Extracting Smart Contracts Tested and Verified in Coq

Danil Annenkov
daan@cs.au.dk
Concordium Blockchain Research Center, Aarhus University, Denmark

Mikkel Milo
m.milo@cs.au.dk
Concordium Blockchain Research Center, Aarhus University, Denmark

Jakob Botsch Nielsen
botsch@cs.au.dk
Concordium Blockchain Research Center, Aarhus University, Denmark

Bas Spitters
spitters@cs.au.dk
Concordium Blockchain Research Center, Aarhus University, Denmark

Abstract

We implement extraction of Coq programs to functional languages based on MetaCoq's certified erasure. As part of this, we implement an optimisation pass to remove unused arguments of functions and constructors, making integration with the (often polymorphic) extracted code easier. This pass is proven correct over a weak call-by-value operational semantics that fits well with many functional languages. We argue that our extraction pipeline applies generally to these functional languages with a small trusted computing base of only MetaCoq and the pretty-printers into these languages. We apply this to two functional smart contract languages, Liquidity and Midlang, which are respectively similar to Ocaml and Elm. This is done in the context of the ConCert framework that allows proving properties of functional smart contracts. The extraction is exemplified by several programs, including a smart contract that uses dependent types, showing that we are able to handle non-trivial programs.

We also contribute a verified boardroom voting smart contract featuring maximum voter privacy such that each vote is kept private except under collusion of all other parties. With ConCert's executable specification our contracts are furthermore fully testable from within Coq. This enables us to integrate property-based testing into ConCert using QuickChick. We show that this is successful by testing complex contracts such as the congress contract (the essence of TheDAO), an escrow contract, an implementation of a Decentralized Finance (DeFi) contract which includes a custom token standard (Tezos FA2), and more.

In total, this gives us a way to write dependent programs in Coq, test them semi-automatically, verify them, and then extract them to functional smart contract languages, while retaining a small TCB.

ACM Reference Format:
Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. 2021. Extracting Smart Contracts Tested and Verified in Coq. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21), January 18–19, 2021, Virtual, Denmark*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3437992.3439934

1 Introduction

Smart contracts are programs running on top of a blockchain. They often control large amounts of cryptocurrency 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, Parity's multi-signature wallet). Therefore, smart contract verification is crucially important. Functional smart contract languages are becoming increasingly popular: e.g. Simplicity [35], Liquidity [8], Plutus [10], Scilla [39] and Midlang[1]. A contract in such a language is just a 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 variants of System F allowing the type checker to catch many errors. For errors that are not caught by the type checker, we show how to use a proof assistant, in particular Coq.

Formal verification is a complex and time-consuming activity, and much time may be wasted attempting to prove false statements (e.g. if the implementation is incorrect). Property-based testing is an automated testing technique with high bug-discovering capability compared to e.g. unit testing. It can provide a preliminary, cost-efficient approach to discover implementation bugs in the contract or mistakes in the statement to be proven.

Once properties of contracts are tested and proved correct, one would like to execute them on blockchains. One way of achieving this is to *extract* the executable code from the formalised development. Various verified developments rely on the extraction feature of proof assistants extensively [12, 13, 17, 24, 29]. However, currently, the standard extraction feature in proof assistants supports conventional functional languages (Haskell, OCaml, Standard ML, Scheme, etc.) by using unsafe features such as type casts, if required. However, this is not possible in many smart contracts languages.

Contributions. We build on the ConCert framework [3] for embedding smart contracts in Coq and the execution

[1] https://developers.concordium.com/midlang

model introduced in [34]. We summarise the contributions of this work as the following.

- We provide a general framework for extraction from Coq to a typed functional language (Section 3). The framework is based on certified erasure [41], which we extend with an erasure procedure for types and inductive definitions. Moreover, we implement and prove correct an optimisation procedure that removes unused arguments and allows therefore for optimising away some computationally irrelevant bits left after erasure. We develop the pretty-printers for the extracted code into two smart contract languages: Liquidity (Tezos and Dune networks) and Midlang (Concordium network). Since Midlang is a fork of the Elm functional language [16], our extraction also supports Elm as a target language.
- We integrate contract verification and testing using QuickChick, a property-based testing framework for Coq, by testing properties on generated execution traces. We show that this would have allowed us to detect several high profile exploits automatically.
- We provide case studies of smart contracts in ConCert by testing and proving properties of an escrow contract and an anonymous voting contract based on the Open Vote Network protocol (Sections 4 and 5). We apply our extraction functionality to study the applicability to the developed contracts.

Our work is the first development applying the property-based testing techniques to smart contract *execution traces* (as opposed testing single-step executions), and extracting the verified programs to smart contract languages.

2 The ConCert framework

The present work builds on and extends the ConCert smart contract certification framework [3]. The ConCert framework features embedding of smart contracts into Coq along with the proof of soundness of the embedding using the MetaCoq project [40]. The embedded contracts are available in the deep embedding (as ASTs) and in the shallow embedding (as Coq functions). Having smart contracts as Coq functions facilitates the reasoning about their functional correctness properties. Moreover, ConCert features an execution model first introduced in [34]. The execution model allows for reasoning on contract execution traces which makes it possible to state and prove temporal properties of interacting smart contracts. The previous work on ConCert [3] mainly concerns with the following use-case: take a smart contract, *embed* it into Coq and verify its properties. This work explores 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.

3 Extraction

The Coq proof assistant features a possibility of extracting the executable content of Gallina terms into OCaml, Haskell and Scheme [31]. This functionality thus enables proving properties of functional programs in Coq and then automatically producing code in one of the supported languages that could be integrated with existing developments or used as a stand-alone program. The extraction procedure is non-trivial since Gallina is a dependently-typed functional language. Recent projects such as MetaCoq [41] and CertiCoq [1] provide formal guarantees that the extraction procedure is correct, but do not support extraction to smart contract languages. The general idea of extraction is to find and mark all parts of a program that do not contribute to computation. That is, types and propositions in terms are replaced with □ (a box). Formally, it is expressed as a translation from CIC (Calculus of Inductive Construction — the underlying calculus of Coq) to $\lambda\square$ [31, 41]. In the present work, by CIC we mean polymorphic, cumulative calculus of inductive constructions (PCUIC), as presented by the authors of [41]. $\lambda\square$ is an untyped version of CIC with an additional constant □. One of the important results of [31] is that the computational properties of the erased terms are preserved. That assumes that the extracted code is untyped, while integration with the existing functional languages requires to recover the typing. In [31] this problem is solved by designing an extraction procedure for types and then using the modified type inference algorithm (based on the algorithm $\mathcal{M}$ [28]) 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 step a bit outside the OCaml type system (even without using dependent types), the extraction will 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, and we know that the term was well-typed before extraction.

