Towards Formally Specifying and Verifying Smart Contract Upgrades in Coq

Derek Sorensen
University of Cambridge, *Department of Computer Science and Technology*, UK

Abstract

Smart contract upgrades are costly from a verification perspective and can be a meaningful source of vulnerabilities when done incorrectly. Unfortunately, there is no established, formal framework through which one can reason about contracts as they undergo upgrades, though much work has been done to verify standalone smart contracts. Instead, one must repeat the full verification process for each contract upgrade, something which relies heavily on fallible intuition, can lead to unexpected vulnerabilities, and drives up the cost of formally verifying smart contracts. We propose a formal framework for contract upgrades in ConCert, a Coq-based smart contract verification tool. Central to this framework is our notion of a *contract morphism*, a theoretical tool which we introduce to formally encode structural relationships between smart contracts, and with which we can formally specify and verify an upgraded contract relative to its previous versions. We argue that ours is a natural framework for specifying and verifying contract upgrades, and hope to offer a first step towards rigorous, efficient specification and verification of contract upgrades.

2012 ACM Subject Classification Theory of computation → Program verification

Keywords and phrases smart contract verification, formal methods, interactive theorem prover, smart contract upgrades

Digital Object Identifier 10.4230/OASIcs.FMBC.2024.1

Supplementary Material *Software*: `https://github.com/differentialderek/FinCert`

1 Introduction

Faulty upgrades are a meaningful source of smart contract vulnerabilities. Costly attacks such as those on Uranium Finance (2021) [8], NowSwap (2021) [4], and Nomad (2022) [7, 9], totaling 241 million USD in lost assets, are a few of many examples of contracts attacked after an erroneous upgrade. Furthermore, because verifying software is time, labor, and resource intensive, it can be difficult to justify formally verifying software which may be upgraded quickly or frequently—a problem shared with other verified software, *e.g.* [16, 21]. Both of these factors limit the effectiveness of formal methods to address security issues in real-world software, inhibiting verification as business and security propositions [18].

What is needed is a practical and formal framework through which to specify and verify contract upgrades. As it stands we have no such framework apart from repeating the formal specification and verification process on a new contract version. Not only are upgrades costly from a verification perspective, as we have no good way of reusing much of the verification work on previous contract versions, but incorrect specifications are themselves a meaningful source of contract vulnerabilities [19]. Thus each time a specification is made from scratch we risk introducing errors of incorrect specification.

To mitigate these issues we introduce a formal framework for specifying and verifying contract upgrades, through which we can reuse formal specification and proof on previous contract versions. This framework relies on the notion of a *contract morphism*, a theoretical tool we introduce that formally encodes structural relationships between smart contracts, and with which we can specify and reason about the structure and behavior of an upgraded contract relative to its previous versions. We argue that this is a natural framework for

5th International Workshop on Formal Methods for Blockchains (FMBC 2024).
Editors: Bruno Bernardo and Diego Marmsoler; Article No. 1; pp. 1:1–1:14
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

specifying and verifying contract upgrades, one which could decrease the cost of formally verifying contract upgrades as well as the risk of introducing vulnerabilities due to incorrect specification.

We proceed as follows. In §2, we survey related work. In §3, we introduce *contract morphisms* as a formal tool to specify and verify contract upgrades. In §4 we give two examples of formally specifying a contract upgrade with contract morphisms. In §5 we discuss formal verification with contract morphisms. We conclude in §6.

2 Related Work

In the realm of smart contracts there is limited formal work on formal reasoning about contract upgrades. Previous work [3, 6] proposes paradigm-shifting methods to either attach formal proofs to smart contracts and their upgrades, which are verified by the chain, or to trust a canonical third party to verify all contract upgrades before deployment. Unfortunately this work is likely impractical, as both solutions require substantial paradigm shifts or re-engineering of blockchain ecosystems. The latter also arguably contradicts the permissionless ethos of blockchain ecosystems by mandating a trusted third party.

