JFP **32**, e11, 60 pages, 2022.
doi:10.1017/S0956796822000077

Extracting functional programs from Coq, in Coq

DANIL ANNENKOV
Computer Science, Aarhus University, Aarhus, Denmark
(*e-mail:* `danil.v.annenkov@gmail.com`)

MIKKEL MILO
Computer Science, Aarhus University, Aarhus, Denmark
(*e-mail:* `mikkelmilo@gmail.com`)

JAKOB BOTSCH NIELSEN
Computer Science, Aarhus University, Aarhus, Denmark
(*e-mail:* `jakob.botsch.nielsen@gmail.com`)

BAS SPITTERS
Computer Science, Aarhus University, Aarhus, Denmark
(*e-mail:* `bas@cs.au.dk`)

Abstract

We implement extraction of Coq programs to functional languages based on MetaCoq's certified erasure. We extend the MetaCoq erasure output language with typing information and use it as an intermediate representation, which we call $\lambda_{\square}^{T}$. We complement the extraction functionality with a full pipeline that includes several standard transformations (e.g. eta-expansion and inlining) implemented in a proof-generating manner along with a verified optimisation pass removing unused arguments. We prove the pass correct wrt. a conventional call-by-value operational semantics of functional languages. From the optimised $\lambda_{\square}^{T}$ representation, we obtain code in two functional smart contract languages, Liquidity and CameLIGO, the functional language Elm, and a subset of the multi-paradigm language for systems programming Rust. Rust is currently gaining popularity as a language for smart contracts, and we demonstrate how our extraction can be used to extract smart contract code for the Concordium network. The development is done in the context of the ConCert framework that enables smart contract verification. We contribute with two verified real-world smart contracts (boardroom voting and escrow), which we use, among other examples, to exemplify the applicability of the pipeline. In addition, we develop a verified web application and extract it to fully functional Elm code. In total, this gives us a way to write dependently typed programs in Coq, verify, and then extract them to several target languages while retaining a small trusted computing base of only MetaCoq and the pretty-printers into these languages.

1 Introduction

Proof assistants offer a promising way of delivering the strongest guarantee of correctness. Many software properties can be stated and verified using the currently available tools such as, e.g. Coq, Agda, and Isabelle. In the current work, we focus our attention on the Coq proof assistant based on dependent-type theory (calculus of inductive constructions—CIC). Since the calculus of Coq is also a programming language, it is possible to execute programs directly in the proof assistant. The expressiveness of Coq's type system allows for writing specifications directly in a program type. These specification can be expressed, for example, in the form of pre- and postconditions using *subset types* implemented in Coq using the dependent pair type (Σ-type). However, in order to integrate the formally verified code with existing components, one would like to obtain a program in other programming languages. One way of achieving this is to *extract* the executable code from the formalised development. Various verified developments rely extensively on the extraction feature of proof assistants (Cruz-Filipe and Spitters, 2003; Filliâtre & Letouzey, 2004; Leroy, 2006; Cruz-Filipe & Letouzey, 2006; Klein *et al.*, 2014). However, currently, the standard extraction feature in proof assistants focuses on producing code in conventional functional languages (Haskell, OCaml, Standard ML, Scheme, etc.). Nowadays, there are many new important target languages that are not covered by the standard extraction functionality.

An example of a domain that experiences rapid development and the increased importance of verification is the *smart contract technology*. Smart contracts are programs deployed on top of a blockchain. They often control large amounts of value and cannot be changed after deployment. Unfortunately, many vulnerabilities have been discovered in smart contracts and this has led to huge financial losses (e.g. TheDAO,[1] Parity's multi-signature wallet[2]). Therefore, smart contract verification is crucially important. Functional smart contract languages are becoming increasingly popular, e.g. Simplicity (O'Connor, 2017), Liquidity (Bozman *et al.*, 2018), Plutus (Chapman *et al.*, 2019), Scilla (Sergey *et al.*, 2019) and LIGO.[3] A contract in such a language is a partial function from a message type and a current state to a new state and a list of actions (transfers, calls to other contracts), making smart contracts more amenable for formal verification. Functional smart contract languages, similarly to conventional functional languages, are often based on a well-established theoretical foundation (variants of the Hindley–Milner type system). The expressive type system, immutability, and message-passing execution model allow for ruling out many common errors in comparison with conventional smart contract languages such as Solidity.

For the errors that are not caught by the type checker, a proof assistant, in particular Coq, can be used to ensure correctness. Once properties of contracts are verified, one would like to execute them on blockchains. At this point, the code extraction feature of Coq would be a great asset, but extraction to smart contract languages is not available in Coq.

There are other programming languages of interest in different domains that are not covered by the current Coq extraction. Among these, Elm (Feldman, 2020)—a functional language for web development and Rust (Klabnik & Nichols, 2018)—a multi-paradigm systems programming language, are two examples.

Another issue we face is that the current implementation of Coq extraction is written in OCaml and is not itself verified, potentially breaking the guarantees provided by the formalised development. We address this issue by using an existing formalisation of the

[1] `https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/` (accessed 2021-07-20).

[2] `https://www.parity.io/the-multi-sig-hack-a-postmortem/` (accessed 2021-07-20).

[3] `https://ligolang.org/` (accessed 2021-07-20).

meta-theory of Coq and provide a framework that is implemented Coq itself. Being written in Coq gives us a significant advantage since it makes it possible to apply various techniques to verify the development itself.

The current work extends and improves the results previously published and presented by the same authors at the conference Certified Programs and Proofs (Annenkov *et al.*, 2021) in January 2021. We build on the ConCert framework (Nielsen & Spitters, 2019; Annenkov *et al.*, 2020) for smart contracts verification in Coq and the MetaCoq project (Sozeau *et al.*, 2020). We summarise the contributions as the following, marking with † the contributions that extend the previous work.

- We provide a general framework for extraction from Coq to a typed functional language (Section 5.1). The framework is based on certified erasure (Sozeau *et al.*, 2019) of MetaCoq. The output of MetaCoq's erasure procedure is an AST of an untyped functional programming language $\lambda_\square$. In order to generate code in typed programming languages, we implement an erasure procedure for types and inductive definitions. We add the typing information for all $\lambda_\square$ definitions and implement an annotation mechanism allowing for adding annotations in a modular way—without changing the AST definition. We call the resulting representation $\lambda_\square^T$ and use it as an intermediate representation. Moreover, we implement and prove correct an optimisation procedure that removes unused arguments. The procedure allows us to optimise away some computationally irrelevant bits left after erasure.
- We implement pre-processing passes before the erasure stage (see Section 5.2). After running all the passes, we generate correctness proofs. The passes include
  - η-expansion;
  - expansion of `match` branches†;
  - inlining†.
- We develop in Coq pretty-printers for obtaining extracted code from our intermediate representation to the following target languages.
  - Liquidity—a functional smart contract language for the Dune network (see Section 5.3).
  - CameLIGO—a functional smart contract language from the LIGO family for the Tezos network (see Section 5.3)†.
  - Elm—a general purpose functional language used for web development (see Section 5.4).
  - Rust—a multi-paradigm systems programming languages (see Section 5.5)†.
- We develop an integration infrastructure, required to deploy smart contracts written in Rust on the Concordium blockchain†.
- We provide case studies of smart contracts in ConCert by proving properties of an escrow contract and an anonymous voting contract based on the Open Vote Network protocol (Sections 6 and 7). We apply our extraction functionality to study the applicability of our pipeline to the developed contracts.

Apart from the extensions marked above, we have improved over the previous work in the following points.

- The erasure procedure for types now covers type schemes. We provide the updated procedure along with the discussion in Section 5.1.1

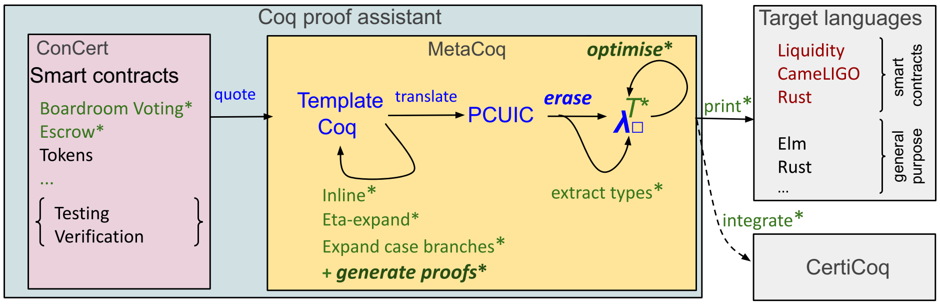

Fig. 1: The pipeline.

- We extract the escrow contract to new target languages and finalise the extraction of the boardroom voting contract, which was not previously extracted. For the Elm extraction, we develop a verified web application that uses dependent types to encode the validity of the data in the application model. We demonstrate how the fully functional web application can be produced from the formalisation.

2 The pipeline

We begin by describing the whole pipeline covering the full process of starting with a program in Coq and ending with extracted code in one of the target languages. This pipeline is shown in Figure 1. The items marked with * (also given in green) are contributions of this work and the items in ***bold cursive*** are verified. The MetaCoq project (Sozeau *et al.*, 2020) provides us with metaprogramming facilities and formalisation of the meta-theory of Coq, including the verified erasure procedure.

We start by developing a program in Gallina that can use rich Coq types in the style of certified programming, see e.g. Chlipala (2013). In the case of smart contracts, we can use the machinery available in ConCert to test and verify the properties of interacting smart contracts. We obtain a Template Coq representation by quoting the program. This representation is close to the actual AST representation in the Coq kernel. We then apply a number of *certifying* transformations to this representation (see Section 5.2). This means that we produce a transformed term along with a proof term, witnessing that the transformed term is equal to the original in the theory of Coq. Currently, we assume that the transformations applied to the Template Coq representations preserve convertibility. Therefore, we can easily certify them by generating simple proofs consisting essentially of the constructor of Coq's equality type `eq_refl`. Although the transformations themselves are not verified, generated proofs give strong guarantees that the behaviour of the term has not been altered. One can configure the pipeline to apply several transformations, in this case, they will be composed together and applied to the Template Coq term. The correctness proofs are generated after all the specified transformations are applied.

The theory of Coq is presented by the predicative calculus of cumulative inductive constructions (PCUIC) (Timany & Sozeau, 2017), which is essentially a cleaned-up version of the kernel representation. The translation from the Template Coq representation to PCUIC is mostly straightforward. Currently, MetaCoq provides the type soundness proof for the translation, but computational soundness (wrt. weak call-by-value evaluation) is not verified. However, the MetaCoq developers plan to close this gap in the near future. Most of the meta-theoretic results formalised by the MetaCoq project use the PCUIC representation (see Section 3 for the details about different MetaCoq representations).

From PCUIC, we obtain a term in an untyped calculus of erased programs $\lambda_\square$ using the verified erasure procedure of MetaCoq. By $\lambda_\square^T$, we denote $\lambda_\square$ enriched with typing information, which we obtain using our erasure procedure for types (see Section 5.1.1). Specifically, we add to the $\lambda_\square$ representation of MetaCoq the following.

- Constants and definitions of inductive types in the global environment store the corresponding "erased" types (`box_type` in Section 5.1.1).
- We explicitly represent *type aliases* (definitions that accept some parameters and return a type) as entries in the extended global environment.
- The nodes of the $\lambda_\square$ AST can be optionally annotated with the corresponding "erased" types (see Section 5.3).

The typing information is required for extracting to typed functional languages. The $\lambda_\square^T$ representation is essentially a core of a pure statically typed functional programming language. Our extensions make it a convenient intermediate representation containing enough information to generate code in various target languages.

The pipeline provides a way of specifying optimisations in a compositional way. These optimisations are applied to the $\lambda_\square^T$ representation. Each optimisation should be accompanied with a proof of computational soundness wrt. the big-step call-by-value evaluation relation for $\lambda_\square$ terms. The format for the computational soundness is fixed and the individual proofs are combined in the top-level statement covering given optimisation steps (see Theorem 2). At the current stage, we provide an optimisation that removes dead arguments of functions and constructors (see Section 5.1.2).

The optimised $\lambda_\square^T$ code is then can be printed using the pretty-printers developed directly in Coq. The target languages include two categories: languages for smart contracts and general-purpose languages. The Rust programming language is featured in both categories. However, the use case of Rust as a smart contract language requires slightly more work for integrating the resulting code with the target blockchain infrastructure (see Section 5.5).

Our trusted computing base (TCB) includes Coq itself, the quote functionality of MetaCoq and the pretty-printing to target languages. While the erasure procedure for types is not verified, it does not affect the soundness of the pipeline (see discussion in Section 5.1).

When extracting large programs, the performance of the pipeline inside Coq might become an issue. In such cases, it is possible to obtain an OCaml implementation of our pipeline using the standard Coq extraction. However, this extends the TCB with the OCaml implementation of extraction and the pre-processing pass, since the proof terms will not be generated and checked in the extracted OCaml code.

Our development is open-source and available in the GitHub repository `https://github.com/AU-COBRA/ConCert/tree/journal-2021`. In the text, we refer to our formalisation using the following link format: `path/to/file.v:lemma_name`. The path starts at the root of the project's repository, and the optional `lemma_name` parameter refers to a definition in the file.

Pretty-printing and formalisation of target languages. The printing procedure itself is a partial function in Coq that might fail (or emit errors in the resulting code) due to unsupported constructs. The unsupported constructs could result from limitations of a target language, or a pretty-printer itself. Similarly to the standard extraction of Coq, the printed code may be untypable. The reason is that the type system of Coq is stronger than the type systems of the target languages. In the standard extraction, it is solved with type coercions, but in most of our target languages, the mechanisms to force the type checker to accept the extracted definitions are missing. We discuss these issues and some solutions in Section 5. Note, that the pipeline gives a guarantee that the $\lambda_{\square}^{T}$ code will evaluate to a correct value. Therefore, if the corresponding pretty-printer is correct, the computational behaviour of the extracted code is sound. We discuss the restrictions for each target language in Sections 5.3–5.5.

As we mentioned earlier, the pretty-printing is part of the TCB. It bridges the gap between $\lambda_{\square}^{T}$ and a target language. This gap varies for various target languages. Elm is very close to $\lambda_{\square}^{T}$. CameLIGO and Liquidity are not too far, but have several restrictions and require mapping $\lambda_{\square}^{T}$ constructs to native types and operations. We also apply several standard extraction techniques when printing to a concrete syntax. For example, we replace absurd cases (unreachable branches) in pattern matching with exceptions or non-terminating functions. Such transformations have certain semantic consequences specific to each target language, which are currently captured only informally by the printing procedures.

From that point of view, the pipeline offers partial correctness guarantees ending with the $\lambda_{\square}^{T}$ representation. To extend the guarantees provided by extraction and close this gap, one needs to consider the semantics of the target languages. That is, can add a translation step from $\lambda_{\square}^{T}$ to the target language syntax and prove the translation correct. Ongoing work at Tezos on formalising the semantics of LIGO languages[4] would allow for connecting our $\lambda_{\square}^{T}$ semantics with the CameLIGO semantics, and eventually get a verified pipeline producing verified Michelson code, directly executed by the Tezos infrastructure.

The gap between $\lambda_{\square}^{T}$ and Rust is larger, and it would be beneficial to provide a translation that would take care of modelling memory allocation, for example. Projects like RustBelt (Jung *et al.*, 2021) and Oxide (Weiss *et al.*, 2019) are aiming to give formal semantics to Rust. However, currently, they do not formalise the Rust surface language.

3 The MetaCoq project

Since MetaCoq (Anand *et al.*, 2018) is integral to our work, we briefly introduce the project structure and explain how different parts of it are relevant to our development.

Template Coq. This subproject adds metaprogramming facilities to Coq. That is, Coq definitions can be *quoted* giving an AST of the original term represented as an inductive data type `term` internally in Coq. The `term` type and related data types in Template Coq are very close to the actual implementation of the Coq kernel written in OCaml, which makes the quote/unquote procedures straightforward to implement. This representation is suitable for

[4] `https://gitlab.com/ligolang/ligo/-/tree/dev/src/coq` (accessed 2022-02-21).

defining various term-level transformations as Coq functions. The results of such transformations can be *unquoted* back to an ordinary Coq definition (provided that the resulting term is well-typed).

The Template Coq metaprogramming facilities are used as the first step in our pipeline. Given a (potentially verified and dependently typed) program in Coq, we can use *quote* to obtain the program's AST that is then transformed, extracted, optimised, and finally pretty-printed to one of the target languages (see Figure 1). The transformations at the level of the Template Coq AST are used as a pre-processing step in the pipeline.

Template Coq features vernacular commands for obtaining the quoted representations, e.g. `MetaCoq Quote Definition ...` . In addition to that, it features the *template monad*, which is similar in spirit to the IO monad and allows for interacting with the Coq environment (quote, unquote, query, and add new definitions). We use the template monad in our pipeline for various transformations whenever such interaction is required. For example, we use it for implementing proof generating transformations (see Section 5.2).

PCUIC. Predicative calculus of cumulative inductive constructions (PCUIC) is a variant of the calculus of inductive constructions (CIC) that serves as the underlying theoretical foundation of Coq. In essence, PCUIC representations is a "cleaned-up" version of the Template Coq AST: it lacks lacks type casts and has the standard binary application, compared to the n-ary application in Template Coq. MetaCoq features translation between the two representations.

Various meta-theoretic results about PCUIC has been formalised in MetaCoq, including the verified type checker (Sozeau *et al.*, 2019). In our development, we use the certified programming approach that relies on the results related to reduction and typing.

Verified erasure. One important part of the MetaCoq project that we build on is the verified erasure procedure. The erasure procedure takes a PCUIC term as input and produces a term in $\lambda_\square$. The meta-theory of PCUIC developed as part of MetaCoq is used extensively in erasure implementation and formalisation of the correctness results. The erasure procedure is quite subtle and its formalisation is a substantial step towards the fully verified extraction pipeline. We discuss the role and the details of MetaCoq's verified erasure in Section 5.1.

4 The ConCert framework

The present work builds on and extends the ConCert smart contract certification framework presented by the three authors of the present work at the conference Certified Programs and Proofs in January 2020 (Annenkov *et al.*, 2020). In this section, we briefly describe relevant parts of ConCert along with the extensions developed in Annenkov *et al.* (2021) and in the present work.

Execution Layer. The execution layer provides a model that allows for reasoning on contract execution traces which makes it possible to state and prove temporal properties of interacting smart contracts. In the functional smart contract model, the contracts consist of two functions.

```
init : Chain→ ContractCallContext→ Setup→ option State
```

The `init` function is called after the contract is deployed on the blockchain. The first parameter of type `Chain` represents the blockchain from a contract's point of view. The `ContractCallContext` parameter provides data about the current call, e.g. caller address, the amount sent to the contract, etc. `Setup` is a user defined type that supplies custom parameters to the initialisation function.

```
receive : Chain→ ContractCallContext→ State→ option Msg
          → option (State * list ActionBody)
```

The `receive` function represents the main functionality of the contract that is executed for each call to the contract. `Chain` and `ContractCallContext` are the same as for the `init` function. `State` represents the current state of the contract and `Msg` is a user-defined type of messages that contract accepts. The result of the successful executions is a new state and a list of *actions* represented by `ActionBody`. The actions can be transfers, calls to other contracts (including the contract itself), and contract deployment actions.

However, reasoning about the contract functions in isolation is not sufficient. A call to `receive` potentially emits more calls to other contracts or to itself. To capture the whole execution process, we define the type of executions traces `ChainTrace` as the reflexive-transitive closure of the proof-relevant relation `ChainStep : ChainState → ChainState → Type`. `ChainStep` captures how the blockchain state evolves once new blocks as added and contract calls are executed.

Extraction Layer. Annenkov *et al.* (2020) presented a verified *embedding* of smart contracts to Coq. This work shows how it is possible to verify a contract as a Coq function and then *extract* it into a program in a functional smart contract language. This layer represents an interface between the general extraction machinery we have developed and the use case of smart contracts. In the case of smart contract languages, it is necessary to provide functionality for integrating the extracted smart contracts with the target blockchain infrastructure. In practice, it means that we should be able to map the abstractions of the execution layer (contract's view of the blockchain, call context data) on corresponding components in the target blockchain.

Currently, all extraction functionality we have developed, regardless of the relation to smart contracts, is implemented in the extraction layer of ConCert. In the future, we plan to separate the general extraction component from the blockchain-specific functionality.

