software engineering challenges, usually due to the > 20 minutes build time of the project, consisting of more than 150k LoC.

First-order inductive types and values We define a boolean function inspecting the syntactic representation of an inductive type and checking whether it is first-order. An inductive type is first-order if the types of all parameters, indices, and arguments of constructors are first-order. With this definition `nat` is first-order, `list` and `Vector.t` are not, but e.g. `listbool` and `vectorbool : nat -> Type` (non-polymorphic variants of `list bool` and `Vector.t bool`) are first-order. We prove that values of first-order inductive type can be characterised by an inductive predicate just admitting constructor applications of first-order inductive types, and that on values of first-order inductive types the erasure function and the erasure predicate agree.

Weak call-by-value evaluation vs. reduction In PCUIC, as usual in type theory, the reduction strategy is neither call-by-value nor call-by-name, and crucially applies to open terms.

The machine-checked correctness theorem of type and proof erasure, tailored to serve as the front end of extraction pipelines, states correctness w.r.t. weak call-by-value evaluation. It is easy to prove that weak call-by-value evaluation is included in reduction, but the converse is of course not true in general. It is however true for terms of inductive first-order type. We are working towards a machine-checked proof of this standardisation result, relying on a newly machine-checked proof of progress for weak call-by-value reduction and, for simplicity, on strong normalisation of reduction.

[1] Joint work with Pierre Giraud, Pierre-Marie Pédrot, and Nicolas Tabareau.

Constructors as functions vs. constructors as blocks In PCUIC, constructors are (curried) functions, e.g. `cons S` is a valid term. Both the CertiCoq representation of inductive types and constructor types in Malfunction are modelled after the OCaml representation, where constructors are n-ary but not functions (see [7] for historic comments).

Currently, PCUIC does not model η-equivalence, as explained in [6]. We thus define an η-expandedness predicate for both PCUIC and $\lambda_\Box$, and implement a syntax transformation returning η-expanded terms. The Coq kernel can be used to certify that the result is indeed (η-)convertible to the original term, a translation validation approach first used in ConCert [2].

Structural fixpoints vs. true fixpoints In PCUIC, recursion is structural and fixpoints are annotated with a principal argument. Fixpoint reduction is triggered once the principal argument exposes a constructor application. We mirror this behaviour also in $\lambda_\Box$ to simplify the correctness proof of erasure. In particular, this means that fixpoints always are functions, and fixpoints applied to some but not enough arguments are considered values.

Programming languages with non-termination do not require structural recursion. In the first intermediate language L2k of the CertiCoq compiler, a fixpoint reduces once a single argument is present, and only non-applied fixpoints are considered values. In Malfunction, the body of a fixpoint needs to be syntactically a λ-expression, and a fixpoint is always unfolded, consequently fixpoints are not considered values.

In general, translating from structural fixpoints in $\lambda_\Box$ to unary fixpoints (treated either like in Malfunction or like in L2k) will not be correct. We identify a subset of PCUIC for which the translation behaves correctly: Fixpoints have to be always syntactically applied to at least $1 + r$ arguments, where r is the index of the structural recursion argument, and the variable corresponding to a recursive call in the body of a fixpoint has to be expanded as well.

We provide a second weak call-by-value evaluation relation for $\lambda_\Box$ which is equivalent on terms with expanded fixpoints. The η-expansion translation also covering fixpoints becomes significantly more involved than our previous version and the one used in ConCert.

Case representation PCUIC represents cases with explicit contexts, containing also the bodies bound in `let` expressions in constructor types [11]. $\lambda_\Box$ does not allow `let` expressions in case contexts. Thus, we provide a verified `let`-expansion pass on PCUIC, where `let`s in constructor types are disallowed, simplifying other future syntax transformations on PCUIC.

Parameter stripping Parameters are irrelevant for case analysis in PCUIC, and are not represented in terms in polymorphically typed languages like OCaml. Thus, it makes sense to remove parameters from inductive types before extracting from $\lambda_\Box$ to a programming language. We provide such a machine-checked parameter stripping pass. We prove that it does not affect weak call-by-value evaluation on eta-expanded terms, and that it preserves eta-expansiveness. Although mathematically not involved, the verification requires around 1500 lines of proofs.

Induction principles and views Application in PCUIC is unary, but e.g. mutual fixpoints contain a list of their bodies. The former makes inspecting the head of an application in recursive functions tedious, while for the latter Coq's generated induction principles do not suffice. We provide views usable with the Equations plugin [12] and manually proved induction lemmas to solve these issues.

Performance Our erasure process can both be used by extracting it to OCaml and inside Coq. We are working on improving performance for both. Building the global context only once, maintaining it through all translations, and using efficient update mechanisms via AVL trees turned out to be crucial and yielded performance gains with a factor of 100. We are currently investigating whether using extensional tries [4] yields further performance gains.

References

[1] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. CertiCoq: A verified compiler for Coq. In *The third international workshop on Coq for programming languages (CoqPL)*, 2017.

[2] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. ConCert: a smart contract certification framework in Coq. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 215–228. ACM, 2020. doi:10.1145/3372885.3373829.

[3] Andrew Appel, Yannick Forster, Anvay Grover, Joomy Korkut, John Li, Zoe Paraskevopoulou, and Matthieu Sozeau. CertiCoq (GitHub repository). 2022. URL: https://github.com/CertiCoq/certicoq.

[4] Andrew W Appel and Xavier Leroy. Efficient Extensional Binary Tries. working paper or preprint, October 2021. URL: https://hal.inria.fr/hal-03372247.

[5] Stephen Dolan. Malfunctional programming. In *ML Workshop*, 2016.

[6] Meven Lennon-Bertrand. À bas l'η — Coq's troublesome η-conversion. In *The first Workshop on the Implementation of Type Systems (WITS)*, 2022.

[7] Xavier Leroy. Post on the Caml mailing list. 2001. URL: https://web.archive.org/web/20170822231903/https://caml.inria.fr/pub/ml-archives/caml-list/2001/08/47db53a4b42529708647c9e81183598b.en.html.

[8] Pierre Letouzey. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq*. PhD thesis, 2004. URL: http://www.pps.jussieu.fr/~letouzey/download/these_letouzey.pdf.

[9] 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*, 2020. URL: https://hal.inria.fr/hal-02167423, doi:10.1007/s10817-019-09540-0.

[10] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! Verification of type checking and erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, 2019. doi:10.1145/3371076.

[11] Matthieu Sozeau, Meven Lennon-Bertrand, and Yannick Forster. The curious case of case: correct & efficient representation of case analysis in coq and metacoq. In *The first Workshop on the Implementation of Type Systems (WITS)*, 2022.

[12] Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in Coq. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019. doi:10.1145/3341690.