ConCert: A Smart Contract Certification Framework in Coq

Danil Annenkov
daan@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 present a new way of embedding functional languages into the Coq proof assistant by using meta-programming. This allows us to develop the meta-theory of the language using the deep embedding and provides a convenient way for reasoning about concrete programs using the shallow embedding. We connect the deep and the shallow embeddings by a soundness theorem. As an instance of our approach, we develop an embedding of a core smart contract language into Coq and verify several important properties of a crowdfunding contract based on a previous formalisation of smart contract execution in blockchains.

CCS Concepts   • **Theory of computation** → **Logic and verification**; **Program verification**; *Type theory*; *Program semantics*; • **Software and its engineering** → **Correctness**; **Formal software verification**; *Software verification.*

Keywords   certified programming, software correctness, Coq, smart contracts, blockchain, functional programming languages

1 Introduction

The concept of blockchain-based smart contracts has evolved in several ways since its appearance. Starting from the restricted and non-Turing-complete Bitcoin script[1] designed to validate transactions, the idea of smart contracts expanded to fully-featured languages such as Solidity running on the Ethereum Virtual Machine (EVM).[2] Recent research on smart contract verification discovered the presence of multiple vulnerabilities in many smart contracts written in Solidity [13, 19]. Several times the issues in smart contract implementations resulted in huge financial losses (for example, the DAO contract and the Parity multi-sig wallet on Ethereum). The setup for smart contracts is unique: once deployed, they cannot be changed and any small mistake in the contract logic may lead to serious financial consequences. This shows not only the importance of formal verification of smart contracts but also the importance of principled programming language design. The third generation smart contract languages tend to employ the functional programming paradigm. A number of blockchain implementations have already adopted certain variations of functional languages as a smart contract language. These languages range from minimalistic and low-level (Simplicity [15], Michelson[3]), intermediate (Scilla [20]) to fully-featured OCaml- and Haskell-like languages (Liquidity [6], Plutus [7, 17]). There is a very good reason for this tendency. Statically typed functional programming languages can rule out many errors. Moreover, due to the absence (or more precise control) of side effects programs in functional languages behave like mathematical functions, which facilitates reasoning about them. However, one cannot hope to perform only stateless computations: the state is inherent for blockchains. One way to approach this is to limit the ways of changing the state. While Solidity allows arbitrary state modifications at any point of execution, many modern smart contract languages represent smart contract execution as a function from a current state to a new state. This functional nature of modern smart contract languages makes them well-suited for formal reasoning.

---

[1]Bitcoin: A peer-to-peer electronic cash system.
https://bitcoin.org/bitcoin.pdf
[2]Ethereum's white paper:
https://github.com/ethereum/wiki/wiki/White-Paper
[3]https://www.michelson-lang.com/

The Ethereum Virtual Machine and the Solidity smart contract language remain one of the most used platforms for writing smart contacts. Due to the permissiveness of the underlying execution model and the complexity of the language, verification in this setting is quite challenging. On the other hand, many modern languages such as Acorn,[4] Liquidity and Scilla, offer a different execution model and a type system allowing to rule out many errors through type checking. Of course, many important properties are not possible to capture even with powerful type systems of functional smart contract languages. For that reason, to provide even higher guarantees, such as functional correctness, one has to resort to stronger type systems/logics for reasoning about programs and employ deductive verification techniques. Among various tools for that purpose proof assistants provide a versatile solution for that problem.

Proof assistants or interactive theorem provers are tools that allow users to state and prove theorems interactively. Proof assistants often offer some degree of proof automation by implementing decision and semi-decision procedures, or interacting with automated theorem provers (SAT and SMT solvers). Some proof assistants allow for writing user-defined automation scripts, or write extensions using a plug-in system. This is especially important, since many properties of programs are undecidable and providing users with a convenient way of interactive proving while retaining a possibility to do automatic reasoning makes proof assistants very flexible tools for verification of smart contracts.

Existing formalisations of functional smart contract languages mostly focus on meta-theory (Plutus [7], Simplicity [15]) or meta-theory and verification using the deep embedding (Michelson [5]). An exception is Scilla [20], which features verification of particular smart contracts in the Coq proof assistant by means of shallow embedding by hand. Simplicity [15] is a low-level combinator-based functional language and its formalisation allows for translating from deep to shallow embeddings for purposes of meta-theoretic reasoning. None of these developments combines deep and shallow embeddings for a *high-level* functional smart contract language in one framework or provide an automatic way of converting smart contracts to Coq programs for convenient verification of concrete smart contracts. We are making a step towards this direction by allowing for deep and shallow embeddings to coexist and interact in Coq.

The contributions of this paper are the following:

1. We develop an approach to verify properties of functional programming languages and of individual programs in one framework. In particular, this approach works for functional smart contract languages and concrete contracts.

2. We describe a novel way of combining deep and shallow embeddings using the meta-programming facilities of Coq (MetaCoq [3]).
3. As an instance of our approach, we define the syntax and semantics of λ_{smart} — a core subset of the Acorn language (the deep embedding) and the corresponding translation of λ_{smart} programs into Coq functions (the shallow embedding).
4. We prove properties of a crowdfunding contract given as a deep embedding (abstract syntax tree) of a λ_{smart} program.
5. We integrate our shallow embedding with the smart contract execution framework [14] allowing for proving safety and temporal properties of interacting smart contracts.

We discuss the details of our approach in Section 2, provide an example of a crowdfunding contract verification in Section 3. In Section 4 we apply our framework to verify a List module of the Acorn standard library and discuss how our development integrates with the execution framework in Section 5. Theorems from Section 2.4 and lemmas from Sections 3, 4 and 5 are proved in our Coq development and available at https://github.com/AU-COBRA/ConCert/tree/artefact.

2 Our Approach

There are various ways of reasoning about properties of a functional programming language in a proof assistant. First, let us split the properties into two groups: meta-theoretical properties (properties of a language itself) and properties of programs written in the language. Since we are focused on functional smart contract languages and many proof assistants come with a built-in functional language, it is reasonable to assume that we can reuse the programming language of a proof assistant to express smart contracts and reason about their properties. A somewhat similar approach is taken by the authors of the hs-to-coq library [23], which translates total Haskell programs to Coq by means of source-to-source transformation. Unfortunately, in this case, it is impossible to reason about the correctness of the translation.

We would like to have two representations of functional programs within the same framework: a deep embedding in the form of an abstract syntax tree (AST), and a shallow embedding as a Coq function. While the deep embedding is suitable for meta-theoretical reasoning, the shallow embedding is convenient for proving properties of concrete programs. We use the meta-programming facilities of the MetaCoq plug-in [3] to connect the two ways of reasoning about functional programs.

The overview of the structure of the framework is given in Figure 1. As opposed to source-to-source translations in the style of hs-to-coq [23] and coq-of-ocaml[5] we would like

---

[4]The Acorn language is an ML-style functional smart contract language currently under development at the Concordium Foundation.

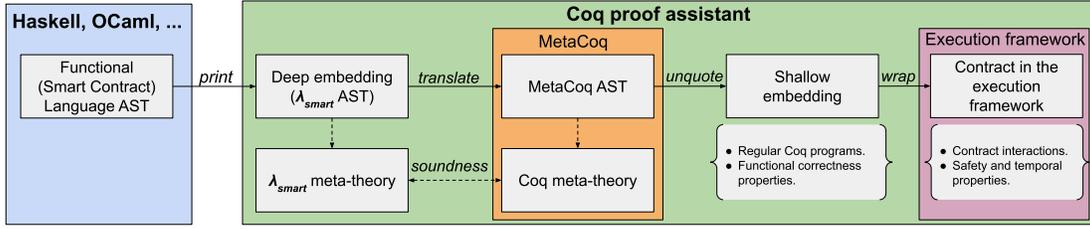[5]The coq-of-ocaml project page: https://github.com/clarus/coq-of-ocaml

Figure 1. The structure of the framework

for all the non-trivial transformations to happen in Coq. This makes it possible to reason within Coq about the translation and formalise the required meta-theory for the language. That is, we start with an AST of a program in a functional language implemented in Haskell, OCaml or some other language, then we generate an AST represented using the constructors of the corresponding inductive type in Coq (deep embedding) by printing the desugared AST of the program. By printing we mean a recursive procedure that converts the AST into a string consisting of the constructors of our Coq representation. The main idea is that this procedure should be as simple as possible and does not involve any non-trivial manipulations since it will be part of a trusted code base. If non-trivial transformations are required, they should happen within the Coq implementation.