5 Extraction

The Coq proof assistant comes with a dependently typed programming language Gallina that allows due to the language's rich type system to write programs together with their specifications in the style of *certified programming* (see e.g. Chlipala, 2013). Coq features a special universe of types for writing program specifications, the universe of propositions `Prop`. For example, the type `{n : nat | 0 < n }` belongs to so-called *subset types*, which are essentially a special case of a dependent pair type (Σ-type). In this example, `0 < n` is a proposition, i.e. it belongs to the universe `Prop`. Subset types allow for encoding many useful invariants when writing programs in Gallina. An inhabitant of `{n : nat | 0 < n }` is a pair with the first component being a natural number and the second component—a *proof* that the number is strictly greater than zero. In the theory of Coq, subset types are represented as an inductive type with one constructor:

```
Inductive sig (A : Type) (P : A→ Prop) : Type :=
  exist : forall x : A, P x→ {x : A | P x}
```

where `{x : A | P x}` is a notation for `sig A P`.

The invariant represented by a second component can be used to ensure, for example, that division by zero never happens since we require that arguments can only be strictly positive numbers. The proofs of specifications are only used to build other proofs and do not affect the computational behaviour of the program, apart from some exceptions called the empty and singleton elimination principle.[5] The `Prop` universe marks such computationally irrelevant bits. Moreover, types appearing in terms are also computationally irrelevant. For example, in System F this is justified by parametric polymorphism. This idea is used in the Coq proof assistant to *extract* the executable content of Gallina terms into OCaml, Haskell, and Scheme. The extraction functionality thus enables proving properties of functional programs in Coq and then automatically producing code in one of the supported languages. The extracted code can be integrated with existing developments or used as a stand-alone program.

The first extraction using `Prop` as a marker for computationally irrelevant parts of programs was introduced by Paulin-Mohring (1989) in the context of the calculus of construction (CoC), which earlier versions of Coq were based on. This first extraction targeted System F_ω, which can be seen as a subset of CoC, allowing one to get the extracted term *internally* in CoC. The current Coq extraction mechanism is based on the theoretical framework from a PhD thesis by Letouzey (2004). Letouzey extended the previous work of Paulin-Mohring (1989) and adapted it to the full calculus of inductive constructions. The target language of the current extraction is untyped, allowing to accommodate more features from the expressive type system of Coq. However, the untyped representation has a drawback; the typing information is still required when extracting to statically typed programming languages. To this end, Letouzey considers practical issues for implementing an efficient extraction procedure, including recovering the types in typed target languages, using type coercions (`Obj.magic`) when required, and various optimisations. The crucial part of the extraction process is the *erasure* procedure that utilises the typing information to prune irrelevant parts. That is, types and propositions in terms are replaced with $\square$ (a box). Formally, it is expressed as a translation from CIC (Calculus of Inductive Constructions) to $\lambda_\square$ (an untyped version of CIC with an additional constant $\square$). The translation is quite subtle and is discussed in detail in by Letouzey (2004). Letouzey also provides two (pen-and-paper) proofs that the translation is computationally sound: one proof is syntactic and uses the operational semantics and the other proof is based on the realisability semantics. Computational soundness means that the original programs and the erased programs compute the same (in a suitable sense) value.

[5] The principle includes important use cases: empty types like `False` and definitions by well-founded recursion with `Acc`.

Having this in mind, we have identified two essential points:

- The target languages supported by the standard Coq extraction do not include many new target languages, that represent important use cases (smart contracts, web programming).
- Since the extraction implementation becomes part of a TCB, one would like to mechanically verify the extraction procedure in Coq itself and the current Coq extraction is not verified.

Therefore, it is important to build a verified extraction pipeline in Coq itself that also allows for defining pretty-printers for new target languages.

Until recently, the proof of correctness of one of the essential ingredients, the erasure procedure, was only done on paper. However, the MetaCoq project made an important step towards verified extraction by formalising the computational soundness of erasure (Sozeau *et al*., 2019, Section 4). The MetaCoq's verified erasure is defined for predicative calculus of cumulative inductive constructions (PCUIC) a variant of CIC that closely corresponds to the meta-theory of Coq. See Section 3 for a brief description of the project's structure and Sozeau *et al*. (2019, Section 2) for the detailed exposition of the calculus. The result of the erasure is a $\lambda_\square$ term, that is, a term in an untyped calculus. On the other hand, integration with typed functional languages requires recovering the types from the untyped output of the erasure procedure. Letouzey (2004) solves this problem by designing an erasure procedure for types and then using a modified type inference algorithm, based on the algorithm $\mathscr{M}$ by Lee & Yi (1998), to recover types and check them against the type produced by extraction. Because the type system of Coq is more powerful than type systems of the target languages (e.g. Haskell or OCaml), not all the terms produced by extraction will be typable. In this case, the modified type inference algorithm inserts type coercions forcing the term to be well-typed. If we start with a Coq term the type of which is outside the OCaml type system (even without using dependent types), the extraction might have to resort to `Obj.magic` in order to make the definition well-typed. For example, the code snippet below

```
Definition rank2 : forall (A : Type), A→ (forall A : Type, A→ A)→ A
  := fun A a f⇒ f _ a.
Extraction rank2.
```

gives the following output on extraction to OCaml:

```
(** val rank2 : 'a1→ (__→ __→ __)→ 'a1 **)
let rank2 a f = Obj.magic f __ a
```

These coercions are "safe" in the sense that they do not change the computational properties of the term, they merely allow to pass the type checking.

5.1 Our extraction

The standard Coq extraction targets conventional general-purpose functional programming languages. Recently, there has been a significant increase in the number of languages that are inspired by these, but due to the narrower application focus are different in various subtle details. We have considered the area of smart contract languages (Liquidity and CameLIGO), web programming (Elm) and general-purpose languages with a functional subset (Rust). They often pose more challenges than the conventional targets for extraction. We have identified the following issues.

1. Most of the smart contract languages[6] and Elm do not offer a possibility to insert type coercions forcing the type checking to succeed.
2. The operational semantics of $\lambda_\square$ has the following rule (Sozeau *et al.*, 2019, Section 4.1): if $\Sigma \vdash t_1 \triangleright \square$ and $\Sigma \vdash t_2 \triangleright v$ then $\Sigma \vdash (t_1\ \ t_2) \triangleright \square$, where $- \vdash - \triangleright -$ is a big-step evaluation relation for $\lambda_\square$, t_1 and t_2 are $\lambda_\square$ terms, and v is a $\lambda_\square$ value. This rule can be implemented in OCaml using the unsafe features, which are, again, not available in most of our target languages. In lazy languages, this situation never occurs (Letouzey, 2004, Section 2.6.3), but most of the languages we consider follow the eager evaluation strategy.
3. Data types and recursive functions are often restricted. E.g. Liquidity, CameLIGO (and other LIGO languages) do not allow for defining recursive data types (like lists and trees) and limits recursive definitions to tail recursion on a single argument. Instead, these languages offer built-in lists and finite maps (dictionaries).
4. Rust has a fully-featured functional subset, but being a language for systems programming, does not have a built-in garbage collector.

We can make design choices, concerning the point above, in such a way that the soundness of the extraction will not be affected, given that terms evaluate in the same way before and after extraction. In the worst case, the extracted term will be rejected by the type checker of the target language. However, some care is needed: the pretty-printing step still presents a gap in verification. A more careful treatment of the semantics of target languages would allow extending the guarantees. Some subtleties come from, e.g. handling of absurd pattern-matching cases and its interaction with optimisations. As for now, we apply the approach inspired by the standard extraction in such situations.

Let us consider in detail what the restrictions outlined above mean for extraction. The first restriction means that certain types will not be extractable. Therefore, our goal is to identify a practical subset of extractable Coq types and give the user access to transformations helping to produce well-typed programs. The second restriction is hard to overcome, but fortunately, this situation should not often occur on the fragment we want to work. Moreover, as we noticed before, terms that might give an application of a box to some other term will be ill-typed and thus, rejected by the type checker of the target language. The third restriction can be addressed by mapping Coq's data types (lists, finite maps) to the corresponding primitives in a target language. The fourth restriction applies only to Rust and means that we have to provide a possibility to "plug-in" a memory management implementation. Luckily, Rust libraries contain various implementations one can choose from.

At the moment, we consider the formalisation of typing in target languages out of scope for this project. Even though the extraction of types is not verified, it does not compromise run-time safety: if extracted types are incorrect, the target language's type checker will

[6] At least, Simplicity, Liquidity, CameLIGO (and other LIGO languages), Love `https://dune.network/docs/dune-node-next/love-doc/reference/love.html` (accessed 2021-08-05), Scilla and Sophia `https://aeternity-sophia.readthedocs.io/` (accessed 2021-08-05).

reject the extracted program. If we followed the work by Letouzey (2004), which the current Coq extraction is based on, giving guarantees about typing would require formalising of target languages type systems, including a type inference algorithm (possibly algorithm $\mathscr{M}$). The type systems of many languages we consider are not precisely specified and are largely in flux. Moreover, for the target languages without unsafe coercions, some of the programs will be untypable in any case. On the other hand, for more mature languages (e.g. Elm) one can imagine connecting our formalisation of extraction with a language formalisation, proving the correctness statement for both the run-time behaviour and the typability of extracted terms.

We extend the work on verified erasure (Sozeau *et al.*, 2019) and develop an approach that uses a minimal amount of unverified code that can affect the soundness of the verified erasure. Our approach adds an erasure procedure for types, verified optimisations of the extracted code and pretty-printers for several target languages. The main observation is that the intermediate representation $\lambda_\square^T$ corresponds to the core of a generic functional language. Therefore, our pipeline can be used to target various functional languages with transformations and optimisations applied generically to the intermediate representation.

Before introducing our approach, let us give some examples of how the verified erasure works and motivate the optimisations we propose.

```
Definition sum_nat (xs : list nat) : nat := fold_right plus 0 xs.
```

produces the following $\lambda_\square$ code:

```
fun xs ⇒ Coq.Lists.List.fold_right □ □ Coq.Init.Nat.add O xs
```

where the □ symbol corresponds to computationally irrelevant parts. The first two arguments to the erased versions of `fold_right` are boxes, since `fold_right` in Coq has two implicit type arguments. They become visible if we switch on printing of implicit arguments:

```
Set Printing Implicit.
Print sum_nat.
(* sum_nat = fun xs : list nat ⇒ @fold_right nat nat Init.Nat.add 0 xs
     : list nat → nat *)
```

In this situation we have at least two choices: remove the boxes by some optimisation procedure, or leave the boxes and extract `fold_right` in such a way that the first two arguments belong to some dummy data type.[7] The latter choice cannot be made for some smart contract languages due to restrictions on recursion (`fold_right` is not tail-recursive), therefore, we have to remap `fold_right` and other functions on lists to the corresponding primitive functions. Generally, removing such dummy arguments in the extracted code is beneficial for other reasons: the code size is smaller and some redundant reductions can be avoided.

[7] There are two rules in the semantics of $\lambda_\square$ that do not quite fit into the evaluation model of the languages we consider: pattern-matching on a box argument and having a box applied to some argument. The pattern-matching on a box case is addressed in the last version of MetaCoq and we include this optimisation in our pipeline. The applied box case requires implementing □ as an argument consuming function, which is impossible in several of our target languages due to the absence of unsafe features. Therefore, we choose to implement □ as the `unit` type, potentially resulting in ill-typed programs after extraction. However, such cases mostly occur due to the use of the subtyping rule `Prop` ≤ `Type`. The examples we considered do not make use of this feature.

In the following example,

```
Definition square (xs : list nat) : list nat := map (fun x ⇒ x * x) xs.
```

the `square` function erases to

```
fun xs ⇒ Coq.Lists.List.map □ □ (fun x ⇒ Coq.Init.Nat.mul x x) xs
```

The corresponding language primitive would be a function with the following signature: `TargetLang.map: ('a → 'b) → 'a list → 'b list`. Clearly, there are two extra boxes in the extracted code that prevent us from directly replacing `Coq.Lists.List.map` with `TargetLang.map`. Instead, we would like to have the following:

```
  fun xs ⇒ Coq.Lists.List.map (fun x ⇒ Coq.Init.Nat.mul x x) xs
```

In this case, we can provide a translation table to the pretty-printing procedure mapping `Coq.Lists.List.map` to `TargetLang.map`. Alternatively, if one does not want to remove boxes, it is possible to implement a more sophisticated remapping procedure. It could replace `Coq.Lists.List.map □ □` with `TargetLang.map`, but it would require finding all constants applied to the right number of arguments (or η-expand constants) and only then replace them. Remapping inductive types in the same style would involve more complications: constructors of polymorphic inductive types will have an extra argument of a dummy type. This would require more complicated pretty-printing of pattern-matching in addition to the similar problem with extra arguments on the application sites.

By implementing the optimisation procedure we achieve several goals: make the code size smaller, remove redundant computations, and make the remapping easier. Removing the redundant computations is beneficial for smart contract languages since it decreases the computation cost in terms of *gas*. Users typically pay for calling smart contracts and the price is determined by the gas consumption. That is, gas serves as a measure of computational resources required for executing a contract. Smaller code size is also an advantage from the point of view of gas consumption. For some blockchains, the overall cost of a contract call depends on its size.

It is important to separate these two aspects of extraction: erasure (given by the translation CIC[8] $\longrightarrow \lambda_\square$) and optimisation of $\lambda_\square$ terms to remove unnecessary arguments. The optimisations we propose remove some redundant reductions, make the output more readable and facilitate the remapping to the target language's primitives. Our implementation strategy of extraction is the following: (i) take a term and erase it and its dependencies recursively to get an environment; (ii) analyse the environment to find optimisable types and terms; (iii) optimise the environment in a consistent way (e.g. in our $\lambda_\square^T$, the types must be adjusted accordingly); (iv) pretty-print the result in the target language syntax according to the translation table containing remapped constants and inductive types.

The mechanism of remapping constants and inductive types is similar to the Coq functionality `Extract Inlined Constant` and `Extract Inductive`. Since we run the extraction pipeline inside Coq, the we use ordinary Coq definitions to build a translation table that we pass to the pipeline. For example, the typical translation table would look as the following.

```
Definition translation_table : list (kername * string) :=
  [ remap <%% Z %%> "int";
    ...
```

[8] Note that by CIC terms we mean in this section a particular version of it formalised in MetaCoq—predicative calculus of cumulative inductive constructions (PCUIC).

```
remap <%% @List.map %%> "TargetLang.map";
... ].
```

We use `remap <%% coq_def %%> "target_def"` to produce an entry in the translation table, where the `<%% coq_def %%>` notation uses MetaCoq to resolve the full name of the given definition. Note that the translation table is an association list with `kername` as keys. The `kername` type is provided by MetaCoq and represents fully qualified names of Coq definitions.

5.1.1 *Erasure for types*

Let us discuss our first extension to the verified erasure presented in Sozeau *et al.* (2019), namely an *erasure procedure for types*. It is a crucial part for extracting to a *typed* target language. Currently, the verified erasure of MetaCoq provides only a term erasure procedure which will erase any type in a term to a box. For example, a function using the dependent pair type (Σ-type) might have a signature involving `sig nat (fun n => n > 10)`, i.e. representing numbers that are larger than 10. Applying MetaCoq's *term* erasure will erase this in its entirety to a box, while we are interested in a procedure that instead erases only the type scheme in the second argument: we expect type erasure to produce `sig nat` $\square$, where the square now represents an irrelevant type.

While our target languages have type systems that are Hindley-Milner based (and therefore, can recover types of functions), we still need an erasure procedure for types to be able to extract inductive types. Moreover, our target languages support various extensions, and their compilers may not always succeed to infer types. For example, Liquidity has overloading of some primitive operations, e.g. arithmetic operations for primitive numeric types. Such overloading introduces ambiguities that cannot be resolved by the type checker without type annotations. CameLIGO requires writing even more types explicitly. Thus, the erasure procedure for types is also necessary to produce such type annotations. The implementation of this procedure is inspired by Letouzey (2004).

We have chosen a semi-formal presentation in order to guide the reader through the actual implementation while avoiding clutter from the technicalities of Coq. We use concrete Coq syntax to represent the types of CIC. We do not provide syntax and semantics of CIC, for more information we refer the reader to Sozeau *et al.* (2019, Section 2). The types of $\lambda_{\square}^{T}$ are represented by the grammar below.

$$\sigma,\ \tau : \mathtt{box\_type} ::= \bar{i} \mid \mathtt{I} \mid \mathtt{C} \mid \sigma\ \ \tau \mid \sigma \longrightarrow \tau \mid \square \mid \mathbb{T}$$

Here $\bar{i}$ represents levels of type variables, `I` and `C` range over names of inductive types and constants respectively. Essentially, `box_type` represents types of an OCaml-like functional language extended with constructors $\square$ ("logical" types) and $\mathbb{T}$ (types that are not representable in the target language). In some cases both $\square$ and $\mathbb{T}$ can later be removed from the extracted code by optimisations, although $\mathbb{T}$ might require type coercions in the target language. Note also that the types do not have binders, since they represent prenex-polymorphic types. The levels of type variables are indices into the list of type variable names starting from the head of the list. E.g. the type `'a` $\to$ `'b` is represented as $\bar{0} \longrightarrow \bar{1}$ for the context $[a; b]$. Additionally, we use colours to distinguish between the CIC terms and the target erased types.

```
$\mathcal{E}^T$ : Ctx → ECtx → term → option $\mathbb{N}$
    → list name × box_type
$\mathcal{E}^T\ \Gamma\ \Gamma_e\ t\ v_n$ := let $t'$ := red$_{\beta\iota\zeta}$ $\Gamma\ t$ in
  let $flag$ := flag_of_type $\Gamma\ t'$ in
  if (is_logical $flag$) then ([], □) else
  match $t'$ with
  | $\bar{i}$ ⇒ Ok([], $\mathcal{E}^T_{var}\ \Gamma_e\ i$)
  | forall $a : A, B$ ⇒
    let $flag$ := flag_of_type $\Gamma\ A$ in
    if (is_logical $flag$) then
      let $(vs_\tau, \tau)$ := $\mathcal{E}^T\ (A :: \Gamma)$ (Other :: $\Gamma_e$) $B\ v_n$ in
      $(vs_\tau, \square \longrightarrow \tau)$
    else if ¬(is_arity $flag$) then
      let $(\_, \sigma)$ := $\mathcal{E}^T\ \Gamma\ \Gamma_e\ A\ v_n$ in
      let $(vs_\tau, \tau)$ := $\mathcal{E}^T\ (A :: \Gamma)$ (Other :: $\Gamma_e$) $B\ v_n$ in
      $(vs_\tau, \sigma \longrightarrow \tau)$
    else let $var$ := index_to_type_var $v_n$ in
      let $(vs_\tau, \tau)$ :=
        $\mathcal{E}^T\ (A :: \Gamma)\ (var :: \Gamma_e)\ B$ (inc_var $v_n$) in
      let $vs$ :=
        if (is_none $v_n$) then $vs_\tau$ else $a :: vs_\tau$ in
      $(vs, \square \longrightarrow \tau)$
  | $(u\ \ v)$ ⇒ let $(hd, args)$ := decompose_app $(u\ \ v)$ in
      let $\sigma$ := $\mathcal{E}^T_{head}\ \Gamma_e\ hd$ in
      ([], $\mathcal{E}^T_{app}\ \Gamma_e\ args\ \sigma$ )
  | Type ⇒ ([], □)   | C ⇒ ([], C)   | I ⇒ ([], I)   | _ ⇒ $\mathbb{T}$
  end
```

```
$\mathcal{E}^T_{app}$ : ECtx → list term
    → box_type → box_type
$\mathcal{E}^T_{app}\ \Gamma_e\ args\ \sigma$ :=
  match $args$ with
  | [] ⇒ $\sigma$
  | $a :: args'$ ⇒
      let $A$ := infer $a$ in
      let $flag$ := flag_of_type $\Gamma\ A$ in
      let $\tau$ :=
        if (is_logical $flag$) then □
        else if (is_sort $flag$) then
          snd ($\mathcal{E}^T\ \Gamma\ \Gamma_e\ a$ None )
        else $\mathbb{T}$ in $\mathcal{E}^T_{app}\ \Gamma_e\ args'\ (\sigma\ \tau)$
  end

$\mathcal{E}^T_{head}$ : ECtx → term → box_type
$\mathcal{E}^T_{head}\ \Gamma_e\ hd$ := match $hd$ with
      | $\bar{i}$ ⇒ $\mathcal{E}^T_{var}\ \ \Gamma_e\ i$
      | $C$ ⇒ $C$   | I ⇒ I   | _ ⇒ $\mathbb{T}$
    end

$\mathcal{E}^T_{var}$ : ECtx → $\mathbb{N}$ → box_type
$\mathcal{E}^T_{var}\ \Gamma_e\ i$ :=
  match $\Gamma_e(i)$ with
  | TV $j$ ⇒ $\bar{j}$   | Other ⇒ $\mathbb{T}$   | Ind I ⇒ I
  end
```

Auxiliary functions and abbreviations:

red$_{\beta\iota\zeta}$: Ctx → term → term	(reduce a term to $\beta\iota\zeta$-whnf)
flag_of_type : Ctx → term → type_flag	(analyse how a type should be erased)
is_logical, is_sort, is_arity : type_flag → bool	(projections from type_flag)
infer : term → term	(infer term's type)
decompose_app : term → term × (list term)	(split application into head and args)

```
index_to_type_var $i$ := match $i$ with Some $i$ ⇒ TV $i$ | None ⇒ Other end
        is_none $v$ := match $v$ with Some _ ⇒ true | None ⇒ false end
```

Fig. 2: Erasure from CIC types to box_type.