Since the extraction implementation becomes part of the trusted computing base (TCB), one would like to have high correctness guarantees for this code. This can be achieved by mechanically verifying the extraction procedure in Coq itself. An important step in this direction was made by the

MetaCoq project [40], which includes *certified erasure* [41] that specifies an erasure procedure as a translation to $\lambda\Box$ in Coq and proves that the evaluations of original and erased terms agree.

3.1 Extraction to functional smart contract languages

We target functional smart contract languages, which often pose more challenges than the conventional targets for extraction.[2] We have identified the following restrictions.

1. Most of the smart contract languages[3] do not offer a possibility to insert type coercions forcing the type checking to succeed.
2. The operational semantics of $\lambda\Box$ has the following rule (see Section 4.1 in [41]): $\Sigma \vdash (\Box\ t) \triangleright t$, where $- \vdash - \triangleright -$ is a big-step evaluation relation for $\lambda\Box$. This rule can be implemented in OCaml using the unsafe features, which are, again, not available in most of the smart contract languages. In lazy languages this situation never occurs (see Section 2.6.3 in [31]), but, most languages for smart contracts 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 the tail recursion on a single argument.[4] Instead, these languages offer built-in lists and finite maps. Scilla exposes only recursors for lists instead of allowing to write recursive functions explicitly.

Regardless of our design choices, the soundness of the extraction (given that terms evaluate in the same way before and after extraction) will not be affected. In the worst case, the extracted term will be rejected by the type checker of a target language. We consider the formalisation of typing in target languages out of scope for this project. It would require formalising target languages' type systems, including a type inference algorithm (possibly algorithm $\mathcal{M}$ [28], if we follow the work in [31]). The type systems of the languages we target are not precisely specified and are largely in flux. Therefore, we do our best to extract as many programs that pass type checking as possible or fail at the extraction stage, due to incompatibilities with the "generic" type system usually found in functional languages (we take prenex-polymorphic System F for that purpose).

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. 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 target language's type checker. The third restriction can be addressed by mapping Coq's data types (lists, finite maps) to the corresponding primitives in a target language.

We extend the work on certified erasure [41] and develop an approach that uses a minimal amount of unverified code that can affect the soundness of the certified erasure. Our approach adds an erasure procedure for types, simple verified optimisations of the extracted code and pretty-printers for target smart contract languages.

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

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

produces the $\lambda\Box$ code:

```
fun xs ⇒ Coq.Lists.List.fold_left □ □ Coq.Init.Nat.add xs 0
```

Where the $\Box$ symbol corresponds to computationally irrelevant parts. The first two arguments to the erased versions of `fold_left` are boxes, since `fold_left` in Coq has two implicit arguments. They become visible if we switch on printing of implicit arguments:

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

In this situation we have at least two choices: remove the boxes by some optimisation procedure, or leave the boxes and extract `fold_left` in such a way that the first two arguments belong to some dummy data type.[5] The latter choice cannot be made for some smart contract languages due to restrictions, therefore, we have to remap `fold_left` and other functions on lists to the corresponding primitive functions. 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
```

[2] Our implementation of the extraction procedure is available in the `extraction` subfolder of the submitted artifact.

[3] At least, Simplicity, Liquidity, CameLIGO (and other LIGO languages), Love, Scilla, Sophia, Midlang.

[4] Some languages do not have this restriction, e.g. Midlang and Love.

[5] There are two more rules in the semantics of $\lambda\Box$ that do not quite fit into the evaluation model of smart contract languages: pattern-matching on a box argument and having a box as an argument to a fixpoint. The matching on $\Box$ occurs when eliminating from logical inductive types with no constructors (e.g. `False`) or from singleton types (e.g. equality type). A special rule for fixpoints is needed because of logical argument to fixpoints used by the accessibility predicate. We address the `False` case in an ad hoc way at the end of Section 3.3. We believe that it is possible to address other cases similarly to the previous works on extraction (Section 4 [30] and Section 2.6 in [31]), apart from the implementation of $\Box$ as an argument consuming function, due to the absence of unsafe features.

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`.

By choosing to implement the optimisation procedure we achieve two goals: remove redundant computations and make the remapping easier. Removing the redundant computations is beneficial for smart contract languages, since it decreases the cost of a computation 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. It is important to emphasise that we can pretty-print code produced by the certified erasure procedure directly. Moreover, it is important to separate these two aspects of extraction: erasure (given by the translation CIC $\longrightarrow \lambda\Box$) and optimisation of $\lambda\Box$ terms to remove unnecessary arguments. The optimisations we propose are simple, 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) optimize the environment in a consistent way; (iv) pretty-print the result in the target language syntax.

Erasure for types. Let us discuss our first extension to the certified erasure presented in [41], namely an *erasure procedure for types.* It is a crucial part for extracting to a *typed* target language. Currently, the verified erasure of MetaCoq does not provide such a procedure. In languages like Liquidity, the type inference for unannotated terms is undecidable due to the overloading of some primitive operations (e.g. arithmetic operations for primitive numeric types). That means that we have to provide some type annotations for the definition in order to resolve ambiguities. Moreover, it is essential to obtain types of a target language in order to extract inductive types. Our implementation of the erasure procedure for types is inspired by [31]. The outline of the procedure is given in Figure 1. We have chosen a semi-formal presentation in order to guide the reader through the actual implementation and avoid cluttering with technicalities of Coq. Additionally, we use colors to distinguish between the CIC terms and the target erased types.

The $\mathcal{E}^T$ function takes four parameters: a context `Ctx` represented as a list of assumptions, an erasure context `ECtx` represented as a sized list (vector) that follows the structure of `Ctx` and contains a translated type variable `TV`, information about an inductive type `Ind`, or a marker for items in `Ctx` that do not fit into the previous categories `Other`. The last two parameters represent terms of CIC corresponding to types and a list of names for type variables used later for printing and for identifying non-prenex types. We do not provide syntax and semantics of CIC, for more information we refer the reader to Section 2 of [41]. The function has a monadic type `result (list name × box_type)`, which is essentially an extended error monad. We use the standard do-notation to chain monadic computations. The result of the computation is a tuple consisting of a list of type variables and a `box_type`, the grammar for which is the the following:

$$\tau\ \sigma ::= \quad \bar{i} \mid I \mid C \mid (\tau\ \sigma) \mid \tau \longrightarrow \sigma \mid \Box \mid \mathbb{T}$$

Here 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 $\Box$ ("logical" types) and $\mathbb{T}$ (types that are not representable in the target language). In many cases both $\Box$ and $\mathbb{T}$ can be removed from the extracted code by optimisations, although $\mathbb{T}$ might require type coercions in the target language.

The functions $\mathcal{E}^T$ and $\mathcal{E}^T_{app}$ are defined by mutual recursion. The `decompose_app` function returns the head of an application and a (possibly empty) list of arguments. We use the notations $|xs|$ to denote the length of xs. In our implementation, we extensively use dependently-typed programming, so the actual type signature of functions in Figure 1 contains also proofs that terms are well-typed. The termination argument is given by a well-founded relation, since the erasure starts out with $\beta\iota\zeta$-reduction using the $\texttt{red}_{\beta\iota\zeta}$ function and then later recurses on subterms of this. Here β is reduction of applied λ-abstractions, ι is reduction of `match` on constructors, and ζ is reduction of the `let` construct. The $\texttt{red}_{\beta\iota\zeta}$ function reduces until the head cannot be $\beta\iota\zeta$-reduced anymore and then stops; it does not recurse on subterms. This reduction function is defined in MetaCoq also by using well-founded recursion. Due to the well-founded recursion we write $\mathcal{E}^T$ as a single function in our formalization by inlining the definition of $\mathcal{E}^T_{app}$; this makes the well-foundedness argument easier. We extensively use the `Equations` Coq plugin [42] in our development to help managing the proof obligation related to well-typed terms and recursion.

An important device used to determine erasable types (the ones we turn into a special target type $\Box$) is the function *flag_of_type* : `Ctx → term → type_flag`, where the return type `type_flag` is defined as a record with three projections: `is_logical`, `is_arity` and `is_sort`.[6] A term

[6]In our implementation, `is_logical` carries a boolean, while `is_arity` and `is_sort` carry proofs or disproofs of convertibility to an arity or sort, respectively.

$$
\begin{aligned}
&\mathcal{E}^T : \mathtt{Ctx} \to \mathtt{ECtx} \to \mathtt{term} \to \mathtt{list\ name}\\
&\quad \to \mathtt{result}\ (\mathtt{list\ name} \times \mathtt{box\_type})\\
&\mathcal{E}^T\ \Gamma\ \Gamma_e\ t\ vs\ := \mathtt{let}\ t' := \mathtt{red}_{\beta\iota\zeta}\ \Gamma\ t\ \mathtt{in}\\
&\quad \mathit{flag} \leftarrow \mathit{flag\_of\_type}\ \Gamma\ t';\\
&\quad \mathtt{if}\ (\mathtt{is\_logical}\ \mathit{flag})\ \mathtt{then\ Ok}\ \square\ \mathtt{else}\\
&\quad \mathtt{match}\ t'\ \mathtt{with}\\
&\quad \mid \bar{i} \Rightarrow Ok(vs, \mathcal{E}^T_{var}\ \Gamma_e\ i)\\
&\quad \mid \mathtt{Type} \Rightarrow \mathtt{Ok}\ \square\\
&\quad \mid \mathtt{forall}\ a : A, B \Rightarrow\\
&\qquad \mathit{flag} \leftarrow \mathit{flag\_of\_type}\ \Gamma\ A;\\
&\qquad \mathtt{if}\ (\mathtt{is\_logical}\ \mathit{flag})\ \mathtt{then}\\
&\qquad\quad (vs_\tau, \tau) \leftarrow \mathcal{E}^T\ (A :: \Gamma)\ (\mathtt{Other} :: \Gamma_e)\ B\ vs\ ;\\
&\qquad\quad \mathtt{Ok}\ (vs_\tau, \square \longrightarrow \tau)\\
&\qquad \mathtt{else\ if}\ \mathit{not}(\mathtt{is\_arity}\ \mathit{flag})\ \mathtt{then}\\
&\qquad\quad (vs_\sigma, \sigma) \leftarrow \mathcal{E}^T\ \Gamma\ \Gamma_e\ A\ vs\ ;\\
&\qquad\quad \mathtt{if}\ (|vs| < |vs_\sigma|)\ \mathtt{then}\ \mathtt{NotPrenex}\\
&\qquad\quad \mathtt{else}\ (vs_\tau, \tau) \leftarrow \mathcal{E}^T\ (A :: \Gamma)\ (\mathtt{Other} :: \Gamma_e)\ B\ vs\ ;\\
&\qquad\qquad \mathtt{Ok}\ (vs_\tau, \sigma \longrightarrow \tau)\\
&\qquad \mathtt{else\ if}\ (\mathtt{is\_sort}\ \mathit{flag})\ \mathtt{then}\\
&\qquad\qquad (vs_\tau, \tau) \leftarrow \mathcal{E}^T\ (A :: \Gamma)\ (\mathtt{TV}|vs| :: \Gamma_e)\ B\ (vs +\!\!+ [a])\ ;\\
&\qquad\qquad \mathtt{Ok}\ (vs_\tau, \square \longrightarrow \tau)\\
&\qquad\qquad \mathtt{else}\ \mathtt{NotPrenex}\\
&\qquad \mid (u\ v) \Rightarrow \mathtt{let}(hd, args) := \mathtt{decompose\_app}\ (u\ v)\ \mathtt{in}\\
&\qquad\quad \sigma \leftarrow \mathcal{E}^T_{head}\ \Gamma_e\ hd;\\
&\qquad\quad \mathcal{E}^T_{app}\ \Gamma_e\ args\ vs\ \sigma\\
&\qquad \mid \mathtt{C} \Rightarrow \mathtt{Ok}\ (vs, \mathtt{C}) \quad \mid \mathtt{I} \Rightarrow \mathtt{Ok}\ (vs, \mathtt{I})\\
&\quad \mathtt{end}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}^T_{app} : \mathtt{ECtx} \to \mathtt{list\ term} \to \mathtt{box\_type} \to \mathtt{list\ name}\\
&\quad \to \mathtt{result}\ (\mathtt{list\ name} \times \mathtt{box\_type})\\
&\mathcal{E}^T_{app}\ \Gamma_e\ args\ vs\ \sigma\ := \mathtt{match}\ args\ \mathtt{with}\\
&\quad \mid [] \Rightarrow \mathtt{Ok}(vs, \sigma)\\
&\quad \mid a :: agrs' \Rightarrow\\
&\qquad A \leftarrow \mathit{type\_of}\ a;\\
&\qquad \mathit{flag} \leftarrow \mathit{flag\_of\_type}\ \Gamma\ A;\\
&\qquad \tau \leftarrow \mathtt{if}\ (\mathtt{is\_logical}\ \mathit{flag})\ \mathtt{then\ Ok}\ \square\\
&\qquad\quad \mathtt{else\ if}\ (\mathtt{is\_sort}\ \mathit{flag})\ \mathtt{then}\\
&\qquad\qquad (vs_\tau, \tau) \leftarrow \mathcal{E}^T\ \Gamma\ \Gamma_e\ a\ vs\ ;\\
&\qquad\qquad \mathtt{if}\ |vs| < |vs_\tau|\ \mathtt{then}\ \mathtt{NotPrenex}\\
&\qquad\qquad \mathtt{else\ Ok}\ \tau\\
&\qquad\quad \mathtt{else\ Ok}\ \mathbb{T};\\
&\qquad \mathcal{E}^T_{app}\ \Gamma_e\ args'\ vs\ (\sigma\ \tau)\\
&\mathcal{E}^T_{head} : \mathtt{ECtx} \to \mathtt{term} \to \mathtt{result\ box\_type}\\
&\mathcal{E}^T_{head} \Gamma_e\, hd := \mathtt{match}\ hd\ \mathtt{with}\\
&\quad \mid \bar{i} \Rightarrow \mathtt{match}\ \Gamma_e(i)\ \mathtt{with}\\
&\qquad \mid \mathtt{Ind}\ I \Rightarrow I\\
&\qquad \mid \_ \Rightarrow \mathtt{Error}\\
&\qquad \mathtt{end}\\
&\quad \mid C \Rightarrow \mathtt{Ok}\ C \quad \mid I \Rightarrow \mathtt{Ok}\ I \quad \mid \_ \Rightarrow \mathtt{Error}\\
&\quad \mathtt{end}\\
&\mathcal{E}^T_{var} : \mathtt{ECtx} \to \mathbb{N} \to \mathit{box\_type}\\
&\mathcal{E}^T_{var}\ \Gamma_e\ i := \mathtt{match}\ \Gamma_e(i)\ \mathtt{with}\\
&\qquad \mid \mathtt{TV}\ i \Rightarrow \bar{i} \quad \mid \mathtt{Other} \Rightarrow \square \quad \mid \mathtt{Ind}\ I \Rightarrow I\\
&\qquad \mathtt{end}
\end{aligned}
$$

Figure 1. Erasure from CIC types to `box_type`

is logical when it is a proposition or being sufficiently applied it gives a proposition: $\mathtt{T} : \mathtt{Prop}, T\ a_0\ \ldots\ a_n\ :\ \mathtt{Prop}$. A term is an arity if it is a (possibly nullary) type scheme: *Type* or $Type \to \ldots \to \mathtt{Type}$ (the same holds for `Type` replaced with `Prop`). Finally `is_sort` tells us if a given term is a sort, i.e. `Prop` or `Type`. For example, `Type` is an arity and a sort, but not logical. $\mathtt{Type} \to \mathtt{Prop}$ is logical, an arity, but not a sort. $\mathtt{forall}\ A : \mathtt{Type},\ A \to A$ is neither of the three. Using erasure for types, we implement an erasure procedure for inductive definitions.

Optimisations. Our second extension of the certified 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: elimination of dead arguments. 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: $(\lambda A\ x.\ t)\ \square \sim \lambda x.\ t$. From erasure, we know that the variable A does not occur free in t[7]. Having in mind this equivalence, we implement in Coq a function with the following signature:

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

The first two parameters are lookup tables for inductive definitions and for constants defining which arguments of constructors and constants to remove. 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:

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

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

This function deargs the body using $dearg$ and additionally removes lambda abstractions according to the masks.

To generate the masks we implement an analysis procedure that finds dead parameters of constants and dead constructor arguments. For parameters of constants we check syntactically that 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. As we noted above erased arguments will be unused and therefore this procedure gives us a safe way of removing many redundant boxes (cf. Section 4.3 in [31]).

For definitions of inductive types, we define the following function:

$$\begin{aligned} dearg\_mib\ &: \texttt{mib\_masks} \to \mathrm{N} \to \texttt{one\_inductive\_body} \\ &\to \texttt{one\_inductive\_body} \end{aligned}$$

This function 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:

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

It is important to supply the same masks for adjusting top-level definitions (of constants and inductive types) and the corresponding application sites to ensure consistency in removing unused arguments.

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 validity of all the masks, meaning that all removed arguments of constants and constructors should be unused in the program. Finally, we assume that the program is η-expanded according to all the masks; that is, all occurrences of constructors and constants are applied enough. We implement a *certifying* procedure that performs η-expansion and generates proofs that the expanded terms are equal to the original ones. Since η-conversion is part of the Coq's conversion, the proofs are essentially just applications of the constructor `eq_refl`.[8]

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

Theorem 1 (Soundness of dearging)**.** *Let Σ be a closed erased environment and t a closed $\lambda\Box$-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.

[8]See `extraction/examples/CounterDepCertifiedExtraction.v` for an example of using the technique in the extraction pipeline.

Here $- \vdash - \triangleright -$ denotes the big-step call-by-value evaluation relation of $\lambda\Box$ terms[9] and values are give 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 $\lambda\Box$ preserves dynamic behaviour as well gives us guarantees that the terms that evaluate in CIC will be evaluated to related results in $\lambda\Box$ after optimisations.

Theorem 1 is a relatively low level statement talking about the dearging optimisation that is used by our extraction. The extraction pipeline itself is more complicated and works as outined at the end of subsection 3.1: it is provided a list of definitions to extract in a well-typed environment and recursively erases these and their dependencies. 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 it has produced an environment the environment is analyzed 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 their theorem that is applicable for our case. Combining this with Theorem 1 allows us to obtain a statement about the full extraction pipeline (excluding pretty-printing).

Theorem 2 (Soundness of extraction)**.** *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 of C from Σ. Then, for any unerasable constructor $Ctor$, if*

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

it holds that

$$\Sigma' \vdash C \triangleright Ctor$$

Here $- \vdash_p - \triangleright -$ denotes the big-step call-by-value evaluation relation for CIC terms. Informally, the above statement can be specialized to say that any program computing a boolean value will compute the same boolean value after extraction.

Of course, one still has to keep in mind the pretty-printing step of the extracted environment and the discrepancies of $\lambda\Box$ and the target language's semantics as we outlined in Section 3.1.

While dearging subsumes deboxing we cannot guarantee that our optimisation removes all boxes even for constants applied to all logical arguments due to cumulativity.[10] E.g. for `@inl Prop Prop True : sum Prop Prop` it is tempting to optimise

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

[10]Here by cumulativity we mean subtyping for universes, e.g. $\mathsf{A} : \mathsf{Type}_i$ is also $\mathsf{A} : \mathsf{Type}_{i+1}$ for some universe level i. Therefore, if a constructor or a constant takes an argument `A : Type`, we can pass `Prop`, since it is a type at the lowest level of the universe hierarchy.