2.1 MetaCoq

The MetaCoq project [3] brings together several subprojects united by the use of meta-programming and formalisation of Coq's meta-theory in Coq. In particular, relevant for this project:

- Template Coq — adds meta-programming facilities to Coq. That is, it provides a way to *quote* Coq definitions by producing an AST represented as an inductive data type `term` in Coq, and *unquote* a well-formed inhabitant of *term* back to a Coq definition.
- PCUIC — formalisation of the meta-theory of Polymorphic Cumulative Calculus of Inductive Constructions (PCUIC), an underlying calculus of Coq.[6]

These features of MetaCoq have been used for defining various syntactic translations from Calculus of Inductive Constructions (CIC) to itself (e.g. parametricity translation [4]), developing a certified compiler CertiCoq [2] and for certifying extraction of Coq terms to the untyped lambda-calculus [10]

Let us consider a simple example demonstrating the quote/unquote functionality.

```
(* Quote *)
Quote Definition id_nat_syn := (fun x : nat ⇒ x).
Print id_nat_syn.
(* tLambda (nNamed "x")
   (tInd (mkInd "nat" 0) []) (Ast.tRel 0) : term *)
```

---

[6]From now on, we will use MetaCoq to refer both to the quote/unquote functionality and to the formalisation of meta-theory.

```
(* Unquote *)
Make Definition plus_one :=
  (tLambda (nNamed "x") (tInd (mkInd "nat" 0 ) [])
        (tApp (tConstruct (mkInd "nat" 0) 1 []) [tRel 0])).
Print plus_one.
(* fun x : nat ⇒ S x : nat → nat *)
```

Our use of MetaCoq explores a new way of using meta-programming in Coq. All existing use cases follow (roughly) the following procedure: start with a Coq term, quote it and perform certain transformations (e.g. syntactic translation, erasure, etc.). In our approach, we go in the different direction: starting with the AST of a program in a functional language we want to reason about, through a series of transformations we produce a MetaCoq AST, which is then unquoted into a program in Coq's Gallina language (shallow embedding). The transformations include conversion from the named to the nameless representation (if required) and translation into the MetaCoq AST. The deep embedding also serves as input for developing meta-theory of the functional language.

2.2 The λ_{smart} Language

As an instance of our approach, we develop an embedding of the "core" of the ACORN smart contract language into Coq. We call this core language λ_{smart}. This language contains all the essential features of a realistic functional language: System F type system, inductive types, general recursion and pattern-matching. The grammar of the language is given below.

$$
\begin{aligned}
\tau\ \sigma\quad &::=\quad \hat{i} \mid I \mid \forall A.\tau \mid \tau\ \sigma \mid \tau \to \sigma \\
p\quad &::=\quad C\ x_1\ \ldots\ x_n \\
e\quad &::=\quad \bar{i} \mid \lambda x : \tau.e \mid \Lambda A.e \mid \text{let } x\ :\ \tau\ =\ e_1 \text{ in } e_2 \mid e_1\ e_2 \\
&\quad\quad\mid \text{case } e\ : I\ \tau_1 \ldots \tau_n \text{ return } \sigma \text{ of} \\
&\quad\quad\quad p_1 \to e_1; \ldots; p_m \to e_m \\
&\quad\quad\mid C_I \mid \text{fix } f\ x : \tau_1 \to \tau_2\ =\ e \mid \tau
\end{aligned}
$$

Here I, C, A, x and f range over strings representing names of inductive types, constructors, type variable names, variable names and fixpoint names respectively. We use de Bruijn indices to represent variables both in expressions and types (denoted $\bar{i}$ and $\hat{i}$ respectively). Textual names A, x and f

are only needed for decoration purposes making resulting Coq code more readable. Note, that λ_{smart} expressions are extensively annotated with typing information. For instance, we annotate lambda abstractions with the domain type, for fixpoints we store the types for the domain and for the codomain. Moreover, case-expressions require explicit type of branches.

The semantics of λ_{smart} is given as a definitional interpreter [18]. This gives us an executable semantics for the language. Moreover, since the core language we consider is sufficiently close to Acorn, our interpreter can serve as a *reference* interpreter. The interpreter is implemented in an environment-passing style and works both with named and nameless representations of variables. Due to the potential non-termination, we define our interpreter using the *fuel idiom*: by structural recursion on an additional argument (a natural number). The interpreter function has the following type:[7]

`eval : global_env → nat → env val → expr → res val`

We will use the notation $\text{eval}^n_{\Sigma, \rho}(e)$ to mean `eval Σ n ρ e`. The `global_env` parameter provides mappings from names of inductives to their constructors. The next three parameters are: "fuel", an evaluation environment and a λ_{smart} expression. Note that "fuel" has a different meaning than "gas" for smart contracts: "fuel" is used to limit the recursion depth to ensure termination and not as a measure of computational efforts. The resulting type is `res val`, where `res` is a error monad. The errors could be either `NotEnoughFuel` denoting that fuel provided was not sufficient to complete the execution, or `EvalError` denoting that execution is stuck. In our Coq development `EvalError` also carries a error message. Values `val` are defined as follows:

$$
\begin{aligned}
v \quad ::= \quad & \text{vConstr}(I, C, v_1 \ldots v_n) \\
& | \; \text{vClosLam}(\rho, x, \tau, e) \\
& | \; \text{vClosFix}(\rho, f, x, \tau_1, \tau_2, e) \\
& | \; \text{vTyClos}(\rho, A, e) \; | \; \text{vTy}(\tau)
\end{aligned}
$$

Note that we annotate our value with types and variable names. This is required to match the λ_{smart} interpreter with the MetaCoq evaluation relation (see Section 2.4).

The outline of the most interesting parts of our interpreter is given in Figure 2. Specifically, we show the evaluation of fixpoints and case-expressions. The rest is quite standard and not essentially different from other works using definitional interpreters (e.g. [1]). An essential part of the fixpoint evaluation is how we extend the evaluation environment ρ: we evaluate the body of the fixpoint e in the environment

extended with the closure of the fixpoint itself which corresponds to the recursive call. A fixpoint binds two variables: the outermost one corresponds to the recursive call and the innermost is a fixpoint's argument. Thus evaluating the fixpoint's body in the environment $v_2 :: v_1 :: \rho$ ensures that once we hit a variable corresponding to the recursive call it will be replaced with the body of the fixpoint again. Note that we perform an additional check if v_2 is a constructor of an inductive type (possibly applied to some arguments). This is necessary to match the evaluation of λ_{smart} expressions with the MetaCoq evaluation relation. Therefore, it is not possible to pass a function as an argument to a λ_{smart} fixpoint. Although this sounds limiting, the main point of the λ_{smart} semantics is to prove the correctness of the embedding to Coq and fixpoints in Coq are limited to structurally recursive definitions.

When evaluating case expressions, our interpreter first evaluates all the types (parameters of the inductive and the type of branches), then the discriminee, and if the latter evaluates to a constructor, executes a simple pattern-matching algorithm. The `match_pat` function returns a branch that matches the discriminee. Next, we evaluate the body of the selected branch in the environment extended with the reverse list of the constructor's arguments.

Note that our interpreter evaluates all the type expressions. This is required since we want to match λ_{smart} evaluation with the evaluation relation of MetaCoq, which evaluates corresponding types. The type evaluation function `eval_ty :` `env val × type → res type` essentially just substitutes values from the evaluation environment and fails if there is no corresponding value found.

Note that we perform some validation of expressions against the evaluation environment. This is again required for our soundness result. We will discuss these questions in Section 2.4.

2.3 Translation to MetaCoq

We define the translations from the AST of λ_{smart} expressions `expr` and types `type` to MetaCoq abstract syntax `term` by structural recursion.[8] On Figure 3 we outline the translation functions. We use Haskell-like notation and blue colour for λ_{smart} expressions and types and Coq-like notation and green colour for MetaCoq terms. We assume that the global environment Σ : `global_env` contains all inductive type definitions mentioned in λ_{smart} expressions. Under this assumption the translation function $[\![-]\!]^t_\Sigma$ is total.