Definition 1 (Erasure for types).
extraction/theories/Erasure.v:erase_type_aux
The erasure procedure for types is given by functions $\mathcal{E}^T$, $\mathcal{E}^T_{app}$ and $\mathcal{E}^T_{head}$ in Figure 2.

The $\mathcal{E}^T$ function takes four parameters. The first is a context Ctx represented as a list of assumptions. The second is an erasure context ECtx represented as a sized list (vector) that follows the structure of Ctx; it contains either a translated type variable TV, information about an inductive type Ind, or a marker Other for items in Ctx that do not fit into the previous categories. The last two parameters represent terms of CIC corresponding to types and an index of the next type variable. The next type variable index is wrapped in the `option` type, and becomes `None` if no more type variables should be produced. The function $\mathcal{E}^T$ returns a tuple consisting of a list of type variables and a `box_type`.

First, we describe the most important auxiliary functions outlined on the Figure 2. An important device used to determine erasable types (the ones we turn into the special target types $\square$ and $\mathbb{T}$) is the function `flag_of_type` : `Ctx` $\to$ `term` $\to$ `type_flag`, where the return type `type_flag` is defined as a record with two fields: `is_logical` and `is_arity`. The `is_logical` field carries a boolean, while `is_arity` carries a proof or a disproof of convertibility to an arity. That is, whether a term is an arity up to reductions. For the purposes of the presentation in the paper, we treat `is_arity` as a boolean value, while in the implementation we use the proofs carried by `is_arity` to solve proof obligations for the definition of $\mathcal{E}^T$.

A type is an *arity* if it is a (possibly nullary) product into a sort: $\forall \vec{a} : \vec{\mathtt{A}}, \mathtt{s}$ for $\mathtt{s} = \mathtt{Type} \mid \mathtt{Prop}$ and $\vec{a} : \vec{\mathtt{A}}$ a vector of (possibly dependent) binders and types. Inhabitants of arities are *type schemes*.

The predicate `is_sort` is not a field of `type_flag`, but it uses the data of `is_arity` field to tell whether a given type is a *sort*, i.e. `Prop` or `Type`. Sorts are always arities; we use `is_sort` to turn a proof of converitibility to an arity into a proof of convertivility to a sort (or return `None` if it is not the case). Finally, a type is *logical* when it is a proposition (i.e. inhabitants are proofs) or when it is an arity into `Prop`: $\forall \vec{a} : \vec{\mathtt{A}}, \mathtt{Prop}$ (i.e. inhabitants are propositional type schemes). As concrete examples, `Type` is an arity and a sort, but not logical. `Type` $\to$ `Prop` is logical, an arity, but not a sort. `forall` A : `Type,` $A \to A$ is neither of the three.

We use the reduction function $\mathtt{red}_{\beta\iota\zeta}$ in order to reduce the term to $\beta\iota\zeta$ weak head normal form. Here, β is reduction of applied λ-abstractions, ι is reduction of `match` on constructors, and ζ is reduction of the `let` construct. The `infer` function is used to recover typing information for subterms. We also make use of the destructuring `let` $(a, b) := \ldots$ for tuples and projections `fst`, `snd`.

The erasure procedure proceeds as follows. First, it reduces the *CIC* term to $\beta\iota\zeta$ weak head normal form. Then, it uses `flag_of_type` to determine how to erase the reduced type. If it is a logical type, it immediately returns $\square$, if not, it performs case analysis on the result. The most interesting cases are dependent function types and applications. For function types, if the domain type is logical, it produces $\square$ for the resulting domain type and erases the codomain recursively. If the domain is a "normal" type (not logical and not an arity), it erases recursively both the domain and the codomain. Otherwise, the type is an arity and we add a new type variable into Γ_e for the recursive call and append a variable name to the returned list. For an application, the procedure decomposes it into a head term and a (possibly empty) list of arguments using `decompose_app`. Then, it uses $\mathcal{E}^T_{head}$ to erase the head and $\mathcal{E}^T_{app}$ to process all the arguments. Note that in the Coq implementation, when we erase an application $(u\ \ v)$, we drop the arguments, if the head of the application is not an inductive or a constant. Other applied `box_type` construtors would correspond to a not well-formed application. In the cases not covered by the case analysis, we emit $\mathbb{T}$.

For example, the type of `map : forall A B : Type, (A → B) → list A → list B` erases to the following.

$$([A; B], (\overline{0} \longrightarrow \overline{1}) \longrightarrow \texttt{list}\ \overline{0} \longrightarrow \texttt{list}\ \overline{1})$$

One of the advantages of implementing the extraction pipeline fully in Coq is that we can use certified programming techniques along with the verified meta-theory provided by MetaCoq. The actual type signatures of the functions in Figure 2 restrict the input to well-typed CIC terms. In the implementation, we are required to show that the reduction machinery we use does not break the well-typedness of terms. For that purpose, we use two results: the reduction function is sound with respect to the relational specification, and the subject reduction lemma, that is, reduction preserves typing.

The termination argument is given by a well-founded relation since erasure starts with $\beta\iota\zeta$-reduction using the $\texttt{red}_{\beta\iota\zeta}$ function and then later recurses on subterms of the reduced term. This reduction function is defined in MetaCoq also by using well-founded recursion. Due to the use of well-founded recursion, we write $\mathscr{E}^T$ as a single function in our formalisation by inlining the definitions of $\mathscr{E}^T_{app}$ and $\mathscr{E}^T_{head}$; this makes the well-foundedness argument easier. We extensively use the `Equations` Coq plugin (Sozeau & Mangin, 2019) in our development to help manage the proof obligations related to well-typed terms and recursion.

The difference with our previous erasure procedure for types given in Annenkov *et al.* (2021) is twofold. First, we make the procedure total. That means that it does not fail in the cases when it hits a non-prenex type, instead, it tries to do its best or emits $\mathbb{T}$ if no further options are possible. That is, for `rank2` from Section 5 we get the following type.

$$([A], \overline{0} \to (\square \to \mathbb{T} \to \mathbb{T}) \to \overline{0})$$

Clearly, the body of `rank2 a f = f □ a` cannot be well-typed, since the type of `a` is not $\mathbb{T}$. However, in some situations, it is better to let the extraction produce some result that could be optimised later. For example, if we never used the `f` argument, it could be removed by optimisations leaving us with the definition whose that does not mention $\mathbb{T}$.

In particular, we have improved the handling of arities that makes it possible to extract programs defined in terms of elimination principles. For example one can define `map` in the following way: `list_rect (fun x ⇒ list B) [] (fun x _ rec ⇒ f x :: rec) xs`. Where `list_rect` is the dependent elimination principle for lists.

```
list_rect : forall (A : Type) (P : list A→ Type),
  P []→
  (forall (a : A) (l : list A), P l→ P (a :: l))→
  forall l : list A, P l
```

Clearly, the type of `list_rect` is too expressive for the target languages we consider. However, it is still possible to extract a well-typed term for the definition of `map` above. The extracted type of `list_rect` looks as follows.

$$([a; p], \overline{1} \longrightarrow (\overline{0} \longrightarrow \texttt{list}\ \overline{0} \longrightarrow \overline{1} \longrightarrow \overline{1}) \longrightarrow \texttt{list}\ \overline{0} \longrightarrow \overline{1})$$

Second, we have introduced an erasure procedure for type schemes. The procedure allows us to handle type aliases, that is, Coq definitions that when applied to some arguments return a type. Type aliases are used quite extensively in the standard library. For example, the standard finite maps `FMaps` contain definitions like `Definition PositiveMap.t : Type → Type := PositiveMap.tree`. In η-expanded form it is a function that take a type and returns a type: `fun T ⇒ PositiveMap.tree T`. Without this extension, we would not be able to extract programs that use such definitions.

Definition 2 (Erasure for type schemes).
`extraction/theories/Erasure.v:erase_type_scheme`
The erasure procedure for type schemes is given by two functions $\mathscr{E}^{TS}$ and $\mathscr{E}^{TS}_{\eta}$ in Figure 3.

The signatures of $\mathscr{E}^{TS}$ and $\mathscr{E}^{TS}_{\eta}$ are similar to $\mathscr{E}^{T}$ but we also add a new context `ACtx` representing the type of a type scheme, which we call arity. So, for an arity $\forall (a:A)\ (b:B) \ldots (z:Z), \mathrm{Type}$, we have $\Gamma_a = [(a,A);(b,B);\ldots;(z,Z)]$. The $\mathscr{E}^{TS}$ function reduces the term and then, if it is a lambda-abstraction, looks at the result of `flag_of_type` for the domain type. If it is a sort (or, more generally, an arity) it adds a type variable. If the reduced term is not a lambda abstraction, we know that it requires η-expansion, since its type is $\forall (a':A'), t$. Therefore, we call $\mathscr{E}^{TS}_{\eta}$ with the arity context $(a',A') :: \Gamma_a$. In $\mathscr{E}^{TS}_{\eta}$, we use the $\uparrow_1 t$ operation that increments the indices of all variables in t by one. The term $((\uparrow_1 t)\ \overline{0})$ denotes an application of a term t with incremented variable indices to a variable with index $\overline{0}$ A simple example of a type scheme is the following:

```
Definition Arrow (A B : Type) := A→ B.
```

It erases to a pair consisting of a list of type variables and a `box_type`:

$$([A;B], \overline{0} \longrightarrow \overline{1})$$

Type schemes that use dependent types can also be erased. For example, one can create an abbreviation for the type of sized lists.

```
Definition vec (A : Type) (n : nat) := {xs : list A | length xs = n}.
```

which gives us the following type alias

$$([A;n], \mathtt{sig}\ (\mathtt{list}\ \overline{0})\ \ \square)$$

where `sig` corresponds to the dependent pair type in Coq given by the notation `{xs : list A | length xs = n } := sig (list A) (fun xs ⇒ length xs = n)`. The erased type can be further optimised by removing the occurrences of $\square$ and irrelevant type variables.

The two changes described above bring our implementation closer to the standard extraction of Coq and allow for more programs to be extracted in comparison with our previous work. Returning $\mathbb{T}$ instead of failing creates more opportunities for target languages that support unsafe type casts.

Having defined the erasure procedure for types, we implement an erasure procedure for inductive definitions. Bringing it all together with the verified erasure of MetaCoq and the erasure for type schemes, we can define a procedure that erases lists of global declarations, which are called *global environments*. We enrich the representation of global environments of the MetaCoq's erasure with the typing information we obtained using $\mathscr{E}^{T}$ and $\mathscr{E}^{TS}$. Each entry in the global environment is represented by the following inductive type.

```
$\mathcal{E}^{TS}$ : Ctx → ECtx → ACtx → term → $\mathbb{N}$
    → list name × box_type
$\mathcal{E}^{TS}\ \Gamma\ \Gamma_e\ [\,]\ t\ v_n$ = ([ ], snd ($\mathcal{E}^{T}\ \Gamma\ \Gamma_e\ t$ None ))
$\mathcal{E}^{TS}\ \Gamma\ \Gamma_e\ ((a', A') :: \Gamma_a)\ t\ v_n$ =
    let $t'$ := red$_{\beta\iota\zeta}\ \Gamma\ t$ in
      match $t'$ with
      | $\lambda\ (a : A).b$ ⇒
        let $flag$ := flag_of_type $\Gamma\ A$ in
        let $v_n'$ := if (is_arity $flag$) then $v_n + 1$
             else $v_n$ in
        let $kind$ := if (is_arity $flag$) then (TV $v_n$)
             else Other in
        let $(vs, \tau)$ :=
          $\mathcal{E}^{TS}\ (A :: \Gamma)\ (kind :: \Gamma_e)\ \Gamma_a\ b\ v_n'$ in
        $(a :: vs, \tau)$
      | _ ⇒ $\mathcal{E}^{TS}_{\eta}\ \Gamma\ \Gamma_e\ ((a', A') :: \Gamma_a)\ t\ v_n$
      end
```

```
$\mathcal{E}^{TS}_{\eta}$ : Ctx → ECtx → ACtx → term → $\mathbb{N}$
    → list name × box_type
$\mathcal{E}^{TS}_{\eta}\ \Gamma\ \Gamma_e\ [\,]\ t\ v_n$ = ([ ], snd ($\mathcal{E}^{T}\ \Gamma\ \Gamma_e\ t$ None ))
$\mathcal{E}^{TS}_{\eta}\ \Gamma\ \Gamma_e\ ((a, A) :: \Gamma_a)\ t\ v_n$ =
    let $flag$ := flag_of_type $\Gamma\ A$ in
    let $v_n'$ := if (is_arity $flag$) then $v_n + 1$
         else $v_n$ in
    let $kind$ := if (is_arity $flag$) then (TV $v_n$)
         else Other in
    let $(vs, \tau)$ :=
      $\mathcal{E}^{TS}_{\eta}\ (A :: \Gamma)\ (kind :: \Gamma_e)\ \Gamma_a\ ((\uparrow_1 t)\ \overline{0})\ v_n'$ in
    $(a :: vs, \tau)$
```

Fig. 3: Erasure for type schemes.

```
Inductive global_decl :=
| ConstantDecl : constant_body → global_decl
| InductiveDecl : mutual_inductive_body → global_decl
| TypeAliasDecl : option (list type_var_info * box_type) → global_decl.
```

where `constant_body` adds the constant's erased type (the `cst_type` field), which is absent in the corresponding definition of MetaCoq's $\lambda_{\square}$:

```
Record constant_body :=
{ cst_type : list name * box_type;  cst_body : option term; }.
```

Moreover, `mutual_inductive_body` is enriched with typing information as well. We explicitly treat type aliases by having a separate entry `TypeAliasDecl`, which corresponds to type schemes. We call the representation above $\lambda_{\square}^{T}$ and use it as an intermediate representation.

5.1.2 *Optimising extracted code*

Our second extension of the verified erasure is *deboxing*—a simple optimisation procedure for removing some redundant constructs (boxes) left after the erasure step. First, we observe that removing redundant boxes is a special case of more general optimisation: dead argument elimination. Informally it boils down to the equivalence $(\lambda x.\ t)\ v \sim t$ when x does not occur free in t. Here $\sim$ means that both sides evaluate to the same value. Then, deboxing becomes a special case: $(\lambda A.\ t)\ \square \sim t$. Where the A variable corresponds, for example, to the type abstraction $\lambda(A : \texttt{Type}).\ t$. Erasure replaces the occurrences of A in t with boxes.[9] Having in mind this equivalence, we implement in Coq a function with the following signature:

$$\mathit{dearg} : \texttt{ind\_masks} \to \texttt{cst\_masks} \to \texttt{term} \to \texttt{term}$$

[9] In our implementation, we do not rely on this property and instead more generally remove unused parameters.

The first two parameters are lookup tables for inductive definitions and for constants defining which arguments of constructors and constants are unused. The information about unused arguments is represented using *masks*—lists of boolean values with `true` denoting the unused arguments. The type `term` represents $\lambda_\square$ terms. The *dearg* function traverses the term and adjusts all applications of constants and constructors using the masks.

We define the following function that processes the definitions of constants:

$$\mathit{dearg\_cst} : \texttt{ind\_masks} \to \texttt{cst\_masks} \to \texttt{constant\_body} \to \texttt{constant\_body}$$

This function deargs the body using *dearg* and additionally removes lambda abstractions in correspondence to the mask for the current constant. Note that, since the masks apply only to constants in the program, we only remove dead arguments of top-level functions: abstractions representing closures are untouched. Additionally, as dearging removes arguments from the top-level function, we must adjust the type signatures produced by the type erasure correspondingly. For example, for the constant `Definition foo (n m k : nat) := n` we get a mask `mask = [ false; true; true]` and the optimised constant `Definition foo (n : nat) := n`

To generate the masks we implement an analysis procedure that finds dead parameters of constants and dead constructor arguments. For arguments of constants, we check syntactically if they do not appear in the body, while for constructor arguments we find all unused arguments in pattern matches and projections across the whole program. This is implemented as a linear pass over each function body that marks all uses of arguments and constructor arguments in that function. As we noted above, the erased arguments will be unused and therefore this procedure gives us a safe way of removing many redundant boxes, cf. Letouzey (2004, Section 4.3).

The syntactic check is quite imprecise; for example, it will not remove a parameter if its only use is to be passed to another function in which it is also unused. To deal with this, the analysis and dearging procedure can be iterated multiple times, but since our main use of the dearging is to remove arguments that are erased, this is not necessary.

For definitions of inductive types, we define the function

$$\mathit{dearg\_mib} : \texttt{mib\_masks} \to \mathbb{N} \to \texttt{one\_inductive\_body} \to \texttt{one\_inductive\_body}$$

which adjusts the definition of one inductive's body of a (possibly) mutual inductive definition. With *dearg_cst* and *dearg_mib*, we can now define a function that removes arguments according to given masks for all definitions in the global environment:

$$\mathit{dearg\_env} : \texttt{ind\_masks} \to \texttt{cst\_masks} \to \texttt{global\_env} \to \texttt{global\_env}$$

Dearging is then done by first analysing the environment to obtain `ind_masks` and `cst_masks` and then applying the *dearg_env* function.

We prove dearging correct under several assumptions on the masks and the program being erased. First, we assume that all definitions in the program are closed, which is a reasonable assumption given by typing. Secondly, we assume that the masks are *valid*, meaning that all removed arguments of constants and constructors should be unused in the program. By unused we mean that the argument does not syntactically appear except for in the binder. The analysis phase outlined above is responsible for generating masks that satisfy this condition, although currently, we do not prove this and instead recheck that the condition holds for the masks that were output. Finally, we assume that the program is η-expanded according to all the masks: all occurrences of constructors and constants should be applied to the arguments that are supposed to be removed. We implement a *certifying* procedure that performs η-expansion and generates proofs that the expanded terms are equal to the original ones (see Section 5.2). The erasure procedure is a pruning transformation, meaning that it does not remove abstractions or arguments in applications, it just replaces some terms with $\square$. Therefore, η-expanded terms are preserved by erasure. We, however, have not formalised this result and currently validate the terms after erasure to ensure that they are applied enough.

Our Coq formalisation features a proof of the following soundness theorem about the *dearg* function.

Theorem 1 (Soundness of dearging).
`extraction/theories/OptimizeCorrectness.v:dearg_correct`
Let Σ be a closed erased environment and t a closed $\lambda_\square$-term such that Σ and t are valid and expanded according to provided masks.
Then

$$\Sigma \vdash t \triangleright v$$

implies

$$dearg\_env(\Sigma) \vdash dearg(t) \triangleright dearg(v)$$

where dearging is done using the provided masks.

Here $- \vdash - \triangleright -$ denotes the big-step call-by-value evaluation relation of $\lambda_\square$ terms[10] and values are given as a subset of terms. The theorem ensures that the dynamic behaviour is preserved by the optimisation function. This result, combined with the fact that the erasure from CIC to $\lambda_\square$ preserves dynamic behaviour as well, gives us guarantees that the terms that evaluate in CIC will be evaluated to related results in $\lambda_\square$ after optimisations.

Theorem 1 is a statement talking only about the dearging optimisation that is used by our extraction. The extraction pipeline itself is more complicated and works as outlined at the end of Section 5.1: it is provided a list of definitions to extract in a well-typed environment and recursively erases these and their dependencies (see the full pipeline in Figure 1). Note that only dependencies that appear in the erased definitions are considered as dependencies; this typically gives an environment that is substantially smaller than the original. Once the procedure has produced an environment, the environment is analysed to find out which arguments can be removed from constructors and constants, and finally, the dearging procedure is invoked.

