Verified Extraction from Coq to OCaml

Yannick Forster, Matthieu Sozeau, Nicolas Tabareau

▶ To cite this version:

Yannick Forster, Matthieu Sozeau, Nicolas Tabareau. Verified Extraction from Coq to OCaml. 2023. hal-04329663

HAL Id: hal-04329663

https://inria.hal.science/hal-04329663

Preprint submitted on 7 Dec 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial 4.0 International License

Verified Extraction from Coq to OCaml

YANNICK FORSTER, MATTHIEU SOZEAU, and NICOLAS TABAREAU, Inria, France

One of the central claims of fame of the Coq proof assistant is extraction, *i.e.,* the ability to obtain efficient programs in industrial programming languages such as OCaml, Haskell, or Scheme from programs written in Coq's expressive dependent type theory. Extraction is of great practical usefulness, used crucially *e.g.,* in the CompCert project. However, for such executables obtained by extraction, the extraction process is part of the trusted code base (TCB), as are Coq's kernel and the compiler used to compile the extracted code. The extraction process contains intricate semantic transformation of programs that rely on subtle operational features of both the source and target language. Its code has also evolved since the last theoretical exposition in the seminal PhD thesis of Pierre Letouzey. Furthermore, while the exact correctness statements for the execution of extracted code are described clearly in academic literature, the interoperability with unverified code has never been investigated formally, and yet is used in virtually every project relying on extraction.

In this paper, we describe the development of a novel extraction pipeline from Coq to OCaml, implemented and verified in Coq itself, with a clear correctness theorem and guarantees for safe interoperability.

We build our work on the MetaCoq project, which aims at decreasing the TCB of Coq's kernel by re-implementing it in Coq itself and proving it correct w.r.t. a formal specification of Coq's type theory in Coq. Since OCaml does not have a formal specification, we make use of the Malfunction project specifying the semantics of the intermediate language of the OCaml compiler.

Our work fills some gaps in the literature and highlights important differences between the operational semantics of Coq programs and their extracted variants. In particular, we focus on the guarantees that can be provided for interoperability with unverified code, identify guarantees that are infeasible to provide, and raise interesting open question regarding semantic guarantees that could be provided. As central result, we prove that extracted programs of first-order data type are correct and can safely interoperate, whereas for higher-order programs already simple interoperations can lead to incorrect behaviour and even outright segfaults.

CCS Concepts: • **Software and its engineering → Compilers**; **Functional languages**; **Formal software verification**; • **Theory of computation → Type theory**.

Additional Key Words and Phrases: Coq, verified compilation, extraction, interactive theorem proving, functional programming

1 INTRODUCTION

Extraction of programs written with a proof assistant based on type theory can be seen as a compilation process from a high-level language to a low-level one (*e.g.*, OCaml, Haskell, or C). This is the standard way to get a certified executable code from a Coq formalisation [Letouzey 2004], that is used for instance by the CompCert compiler [Leroy 2006], the CertiCoq project [Anand et al. 2017], the verified Javascript reference interpreter JSCert [Bodin et al. 2014], or the Velús verified compiler for Lustre [Bourke et al. 2017]. From this point of view, extraction is part of the *trusted code base* (TCB) of the proof assistant and thus should come with strong guarantees.

Since the behavior of the extraction process is crucial for the trust in formally verified software, it seems natural to have clear and formally verifiable guarantees about this process. Precisely, one would expect two guarantees: First, preservation of the operational semantics, that is if a program evaluated to a value in the proof assistant, its extracted version evaluates to a corresponding value in the target language. Second, preservation of typing, that is any extracted program is well-typed in the target language.

Authors' address: Yannick Forster, yannick.forster@inria.fr; Matthieu Sozeau, matthieu.sozeau@inria.fr; Nicolas Tabareau, nicolas.tabareau@inria.fr, Inria, Nantes, France.

Unfortunately, there are severe limitations to both properties. For the preservation of operational semantics, one needs to first capture the operational semantics of the target language, and notions such as "corresponding value". For the case of OCaml, there is no agreed upon formal semantics, and defining correspondence relations is intricate. Of course, on values of first-order data type such as natural numbers or booleans, the definition is straightforward, but users of extraction are usually interested in extracting programs that take arguments as inputs and produce values. This means that we need to define what it means for a function written in the low-level target language to be equivalent to a program written in a proof assistant to give any interesting operational specification of extraction.

The situation on the typing side is even less satisfactory. For extraction, the type system of the source language usually features general polymorphism or even dependent types, and is thus much more expressive that the one of the target language. In practice, the extracted code thus needs to use unsafe typing features to produce a piece of code that may be accepted by the typechecker of the target language. For instance, the extraction from Coq to OCaml quickly makes use of the unsafe type cast operation `Obj.magic`. If one wants to formally specify the behaviour of such programs, one quickly encounters Xavier Leroy's famous motto:[1]

"*Repeat after me: "`Obj.magic` is not part of the OCaml language".*"

Indeed, as soon as one starts using such unsafe features, one has absolutely no guarantee from the type checker anymore, and efforts to make the rest of the extracted code welltyped seem more or less useless.

Letouzey's seminal PhD thesis [Letouzey 2004] presents the current extraction mechanism of the Coq proof assistant and discusses correctness guarantees. The mathematical development is mainly centered around an intermediate language called $\lambda_\square$, which has a formal semantics that is similar to Coq's operational semantics. The translation from $\lambda_\square$ to languages such as OCaml is then left unspecified, due to a lack of formal semantics for OCaml. However, the discrepancy between provided guarantees and uses of extraction in practice can already be explained in terms of $\lambda_\square$. Letouzey proves two central theorems: First, that the operational behaviour of erased $\lambda_\square$ terms and Coq terms is in agreement through a (non-deterministic) erasure relation [Letouzey 2004, Thm. 6, pg. 60]. Secondly, that higher order functions, *e.g.,* of type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$, behave correctly when they are called on $\lambda_\square$ functions that extensionally agree on a Coq function of the same type [Letouzey 2004, Thm. 9, pg. 74]. The second result is proved using a logical relation, and, provided a formal semantics of OCaml and a proof of the first result regarding OCaml, could be extended to OCaml as well. However, this means that the only safe interoperability of extracted Coq functions is, in practice, with other extracted Coq functions.

We address three shortcomings of the state of the art in the present paper:

(1) Due to a lack of formal semantics for OCaml, the extraction process is not and cannot be formally specified and thus not proved correct.
(2) Interoperation of higher-order functions with (potentially) nonterminating or effectful OCaml programs is not captured by the correctness guarantees, because they do not extensionally agree with a Coq program.
(3) Even though a type interface is provided by extraction, the use of `Obj.magic` means that there are no guarantees by OCaml's type checker.

We propose to address those issues in the particular setting of extraction from programs written in the Coq proof assistant to the OCaml programming language. For other proof assistants such as Lean or Agda, and other target languages such as Scheme or Haskell, similar but subtly different

[1] Message on the OCaml mailing list: https://sympa.inria.fr/sympa/arc/caml-list/2009-10/msg00181.html

issues arise. We however hope that by the exposition of this special case, our techniques can be transported and reused for other source and target languages.

To solve the issue of trust regarding operational correctness (1), we contribute a formal semantics of the Malfunction language by Dolan [2016]. Malfunction has three central design goals:

- it is "a thin wrapper around OCaml 's [...] intermediate representation", and thus trivial to compile to it,[2]
- its "syntax is simplified, to allow easy code generation",
- and its semantics is stricter in most cases than that of the OCaml intermediate representation, to be robust to future changes to OCaml.

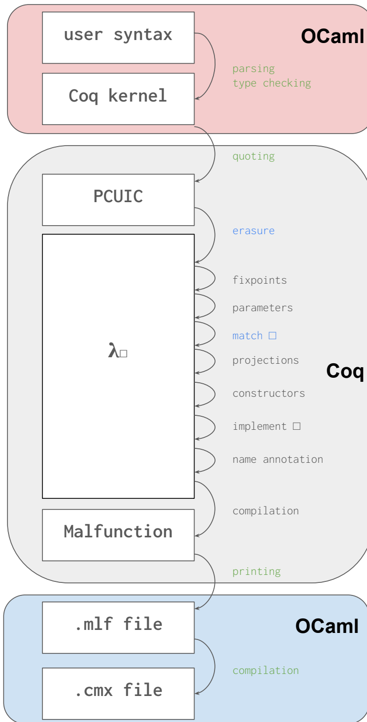
We then give a machine-checked proof of preservation of evaluation behaviour, connecting the formal semantics of Coq to the formal semantics of Malfunction by a proof in Coq. The proof is split into a long pipeline of intermediate passes to make the proof engineering overhead manageable. The central theorem talks about terms of *first-order data type*, *i.e.,* of inductive type were all constructors arguments are of first-order data type.

To solve the issue of interoperability (2), we prove that for *first-order functions*, *i.e.,* functions from first-order data types to first-order data types, *any* interoperation with OCaml code is safe, even if the Coq code internally uses higher-order features and proofs. In the next section, we provide a counter-examples that show that outside this setting, it is possible to get extracted code that produces a segmentation fault when called with a well-typed OCaml argument. This is because most of the target languages of extraction can perform effects such as writing to reference cells, and the strong guarantees provided by the proof assistant only apply to pure functions.

To solve the typing issue (3), we advocate for a drastic change of perspective on extracted code: Instead of hoping that the extracted code is well-typed in the target language and correcting problems heuristically using unsafe casts, we advocate for just exposing safe first-order function interfaces as discussed in the previous paragraph, and using untyped but verified code for the internal, higher-order parts of the code. Running the typechecker of the target language is superfluous anyways: the generated code is already type checked by the proof assistant, providing strong guarantees. Thus what we contribute in this paper is a definition of a semantic typing judgment for the target language [Dreyer 2018] à la realisability, and a formal proof that the extracted code from a Coq program with a simple enough type realizes the corresponding erased type in OCaml. The type system of OCaml is one way to statically check that a program realises a type, but Coq's type system plus the extraction mechanism should be seen as another way to guarantee realisability, eliminating the need to use the OCaml type checker for our purposes. Targeting Malfunction is in line with these considerations, since it is untyped but supports exposing a typed interface.

Overall, our compilation chain is depicted in Fig. 1 and can be summarised as follows. A program is type-checked by the Coq kernel (red area). It is then quoted using a quotation mechanism that is essentially the identity function into MetaCoq's term representation of Coq itself [Sozeau et al. 2019a], with the assumption that the quotation yields a program that is well-typed according to MetaCoq's typing judgment. The theory of the kernel of Coq formalised in MetaCoq is called PCUIC. We perform verified type and proof erasure [Sozeau et al. 2019b], targeting the untyped λ-calculus $\lambda_\square$. We then perform several transformation steps on $\lambda_\square$, namely to change the representation of fixpoints, to remove parameters, to remove case analysis on proofs, to implement native projections, to treat constructors as blocks, to implement erasure residues as functions, and to annotate names. All this is in the grey area. The final transformation is a translatation to representation of Malfunction in Coq. To get an executable (blue area), we finally print

[2]The OCaml compiler's intermediate language is called `Lambda`. To avoid confusion with $\lambda_\square$, we do not use the name.




```
From VerifiedExtraction
 Require Import Loader.

Definition function_or_ℕ
 : ∀ (b:𝔹), if b then 𝔹 → 𝔹 else ℕ :=
 fun b ⇒
  match b with
  | true ⇒ fun x ⇒ x
  | false ⇒ S 0
  end.

Definition function := function_or_ℕ true.

MetaCoq Verified Extraction
 function.
MetaCoq Run Print mli function.

(* Prints:
   type 𝔹 = True | False
   val function : 𝔹 → 𝔹 *)
```

Output in file out.mlf:

```
(module
  ($def_function_or_nat
    (lambda ($b)
      (let ($discr $b)
        (switch $discr
          ((0 0) (lambda ($x) $x))
          ((1 1) (block (tag 0) 0))))))
  ($def_function
    (apply $def_arity_function 0))
  ($function $def_function)
(export $function))
```

Fig. 1. Left: Overview of the extraction process. Red background indicates parts of the Coq proof assistant implementation in OCaml, grey parts implemented and verified in Coq, blue parts of the Malfunction compiler implementation in OCaml. Green font means the implementations are part of the TCB. Blue font means that we are using work by Sozeau et al. [2019b].
Right: Example invocation of the verified extraction plugin and its output.

the s-expression representation of the resulting Malfunction program to a file and use the Malfunction compiler to compile to OCaml compilation units (.cmx).

Before diving into the technical details of our formal setting, let us look into examples that illustrate our claims and the current limitations of Coq's official extraction mechanism to OCaml.

1.1 Scope and Limitations of Extraction to OCaml

We here discuss general limitations of any extraction to OCaml, to which any extraction process – and thus Coq's current extraction process as well as our verified replacement – are subject to.