In λ_{smart}, we have two kinds of de Bruijn indices: the one for type variables $\hat{i}$ and the one for term variables $\bar{i}$. In the

---

[7]In our development, the interpreter supports two modes of evaluation: with named variables and with de Bruijn representation of variables (nameless). For the purposes of this paper, we will focus on the nameless mode.

[8]By MetaCoq `term` we mean a corresponding inductive type from the PCUIC formalisation. In our Coq development, before unquoting a PCUIC term, we translate it into a kernel representation. This translation is almost one-to-one, apart from the application case: it is unary in PCUIC and n-ary in the kernel representation, but this part is quite straightforward to handle. Eventually, this translation will be included in the MetaCoq project

$$\mathrm{eval}^0_{\Sigma,\rho}(e) \overset{def}{=} \mathtt{NotEnoughFuel}$$

$$\mathrm{eval}^{S\ n}_{\Sigma,\rho}(e) \overset{def}{=} \mathtt{match}\ e\ \mathtt{with}$$

```
                  . . .
                  | fix f x : τ₁ → τ₂ = e ⇒
                      τ'₁ ← eval_ty(τ₁);
                      τ'₂ ← eval_ty(τ₂);
                      validate(ρ, 2, e);
                      Ok (vClosFix(ρ, f, x, τ'₁, τ'₂, e))
                  | e₁ e₂ ⇒ v₂ ← evalⁿ_{Σ,ρ}(e₂);
                            v₁ ← evalⁿ_{Σ,ρ}(e₁);
                            match v₁ with
                            . . .
                            | vClosFix(ρ, f, x, τ₁, τ₂, e) ⇒
                                if isConstr(v₂) then evalⁿ_{Σ, v₂::v₁::ρ}(e)
                                              else EvalError
                      end
```

```
                  | case e : I τ₁ . . . τ_k return σ of bs ⇒
                          validate_branches(ρ, bs);
                          _ ← eval_ty(σ);
                          _ ← monad_map eval_ty [τ₁; . . . ; τ_k];
                          v ← evalⁿ_{Σ,ρ}(e);
                          match v with
                          | vConstr(I', C, args) ⇒
                              let(_, tys) := resolve_ctor(Σ, I', C) in
                              if (I = I') then
                                  e' ← match_pat(C, n, tys, args, bs);
                                  evalⁿ_{Σ, rev(args)+ρ}(e')
                              else EvalError
                          | _ ⇒ EvalError
                          end
                  end
```

Figure 2. λ_{smart} interpreter.

translation, we map both of them to the single kind of indices of MetaCoq. Constructors are translated to MetaCoq constructors by first looking up the corresponding constructor number in the global environment. In MetaCoq (and in the kernel of Coq) constructors are represented as numbers corresponding to the position of a constructor in the list of constructors for a given inductive type. The type of global environments `global_env` is a list of definitions of inductive type. The functions

$$\texttt{resolve\_ind} : \texttt{global\_env} \times \texttt{ident}$$
$$\rightarrow \texttt{option (list constr)}$$
$$\texttt{resolve\_ctor} : \texttt{global\_env} \times \texttt{ident} \times \texttt{ident}$$
$$\rightarrow \texttt{option}\ (\mathbb{N} \times \texttt{constr})$$

are used to look up for inductive type and their constructors. Particularly, `resolve_ind` returns a list of constructors for a given textual name of an inductive type, while `resolve_ctor` returns a position of a constructor in the list of constructor definitions and the constructor definition itself. Translation of a λ_{smart} constructor C_I looks up the corresponding constructor position in the list of constructors for the inductive I, a translated constructor in MetaCoq is basically a number (a position) annotated with the name of the inductive type: C_I (and universes, but these are not relevant for us right now). In the translation of `fix`, the type of a fixpoint is translated into a Π-type in MetaCoq. Therefore, the indices of free variables in the codomain type must be incremented ("lifted") by 1, since Π-types bind a variable. We denote such increments by n as $\uparrow_n$. The other feature of the fixpoint translation is that the body of a fixpoint in MetaCoq becomes a lambda abstraction and since all lambda abstractions must be explicitly annotated with a type of the domain, we provide this type. Again, we have to lift free variables in the type, because the outermost variable index corresponds to the body of the fixpoint itself.

By far the most complex translation case is the pattern-matching. The first complication stems to the representation of branches for `match` in MetaCoq: the branches should be arranged in the same order as constructors in the definition of the corresponding inductive type. In this case, there is no need to store constructor names in each pattern. On the other hand, in λ_{smart} we choose more user-friendly implementation: patterns are explicitly named after constructors and might follow in an order that is different from how the order in the inductive type definition. For that reason, we first resolve the inductive type from the global environment to get a list of constructors. Then, for each constructor in the list, we call the `branch` function. As one can see from Figure 3, the translated branches follow the same order as in they appear in the list of resolved constructors, i.e. $c_1 \ldots c_m$.

The second difficulty arises from the pattern representation. In MetaCoq, patterns are desugared to more basic building blocks: lambda abstractions. Therefore, every pattern becomes an iterated lambda term. Before we explain how the `branch` function works, let us first describe the representation of inductive types in the global environment. Each inductive type definition consists of a name, number of parameters, and the list of constructors. In turn, each constructor consists of a name and a list of argument types. Since λ_{smart}'s type system does not feature dependent types, it is sufficient to store a list of arguments for each constructor instead of a full type. Each type in the list of constructor arguments can refer to parameters as if they were bound at the top level for each type. For example, the type of finite maps (in the form of association lists) would look like the following (we use concrete syntax here for the presentation purposes):

```
data AcornMap #2 = MNil [] | MCons [î, ô, AcornMap î ô]
```

In the example above, AcornMap has two parameters (the number of parameters is specified after the type name): a type of keys, and a type of values. The MNil constructor does not take any arguments, and MCons takes a key, a value and an inhabitant of AcornMap. Given this representation of constructors, we continue the explanation of the pattern-matching translation. When pattern-matching on a parameterised inductive, the inductive is applied to some parameters. In order to propagate these parameters to the corresponding constructors, we have to substitute the concrete parameters into each type in the constructor arguments list. We use $t\{ts\}$ to denote the MetaCoq parallel substitution of a list of terms ts into a term t. In the branch function, we first look up a branch body e in the list of branches by comparing the constructor name to the pattern name. Next, we project constructor argument types from a given constructor. Further, since patterns become iterated lambdas in MetaCoq, we need to provide a type for each abstracted variable. Since Coq is dependently typed, variables bound by each lambda abstraction might appear in the types that appear later in the term. Thus, to avoid variable capture in types of lambda arguments, we need to lift the translated types. This is exactly what happens in the branch function: the resulting MetaCoq term is an iterated lambda abstraction and each argument type is lifted according to the number of preceding lambda abstractions in the translation of a pattern.

The translation of types is mostly straightforward. For the universal types, we choose to produce a Π-type with the domain in Set. Such a choice allows us to avoid dealing with universe levels explicitly, as it is required in MetaCoq. The translation we present works only for a *predicative* fragment of λ_{smart}, but Acorn's surface language supports only a prenex form of universal types, where all the quantifiers appear at the topmost level. Therefore, this is not a limitation from a practical point of view. Another thing to note is that the translation of *definitions* of inductive types is not shown in Figure 3, although it is implemented in our Coq development. Instead of giving the full translation in the paper (which involves subtle de Bruijn indices manipulations), let us consider an example. We continue with the finite maps AcornMap. The tricky bit in the translation is to produce a correct type for each constructor considering the number of parameters and taking into account that each Π-type binds a new variable. Moreover, in MetaCoq the inductive type being defined becomes a topmost variable as well.

The resulting MetaCoq definition of AcornMap (again, in the concrete syntax, but with explicit indices in place of variable names) looks as follows:

```
Inductive AcornMap (A1 A2 : Set) :=
| MNil : 2̄ 1̄ 0̄
| MCons : forall (_ : 1̄) (_ : 1̄) (_ : 4̄ 3̄ 2̄), 5̄ 4̄ 3̄,
```

Let us consider the type of MNil. The index $\bar{2}$ refers to the topmost variable being the inductive type itself, index $\bar{1}$ refers to the parameter A1 and $\bar{0}$ to the parameter A2.

Our Coq development contains the definition of AcornMap using the deep embedding as well as standard operations of finite maps.[9] Moreover, we demonstrate how one can covert definitions, given as a deep embedding, to regular Coq definitions by translating and unquoting them. In the same time, one can run programs directly on deep embedding using the interpreter (Figure 2).