In the context of software more generally, much work has gone into ensuring that software upgrades are carried out safely with formal methods [10, 12, 21]. Recent work has begun to address the issue of adapting formal proofs in a proof assistant to changes in software in order to lower the cost of formally verified software which may undergo regular upgrades [16]. This problem is complicated by the computable nature of proofs in proof assistants like Coq; chosen data types strongly influence the structure of proofs, making adaptation difficult [11]. A notable contribution to this work is Ringer *et al.*'s work on *proof repair* [14, 15], which seeks to relate a new program version to the old—by type equivalences or by comparing inductive structures—and thereby reuse previously-completed proofs on the updated code.

Drawing on this previous work, particularly Ringer *et al.*'s idea of reusing formal proofs by way of structural similarities between programs, our goal is to provide a framework for using formal methods to formally specify and verify smart contract upgrades. Contract morphisms (§3) will be our primary theoretical tool for specifying and verifying contract upgrades. Their purpose is to formally encode a structural relationship between smart contracts which can be used for both formal specification and proof reuse. With contract morphisms we address the problem of formal reasoning about contract upgrades, but in contrast to previous work on the subject our proposed framework does not require the paradigm-shifting reengineering of blockchain systems in order to be used.

Finally, we note that for smart contracts there is a distinction between contract upgrades and contract *upgradeability*. Some contracts come with a predefined logic to handle upgrades and avoid hard forks, the most popular of these on Ethereum being the Diamond framework [13]. However, they are complicated contracts as their specifications include the upgradeability functionality and governance, as well as the functionality of a given version of the contract. We will only consider upgrades via hard forks in this paper, leaving the question of rigorous formal specification and verification of upgradeable contracts to future work.

3 Contract Morphisms

In what follows we define *contract morphisms*, a theoretical tool which codifies formal relationships between smart contracts. In later sections we use them to formally specify and verify contract upgrades. We argue that this provides our desired formal framework.

3.1 Morphisms of Pure Functions

Before focusing on the specific case of smart contracts, we consider the more general case of programs formalized as pure functions. Take types A, A' and B, B', and two functions $p : A \to B$ and $q : A' \to B'$. A *morphism* from p to q is a or a pair of functions f_i and f_o which form a commutative square,

$$\begin{array}{ccc} A & \xrightarrow{f_i} & A' \\ {\scriptstyle p}\downarrow & /\!\!/ & \downarrow{\scriptstyle q} \\ B & \xrightarrow{f_o} & B' \end{array}$$

i.e. for which

$$q \circ f_i = f_o \circ p.$$

Together, we call f_i and f_o the morphism

$$f : p \to q.$$

Via f_i and f_o, the commutative square like the above maps inputs and outputs of p to inputs and outputs of q. If p and q are programs (in particular, pure functions), we can also interpret this as execution traces of p to execution traces of q, such that transforming the inputs of p into those of q with f_i, and then applying q is the same as applying p first and then transforming the outputs over f_o.

We can define composition of morphisms easily as the composition of commutative squares. That is, given functions p, q, and r, and morphisms

$$f' : p \to q \text{ and } f'' : q \to r,$$

we can define a morphism $f := f'' \circ f' : p \to r$ by the outer square of the following diagram,

$$\begin{array}{ccccc} A & \xrightarrow{f'_i} & A' & \xrightarrow{f''_i} & A'' \\ {\scriptstyle p}\downarrow & /\!\!/ & \downarrow{\scriptstyle q} & /\!\!/ & \downarrow{\scriptstyle r} \\ B & \xrightarrow{f'_o} & B' & \xrightarrow{f''_o} & B'' \end{array}$$

which is commutative if each of the inner squares are commutative. Note that composition is associative, assuming the underlying functions are associative, and that we have the obvious identity morphism $f_{\text{id}} : p \to p$ given by $f_i, f_o := id$,

$$\begin{array}{ccc} A & \xrightarrow{\text{id}} & A \\ {\scriptstyle p}\downarrow & /\!\!/ & \downarrow{\scriptstyle p} \\ B & \xrightarrow{\text{id}} & B \end{array}$$

which commutes trivially. Thus given a well-defined class of functions, which in our case will be smart contracts modeled in Coq by pure functions, we have a category on those functions with morphisms given by commutative squares on those pure functions.