First, a situation where OCaml's type system is too weak arrives as soon as one uses higher-order dependent types and exposes this function to be called from unverified OCaml code. We are going to develop a function `assumes_purity` : (unit → 𝔹) → 𝔹 where applying it to an (effectful) OCaml function `impure` leads to a segmentation fault. We here consider the example of a function `function_or_ℕ` that expects a boolean value b and returns a a function from 𝔹 to 𝔹 if b is `true` and a natural number otherwise, inspired by Letouzey's discussion [2004, §3.1.3, 3.2.1.].

```
Definition function_or_ℕ : ∀ (b:𝔹), if b then 𝔹 → 𝔹 else ℕ :=
 fun b ⇒ match b with true ⇒ fun x ⇒ x | false ⇒ S 0 end.
```

Since OCaml's type system does not support type-on-term dependency, there is no way to give a type to the extracted version of function_or_ℕ and thus the extraction implemented in Coq as of today reads:

```
(** val function_or_ℕ : 𝔹 → Obj.t **)
let function_or_ℕ = function | True → Obj.magic (fun x → x) | False → Obj.magic (S 0)
```

The type Obj.t is actually a synonym to the universal type, which can be used to type any term using Obj.magic, meaning that function_or_ℕ cannot be typed in OCaml without unsafe casts. The use of Obj.magic, besides annihilating the interest of the type system, also pollutes the rest of the code, because any use of function_or_ℕ must then be enclosed in an Obj.magic invocation to cast the return type to 𝔹 → 𝔹 or ℕ. In practice, the extraction tries to avoid the greedy use of Obj.magic which has the even less satisfactory consequence of producing ill-typed terms in some complex situations [Sozeau et al. 2019b, §3.6].

This taken aside, let us continue our example and define a function that takes a boolean value b and, as input, again either function from 𝔹 to 𝔹 or a natural number (depending on b) and applies the function to an argument in the first case while just returning a default value in the second:

```
Definition apply_function_or_ℕ : ∀ b : 𝔹, (if b then 𝔹 → 𝔹 else ℕ) → 𝔹 :=
 fun b ⇒ match b with true ⇒ fun f ⇒ f true | false ⇒ fun _ ⇒ false end.
```

Again, the OCaml extraction has to use unsafe casts:

```
(** val apply_function_or_ℕ : 𝔹 → __ → 𝔹 **)
let apply_function_or_ℕ b f = match b with | True → Obj.magic f True | False → False
```

We now use apply_function_or_ℕ with function_or_ℕ as argument, while letting the boolean be determined by a function from unit to 𝔹 applied to the only element tt of unit:

```
Definition assumes_purity : (unit → 𝔹) → 𝔹 :=
 fun f ⇒ apply_function_or_ℕ (f tt) (function_or_ℕ (f tt)).
```

This function can be extracted to an OCaml function whose type is a perfectly valid OCaml type:

```
(** val assumes_purity : (unit → 𝔹) → 𝔹 **)
let assumes_purity f = apply_function_or_ℕ (f ()) (function_or_ℕ (f ()))
```

Now on the Coq side the function assumes_purity is well-typed under the invariant that f tt always returns the same value. Thus, nothing will go wrong in OCaml as long as assumes_purity is always called with an OCaml function of type unit → 𝔹 that always returns the same value.

If this invariant however is broken, the internal unsafe casts Obj.magic become invalid, and all guarantees are lost. And indeed, it is easy to break this invariant in OCaml by using a reference:

```
let impure : unit → 𝔹 = let x : 𝔹 ref = ref False in
 fun _ → match !x with False → (x := True; False) | True → True
```

The function impure returns False the first time it is called and True afterwards. When applied to assume_purity, the execution is essentially equivalent to apply_function_or_ℕ True (function_or_ℕ False), meaning apply_function_or_ℕ expects a function as argument, but gets a natural number. Thus, the program tries to evaluate the application of Obj.magic False True which induces a segmentation fault:

```
assumes_purity impure
(** Segmentation fault: 11 **)
```

This examples illustrates the fact that even if an extracted program has a valid OCaml interface, this does not mean that its interaction with well-typed OCaml pieces of code will not go wrong.

1.2 What Guarantees Can We Expect From Extracted Code?

The situation looks pretty bad. On one side, the target type system is fundamentally too weak to support extraction of dependently typed programs. On the other side, even on the common part of the type system, the semantics associated to the type is very different between the two systems. In particular, because CIC is a pure language, it satisfies strong invariants, such as the fact that a function always returns the same output on the same input. But by using dependent types, those invariants at the computational level can be lifted at the type level. This is the case for instance for the `pure_apply` function. Thus, one can wonder whether we can expect any guarantee at all for the interaction between an extracted program and OCaml pieces of code?

This paper provides the first positive answer in two steps. First, we do not expect extraction to produce a well-typed OCaml program, but rather we prove that it produces an untyped low-level functional code that operationally behaves the same as Coq's initial program. Then, by defining a realisability semantics for low-level pieces of functional code over the OCaml type system, we can show that for first order types (*i.e.,* functions whose domain and codomain are first-order data types) the extracted code actually meets its specification and can thus interact safely with any pieces of OCaml code.

Note that this result is significantly less restrictive than it might seem, *e.g.,* the benchmarks used for evaluation of compilation times in CertiCoq readily satisfy the constraints.

Major programs relying on extraction as of today might not all be in this shape, but for reasons related to convenience and thus they could be brought to this shape. For instance for CompCert, the main compilation function[3] would satisfy this condition if not for a representation of integers as unbounded numbers with an explicit proof of a bound (which could easily be re-organised into a syntax type without this bounding constraint and a first translation phase that establishes it). For our own extraction procedure, the only hinderance is representing non-empty lists (for universe instances) via `X * list X` rather than a pair of a list and a non-emptiness proofs.

1.3 A General Plan to Support Certified Extraction Towards Several Languages

In the process of proving operational correctness and realisability properties of extraction, there is a sizable part that is independent of the target language and can be defined commonly. To make this explicit, $\lambda_\square$ has been introduced by Letouzey [2004] as a common intermediate untyped language in which an original Coq term can be erased. Sozeau et al. [2019b] have mechanised a weak call-by-value variant of $\lambda_\square$ in Coq based on the MetaCoq project. They give a machine-checked proof of type and proof erasure, removing type information and proof terms that are in `Prop` (or `SProp`), and replace everything with a canonical $\square$ term.

In this paper, we show how to define a compilation pipeline that gradually transform a $\lambda_\square$ term to a term that computes the same values, but is operationally closer to the semantics of the target language. These different phases correspond to compilation phases usually provided by compilers and are not specific to extraction to OCaml. They consist for instance in removing the parameters of constructors of inductive types that are required in Coq for type inference but have no computational content and thus can be discarded in a simply typed or untyped setting. There are also other phases such as η-expansion of constructors of inductive types that are not fully syntactic and require typing to be defined. Those phases need thus to be performed before erasure to $\lambda_\square$.

As of today, this pipeline based on $\lambda_\square$ is already use in the CertiCoq project [Anand et al. 2017]. It could easily be used *to target* other call-by-value languages as well, *e.g.,* blockchain-based languages as covered in the ConCert project [Annenkov et al. 2020], or even languages like Rust.

[3] https://compcert.org/doc/html/compcert.driver.Compiler.html#transf_c_program

It could also be used *as a target* for other proof assistants. Indeed, it seems feasible to get extraction from the Agda or Lean proof assistants to OCaml via Malfunction by implementing type and proof erasure from Agda or Lean to $\lambda_{\square}$.

1.4 Malfunction vs. Other Target Languages

In order to give a machine-checked implementation of extraction, we require a specification of the target language in Coq. This specification will remain part of the TCB: it has to be correct for the verification to be meaningful. Thus, it is desirable to have a *clear*, easy-to-review specification.

One path would be to specify the operational semantics of OCaml in Coq. Such a semantics could be obtained by translating the semantics of OCaml given in prose, or by reverse-engineering a specification from a compiler, interpreter, or an abstract machine for OCaml. However, the semantics of OCaml is complex, and it would require a major amount of trust to believe that the specification indeed is correct. Furthermore, several parts of OCaml are intentionally unspecified, *e.g.,* the evaluation order of function call arguments.

Another path would be to directly specify the operational semantics of Lambda, the intermediate language of the OCaml compiler. However, the semantics of the OCaml intermediate representation is intentionally unspecified (even the `-dlambda` option of the compiler to generate the intermediate representation from OCaml code is undocumented), and changes between releases.

Lastly, it is also desirable that the chosen semantics be *economical*, in the sense that it can be used in a verification project without generating an overhead which lets even easy tasks take months.

Thus, to have a clear and economical specification, it seems natural to choose Malfunction as target language for extraction: Malfunction has a clear semantics provided by an interpreter, it has been around the (scientific part of the) OCaml community for several years, and it can serve as a stable interface to deal with changes of the semantics of Lambda. Furthermore, Malfunction is fully specified: *e.g.,* evaluation order of operators is clearly specified to be left-to-right.

1.5 Plan of the Paper

In Section 2, we present the statement and main idea of the proof of the main correctness theorem. Then we proceed in Section 3 to a phase-by-phase description of each necessary transformation outlined in Fig. 1. We then discuss in Section 4 our contribution as a practical tool to replace current Coq's extraction to OCaml. Section 5 explains how we bootstrap our extraction mechanism to obtain a self-hosted compiler. We compare the performance of our extraction to Coq's extraction and CertiCoq in Section 6. Finally, we discuss related and future work and conclude.

Links to the available online formalisation are provided using the syntax "`myNotion` [myFile.v]". The links are voluntarily obfuscated for double blind but they are explicit enough so that they can be mapped to the anonymous supplementary material provided with this paper.

2 THE CORRECTNESS THEOREMS

In this section, we technically explain the *statement* of our two central theorems, providing key elements of their proofs. From a bird's-eye view, the first central theorem is a simple simulation theorem for a compiler, saying that if a Coq program t of first-order data type is convertible to an irreducible term v, then its translation to Malfunction evaluates to the value translation of v. A first-order data type (`firstorder_ind` [PCUICFirstOrder.v]) is a mutual inductive type whose constructors only have arguments of first-order data type. The value translation is a a simple function translating a constructor application to either an integer if all arguments are parameters of the constructor, or to an applied tagged block if they are not. The second central theorem deals with first-order functions, that is functions whose arguments and return value are of first-order data types. It states that the Malfunction program obtained by extracting a Coq first-order

```
Inductive term :=
| tRel (n : ℕ)
| 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 : ℕ) (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 : ℕ)
| tCoFix (mfix : mfixpoint term) (idx : ℕ)
| tInt (i : PrimInt63.int)
| tFloat (f : PrimFloat.float).
```

```
Inductive t :=
| Mvar of ℕ
| Mlambda of Ident.t list * t
| Mapply of t * t list
| Mlet of binding list * t
| Mnum of numconst
| Mstring of string
| Mglobal of Longident.t
| Mswitch of t * (case list * t) list
(* Numbers *)
| Mnumop1 of unary_num_op * numtype * t
| Mnumop2 of binary_num_op * numtype * t * t
| Mconvert of numtype * numtype * t
(* Vectors *)
| Mvecnew of vector_type * t * t
| Mvecget of vector_type * t * t
| Mvecset of vector_type * t * t * t
| Mveclen of vector_type * t
(* Lazy *)
| Mlazy of t
| Mforce of t
(* Blocks *)
| Mblock of int * t list
| Mfield of int * t
with binding :=
| Unnamed of t | Named of Ident.t * t
| Recursive of (Ident.t * t) list.
```

Fig. 2. Syntax of Coq (`term` [Ast.v]) and Malfunction (`t` [Malfunction.v]) formalised in Coq

function can be applied safely to *any* pieces of OCaml (compiled to Malfunction) that has the right corresponding type. In particular, we prove that the kind of counter-examples such as `assumes_purity` of Section 1.1 cannot be produced for first-order functions. So interoperability with unverified OCaml programs is safe for this fragment.

2.1 Malfunction and Coq

The internal language of the kernel of Coq (Fig. 2, top) is formalised in MetaCoq [Sozeau et al. 2019a] as the inductive type `term` [Ast.v]. On the computational level, it features basic constructs of functional language such as (DeBruijn) variables (`tRel`), lambda abstraction (`tLambda`), (n-ary) application (`tApp`) and a `tLetIn` construct. It also features inductive and co-inductive features, `tConstruct` for constructors of (co-)inductive types, `tCase` for pattern-matching, fixpoints (`tFix`) and co-fixpoints (`tCoFix`), and a notion of projections (`tProj`) which corresponds to a negative presentation of record types. Coq also features constructions that are very specific to dependent type theory. In particular, types appear in the syntax of terms and can be manipulated as any other terms. Thus, it features terms `tSort` for universe, `tInd` for inductive types (it takes as arguments on `inductive` definition and a universe `Instance.t`) and `tProd` for dependent function types and a `tCast` function to explicitly cast a term to a given type. Finally, Coq features a primitive notion of integers (`tInt`) and floats (`tFloat`). The syntax comes with a notion one-step reduction `red1` [PCUICReduction.v] , noted $\Sigma ; \Gamma \vdash t \rightsquigarrow t'$, of a term `t` into `t'` in the `global_env_ext` [PCUICAst.v] Σ and in the local context Γ. The reflexive, transitive closure of one-step reduction is noted $\Sigma ; \Gamma \vdash t \rightsquigarrow^* t'$. It also comes with a specification of `typing` [PCUICTyping.v], $\Sigma ; \Gamma \vdash t : T$. All of these definitions are quite complex in detail, but a deep understanding is not required to understand the rest of this paper and we refer the interested reader to Sozeau et al. [2019b] for further explanations.