MetaCoq's correctness proof of erasure requires the full environment to be erased. Since we only erase dependencies, we prove a strengthened version of the erasure correctness theorem that is applicable for our case. Combining this with Theorem 1 allows us to obtain a statement about the extraction pipeline (starting from the PCUIC representation and excluding the pretty-printing).

[10] The relation is part of MetaCoq. We contributed to fixing some issues with the specification of this relation.

Theorem 2 (Soundness of extraction).
`extraction/theories/ExtractionCorrectness.v:extract_correct`
Let Σ be a well-typed axiom-free environment and let C be a constant in Σ. Let Σ' be the environment produced by successful extraction (including optimisations) of C from Σ. Then, for any unerasable constructor $Ctor$, if

$$\Sigma \vdash_p C \triangleright Ctor$$

it holds that

$$\Sigma' \vdash \texttt{C} \triangleright Ctor$$

Here $- \vdash_p - \triangleright -$ denotes the big-step call-by-value evaluation relation for CIC terms. Informally, the above statement can be specialised to say that any program computing a boolean value will compute the same value after extraction. Of course, one still has to keep in mind that the pretty-printing step of the extracted environment is not verified, and there are discrepancies of $\lambda_\square$ and the target language's semantics as we outlined in Section 5.1. We also discuss in Section 5.1.4 how the handling of absurd cases (e.g. `False_rect`) interacts with the dearging transformation. With the translation from the Template Coq representation to PCUIC (ongoing work in the MetaCoq team), we would have a complete end-to-end theorem covering all steps that happen after quoting a Gallina term and before printing the resulting code. The `quote` functionality would still remain the the TCB.

While the statement does not say anything about constructor applications,[11] it does informally generalise to any value that can be encoded as a number, since it can be used to show that each bit of the output will be the same.

One of the premises of Theorem 2 is that the environment is axiom-free, which is required for the soundness of erasure as stated in MetaCoq and adapted in our work. In general, we cannot say anything about the evaluation of terms once axioms are involved. One possible way of fixing this issue is by following the semantic approach as in Letouzey (2004, Section 2.4).

While dearging subsumes deboxing we cannot guarantee that our optimisation removes all boxes even for constants applied to all logical arguments due to cumulativity.[12] E.g. for `@inl Prop Prop True : sum Prop Prop` it is tempting to optimise the extracted version `inl □ □ □` into just `inl`, but the optimised definition of the `sum` type will still have the `inl` constructor that takes one argument, because its type is `inl : forall A B : Type, A → A + B` and the argument `A` is, in general, relevant for computation.

As mentioned previously, the dearging of functions removes parameters which means that it must also adjust the type signatures of those functions. In addition to this adjustment of type signatures, we also do a final pass to remove logical inductive type parameters. We define functions that perform *deboxing of types* `box_type` and, eventually on the contents

[11] It is hard to give an easily understandable statement since dearging removes applications.

[12] By cumulativity we mean subtyping for universes. Particularly relevant for us: if `A : Prop`, then it is also `A : Type`$_i$ for some level i. Therefore, if a function takes an argument `A : Type`, we can pass `Prop`, since it is at the lowest level of the universe hierarchy.

of the global environment.

$$\textit{debox\_box\_type} : \texttt{box\_type} \to \texttt{box\_type}$$
$$\textit{debox\_env\_types} : \texttt{global\_env} \to \texttt{global\_env}$$

Note that the deboxing of types does not affect the structure of arrow types, since these are connected to dearging. Instead, it recursively finds inductive types `I` and type constants `C` applied to boxes □ and filters out the boxes. This step is completely orthogonal to the dearging of terms and serves only to remove useless type parameters. This does not affect the dynamic semantics, but mistakes in it might mean that the code does not type-check in the target language.

For a concrete example, sigma types are defined in Coq as

```
Inductive sig (A : Type) (P : A→ Prop) :=
  exist : forall x : A, P x→ sig A P
```

In the constructor, `P` is a type scheme while the argument of type `P x` is a proof, so these are erased by type erasure, resulting in the type `A` ⟶ □ ⟶ `sig A` □. The analysis will show that the proof argument is never used since any use is also erased. This means the constructor is changed to `A` ⟶ `sig A` □ as part of the dearging process, and any use of this constructor in a function (e.g. for pattern matching, or to construct a value) is similarly adjusted. Finally, removal of logical type parameters (deboxing of types) means that the type parameter P is completely removed from the type, giving the final constructor type as `A` ⟶ `sig A`. Function signatures using `sig` are also adjusted correspondingly, having the P argument removed.

After applying the optimisations, we pretty-print the optimised code to several functional languages. We discuss issues related to extraction to Liquidity and CameLIGO in Section 5.3, to Elm in Section 5.4, and to Rust in Section 5.5.

5.1.3 Optimisations and library APIs

While it is often beneficial to remove all unused arguments, it is also necessary, in some cases, to maintain compatibility with APIs when extracting libraries. That is, some arguments should not be removed. The signatures of the extracted functions exposed to external code should not depend on their implementation.

The separation between analysis and optimisation helps us make the pipeline more configurable. We allow the users to override masks for arbitrary constants and inductive types. This way, one can provide masks preserving all (or some required number of) arguments during the optimisation step. The users can decide what functions should have a fixed signature by providing appropriate masks. The signature-preserving masks can be generated by a separate analysis procedure. For example, such a procedure can generate a mask for the occurrences of logical types in signatures to avoid removing dead arguments that can change the signatures. One can check whether this mask is a sub-mask of the mask returned by our current analysis and use it to override dearging for selected definitions. Thus, our dearging can reproduce the same behaviour as the standard Coq extraction, while allowing for more aggressive optimisations if required.

In fact, overriding masks has proven to be useful for integrating smart contracts with the blockchain infrastructure. In this case, the main contract functions must have a fixed signature to match the harness we use to run the extracted code. Another situation where it is important to keep some arguments is discussed in Section 5.1.4.

5.1.4 Handling absurd cases

Our approach should be able to handle the cases when Coq programs contain some unreachable code, originating from provably impossible cases. As an example let us consider the following program in Coq.

```
Program Definition safe_head {A} (non_empty_list : {l : list A | length l > 0}) : A :=
  match non_empty_list as l' return l' = non_empty_list→ A  with
  | []⇒ fun _⇒ False_rect _ _
  | hd :: tl⇒ fun _⇒ hd
  end eq_refl.
```

The type of the program ensures that one always can take the first element of the input list. In the body of the program, we have to deal with two cases for the given list. Clearly, we should never hit the empty list case. Therefore, we use `False_rect : forall P : Type, False → P` that allows us to construct anything, provided we have a contradiction at hand. Using the `Program` tactic we construct such proof from the fact that `length [] > 0` is in fact a contradiction. In Coq, this definition gives us a total function `safe_head`, which we then can use in other definitions, provided that we can construct an element of `{l : list A | length l > 0}`. For example, we can use it in the following program.

```
Program Definition head_of_repeat_plus_one {A} (n : nat) (a : A) : A
  := safe_head (repeat a (1+n)).
Next Obligation. intros. cbn. lia. Qed.
```

However, in the extracted code, `safe_head` must return some value of the appropriate type in the case of an empty list. It can be done in different ways, depending on the features available in a target language. One way of doing this would be to throw an exception in languages that support this kind of side effect. E.g. the standard Coq extraction to OCaml uses `assert false` for that purpose. For the languages that do not support exceptions, we can use non-termination for the same purpose. Therefore, once we encounter pattern-matchings on any empty inductive type (an inductive type with no constructors, e.g. `False`) we replace it with a construct that throws an exception or a call to a non-terminating function.

There is an issue, however, with this naive approach. It is related to subtle interactions of constants that eliminate from an empty type, like `False_rect`, with our dearging optimisation in eager languages. In our initial version, we produced masks for constants like `False_rect` that instructed the optimisation procedure to remove all arguments. Even though it is sound at the $\lambda_\square^T$ level by Theorem 1, at the pretty-printing stage, the body of `False_rect` will be replaced with something like `assert false`, which will be immediately evaluated in eager languages, leading to exceptions. That is, the soundness guarantees of Theorem 1 could be broken by the actual generated code. In our current implementation, we have adopted the following strategy, following Letouzey (2004), At the analysis stage, if all the arguments of a constant are logical (i.e. of type $\square$ or $\mathbb{T}$) we generate a mask that keeps one argument, guarding the constant's body by a lambda abstraction. That is, we never remove all the arguments of constants that have purely logical arguments. This also applies to `Acc_rect` and related machinery for defining functions by well-founded recursion, where removing all arguments may lead to non-termination.

```
Definition storage := Z.
Inductive msg := Inc (_ : Z) | Dec (_ : Z).
Program Definition inc_counter (st : storage) (inc : {z : Z | 0 <? z}) :
  {new_st : storage | st <? new_st} := st + inc. (* proof omitted *)
Program Definition dec_counter (st : storage) (dec : {z : Z | 0 <? z}) :
  {new_st : storage | new_st <? st} := st - dec. (* proof omitted *)
Definition my_bool_dec := Eval compute in bool_dec.

Definition counter (msg : msg) (st : storage)
  : option (list operation * storage) :=
  match msg with
  | Inc i ⇒ match (my_bool_dec (0 <? i) true) with
    | left h ⇒ Some ([], proj1_sig (inc_counter st (exist _ i h)))
    | right _ ⇒ None
    end
  | Dec i ⇒ match (my_bool_dec (0 <? i) true) with
    | left h ⇒ Some ([], proj1_sig (dec_counter st (exist _ i h)))
    | right _ ⇒ None
    end
  end.
```

Fig. 4: The `counter` contract.

The separation of analysis and optimisation gives us a flexible way of implementing the standard fixes for such issues. However, these situations highlight the fact that the final extraction guarantees can be given only after careful consideration of the semantics of target languages. Formally, it can be fixed by connecting our development to formalisations of target languages, as we sketched in Section 2. We remark on how the pattern-matching on empty types is implemented in our targets in the corresponding sections.

5.1.5 The counter contract

As an example, let us consider a simple smart contract represented as a Gallina function. The state of the contract is an integer number and it accepts increment and decrement messages (Figure 4, `extraction/examples/CounterSubsetTypes.v`). The main functionality is given by the two functions `inc_counter` and `dec_counter`. We use subset types to encode the functional specification of these functions. E.g. for `inc_counter` we encode in the type that the result of the increment is greater than the previous state given a positive increment. Subset types are represented in Coq as dependent pairs (Σ-types). For example a positive integer is encoded as `{z : Z | 0 <? z}`, where the second component is a proposition `0 <? i = true` (we use an implicit coercion from booleans to propositions). Similarly, we encode the specification `dec_counter`. The `counter` function validates the input and provides a proof that the input satisfies the precondition (of being positive). The functions `inc_counter` and `dec_counter` are defined only for positive increments and decrements, therefore, we do not need to validate the input again. Note that in order to construct an inhabitant of `positive`, we use the decidability of equality for booleans `bool_dec : forall b1 b2 : bool, {b1 = b2} + {b1 <> b2}` that gives us access to the proof of `0 <? i`. We will use the example from Figure 4 in subsequent sections for showing how it can be extracted to concrete target languages.

5.2 Proof-generating transformations

The optimisation pass in our pipeline (see Figure 1) expects constants and constructors to be applied to all logical arguments in order to be valid. Moreover, some constants have types that are too expressive for the target languages that can make the extracted programs untypable. However, the constants can be specialised in a way that the extracted code is well-typed. In order to ensure that our input satisfies these and some other technical requirements, we apply transformation passes at the very beginning of our pipeline—at the Template Coq level (see Figure 1). These transformation passes are implemented as unverified functions transforming the Template Coq AST. In order to ensure that the passes are computationally sound, we apply the *certifying* approach to transformation. It is similar to how certifying compilers are used to produce proof-carrying code (Necula, 1997). The overall idea is that the transformation produces a new program (in our case it is the same language) and a proof term that the desired property is preserved by the transformation. Each transformation in the Template Coq part of the pipeline has the following type

```
transform: global_env→ Result global_env string
```

where `global_env` is the Template Coq global environment (list of top level declarations), `Result` is a error monad. Given a list of transforms, we can compose them using the fact that `Result` is a monad. In fact, we can reuse the same way of composing transformation for different passes in our pipeline and define a common type of transformations as `Definition Transform (A : Type) := A → result A string`. As a result, we define the composition of transformation in the usual monadic way.

After successfully completing all the transformations, we can generate proofs that the definitions we transformed behave in the same way as the originals. All the transformations we have considered have one property in common: they produce terms that are definitionally equal to the originals. Definitional equality in Coq means that the two terms are *convertible*, i.e. equivalent with respect to $\beta\delta\iota\zeta$-reduction, η-expansion and irrelevant terms in `SProp`.[13] Where β and η are standard and δ means constant unfolding, ι—reduction of `match` on a constructor, ζ—`let .. in` reduction. From the computational point of view, convertible terms represent the same program. The fact that the terms are convertible gives us a simple way of generating the correctness proofs. Let `trans` be a transformation function and `t0 : A` a term. If `trans t0 = Ok t1`, i.e. the application of this function to `t0` succeeds with some transformed term `t1`, we can construct the following proof term:

```
@eq_refl A t1 : t0 = t1
```

This proof term shows that we can prove that the two terms `t0` and `t1` are equal in the theory of Coq. Moreover, this term is well-typed only if `t0` and `t1` are convertible.

We use the following approach to generating the proof terms:

- Given a definition `def`, quote it along with all the dependencies, producing a global environment Σ_0 with quoted terms.
- Apply the composed transformations to all elements in the original global environment Σ_0 and get the transformed environment Σ_1.
- For each constant from Σ_0 find a corresponding constant in Σ_1.
- If a constant is found, compare the constant bodies for *syntactic* equality (it is possible since we operate in meta-theory). In case the bodies are not equal—add (unquote)

[13] See more about the conversion mechanism in Coq's manual: https://coq.inria.fr/refman/language/core/conversion.html (accessed 2021-07-23).

a new definition from Σ_1 to the current scope; if they are equal, or constant not found—do nothing.
- If `def` or its dependencies were affected by the transformation, generate a proof term and add (unquote) it to the current scope.

The certifying approach is quite flexible wrt. changes and additions of new passes since no modifications of proofs are required, provided that the passes preserve convertibility. This is a big advantage in our setting when fine-tuning of the transformations is required for achieving the desired result (see, for example, the inlining transformation below). Potentially, the pass can be extended with more general optimising transformations like partial evaluation.

Below, we describe transformations currently implemented in our framework.

η-expansion `extraction/theories/CertifyingEta.v`**.** The idea is to find partially applied constants (or constructors) and expand them by introducing lambda-abstractions. For example for a term `let f := fun n ⇒ add n in f 0 0` would be expanded (if we demand full η-expansion) to `let f := fun n m ⇒ add n m in f 0 0`. The extent to which the expansion is performed is controlled by lookup tables mapping the names of constants (or constructor information) to a number, indicating the number of arguments that should be added, and the constant's (constructor's) type. The typing information is required for introducing the lambda-abstractions since the Template Coq unquote functionality expects a fully specified term, and all binders typically have explicit types in the AST. Calling the type checker would introduce too much overhead, therefore, we keep the required information in the lookup tables. The transformation is mostly standard but requires a bit of care when dealing with types of lambda-abstractions. Let us consider an example, writing all relevant types explicitly. For the following code `let f : list nat → list nat := @cons nat 0 in f []`. Our expansion table will contain the type of `cons : forall {A : Type}, A → list A → list A`. In order to introduce a lambda-abstraction, we need to know the type of the last argument of `cons`. Therefore, we need to specialise the type of `cons` wrt. the arguments it is applied to. We do so by substituting the arguments, to which the constant or a term is applied, into the term's type.

Since our main use case for η-expansion is to ensure that constants and constructors are applied to all logical arguments, we use the masks generated by the analysis phase for optimisations (see Section 5.1) to compute to which extent constants and constructors should be η-expanded.

Since η-equality is part of Coq's conversion mechanism, the resulting terms will be convertible to the originals.

Expansion of `match` branches. This transformation is tightly related to the representation of the `match` construct in Coq. The branches are represented as a list with each position corresponding to a constructor of the type of the discriminee.[14] Each element of the list of branches is a pair with the first component being a number of a constructor's arguments, and the second—a term, that can be applied to the number of arguments, specified in the

[14] By discriminee we mean a term on which the pattern-matching is performed.

first component. The second component might be not η-expanded. Let us consider a simple (contrived) example.

```
Definition match_list_id (xs : list nat) : list nat :=
match xs with
| [] ⇒ []
| cons x xs ⇒ cons x xs
end.
```

The internal representation of the branches is a list, admitting different ways of representing the second branch. For example, it is perfectly fine to just use the cons constructor applied only to the type of elements, but not to the other two arguments. This list looks as follows in term of the AST constructors (we abbreviate the MetaCoq representation of the list of natural numbers as LIST and the type of natural numbers as NAT).

```
  [(0, tApp (tConstruct LIST 0 []) [NAT]);
   (2, tApp (tConstruct LIST 1 []) [NAT])]
```

The pretty-printing procedure expects that all the branches start with lambdas if the corresponding patterns have arguments. This invariant makes it possible to print the patterns in the usual way, with the top lambda-abstractions becoming pattern variables. Therefore, we would like to expand the second branch, so it has the following shape (we abbreviate the binder information for lambda-abstractions as X and XS):

```
tLambda X NAT
  (tLambda XS LIST
    (tApp (tConstruct LIST 1 []) [NAT; tRel 1; tRel 0])))])
```

Or, written in the concrete syntax `fun (x : nat) (xs : list nat) ⇒ cons x xs`.

In most cases, writing a program in Coq does not lead to unexpanded representation of branches, but we have noticed that certain automatically generated definitions, like eliminators, might contain branches that are not expanded enough. One example of such a definition is sig_rect, an eliminator for the sig type from the standard library of Coq. Without expansion, such definitions would prevent us from using our extraction pipeline.

The implementation of the branch expansion is similar to the η-expansion pass with one subtlety. As we have noted before, we need to specify types for each binder introduced by lambda-abstractions. Getting the information about the type of branches is quite complicated and with the current representation of branches in Template Coq would require running type inference. Instead, we use a recent feature of Template Coq, called *holes*. Holes in Coq are represented by so-called existential variables, that can be manipulated by tactics and instantiated by the elaboration mechanism. In our case, the surrounding context provides enough information for these variables to be instantiated. Implementation-wise, due to similarities with the "regular" η-expansion, the passes are defined together.

Inlining extraction/theories/CertifyingInlining.v. The motivation for having an inlining pass is that some definitions that are not typable in the extracted code, become typable after inlining and specialising the bodies. Inlining also helps to overcome some potential performance issues. We have two common examples of this kind.

- Dependent eliminators. The code produced after extraction might be not typable because the original type is more expressive than prenex polymorphism in our target languages. Languages like CameLIGO do not support polymorphism at all. Moreover, using eliminators like bool_rect (non-dependent version of it is essentially `if_then_else`) is impractical, because the target languages use the call-by-value evaluation strategy. Therefore, evaluating expressions like `bool_rect _ branch1 branch_2 cond` will effectively lead to evaluating both branches regardless of the condition, while we would like it to behave similarly to `if_then_else`. After inlining, `bool_rect` unfolds to pattern-matching and behaves as expected.

- General definition of a monad. The definition of a monad uses rank-2 polymorphism, which, again, goes beyond the supported types in the target languages. But inlining concrete instances of `bind` and `return` allows us to avoid this issue and continue using high-level abstraction in Coq while extracting the well-typed code.

In our framework, the inlining function has the following signature.

```
template_inline : (kername → bool) → Transform global_env
```

The argument is a function indicating which constants should be inlined. Apart from just inlining the bodies of specified constants, we also perform ι and β-reductions. The extent to which the term is reduced is determined empirically from the applications to extraction. Clearly, since inlining is δ-reduction, accompanied with ι- and β-reductions, the resulting terms are convertible to the original ones since all these reductions are part of Coq's conversion mechanism.

5.3 Extracting to Liquidity and CameLIGO

Liquidity is a functional smart contract language for the Tezos and Dune blockchains inspired by OCaml. It compiles to Michelson[15]—a stack-based functional core language supported directly by the blockchain, developed by Tezos.

LIGO is another functional smart contract language for Tezos that compiles to Michelson. LIGO has several concrete syntaxes: PascaLIGO, ReasonLIGO, and CameLIGO. We target the CameLIGO syntax due to its similarity with Coq.