2.4 Translation Soundness

Since the development of the meta-theory of Coq itself is one of the aims of MetaCoq we can use this development to show that the semantics of λ_{smart} agrees with its translation to MetaCoq (on terminating programs). The idea is to compare the results of the evaluation of λ_{smart} expressions with the weak head call-by-value evaluation relation of MetaCoq up to the appropriate conversion of values. This conversion of values is a non-trivial procedure: λ_{smart} values contain *closures*, while the MetaCoq evaluation relation is substitution based and produces a subset of *terms* in the weak head normal form. Therefore, if we want to eventually convert λ_{smart} values to MetaCoq terms, first, we need to substitute environments into the closures' bodies. E.g. for vClosLam(ρ, x, τ, e) we need to substitute all the values from ρ into e. This is not possible to do directly, because we cannot substitute values into expressions. Thus, we need to convert all the values to expressions in the environment ρ. But this, in turn, requires substituting environments in closures again. To break this circle, we take inspiration from [11] and first define substitution functions purely on λ_{smart} expressions and types:

$$\tau[-] : \text{env expr} \rightarrow \text{option type}$$
$$e[-] : \text{env expr} \rightarrow \text{option expr}$$

These functions implement parallel substitution of the environment represented as a list of expressions. Unfortunately, these functions are partial, due to the fact that we use one environment for term-level values and for type-level-values. We can make this function total by imposing a well-formedness condition. With these substitution operations we now can define a conversion procedure from λ_{smart} values back to expressions:

$$\text{of\_val}(\text{vClosLam}(\rho, x, \tau, e)) \stackrel{def}{=} \text{let } \rho' := \text{map of\_val } \rho \text{ in}$$
$$(\lambda x : \tau.e)[\rho']$$

$$\text{of\_val}(\text{vClosFix}(\rho, f, x, \tau_1, \tau_2, e)) \stackrel{def}{=} \text{let } \rho' := \text{map of\_val } \rho \text{ in}$$
$$(\text{fix } f \; x : \tau_1 \rightarrow \tau_2 = e)[\rho']$$

$$\text{of\_val}(\text{vConstr}(I, C, v_1 \ldots v_n)) \stackrel{def}{=}$$
$$C_I \text{ of\_val}(v_1) \ldots \text{of\_val}(v_n)$$

---

[9]See theories/examples/FinMap.v file in our Coq development

$$[\![-]\!]^t : \texttt{global\_env} \times \texttt{expr} \to \texttt{term} \qquad\qquad [\![-]\!]^T : \texttt{type} \to \texttt{term}$$

$$[\![\hat{i}]\!]^t_\Sigma \overset{def}{=} \bar{i} \qquad\qquad\qquad\qquad [\![\hat{i}]\!]^T \overset{def}{=} \bar{i}$$

$$[\![\lambda x : \tau.e]\!]^t_\Sigma \overset{def}{=} \texttt{fun}\,(x : [\![\tau]\!]^T) \Rightarrow [\![e]\!]^t_\Sigma \qquad\qquad [\![I]\!]^T \overset{def}{=} I$$

$$[\![\Lambda A.e]\!]^t_\Sigma \overset{def}{=} \texttt{fun}\,(A : \texttt{Set}) \Rightarrow [\![e]\!]^t_\Sigma \qquad\qquad [\![\tau]\!]^T \overset{def}{=} [\![\tau]\!]^T$$

$$[\![\texttt{let}\,x : \tau = e_1\,\texttt{in}\,e_2]\!]^t_\Sigma \overset{def}{=} \texttt{let}\,x : [\![\tau]\!]^T := [\![e_1]\!]^t_\Sigma\,\texttt{in}\,[\![e_2]\!]^t_\Sigma \qquad [\![\forall A.\tau]\!]^T \overset{def}{=} \texttt{forall}\,(A : \texttt{Set}),\,[\![\tau]\!]^T$$

$$[\![e_1\,e_2]\!]^t_\Sigma \overset{def}{=} [\![e_1]\!]^t_\Sigma\,[\![e_2]\!]^t_\Sigma \qquad\qquad [\![\tau_1\,\tau_2]\!]^T \overset{def}{=} [\![\tau_1]\!]^T\,[\![\tau_2]\!]^T$$

$$[\![C_I]\!]^t_\Sigma \overset{def}{=} \texttt{let}(C, \_) := \texttt{resolve\_ctor}(\Sigma, C, I) \qquad [\![\tau_1 \to \tau_2]\!]^T \overset{def}{=} \texttt{forall}\,(\_ :[\![\tau_1]\!]^T),\,\uparrow_1 [\![\tau_2]\!]^T$$
$$\texttt{in}\,C_I$$

$$[\![\texttt{fix}\,f\,x : \tau_1 \to \tau_2 = e]\!]^t_\Sigma \overset{def}{=} \texttt{fix}\,(f : \texttt{forall}\,(\_ : [\![\tau_1]\!]^T),\,\uparrow_1 [\![\tau_2]\!]^T) := \texttt{fun}\,(x : \uparrow_1 [\![\tau_1]\!]^T) \Rightarrow [\![e]\!]^t_\Sigma$$

$$\left[\!\!\!\left[\begin{matrix}\texttt{case}\quad e : I\,\tau_1\dots\tau_n\,\texttt{return}\,\sigma\,\texttt{of}\\ p_1 \to e_1\\ \dots\\ p_m \to e_m\end{matrix}\right]\!\!\!\right]_\Sigma \overset{def}{=} \begin{matrix}\texttt{match}\quad [\![e]\!]^t_\Sigma\,\texttt{as}\,\_\,\texttt{in}\,I\,[\![\tau_1]\!]^T\dots[\![\tau_n]\!]^T\,\texttt{return}\,\uparrow_1 [\![\sigma]\!]^T\,\texttt{with}\\ \texttt{branch}(\overrightarrow{\tau_i}, \overrightarrow{p_i \to e_i}, c_1)\\ \dots\\ \texttt{branch}(\overrightarrow{\tau_i}, \overrightarrow{p_i \to e_i}, c_m)\end{matrix}$$

where

$$c_1\dots c_m \overset{def}{=} \texttt{resolve\_ind}(\Sigma, I) \qquad \overrightarrow{\tau_i} \overset{def}{=} \tau_1,\dots,\tau_n \qquad [\![\overrightarrow{\tau_i}]\!]^T \overset{def}{=} [\![\tau_1]\!]^T,\dots,[\![\tau_n]\!]^T \qquad \overrightarrow{p_i \to e_i} \overset{def}{=} p_1 \to e_1,\dots,p_m \to e_m$$

$$\texttt{branch} : (\texttt{list}\,\texttt{type}) \times (\texttt{list}\,(\texttt{pat} \times \texttt{expr})) \times \texttt{constr} \to \texttt{term}$$

$$\texttt{branch}(\overrightarrow{\tau_i}, \overrightarrow{p_i \to e_i}, c) \overset{def}{=} \texttt{let}\,e := \texttt{find}(\texttt{ctor\_name}(c), \overrightarrow{p_i \to e_i})\,\texttt{in}$$
$$\texttt{let}\,\sigma_1, \sigma_2, \dots, \sigma_k := \texttt{ctor\_args}(c)\,\texttt{in}$$
$$\texttt{fun}\,(x_1 : [\![\sigma_1]\!]^T\{[\![\overrightarrow{\tau_i}]\!]^T\})\,(x_2 : [\![\sigma_2]\!]^T\{\uparrow_1 [\![\overrightarrow{\tau_i}]\!]^T\})\dots(x_k : [\![\sigma_k]\!]^T\{\uparrow_{k-1} [\![\overrightarrow{\tau_i}]\!]^T\}) \Rightarrow [\![e]\!]^t_\Sigma$$

Figure 3. Translation to MetaCoq

$$\texttt{of\_val}(\texttt{vTyClos}(\rho, A, e)) \overset{def}{=} \texttt{let}\rho' := \texttt{map}\,\texttt{of\_val}\,\rho\,\texttt{in}$$
$$(\Lambda A.e)[\rho']$$
$$\texttt{of\_val}(\texttt{vTy}(\tau)) \overset{def}{=} \tau$$

Once we have a way of converting values to expressions, we can use the translation function $[\![-]\!]^t$ to produce MetaCoq terms. This gives us a direct way of comparing the evaluation results. Before we state the soundness theorem, we give an overview of some important lemmas forming a core of the proof. First of all, let us mention certain well-formedness conditions for the environments involved in our definitions. It is very important to carefully set up these conditions before approaching the soundness proof.