Malfunction (Fig. 2, bottom) specifies the following syntactic constructs relevant for extraction: named variables, *n*-ary abstraction and application, mutual recursion via `Mlet` rec, tagged *n*-ary `Mblock`, an operation to project out the *i*-th field of a block, and a `Mswitch` statement branching over the tag of a block. Besides Malfunction features lazyness, primitive integers and mutable vectors that are not relevant to extraction from Coq.

Operationally, Malfunction and Coq are relatively simple. However, since Coq is a dependent type theory, it has first-class types and proofs that can play roles in computations and have no

counterpart in MALFUNCTION, *i.e.,* they have to be erased. The further semantic differences are subtle. We take care of them one-by-one, to avoid combinatorial explosion in proofs.

Still, combinatorial explosion lurks behind every corner. After all, we are dealing with the real semantics of COQ rather than an idealised type theory. We thus put focus on finding economical proofs, often proving a slightly weaker property than what could be proven which is however still strong enough to establish the wanted theorem. Standard techniques like using observational equivalence of programs are for instance not economical: An induction over the derivation of observational equivalence would have to deal with far more than 50 cases. Instead, our results are all stated w.r.t. a big-step evaluation relation of programs.

The reasons we restrict to values of first-order data types then become apparent: First, those are the only values where one can meaningfully talk about their normal form w.r.t. evaluation, because they are just constructor applications. Second, it becomes possible to prove that a weak call-by-value evaluation relation computes the same values—rather than just observationally equivalent ones—as COQ's unrestricted conversion becomes possible. Third, type and proof erasure is only complete in this fragment, because detecting whether a certain term is erasable sometimes requires evaluating it fully, and full evaluation being to inefficient is a central reason extraction processes are needed.

2.2 Operational Semantics of MALFUNCTION in COQ

MALFUNCTION [Dolan 2016] is a "thin wrapper" around the untyped intermediate language of the OCAML compiler. It comes with an interpreter written in OCAML, and a compiler hooking into the OCAML compiler's pipeline to either produce bytecode or native `cmx` files. Thus, it can make use of OCAML's optimisations. Both interpreter and compiler expect its input in s-expression format. As such, MALFUNCTION is supposed to be easy to generate rather than easy to read, and thus fits our purpose perfectly. Explicitly, whenever the interpreter computes a value, the compiled code is expected to produce the same value. Thus, we can derive MALFUNCTION's semantics in COQ purely from the interpreter, and as long as the derived semantics indeed matches, we have a guarantee that the compiled code behaves correctly by adding the MALFUNCTION compiler to the intermediate representation (which is essentially the identity function) and the OCAML compiler to our TCB.

The interpreter is defined in less than 300 lines of (impure) OCAML code and serves as the specification. It uses an environment for variables and a higher-order abstract syntax (HOAS) representation [Pfenning and Elliott 1988] of values for function thunks, *i.e.,* reuses OCAML functions to represent MALFUNCTION function thunks. Most other features of MALFUNCTION are specified using the corresponding feature in OCAML and mutable references: The operational semantics of the `Mlazy` constructor is defined using the corresponding `lazy` primitive in OCAML, mutable vectors are defined using mutable references and for `let rec` constructs, the mutual recursive knot is tied via references.

To define a specification for MALFUNCTION in COQ, we need to define a semantics matching the interpreter in COQ, and for this, we cannot rely on impure features of the language. This means that we need to make explicit the notion of `heap` [SematnicsSpec.v] in the operational semantics. To make sure that our notion of semantics is closely related to the initial version in OCAML, we proceed in two steps.

First, to disentangle the (non-terminating) interpreter from our formal proof, we provide an inductive, weak call-by-value evaluation relation `eval` [SemanticsSpec.v] in COQ, closely following the OCAML interpreter, but using the heap explicitly instead of relying on imperative feature of the language. This relation requires a local environment `locals` to be defined on open terms. We simply specify it using a function in COQ:

```
Inductive eval (locals : Ident.t → value) : heap → t → heap → value → Prop := ...
```

We also provide a functional interpretation of `let rec` instead of relying on the imperative one. It depends on an explicit strictly-positive representation of values.

Second, as a sanity check, we define a Coq function acting exactly as the interpreter for Malfunction implemented in OCaml, with the difference that we use an explicit heap and provide a machine-checked proof that whenever a program evaluates to a value for this predicate, then the interpreter computes this value. This is only possible by switching off termination checking in Coq locally because the evaluated term may be non terminating so there is no guarantee that the interpreter will terminate in Coq.

Then, to trust that the resulting semantics matches the intended semantics as specified in Malfunction (i) the reference interpreter of Malfunction written in OCaml has to be correct w.r.t. the compiler, (ii) our pure Coq implementation of the interpreter has to be correct, which can be validated by extracting it to OCaml and unit testing against the reference interpreter.

The unsafe proof that our relational semantics relates to the Malfunction interpreter is just a sanity check that our formal semantics relates to the proposal of [Dolan 2016] and is not required in our correctness theorems. Therefore, the fact that it switches off the termination checker is not crucial. However, it should be noticed that by making this proof, we actually found a bug in the bound check for vectors of Malfunction reference interpreter. The interpreter was performing an out of bounds vector access and subsequently crashing rather than reporting the access. We also discovered a miscompilation of pattern matchings in the Malfunction compiler due to a change in OCaml 4.14+`flambda` (compiled programs were computing the wrong value or even segfaulting).

2.3 Extraction Theorem For First-Order Data Types

We now explain the exact statement of the first correctness theorem about terms of first-order data type detailing the exact pre-conditions.

We use Coq syntax to ease the task of ensuring that the theorem is exactly as claimed also for readers with less experience reading Coq code of the theorem `verified_malfunction_pipeline_theorem` [Pipeline.v].

We assume that we are working in a well-formed global environment containing declarations of definitions and inductive types, where all definitions are η-expanded. We explain the need for η-expanded terms in the next section.

```
Variable Σ : global_env_ext.
Variable ΣH : wf_ext Σ.
Variable Σexp : expanded_global_env Σ.
```

We then fix a Coq-term `t`, which is also η-expanded and whose type in this environment is a first-order data type `i` at some universe level `u` with parameters and arguments `args`:

```
Variable t : term.
Variable expt : expanded Σ [] t.
Variables (i : inductive) (u : Instance.t) (args : list PCUICAst.term).
Variable fo : firstorder_ind Σ (firstorder_env Σ) i.
Variable typing : Σ ; [] ⊢ t : mkApps (tInd i u) args.
```

To simplify the proof slightly, we also assume that all inductives in the environment have no parameters. This assumption could be removed at the cost of a slightly more tedious proof, we plan to do so for the final version of this paper.[4]

[4]Note that the restriction however does not really inhibit the expressivity of the theorem, since instead of parameters all inductives could simply declare their parameters as indices.

```
Variable noParam : ∀ i mdecl, lookup_env Σ i = Some (InductiveDecl mdecl) → ind_npars mdecl = 0.
```

We then assume that t reduces using Coq's reduction to a term v that is irreducible. Note that Coq's reduction relations is maximally general and thus agnostic towards the specific reduction order used.

```
Variable v : term.
Variable v_red : Σ ; [] ⊢ t ⇝* v.
Variable v_irred : ∀ t', ( Σ ; [] ⊢ v ⇝ t' ) → False.
```

We then define the environment Σ' as the result of applying the middle-end of the extraction pipeline to the environment Σ and assume that there is a Malfunction environment containing exactly the values of every definition of Σ' compiled to Malfunction.

Lastly, we assume that after erasure, only extractable inductive types remain in the global environment. Concretely, this enforces some numerical maximums on number of constructor and number of arguments of constructors and that the environment contains no axioms. This numerical checks are here because for instance Coq's inductive types can have as many constructors as needed, whereas in OCaml and Malfunction there are explicit bounds on their numbers. For technical reasons, we prefer to deal with it as an axiom that is invoked exactly once in the proof of the erasure pipeline. This approach allows us to still get an (unsafe) extraction procedure for Coq's terms that do not meet those restriction. See Section 3 for more details.

We can then prove that in this environment, the compilation of t evaluates to the value translation of v in Malfunction, leaving the heap h it is run in unchanged:

```
Theorem verified_malfunction_pipeline_theorem : ∀ (h:heap),
 eval Σ' empty_locals h (compile_malfunction_pipeline expt typing) h (compile_value_mf Σ v).
```

In the above, compile_malfunction_pipeline is the function compiling a Coq term to a Malfunction term and compile_value_mf is the function compiling a Coq first-order value (*i.e.,* a value in a first-order data type) to a Malfunction value. The function empty_locals is used to encode the empty local environment and returns an error on every identifier.

For instance, the irreducible Coq term @cons 𝔹 true (@cons 𝔹 false (@nil 𝔹)) (*i.e.,* the list [true, false]) is translated to block (tag 0) 0 (block (tag 0) 1) 0, because cons is the first constructor of list with arguments (thus tag 0), true is the first constructor of 𝔹 not taking arguments (thus 0), false is the second constructor of 𝔹 not taking arguments (thus 1), and nil is the first constructor of list not taking arguments (thus 0). This semantics follows OCaml's internal representation of values in order to enable interoperability.

2.4 Extraction Theorem For First-Order Functions

Outside the first-order data type fragment, there is no hope to state and prove a meaningful theorem in terms of evaluation to particular values because there is no control on the translation of the body of functions, which are open terms. But we can however characterise the compilation of first-order functions in terms of interoperabilty—or to phrase it in a more foundational manner, in terms of realisability semantics. To this end, we define a notion of values realising in Malfunction a first-order data type coming from Coq. Concretely, we define a notion of OCaml type, camlType [RealizabilitySemantics.v], that only contains function types (Arrow) and algebraic data types (Adt).

```
Inductive camlType : Set :=
  Arrow : camlType → camlType → camlType
 | Rel : ℕ → camlType
 | Adt : kername → ℕ → list camlType → camlType.
```

The type `Rel n` refers to the n-th mutually defined data type in an `Adt`. We can construct a function `CoqType_to_camlType` [Firstorder.v] that builds an `Adt` from any first-order data type of Coq. Finally, to state the correctness theorem for firstorder function, we need to define what it means for a value `v` to be a realiser of `adt` at number `ind`, noted `v ⊩ (adt , ind)`. The number specifies which type of the potentially mutually defined `Adt` is realised.

Then, with similar hypothesis as in Section 2.3, we can prove the following interoperability theorem (`interoperability_firstorder_function` [Firstorder.v][5]) for a first-order function `f`. To reduce technical details, we assume a function that maps one type of a mutual first-order inductive type `mind` to another—the generalisation to arbitrary first-order functions is mathematically straightforward, but contains a lot of tedious technical manipulations.

```
Theorem interoperability_firstorder_function :
 ∀ (typing : Σ ; [] ⊢ f : tProd na (tInd (mkInd kn ind) []) (tInd (mkInd kn ind') [])),
 let adt := CoqType_to_camlType mind fo in
 let compile_f := compile_malfunction_pipeline expt typing in
 ∀ arg : t,
   (∀ (h h' : heap) (v : value), eval [] empty_locals h arg h' v → v ⊩ (adt , ind)) →
    ∀ (h h' : heap) (v : value), eval [] empty_locals h (Mapply (compile_f, [arg])) h' v → v ⊩ (adt , ind').
```

The crux of the proof lies in three main ingredients. First, se show that compilation to Malfunction is modular in the sense that a non-erasable application (e.g., of firstorder type) in Coq is compiled to the application of the compiled function to its compiled argument in Malfunction. Second, it uses the fact that *any* value realising an `Adt` coming from a first-order data type can be obtained by the compilation of a Coq term (`camlValue_to_CoqValue` [Firstorder.v]), *i.e.,* we prove a decompilation theorem from Malfunction values to Coq values. Lastly, it crucially uses the fact that the evaluation of compiled terms does not change the heap; *i.e.,* compilation to Malfunction produces *pure* terms.

Note that the second aspect is not valid outside the first-order fragment, because the function `impure` defined in Section 1.1 does not correspond to the compilation of any Coq function.

3 THE MAN IN THE MIDDLE: $\lambda_\square$, AN INTERMEDIATE PLATFORM FOR EXTRACTION

In this section, we describe the middle-end of the extraction process. The language $\lambda_\square$ was introduced by Letouzey [2004] as an operational model of Coq after computationally irrelevant parts—*i.e.,* types and proofs—are all erased to the same particular value □. It was first mechanised by Sozeau et al. [2019b] in their correctness proof of type and proof erasure, with a focus on weak call-by-value semantics. We integrate this erasure procedure into a pipeline with multiple translation steps to finally extract to OCaml.