In the coming sections, given a morphism $f : p \to q$, we might consider the case that q is an upgraded version of p. Because f relates execution traces of q to those of p, we will see this can be used to reason formally about q in terms of p, both in specification and verification.

3.2 Contract Morphisms in ConCert

In ConCert, a Coq-based tool for smart contract verification which models the execution semantics of third-generation blockchains [2] and features verified extraction to various blockchains [1], smart contracts are formalized with a `Contract` type as a pair of pure, stateful functions `init` and `receive`. The `init` function governs contract initialization and the `receive` function governs contract calls. The `Contract` type is polymorphic, parameterized by four types: `Setup`, `Msg`, `State`, and `Error` which, respectively, govern the data necessary for contract initialization, contract calls, contract storage, and contract errors.

For a contract

```
C : Contract Setup Msg State Error
```

the type signatures of each component function `(init C)` and `(receive C)` are given as follows, where the types `Chain` and `ContractCallContext` are ConCert-specific types used to model the underlying blockchain and context.

```
init C : Chain → ContractCallContext → Setup → result State Error.

receive C : Chain → ContractCallContext → State → option Msg →
              result (State ∗ list ActionBody) Error.
```

Listing 1 Type signature of the `init` and `receive` functions, respectively, of a smart contract in ConCert.

Now consider contracts `C1` and `C2`,

```
C1 :  Contract Setup1 Msg1 State1 Error1

C2 :  Contract Setup2 Msg2 State2 Error2.
```

We define a data type of *morphisms* between contracts `C1` and `C2`,

```
ContractMorphism C1 C2.
```

This data type consists firstly of four *component functions* between the contract types of `C1` and `C2`—the `Setup`, `Msg`, `State`, and `Error` types respectively.

- `setup_morph :  Setup1 -> Setup2`
- `msg_morph   :  Msg1   -> Msg2`
- `state_morph :  State1 -> State2`
- `error_morph :  Error1 -> Error2.`

We can use these component functions to make commutative squares like those we saw in §3.1 for each of the `init` and `receive` functions. For `init`, the horizontal arrows of the squares are given by the functions `mA_init` and `mB_init`. For `receive`, the horizontal arrows are given by the functions `mA_recv` and `mB_recv`. See Listing 2 for the definition of these functions in terms of the four component functions given above.

$$
\begin{array}{ccc}
A_{\texttt{init}} & \xrightarrow{\texttt{mA\_init}} & A'_{\texttt{init}} \\
{\scriptstyle\texttt{init}}\downarrow & /\!/ & \downarrow{\scriptstyle\texttt{init}'} \\
B_{\texttt{init}} & \xrightarrow{\texttt{mB\_init}} & B'_{\texttt{init}}
\end{array}
\qquad
\begin{array}{ccc}
A_{\texttt{recv}} & \xrightarrow{\texttt{mA\_recv}} & A'_{\texttt{recv}} \\
{\scriptstyle\texttt{receive}}\downarrow & /\!/ & \downarrow{\scriptstyle\texttt{receive}'} \\
B_{\texttt{recv}} & \xrightarrow{\texttt{mB\_recv}} & B'_{\texttt{recv}}
\end{array}
$$

```
(* functions to form a commutative square on init *)
mA_init :=
    fun (c : Chain) (ctx : ContractCallContext) (s : Setup) =>
    (c, ctx, setup_morph s).
mB_init := fun (res : result State Error) =>
    match res with
    | Ok init_st => Ok (state_morph init_st)
    | Err e => Err (error_morph e)
    end.

(* functions to form a commutative square on receive *)
mA_recv := fun (c : Chain) (ctx : ContractCallContext)
    (st : State) (op_msg : option Msg) =>
    (c, ctx, state_morph st, option_map msg_morph op_msg).
mB_recv := fun (res : result (State * list ActionBody) Error) =>
    match res with
    | Ok (init_st, nacts) => Ok (state_morph init_st, nacts)
    | Err e => Err (error_morph e)
    end.
```

Listing 2 The functions which we use for the horizontal arrows of a pair of commutative squares `f_init : init C1 -> init C2` and `f_recv : receive C1 -> receive C2`, respectively, in the definition of a contract morphism.