Definition 1. *For a global environment Σ : global_env, evaluation environment ρ : env expr and λ_{smart} value v : val we say that*

(WF.i) Σ is well-formed if for all definitions of inductive types, each constructor type is closed for the given number of parameters of the inductive type. E.g. if an inductive type has n parameters, then the type of each constructor has at most n free variables.

(WF.ii) ρ is well-formed wrt. an expression e when for any type variables mentioned in e, if there is a corresponding expression in ρ it corresponds to a type.

(WF.iii) a value v is well-formed if all the expressions and types in the closures are appropriately closed wrt. corresponding environments in closures and ρ is well-formed in the sense of (WF.ii). E.g. for $\texttt{vClosLam}(\rho, x, \tau, e)$ we have: ρ contains only well-formed values, e has at most $|\rho| + 1$ free variables ($|\rho|$ is the size of ρ), τ is closed type value, and ρ is well-formed wrt. e. Additionally, for $\texttt{vConstr}(I, C, \text{args})$, we require that $\texttt{resolve\_ctor}(\Sigma, I, C)$ returns some value.

Now, we will state several lemmas crucial for the soundness proof. We will emphasise the use of the conditions *(WF.i),(WF.ii)* and *(WF.iii)* throughout these lemmas. We will use the following additional notations: $t\{ts\}$ for the MetaCoq parallel substitution as in the translation, $[\![\rho]\!]^t_\Sigma$ for translation of all the expressions in ρ from λ_{smart} to MetaCoq and $|\rho|$ for the environment size.

Lemma 1 (Environment substitution). *For any λ_{smart} expression e, well-formed global environment Σ (WF.i), well-formed environment ρ wrt. e (WF.ii), such that all the expressions in ρ*

are closed the following holds

$$\llbracket e[\rho] \rrbracket_\Sigma^t = (\llbracket e \rrbracket_\Sigma^t)\{\llbracket \rho \rrbracket_\Sigma^t\}$$

Lemma 1 says that environment substitution commutes with the translation. The well-formedness condition on ρ ensures that the environment substitution functions are total.

Lemma 2 (Well-formed values). *For any λ_{smart} expression e, such that e has at most $|\rho|$ free variables, number of steps n, well-formed evaluation environment ρ, such that all the values in ρ are well-formed, if evaluation of e terminates with some value v, i.e. $\mathrm{eval}^n_{\Sigma, \rho}(e) = \mathrm{Ok}\ v$, then the value v is well-formed ((WF.iii)).*

In the interpreter in Figure 2 we perform certain dynamic checks denoted by validate and validate_branches. These checks ensure that the condition *(WF.iii)* is satisfied for values produces by the interpreter. For well-typed expressions, this condition would be automatically satisfied, but currently, we focus on dynamic semantics.

We use the interpreter for λ_{smart} expressions and call-by-value evaluation relation of MetaCoq to state the soundness theorem. The MetaCoq evaluation relation is a subrelation of the transitive reflexive closure of the one-step reduction relation and designed to represent the evaluation of ML languages at Coq level.

Theorem 1 (Soundness). *For any λ_{smart} expression e, number of steps n, well-formed global environment Σ, evaluation environment ρ, such that all the values in ρ are well-formed and $e[\rho]$ is closed, if $\mathrm{eval}^n_{\Sigma, \rho}(e) = \mathrm{Ok}\ v$, for some value v, then $\llbracket e[\rho] \rrbracket_\Sigma^t \Downarrow \llbracket \mathrm{of\_val}(v) \rrbracket_\Sigma^t$, where $-\Downarrow-$ is the call-by-value evaluation relation of MetaCoq.*

Proof. By induction on the number of steps of the interpreter n. The base case is trivial, since we assume that the interpreter terminates. In the inductive step, the proof proceeds by case analysis on e using Lemma 1 in the cases involving substitution (e.g. cases for let, application and case-expressions) and using Lemma 2 to obtain premises that all the values in ρ are well-formed required for applying induction hypotheses.   □

Corollary 1 (Soundness for closed expressions). *For any closed λ_{smart} expression e, number of steps n, well-formed global environment Σ, if $\mathrm{eval}^n_{\Sigma, []}(e) = \mathrm{Ok}\ v$, then $\llbracket e \rrbracket_\Sigma^t \Downarrow \llbracket \mathrm{of\_val}(v) \rrbracket_\Sigma^t$.*

Proof. By Theorem 1, using the fact that the empty evaluation environment is trivially well-formed and the fact that substituting the empty environment does not change e.   □

We can see our translation to MetaCoq as some form of denotational semantics. With this view, we can obtain the adequacy result.

Theorem 2 (Adequacy for terminating programs). *For any closed λ_{smart} expression e, well-formed global environment Σ, if*

evaluation of e terminates with value v in n steps and $\llbracket e \rrbracket_\Sigma^t \Downarrow t$ for some term t, then $t = \llbracket \mathrm{of\_val}(v) \rrbracket_\Sigma^t$.

Proof. By Corollary 1, using the fact that the MetaCoq evaluation relation is deterministic.   □

Theorem 2 readily allows, for example, transferring program equivalence from MetaCoq derivations to the λ_{smart} interpreter, provided that the values are not higher-order (i.e. not closures).

In general, we conjecture that one can show adequacy for any program for which there exists a derivation of the MetaCoq big-step evaluation relation. Such a theorem should be stated for well-typed λ_{smart} expressions. Moreover, transferring properties proved for Coq functions to the corresponding evaluations using the interpreter in general also requires resorting to the typing argument. Currently, we do not formalise static semantics and leave these points as future work.

We assume that the unquote functionality of MetaCoq is implemented correctly. From that perspective, unquote becomes part of the trusted computing base, but we would like to emphasise that one of the goals of the MetaCoq project is to implement the actual kernel of Coq in Coq itself. The current MetaCoq data type term corresponds directly to the constr data type from Coq's kernel. Therefore, unquote is a straightforward one-to-one mapping of MetaCoq data types to the corresponding OCaml data types of Coq's kernel. This is in contrast to projects like hs-to-coq and coq-of-ocaml for which the whole translation has to be trusted.

We developed a full formalisation of theorems and lemmas presented in this section in our framework in Coq. We do not use any extra axioms throughout our development, but Theorem 2 uses the determinism of the MetaCoq evaluation relation and the proof of this fact is currently under development in the MetaCoq project. Being able to relate the semantics of λ_{smart} to the semantics of Coq through Coq's meta-theory formalisation gives strong guarantees that our shallow embedding reflects the actual behaviour of λ_{smart}. The described approach provides a more principled way of embedding of functional languages than the source-to-source based approaches. Moreover, the translation involves manipulation of de Bruijn indices, which is often quite hard to get right. Various mistakes in non-trivial places were discovered and fixed in the course of the formalisation.

3 The Crowdfunding Contract

As an example of our approach, we consider verification of some properties of a crowdfunding contract (Figure 4). Such a contract allows arbitrary users to donate money within a deadline. If the crowdfunding goal is reached, the owner can withdraw the total amount from the account after the deadline has passed. Also, users can withdraw their donations after the deadline if the goal has not been reached. Contracts like this are standard applications of smart contracts and

appear in a number of tutorials.[10] We follow the example of Scilla [20] and adopt a variant of a crowdfunding contract as a good instance to demonstrate our verification techniques.

We extensively use a new feature of Coq called "custom entries" to provide a convenient notation for our deep embedding.[11] The program texts in Figure 4 written inside the special brackets [\ ... \], [| ... |], and [! ... !] are parsed according to the custom notation rules. For example, without using notations the definition of `action_syn` looks as follows:

```
gdInd Action 0 [("Transfer", [(nAnon, tyInd "nat");
                    (nAnon, tyInd "nat")]);("Empty", [])] false.
```

This AST might be printed directly from the smart contract AST by a simple procedure (as we use in Section 4). We start by defining the required data structures such as `State` and `Msg` meaning contract state and messages accepted by this contract. We pre-generate string constants for corresponding names of inductive types, constructors, etc. using the MetaCoq template monad.[12] This allows for more readable presentation using our notation mechanism. Currently, we use the `nat` type of Coq to represent account addresses and currency. Eventually, these types will be replaced with corresponding formalisations of these primitive types. We also use abbreviations for the result type and for certain types from the blockchain infrastructure, which we are going to explain later.