Those multiple steps are necessary because, like Coq, $\lambda_\square$ has higher-order constructors, structural fixpoints with principal arguments that have to reduce to a constructor for the fixpoint to unfold, and no dedicated evaluation strategy. On the other hand, constructors in OCaml are blocks (*e.g.,* `cons` by itself is not well-typed, only `cons(x,l)` is), `let rec` does not indicate a special principal argument, and evaluation is a variant of weak call-by-value reduction. These subtle differences pose challenges for machine-checked reasoning, since they are traditionally ignored in proofs on paper. We parametrise the evaluation relation of $\lambda_\square$ in various flags which can be used to alter it to avoid code duplication, and give separate correctness proofs while gradually translating the

[5] At the time of writing, the proof of this theorem is done over an abstract notion of compilation that satisfies the properties mentioned in the text. The formal connection with our concrete pipeline properties—in particular dealing with the concrete pre- and post-conditions of the pipeline—requires still few days of work.

```
Definition program env term := env * term.

Context {env env' env'': Type}.
Context {term term' term'': Type}.
Context {value value' value'': Type}.
Context {eval : program env term → value → Prop}.
Context {eval' : program env' term' → value' → Prop}.
Context {eval'' : program env'' term'' → value'' → Prop}.

Record Transform.t :=
{ name : string;
  pre : program env term → Prop;
  post : program' env' term' → Prop;
  transform : ∀ p : program env term, pre p → program' env' term';
  correctness : ∀ input (p : pre input), post (transform input p);
  obseq : program env term → program' env' term' →
        value → value' → Prop;
  preservation : ∀ p v (pr : pre p), eval p v →
    let p' := transform p pr in
    ∃ v', eval' p' v' ∧ obseq p pr p' v v' }.

Definition Transform.compose :
    (o : Transform.t env env' term term' value value' eval eval')
    (o' : Transform t env' env'' term' term'' value' value'' eval' eval''),
    (∀ p : program env' term', post o p → pre o' p) →
    Transform.t env env'' term term'' value value'' eval eval''.

Notation " o ▹ o' " := (Transform.compose o o' _)
```

```
Program Definition erasure_pipeline : Transform.t :=
(* Build an efficient lookup table *)
build_template_program_env ▹
(* Eta-expand constructors and fixpoint *)
eta_expand ▹
(* Casts are removed, application is binary,
   case annotations are inferred *)
template_to_pcuic_transform ▹
(* Erasure of proofs terms in Prop and types *)
erase_transform ▹
(* Simulation of the guarded fixpoint rules
   with a single unguarded one *)
guarded_to_unguarded_fix ▹
(* Remove all constructor parameters *)
remove_params_optimization ▹
(* Rebuild the efficient lookup table *)
rebuild_wf_env_transform ▹
(* Remove cases/projections on propositions *)
optimize_prop_discr_optimization ▹
(* Rebuild the efficient lookup table *)
rebuild_wf_env_transform ▹
(* Inline projections to cases *)
inline_projections_optimization ▹
(* Rebuild the efficient lookup table *)
rebuild_wf_env_transform ▹
(* First-order constructor representation *)
constructors_as_blocks_transformation.
```

Fig. 3. The erasure pipeline

differences between the languages away. The complete strung-together phases of the pipeline are described in Fig. 3.

Each phase is an instance of `Transform.t`, parameterised by a type of input and output programs (`program` and `program'`), values (`value` and `value'`) and their evaluation relations (`eval` and `eval'`). Each phase is a `transform` taking input programs respecting the precondition `pre` and producing output programs that validate the postcondition `post`, as the `correctness` theorem states. Additionally, programs that respect the precondition and evaluate to a value v get transformed into programs that also evaluate to an observationally equivalent value v'. In most cases, the observational equivalence of values will be syntactic equality between a value and a translated value, *i.e.,* `transform` v = v'. Two transform phases o and o' can be composed (noted o ▹ o') when the postcondition of o implies the precondition of o'.

The erasure pipeline as a whole transforms a program from TemplateCoq (an environment and term directly quoted from Coq) into an erased program, with term asts representing the value types and evaluation from TemplateCoq (weak call-by-value evaluation of Gallina terms) being simulated by evaluation of erased programs, where evaluation has no rule for pattern-matching on □ and uses an unguarded fixpoint expansion rule (i.e., one argument is enough to trigger fixpoint unfolding) and a representation of constructors as blocks.

In this section, we concisely outline the different phases, focusing on the pre-conditions for their correctness, the post-conditions they establish, as well as the evaluation relations and observational equivalence on values they use when it is not simply syntactic equality. A more detailed overview of the passes is given in Appendix A.

Eta-expand. This phase η-expands all fixpoints and constructors. For instance, the term `map` cons, which is a well-typed function in Coq but not in OCaml, is η-expanded to the convertible term

`map (fun x l ⇒ cons x l)` which is also well-typed in OCaml. Because MetaCoq does not include η-expansions in its theory yet, for reasons discussed by Lennon-Bertrand [Lennon-Bertrand 2022], the correctness proof of this phase is just assumed together with the correctness of the quotation phase. The pre-condition is well-typedness of `t` in the global environment Σ, written $\exists$ `T`, Σ ; [] ⊢ `t : T` in the Coq formalisation. The post-condition is well-typedness of the return term `t'` and η-expandedness of all constant definition in Σ and `t'`. Both evaluation relations are weak call-by-value evaluation of the formalisation of Coq in Coq.

Translation to PCUIC. PCUIC is the idealised calculus underlying Coq used in MetaCoq. Technically, this translation removes casts and makes application unary to ease meta-theoretical proofs. The translation is always valid there is no pre-condition, and the post-condition is that the translated term and environment do not contain casts. The main property of this phase is that it preserves well-typedness. The evaluation relation of the target is weak call-by-value evaluation of PCUIC.

Lets in constructor types. Constructors in Coq can be defined as `C (x1 : X1) (x2 := t2) (x3 : X3)` where `X3` can use both `x1` and `x2`, and dually, pattern-matching can bind these defined arguments, while these let-bindings are not actually stored in constructor applications. We translate away such bindings via substitution to ease several verifications later on. As far as we know, Coq is the only language supporting let-bindings in constructor types, which is a useful feature when specifying inductive relations. This translation brings the language closer to usual presentations of algebraic datatypes. The phase has no pre-condition and ensures that in the translated term, constructor types are functional and pattern-matching reduction only involves a simple substitution. It preserves well-typedness and η-expandedness of fixpoints and constructors and the evaluation relation is simplified. This phase is the least economical in terms of proof-engineering as we must show a type preservation theorem for the whole theory of PCUIC: the proofs amount to 10kLoC.

Type and proof erasure. This phase goes from PCUIC to $\lambda_\square$. We use the verified type and proof erasure procedure due to Sozeau et al. [2019b]. The pre-condition of this phase requires well-typedness of the term. The post-condition cannot be well-typedness anymore because $\lambda_\square$ is untyped and erasure breaks typing. However, to still control the shape of the output term, the post-condition gives a well-scopedness condition. This phase preserves η-expand-edness of constructors and fixpoints and the target evaluation relation is the weak call-by-value relation of $\lambda_\square$, with structural fixpoints, `match` on □, and higher-order constructors. The observational equivalence on values for this phase is more complex and makes use of an erasure relation, which only agrees with the syntactic equality on first-order values. This weaker notion of equivalence is however enough to get our final theorem because it will ultimately be restricted to first-order values.

Unary fix translation. The evaluation of fixpoints in Coq and up-to this point of the pipeline is performed by evaluating its structural argument to a constructor application and then unfolding the fixpoint. The structural argument is the one guaranteeing termination and thus other unfolding strategies are not safe. However, in OCaml, evaluation of fixpoints is performed in a similar way as β-reduction, by *always* evaluating its first argument to a value and then unfolding the fixpoint. This rule is called the "unary fixpoint rule". So the purpose of this phase, which is the identity on programs, is to justify that enabling the unary fixpoint rule in the target preserves evaluation of programs. It crucially relies on fixpoints being eta-expanded so that when they get unfolded in an evaluation path, we have a guarantee that their structural argument already evaluates to a constructor.

No parameters. This phase removes parameters from inductive types in the global context and from constructor applications. It requires η-expandedness of constructors to be performed. As an output, it guarantees that programs are well-formed, when the use of parameters is disallowed.

Remove match on □. This phase removes case analysis on □, the residue of erased computational case analysis on proofs, by replacing occurrences of the term `match □ with C a1 ... an ⇒ t end` immediately by t applied to n-many □. This is necessary, because we are going to implement □ as a recursive function in Malfunction, meaning a naive translation of the case analysis would produce segmentation faults. This phase has no pre- and post-conditions as it is purely syntactic. Its unique interest is to justify disabling the rule for `match` on □ in the target.

Inline projections. Primitive projections correspond to a negative treatment of pattern-matching when the inductive type under consideration is a record type. This produces more efficient code and enables η-conversion for record types that have at least one constructor. This transformation phase inlines primitive projections as `match`, it requires no pre-condition and guarantees syntactically that no projections are left in the translated program.

Constructors as blocks. Because constructors of inductive types are fully applied in OCaml and Malfunction, this phase translates any application of a constructor $C\,n\,[\,]$ (where n is the name and $[\,]$ is the a priori empty list of fields of the constructor) to a full list of arguments $a_1, \ldots, a_n$ to $Cn[a_1, \ldots, a_n]$. This transformation has no specific pre- and post-conditions, and its puporse is to enable the rules treating constructors as blocks in the evaluation of a program.

Enforce extractability. For extraction in Malfunction to be correct, we need to check that

(1) inductives have at most as many constructors as fit into OCaml's integer type,
(2) inductives have at most 200 constructors non-constant constructors,
(3) constructors have at most as many arguments as fit into OCaml's array type,
(4) no axioms occur in the global environment,
(5) no □, named or existential variables, projections, or co-fixpoints remain in the term or context.

Note that we explicitly assume these conditions are fulfilled for the proof an `Axiom` and do not abort extraction if they are not fulfilled. This is in order to ensure that also programs that do not fall in the scope of the correctness theorem can be extracted, especially ones which have axioms.

Implement □. This phase implements □ as a fixpoint that consumes its argument, *i.e.*, as `fix f x := f`. It requires well-formedness as pre- and post-conditions and ensures that □ does not appear anymore in the translated program.

Linearise case. Let-binds the discriminee of cases, *i.e.*, replaces `match d with ... end` by `let x := d in match x with ... end`. This makes the translation to `switch` and `field` later in Malfunction preserve complexity of programs. The reason we perform this translation at this stage is that it is easier to verify with variables represented as de Bruijn indices, but harder once variables are names as it will be the case in next phase.

Named variables and environment semantics. This phase translates terms with de Bruijn indices to non-capturing, named terms. For named terms, an environment semantics with a dedicated value type (containing amongst others closures and recursive closures) is used. We use two unfolding relations, unfolding respectively a named term or a value in an environment to a term with de Bruijn variables. This phase has no pre-condition and ensures as post-condtion that there exists a well-formed term s such that t (with names) unfolds to s. After this phase, evaluation is switched to a named environment semantics.

Global environments. Several of the passes need to do lookups on the global environment, *e.g.*, to compute the number of parameters of an inductive type. Global environments are represented as an association list of unique identifiers to declarations of constants and inductive types in MetaCoq. While this allows a straightforward formalisation of the well-formedness predicate on global contexts naturally representing dependencies between declarations, this is a very suboptimal datastructure to use when transforming programs, as they generally need to lookup information in the global environment, which can contain thousands of declarations. This problem is well-known

in the literature about compiler programming in Coq, with specialists resorting to highly optimised tree datastructures [Appel and Leroy 2023] to avoid performance issues inside Coq and after extraction. Hence we intersperse transformations that turn a well-formed environment represented as a list into a lookup table based on AVL trees that can be used for fast lookups. Some of our transformations, *e.g.,* erasure itself, are nested, so they need to follow the order of declarations given by the association list and likewise produce association lists as outputs, while relying on the efficient environment representation for lookups. A subtletly that appears during formalisation is that we generally need to show global environment weakening lemmas on our transformations so that we don't need to extend our efficient global environment structures after handling each entry in the environment but can leave it untouched for a whole pass. This solely rests on the fact that identifiers are globally unique in a well-formed environment, with this well-formedness condition being carried around throughout the pre- and post-conditions of all our transformations.

Compilation from $\lambda_\square$ to Malfunction. The last pass in our extraction pipeline replaces the case analysis `match x with C1 a1 ... an ⇒ t1 | ... end` by `switch x with C1 ⇒ (fun a1 ... an ⇒ t1) (field 1 x) ... (field n x) | ... end`, translates constructors without arguments to integers, constructors with arguments to `block (tag n) ...`, and mutual fixpoints to mutual `let rec` blocks. Its precondition is that the term is extractable, *i.e.,* the five conditions explained above for the enforce extractability phase. The post-condition ensures that the resulting Malfunction program is well-formed, meaning there are no free variables, all constants are declared, blocks are tagged with an integer less than 200, and every occurring `switch` does not have an empty list of branches.

4 VERIFIED EXTRACTION AS A DROP-IN REPLACEMENT

In this section, we discuss our contribution from the point of view of a practical tool, neglecting its formal guarantees, and thereby comparing it to Coq's current extraction to OCaml [Letouzey 2004] which is not formally verified.