```
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.
```

Figure 2. The `counter` contract

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 computations.

In addition to the deboxing of terms, we implement a deboxing procedure for removing boxes from types (`box_type`) and inductive definitions. This step does not affect the dynamic semantics, but mistakes in it might mean that the code does not type check in the target language. Having these, we pretty-print the optimised code to a functional smart contract language. We exploit the fact that $\lambda\square$ is similar to a subset of a generic functional language. Therefore, a pretty-printer can take an extracted and deboxed term and produce a program in Liquidity, Midlang or another language that is sufficiently close to $\lambda\square$. Currently, we support most of the constructs of $\lambda\square$ in our code generators, apart from primitive projections and coinductive definitions. We discuss issues related to extraction to Liquidity in Section 3.2 and to Midlang 3.3.

The* `counter` *contract. As an example, let us consider a simple counter contract with the state being just an integer number and accepting increment and decrement messages (Figure 2). The main functionality is given by the two functions `inc_counter` and `dec_counter`. We use *refinement 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. Refinement 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 to lift the boolean less-then comparison function denoted as `<?` 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}`[11] that gives us access to the proof of `0 <? i`. We will use the example from Figure 2 in subsequent sections for showing how it can be extracted to concrete target languages.

3.2 Extracting to Liquidity

Liquidity is a functional smart contract language for the Tezos and Dune blockchains inspired by OCaml. It compiles to Michelson developed by Tezos — a stack-based functional language that runs directly on the blockchain. Compared to a conventional functional language, Liquidity has many restrictions. E.g. data types are limited to non-recursive inductive types, support for recursive definitions is limited to tail recursion on a single argument. That means that one is forced to use primitive container types to write programs. In turn, from the extraction point of view, that means that 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 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. Before extraction, we validate the code to identify if constructors are fully applied and if so, we pass the code to the pretty-printer that wraps the arguments into a tuple. We apply a similar procedure to recursive calls.

Another issue is related to the type inference in Liquidity. Due to the support of overloaded operations on numbers, type inference is undecidable and 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 primitives.

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

[11] We simplify `bool_dec` using `Eval compute in` ... in order to unfold the recursor `bool_rec` in the definition of `bool_dec`. The types of recursors are not in the prenex form, therefore they give an error during erasure. Eventually, we want to implement a separate pass for unfolding constants like `bool_rec`