The `trans_global_dec : global_dec → mutual_inductive_entry` function takes the syntax of the data type declarations and produces an element of `mutual_inductive_entry` — a MetaCoq representation for inductive types. For each of our deeply embedded data type definitions, we produce corresponding definitions of inductive types in Coq by using the `Make Inductive` command of MetaCoq that "unquotes" given instances of the `mutual_inductive_entry` type. Similar notation mechanism is used to write programs using the deep embedding. The definition of `crowdfunding` represents a syntax of the crowdfunding contract. We translate the crowdfunding contract's AST into a MetaCoq AST using the `expr_to_term : global_env → expr → term` function (corresponding to $[\![-]\!]_\Sigma^t$ in Figure 3). Here, `global_env` is a global environment containing declarations of inductive types used in the function definition, `expr` is a type of λ_{smart} expressions, and `term` is a type of MetaCoq terms. Before translating the λ_{smart} expressions, we apply the `indexify` function that converts named variables into de

Bruijn indices. The result of these transformations is unquoted with the `Make Definition` command. After unquoting the translated definitions, they are added to Coq's global environment and available for using as any other definitions. The crowdfunding contract consists of two functions:

```
init : SimpleContractCallContext → nat → Z → State_coq
```

```
receive : SimpleChain → SimpleContractCallContext
          → Msg_coq → State_coq
          → option (State_coq × list SimpleActionBody)
```

Here `SimpleChain` is a "contract's view" of a blockchain allowing for accessing, among other parameters, current slot number (used as a current time); `SimpleContractCallContext` is a contract call context containing transferred amount, sender's address and other information available for inspection during the contract call. The type names with the "coq" suffix correspond to the unquoted data types from the Figure 4.

The `init` function sets up an initial state for a given deadline and goal. The `receive` function corresponds to a transition from a current state of the contract to a new state. We will provide more details about the execution model in Section 5. In the current section we focus on functional correctness properties using pre- and post-conditions. Similarly to [20], we prove a number of properties of the contract using the shallow embedding:

(P.i) the contract `receive` function preserves the following invariant (unless the "done" flag is set to `true`): the sum of individual contributions is equal to the balance recorded in the contract's state;

(P.ii) the donations can be paid back to the backers if the goal is not reached within a deadline;

(P.iii) donations are recorded correctly in the contract's state;

(P.iv) backers cannot claim their contributions if the campaign has succeeded.

The lemma corresponding to the property (P.i) is given below.

```
Lemma contract_state_consistent BC CallCtx msg :
  {{ consistent_balance }}
    receive BC CallCtx msg
  {{ fun fin txs ⇒ consistent_balance fin }}.
```

In the example above we use the Hoare triple notation {{ P }} c {{ Q }} to state pre- and post-conditions for the state before and after the contract call. The post-condition also allows for stating properties of outgoing transactions. We define `consistent_balance` as follows:

```
Definition consistent_balance (lstate : State_coq) :=
  ~ lstate.(done_coq) →
  sum_map (donations_coq lstate) = balance_coq lstate.
```

i.e. the contract balance is consistent if before the contract is marked as "done" the sum of individual contributions is equal to the balance.

Given the definitions above, one can read the lemma in the following way: if the balance was consistent in some initial

---

```
(* Defining AST using customised notations        *)
(* Brackets [\ \] delimit the scope of global definitions, *)
(*        [| |] the scope of programs,              *)
(*        [! !] the scope of types                  *)
(* Local state *)
Definition state_syn : global_dec :=
[\ record State :=
    mkState { balance : Money ;
             donations : Map;
             owner : Address;
             deadline : Nat;
             done : Bool;
             goal : Money } \].
Make Inductive (trans_global_dec state_syn).

(* Messages. Constructors do not take any arguments.*)
Definition msg_syn :=
  [\ data Msg =
        Donate [_]
      | GetFunds [_]
      | Claim [_] \].

Make Inductive (trans_global_dec msg_syn).

(* Abbreviations for types of the blockchain infrastructure *)
Notation SActionBody := "SimpleActionBody".
Notation SCtx := "SimpleContractCallContext".
Notation SChain := "SimpleChain".

(* An abbreviation for the return type *)
Notation "'Result'" := [! "prod" State ("list" "SimpleActionBody") !]
                    (in custom type at level 2).

(* Initialisation function *)
Definition crowdfunding_init : expr :=
[| \c : SCtx ⟹ \dl : Nat ⟹ \g : Money ⟹
  mkState 0z MNil dl (ctx_from c) False g |].

Make Definition init :=
  (expr_to_term Σ' (indexify nil crowdfunding_init)).
```

```
(* The main functionality *)
Definition crowdfunding : expr :=
[| \chain : SChain ⟹ \c : SCtx ⟹ \m : Msg ⟹ \s : State ⟹
    let bal : Money := balance s in
    let now : Nat := cur_time chain in
    let tx_amount : Money := amount c in
    let sender : Address := ctx_from c in
    let own : Address := owner s in
    let accs : Map := donations s in
    case m : Msg return Maybe Result of
      | GetFunds →
      if (own == sender) && (deadline s < now) && (goal s ≤bal) then
        Just (Pair (mkState 0z accs own (deadline s) True (goal s))
                    [Transfer bal sender])
      else Nothing : Maybe Result
      | Donate → if now ≤deadline s then
        (case (mfind accs sender) : Maybe Money return Maybe Result of
          | Just v →
            let newmap : Map := madd sender (v + tx_amount) accs in
            Just (Pair (mkState (tx_amount + bal) newmap own
                              (deadline s) (done s) (goal s)) Nil)
          | Nothing →
            let newmap : Map := madd sender tx_amount accs in
            Just (Pair (mkState (tx_amount + bal) newmap own
                              (deadline s) (done s) (goal s)) Nil))
        else Nothing : Maybe Result
      | Claim →
      if (deadline s < now) && (bal < goal s) && (∼ done s) then
        (case (mfind accs sender) : Maybe Money return Maybe Result of
          | Just v → let newmap : Map := madd sender 0z accs in
            Just (Pair (mkState (bal−v) newmap own (deadline s) (done s) (goal s))
                      [Transfer v sender])
          | Nothing → Nothing)
        else Nothing : Maybe Result |].


Make Definition receive :=
  (expr_to_term Σ' (indexify nil crowdfunding)).
```

Figure 4. The crowdfunding contract

state, then execution of the receive method gives a new state in which the balance is again consistent. Note that the receive is a "regular" Coq function and it is a shallow embedding of the corresponding crowdfunding definition (Figure 4) produced automatically by our translation.

4 Verifying Standard Library Functions

In our Coq development, we show how one can verify Acorn library code by proving λ_{smart} functions (obtained by printing the Acorn AST as λ_{smart} AST) equivalent to the corresponding functions from the standard library of Coq. In particular, we provide an example of such a procedure for certain functions on lists. The similar approach is mentioned as a strong side of Coq in comparison to Liquid Haskell [27]. In general, our framework will be applicable for verification of standard libraries of various functional languages (not even necessarily languages for smart contracts) since data types

such as lists, trees, finite maps, etc. are ubiquitous in functional programming. In addition, λ_{smart} is essentially a pure fragment of various functional general-purpose languages (ML-family, Elm) and smart contract languages (Liquidity, Simplicity, Sophia[13]) making it a good target for integration.