Coq's current extraction has several features that we have not discussed until now:

(1) It allows the user to enable optimisations of the generated code via `Set Extraction Optimize`. Optimisations are *e.g.,* commutation of function application and `match`, simplifying `lets` where the bound term only occurs once in the body, etc.
(2) It generates a `.mli` interface by re-inferring types of the generated code using the type inference algorithm $\mathcal{M}$ [Lee and Yi 1998]. Because the code might be ill-typed in OCaml's type system, `Obj.magic` is heuristically inserted, resulting in the types of extracted code to contain `Obj.t`.
(3) It supports separate extraction to `.ml` files matching the module structure of the corresponding Coq code.
(4) It allows fine-tuning of the extracted code by the user providing `Extract Constant` and `Extract Inductive` directives. This is for instance used to map Coq's $\mathbb{B}$ type to the one of OCaml (which has an exactly swapped memory representation).
(5) It allows fine-tuning of the extracted code by the user providing a `Extract Inline` directive, meaning a constant is always unfolded.
(6) It extracts terms of type `{x : A | P x}` where `P` is a propositional predicate to terms of type corresponding to `A` in OCaml.

Regarding (1), the benchmarks in Section 6 indicate that the optimisations do not have a relevant impact on performance. Instead, they seem mainly to have readability and following standard certain OCaml programming paradigms of the generated code in mind. This is not relevant to our work because we provide low level code with formal guarantees instead.

Regarding (2), since we bypass OCaml's type checker, we only have to generate `.mli` files for the parts of the extracted code that can and should be exposed, *i.e.,* for first-order functions. We provide an `.mli` printer for such functions, which has a simple implementation since it has to solve a significantly simpler problem than the type reinference algorithm. Nevertheless, we support extracting higher-order code, but require the user to annotate the types in the `.mli` file manually.

Regarding (3), separate extraction is mainly a concern for readability of code, which we have not considered. However, we could easily implement it if users demand for it.

Regarding (4), the reason we do not provide such directives is that it is almost impossible to guarantee their correctness. However, it would not be hard to do so if users demand for it—with the cost that it is easy to break correctness. There are three frequent use cases:

First, to direct extraction to extract Coq's $\mathbb{B}$ to OCaml's `bool`, which is a subtly different type: Coq's `Inductive` $\mathbb{B}$:= `true` | `false` has `true` as first constructor, which will thus be represented as integer 0 in memory, whereas OCaml's `type bool = false | true` has `false` as first constructor, thus `true` being represented as integer 1. Here, we advocate for using such type conversion explicitly when calling and receiving arguments of an extracted Coq function explicitly, *i.e.,* by converting between the two types.

Secondly, to direct extraction to extract Coq's unbounded $\mathbb{N}$ to OCaml's `int`. However, this extraction is plain wrong, because `int` is not unbounded. It is however still correct on small enough numbers. Here, we thus advocate for proving this property for small enough numbers explicitly on the Coq side. This is possible because Coq features a primitive integer type, see below.

Thirdly, to direct extraction to extract Coq's primitive integer and float types to the corresponding types in OCaml. This use case is already supported since Coq's primitive types are declared as axioms, and our extraction just maps all axioms to a call to a module called `Axioms` that has to be provided by the user. We however lack a correctness proof, see Section 8.

Regarding (5), inlining definitions during extraction is mainly concerned with readability of code and thus not relevant for us. Where inlining is performance-critical, it will likely be performed by OCaml's compiler.

Regarding (6), unboxing such types would potentially even be beneficial for performance, see Section 8 on future work.

Thus, we estimate that our verified re-implementation of Coq's extraction is almost ready to replace Coq's current extraction mechanism on safe use-cases.

5 BOOTSTRAPPING A SELF-HOSTING COMPILER

A compiler is called *self-hosting* if it is implemented in the same language it is a compiler for. This is the case for us: Our extraction process is implemented in Coq, and translates Coq to Malfunction. The process to obtain a first executable (*e.g.,* by using a different) compiler is called *bootstrapping*.

Since Coq is a dependent type theory, already its type checker comes with the ability to execute programs. Thus, in principle, two ways of bootstrapping are conceivable:

First, using the existing, unverified extraction from Coq to OCaml (implemented in OCaml). This means that in the bootstrapping process, this unverified extraction algorithm becomes part of the TCB.

Secondly, by executing the extraction pipeline live in Coq using Coq's execution mechanisms once on itself. Unfortunately, execution in Coq is not very performant (which, of course, is the main reason why extraction is needed). In this case, we could not manage to make the process terminate, because Coq is exhausting its stack. Even setting `ulimit -s unlimited` does not help on our working machines—but it would be an interesting experiment to let it run longer on a more performant server.

Our bootstrapping method via the existing extraction to OCaml works. However, we have to patch the generated OCaml files to circumvent some bugs of the current extraction mechanism. Patching is fine because we have no guarantee on this phase anyway, but it illustrates the limitations of current extraction mechanism. By statically adding the resulting `.mlf` file to the repository, and using it to re-extract new future versions of the extraction pipeline, the use of the existing extraction can be limited to the initial bootstrapping once and for all, meaning it is not crucial that it is kept a part of Coq in the future.

Note that our correctness theorem does not yet apply to the bootstrapped extraction pipeline. Concretely, we would have to (i) slightly change the input format to be a first-order data type by modelling annotated universe instances with potentially empty lists instead of provably non-empty lists, and then do a validation pass on the new input format ensuring the lists are not empty. (ii) cover primitive operations for integers and floats in the theorem, because they are needed for printing integers and floats to strings in the output.

Especially the second part requires a global change to MetaCoq, most crucially to its operational semantics and typing judgment, so we leave it for future work.

6 BENCHMARKS

We compare the performance of extracted code comparing Coq's current extraction to OCaml, CertiCoq's extraction to C, and our verified extraction to Malfunction. All benchmarks are run on a 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz and Coq version `8.17.1`. We use the same benchmark set as used in the evaluation of CertiCoq by Paraskevopoulou [2020], namely:

- `demo1`, appending two lists of booleans of length 500 and 300.
- `demo2`, mapping negation on a list of 100 booleans (using the higher-order function `map`).
- `list_sum`, computing the sum of a list containing 100 times the number 1.
- `vs_easy`, compiling the VeriStar [Stewart et al. 2012] prover for a fragment of separation logic and deciding the validity of an easy entailment.
- `vs_hard`, same as `vs_easy`, but with a harder entailment.
- `binom`, constructing two binomial queues, merging them, and finding the maximum element, using a verified implementation of binomial queues [Appel 2023; Vuillemin 1978].
- `color`, coloring a graph using a formally verified implementation of the Kempe/Chaitin algorithm [Appel 2023; Chaitin et al. 1981]. Note that this benchmark is not executed with OCaml extraction, because the generated code contains type errors.
- `sha`, computing the SHA-256 hash of a string consisting of 484 characters.

For OCaml we use version 4.14.0 with `flambda` enabled and both the bytecode and native-code compilers. We set and unset the (heuristic, unproved) optimisations Coq's current extraction performs on the code. For CertiCoq, we use commit version `a5323b8` from November 2023. We compile the resulting code with `gcc` with both no optimisations and `gcc -O1`. For verified extraction to Malfunction, we compare performance by disabling optimisations when calling the OCaml compiler pipeline, or calling it with the default of two rounds of optimisations.

In total, we compile code by eight different means (the results are depicted in Table 1): By Coq's current extraction to OCaml, with `Set Extraction Optimize` and using the `ocamlc` bytecode compiler (1st column) or the `ocamlopt` native-code compiler (2nd column). By Coq's current extraction to OCaml, with `Unset Extraction Optimize` and using the `ocamlc` bytecode compiler (3rd column) or the `ocamlopt` native-code compiler (4th column). By CertiCoq, using `gcc` for C compilation (5th column) or `gcc -O1` for C compilation (6th column). By verified extraction to Malfunction, using `malfunction cmx -O0` compilation (7th column) or `malfunction cmx -O2` compilation (8th column).

	ocamlc Extraction Optimize	ocamlopt Extraction Optimize	ocamlc	ocamlopt	CertiCoq gcc	CertiCoq gcc -O1	mlf -O0	mlf -O2
demo1	1.4	1.2	1.4	1.1	2.7	1.8	1.1	1.2
demo2	0.6	0.4	0.6	0.3	0.6	0.5	0.4	0.4
list_sum	5.2	1.8	4.2	1.7	4.1	5.2	1.9	1.7
vs_easy	1196.3	59.5	1390.6	74.4	190.2	154.2	181.1	65.5
vs_hard	5572.0	707.5	6331.9	684.9	1429.5	1268.6	1635.0	951.8
binom	971.0	182.5	963.1	141.5	166.6	174.5	150.4	150.1
color	X	X	X	X	785.5	706.1	1068.2	651.8
sha_fast	3076.5	1089.8	3167.9	914.9	1329.4	1306.2	1239.8	985.9

Table 1. Time in milliseconds for 50 runs of the individual benchmarks.

We observe that compiling our code with the default optimisations has performance very similar to using the OCaml native-code compiler `ocamlopt`. On some benchmarks our extraction produces slightly faster code `vs_easy`, whereas on others it produces slightly less performant code. The performance difference can be explained for instance by differences in pattern matching compilation, where the OCaml compiler tries to optimise heavily and Malfunction performs some mild optimisations. In all cases, we outperform CertiCoq with `gcc`.

Unrelated to the contributions of this paper, we observe that `Set Extraction Optimize` has a somewhat unpredictable effect on the performance: Sometimes it leads to a slight performance gain (*e.g.,* for the `vs_easy` benchmark), sometimes it leads to a slight slowdown (*e.g.,* for the `binom` benchmark). However, this means that the current optimisations are mainly of cosmetical nature, *i.e.,* are concerned with obtaining more efficient code. The notable exception could be unboxing dependent pair types {x : A | P x} with a propositional predicate P (which can be seen as refinement or subset types) such that elements of this type are treated as elements of type A, rather than as pairs of an element of type A and a □. In future work, we would like to analyse whether performing such an optimisation at type and proof erasure time is beneficial.

7 RELATED WORK

There are several ongoing projects on obtaining correct programs in industrial programming languages like OCaml, Haskell, or C from programs in richly typed dependent programming languages like Coq, Lean, Idris, and Agda, or the simple type theories underlying Isabelle/HOL, HOL light, or HOL4. Our project is closely related to all of them, but has a separable motivation. We want to replace the current extraction process of Coq to OCaml [Letouzey 2004] with a verified process, without causing maintenance issues in the projects using extraction.

The ConCert project [Annenkov et al. 2021, 2020] is a platform to extract Coq code to ML-like and blockchain languages. Since most target languages they consider are not specified and their implementation is under constant evolution, the final phase of their extraction process is trusted pretty-printing.

The CertiCoq project [Anand et al. 2017; Appel et al. 2022] aims at extracting Coq code to the C light interface of CompCert. The objectives of the project are different in that it is not supposed to replace Coq's extraction, but rather supplement it. CertiCoq relies on the $\lambda_{\square}$ pipeline presented in and contributed by this paper. Most but not all following phases of CertiCoq are verified, meaning it still lacks an end-to-end theorem.

The Candle project [Abrahamsson et al. 2022] provides a verified implementation of HOL light in CakeML, and automatic extraction from HOL4 to CakeML is available [Myreen and Owens 2014]. Similarly, functions defined in Isabelle can be extracted [Hupel and Nipkow 2018], and a proof

checker of Isabelle has been verified in Isabelle based on this [Nipkow and Roßkopf 2021]. Œuf [Mullen et al. 2018] formalises a subset of Coq and provides a verified compiler from this language to Assembly code. Pit-Claudel et al. [Pit-Claudel et al. 2022] report on a proof-producing translation of functional models into low-level code via relational compilation.

8 FUTURE WORK

Our current extraction pipeline makes the axiomatic assumption that η-expanding fixpoints and constructors does not change the values of first-order data types w.r.t. weak call-by-value evaluation. The easiest way to prove this is to extend MetaCoq's reduction relation with a suitable notion of η-conversion. The difficulties of this are discussed by Lennon-Bertrand [2022].

Regarding features of Coq we support, we are missing two central syntactic constructs: Primitive operations on integers, floats, and arrays and co-fixpoints. For the former, one would have to extend the reduction semantics of MetaCoq with primitive operations, and thread the corresponding correctness proofs through the pipeline. In the target, one then has to prove that these functions can be correctly implemented using certain Malfunction programs. Most likely, a program logic for Malfunction would be helpful here, or alternatively a program logic for an envisioned variant of OCaml with formal semantics that can be compiled to Malfunction. For the latter, there are different options: One is to implement co-fixpoints without sharing as thunked fixpoints. A second would be to use Malfunction's `lazy` construct, at the cost of letting extraction produce impure programs altering the heap.

Regarding an extension of our correctness theorem, we would like to work in the direction of proving a theorem of interoperability with an arbitrary higher-order fragment as long as the only effect used by the unverified OCaml program is nontermination, *i.e.,* requiring that the heap remains unchanged. However, such properties cannot be enforced by OCaml's type-checker automatically, meaning they would have to be manually verified by the user.