```
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)
```

(a) Liquidity

```
type Sig a = Exist a
type Msg = Inc Int | Dec Int
type alias Storage = Int
type Sumbool = Left | Right

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

inc_counter : Storage → Sig Int → Sig Storage
inc_counter st inc = Exist (add st (proj1_sig inc))
...
counter : Msg → Storage → Option (Prod Transaction Storage)
counter msg st =
  case msg of
    Inc i → case my_bool_dec (lt 0 i) True of
            Left → Some (Pair Transaction.none (proj1_sig (inc_counter
                   st (Exist i))))
            Right → None
    Dec i → ...
```

(b) Midlang

Figure 3. Extracted code.

because the signature of the entry point in Liquidity is `params` → `storage` → (`operation list`)* `storage`, where `papams` 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 supports a side effect `failwith`, the entry-point function still can 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 extracted counter contract code is given in Figure 3a. We omit some wrapper code and the "prelude" definitions and leave the most relevant parts (see Appendix A for the full version). 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 does not support variant types with only a single constructor. Ideally, this class of transformations can be added as an optimisation for inductives 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 2, we successfully applied the developed extraction to several variants of the counter contract, to the crowdfunding contract described in [3] and to an interpreter for a simple expression language. The latter example shows the possibility of extracting certified interpreters for domain-specific languages such as Marlowe [27], CSL [21] and the CL language [2, 4] representing 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 programs. Moreover, in many cases, our optimisation procedure removes all the boxes resulting in cleaner code.

3.3 Extracting to Midlang and Elm

Midlang is a functional smart contract language for the Concordium blockchain. It is a fork of Elm [16] — a general-purpose functional language used for web development. Being close to Elm means that it is a fully-featured functional language that supports many usual functional programming idioms. Compared to Liquidity, Midlang is a better target for code extraction, since it does not have the limitations pointed out in Section 3.2.

We use the same example from Figure 2 to demonstrate extraction to Midlang (Figure 3b, see Appendix B for the full version). Similarly to Liquidity, extracted code does not contain logical parts (e.g. proofs of being positive). The `sig` type of Coq extracts to the type definition `Sig` with a single constructor being a simple wrapper around the value. In Midlang we do not have to "unwrap" the value from the `Exist` constructor in an ad-hoc way since single constructor data types are allowed, but one could still imagine this as an optimisation.

Extraction to Midlang poses some challenges which are inherited from Elm. For example, Midlang 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 Midlang is indentation sensitive that requires tracking of indentation levels. Various naming conventions apply to Midlang identifiers, e.g. function names start with a lower-case character, types and constructors — with an upper case character. Many convenient functions are missing in Coq's standard `String` module, so we had to implement required functionality for string manipulation ourselves.

We have tested the support for Midlang extraction on several examples including the contract from Figure 2 and the escrow contract described in Section 4. Both the counter and the escrow contracts were successfully extracted and compiled with the Midlang compiler. However, the escrow contract requires more infrastructure for mapping the ConCert blockchain formalisation definitions to the corresponding Midlang primitives. Since Midlang is a fork of Elm [16], code that does not use any blockchain specific primitives is also extractible to Elm. We tested the extracted code with Elm compiler by generating a simple test for each extracted function. We implemented several tests extracting functions on lists from Coq's standard library and functions using refinement types. The `safe_head` (a head of a non-empty list) example uses the elimination principle `False_rect`. We support this by an ad hoc remapping of `False_rect` to `false_rec _ = false_rec ()`. Since we know that the impossible case never happens, we can use this "infinite loop" function in its place. Moreover, we extracted the Ackermann function `ackermann : nat * nat → nat` defined using well-founded recursion which uses the lexicographic ordering on pairs. This shows that extraction and computation of definitions based on the accessibility predicate `Acc` is possible. Computation with `Acc` is studied in more detail in [42].

4 The Escrow contract

As an example of a nontrivial contract we can extract we describe in this section an *escrow* contract.[12] 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 [14].

Because goods are not delivered on chain there is no way for the contract to verify that the buyer has received the item. Instead, we incentivise 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 with the deployment) $2x$.
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 he has received the item. He 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 his money back after the deadline. Note that when the buyer has received the item, he can choose not to notify the smart contract that this has happened. In this case he 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 around 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)**.** 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 in π 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 4 (Escrow correctness)**.** *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.*

In Section 6.1 we describe how this property can also be tested automatically by using QuickChick.

Through extraction we can get versions of the verified Escrow in other languages. Theorem 4 relies on the execution *model* of the actual implementation of a transaction scheduler. Therefore, the execution model is part of the TCB. However, the model provides a good approximation of execution layers used for *functional* smart contracts. While we do not have a formal proof that Theorem 4 translates to the version produced by extraction, this still gives us high level of confidence given by the use of the certified erasure and optimisations.

[12] See `execution/examples/Escrow.v` in the submitted artifact.

5 The boardroom voting contract

Hao, Ryan and Zieliński developed the Open Vote Network protocol [20], 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 [32]. 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.[13] 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 [20].

The contract accepts messages given by the following 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 messages 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.

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 5 (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.*

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 are currently working on extracting it in a performant manner.

6 Property-based testing of smart contracts

With ConCert's executable specification our contracts are fully testable from within Coq.[14] This enables us to integrate property-based testing into ConCert using QuickChick [36]. It serves as a cost-effective, semi-automated approach to discover bugs and it increases reliability that the implementation is correct. Testing may be used either as a preliminary step to support formal verification or as a complementary approach whenever the properties become too involved to prove. Property-based testing is an automated, generative approach to software testing where test data is automatically generated and tested against some executable specification, and any failed test case is reported. As opposed to example-based testing, where the user manually constructs and executes a few test cases, property-based testing can cover a much larger input scope by generating thousands of "arbitrary" test data. Intuitively, this gives higher assurance in the

[13] See `execution/examples/BoardroomVoting.v` in the submitted artifact.

[14] The testing framework for smart contracts is available in the `execution/tests` subfolder of the submitted artifact.