The functions defined above give us squares, but to finish the definition of contract morphisms we need these squares to commute. Thus our definition includes two coherence conditions, one for the `init` square and one for the `receive` square, which are given as follows.

```
(* The coherence condition that makes the init square commute *)
init_coherence: forall c ctx s,
(match (init C1 c ctx s) with
    | Ok init_st => Ok (state_morph init_st)
    | Err e => Err (error_morph e)
    end) =
(init C2 c ctx (setup_morph s)).

(* The coherence condition that makes the receive square commute *)
recv_coherence : forall c ctx st op_msg,
(match (receive C1 c ctx st op_msg) with
    | Ok (new_st, new_acts) => Ok (state_morph new_st, new_acts)
    | Err e => Err (error_morph e)
    end) =
(receive C2 c ctx (state_morph st) (option_map msg_morph op_msg)).
```

Thus a contract morphism

`m : ContractMorphism C1 C2`

is defined as a pair of commutative squares, each of which are morphisms between the respective `init` and `receive` functions of each contract. We give the formal definition of a contract morphism in Listing 3.

As the name *morphism* suggests, we should expect contract morphisms to behave like morphisms in a well-defined category. That is, we should have an associative composition operation on morphisms, and for every contract `C` should have an identity morphism

```
Record ContractMorphism
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2) :=
    build_contract_morphism {
        (* the components of a morphism *)
        setup_morph : Setup1 → Setup2 ;
        msg_morph  : Msg1   → Msg2  ;
        state_morph : State1 → State2 ;
        error_morph : Error1 → Error2 ;
        (* coherence conditions *)
        init_coherence : forall c ctx s,
            result_functor state_morph error_morph (init C1 c ctx s) =
            init C2 c ctx (setup_morph s) ;
        recv_coherence : forall c ctx st op_msg,
            result_functor (fun '(st, l) ⇒ (state_morph st, l))
                error_morph
                (receive C1 c ctx st op_msg) =
            receive C2 c ctx (state_morph st)
                (option_map msg_morph op_msg) ;
}.
```

Listing 3 The formal definition of a contract morphism in ConCert, consisting of four component functions and two coherence conditions, which together give a pair of commutative squares.

`id_C : ContractMorphism C C`

with which composition is trivial.

Indeed, this is the case. We can compose morphisms by composing the morphism component functions. We have two results,

`compose_init_coh` and `compose_recv_coh`,

which show that coherence of the composed morphism follows from the coherence conditions of each individual morphism. These results simply show that commutative squares compose, as we saw in §3.1, giving us a well-defined composition function `compose_cm`.

```
compose_cm : forall C1 C2 C3
    (g : ContractMorphism C2 C3) (f : ContractMorphism C1 C2) : ContractMorphism C1 C3.
```

We also have a proof that composition is associative, drawing on the associativity of component functions, and we have the obvious identity morphism, given by four identity component functions, such that composition with the identity is trivial.

```
Definition id_cm (C : Contract Setup Msg State Error) :
  ContractMorphism C C := {|
      (* components *)
      setup_morph := id ;
      msg_morph   := id ;
      state_morph := id ;
      error_morph := id ;
      (* coherence conditions *)
      init_coherence := init_coherence_id C ;
      recv_coherence := recv_coherence_id C ;
  |}.
```

This gives us a well-defined category **Contracts** of smart contracts, with objects given by the `Contract` type and morphisms given by the `ContractMorphism` type.

Note that in many categories, *e.g.* the categories of sets, topological spaces, or groups, morphisms are structure-preserving functions. So too for us. The existence of a morphism

```
f : ContractMorphism C1 C2
```

indicates a structural and mathematical relationship between contracts `C1` and `C2`, in particular relating their execution traces via the four component morphisms. As we will see, this relationship can be exploited to prove theorems about one contract in terms of another contract, something which we will do here in the case of contract upgrades and upgradeability.