Regarding optimisations implemented in Coq's current extraction mechanism, our benchmarks seem to indicate that they have no impact on performance. Thus, the only optimisation we plan on implementing is the one unboxing elements of subset types, *i.e.,* extracting terms of type {x : A | P x} to the type corresponding to A.

Lastly, the middle-end of our extraction pipeline based on $\lambda_\square$ could be used as a more general platform for verified extraction. It is conceivable to implement type and proof erasure from other type theory-based proof assistants such as Lean or Agda to $\lambda_\square$, and it is conceivable to implement back ends to other programming languages than OCaml/ Malfunction. The CertiCoq project in its current stage can already be seen as such a back end to C.

9 CONCLUSION

We contribute a verified extraction process from Coq to the OCaml ecosystem via Malfunction, implemented and verified in Coq itself. Our proofs consider the actual semantics of Coq and the actual semantics of Malfunction, with no idealisations. Including the reusable intermediate pipeline based on $\lambda_\square$, our contributed code has about 40k lines of code.

Our central theorem covers first-order function, *i.e.,* functions that take elements of ground type as input and output. We observe that this is the strongest possible theorem for interaction with unverified programs type-checked by the OCaml type checker [6], since even slightly more higher-order functions cannot be safely called from OCaml programs under all circumstances.

Besides the specific results for extraction of programs from Coq to OCaml, we believe that the exposition has several valuable contributions: (i) it discusses proof techniques that should scale to

[6]With the exception of also covering Coq's primitive integers, floats and arrays, which we leave to future work.

get similar results for other proof assistants and other target languages, (ii) it gives an exposition how to extract from type theory to (call-by-value) programming languages in general, and makes the invariants explicit and (iii) it collects known techniques for compilation, which are however nowhere presented in a unified, accessible way.

REFERENCES

Oskar Abrahamsson, Magnus O Myreen, Ramana Kumar, and Thomas Sewell. 2022. Candle: A Verified Implementation of HOL Light. (2022). https://doi.org/10.17863/CAM.84121

Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *CoqPL*. Paris, France. http://conf.researchr.org/event/CoqPL-2017/main-certicoq-a-verified-compiler-for-coq

Danil Annenkov, Mikkel Milo, and Bas Spitters. 2021. Code Extraction from Coq to ML-like languages. (2021). https://github.com/AU-COBRA/ConCert/blob/master/papers/ML-family.pdf

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

Andrew Appel, Yannick Forster, Anvay Grover, Joomy Korkut, John Li, , Zoe Paraskevopoulou, Kathrin Stark, and Matthieu Sozeau. 2022. CertiCoq (GitHub repository). (2022). https://certicoq.github.io

Andrew W. Appel. 2023. *Verified Functional Algorithms*. https://softwarefoundations.cis.upenn.edu/vfa-current/index.html Version 1.5.4.

Andrew W. Appel and Xavier Leroy. 2023. Efficient Extensional Binary Tries. *Journal of Automated Reasoning* 67 (2023), article 8. https://doi.org/10.1007/s10817-022-09655-x

Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. *SIGPLAN Not.* 49, 1 (jan 2014), 87–100. https://doi.org/10.1145/2578855.2535876

Timothy Bourke, Lélio Brun, Pierre-Evariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A Formally Verified Compiler for Lustre. In *PLDI 2017 - 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Barcelone, Spain. https://hal.inria.fr/hal-01512286

Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. 1981. Register allocation via coloring. *Computer languages* 6, 1 (1981), 47–57.

Stephen Dolan. 2016. Malfunctional programming. In *ML Workshop*. https://www.mlworkshop.org/2016-9.pdf

Derek Dreyer. 2018. The Type Soundness Theorem That You Really Want to Prove (and Now You Can). Milner Award Lecture at POPL 2018. https://www.youtube.com/watch?v=8Xyk_dGcAwk

Carlos Eduardo Giménez. 1996. *Un calcul de constructions infinies et son application à la vérification de systèmes communicants*. Ph. D. Dissertation. Ecole Normale Supérieure de Lyon. ftp://ftp.inria.fr/INRIA/LogiCal/Eduardo.Gimenez/thesis.ps.gz

Lars Hupel and Tobias Nipkow. 2018. A verified compiler from Isabelle/HOL to CakeML. In *European Symposium on Programming*. Springer, 999–1026.

Oukseh Lee and Kwangkeun Yi. 1998. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems* 20, 4 (July 1998), 707–723. https://doi.org/10.1145/291891.291892

Meven Lennon-Bertrand. 2022. À bas l'η — Coq's troublesome η-conversion. In *The first Workshop on the Implementation of Type Systems (WITS)*. https://www.meven.ac/documents/22-WITS-abstract.pdf

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, 42–54. http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf

Pierre Letouzey. 2004. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq*. Thèse de Doctorat. Université Paris-Sud. http://www.pps.jussieu.fr/~letouzey/download/these_letouzey.pdf

Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. 2018. Œuf: minimizing the Coq extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 172–185. https://doi.org/10.1145/3167089

Magnus O Myreen and Scott Owens. 2014. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming* 24, 2-3 (2014), 284–315.

Tobias Nipkow and Simon Roßkopf. 2021. *Isabelle's Metalogic: Formalization and Proof Checker*. Springer International Publishing, 93–110. https://doi.org/10.1007/978-3-030-79876-5_6

Zoe Paraskevopoulou. 2020. *Verified Optimizations for Functional Languages.* Ph. D. Dissertation. USA. Advisor(s) Appel, Andrew. AAI28153473.

Christine Paulin-Mohring. 1996. *Définitions Inductives en Théorie des Types d'Ordre Supérieur.* Habilitation à diriger les recherches. Université Claude Bernard Lyon I. http://www.lri.fr/~paulin/PUBLIS/habilitation.ps.gz

F. Pfenning and C. Elliott. 1988. Higher-order abstract syntax. *ACM SIGPLAN Notices* 23, 7 (June 1988), 199–208. https://doi.org/10.1145/960116.54010

Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. 2022. Relational compilation for performance-critical applications: extensible proof-producing translation of functional models into low-level code. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 918–933. https://doi.org/10.1145/3519939.3523706

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2019a. The MetaCoq Project. (April 2019). http://www.irif.fr/~sozeau/research/publications/drafts/The_MetaCoq_Project.pdf Submitted.

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

Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Botsch Nielsen, Nicolas Tabareau, and Théo Winterhalter. 2023. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. (April 2023). https://inria.hal.science/hal-04077552 working paper or preprint.

Matthieu Sozeau and Cyprien Mangin. 2019. Equations Reloaded. *PACMPL* 3, ICFP (August 2019), 86–115. https://doi.org/10.1145/3341690

Gordon Stewart, Lennart Beringer, and Andrew W. Appel. 2012. Verified heap theorem prover by paramodulation. *ACM SIGPLAN Notices* 47, 9 (Sept. 2012), 3–14. https://doi.org/10.1145/2398856.2364531

Jean Vuillemin. 1978. A data structure for manipulating priority queues. *Commun. ACM* 21, 4 (April 1978), 309–315. https://doi.org/10.1145/359460.359478

A DETAILS ON $\lambda_\square$ PASSES

We here give more details for selected passes of the middle-end of our extraction process.

A.1 Eta-Expansion

The correctness proof of the translation of higher-order constructors to constructors as blocks in Appendix A.6 is straightforward if constructor applications are eta-expanded, in the sense that constructors are never only partially applied. Similarly, the translation from structural fixpoints to unary fixpoints benefits from fixpoints being syntactically fully applied.

It is straightforward to implement a translation which on the fly eta-expands constructors and fixpoints. However, lacking η-expansion in MetaCoq, proving correctness is not for free. Three strategies suggest themselves:

(1) Eta-expand on the fly *e.g.,* during type and proof erasure, and prove that the result computes correctly, which will require defining observational equivalence in the (untyped) target language $\lambda_\square$.
(2) Eta-expand Coq terms first, and prove that the result is observationally equivalent, which will require defining observation equivalence in the source, *i.e.,* either TemplateCoq or PCUIC.
(3) Use a translation validation approach by expanding the Coq term and using η-conversion of the Coq kernel (not the verified MetaCoq conversion check).

Inspired by the ConCert project [Annenkov et al. 2020], which uses a similar approach for similar reasons, we choose to work with the last option. Once MetaCoq supports η-expansion, the translation validation can be replaced by a correctness proof of the translation following option (2).

The crucial theorem proves that the result of the translation is expanded, with the following definition (where we only show the interesting cases):

The `expanded` predicate is parameterised by a context Γ that records for each de Bruijn variable how much it should be eta-expanded: this is used to force the eta-expansion of fixpoint variables. For constructors, we ensure that they are applied to *at least* the sum of the number of parameters of their inductive type and the number of assumptions of their argument context (definitions in the context of arguments do not appear at applications). Together with the typing precondition of `eta_expand_expanded`, we can strengthen this claim and guarantee that each constructor takes exactly the right number of arguments. However, `expanded` is stable by substitution while the more stringent requirement is not, unless we only work with well-typed terms. The precondition on the contexts Γ' and Γ ensures that each variable that is expanded `n` times indeed has a universally quantified type which allows at least `n` applications. Finally, the condition on Σ`g` provides the connection between the association list of declarations Σ and the efficient lookup table Σ`g`: they are observationally equivalent with respect to lookups.

The η-expansion functions themselves are relatively straightforward: `eta_single` builds an `n` -ary η-expansion of a term `mkApps t args` starting with its type `ty`. Indeed we need to decorate the newly introduced λ-abstractions with their appropriate type for the expansion to be type preserving. For constructors, we lookup their type and expected number of arguments from the global environment in `eta_constructor`. For fixpoints, ín `eta_fixpoint`, we only need to η-expand them up-to their recursive argument, which is stored as `d.(rarg)` in the fixpoint block definition. Finally, the `eta_expand` function maps through a term and applies η-expansion in the variable, constructor and fixpoint cases.

A.2 Type and Proof Erasure

The next pass is type and proof erasure, described by Letouzey [Letouzey 2004] and proved correct in Coq for the case of weak call-by-value evaluation by Sozeau et al. [Sozeau et al. 2019b, 2023]. We

briefly recall the structure of the proofs and the relevant theorems, but use mathematical syntax rather than Coq code to indicate that the results do not form a part of the contribution of the present paper.

Following Letouzey, we define an erasure function $\mathcal{E}$ yielding a $\lambda_{\square}$ term provided a well-typed term of Coq's type theory. However, this erasure function is not closed under (weak call-by-value) evaluation, *i.e.,* if t evaluates to v, then not necessarily $\mathcal{E}t$ evaluates to $\mathcal{E}v$. Due to polymorphism such a property could only be reached by letting $\mathcal{E}$ perform (partial) evaluation on the source, which is contrary to the goals of extraction. Thus, Letouzey sets up a non-deterministic erasure relation which we write as "t erases to t'" and proves the following theorem, which Sozeau et al. verify in MetaCoq.

Theorem A.1. *Well-typed terms t erase to $\mathcal{E}t$, and if a well-typed term t evaluates to v and erases to t', then v erases to v' to which t' evaluates.*

The erasure function and relation only agree on terms if there is nothing to erase: this is the case for first-order values, *i.e.,* the values of Coq types which neither contain proofs nor functions. We say that a term is a first-order value if it is the application of a constructor to first-order values. We say that a type is first-order if it is inductive, and all its parameters, indices, and field types are first-order.

Lemma A.2. *(1) If t is of first-order type and evaluates to v, then v is a first-order value. (2) If v is well-typed, a first-order value, and erases to v', then $\mathcal{E}v = v'$.*

In general, reduction in Coq is not specified to follow any particular strategy or path. However, evaluation specified by Malfunction is weak (*i.e.,* no evaluation under function binders) and call-by-value (*i.e.,* arguments are always evaluated before a function call). We prove the following:

Theorem A.3. *If t is of first-order type and reduces to the irreducible term v, then t weak call-by-value evaluates to v. And in particular, then $\mathcal{E}t$ evaluates to $\mathcal{E}v$.*

A.3 Unary Fix Translation

The next pass is the identity as transformation, but changes a flag in the evaluation relation to use a unary rule for fixpoints, rather than a structural rule. Having a structural fixpoint rule with one argument being identified as the primary argument is one of the most special peculiarities of Coq, and, consequently, $\lambda_{\square}$.

Letouzey explains how this rule does not pose problems when extracting to OCaml and Haskell informally. He relies on two arguments [Letouzey 2004, §2.6.4]:

- that in the user syntax of Coq, the body of a fixpoint with the n-th argument being the principal argument has to start with n abstractions syntactically. While this is indeed true for the user syntax, it is not true for the type theory implemented in Coq, and thus not true for PCUIC. Here, the lambdas only have to occur after reduction.
- that one can implement a translation of general structural reduction to recursors by using the accessibility predicate Acc, as suggested by Paulin-Mohring [Paulin-Mohring 1996, p. 103] in 1996. However, with the current implementation of Coq's guardedness checker updated by Gimenez [Giménez 1996] in 1998, no such translation is currently known.

Instead of solving this generally hard problem in type theory, we work around it by observing that for eta-expanded fixpoints the two semantics indeed agree. Eta-expandedness of fixpoints was already established in the first pass, we make use of this as a pre-condition here. We omit details of the proof, which is straightforward.