```
Conjecture example_prop :
  forall (x y : nat), y = square x → sqrt y = x.
QuickChick example_prop.
(* Passed 10000 tests (0 discards) *)
```

Figure 4. Simple example usage of QuickChick.

validity of the test, and increases the potential of discovering bugs.

QuickChick is a property-based testing framework in Coq inspired by Haskell's Quick*Check* [11]. The user provides an executable specification (e.g. a decidable property) and input generators for the necessary data types. QuickChick will then test that all generated test cases pass, reporting any discovered counterexample. In many cases the input generators can be derived either partially or fully automatically, and QuickChick provides generators for common data types such as `nat`, `Z`, `bool`, and `list`. As a simple example, Figure 4 shows how to test an inverse property between `square` and `sqrt` using QuickChick. QuickChick can execute this test because it has a built-in generator for arbitrary `nat`s, and because equality on `nat` is decidable, and therefore the entire property is decidable. Internally, QuickChick converts `example_prop` to the executable term `y = square x ==> sqrt y = x` where `==>` is an executable variant of implication that discards a test whenever the pre-condition is false[15].

Since the testing is intended to support verification, we should be able to test the same properties as those we wish to prove (assuming the property is decidable). These properties are usually stated in terms of blockchain execution traces. This poses the key question of how to generate "arbitrary" execution traces. An execution trace in ConCert is a sequence of blocks, each containing some number of `Action`s (which may be either transactions, contract calls, or contract deployment). Specifically, we must consider how to generate arbitrary contract calls for a given contract. Previous works on property-based testing of smart contracts such as Echidna [18] and Brownie[16] employ a fuzzing-like approach where payload data of contract calls are populated with entirely random data. The advantage of this approach is that it can be completely automated, however the generated data may not provide good enough coverage for contracts with endpoints having complex conditions for successful computation. In these cases, large proportions of the generated data will be discarded during testing, which leads to worse performance, lower test coverage, and worse expected bug discovering capability. The work [18] mitigates this by using a coverage-based, self-improving algorithm for test generation.

[15]In the example in Figure 4 QuickChick reported 0 discards. This is because QuickChick is able to automatically derive a generator satisfying the inductive predicate `x = y`. This is in general not always possible.

[16]Property-based testing framework for EVM:https://github.com/eth-brownie/brownie

```
Definition gEscrowMsg (e : Environment) (state : Escrow.State)
  : GOpt Action :=
  (* creates a call to the escrow contract with some msg *)
 let call caller amount msg := returnGenSome
    (build_act caller
     (act_call contract_addr amount (serialize m))) in
 (* pick one gen at random, backtrack if it fails *)
 backtrack [(1, if e.(account_balance) state.(buyer) <? 2
                then returnGen None
                else call state.(buyer) 2 commit_money) ;
            (1, call state.(buyer) 0 confirm_item_received) ;
            (1, addr ← elems [state.(seller); state.(buyer)] ;;
                call addr 0 withdraw) ].
```

Figure 5. Generator of "arbitrary" Escrow messages.

We make a trade-off to overcome this issue by sacrificing some automation and instead require the user to supply specialized generators for the message type of the contracts under test, rather than automatically deriving these generators. From this, the framework automatically derives a generator of provably valid execution traces which is used for subsequent tests. For example, if the user supplies a generator for the `Msg` type of the Escrow contract presented in Section 4, the generated traces will contain contract calls to the Escrow contract (assuming it is deployed on the test chain) using this generator.

Our testing framework supports three kinds of testable properties: (1) a testable notion of universal quantification (which is realized by just executing, and asserting success of, 10.000 test cases) on any executable property on the `ChainBuilder` type. This type represents a chain state along with a proof-relevant execution trace that led to this state. (2) A hoare-triple style pre- and post-condition assertions on the `receive` function of a given contract, and (3) reachability of chain states satisfying some decidable property.

1. `forall (c : ChainBuilder), P c`
2. `{pre}SomeContract.receive{post}`
3. `init_chain ⇝ (fun cs : ChainState ⇒ P cs)`

6.1 Testing the Escrow contract

The Escrow contract was described and proved correct in Section 4. The entire Coq proof, including auxiliary lemmas about the Escrow contract, is about 500 lines. Thus, it is a significant effort and in the presence of bugs in the implementation much time and effort could be wasted. We demonstrate how to use the testing framework as a preliminary step in formal verification to potentially discover any bugs. The correctness property of the Escrow was defined in Theorem 4. Supposing the Escrow is finished in some block, then calculating the net balance effect is a decidable proposition, and can easily be proven so by coercing it to a `bool`. In order to be able to test this property we first need to derive a generator for traces containing calls to a deployed Escrow contract. On Figure 5 we show such a generator.

Omitting some boilerplate code for the initial chain setup, we can now test the correctness property with:

```
QuickChick (forall c : ChainBuilder, escrow_correct_P c).
(* Passed 10000 tests (23743 discards) *)
```

If we manually insert a bug where the withdrawable amount is calculated differently, then QuickChick immediately reports a counterexample. Aside from the example shown here, we have tested various other contracts implemented in ConCert such as ERC-20 tokens, FA2 tokens, a congress contract similar to "The DAO", and a liquidity exchange protocol consisting of multiple interacting contracts. We used the testing framework to successfully discover well known contract vulnerabilities:

- The "DAO attack" where, at the time, $50 million worth of cryptocurrency was stolen due to a re-entrancy bug.[17]
- An exploit of the UniSwap exchange protocol on the Lendf.me platform where an attacker could perform a re-entrancy attack during token exchange to obtain arbitrary profit.[18] This attack was possible for any ERC-777 compliant tokens.
- A recent exploit the iToken contract where an attacker could mint arbitrary tokens for themselves by performing a self-transfer due to a bug in the implementation of the transfer function.[19]

This shows the effectiveness of our approach, not just for testing contracts in isolation, but also for testing multiple, interacting contracts.

7 Related work

Works related to the extraction part can be split in two categories. The most relevant related works are concerned with extraction to statically typed functional programming languages. Several proof assistants share this feature (Coq [30], Isabelle [7], Agda [26]) and allow targeting conventional functional languages such as Haskell, OCaml or Standard ML. However, extraction in Isabelle/HOL [7] is slightly different form the Coq and Agda, since the in 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. Clearly, the correctness of the extraction code is crucial for producing correct executable programs. This is addressed by several developments for Isabelle [19, 22]. The work [22] features verified compilation from Isabelle/HOL to CakeML [25]. It also implements meta-programming facilities for quoting Isabelle/HOL terms similar to MetaCoq. Moreover, the quoting procedure produces a proof that the quoted terms corresponds to the original ones. The current extraction implemented in the Coq proof assistant is not verified, however, the theoretical basis for it is well-developed by Letouzey [31]. The MetaCoq project [41] 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 more diverse field of functional smart contract languages.

Another category of related approaches focuses on execution of dependently typed languages. Although the techniques used in these approaches are similar, one does not need to fit the extracted code into the type system of a target language. The dependently-type programming language Idris uses erasure techniques for efficient execution [9]. The master's thesis [37] explores the applicability of dependent types to smart contract development and extend the Idris compiler with Ethereum Virtual Machine code generation. For the Coq proof assistant, the work [5] develops an approach for efficient convertibility testing of untyped terms acquired from fully typed CIC terms. The Œuf project [33] 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 inductives). In [38], the authors report on extraction of embedded into Gallina domain-specific languages into an imperative intermediate language which 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 [1]. 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 [29]. We hope that our work can be used as an optimisation step at the early stage of the CertiCoq compilation pipeline.

The boardroom voting is based on the Open Vote Network by Hao, Ryan and Zieliński [20]. In this paper there are paper proofs showing the computation of the tally correct. As part of proving the boardroom voting contract correct we have mechanized the required results from their paper.

An Ethereum version of the boardroom voting was developed by McCorry, Shahandashti and Hao [32]. However, the contract is not formally verified. Their version uses elliptic curves instead of finite fields to achieve the same security guarantees with much smaller key sizes and therefore more efficient computation. Our contract uses finite fields and is less efficient.

Previous work in testing of smart contracts have been done in Echidna [18], Brownie, and ContractFuzzer [23]. A common denominator for these works is that they choose a

[17] https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/

[18] https://forum.openzeppelin.com/t/exploiting-uniswap-from-reentrancy-to-actual-profit/1116

[19] https://bzx.network/blog/incident

fuzzing approach where transactions are generated at random. This leads to poor test coverage, and each work employs different automated methods to improve test coverage. Unlike our testing framework, which allows for testing global properties about entire execution traces, these works only support testing assertional properties about single steps of execution.

8 Conclusion and Future Work

We have presented several extensions to the ConCert smart contract certification framework: certified extraction, integration of the ConCert execution model with QuickChick and two verified smart contracts (Escrow and Boardroom Voting) used as case studies for the developed techniques. Currently, we support two target languages for smart contract extraction: Liquidity and Midlang. Since Midlang is a derivative of the Elm programming language our extraction also allows targeting Elm. Our extraction technique extends the certified erasure [41] and allows for targeting various functional smart contract languages. Our experience shows that the extraction allows to cover many interesting examples of smart contracts and in general is well-suited for Coq programs in a fragment of Gallina that corresponds to a generic polymorphic functional language extended with refinement types. We believe that with minor modifications of the Liquidity pretty-printer, we will be able to target the languages from the LIGO family by Tezos and other functional smart contract languages, which we leave as future work. We plan to finalise the extraction of the boardroom voting contract so it performs well in the practical setting. On way of achieving this would be to integrate it with extracted high-performance cryptographic primitives using the approach of FiatCrypto [6, 15]. Supporting optimisations such as removing singleton inductives (e.g. `Acc`) and inductives with no constructors (e.g. `False`) in a principled way is also among our future plans.

References

[1] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *CoqPL'2017*.

[2] Danil Annenkov and Martin Elsman. 2018. Certified Compilation of Financial Contracts. In *PPDP'2018*. 13 pages. https://doi.org/10.1145/3236950.3236955

[3] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. 2020. ConCert: A Smart Contract Certification Framework in Coq. In *CPP'2020*. https://doi.org/10.1145/3372885.3373829

[4] Patrick Bahr, Jost Berthold, and Martin Elsman. 2015. Certified Symbolic Management of Financial Multi-Party Contracts. *SIGPLAN Not.* (2015), 13 pages. https://doi.org/10.1145/2784731.2784747

[5] Bruno Barras and Benjamin Grégoire. 2005. On the Role of Type Decorations in the Calculus of Inductive Constructions. In *CSL*. https://doi.org/10.1007/11538363_12

[6] Bas Spitters Benjamin S. Hvass, Diego F. Aranha. 2020. High-assurance field inversion forpairing-friendly primes. Extended abstract. *FMBC'2020*.

[7] Stefan Berghofer and Tobias Nipkow. 2002. Executing Higher Order Logic. In *Types for Proofs and Programs*, Paul Callaghan, Zhaohui Luo, James McKinna, Robert Pollack, and Robert Pollack (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 24–40. https://doi.org/10.1007/3-540-45842-5_2

[8] Çagdas Bozman, Mohamed Iguernlala, Michael Laporte, Fabrice Le Fessant, and Alain Mebsout. 2018. Liquidity: OCaml pour la Blockchain. In *JFLA18*.

[9] Edwin Brady, Conor McBride, and James McKinna. 2004. Inductive Families Need Not Store Their Indices. In *Types for Proofs and Programs*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.). Springer Berlin Heidelberg, 115–129. https://doi.org/10.1007/978-3-540-24849-1_8

[10] James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. 2019. System F in Agda, for fun and profit. In *MPC'19*. https://doi.org/10.1007/978-3-030-33636-3_10

[11] Koen Claessen and John Hughes. 2011. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices* 46, 4 (2011), 53–64. https://doi.org/10.1145/351240.351266

[12] Luís Cruz-Filipe and Pierre Letouzey. 2006. A Large-Scale Experiment in Executing Extracted Programs. *Electron. Notes Theor. Comput. Sci.* (2006). https://doi.org/10.1016/j.entcs.2005.11.024

[13] Luís Cruz-Filipe and Bas Spitters. 2003. Program Extraction from Large Proof Developments. In *Theorem Proving in Higher Order Logics*. https://doi.org/10.1007/10930755_14

[14] Stefan Dziembowski, Lisa Eckey, and Sebastian Faust. 2018. FairSwap: How To Fairly Exchange Digital Goods. In *ACM Conference on Computer and Communications Security*. ACM, 967–984. https://doi.org/10.1145/3243734.3243857

[15] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2019.00005

[16] Richard Feldman. 2020. *Elm in Action*.

[17] Jean-Christophe Filliâtre and Pierre Letouzey. 2004. Functors for Proofs and Programs. In *Programming Languages and Systems*, David Schmidt (Ed.). Springer Berlin Heidelberg, 370–384. https://doi.org/10.1007/978-3-540-24725-8_26

[18] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 557–560. https://doi.org/10.1145/3395363.3404366

[19] Florian Haftmann and Tobias Nipkow. 2007. A code generator framework for Isabelle/HOL. In *Department of Computer Science, University of Kaiserslautern*.

[20] Feng Hao, Peter YA Ryan, and Piotr Zieliński. 2010. Anonymous voting by two-round public discussion. *IET Information Security* 4, 2 (2010). https://doi.org/10.1049/iet-ifs.2008.0127

[21] Fritz Henglein, Christian Kjær Larsen, and Agata Murawska. 2020. A Formally Verified Static Analysis Framework for Compositional Contracts. In *Financial Cryptography and Data Security (FC)*. https://doi.org/10.1007/978-3-030-54455-3_42

[22] Lars Hupel and Tobias Nipkow. 2018. A Verified Compiler from Isabelle/HOL to CakeML. In *Programming Languages and Systems*, Amal Ahmed (Ed.). 999–1026. https://doi.org/10.1007/978-3-319-89884-1_35

[23] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. *CoRR* abs/1807.03932 (2018). arXiv:1807.03932 http://arxiv.org/abs/1807.03932

[24] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM T. Comput. Syst.* 32, 1 (2014), 2:1–2:70. https://doi.org/10.1145/2560537

[25] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. ACM, 179–191. https://doi.org/10.1145/2535838.2535841

[26] W. H. Kusee. 2017. Compiling Agda to Haskell with fewer coercions. Master's thesis.

[27] Pablo Lamela Seijas and Simon Thompson. 2018. Marlowe: Financial Contracts on Blockchain. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, Tiziana Margaria and Bernhard Steffen (Eds.). https://doi.org/10.1007/978-3-030-03427-6_27

[28] Oukseh Lee and Kwangkeun Yi. 1998. Proofs about a Folklore Let-Polymorphic Type Inference Algorithm. *ACM Trans. Program. Lang. Syst.* (1998). https://doi.org/10.1145/291891.291892

[29] Xavier Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*. 42–54. https://doi.org/10.1145/1111037.1111042

[30] Pierre Letouzey. 2003. A New Extraction for Coq. In *Types for Proofs and Programs*. 200–219. https://doi.org/10.1007/3-540-39185-1_12

[31] Pierre Letouzey. 2004. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. Ph.D. Dissertation. Université Paris-Sud.

[32] Patrick McCorry, Siamak F Shahandashti, and Feng Hao. 2017. A smart contract for boardroom voting with maximum voter privacy. In *FC 2017*. https://doi.org/10.1007/978-3-319-70972-7_20

[33] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. 2018. Œuf: Minimizing the Coq Extraction TCB. In *CPP 2018*. https://doi.org/10.1145/3167089

[34] Jakob Botsch Nielsen and Bas Spitters. 2019. Smart Contract Interactions in Coq. In *FMBC'2019*. https://doi.org/10.1007/978-3-030-54994-7_29

[35] Russell O'Connor. 2017. Simplicity: A New Language for Blockchains *(PLAS17)*. https://doi.org/10.1145/3139337.3139340

[36] Zoe Paraskevopoulou, Catalin Hritcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015. Foundational Property-Based Testing. In *6th International Conference on Interactive Theorem Proving (ITP) (Lecture Notes in Computer Science, Vol. 9236)*, Christian Urban and Xingyuan Zhang (Eds.). Springer, 325–343. https://doi.org/10.1007/978-3-319-22102-1_22

[37] Jack Pettersson and Robert Edström. 2016. Safer smart contracts through type-driven development. Master's thesis.

[38] Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. 2020. Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs. In *Automated Reasoning*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). 119–137. https://doi.org/10.1007/978-3-030-51054-1_7

[39] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan. 2019. Safer Smart Contract Programming with Scilla. In *OOPSLA19*. https://doi.org/10.5281/zenodo.3368504

[40] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *Journal of Automated Reasoning* (18 Feb 2020). https://doi.org/10.1007/s10817-019-09540-0

[41] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2019. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. In *POPL'2019*. https://doi.org/10.1145/3371076

[42] Matthieu Sozeau and Cyprien Mangin. 2019. Equations Reloaded: High-Level Dependently-Typed Functional Programming and Proving in Coq. *Proc. ACM Program. Lang.* 3, ICFP (2019). https://doi.org/10.1145/3341690

Appendices

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 Midlang