Compared to a conventional functional language, Liquidity and CameLIGO have many restrictions, mostly inherited from Michelson. Hence, we briefly present the key issues when extracting to these languages.

In both Liquidity and CameLIGO, data types are limited to non-recursive inductive types, and support for recursive definitions is limited to tail recursion on a single argument.[16] That means that one is forced to use primitive container types to write programs. Therefore, the functions on lists and finite maps must be replaced with "native" versions in the extracted code. We achieve this by providing a translation table that maps names of Coq functions to the corresponding Liquidity/CameLIGO primitives. Moreover, since the recursive functions can take only a single argument, multiple arguments need to be packed into a tuple. The same applies to data type constructors since the constructors take a tuple of arguments. Currently, the packing into tuples is done by the pretty-printers after verifying that constructors are fully applied.

[15] `https://tezos.gitlab.io/active/michelson.html` (accessed 2021-07-21).

[16] We reported the restrictions to the developers: recursive functions `https://github.com/OCamlPro/liquidity/issues/265` and `https://gitlab.com/ligolang/ligo/-/issues/1248`, data types `https://github.com/OCamlPro/liquidity/issues/266`.

Another issue is related to the type inference in Liquidity and CameLIGO. Due to the support of overloaded operations on numbers, type inference requires type annotations. We solve this issue by providing a "prelude" for extracted contracts that specifies all required operations on numbers with explicit type annotations. This also simplifies the remapping of Coq operations to the Liquidity/CameLIGO primitives. Moreover, we produce type annotations for top-level definitions.

In Coq `Record`s are simply inductive types with one constructor. At the pretty-printing stage, we identify any such types and print them as native records. Liquidity does not allow records with only a single field, or inductive types with one constructor. In this case, we print the type as an alias for the type of the field/constructor. For example, consider the Coq record below, and the function `get_x` which retrieves the `x` field of the record using Coq's built-in record projection syntax.

```
Record A := {
  x : nat;
}.
Definition get_x (n : A) : nat := n.(x).
```

This printed to Liquidity as

```
type a = nat
let get_x (n : a) = n
```

Note in particular how the projection `a.(x)` is printed simply as `a`.

As a consequence of these restrictions on Liquidity, one should use either type aliases or single-field records in place of inductive types with one constructor in the Coq code. These restrictions only apply if the contract is to be extracted to Liquidity. For the other target languages, these restrictions do not apply.

Higher-order functions in Liquidity. Some standard functional programming patterns do not work in Liquidity due to some non-standard features of its type system. For example, the type of a closure contains information about the environment to which it is closed.[17] For that reason, some programs, which are completely unproblematic in many functional languages are not accepted by the Liquidity compiler. For example, the following program refuses to compile

```
let my_map (f : int→ int) (xs : int list) =
  List.map f xs

let bar (i : int) (xs : int list) =
  my_map (fun (x : int)→ x + i) xs
```

producing a type error `Types ((int -> int)[@closure :int]) and int -> int are not compatible`. The `my_map` function expects a function of type `nat → nat`, but the call of `my_map` in the body of `bar` gets a function of a different type: `(int → int)[@closure :int]`, where `:int` refers to the type of the variable `i`. This makes using higher-order functions highly problematic. Moreover, this problem extends to closures returned from different branches of `match` expressions, limiting the number of programs that one can extract to Liquidity without additional efforts.

Handling absurd cases. We follow the general strategy outlined in Section 5.1.4. Both Liquidity and CameLIGO feature an effectful operation `failwith`, which allows for

[17] See https://github.com/OCamlPro/liquidity/issues/264.

interrupting the contract execution. We identify pattern-matchings with no branches at the pretty-printing stage and insert the `failwith` operation. However, inlining some constants (e.g. `False_rect`) is required in order to make examples like `safe_head` to compile with the CameLIGO compiler, otherwise, extraction produces polymorphic definitions, which are not supported. In Liquidity, however, we run into issues unrelated to the handling of absurd cases. The `failwith` works as expected, but closure types carry information about the environment wrt. which they are closed. Dependent pattern-matching in Coq produces code with many closures, which are not accepted by the Liquidity compiler. Therefore, extraction of programs to Liquidity that extensively uses dependent pattern-matching is currently limited.

Explicit type annotations in CameLIGO. LIGO's typechecker is not able to infer types in some instances. These are

- 0-ary constructors of primitive parametric types, e.g. `None` and the empty list `[ ]`;
- function types and in particular arguments of lambda expressions;
- the type of `failwith`.

Therefore we need to add explicit type annotations to these terms. To do this, we augment $\lambda_\Box$ terms with their types, obtained using the erasure procedure for types Section 5.1.1. We designed a general annotation procedure that can add arbitrary data to the $\lambda_\Box$ AST nodes without changing the AST representation itself. This is achieved using dependent types: we implement a procedure that for each AST node builds a (nested) product type recursively. The fragment of the procedure is given below.

```
Fixpoint annots {A : Type} (t : term) : Type :=
  match t with
  | tLambda _ body⇒ A * annots body
  | tApp hd arg⇒ A * (annots hd * annots arg)
  ...
  end.
```

For CameLIGO, we specialise our annotation machinery to `box_type` giving us the type of annotated terms `annots box_type t` for any term `t` of $\lambda_\Box$. This type-augmented representation is (optional) part of our intermediate representation $\lambda_\Box^T$.

The CameLIGO pretty-printer function recurses on the annotated terms and utilises the typing information whenever is necessary. Since the annotation pass is done separately and independently from the pretty-printer, it may be used for other purposes or new target languages in the future.

No polymorphic types in CameLIGO. Unlike Liquidity, CameLIGO does not currently support user-defined polymorphic types, but there is ongoing work to support polymorphic types in the near future. One possibility to circumvent this restriction is to implement a full specialisation pass that produces completely monomorphised code. However, with the prospect of support for polymorphic types, we instead simply ignore this restriction, although we are aware not all the examples will type check currently.[18]

[18] Update from Feb, 2022. The changes from the MR `https://gitlab.com/ligolang/ligo/-/merge_requests/1294` were incorporated into the LIGO compiler. Starting from v0.31.0 CameLIGO supports polymorphic functions and parametric types. The example code from Appendix B is well-typed and compiles to Michelson.

Integration. In order to generate code for a contract's entry points (functions through which one can interact with the contract), we need to wrap the calls to the main functionality of the contract into a `match` construction. This is required because the signature of the entry point in Liquidity and CameLIGO is `params` → `storage` → (`operation list`) ∗ `storage`, where `params` is a user-defined type of parameters, `storage` a user-defined state and `operation` is a transfer of contract call. The signature looks like a total function, but since Liquidity and CameLIGO support a side effect `failwith`, the entry-point function can still fail. On the other hand, in our Coq development, we use the `option` monad to represent computations that can fail. For that reason, we generate a wrapper that matches on the result of the extracted function and calls `failwith` if it returns `None`.

The `ChainBase` type class represents an address abstraction, specifying which properties are required from a type of addresses for accounts. Smart contracts defined using the execution layer infrastructure are abstracted over a `ChainBase` instance. That means that types `Chain` and `ContractCallContext`, along with `init` and `receive` functions will get an additional parameter corresponding to the `ChainBase` instance. When printing the contract code, we need to remap `Chain` and `ContractCallContext` to their representation in the target language, and the dependency on `ChainBase` makes it problematic. We define a specialisation procedure that specialises all definitions dependent on `ChainBase` to an axiomatic instance and removes the corresponding parameter. Currently, this procedure is defined on PCUIC representation and is not verified.

Examples. The extracted counter contract code to Liquidity and CameLIGO is given, respectively, in Figure 5(a) and (b). We omit some wrapper code and the "prelude" definitions and leave the most relevant parts (see Appendices A and B for the full versions). As one can see, the extraction procedure removes all "logical" parts from the original Coq code. Particularly, the `sig` type of Coq becomes a simple wrapper for a value (`type 'a sig_ = 'a` in the extracted code). Currently, we resort to an ad hoc remapping of `sig` to the wrapper `sig_` because Liquidity and CameLIGO do not support variant types with only a single constructor. The remapping is done using the translation table, as described in the end of Section 5.1. Ideally, this class of transformations can be added as an optimisation for inductive types with only one constructor taking a single argument. This example shows that for certain target languages optimisation is a necessity rather than an option.

We show the extracted code for the `coq_inc_counter` function and omit `coq_dec_counter`, which is extracted in a similar manner. These functions are called from the `counter` function that performs input validation. Since the only way of interacting with the contract is by calling `counter` it is safe to execute them without additional input validation, exactly as it is specified in the original Coq code.

Apart from the example in Figure 4, we successfully applied the developed extraction to several variants of the counter contract, the crowdfunding contract described in Annenkov *et al.* (2020), the contracts from Sections 6 and 7 and an interpreter for a simple expression language. The latter example shows the possibility of extracting certified interpreters for domain-specific languages such as Marlowe (Lamela Seijas & Thompson, 2018), CSL (Henglein *et al.*, 2020) and the CL language (Bahr *et al.*, 2015; Annenkov & Elsman,

(a)

```
type 'a sig_ = 'a
let exist_ a = a

type coq_msg = Coq_Inc of int | Coq_Dec of int
type storage = int
type coq_sumbool = Coq_left | Coq_right


let coq_inc_counter (st : storage) (inc : int sig_) =
     exist_ (addInt st ((fun x → x) inc))
...

let coq_counter (msg : coq_msg) (st : storage) =
match msg with
  Coq_Inc i →
  (match coq_my_bool_dec (ltInt 0 i) true with
    Coq_left  →
    Some ([], ((fun x → x)
       (coq_inc_counter st (exist_ (i)))))
  | Coq_right  → None)
| Coq_Dec i → ...
  | Coq_right  → None)
```

Liquidity

(b)

```
type 'a sig_ = 'a
let exist_ (a : _a) : _a = a
let id_func (a : _a) : _a = a
type coq_msg = Coq_Inc of int | Coq_Dec of int
type storage = int
type coq_sumbool = Coq_Left | Coq_Right
let coq_Transaction_none  = ([]: (operation) list)

let coq_inc_counter (st : storage) (inc :  int sig_) =
  exist_ ((addInt st (id_func inc)))
...

let coq_counter (msg : coq_msg) (st : storage) =
match msg with
  Coq_Inc (i) →
  (match coq_bool_dec true (ltInt 0 i) with
    Coq_Left → (Some
    (coq_Transaction_none,
    (id_func (coq_inc_counter st (exist_ (i))))))
  | Coq_Right  →
(None: (operation list * storage) option))
| Coq_Dec (i) → ...
```

CameLIGO

Fig. 5: Extracted code.

2018). This represents an important step towards safe smart contract programming. The examples show that smart contracts fit well to the fragment of Coq that extracts to well-typed Liquidity and CameLIGO programs. Moreover, in many cases, our optimisation procedure removes all the boxes resulting in cleaner code.

Gas consumption. Running smart contracts on a blockchain requires so-called "gas", which serves as a measure of computational efforts. The execution environment calculates the gas consumption according to the cost of each operation and uses it to terminate the execution if the maximum gas consumption is reached. The caller pays a fee in the blockchain's internal currency for calling a contract. If the fee is too low and the gas consumption is too high, there is a chance that the transaction will not be included in a block. This behaviour is slightly different from Ethereum, but we will not provide the details here.

We have deployed and executed some of our extracted contracts on test networks (these networks use the same execution model as the main one, but no real money is required to run contracts). Comparing the gas consumption shows that extracted contracts perform well in the realistic setting, even though, the extracted code consumes more gas compared to a hand-written implementation. We have compared the ERC20 token implementation against the Liquidity code from the online IDE. The extracted code consumes 2–2.5 times more gas, but the consumption is quite far from reaching the hard limit on a single contract call. We have also experimented with the prototype DSL interpreter extracted from our Coq developments on the Tezos network. The recommended fees, converted to US dollars were lower than \$0.03 even for DSL programs with 100 instructions. Such transaction costs can be considered negligible. Most of the gas consumption can be attributed to type checking of a contract for each call, which, of course, depends on its length and complexity. However, gas consumption and the associated fees become smaller with each update of the Tezos network, making the transaction fees negligible for many common use cases. The threshold on gas consumption also increases, allowing for more expressive smart contracts. Therefore, our smart contract extraction is able to deliver verified code with reasonable execution costs.

5.4 Extracting to Elm

Elm (Feldman, 2020) is a general purpose functional language used for web development. It is based on an extended variant of the Hindley–Milner type system and can infer types without user-provided type annotations in most situations. However, we generate type annotations for top-level definitions to make the extracted code more readable. Moreover, unlike Liquidity, there is no restriction in Elm regarding data types with one constructor. That allows implementing a simple extraction procedure for Coq records as data types with one constructor and projections defined as functions that pattern-match on this constructor. Compared to Liquidity and CameLIGO, Elm is a better target for code extraction, since it does not have some limitations pointed out in Section 5.3.

Extraction to Elm also poses some challenges. For example, Elm does not allow shadowing of variables and definitions. Since Coq allows for a more flexible approach to naming, one has to track scopes of variables and generate fresh names in order to avoid clashes. The syntax of Elm is indentation sensitive, so we are required to track indentation levels. Various naming conventions apply to Elm identifiers, e.g. function names start with a lower-case character, types and constructors—with an upper case character, requiring some names to be changed when printing.

Handling absurd cases. We follow the general strategy outlined in Section 5.1.4. Elm is a pure functional language and does not feature exceptions, which we could use to handle the absurd cases. However, there is one side-effect at our disposal, namely, non-termination. Therefore, we define the following constant.

```
false_rec : () → a
false_rec _ = false_rec ()
```

At the pretty-printing stage, we identify pattern-matchings with no branches and insert a call to `false_rec`.

Examples `extraction/examples/ElmExtractExamples.v`**.** We tested the extracted code with the Elm compiler by generating a simple test for each extracted function. We implemented several examples by extracting functions on lists from the standard library of Coq, functions using subset types and functions that eliminate absurd cases by exploiting contradictions. All our examples resulted in well-typed Elm code after extraction. Particularly, in Figure 6(b) one can see the `safe_head` function (a head of a non-empty list) from Section 5.1.4. The example uses the elimination principle `False_rect` in the case of an empty list, exploiting the contradiction with the assumption that the input list is non-empty. We used the usual style of writing functions with dependent types in Coq with the help of the `Program` tactic.

As a result, the logical parts corresponding to proofs are erased and Coq's implementation of subset types is extracted as a simple wrapper `type Sig a = Exist a`. In the impossible (absurd) case, `safe_head` calls `false_rect`, which is implemented in terms of `false_rec`, using our strategy of handling absurd cases. We also extract a example function that uses `safe_head`:

```
Program Definition head_of_repeat_plus_one {A} (n : nat) (a : A) : A
  := safe_head (repeat a (1+n)).
```

(a)

```
type List a
  = Nil
  | Cons a (List a)


-- appending two lists
app : List a → List a → List a
app l m =
  case l of
    Nil →
      m
    Cons a l1 →
      Cons a (app l1 m)


-- reversing a list
rev : List a → List a
rev l =
  case l of
    Nil →
      Nil
    Cons x l2 →
      app (rev l2) (Cons x Nil)
```

`app` and `rev` in Elm

(b)

```
false_rec : () → a
false_rec _ = false_rec ()

-- definitions of Nat and List are omitted
...

type Sig a = Exist a

proj1_sig : Sig a → a
proj1_sig e =
  case e of
    Exist a → a

false_rect : () → p
false_rect p = false_rec ()

safe_head : Sig (List a) → a
safe_head non_empty_list =
  (case proj1_sig non_empty_list of
     Nil → \x → false_rect ()
     Cons hd tl → \x → hd) ()

head_of_repeat_plus_one : Nat → a → a
head_of_repeat_plus_one n a =
  safe_head (Exist (repeat a (add (S O) n)))
```

`safe_head` in Elm

Fig. 6: Extracted test code in Elm.

Clearly, it is always safe to take the first element from a list that is generated by repetition $1 + n$ times. Therefore, the whole program is safe to use and the absurd case will never be hit. When extracting programs as libraries, one could move *some* static checks to runtime to ensure that the invariants, expected by dependently typed functions are preserved.

We also extract the Ackermann function `ackermann : nat * nat → nat` defined using well-founded recursion which uses the lexicographic ordering on pairs. This shows that extraction of definitions based on the accessibility predicate `Acc` is possible. Computation with `Acc` is studied in more detail in Sozeau & Mangin (2019).

Verified Web Application `extraction/examples/ElmForms.v`**.** We develop a more Elm-specific example in Coq: a simple web application. A typical Elm web application follows the Elm architecture (TEA):

1. Model—a data type representing the state of the application.
2. Update—a function that takes a message, a previous state (model instance) and returns a new state and, potentially, commands (e.g. sending data to a server, etc.)
3. View—a function that turns the model into HTML. HTML is generated using special Elm functions, available as part of the Elm standard library.

If we look at the first two items, they look very similar to the smart contract execution model. At the moment, we do not provide an Elm-specific execution model as part of our framework, but we can leverage Coq dependent types to encode some invariants of the model and then use our extraction pipeline to produce an Elm web application. Therefore, we implement the model and the update functionality along with validation rules in Coq, extract it to Elm and combine it with hand-written rendering code (the view).

The example we consider is inspired by the Elm guide on forms. Our application consists of an input form, a validator and a view for rendering a list of valid items. The input form consists of three fields: a username, user's password, and a field to re-enter the password. In Coq, we model it with the following code.

```
Record Entry := { name : string;
                  password : string;
                  passwordAgain : string }.
```

This part of the model contains "raw" data, as entered by a user. We define then "valid data" using the subset types of Coq.

```
Definition ValidEntry :=
  {entry : Entry | entry.(name) ≠ "" ∧
                   8 ≤? String.length entry.(password) ∧
                   entry.(password) =? entry.(passwordAgain)}.
```

We use the boolean versions of less-or-equal (≤?) and equality (=?) on strings, which are implicitly coerced to propositions using `is_true (p : bool) : Prop := p = true`. This representation makes the interaction with the validation function easier. Then, we define a type of entries that we are going to store in the list of users in the model. In the same way, we define what it means to be valid for such stored entries.

```
Record StoredEntry :=
  { seName : string; sePassword : string }.

Definition ValidStoredEntry :=
  { entry : StoredEntry | entry.(seName) ≠ "" ∧
                          8 ≤? String.length entry.(sePassword)}.
```

Having defined `ValidStoredEntry`, we can then proceed with the definition of a model for the whole application.

```
Record Model :=
  { (** A list of valid entries such with unique user names *)
    users : {l : list ValidStoredEntry | NoDup (seNames l)};
    (** A list of errors after validation *)
    errors : list string;
    (** Current user input *)
    currentEntry : Entry }.
```

As one can see, users in our model are represented as a list of valid entries without duplication of names. Next, we define the messages for updating the model.

```
(** Messages for updating the model according to the current user input *)
Inductive Msg :=
  | MsgName (_ : string)
  | MsgPassword (_ : string)
  | MsgPasswordAgain (_ : string).

(* Messages for updating the current entry and adding the current entry
   to the list of users *)
Inductive StorageMsg :=
   Add
 | UpdateEntry (_ : Msg).
```

Now, we can define a function that performs updates to the model by interpreting the messages it receives.

```
Program Definition updateModel : StorageMsg→ Model→ Model * Cmd StorageMsg
  := fun msg model⇒
       match msg with
```

```
| Add⇒ match validateModel model with
        | [] ⇒ let validEntry : ValidEntry := model.(currentEntry) in
               let newValidStoredEntry : ValidStoredEntry :=
                 toValidStoredEntry validEntry in
               let newList := newValidStoredEntry :: model.(users) in
               (model<| users := newList |>, none)
        | errs⇒ (model<| errors := errs |>, none)
        end
| UpdateEntry entryMsg⇒
        (model<|currentEntry := updateEntry entryMsg model.(currentEntry) |>, none)
end.
```

We use the record update notation `model<| users := newList |>` that uses type classes and Template Coq-based generation of field setters (part of our development). We also use the standard way of working with subset types in Coq using `Program` command that allows writing code in the style of regular functional programming while manipulating richer types under the hood. `Program` inserts projections from the values of subset types and constructs values, leaving the proof component as an obligation that the user can prove later.

The main idea behind having valid entries is that most of the functionality of our web application manipulates valid data. This guarantees that no invariants can be broken by these functions. The validation is performed only once, at the "entry point" of our application, the `updateModel` function and it is driven by the validity predicates of the components of the model. Therefore, when writing a validation function, it would be impossible to miss some validation rule, because valid data requires explicit proofs of validity. Since the Elm architecture guarantees that the only way our model is updated when users interact with the web application is by calling the `updateModel` function, we know that in the extracted code the model invariant will not be broken.