Since the translation is the identity, we do not have to define any function and can immediately state the correctness theorem:

```
Lemma eval_opt_to_target Σ t v : isEtaExp_env Σ → wf_glob Σ → eval Σ t v → isEtaExp Σ [] t → eval Σ t v.
```

The crux of the proof is that if all fixpoint applications are η-expanded, then during weak call-by-value evaluation, if we encounter a fixpoint application it will always have the structural argument reduce to a value. As the fixpoint body start with at least as many λ-abstractions as the structural argument, the unary fixpoint expansion rule plus as many β-reduction steps simulates the guarded fixpoint rule.

A.4 Removal of Parameters

This pass removes parameters from inductives in the global context and from constructor applications. This is mainly an optimisation.

As the constructors are η-expanded and the parameters never participate to the reduction, they can be removed without affecting the evaluation. They just will not appear in values of inductive types anymore.

We do not go into details.

A.5 Remove Match on □

To ensure that the application □□ reduces to □ we will implement □ as a recursive function in the target that consumes and ignores its argument. We then need to ensure that subexpressions of the form `match □ with` $Ca_1 \ldots a_n \Rightarrow t$ `end` do not occur, since case analysis on a function will result in a segfault. Thus, we statically replace such case analyses immediately by t applied to n times □. We do however keep *empty* pattern-matchings on □ as they correspond to case analysis on empty types. These are stuck terms that are guaranteed never to occur on the evaluation path to a value, i.e. dead code.

This translation checks if a singleton case analysis or projection is performed on a propositional inductive type (in Prop or SProp) and expands it according to the number of arguments of its single constructor. Similarly to let-binding expansion, this statically performs a reduction that is otherwise performed by evaluation using a specific case-on-prop rule, so it is straightforward to show preservation of evaluation.

A.6 Constructors as Blocks

The syntax of $\lambda_\square$ allows two forms of applying a constructor with name `n` to an argument `a`. First, via explicit application `tApp (tConstruct c []) a`. Secondly, via the fields of the constructor `tConstruct c [a]`. Depending on which switch is set, the evaluation semantics will treat them differently. In this pass, we translate from the first to the second representation.

We first show the adapted rules of the evaluation relation which treats constructors as blocks:

```
Definition atom {wfl : WcbvFlags} Σ t :=
 match t with
 | tBox | tCoFix _ _ | tLambda _ _ | tFix _ _ | tPrim _ ⇒ true
 | tConstruct ind c [] ⇒ negb (@with_constructor_as_block wfl) ∧ isSome (lookup_constructor Σ ind c)
 | _ ⇒ false
 end.
```

```
Inductive eval {wfl : WcbvFlags} : term → term → Set :=
 (** Case *)
 | eval_iota_block ind pars cdecl discr c args brs br res :
  @with_constructor_as_block wfl = true →
  eval discr (tConstruct ind c args) →
  constructor_isprop_pars_decl Σ ind c = Some (false, pars, cdecl) →
  nth_error brs c = Some br →
  #|args| = pars + cdecl.(cstr_nargs) →
  #|skipn pars args| = #|br.1| →
  eval (iota_red pars args br) res →
  eval (tCase (ind, pars) discr brs) res

(** Constructor congruence *)
| eval_construct_block ind c mdecl idecl cdecl args args' :
 @with_constructor_as_block wfl = true →
 lookup_constructor Σ ind c = Some (mdecl, idecl, cdecl) →
 #|args| = cstr_arity mdecl cdecl →
 All2_Set eval args args' →
 eval (tConstruct ind c args) (tConstruct ind c args')
```

Note that the application constructor in $\lambda_{\square}$ is unary, *i.e.*, it applies a function to exactly one argument. However, for the present translation, we need to accumulate *all* arguments to a function. This pattern can easily be implemented using an accumulator argument. We here show how to solve it once and for all in Coq, by using a view and the equations plugin [Sozeau and Mangin 2019]:

```
Section transform_blocks.
 Variable (Σ : GlobalContextMap.t).

 Section Def.
  Import TermSpineView.

  Equations? transform_blocks (t : term) : term
   by wf t (fun x y : EAst.term ⇒ size x < size y) :=
   | e with TermSpineView.view e := {
   | tApp u v napp nnil with construct_viewc u :=
    { | view_construct ind i block_args with GlobalContextMap.lookup_constructor_pars_args Σ ind i := {
       | Some (npars, nargs) ⇒
        let args := map_InP v (fun x H ⇒ transform_blocks x) in
        let '(args, rest) := MCList.chop nargs args in
        EAst.mkApps (EAst.tConstruct ind i args) rest
       | None ⇒
        let args := map_InP v (fun x H ⇒ transform_blocks x) in
        EAst.tConstruct ind i args }
     | view_other _ _ ⇒ mkApps (transform_blocks u) (map_InP v (fun x H ⇒ transform_blocks x)) }

   | tConstruct ind i block_args ⇒ EAst.tConstruct ind i []
   (* ... *).

  Definition switch_constructor_as_block fl : WcbvFlags :=
   EWcbvEval.Build_WcbvFlags fl.(@with_prop_case) fl.(@with_guarded_fix) true.

 End Def.
```

The definition is done by well-founded induction on the size of the term, and we immediately apply a view of the term which decomposes it into a (non-applicative) head and a spine of arguments. We can then check if the head of the term is a constructor and move the application's arguments to the argument field of the `tConstruct` constructor. We rely on η-expandedness of constructors to ensure correctness. The application of the resulting constructor to the `rest` arguments might seem uncanny, as it would mean that constructors can be over-applied. This is only a trick used

```
Section Wcbv.
 Context Σ( : global_declarations).

 Inductive eval (E : environment) : term → value → Set :=
 | eval_var na v :
   lookup E na = Some v →
   eval E (tVar na) v

 | eval_fix mfix idx nms :
   ∀ (Hlen : (idx < #|mfix|)),
   List.∀b (isLambda ∘ dbody) mfix →
   NoDup nms →
   Forall2 (fun d n ⇒ nNamed n = d.(dname)) mfix nms →
   eval E (tFix mfix idx) (vRecClos (map2 (fun n d ⇒ (n, d.(dbody))) nms mfix) idx E)

 | eval_fix_unfold f mfix idx a na na' av fn v E' :
   eval E f (vRecClos mfix idx E') →
   ∀b (fun x ⇒ isLambda (snd x)) mfix →
   NoDup (map fst mfix) →
   nth_error mfix idx = Some (na', tLambda (nNamed na) fn) →
   ~ In na (map fst mfix) →
   eval E a av →
   eval (add na av (add_multiple (List.rev (map fst mfix)) (fix_env mfix E') (E'))) fn v →
   eval E (tApp f a) v
```

Fig. 4. Evaluation semantics for named terms

to facilitate the commutation lemmas of this transformation with substitution, which otherwise would require a typing argument.

A.7 Named Variables and Environment Semantics

We describe a variant of $\lambda_\square$ with named variables, which is based on environments rather than explicit substitution, and consequently uses a dedicated value type. We use a notion of unfolding a dedicated value in a given environment to a term.

We first describe the type of values, which comes with four constructors:

```
Inductive value : Set :=
| vBox
| vConstruct (ind : inductive) (c : ℕ) (args : list (value))
| vClos (na : ident) (b : term) (env : list (ident * value))
| vRecClos (b : list (ident * term)) (idx : ℕ) (env : list (ident * value)).
```

The `vBox` constructor represents the value of the □ construct of $\lambda_\square$. The `constructor` constructor represents the constructor of an inductive `ind` with index `c` with a list of its arguments, which are fully evaluated as well. The `vClos` constructor represents a closure, *i.e.,* the value of an abstraction. It carries the bound name `na` of the closure, the body `b` which is a (non-evaluated) term, and an environment `env` with bindings for the names of `b`. Similarly, the `vRecClos` constructor contains a list of (mutual) names and bodies as well as an index `idx`, indicating which body is represented, and an environment `env` once again binding names in the mutual bodies.

Next, we describe the evaluation relation on named terms in a global environment Σ and named environment `E` in Fig. 4.

The `eval_var` rule states that a variable evaluates to a value `v` according to the environment `E`. The `eval_fix` rule states that a fixpoint `tFix mfix idx` evaluates to a recursive closure in environment `E` which carries the names `nms` and bodies of the fixpoint `mfix` as well as the number `idx` and the environment `E`, where `nms` is the list of names bound in the fixpoint (ruling out anonymous

```
Reserved Notation "Γ ; E ⊩ s ∼ t" (at level 50, E, s, t at next level).

Inductive represents : list ident → environment → term → term → Set :=
| represents_bound_tRel Γ n na E : nth_error Γ n = Some na → Γ ; E ⊩ tVar na ∼ tRel n
| represents_unbound_tRel E na v Γ s : lookup E na = Some v → represents_value v s → Γ ; E ⊩ tVar na ∼ s
% | represents_tLambda Γ E na na' b b' : (na :: Γ) ; E ⊩ b ∼ b' → Γ ; E ⊩ tLambda (nNamed na) b ∼ tLambda na' b'
| represents_tFix Γ E mfix mfix' idx nms :
  List.∀b (isLambda ∘ dbody) mfix →
  NoDup nms →
  All2_Set (fun d n ⇒ d.(dname) = nNamed n) mfix nms →
  All2_Set (fun d d' ⇒ (List.rev nms ++ Γ) ; E ⊩ d.(dbody) ∼ d'.(dbody)) mfix mfix' →
  Γ ; E ⊩ tFix mfix idx ∼ tFix mfix' idx
with represents_value : value → term → Set :=
| represents_value_tBox : represents_value vBox tBox
| represents_value_tClos na E s t na' : [na] ; E ⊩ s ∼ t → represents_value (vClos na s E) (tLambda na' t)
| represents_value_tConstruct vs ts ind c : All2_Set represents_value vs ts → represents_value (vConstruct ind c vs) (
      tConstruct ind c ts)
| represents_value_tFix vfix i E mfix :
  All2_Set (fun v d ⇒ isLambda (snd v) × (List.rev (map fst vfix) ; E ⊩ snd v ∼ d.(dbody))) vfix mfix → represents_value (
      vRecClos vfix i E) (tFix mfix i)
where "Γ ; E ⊩ s ∼ t" := (represents Γ E s t).
```

Fig. 5. Unfolding relation between named terms / values and de Bruijn terms

binders via wildcards). Furthermore, there is a precondition ensuring that that every mutual body is syntactically an abstraction. Lastly, the `eval_fix_unfold` rule states that an application `tApp f a` evaluates to value `v` if `f` evaluates to a recursive closure `vRecClos mfix idx E'` where all bodies in `mfix` are syntactically abstractions, where the names in `mfix` are duplicate-free, the mutual body at position `idx` in `mfix` is an abstraction binding name `na` with body `fn`, and `na` is not occurring in the recursive names of the fixpoint. Furthermore, `a` has to evaluate to a value `av`, and evaluating `fn` yields `v` in the environment `E'` extended with all recursive bindings of `mfix` and `na` bound to `av`.

Next, we describe a representation relation between de Bruijn terms s and named terms t in a context Γ and environment E. Intuitively $\Gamma; E \Vdash s \sim t$ means that given a term t with free variables in Γ, unfolding the term s in the environment E yields t. The relation is mutually inductive with a relation $\Vdash_{\mathrm{v}} v \sim t$ between values v and de Bruijn terms t. We describe the rules for `tRel` and `tFix`, as well as all rules for the value relation in Fig. 5.

The `represents_bound_tRel` rule unfolds a (bound) named variable `tVar na` to a de Bruijn variable `tRel n` by looking up the `n`-th name in Γ. The `represents_unbound-tRel` rule unfolds an (unbound) named variable coqetVar na to a de Bruijn term `s` by looking up the value `v` at name `na` in the environment `E`, and by then unfolding `v` to `s`. The `represents_tFix` rule ensures unfolds `tFix mfix idx` to `tFix mfix' idx` by ensuring that all bodies in `mfix` are syntactically `tLambda`s, ensuring that all bound names in `mfix` are indeed explicit names (identified by the list `nms`) rather than wildcards, and that these names are duplicate-free. Lastly, the bodies have to unfold pointwise, in the context `List.rev nms ++ Γ`. Note that the `List.rev` comes in since the *last* mutual fixpoint bound in `mfix` will have de Bruijn index 0.