```
import Basics exposing (..)
import Blockchain exposing (..)
import Bool exposing (..)
import Int exposing (..)
import Maybe exposing (..)
import Order exposing (..)
import Transaction exposing (..)
import Tuple exposing (..)

type Msg
  = Inc Int
  | Dec Int

type alias Storage = Int

type Option a
  = Some a
  | None

type Prod a b
  = Pair a b

type Sumbool
  = Left
  | Right

my_bool_dec : Bool → Bool → Sumbool
my_bool_dec b1 b2 =
  (case b1 of
     True →
       \x → case x of
              True →
                Left
              False →
                Right
     False →
       \x → case x of
              True →
                Right
              False →
                Left) b2

type Sig a
  = Exist a

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

inc_counter : Storage → Sig Int → Sig Storage
inc_counter st inc =
  Exist (add st (proj1_sig inc))
```

```
dec_counter : Storage → Sig Int → Sig Storage
dec_counter st dec =
  Exist (sub st (proj1_sig dec))

counter : Msg → Storage → Option (Prod Transaction Storage)
counter msg st =
  case msg of
    Inc i →
      case my_bool_dec (lt 0 i) True of
        Left →
          Some (Pair Transaction.none (proj1_sig (inc_counter st (
                Exist i))))
        Right →
          None
    Dec i →
      case my_bool_dec (lt 0 i) True of
        Left →
          Some (Pair Transaction.none (proj1_sig (dec_counter st (
                Exist i))))
        Right →
          None
```