In many categories there are also different classes of morphisms, including injections (embeddings, monomorphisms), surjections (quotients, epimorphisms), and isomorphisms. Injections, or embeddings, typically preserve the structure of the domain faithfully within the codomain, essentially identifying a copy of the domain within the codomain. Surjections typically represent a compression of some kind, and the information lost in the compression can frequently be described by a kernel object. As we will see, we also have injective and surjective contract morphisms, which are given when the four component functions are, respectively, injective or surjective, and which follow analogous intuitions.

4 Morphisms to Formally Specify and Verify Contract Upgrades

Our goal now is to use contract morphisms as a tool to formally specify and verify contract upgrades in ConCert. Consider a contract upgrade from the perspective of a formal specification. Contracts are usually upgraded with a goal that relates the new to the previous contract version, whether it be to patch a bug, add functionality, or improve contract features. Thus the new specification relates to the old—it should eliminate a vulnerability but preserve all other functionality, be backwards compatible while adding functionality, or make improvements such as greater gas-efficiency without deviating from the behavior of the previous contract version. Of course, in practice an upgraded contract is not formally specified in relation to an older version, but rather by altering the old specification into the new, or simply starting from scratch and writing a new specification by hand. As discussed in §1, this can be a source of vulnerabilities.

In this section, we will formally specify contract upgrades in two examples using contract morphisms. The advantage of using morphisms is that we are able to clearly articulate the intent of an upgrade in the formal specification by way of a morphism in such a way that formal verification consists of producing a morphism between the updated contract implementation and a previous version which meets the required specification.

▶ **Example 1** (Swap Contract Upgrade)**.** Consider a smart contract `C1` that prices and executes trades, *e.g.* a decentralized exchange (DEX) or an automated market maker (AMM) [22]. Suppose that we wish to upgrade `C1` to a contract `C2` so that it calculates trades at higher precision by a factor of ten, meaning that the internal token balances in storage have one more decimal place, and the trade calculation is able to calculate at one decimal place greater in precision. Then in ConCert our contract `C1` will have a storage type which keeps track of internal token balances, exposed by a function `get_bal`.

```
Context { storage : Type } { get_bal : storage → N }.
```

It will also have a `TRADE` entrypoint which accepts a payload of some type, `trade_data`, characterized by an entrypoint type, `entrypoint`, and an associated typeclass, `Msg_Spec`.

```
Class Msg_Spec (T : Type) := {
    (* the trade entrypoint *)
    trade : trade_data → T ;
    (* for any other entrypoint types *)
    other : other_entrypoint → option T ;
}.

(* We assume an entrypoint conforming to Msg_Spec *)
Context { entrypoint : Type } `{ e_msg : Msg_Spec entrypoint }.
```

Listing 4 We assume an entrypoit type `entrypoint`, characterized by a typeclass `Msg_Spec`, which includes a trade function `trade`.

Now assume that `C1` has some internal function `calculate_trade` that it uses to calculate how many tokens will be traded out for a given contract call to the `TRADE` entrypoint. The trade quantity, internal token balances, and the `calculate_trade` function will all be accurate up to some decimal place, commonly 9 in the wild, formalized in the following specification, `spec_trade`, of `C1`.

```
(* the specification of C1's trading functionality with regards to the
    calculate_trade function *)
Definition spec_trade : Prop :=
    forall cstate chain ctx trade_data cstate' acts,
    (* for any successful call to C1's trade entrypoint, *)
    receive C1 chain ctx cstate (Some (trade trade_data)) =
    Ok(cstate', acts) →
    (* the balance in storage updates according to the
       calculate_trade function *)
    get_bal cstate' =
    get_bal cstate + calculate_trade (trade_qty trade_data).
```

Listing 5 The formalized proposition that `C1` uses `calculate_trade` to price trades.

The property of Listing 5, `spec_trade`, is a specification with regards to which `C1` is assumed to be correct.