We first prove that if a value unfolds to a de Bruijn term `s` and `s` evaluates to `t`, then `v` also unfolds to `t`.

```
Lemma eval_represents_value Σ s t :
  EWcbvEval.eval Σ s t → ∀ v, ⊩ v ∼ s → ⊩ v ∼ t.
```

Next, we prove a theorem relating unfolding and substitution:

```
Lemma represents_subst E s s' t v na :
 ~ In na (map fst E) →
 ⊩ v ~ t →
 [na] ; E ⊩ s ~ s' →
 [] ; add na v E ⊩ s ~ csubst t 0 s'.
```

By induction, we can then lift the theorem about parallel substitution with a list:

```
Lemma represents_substl_rev E s s' ts nms vs Γ :
 (∀ na, In na nms → ~ In na (map fst E)) →
 NoDup nms →
 #|nms| = #|vs| →
 All2 represents_value vs ts →
 (nms ++ Γ) ; E ⊩ s ~ s' →
 Γ ; add_multiple nms vs E ⊩ s ~ substl ts s'.
```

Next, we introduce a well-formedness predicate `wf v` on values:

The predicate relies on a definition of sunny terms, see Fig. 6, which are terms that are not shadowing. For instance, an abstraction `tLambda (nNamed na) M` is not shadowing a list of names Γ if `na` does not occur in Γ and `M` does not shadow the extended context `na :: Γ`. We first prove that evaluation a sunny term yields a well-formed value:

```
Lemma eval_wf Σ E s v :
 Forall (fun d ⇒ match d.2 with ConstantDecl (Build_constant_body (Some d)) ⇒ sunny [] d | _ ⇒ true end) Σ →
 All (fun x ⇒ wf (snd x)) E →
 sunny (map fst E) s →
 eval Σ E s v →
 wf v.
```

Lastly, we introduce an annotation function turning any term into a sunny term:

```
Fixpoint annotate (s : list ident) (u : term) {struct u} : term :=
 match u with
 | tRel n ⇒ match nth_error s n with | Some na ⇒ tVar na | None ⇒ tRel n end
 | tLambda na M ⇒ let na' := match na with
          | nNamed na ⇒ if in_dec (ReflectEq_EqDec _) na s then gen_fresh na s else na
          | nAnon ⇒ gen_fresh "wildcard" s
          end in
          tLambda (nNamed na') (annotate (na' :: s) M)
 (* ... *) end.

Lemma nclosed_represents Σ Γ E s :
 wellformed Σ Γ#|| s → Γ ; E ⊩ annotate Γ s ~ s.

Lemma sunny_annotate Γ s :
 sunny Γ (annotate Γ s).
```

We can then prove the central theorem:

```
Lemma eval_to_eval_named_full Σ( Σ' : global_context) s t :
 wf_glob Σ →
 Forall (fun d ⇒ match d.2 with ConstantDecl (Build_constant_body (Some d)) ⇒ sunny [] d | _ ⇒ true end) Σ' →
 All2 (fun d d' ⇒ d.1 = d'.1 × match d.2 with ConstantDecl (Build_constant_body (Some body)) ⇒
  ∑ body', d'.2 = ConstantDecl (Build_constant_body (Some body')) × [] ; [] ⊩ body' ~ body
 | decl ⇒ d'.2 = decl
 end) Σ Σ' →
 EWcbvEval.eval Σ s t →
 wellformed Σ 0 s →
 ∑ v, ⊩ v ~ t × eval Σ' [] (annotate [] s) v.
```

The actual proof of the theorem is by a generalised induction for any term `u` with Γ ; E ⊩ u ~ s and `sunny (map fst E) u`.

```
Fixpoint sunny Γ (t : term) : 𝔹 :=
 match t with
 | tRel i ⇒ true
 | tLambda (nNamed na) M ⇒ fresh na Γ ∧ sunny (na :: Γ) M
 | tApp u v ⇒ sunny Γ u ∧ sunny Γ v
 (* ... *)
 end.

Inductive wf : value → Type :=
 | wf_vBox : wf vBox
 | wf_vClos na b E : ~ In na (map fst E) → sunny (na :: map fst E) b → All (fun v ⇒ wf (snd v)) E → wf (vClos na b E)
 | wf_vConstruct ind c args : All wf args → wf (vConstruct ind c args)
 | wf_vRecClos vfix idx E :
    NoDup (map fst vfix) →
    (∀ nm, In nm (map fst vfix) → ~ In nm (map fst E)) →
    All (fun t ⇒ sunny (map fst vfix ++ map fst E) (snd t)) vfix →
    All (fun v ⇒ wf (snd v)) E →
    wf (vRecClos vfix idx E).
```

Fig. 6. Non-shadowing terms

A.8 Translation to Malfunction

We now describe the final translation of the version of $\lambda_\square$ with unary fixpoints, constructors as blocks, no case analysis on $\square$, using named variables with an environment semantics and an explicit value type to Malfunction in more details.

Three aspects of the translation functions are interesting:

First, constructors without arguments are solely represented as integers in OCaml, rather than as empty blocks with only a tag.[7] To be able to interface with OCaml programs in the next section we use the same representation, even though not technically necessary for the first-order correctness theorem to hold. Since constructors are fully eta expanded, we can check whether they take arguments or not by case analysis on their field. See the first highlighted block in Fig. 7.

Secondly, $\lambda_\square$ has an explicit case analysis construct, whereas Malfunction only has native `switch` and `proj` constructs. The operational idea is to use `switch` to decide on the branch, and then to replace the variables bound in the branches by the respective projections.

```
match d with
| A ⇒ x
| C a b ⇒ f
end
```

```
switch d with
| 0 ⇒ x
| (tag 0) ⇒ (fun a b ⇒ f) (proj 0 d) (proj 1 d)
end
```

Note that due to the weak call-by-value semantics we cannot substitute the variables directly, because such a translation would only be correct up to observational congruence. The introduced beta-redices might be unnecessary, but we leave it to the OCaml compiler to simplify them. As a result, our correctness statement can be stated again in terms of evaluation semantics only.

[7] https://v2.ocaml.org/manual/intfc.html#ss:c-concrete-datatypes

```
Section Compile.
 Context Σ( : global_declarations).

 Definition Mapply_u t a := match t with Mapply (fn, args) ⇒ Mapply (fn, List.app args [a]) | _ ⇒ Mapply (t, [a]) end.

 Equations? compile (t: term) : Malfunction.t
  by wf t (fun x y : EAst.term ⇒ size x < size y) :=
   | tRel n ⇒ Mstring "tRel"
   | tBox ⇒ Mstring "tBox"
   | tLambda nm bod ⇒ Mlambda ([(BasicAst.string_of_name nm)], compile bod)
   | tLetIn nm dfn bod ⇒ Mlet ([Named ((BasicAst.string_of_name nm), compile dfn)], compile bod)
   | tApp fn arg ⇒
     Mapply_u (compile fn) (compile arg)
     | tConst nm ⇒ Mglobal (Kernames.string_of_kername nm)
  | tConstruct i m args ⇒
    match lookup_constructor_args Σ i with
    | Some num_args ⇒
      match args with
      | [] ⇒ Mnum (numconst_Int (int_of_ℕ (nonblocks_until m num_args)))
      | _ ⇒ Mblock (int_of_ℕ (blocks_until m num_args), map_InP args (fun x H ⇒ compile x))
    | None ⇒ Mstring "error: inductive not found"
    end
   | tCase i mch brs ⇒
    match lookup_constructor_args Σ (fst i) with
    | Some num_args ⇒
      Mcase (num_args, compile mch, map_InP brs (fun br H ⇒ (rev_map (fun nm ⇒ (BasicAst.string_of_name nm)) (fst br),
      compile (snd br))))
   | None ⇒ Mstring "inductive not found"
   end
   | tFix mfix idx ⇒
     let bodies := map_InP mfix (fun d H ⇒ ((BasicAst.string_of_name (d.(dname))), compile d.(dbody))) in
     Mlet ([Recursive bodies], Mvar (fst (nth (idx) bodies ("", Mstring ""))))
  | tProj (Kernames.mkProjection ind _ nargs) bod with lookup_record_projs Σ ind :=
    { | Some args ⇒
        let len := List.length args in
        Mfield (int_of_ℕ (len - 1 - nargs), compile bod)
      | None ⇒ Mstring "Proj" }
  | tCoFix mfix idx ⇒ Mstring "TCofix"
  | tVar na ⇒ Mvar na
  | tEvar _ _ ⇒ Mstring "Evar"
  | tPrim p ⇒ to_primitive p.
  Proof.
   all: try (cbn; lia).
   - subst args. eapply (In_size id size) in H.
    unfold id in H. change size with (fun x ⇒ size x) at 2. cbn [size]. exact H.
   - eapply (In_size snd size) in H. cbn in *.
    lia.
   - eapply (In_size dbody size) in H. cbn in *. lia.
  Qed.

End Compile.
```

Fig. 7. Final compilation phase to Malfunction

```
Definition blocks_until i (num_args : list ℕ) :=
 #| filter (fun x ⇒ match x with 0 ⇒ false | _ ⇒ true end) (firstn i num_args)|.

Definition nonblocks_until i num_args :=
  #| filter (fun x ⇒ match x with 0 ⇒ true | _ ⇒ false end) (firstn i num_args)|.

Definition Mcase : list ℕ * t * list (list Ident.t * t) → t :=
 fun '(num_args, discr, brs) ⇒
  Mswitch (discr, mapi (fun i '(nms, b) ⇒
    (match nth_error num_args i with
    | Some 0 ⇒ [Malfunction.Intrange (int_of_ℕ (nonblocks_until i num_args), int_of_ℕ (nonblocks_until i num_args))]
    | _hasargs ⇒ [Malfunction.Tag (int_of_ℕ (blocks_until i num_args))]
    end,
    Mapply_ (Mlambda_ (nms, b), mapi (fun i _ ⇒ Mfield (int_of_ℕ i, discr)) (nms)))) brs).
```

```
Fixpoint compile_value `{H : Heap} Σ( : EAst.global_declarations) (s : EWcbvEvalNamed.value) : SemanticsSpec.value :=
 match s with
 | vBox ⇒
   RClos (fun _ ⇒ fail "empty", ["reccall"], [RFunc ("_", Malfunction.Mvar "reccall")], 0)
 | vClos na b env ⇒ Func ((fun x ⇒ match lookup (map (fun '(x,v) ⇒ (x, compile_value Σ v)) env) x with Some v ⇒ v | None ⇒ fail "
      notfound" end), na, compile Σ b)
 | vConstruct i m [] ⇒
   match lookup_constructor_args Σ i with
   | Some num_args ⇒ let num_args_until_m := firstn m num_args in
            let index := #| filter (fun x ⇒ match x with 0 ⇒ true | _ ⇒ false end) num_args_until_m| in
            SemanticsSpec.value_Int (Malfunction.Int, BinInt.Z.of_ℕ index)
   | None ⇒ fail "inductive not found"
   end
 | vConstruct i m args ⇒
   match lookup_constructor_args Σ i with
   | Some num_args ⇒ let num_args_until_m := firstn m num_args in
            let index := #| filter (fun x ⇒ match x with 0 ⇒ false | _ ⇒ true end) num_args_until_m| in
            Block (int_of_ℕ index, map (compile_value Σ) args)
   | None ⇒ fail "inductive not found"
   end
 | vRecClos mfix idx env ⇒
    RClos ((fun x ⇒ match lookup (map (fun '(x,v) ⇒ (x, compile_value Σ v)) env) x with Some v ⇒ v | None ⇒ fail "notfound" end),
       (map fst mfix),
       map (fun '(_, b) ⇒
        match b with
        | EAst.tLambda na bd ⇒ RFunc ((BasicAst.string_of_name na), compile Σ bd)
        | _ ⇒ Bad_recursive_value
        end
       ) mfix,
       idx)
 end.
```

Fig. 8. Compilation function for values

Thirdly, to translate Coq's `fix`, we use a recursive `let` in Malfunction, as hightlighted in Fig. 7. We show the compilation function of values in Fig. 8. The translation is straightforward and should not require more explanation.

To get an elegant verification of the correctness of the translation of fixpoints it is crucial that we use recursive closures as values, rather than trying to build a function closure `Func` with a circularity in the body, *e.g.*, through a fixed-point combinator.

The correctness statement reads:

```
Lemma compile_correct `{Hp : Heap} Σ Σ' s t Γ Γ' h :
  (∀ i mb ob, EGlobalEnv.lookup_inductive Σ i = Some (mb, ob) → #|ob.(EAst.ind_ctors)| < Z.to_ℕ Malfunction.Int63.wB /\
   ∀ n b, nth_error ob.(EAst.ind_ctors) n = Some b → b.(EAst.cstr_nargs) < int_to_ℕ PArray.max_length) →
  (∀ na, Malfunction.Ident.Map.find na Γ' = match lookup Γ na with Some v ⇒ compile_value Σ v | _ ⇒ fail "notfound" end) →
  (∀ c decl body v, EGlobalEnv.declared_constant Σ c decl → EAst.cst_body decl = Some body → EWcbvEvalNamed.eval Σ [] body
       v → In ((Kernames.string_of_kername c), compile_value Σ v) Σ') →
  EWcbvEvalNamed.eval Σ Γ s t → SemanticsSpec.eval Σ' Γ' h (compile Σ s) h (compile_value Σ t).
```

The first two lines state preconditions about the global environment: All declared inductives can have at most as many constructors as there are machine integers (in order for tags to still fit into a block), and the arity of constructors cannot be higher than the maximal length of an array (in order for the arguments to fit into a block).

The third line relates the local $\lambda_\square$ environment Γ with the Malfunction environment Γ', whereas the fourth line relates the global $\lambda_\square$ environment Σ with the Malfunction environment Σ'.

The final line then states the correctness: Whenever a $\lambda_\square$ term evaluates, the compiled term evaluates to the value produced by `compile_value`.