The term produced by `Program` might be quite complex, due to the transformations and elaboration required to produce a fully typed term. Our extraction pipeline is able to cope with the terms generated by `Program` and can be run completely in Coq itself. We define the required remappings to replace the usage of standard functions with Elm counterparts. E.g. we remap Coq types `string`, `list`, `bool` and the product type to the corresponding types in Elm. We also remap natural numbers of Coq to type of bounded integers `Int`. In principle, using bounded numbers might be a problem, but in our case, the only use of numbers is for computing the password length, and `String.length` in Elm has type `String → Int`. Therefore, our choice is coherent with the assumptions about string length in Elm.

We use the inlining pass in the pipeline to inline some of the record update infrastructure. Inlining also prevents us from generating a type alias (related to the records update infrastructure) that is invalid in Elm due to an unused type parameter, which is not allowed.

Overall, we show that one can use the usual certified programming style in Coq in order to implement the logic of a web application that can be then extracted to a fully functional Elm web application (provided that the view functionality is written directly in Elm). The generated application is well-typed in Elm, even though we have used dependent types extensively.

5.5 Extracting to Rust

Rust is a mixed paradigm general-purpose programming language that features many of the same concepts as functional programming languages. It is aimed at being a fast language with low overhead, which also makes it an attractive smart contract programming language. Therefore, it provides a lot of control and is also a relatively low-level programming language. The Concordium blockchain toolchain uses Rust as its primary programming language for writing smart contracts. The actual code that is executed on-chain is WebAssembly. WebAssembly is designed to be a safe, portable and efficient low-level language with well-defined semantics making it well-suited for verification in a proof assistant (Watt, 2018). Like Rust, WebAssembly does not feature a garbage collector, making it a good target for compiling Rust.

When writing smart contracts in Rust, the implementors have more ways of controlling performance. One of the most expensive operations on blockchains is updating the contract's state. Rust allows for destructive updates of the mutable contract state with the precise control of the serialisation/deserialisation process allowing for careful performance tuning. Using the Concordium toolchain, smart contracts written in Rust are compiled to WebAssembly modules using LLVM. The WebAssembly modules can be then deployed and executed on-chain.

Rust has a powerful functional subset that includes

- Sum/product types
- Pattern matching
- Higher-order functions and closures
- Immutability by default
- Everything-is-an-expression
- A Hindley–Milner (without let-polymorphism) based type system

These features make Rust a suitable and relatively straightforward target for printing from $\lambda^{T}_{\square}$. However, as Rust is a low-level language giving a lot of control, it also comes with its own set of challenges.

Extracting data types. The type system of Rust features the concept of *ownership*—variable bindings have ownership of what they bind. The ownership system is a substructural type system allowing for precise control of the resource usage: allocating, deallocating, copying, etc. It is used in Rust as a zero-cost abstraction allowing for safe memory management. The part of the compiler called the *borrow checker* checks that the resources are used according to the ownership rules.

In Rust, the programmer controls whether fields of data structures are stored by-value or through indirection. For recursive data structures, such as linked lists, it is necessary to use indirection since otherwise, the size of the data type would be infinite. Concretely, this means that a type such as

```
Inductive list (A : Type) :=
  | nil
  | cons (head : A) (tail : list A).
```

cannot be extracted in a straightforward way where the tail is just of type `list A`. Instead, it is necessary to use indirection to store a form of a pointer to the tail of the list. In Rust, there are several ways to store indirection, including C-like raw pointers, borrowed references, owned references, and through reference counting (the `Rc` and `Arc` types). The benefit of the `Box`, `Rc`, and `Arc` types is that ownership is managed implicitly, while for raw pointers and borrowed references it is necessary to store the data somewhere else. *Borrowing* means passing a reference to something without transferring the ownership. Roughly speaking, a borrowed reference can be seen as a pointer with additional static guarantees provided by the type system of Rust. An owned reference `Box<T>` is a smart pointer, pointing to the data allocated on the heap.

Since functional languages generally rely on sharing to perform well the same sharing should be supported in the final extracted Rust program. In particular, this disqualifies owned references as those can only be shared through expensive copying. Reference counted types in Rust require explicit cloning to manually indicate when the reference count must be incremented. This complicates extraction significantly as extraction then has to determine that it needs to insert such clonings when passing arguments and when capturing local variables for closures.

As a result of these considerations, the extraction uses borrowed references to store nested data types. Such references are trivially copyable and can be shared freely, but require that the data be stored somewhere else. Additionally, this requires data structures to be generalised over a *lifetime*. Lifetimes are used by the borrow checker to determine how long references are valid. In some situations, they can be inferred by the compiler. However, storing references in data types require explicit lifetime parameters. For uniformity, we add a lifetime to all data types we extract. As Rust datatypes must use all lifetimes and type parameters they introduce, the extraction also adds "phantom" uses of these through the use of `PhantomData`, a zero-cost Rust type meant to specify to the Rust compiler that a lifetime or type parameter is allowed to be unused. For uniformity, such PhantomData types are emitted as the first member of all data types in all constructors, leading to a final extraction of lists as

```
enum list<'a, A> {
  nil(PhantomData<&'a A>),
  cons(PhantomData<&'a A>, A, &'a list<'a, A>)
}
```

where the `'a` parameter is lifetime, and `A` is a type parameter. The `&'a` syntax corresponds to a borrowed reference with lifetime `'a`.

Memory model differences. Rust is an unmanaged language without a garbage collector. When extracting from a language like Coq, in which all memory allocation is handled implicitly for the programmer, this leads to some challenges. This is made significantly easier when it is noted that smart contract execution is self-contained and very short-lived. Due to this, it is feasible to allocate as much memory as necessary during the execution and then clean up only after the execution is done, a technique known as region-based memory allocation. The extraction can thus use an off-the-shelf library that implements region-based memory allocation; in particular, the Bumpalo library is used.

For more general-purpose extraction of programs that may be long-running, we are considering using a conservative garbage collector such as the Boehm–Demers–Weiser (Boehm & Weiser, 1988; Boehm *et al.*, 1991) garbage collector. Here the challenge

```
pub enum nat<'a> {
  O(PhantomData<&'a ()>),
  S(PhantomData<&'a ()>, &'a nat<'a>)
}

struct Program {
  __alloc: bumpalo::Bump,
}

impl<'a> Program {
  fn new() → Self {
    Program {
      __alloc: bumpalo::Bump::new(),
    }
  }

  fn alloc<T>(&'a self, t: T) → &'a T {
    self.__alloc.alloc(t)
  }

  fn closure<TArg, TRet>(&'a self, F: impl Fn(TArg) → TRet + 'a) → &'a dyn Fn(TArg)
 → TRet {
    self.__alloc.alloc(F)
  }

  fn add(&'a self, n: &'a nat<'a>, m: &'a nat<'a>) → &'a nat<'a> {
    match n {
      &nat::O(_) ⇒ { m },
      &nat::S(_, p) ⇒ { self.alloc(nat::S(PhantomData, self.add(p, m))) },
    }
  }

  fn add__curried(&'a self) → &'a dyn Fn(&'a nat<'a>) → &'a dyn Fn(&'a nat<'a>)
 → &'a nat<'a> {
    self.closure(move |n| { self.closure(move |m| { self.add(n, m) }) })
  }
}
```

Fig. 7: The add function extracted to Rust.

lies in implementing the right heuristics to figure out when garbage collection should be invoked during an extracted program.

Extraction produces a structure `Program` that contains the region (or arena) of memory that can be allocated from. Structures `struct` are similar to records, but in addition to fields can have functions associated with them. Such functions are called methods. The first parameter of a method is `&self`, a reference to the instance of the `struct`. Methods are defined in the corresponding `impl` block.

We extract the entire program as methods on the `Program` structure that can then access the region when memory allocation is required. As an example, consider the function `add : nat → nat → nat`. Note that we produce two versions of `add`: curried and uncurried. The curried version returns a closure implemented using dynamically dispatched trait objects of type `Fn`. We discuss the details of closure extraction and partial applications below. The extracted code of the `add` function with all of its dependencies is presented in Figure 7.

Our Rust extraction also supports remapping and that `nat` normally would be remapped to either a big-integer type or to the `u64` type using checked arithmetic. With the example we have considered, including functions on lists and other inductive types, extracting a *complete implementation* of a program produces well-typed code, also wrt. ownership. However, we have not attempted to fully extract complex container libraries, such as `FMaps`.

It is sometimes necessary to remap Coq container types (lists, finite maps) to corresponding Rust libraries. In this case, one has to take care of the ownership issues manually for the corresponding remapped fragments of code. This can be done by first defining a wrapper function manually in Rust, having in mind the memory model of the extracted code. One can then make sure that the wrapper code compiles, so it can be added to a translation table for pretty-printing.

Handling 'monomorphised' closures. In order to handle polymorphic functions (functions with type parameters), the Rust compiler performs a transformation called *monomorphisation*. That means that the compiler generates copies of a generic function with parameters replaced with concrete types, used in the program. Rust implements closures in an efficient way by combining their code with the environment they capture into an anonymous type. Functions can be monomorphised with respect to these types, allowing the use of closures to be a zero-cost abstraction. For example, closures can be inlined as if they are normal functions or stored directly in data structures. However, the semantics of such closures are different from the semantics of closures in traditional functional languages.

In Coq, a closure behaves like any other function and is fully compatible with other functions of that function type. For example, it is possible and unproblematic to store multiple different closures of the same function type in a list. This uniform behaviour does not carry over to Rust's default treatment of closures: when storing a closure in a list, the list must be typed over the anonymous closure type that the compiler has generated automatically. Therefore, it is not possible to store two different closures, even of the same function type, in such a list.

Rust still allows for semantics that match Coq's at the cost of some performance through *trait objects*. Trait objects use virtual dispatch to allow for example closures to behave uniformly as functions, hiding away the associated environment.[19] Trait objects can exist only as a reference and extraction must thus allocate closures and turn them into references. The extraction automatically provides the helper function `closure` that performs this allocation using the same region-based allocation as described above. In some cases, the Rust compiler requires annotations when using closures through allocated trait objects. Therefore, we use the following wrapper to aid the type inference.

```
fn hint_app<TArg, TRet>(f: &dyn Fn(TArg) → TRet) → &dyn Fn(TArg) → TRet { f }
```

The `dyn` keyword denotes the fact that the `Fn` trait object is dynamically dispatched. We conservatively insert the wrapper whenever the head of the application is not a constant, a constructor, or an application.

Partial applications. Rust requires all functions to be fully applied when called, unlike Coq which supports partial application. Partial applications can be emulated easily through closures, by generating both curried versions and uncurried versions of functions.

[19] Note that Rust also has "regular" traits that are similar to type classes and are statically dispatched.

However, using closures is less efficient, so as an optimization the extraction avoids closures when possible. Concretely, this results in both curried and uncurried versions as is seen in the extraction above, with the curried version calling into the uncurried version.

Internal fixpoints. Coq supports recursive closures through the `fix` construct. In comparison, Rust does not have similar support for recursive closures and supports only recursive local functions which do not allow capturing. This means that only top-level recursive Coq functions can straightforwardly be made recursive during extraction; when a fixpoint is used internally (for example, through a `let fix` binding), there is no simple way to extract this. To work around this issue, we apply a technique known as "Landin's knot" (Landin, 1964). Namely, our extraction uses recursion through the heap. Concretely, when an internal fixpoint is encountered, extraction produces code that allocates a cell on the heap to store a reference to the closure. The closure can access this heap cell and thus access itself when it needs to recurse. To exemplify, a straightforward definition of the Ackermann function in Coq uses nested recursion:

```
Fixpoint ack (n m : nat) : nat :=
  match n with
  | O ⇒ S m
  | S p ⇒
    let fix ackn (m : nat) :=
          match m with
          | O ⇒ ack p 1
          | S q ⇒ ack p (ackn q)
          end
    in ackn m
  end.
```

Non-mutual top-level fixpoints are extracted as a common special case by transforming the fixpoint's arguments into lambda-abstractions before printing. See `/extraction/theories/TopLevelFixes.v`. The inner fixpoints are extracted using "Landin's knot". Our extraction produces the following Rust code.

```
fn ack(&'a self, n: &'a Nat<'a>, m: &'a Nat<'a>) → &'a Nat<'a> {
  match n {
    &Nat::O(_) ⇒ { self.alloc(Nat::S(PhantomData, m)) },
    &Nat::S(_, p) ⇒ {
      let ackn = {
        let ackn = self.alloc(std::cell::Cell::new(None));
        ackn.set(Some(
          self.closure(move |m2| {
            match m2 {
              &Nat::O(_) ⇒ {
                self.ack(
                  p,
                  self.alloc(Nat::S(PhantomData, self.alloc(Nat::O(PhantomData))))
                )
              },
              &Nat::S(_, q) ⇒ { self.ack(p, ackn.get().unwrap()(q)) },
            }
          })));
        ackn.get().unwrap()
      };
      ackn(m)
    },
  }
}
```

Mutual internal fixpoints are also supported. In fact any mutual fixpoint is extracted as an internal fixpoint, even if it is a top-level definition.

Handling absurd cases. We follow the general strategy outlined in Section 5.1.4. The natural choice for implementing the elimination principle for an empty type is to use Rust's `panic`! macro. In this case the elimination principle for `False` extracts to the following Rust code.

```
fn False_rect<P: Copy>(&'a self, u: ()) → P {
  panic!("Absurd case!")
}
```

We identify pattern-matchings with no branches at the pretty-printing stage and insert the `panic`! macro.

Integrating with Concordium `extraction/theories/ConcordiumExtract.v`. Concordium maintains the smart contract state in a serialised form, i.e. as an array of bytes. Similarly, when a smart contract is called, its message is passed as an array of bytes. To aid in conversion between the smart contract's data types and these byte arrays, the Concordium toolchain provides automatic derivation of serialisers and deserialisers between arrays of bytes and standard Rust data types. This conversion, however, does not support references, as it is unclear how to deserialise into a reference. In addition, the ConCert smart contracts extracted are not immediately compatible with the signatures expected by Concordium.

To aid in connecting between ConCert and Concordium a standard library is provided by ConCert. This standard library includes several helper types that extracted smart contracts depend on, and additionally also provide procedural macros that can be used to derive serializers and deserializers that, through the use of regions, support deserialising references. When a smart contract is extracted, it automatically has serializers and deserializers derived for its structures, and the extraction takes care to generate glue code that properly performs deserialisation and serialization with a proper region. Finally, the glue code also connects between Concordium's expected smart contract signature and the one extracted by ConCert.
We proceed to highlight some case studies using the ConCert framework.

6 The escrow contract

In this section, we present an *escrow* contract `execution/examples/Escrow.v`. It is an example of a nontrivial contract we can verify and extract. The purpose of this contract is to enable a seller to sell goods in a trustless setting via the blockchain. The Escrow contract is suited for goods that cannot be delivered digitally over the blockchain; for goods that can be delivered digitally, there are contracts with better properties, such as FairSwap (Dziembowski *et al*., 2018).

Because goods are not delivered on-chain, there is no way for the contract to verify that the buyer has received the item. Instead, the contract incentivises the parties to follow the protocol by requiring that both parties commit additional money that they are paid back at the end. Assuming a seller wants to sell a physical item for x amount of currency, the contract proceeds in the following steps:

1. The seller deploys the contract and commits (by including the amount with the deployment call) $2x$.
2. The buyer commits $2x$ before a deadline.
3. The seller delivers the goods (outside of the smart contract).
4. The buyer confirms (by sending a message to the smart contract) that they have received the item. They can then withdraw x from the contract while the seller can withdraw $3x$ from the contract.

If there is no buyer who commits funds, the seller can withdraw their money back after the deadline. Note that when the buyer has received the item, they can choose not to notify the smart contract that this has happened. In this case, they will lose out on x, but the seller will lose out on $3x$. In our work, we assume that this does not happen, and we consider the exact game-theoretic analysis of the protocol to be out of scope. Instead, we focus on proving the *logic* of the smart contract correct under the assumption that both parties follow the protocol to completion. The logic of the Escrow is implemented in approx. a hundred lines of Gallina code. The interface to the Escrow is its message type given below.

```
Inductive Msg :=
  commit_money
| confirm_item_received
| withdraw.
```

To state correctness, we first need a definition of what the escrow's effect on a party's balance has been.

Definition 3 (Net balance effect).
execution/examples/Escrow.v:net_balance_effect
Let π be an execution trace and a be an address of some party. Let T_{from} be the set of transactions from the Escrow to a in π, and let T_{to} be the set of transactions from a to the contract in π. Then the net balance effect of the Escrow on a is defined to be the sum of amounts in T_{from}, minus the sum of amounts in T_{to}.

The Escrow keeps track of when both the buyer and seller have withdrawn their money, after which it marks the sale as completed. This is what we use to state correctness.

Theorem 3 (Escrow correctness).
execution/examples/Escrow.v:escrow_correct
Let π be an execution trace with a finished Escrow for an item of value x. Let S be the address of the seller and B the address of the buyer. Then:

- *If B sent a `confirm_item_received` message to the Escrow, the net balance effect on the buyer is $-x$ and the net balance effect on the seller is x.*
- *Otherwise, the net balance effects on the buyer and seller are both* 0.

Below, we show how the informal statement of Theorem 3 is implemented in Coq using the infrastructure provided by the execution layer (see Section 4). In the comments, we point out the corresponding parts and notations from the informal statement of the theorem.

```
Theorem escrow_correct
        {ChainBuilder : ChainBuilderType}
        prev new header acts :
(* For a trace (π) ending with a successful addition of a block (reachability) *)
builder_add_block prev header acts = Ok new→
let trace := builder_trace new in
forall caddr,
  env_contracts new caddr = Some (Escrow.contract : WeakContract)→
  exists (depinfo : DeploymentInfo Setup)
         (cstate : State)
         (inc_calls : list (ContractCallInfo Msg)),
    deployment_info Setup trace caddr = Some depinfo ∧
    contract_state new caddr = Some cstate ∧
    incoming_calls Msg trace caddr = Some inc_calls ∧
    (* the value of the item (x) *)
    let item_worth := deployment_amount depinfo / 2 in
    (* the address of the seller S *)
    let seller := deployment_from depinfo in
    (* the address of the buyer B *)
    let buyer := setup_buyer (deployment_setup depinfo) in
    is_escrow_finished cstate = true→
    (* the net balance effect is x on the seller and −x on the buyer *)
    (buyer_confirmed inc_calls buyer = true ∧
     net_balance_effect trace caddr seller = item_worth ∧
     net_balance_effect trace caddr buyer =  item_worth ∨
     (* otherwise, the net balance effects on the buyer and seller are both 0. *)
     buyer_confirmed inc_calls buyer = false ∧
     net_balance_effect trace caddr seller = 0 ∧
     net_balance_effect trace caddr buyer = 0).
```

In Coq, we first prove a slightly more general statement of the theorem (`escrow_correct_strong`), which is then used to prove the statement that corresponds to Theorem 3. The proof is by induction on the structure of the contract's execution trace `ChainTrace`. We use a specialised induction principle that allows for better proof structure. Moreover, we provide textual hints for the user for each case when applying the inductive principle in the interactive mode.

Extracting the contract `extraction/examples/EscrowExtract.v`. We have successfully extracted the escrow contract to Rust, CameLIGO, and Liquidity. For CameLIGO and Liquidity, we remap the `Amount` type (which is just an alias for `Z`) to `tez`, the on-chain currency. We also remap the fields of `Chain` and `ContractCallContext` to equivalent API calls in CameLIGO/Liquidity. For example, the `ctx_contract_balance` field of `ContractCallContext` is remapped to `Tezos.balance` for CameLIGO, and `Current.balance` for Liquidity.

Liquidity has a small caveat that it does not allow external functional calls in the initialisation function. Using the inlining transformation described in Section 5.2, we ensure that the necessary function definitions are inlined in the initialisation function. Furthermore, we also inline various monad instances implicitly used in the contract code, such as the instance for `Monad option`, since higher-kinded types are not supported in CameLIGO and Liquidity.

The Rust version of the escrow contract was successfully deployed and instantiated on the Concordium's test network. This demonstrates that the integration infrastructure is fully functional. The size of the resulting WebAssembly executable that was obtained by compiling the extracted contract is about 39KB, while the threshold is 64KB.