Figure 5 shows how we obtain the shallow embedding from the λ_{smart} List module of the Acorn standard library. We start with the concrete syntax (Figure 5a). Next, from the Acorn parser, which is a part of the Concordium infrastructure, we obtain an AST. This AST is printed to obtain a Coq representation (which we call λ_{smart}) using a simple printing procedure implemented in Haskell (Figure 5b). From the module AST in Coq we produce a shallow embedding by using the translation described in Section 2.3 and the TemplateMonad of MetaCoq to unquote all the definitions in the module. We use translateData and translateDefs function

---

[13]A functional smart contract language based on ReasonML: https://dev.aepps.com/aepp-sdk-docs/Sophia.html

to translate and unquote all the definitions of data types and functions respectively. The `translateDefs` is defined as follows:

```
Fixpoint translateDefs (Σ : global_env) (es : list (string * expr))
  : TemplateMonad unit:=
  match es with
  | [] ⇒ tmPrint "Done."
  | (name, e) :: es' ⇒
    coq_expr ← tmEval all (expr_to_term Σ(reindexify 0 e)) ;;
    print_nf ("Unquoted: " ++ name);;
    tmMkDefinition name coq_expr;;
    translateDefs Σes'
  end.
```

The `expr_to_term` Σ corresponds to $[\![-]\!]^t_\Sigma$ from Figure 3. In the Acorn AST received from the parser, the index spaces for type variables and term variables are separated. We merge them into a single index space with the `reindexify` function. Finally, `tmMkDefinition` unquotes the translated MetaCoq term and adds it to the Coq environment. After that, we can interact with the unquoted definitions as if they were written by hand.

On the shallow embedding we establish an isomorphism between Acorn lists and Coq lists by defining two functions `to_acorn` and `from_acorn` composing to identity. We can state the following: `foldr f a l = fold_right f a (from_acorn l)` where `foldr` is an Acorn function and `fold_right` comes from the standard library of Coq. Similarly for the list concatenation. Now, we can transfer properties of these functions without the need of reproving:

```
Lemma foldr_concat (A B : Set) (f : A → B → B)
                   (l l' : AcornList A) (i : B) :
  foldr f i (concat l l') = foldr f (foldr f i l') l.
Proof. autorewrite with hints;apply fold_right_app. Qed.
```

Currently, we use `autorewrite` to automate such proofs, but in the future, we consider using more principled techniques like [25].

5 The Execution Framework

In the context of blockchains smart contracts are small programs that are published to the nodes of the system and associated with some address. Calls happen when a transaction is made to this address and nodes execute the program when seeing such a transaction, which additionally can contain input parameters to the program. Smart contracts typically survive across calls and are thus long-lived stateful objects that end up being executed multiple times. In addition smart contracts interact with the blockchain in various ways, for example by making calls to other smart contracts or by transferring money owned by the smart contract into other accounts. The blockchain software thus keeps track of extra information about the smart contract: its monetary balance and the local state that the particular smart contract wishes to persist between calls.

For these reasons it does not suffice to prove only functional correctness properties if one wants to achieve strong guarantees. As an example, we would like to prove that the crowdfunding contract in Figure 4 has enough money when it attempts to pay back the funders. Here simple functional correctness is uninteresting since the crowdfunding contract just takes its own balance as an input to the function. Instead we would like a more comprehensive model of a blockchain capturing the semantics of transactions that affect state such as the balance. Such a model is given in [14] which provides a Coq formalization of a small-step operational semantics of blockchain execution. In this framework one can reason about multiple deployed and interacting contracts with the system tracking the balance and local state of each contract. This small-step semantics is used to define a trace type, and we can specify our stronger safety properties as properties that hold for any blockchain state reachable through a trace. Here we differentiate between blockchain state, which is the full state of the entire blockchain, and local state, which simply is the state that some particular contract wants the blockchain to persist for it. Logically the local state of a smart contract is one part of the blockchain state, but the blockchain state also contains information like the balance of each account in the system. By using the traces it is furthermore possible to define temporal properties.

As a running example, we continue with the crowdfunding contract from Section 3. We demonstrate how the property (P.i) (Section 3) can be transformed into a safety property and prove that the balance stored in the internal state of the contract is always less or equal to the actual balance recorded in the blockchain state.

In [14] contracts are represented as two functions `init` and `receive`. The `init` function is called when the contract is deployed and allows the contract to establish its initial local state, while the `receive` function is called after deployment when transactions are sent to the contract. These functions are provided with information about the blockchain and the `receive` function allows the contract to interact with the blockchain more actively, such as by making transactions to other accounts. The `init` and `receive` functions are partial allowing contracts to communicate that they were called with invalid parameters.

To be able to state lemmas, we need to do some extra work by wrapping the functions `init` and `receive` from Figure 4 to have compatible signatures. In the execution framework, these functions take `Chain` and `ContractCallContext` types that are generalized over the type of addresses used. For our particular contract we use "plain" inductive types `SimpleChain` and `SimpleContractCallContext` for which addresses are always natural numbers. Thus we instantiate the execution framework's addresses as natural numbers and then define conversion functions between the types from the execution framework and the simple variants used by the crowdfunding

```
module ListBase where

import Prod
import Bool

data List a = Nil [] | Cons [a, (List a)]

definition foldr a b (f :: a → b → b)
   (initVal :: b) =
   letrec go (xs :: List a) :: b =
       case xs of
          Nil → initVal
          Cons x xs' → f x (go xs')
          in go
...
```

(a) A fragment of Acorn code

```
Definition Data :=
[gdInd "List" 1 [("Nil_coq", []);
    ("Cons_coq", [(None, tyRel 0);
      (None, (tyApp (tyInd "List")
       (tyRel 0)))])] false].

Definition Functions :=
[("foldr", eTyLam "A" (eTyLam "A"
  (eLambda "x" (tyArr (tyRel 1)
  (tyArr (tyRel 0) (tyRel 0)))
  (eLambda "x" (tyRel 0)
  (eLetIn "f" (eFix "rec" "x" . . .)))));
  . . .
```

(b) A fragment of λ_{smart} AST (deep embedding)

```
Import AcornProd.
Import AcornBool.

Run TemplateProgram (translateData Data).

Definition gEnv := StdLib.Σ ++ Data ++
    AcornBool.Data ++ AcornProd.Data.
Run TemplateProgram (translateDefs gEnv
    Functions).
Print foldr.
(* fun (A A0 : Set)(x : A → A0 → A0)
     (x0 : A0) ⇒
  fix rec (x1 : List A) : A0 :=
    match x1 with
    | @Nil_coq _ ⇒ x0
    | @Cons_coq _ x2 x3 ⇒
       x x2 (rec x3)
    end *)
```

(c) Shallow embedding

Figure 5. Translating Acorn list functions to Coq

contract. In effect the resulting functions have the following types:

```
wrapped_init : Chain → ContractCallContext
              → Setup
              → option State_coq

wrapped_receive : Chain → ContractCallContext
              → State_coq → option Msg_coq
              → option (State_coq * list ActionBody)
```

The Setup type here is just for packing together parameters to the init function: deadline and goal. Another requirement for using the execution framework is to provide instances for serialisation/deserialisation of the local state (State_coq) and messages (Msg_coq). These instances can be automatically generated by the execution framework for the simple non-recursive data types used in the crowdfunding contract. With these instances in place, we can put together init and receive to define a contract:

```
Definition cf_contract : Contract Setup Msg_coq State_coq :=
  build_contract wrapped_init init_proper wrapped_receive
      receive_proper.
```

Here init_proper and receive_proper are proofs showing that the wrapped functions respect extensional equality on the input parameters.

Now we are ready to formulate a safety property of the crowdfunding contract:

```
Lemma cf_balance_consistent bstate cf_addr lstate :
  reachable bstate →
  env_contracts bstate cf_addr = Some (cf_contract :
      WeakContract) →
  cf_state bstate cf_addr = Some lstate →
  consistent_balance lstate.
```

One can read this lemma as follows: for any reachable blockchain state bstate, for an instance of the crowdfunding contract deployed at the address cf_addr with some local state lstate, the balance recorded in the local state is consistent with the map of individual contributions.

Next, we state and prove the following lemma:[14]

```
Lemma cf_backed_after_block {ChainBuilder : ChainBuilderType}
        prev hd acts new cf_addr lstate :
builder_add_block prev hd acts = Some new →
env_contracts new cf_addr = Some (cf_contract : WeakContract) →
cf_state new cf_addr = Some lstate →
(account_balance (env_chain new) cf_addr>=balance_coq lstate)%Z.
```

This lemma says that after adding a new block,[15] for any instance of the crowdfunding contract deployed at the address cf_addr, the actual contract balance (the one recorded in the blockchain state) is greater or equal to the balance recorded in the local state (used in all operations of the contract's "logic"). In essence our integration with the execution framework allows us to prove that the contract tracks its own balance correctly even in the face of potential reentrancy or nontrivial interactions with other contracts and accounts. This is unlike previous work such as [20] which only considers a single contract in isolation, or [5], which focuses on functional correctness properties. This lemma in combination with cf_balance_consistent also gives a formal

---

[14]The lemma presented in the paper is a corollary of a more general theorem which generalises over any outstanding actions, somewhat similarly to a theorem about the Congress contract in [14]. We leave out the details here, but full proofs of all the lemmas from this section is available in our Coq development.