7 The boardroom voting contract

Hao, Ryan, and Zielińsky developed the Open Vote Network protocol (Hao *et al.*, 2010), an e-voting protocol that allows a small number of parties ('a boardroom') to vote anonymously on a topic. Their protocol allows tallying the vote while still maintaining maximum voter privacy, meaning that each vote is kept private unless all other parties collude. Each party proves in zero-knowledge to all other parties that they are following the protocol correctly and that their votes are well-formed.

This protocol was implemented as an Ethereum smart contract by McCorry, Shahandashti and Hao (McCorry *et al.*, 2017). In their implementation, the smart contract serves as the orchestrator of the vote by verifying the zero-knowledge proofs and computing the final tally.

We implement a similar contract in the ConCert framework `execution/examples/BoardroomVoting.v`. The original protocol works in three steps. First, there is a sign-up step where each party submits a public key and a zero-knowledge proof that they know the corresponding private key. After this, each party publishes a commitment to their upcoming vote. Finally, each party submits a computation representing their vote, but from which it is computationally intractable to obtain their actual private vote. Together with the vote, they also submit a zero-knowledge proof that this value is well-formed, i.e. it was computed from their private key and a private vote (either 'for' or 'against'). After all parties have submitted their public votes, the contract is able to tally the final result. For more details, see the original paper (Hao *et al.*, 2010). The contract accepts messages given by the type:

```
Inductive Msg :=
| signup (pk : A) (proof : A * Z)
| commit_to_vote (hash : positive)
| submit_vote (v : A) (proof : VoteProof)
| tally_votes.
```

Here, `A` is an element in an arbitrary finite field, `Z` is the type of integers and `positive` can be viewed as the type of finite bit strings. Since the tallying and the zero-knowledge proofs are based on finite field arithmetic, we develop some required theory about $\mathbb{Z}_p$ including Fermat's theorem and the extended Euclidean algorithm. This allows us to instantiate the boardroom voting contract with $\mathbb{Z}_p$ and test it inside Coq using ConCert's executable specification. To make this efficient, we use the Bignums library of Coq to implement operations inside $\mathbb{Z}_p$ efficiently.

The contract provides three functions `make_signup_msg`, `make_commit_msg` and `make_vote_msg` meant to be used off-chain by each party to create the messages that should be sent to the contract. As input, these functions take the party's private data, such as private keys and the private vote, and produces a message containing derived keys and derived votes that can be made public, and also zero-knowledge proofs about these. We prove the zero-knowledge proofs attached will be verified correctly by the contract when these functions are used. Note that, due to this verification done by the contract, the contract is able to detect if a party misbehaves. However, we do not prove formally that incorrect proofs do not verify since this is a probabilistic statement better suited for tools like EasyCrypt or SSProve (Abate *et al.*, 2021).

When creating a vote message using `make_vote_msg` the function is given as input the private vote: either 'for', represented as 1, and 'against', represented as 0. We prove that the contract tallies the vote correctly assuming that the functions provided by the boardroom voting contract are used. Note that the contract accepts the `tally_votes` message only when it has received votes from all public parties, and as a result stores the computed tally in its internal state. We give here a simplified version of the full correctness statement which can be found in the attached artifact.

Theorem 4 (Boardroom voting correct).
`execution/examples/BoardroomVoting.v:boardroom_voting_correct`
Let π be an execution trace with a boardroom voting contract. Assume that all messages to the Boardroom Voting contract in π were created using the functions described above. Then:

- *If the boardroom voting contract has accepted a `tally_votes` message, the tally stored by the contract equals the sum of private votes.*
- *Otherwise, no tally is stored by the contract.*

Below, we show how the informal statement of Theorem 4 is implemented in Coq using the infrastructure provided by the execution layer (see Section 4). In the comments, we point out the corresponding parts and notations from the informal statement of the theorem.

```
Theorem boardroom_voting_correct
      (bstate : ChainState)
      (caddr : Address)
      (* For any trace (π) from the initial state to a reachable state [bstate] *)
      (trace : ChainTrace empty_state bstate)
      (* a list of all public keys, in the order of signups *)
      (pks : list A)
      (* a function mapping a party to information about them *)
      (parties : Address→ SecretVoterInfo) :
  env_contracts bstate caddr = Some (boardroom_voting : WeakContract)→
  exists (cstate : State)
         (depinfo : DeploymentInfo Setup)
         (inc_calls : list (ContractCallInfo Msg)),
    deployment_info Setup trace caddr = Some depinfo ∧
    contract_state bstate caddr = Some cstate ∧
    incoming_calls Msg trace caddr = Some inc_calls ∧

    (* assuming that the message sent were created with the functions
       provided by this smart contract *)
    MsgAssumption pks parties inc_calls→

    (* ..and that people signed up in the order given by 'index' and 'pks' *)
    SignupOrderAssumption pks parties inc_calls→

    (* ..and that the correct number of people register *)
    (finish_registration_by (setup cstate) < Blockchain.current_slot bstate→
     length pks = length (signups inc_calls))→

    (* then if we have not tallied yet, the tally is none *)
    ((has_tallied inc_calls = false→ tally cstate = None) ∧
         (* or if we have tallied yet, the tally is correct *)
         (has_tallied inc_calls = true→
          tally cstate = Some (sumnat (fun party⇒ if svi_sv (parties party) then 1 else 0)
                                      (map fst (signups inc_calls))))).
```

Similarly to the escrow contract from Section 6, we first prove a more general theorem using the specialised induction principle for the execution traces.

Extracting the contract extraction/examples/BoardroomVotingExtractionCameLIGO.v. The boardroom voting contract gives a good benchmark for our extraction as it relies on some expensive computations. It drives our efforts to cover more practical cases, and we have successfully extracted it to CameLIGO. Extraction to Liquidity produces code that fails to type check due to the closure typing as it is presented in Section 5.3. Extraction to Rust should be possible with the appropriate extraction setup. We leave this as future work.

The main problem with extraction for this contract is the use of higher-kinded types. In particular, the implementation of the contract uses finite maps from the std++ library, which implicitly rely on higher-kinded types. In addition, the contract uses monadic binds, implemented via type classes that require passing type families around. Furthermore, the arithmetic operations and developed theory is captured in the type class `BoardroomAxioms (A : Type)`, where `A` is the element type of the finite field, and is instantiated to $\mathbb{Z}_p$ for extraction. All of this is not representable in prenex-polymorphic type systems, and our target languages follow a similar typing discipline to prenex-polymorphism. While we could adjust the implementation to avoid relying on higher kinded types, we instead prefer to improve the extraction to work on more examples. In particular, for our cases, we have identified that a few steps of reduction is enough for most of the higher kinded types to disappear. For example, the signature of `bind` is `forall m : Type → Type, Monad m → forall t u : Type, m t → (t → m u) → m u` which, when it appears in the contract, typically looks like `bind option option_monad ...` where `option_monad` is some constant that builds a record describing the option monad. After very few steps of reduction, this reduces to the well-known bind for options, which is unproblematic to extract. At this point, the pre-processing pass (see Section 5.2) comes in handy and the inlining functionality is sufficient to produce definitions that are well-typed after extraction.

For the `BoardroomAxioms` type class, on which the entire contract is parameterised over, we would need a specialisation pass similar to the `ChainBase` specialisation described in Section 5.3. It could be possible with a more general technique, such as partial evaluation. We leave this as future work, and in the meantime create a copy of the contract where we have inlined $\mathbb{Z}_p$ in place of `A` execution/examples/BoardroomVotingZ.v.

The extracted contract is 470LOC of CameLIGO code, compiled Michelson code is 3KLOC, and the size of the contract in the binary format is 49KB. The size exceeds the limit on direct contract deployment. However, there are mechanisms in the Tezos blockchain, e.g. *global constants* allowing for deployment of contracts exceeding the limit. As future work, it would be interesting to evaluate the gas consumption of the contract. However, without native support for some cryptographic primitives, we expect the gas consumption to be relatively high.

8 Related work

Extraction to statically typed languages. The works in this direction are the most relevant for the present development. By *extraction* we mean obtaining source code in a target language which is accepted by the target language compiler and can be further integrated with existing systems. Several proof assistants share this feature: Coq (Letouzey, 2003), Isabelle (Berghofer & Nipkow, 2002), Agda (Kusee, 2017). They allow targeting conventional functional languages such as Haskell, OCaml or Standard ML. However, extraction in Isabelle/HOL is slightly different from Coq and Agda, since in the higher-order logic of Isabelle/HOL programs are represented as equations and the job of the extraction mechanism is to turn them into executable programs. Moreover, Isabelle/HOL does not feature dependent types, therefore the type system of programs is closer to the extraction targets, in contrast to Coq and Agda, where one has to make additional efforts to remove proofs from terms.

Clearly, the correctness of the extraction code is crucial for producing correct executable programs. This is addressed by several developments for Isabelle (Haftmann & Nipkow, 2007; Hupel & Nipkow, 2018). Hupel & Nipkow (2018) present verified compilation from Isabelle/HOL to CakeML (Kumar *et al*., 2014). It also implements metaprogramming facilities for quoting Isabelle/HOL terms similar to MetaCoq. Moreover, the quoting procedure produces a proof that the quoted terms correspond to the original ones. The current extraction implemented in the Coq proof assistant is not verified. Although the theoretical basis for it is well-developed by Letouzey (2004), Coq's extraction also includes unverified optimisations that are done together with extraction, making it harder to compare it with the formal treatment given by Letouzey. So, the unverified extraction even lacks a full paper proof. Our separation between erasure and optimisation facilitates such comparisons, and allows reuse of the optimisation pass in a standalone fashion in other projects. The MetaCoq project (Sozeau *et al*., 2019) aims to formalise the meta-theory of the calculus of inductive constructions and features a verified erasure procedure that forms the basis for extraction presented in this work. We also emphasise that the previous works on extraction targeted conventional functional languages (e.g. Haskell, OCaml, etc.), while we target the more diverse field of functional smart contract languages.

Šinkarovs & Cockx (2021) present an approach for defining embeddings and extraction procedures at the same time in Agda. The approach is best suited for domain-specific languages and characterises the subset of Agda from which extraction is possible by the successful execution of the extraction procedure. Currently, it seems impossible to establish semantic preservation properties for the extraction/embedding procedures, because the meta-theory of Agda is not formalised. In our setting, we mostly work with general-purpose languages. In this case, applying this approach seems to be problematic, since embedding a general-purpose language can be a non-trivial effort. However, for certain domain-specific contract languages, e.g. Marlowe (Lamela Seijas & Thompson, 2018), CSL (Henglein *et al*., 2020), CL (Bahr *et al*., 2015; Annenkov & Elsman, 2018), the approach of Šinkarovs & Cockx (2021) looks promising. It would be interesting to reproduce the approach in Coq, with the additional benefit of reasoning about the semantics preservation using the MetaCoq formalisation. Currently, we have an example of a simple DSL interpreter extracted from Coq (see examples in Section 5.3) which could be accompanied by an embedding.

The recent developments in quantitative type theory (QTT) by Atkey (2018) offer an interesting perspective on erasure. QTT allows for tracking the resource usage in types, and this information can be used to identify what can be erased at run-time. Agda's GHC backend uses QTT-inspired erasure annotations (Danielsson, 2019) in order to remove computationally irrelevant parts of extracted Haskell programs. However, in our case, it would require significantly changing the underlying theory of Coq. Therefore, such techniques are currently not available to us.

Miquel (2001) develops the implicit calculus of constructions (ICC) that offers an alternative to using `Prop` for separating the computational content from specifications and proofs. ICC adds an *implicit product* type $\forall x : T.U$ allowing for quantifying over x without introducing extra binders at the term level. However, the type checking in ICC is undecidable. Barras & Bernardo (2008) present an annotated variant ICC*, which recovers the decidability. The terms of ICC* can be extracted to ICC by removing the annotations. In the PhD thesis by Bernardo (2015), ICC and its annotated variant were extended with dependent pairs (Σ-types), including an implicit version (similarly to the implicit product). One benefit of using ICC-like type systems is that it allows for more definitional equalities. E.g. for two dependent pairs `(a,p1)` and `(b,p2)` (`p1` and `p2` are proofs of some property on `a` and `b`) are equal whenever the first components are equal. The proofs of the same property are definitionally equal in such systems. The same definitional equality can be obtained in Coq using the universe of definitionally proof-irrelevant propositions `SProp` (Gilbert *et al.*, 2019). However, ICC* allows for making binders of an arbitrary type irrelevant, prohibiting their use in computationally relevant parts of a program. Effectively, it means that irrelevant arguments do not occur in terms of pure ICC (after erasure), but can be used without any restrictions in the codomain of the implicit product type. E.g. `fun {n} (v : vec n) ⇒ n` is ill-typed in ICC*, where `vec` is the type of sized lists (also called vectors). This restriction cannot be expressed using `SProp`. Moreover, implementing the conversion test through extraction to pure ICC gives a very expressive subtyping relation. For example, vectors in this system would be subtypes of lists (using the impredicative encodings for vectors and lists). The approach of ICC looks promising and Barras & Bernardo (2008) report that ICC* allows for a simpler implementation.[20] However, it seems that ICC has not been extended to handle the full calculus of inductive constructions.

Mishra-Linger & Sheard (2008) consider an approach similar to ICC in the context of pure type systems (PTS). The present two calculi: EPTS (Erasure Pure Type Systems—a calculus of annotated terms, similar to ICC*) and IPTS (Implicit Pure Type Systems, similar to ICC). The EPTS calculus features *phase distinction* annotations for distinguishing between compile-time and run-time computations. The authors define an erasure procedure from EPTS to IPTS and briefly discuss some implementation issues. It seems that the implementation of the presented system is not currently available.

Execution of dependently typed languages. Related works in this category are concerned with compiling a dependently-typed language to a low-level representation. Although the techniques used in these approaches are similar to extraction, one does not need to fit the

[20] A prototype implementation is available for older versions of Coq: `http://www.lix.polytechnique.fr/Labo/Bruno.Barras/coq-implicit/`.

extracted code into the type system of a target language and is free to choose an intermediate compiler representation. The dependently typed programming language Idris uses erasure techniques for efficient execution (Brady *et al.*, 2004). The Idris 2 implementation (Brady, 2021) implements QTT for both tracking the resource consumption and the run-time irrelevance information.

Barras & Grégoire (2005) develop an approach for efficient convertibility testing of untyped terms acquired from fully typed CIC terms for the Coq proof assistant. The Œuf project (Mullen *et al.*, 2018) features verified compilation of a restricted subset of Coq's functional language Gallina (no pattern-matching, no user-defined inductive types—only eliminators for particular inductive types). Pit-Claudel *et al.* (2020) report on the extraction of domain-specific languages embedded into Gallina to an imperative intermediate language that can be compiled to efficient low-level code. And finally, the certified compilation approach to executing Coq programs is under development in the CertiCoq project (Anand *et al.*, 2017). The project uses MetaCoq for quotation functionality and uses the verified erasure as the first stage. After several intermediate stages, C light code is emitted and later compiled for a target machine using the CompCert certified compiler (Leroy, 2006). Since we implement our pass as a standalone optimisation on the same AST that is used in CertiCoq, our pass can be integrated in a relatively straightforward fashion in CertiCoq (see Section 9).

Dead arguments elimination. The techniques of removing computationally useless (dead) code were extensively studied in the context of simply-typed (Berardi, 1996) and polymorphic (Boerio, 1994) λ-calculi. The techniques were extended to the calculus of constructions (CoC) by Prost (1995). These techniques analyse the terms to identify unused parts and mark them. As a result, one obtains a *typed* term with some redundancy removed. This captures the proofs that do not contribute to the final result.

We follow the approach initially developed by Paulin-Mohring (1989) for CoC and later adapted and extended to CIC by Letouzey (2004). Namely, all computationally irrelevant propositions must be placed in a special universe `Prop`. However, we apply a pass that removes dead arguments *after* erasure. Letouzey mentioned in his PhD thesis, that doing so has the benefit of working with smaller terms (since large parts are replaced with the $\square$ constructor). Moreover, Letouzey (2003) says that implementation of extraction contains "a workaround designed to safely remove most of the external dummy lambdas". We demonstrate that this workaround can be replaced with a more general and principled optimisation (see Section 5.1.2). Current Coq *implementation*, performs a similar optimisation pass that removes logical arguments. However, it seems that the formal treatment of the transformations is currently lacking. Moreover, we believe that our dearg transformation is flexible enough to accommodate the behaviour of the standard extraction, as we sketched in Section 5.1.3.

Paraskevopoulou *et al.* (2021) implement in CertiCoq several certified optimisation passes. These passes are performed on the *untyped* λ_{ANF} representation and include dead parameter elimination that is similar to our dearg transformation. Our transformation, however, is defined for the $\lambda_\square$ representation, that precedes λ_{ANF} in the CertiCoq pipeline. Our *typed* intermediate representation $\lambda_\square^T$ retains a close connection between the extracted type and the corresponding $\lambda_\square$ term. The λ_{ANF} representation, which is further down the pipeline, is less suited for extraction to typed languages, as it is harder to establish a connection to the original type of the program. Our verified dearg transformation also allows for more general removal of unused constructor parameters. A similar pass in CertiCoq does not handle logical arguments of constructors and is not part of the certified pipeline. We have integrated our pass with the CertiCoq project and have sent a pull request to the CertiCoq repository.[21] With small modifications, the pass seems to be beneficial for the CertiCoq pipeline and can potentially replace a similar unverified pass, but it is not yet merged into the main CertiCoq development. The main reason for that is that our optimisation assumes that constants and constructors are applied to all arguments we remove. That means that there should be an η-expansion pass in the pipeline, which we solve with our proof generating approach. However, there is no such pass in CertiCoq, but there are plans to add such a pass to the MetaCoq development. After that, our optimisation pass could be fully integrated into CertiCoq.

9 Conclusion and future work

We have presented an extraction pipeline implemented completely in the Coq proof assistant. This approach has an important advantage: we can use Coq for providing strong correctness guarantees for the extraction process itself by verifying the passes of the pipeline. The whole range of certified programming and proof techniques becomes applicable since the pipeline consists of ordinary Coq definitions. Our extraction relies on the MetaCoq verified erasure procedure, which we extend with data structures required for extraction to our target languages. Our pipeline addresses new challenges originating from the target languages we have considered and can be extended with new transformations if required.

The developed approach allows for targeting various functional languages. Currently, we support two target languages for smart contract extraction (Liquidity and CameLIGO) and two general-purpose languages (Elm and Rust). Rust is also used as a smart contract language for the Concordium blockchain. We have tested our extraction pipeline on various example programs. In the domain of smart contracts, we have considered several examples both designed for demonstration purposes and representing real-world use cases. The short descriptions of the contracts are given below.

- `Counter`—a simple contract featuring the increment and decrement functionality (similar to the example in Figure 4, but without using the advanced Coq types).
- `Counter (subset types)`—the example in Figure 4.
- `ERC20 token`—an implementation of a widely used token standard.
- `Crowdfunding`—a smart contract representing a common use case, also knows as Crowdsale, Kickstarter-like contract, ICO contract, etc
- `DSL Interpreter`—a simple interpreter, demonstrating a feasibility of embedding interpreted DSLs.
- `Escrow`—an implementation of an escrow (see Section 6).
- `Boardroom voting`—an implementation an anonymous e-voting protocol (see Section 7).

[21] `https://github.com/CertiCoq/certicoq/pull/29` (accessed 2022-02-21).

The examples we have considered confirm that our pipeline is suitable for extracting real-world smart contracts.

In general, our experience shows that the extraction is well-suited for Coq programs in a fragment of Gallina that corresponds to a generic polymorphic functional language extended with subset types. This fragment is sufficient to cover most of the features found in functional smart contract languages and is suitable for extracting many programs to Rust and Elm, resulting in well-typed code. Our pipeline allows for implementing, testing, verifying and extracting programs from Coq to new target languages while retaining a small TCB.

Note, however, that the target languages we consider are different from traditional extraction targets: they are often restricted in various ways. That introduces a difference in the extraction methodology: the standard extraction aims to capture all of Gallina and produce executable code for *any* Coq function, possibly using type coercions (like `Obj.magic`). On the other hand, we identify a suitable fragment that is *extractable* after applying the pre-processing step. This, however, comes at a cost: the user has to use a limited number of constructs and reiterate the extraction process several times if the extracted code is not accepted by the target compiler. So far, many failures can only be seen at the final stage and the debugging process is not ideal. One solution we envision is to introduce a possibility in Coq of restricting to a subset of Gallina using something similar to the `Program` environment. This feature would allow to catch certain issues earlier, before extraction, making the process more convenient for the user. This convenience is, however, orthogonal to the pipeline itself: we can add various steps at the beginning of the pipeline, leaving the rest of the pipeline intact.

The *embedding* techniques give more precise control over the resulting code, but often complicate proving properties of embedded programs. Perhaps, combining the embedding techniques, considered by Annenkov *et al.* (2020) and Šinkarovs & Cockx (2021) with extraction could give more control over the resulting code while retaining the simplicity of reasoning about Coq functions.

As future work one can imagine various additional and improved optimisations, that fit well with the infrastructure we have developed. For example, removing singleton inductive types (e.g. `Acc`), "unboxing" the values built from one-argument constructors application (originating from inductive types with one constructor, e.g. constructors of a subset type `sig`). The proof-generating pass allows for inlining and specialising some definitions which might not be typable after extraction, since our targets do not feature unsafe type casts, like OCaml's `Obj.magic`. Our pipeline is well-suited for adding new conversion-preserving transformations at a very low cost: one just has to write a function, with the signature `global_env` → `Result global_env string` and include it in the list of transformations. The proofs of correctness will be generated automatically after all the transformations have been applied. Even though not all the transformations are conversion preserving, we have given two useful examples of such transformations. Moreover, our pipeline can accommodate more general transformations. For example, partial evaluation, as it is presented by Tanaka (2021), is conversion-reserving for a subset of Gallina. It could be implemented in Coq directly (instead of a plugin) using the metaprogramming facilities of MetaCoq.

We plan also to improve the boardroom voting contract extraction. First, we would like to implement more machinery for program specialisation (like partial evaluation mentioned above), making the manual adjustments of the boardroom voting contract unnecessary. Second, we would like to integrate it with extracted high-performance cryptographic primitives using the approach of FiatCrypto (Erbsen *et al*., 2019). For example, the Open Vote Network protocol, on which the contact is based, depends on computations in a Schnorr group, a large prime-order subgroup of the multiplicative group of integers modulo some prime p. An efficient Rust implementation of a Schnorr group can be obtained from FiatCrypto. We can then replace our naive implementation by this highly optimised implementation.

Since we have already considered new target languages from the ML-family (Elm, Liquidity and CameLIGO), we expect that our pipeline can also be used for extracting to OCaml, similarly to the standard Coq extraction. Currently, the pipeline cannot be used directly as a replacement for the standard extraction. The standard extraction of Coq implements more optimisations than we support in our pipeline. However, our development enables adding more verified optimisations in a compositional manner, giving a systematic way of improving the extraction output. Another issue is that inserting unsafe type coercions (`Obj.magic`) is currently not supported by our development, due to the absence of such mechanisms in most of our targets. Implementing extraction to OCaml could be done by connecting the $\lambda_{\square}^{T}$ representations with the formalisation of a suitable fragment of OCaml including type inference. Such integration would make it possible to use the type inference algorithm to find places where coercions are necessary (Letouzey, 2004, Section 3.2). As the first step towards making our pipeline available for further improvements, we plan to integrate it with the MetaCoq project.

Acknowledgments

This work was partially supported by the Danish Industry Foundation in the Blockchain Academy Network project, the Concordium Blockchain Research Center (COBRA), and the Tezos Foundation.

Conflicts of interest

Aarhus University (PI:Spitters) has received grants from the Concordium Foundation and the Tezos Foundation.

Supplementary materials

For supplementary material for this article, please visit https://doi.org/10.1017/S0956796822000077

References

Abate, C., Haselwarter, P. G., Rivas, E., Muylder, A. V., Winterhalter, T., Hritcu, C., Maillard, K. & Spitters, B. (2021) SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. Cryptology ePrint Archive, Report 2021/397. Available at: `https://eprint.iacr.org/2021/397`.

Anand, A., Appel, A., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O., Sozeau, M. & Weaver, M. (2017) CertiCoq: A verified compiler for Coq. In CoqPL'2017.

Anand, A., Boulier, S., Cohen, C., Sozeau, M. & Tabareau, N. (2018) Towards certified metaprogramming with typed template-Coq. In ITP18. LNCS, vol. 10895, pp. 20–39. Available at: https://hal.archives-ouvertes.fr/hal-01809681

Annenkov, D. & Elsman, M. (2018) Certified compilation of financial contracts. In PPDP'2018.

Annenkov, D., Milo, M., Nielsen, J. B. & Spitters, B. (2021) Extracting smart contracts tested and verified in Coq. In CPP 2021. Association for Computing Machinery, pp. 105–121. Available at: https://doi.org/10.1145/3437992.3439934

Annenkov, D., Nielsen, J. B. & Spitters, B. (2020) ConCert: A smart contract certification framework in Coq. In CPP'2020.

Atkey, R. (2018) Syntax and semantics of quantitative type theory. In Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science. LICS'18. New York, NY, USA: Association for Computing Machinery, pp. 56–65.

Bahr, P., Berthold, J. & Elsman, M. (2015) Certified symbolic management of financial multi-party contracts. *SIGPLAN Not.* **50**(9), pp. 315–327.

Barras, B. & Bernardo, B. (2008) The implicit calculus of constructions as a programming language with dependent types. In *Foundations of Software Science and Computational Structures*, Amadio, R. (ed.). Springer Berlin Heidelberg, pp. 365–379.

Barras, B. & Grégoire, B. (2005) On the role of type decorations in the calculus of inductive constructions. In CSL.

Berardi, S. (1996) Pruning simply typed λ-terms. *J. Logic. Comput.* **6**(5), 663–681. Available at: https://doi.org/10.1093/logcom/6.5.663.

Berghofer, S. & Nipkow, T. (2002) Executing higher order logic. In *Types for Proofs and Programs*, Callaghan, P., Luo, Z., McKinna, J., Pollack, R. & Pollack, R. (eds). Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 24–40.

Bernardo, B. (2015) Un Calcul des Constructions implicite avec sommes dépendantes et à inférence de type décidable. Theses, École polytechnique. Version soutenance. Available at: https://hal.inria.fr/tel-01197380

Boehm, H., Demers, A. J. & Shenker, S. (1991) Mostly parallel garbage collection. In PLDI. ACM, pp. 157–164.

Boehm, H. & Weiser, M. D. (1988) Garbage collection in an uncooperative environment. *Softw. Pract. Exp.* **18**(9), 807–820.

Boerio, L. (1994) Extending pruning techniques to polymorphic second order λ-calculus. In *Programming Languages and Systems—ESOP'94*, Sannella, D. (ed.). Springer Berlin Heidelberg.

Bozman, c., Iguernlala, M., Laporte, M., Le Fessant, F. & Mebsout, A. (2018) Liquidity: OCaml pour la Blockchain. In JFLA18.

Brady, E. (2021) Idris 2: Quantitative type theory in practice. In 35th European Conference on Object-Oriented Programming (ECOOP 2021), Møller, A. & Sridharan, M. (eds). Leibniz International Proceedings in Informatics (LIPIcs), vol. 194. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 9:1–9:26.

Brady, E., McBride, C. & McKinna, J. (2004) Inductive families need not store their indices. In *Types for Proofs and Programs*, Berardi, S., Coppo, M. & Damiani, F. (eds). Springer Berlin Heidelberg, pp. 115–129.

Chapman, J., Kireev, R., Nester, C. & Wadler, P. (2019) System F in Agda, for fun and profit. In MPC'19.

Chlipala, A. (2013) *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press.

Cruz-Filipe, L. & Letouzey, P. (2006) A large-scale experiment in executing extracted programs. *Electron. Notes Theor. Comput. Sci.* **151**(1), pp. 75–91.

Cruz-Filipe, L. & Spitters, B. (2003) Program extraction from large proof developments. In *Theorem Proving in Higher Order Logics*.

Danielsson, N. A. (2019) Logical properties of a modality for erasure. Accessed July 7, 2021. Available at: http://www.cse.chalmers.se/~nad/publications/danielsson-erased.pdf.

Dziembowski, S., Eckey, L. & Faust, S. (2018) FairSwap: How to fairly exchange digital goods. In ACM Conference on Computer and Communications Security. ACM, pp. 967–984.

Erbsen, A., Philipoom, J., Gross, J., Sloan, R. & Chlipala, A. (2019) Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In IEEE Symposium on Security and Privacy.

Feldman, R. (2020) *Elm in Action*. Manning.

Filliâtre, J.-C. & Letouzey, P. (2004) Functors for proofs and programs. In *Programming Languages and Systems*, Schmidt, D. (ed.). Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 370–384.

Gilbert, G., Cockx, J., Sozeau, M. & Tabareau, N. (2019) Definitional proof-irrelevance without K. *Proc. ACM Program. Lang.* **3**(POPL), pp. 1–28.

Haftmann, F. & Nipkow, T. (2007), A code generator framework for Isabelle/HOL. In Department of Computer Science, University of Kaiserslautern.

Hao, F., Ryan, P. Y. & Zieliński, P. (2010) Anonymous voting by two-round public discussion. *IET Inf. Security* **4**(2), pp. 62–67.

Henglein, F., Larsen, C. K. & Murawska, A. (2020) A formally verified static analysis framework for compositional contracts. In *Financial Cryptography and Data Security (FC)*.

Hupel, L. & Nipkow, T. (2018) A verified compiler from Isabelle/HOL to CakeML. In *Programming Languages and Systems*, Ahmed, A. (ed.), pp. 999–1026.

Jung, R., Jourdan, J.-H., Krebbers, R. & Dreyer, D. (2021) Safe systems programming in rust. *Commun. ACM* **64**(4), 144–152.

Klabnik, S. & Nichols, C. (2018) *The Rust Programming Language*. USA: No Starch Press.

Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R. & Heiser, G. (2014) Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.* **32**(1), 2:1–2:70.

Kumar, R., Myreen, M. O., Norrish, M. & Owens, S. (2014) CakeML: A verified implementation of ML. In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'14. ACM, pp. 179–191. Available at: http://doi.acm.org/10.1145/2535838.2535841

Kusee, W. H. (2017) *Compiling Agda to Haskell with Fewer Coercions*. Master's thesis.

Lamela Seijas, P. & Thompson, S. (2018) Marlowe: Financial contracts on blockchain. In International Symposium o Leveraging Applications of Formal Methods, Verification and Validation, Margaria, T. & Steffen, B. (eds). Industrial Practice.

Landin, P. J. (1964) The mechanical evaluation of expressions. *Comput. J.* **6**, 308–320.

Lee, O. & Yi, K. (1998) Proofs about a Folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.* **20**(4), pp. 707–723.

Leroy, X. (2006) Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In POPL, pp. 42–54.

Letouzey, P. (2003) A new extraction for Coq. In *Types for Proofs and Programs*, pp. 200–219.

Letouzey, P. (2004) *Programmation fonctionnelle certifiée – L'extraction de programs dans l'assistant Coq*. PhD thesis, Université Paris-Sud. English version: https://www.irif.fr/~letouzey/download/these_letouzey_English.ps.gz.

McCorry, P., Shahandashti, S. F. & Hao, F. (2017) A smart contract for boardroom voting with maximum voter privacy. In FC 2017.

Miquel, A. (2001) The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications, TLCA'01. Springer-Verlag, pp. 344–359.

Mishra-Linger, N. & Sheard, T. (2008) Erasure and polymorphism in pure type systems. In *Foundations of Software Science and Computational Structures*, Amadio, R. (ed.). Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 350–364.

Mullen, E., Pernsteiner, S., Wilcox, J. R., Tatlock, Z. & Grossman, D. (2018) Œuf: Minimizing the Coq extraction TCB. In CPP 2018.

Necula, G. C. (1997) Proof-carrying code. In Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'97. New York, NY, USA: Association for Computing Machinery, pp. 106–119. Available at: https://doi.org/10.1145/263699.263712

Nielsen, J. B. & Spitters, B. (2019) Smart contract interactions in Coq. In FMBC'2019.

O'Connor, R. (2017) Simplicity: A new language for blockchains. In PLAS17.

Paraskevopoulou, Z., Li, J. M. & Appel, A. W. (2021) Compositional optimizations for certicoq. *Proc. ACM Program. Lang.* **5**(ICFP). Available at: https://doi.org/10.1145/3473591.

Paulin-Mohring, C. (1989) Extracting Fω's programs from proofs in the calculus of constructions. In Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages', POPL'89. New York, NY, USA: Association for Computing Machinery, pp. 89–104.

Pit-Claudel, C., Wang, P., Delaware, B., Gross, J. & Chlipala, A. (2020) Extensible extraction of efficient imperative programs with foreign functions, manually managed memory, and proofs. In *Automated Reasoning*, Peltier, N. & Sofronie-Stokkermans, V. (eds), pp. 119–137.

Prost, F. (1995) Marking techniques for extraction. Research Report LIP RR-1995-47, Laboratoire de l'informatique du parallélisme. Available at: https://hal-lara.archives-ouvertes.fr/hal-02102062.

Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A. & Chan, K. (2019) Safer smart contract programming with Scilla. In OOPSLA19.

Šinkarovs, A. & Cockx, J. (2021) Choosing is Losing: How to combine the benefits of shallow and deep embeddings through reflection.

Sozeau, M., Anand, A., Boulier, S., Cohen, C., Forster, Y., Kunze, F., Malecha, G., Tabareau, N. & Winterhalter, T. (2020) The metacoq project. *J. Autom. Reas.* **64**, pp. 947–999.

Sozeau, M., Boulier, S., Forster, Y., Tabareau, N. & Winterhalter, T. (2019) Coq Coq Correct! verification of type checking and erasure for Coq, in Coq. In POPL'2019.

Sozeau, M. & Mangin, C. (2019) Equations reloaded: High-level dependently-typed functional programming and proving in Coq. *Proc. ACM Program. Lang.* **3**(ICFP) **86**, pp. 1–29.

Tanaka, A. (2021) Coq to C translation with partial evaluation. In Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2021, pp. 14–31.

Timany, A. & Sozeau, M. (2017) Consistency of the predicative calculus of cumulative inductive constructions (pCuIC). CoRR abs/1710.03912. Available at: http://arxiv.org/abs/1710.03912

Watt, C. (2018) Mechanising and verifying the webAssembly specification. In Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, New York, NY, USA, pp. 53–65.

Weiss, A., Gierczak, O., Patterson, D., Matsakis, N. D. & Ahmed, A. (2019) Oxide: The Essence of Rust. arXiv e-prints.

A Extracted code for the `counter` contract in Liquidity

```
let[@inline] fst (p : 'a * 'b) : 'a = p.(0)
let[@inline] snd (p : 'a * 'b) : 'b = p.(1)
let[@inline] addInt (i : int) (j : int) = i + j
let[@inline] subInt (i : int) (j : int) = i - j
let[@inline] ltInt (i : int) (j : int) = i < j
type 'a sig_ = 'a
let exist_ a = a

type coq_msg = Coq_Inc of int | Coq_Dec of int
type coq_SimpleCallCtx = (timestamp * (address * (tez * tez)))
type storage = int
    type coq_sumbool = Coq_left | Coq_right

    let coq_my_bool_dec (b1 : bool) (b2 : bool) = (if b1 then fun x→
    if x then Coq_left else Coq_right else fun x→ if x then Coq_right else Coq_left) b2

    let coq_inc_counter (st : storage) (inc : ( (int) sig_)) =
      exist_ ((addInt st ((fun x→ x) inc)))

    let coq_dec_counter (st : storage) (dec : ( (int) sig_)) =
      exist_ ((subInt st ((fun x→ x) dec)))

    let coq_counter (msg : coq_msg) (st : storage) =
      match msg with
      | Coq_Inc i→
       (match coq_my_bool_dec (ltInt 0 i) true with
        | Coq_left → Some ([],
          ((fun x→ x) (coq_inc_counter st (exist_ (i)))))
        | Coq_right → None)
      | Coq_Dec i→
         (match coq_my_bool_dec (ltInt 0 i) true with
         | Coq_left → Some ([], ((fun x→ x) (coq_dec_counter st (exist_ (i)))))
         | Coq_right → None)

    let%init storage (setup : int) =
      let inner (ctx : coq_SimpleCallCtx) (setup : int) = let ctx' = ctx in
      Some setup in
      let ctx = (Current.time (),
        (Current.sender (), (Current.amount (),Current.balance ()))) in
      match (inner ctx setup) with
      | Some v→ v
      | None→ failwith ()

    let wrapper param (st : storage) =
      match coq_counter param st with
      | Some v→ v
      | None→ failwith ()

    let%entry main param st = wrapper param st
```

B Extracted code for the `counter` contract in CameLIGO

```
[@inline] let addInt (i : int) (j : int) = i + j
[@inline] let subInt (i : int) (j : int) = i   j
[@inline] let multInt (i : int) (j : int) = i * j
[@inline] let divInt (i : int) (j : int) = i / j
[@inline] let modInt (a : int)(b : int) : int = int (a mod b)
[@inline] let leInt (i : int) (j : int) = i ≤ j
[@inline] let ltInt (i : int) (j : int) = i < j
[@inline] let eqInt (i : int) (j : int) = i = j

(* ConCert's call context *)
type cctx = {
  ctx_from_ : address;
  ctx_contract_address_ : address;
  ctx_contract_balance_ : tez;
  ctx_amount_ : tez }

(* a call context instance with fields filled in with required data *)
let cctx_instance : cctx=
{ ctx_from_ = Tezos.sender;
  ctx_contract_address_ = Tezos.self_address;
  ctx_contract_balance_ = Tezos.balance;
  ctx_amount_ = Tezos.balance }

type chain = {
  chain_height     : nat;
  current_slot     : nat;
  finalized_height : nat }

let dummy_chain : chain = {
  chain_height     = Tezos.level;
  current_slot     = Tezos.level;
  finalized_height = Tezos.level }

type 'a sig_ = 'a
let exist_ (a : _a) : _a = a
let id_func (a : _a) : _a = a

type chain = {
        chain_height     : nat;
        current_slot     : nat;
        finalized_height : nat;
        account_balance  : address→ nat }
let dummy_chain : chain = {
        chain_height     = 0n;
        current_slot     = 0n;
        finalized_height = 0n;
        account_balance  = fun (a : address)→ 0n }

type coq_sumbool =  Coq_left | Coq_right

type storage = int

type coq_msg =  Coq_Inc of (int) | Coq_Dec of (int)

let coq_bool_dec (b1 : bool) (b2 : bool) = (if b1 then fun (x : bool)→
if x then Coq_left else Coq_right else fun (x : bool)→ if x then Coq_right else Coq_left) b2

let coq_Transaction_none  = ([]: (operation) list)

let coq_inc_counter (st : storage) (inc :  (int) sig_) = exist_ ((addInt st (id_func inc)))
let coq_dec_counter (st : storage) (dec :  (int) sig_) = exist_ ((subInt st (id_func dec)))

let coq_counter (msg : coq_msg) (st : storage) = match msg with
   Coq_Inc (i)→
   (match coq_bool_dec true (ltInt 0 i) with
      Coq_left  → (Some ( (coq_Transaction_none, (id_func (coq_inc_counter st (exist_ (i)))))))
    | Coq_right  → (None: ((operation list * storage)) option))
 | Coq_Dec (i)→
   (match coq_bool_dec true (ltInt 0 i) with
      Coq_left  → (Some ( (coq_Transaction_none, (id_func (coq_dec_counter st (exist_ (i)))))))
    | Coq_right→ (None: ((operation list * storage)) option))

let coq_counter_wrapper (c : chain) (ctx : cctx) (s : storage) (m :
(coq_msg) option) = let c_ = c in
  let ctx_ = ctx in
  match m with
   Some (m0)→ (coq_counter m0 s)
 | None  → (None: ((operation list * storage)) option)

let init (setup : int) : storage =
  let inner (ctx : cctx) (setup : int) : (storage) option =
    let ctx_ = ctx in
    Some (setup) in
  let ctx = cctx_instance in
  match (inner ctx setup) with
```

```
    Some v→ v
  | None→ (failwith (""): storage)

type init_args_ty = int
let init_wrapper (args : init_args_ty) = init args

type return = (operation) list * (storage option)
type parameter_wrapper = Init of init_args_ty | Call of coq_msg option

let wrapper (param, st : parameter_wrapper * (storage) option) : return =
  match param with
    Init init_args→ (([]: operation list), Some (init init_args))
  | Call p→ (
    match st with
     Some st→ (match (coq_counter_wrapper dummy_chain cctx_instance  st p) with
                  Some v→ (v.0, Some v.1)
                | None→ (failwith ("") : return))
   | None→ (failwith ("cannot call this endpoint before Init has been called"): return))
let main (action, st : parameter_wrapper * storage option) : return = wrapper (action, st)
```