[15]For our purpose a block can be thought of as just a list of transactions.

argument that the contract's account has enough money to cover all individual contributions.

Having our shallow embedding integrated with the execution framework, one can imagine many more important safety and temporal properties of the contract. For example, if the contract is funded, there will ever be at most one outgoing transaction (to the owner), or if the contract is not funded and the deadline has passed users can get their contributions back. We leave this for future work.

6 Related Work

In this work, we focus on modern smart contract languages based on a functional programming paradigm. In many cases, various small errors in smart contracts can be ruled out by the type systems of these languages. Capturing more serious errors requires employing such techniques as deductive verification (for verification of concrete contracts) and formalisation of meta-theory (e.g. to ensure the soundness of type systems). The formalisation of the Simplicity language [15] features well-developed meta-theory, including a formalisation of the operational semantics of the Bit Machine allowing for reasoning about computational resources. But this formalisation does not focus on using the shallow embedding for proving properties of smart contracts and Simplicity is a low-level language in comparison to λ_{smart} (Simplicity is a non-Turing-complete first-order language and does not feature algebraic data types).

The work on Scilla [20] focuses on verification of concrete smart contracts in Coq. It considers a crowdfunding smart contract example translated into Coq by hand, and the correspondence to Scilla's meta-theory is not clear. A recent paper on Scilla [21] gives a formal definition of the language semantics, but does not feature mechanised proofs.

The formalisation of the Plutus Core language [7] covers the meta-theory of System F_ω^μ — a polymorphic lambda calculus with higher-order kinds and iso-recursive types. The main difference with λ_{smart} is the absence of "native" inductive types. Another work on Plutus [17] shows how to compile inductive types into System F_ω^μ. The compilation procedure is type-preserving (which follows from the intrinsic encoding used in the formalisation); computational soundness is left as future work.

The Michelson language formalisation [5] defines an intrinsic encoding of the language expression along with its interpreter in Coq. The Coq development uses a weakest precondition calculus on deeply-embedded Michelson expressions for smart contract verification. The semantics of the Liquidity language given in [6] provides the rules for compilation to Michelson, but there is no corresponding formalisation in a proof assistant. The Sophia smart contract language also belongs to the family of functional smart contract languages based on ReasonML, but there is no corresponding formalisation available.

There are several well-developed formalisations of variations of System F [11, 12, 24, 28]. Among these, [11] features an interpreter and helped us to shape our representation of the language in Coq.

The purpose of the `Program` tactic in Coq [16, 22] is to embed a functional language into Coq allowing for writing specifications in types. Our work can be seen as the first step towards making a certified version of such a tactic.

Finally, meta-programming techniques have also been shown to be useful in the dependently typed setting in other proof assistants: Agda's Reflection library [26], meta-programming frameworks in Lean [9] and Idris [8] employ techniques similar to MetaCoq.

7 Conclusion and Future Work

We have presented the ConCert smart contract verification framework. An important feature of our approach is the ability to both develop a meta-theory of a smart contract language and to conveniently reason about particular programs (smart contracts). We proved soundness theorems relating meta-theory of the smart contract language with the embedding. Such an option is usually not available for source-to-source translations. We applied our approach to the development of an embedding of the λ_{smart} smart contract language and provided a verification example of a crowdfunding contract starting from the contract's AST. We also demonstrated how our framework can be used to verify "standard library" functions common to functional programming languages by proving them equivalent to Coq standard library functions. Moreover, we integrated the shallow embedding with the smart contract execution framework which gives access to properties related to the interaction of smart contracts with the underlying blockchain and with each other. Our work addresses a number of future work points from a recent Scilla paper [21]: we provide a shallow embedding, integrate it with a reasoning framework for safety and temporal properties and implement a reference evaluator in Coq. To the best of our knowledge, the ConCert framework together with the execution model described in Section 5 is the first development allowing to verify functional smart contracts against a blockchain execution model by automatic translation to the shallow embedding.

Our framework is general enough to be applied to other functional smart contract languages. We consider benchmarking our development by developing "backends" for translation of other languages (e.g. Liquidity, Simplicity, etc.). Extending the formalisation of the λ_{smart} language meta-theory is also among our goals for the framework. An important bit of the meta-theory is the cost semantics allowing for reasoning about "gas". We would like to give a cost semantics for the deep embedding and explore how it can be extended on the shallow embedding. Another line of future work is extending ACORN with specification annotations

Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters

similar to the Russell language [22] used for Coq's `Program` tactic. That would allow programmers to specify properties of smart contract that become obligations in the resulting Coq translation.

Acknowledgments

References

[1] Nada Amin and Tiark Rompf. Type Soundness Proofs with Definitional Interpreters. In *POPL17*, pages 666–679. ACM, 2017.

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

[3] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards Certified Meta-Programming with Typed Template-Coq. In *ITP18*, volume 10895 of *LNCS*, pages 20–39, 2018.

[4] Abhishek Anand and Greg Morrisett. Revisiting Parametricity: Inductives and Uniformity of Propositions. CoqPL18, 2018.

[5] Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Zhenlei Pesin, and Julien Tesson. Mi-Cho-Coq, a framework for certifying Tezos Smart Contracts. FMBC19, 2019.

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

[7] James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. System F in Agda, for fun and profit. In *MPC19*, 2019.

[8] David Christiansen and Edwin Brady. Elaborator Reflection: Extending Idris in Idris. In *ICFP 2016*, pages 284–297. ACM, 2016.

[9] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, 1(ICFP):34:1–34:29, 2017.

[10] Yannick Forster and Fabian Kunze. A certifying extraction with time bounds from Coq to call-by-value λ-calculus. *CoRR*, abs/1904.11818, 2019.

[11] Jacques Garrigue. A Certified Implementation of ML with Structural Polymorphism. In Kazunori Ueda, editor, *Programming Languages and Systems*, pages 360–375, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[12] András Kovács. System F Omega. https://github.com/AndrasKovacs/system-f-omega.

[13] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *CCS16*, pages 254–269, 2016.

[14] Jakob Botsch Nielsen and Bas Spitters. Smart Contract Interactions in Coq. FMBC19, 2019.

[15] Russell O'Connor. Simplicity: A New Language for Blockchains. PLAS17, pages 107–120. ACM, 2017.

[16] Catherine Parent. Synthesizing proofs from programs in the calculus of inductive constructions. In Bernhard Möller, editor, *Mathematics of Program Construction*, pages 351–379, 1995.

[17] Michael Peyton Jones, Vasilis Gkoumas, Roman Kireev, Kenneth MacKenzie, Chad Nester, and Philip Wadler. Unraveling recursion: compiling an IR with recursion to System F. In *MPC19*, 2019.

[18] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, Dec 1998.

[19] Ilya Sergey and Aquinas Hobor. A Concurrent Perspective on Smart Contracts. In *Financial Cryptography and Data Security*, pages 478–493, 2017.

[20] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a Smart Contract Intermediate-Level LAnguage. *CoRR*, abs/1801.00687, 2018.

[21] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan. Safer Smart Contract Programming with Scilla. In *OOPSLA19*, 2019.

[22] Matthieu Sozeau. Subset coercions in Coq. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs*, pages 237–252. Springer Berlin Heidelberg, 2007.

[23] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total Haskell is Reasonable Coq. CPP18, pages 14–27. ACM, 2018.

[24] Sandro Stucki. A formalization of the polymorphic lambda calculus extended with iso-recursive types . https://github.com/sstucki/system-f-agda.

[25] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. Equivalences for Free: Univalent Parametricity for Effective Transport. *Proc. ACM Program. Lang.*, 2(ICFP):92:1–92:29, July 2018.

[26] Paul van der Walt and Wouter Swierstra. Engineering proof by reflection in Agda. In Ralf Hinze, editor, *Implementation and Application of Functional Languages*, pages 157–173. Springer, 2013.

[27] Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. A Tale of Two Provers: Verifying Monoidal String Matching in Liquid Haskell and Coq. *SIGPLAN Not.*, 52(10):63–74, September 2017.

[28] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System fc with explicit kind equality. In *ICFP13*, pages 275–286. ACM, 2013.