Now we wish to upgrade `C1` to a new contract `C2` such that `C2` calculates trades and keeps balances at one decimal place higher of accuracy. We will first have a specification for `C2` which is analogous to `spec_trade` in Listing 5, which says that `C2` uses some new function, `calc_trade_precise`, to calculate its trades.

```
(* The specification of C2's trading functionality with regards to the
    calculate_trade_precise function. This is analogous to spec_trade *)
Definition spec_trade_precise : Prop :=
    forall cstate chain ctx trade_data cstate' acts,
    (* for a successful call to C2's trade entrypoint, *)
    receive C2 chain ctx cstate (Some (trade trade_data)) = Ok (cstate', acts) →
    (* the balance in storage updates according to the
       calculate_trade_precise function *)
    get_bal cstate' =
    get_bal cstate +
    calculate_trade_precise (trade_qty trade_data).
```

Listing 6 The formalized proposition that `C2` uses `calculate_trade_precise` to price trades.

Our goal now is to use a contract morphism to complete the formal specification of `C2` in terms of `C1`. Our specification is this: A correct implementation of the upgraded contract `C2` must satisfy `spec_trade_precise` and be accompanied by a contract morphism

```
f : ContractMorphism C2 C1
```

with the following five properties, stated formally in Listing 7:

1. `msg_morph f` rounds down the precision of messages to `trade` by a factor of `10`
2. `msg_morph f` is the identity morphism on all messages aside from messages to `trade`
3. `state_morph f` rounds down on the balances kept in storage exposed by `get_bal`
4. `error_morph f` and `setup_morph f` are the respective identity functions

```
(* FORMAL SPECIFICATION:
   An upgrade C2 must admit a morphism
   f : ContractMorphism C2 C1
   with the following properties: *)

(* 1. msg_morph f rounds trades down when it maps inputs of the receive function *)
Definition f_recv_input_rounds_down
   (f : ContractMorphism C2 C1) : Prop :=
   forall t', exists t,
   (msg_morph C2 C1 f) (trade t') = trade t ∧
   trade_qty t = (trade_qty t') / 10.

(* 2. msg_morph f only affects the trade entrypoint *)
Definition f_recv_input_other_equal
   (f : ContractMorphism C2 C1) : Prop :=
   forall msg o,
   (* for calls to all other entrypoints, *)
   msg = other o →
   (* f is the identity *)
   option_map (msg_morph C2 C1 f) (other o) = other o.

(* 3. state_morph f rounds down on the storage *)
Definition f_state_morph (f : ContractMorphism C2 C1) : Prop :=
   forall st, get_bal (state_morph C2 C1 f st) = (get_bal st) / 10.

(* 4. error_morph f and setup_morph f are the identity functions *)
Definition f_recv_output_err (f : ContractMorphism C2 C1) : Prop :=
   (error_morph C2 C1 f) = id.

Definition f_init_id (f : ContractMorphism C2 C1) : Prop :=
   (setup_morph C2 C1 f) = id.
```

Listing 7 The formal specification of the upgrade from `C1` to `C2`.

The meaning of a morphism `f` satisfying the above conditions, as a specification, is in the *coherence conditions* of `f`. We know that every possible execution trace of `C2` has a corresponding execution trace in `C1`, and we know that the input messages are identical except that `C2` accepts trades at a higher level of precision. The coherence conditions also tell us that the state of `C2` is always related to the analogous state of `C1`, expressed in the function `state_morph`. With regards to the trading functionality of our new contract `C2`, we know that the balance kept in the storage of `C2`, which is affected by trades, will always be identical to the analogous balance of `C1` after rounding down, which we can formally prove.

```
Theorem rounding_down_invariant bstate caddr
    (trace : ChainTrace empty_state bstate):
    (* Forall reachable states with contract at caddr, *)
    env_contracts bstate caddr = Some (C2 : WeakContract) →
    (* cstate is the state of the contract AND *)
    exists (cstate' cstate : storage),
    contract_state bstate caddr = Some cstate' ∧
    (* cstate is contract-reachable for C1 AND *)
    cstate_reachable C1 cstate ∧
    (* such that for cstate, the state of C1 in bstate,
       the balance in cstate is rounded-down from the
       balance of cstate' *)
    get_bal cstate = (get_bal cstate') / 10.
```

Listing 8 All reachable states of `C2` round down to their corresponding states in `C1`.

Most importantly, `f` guarantees a relationship between the trading functionality of `C2` and that of `C1`: `C2` emulates the exact same trading behavior as `C1` after rounding down one decimal place in precision. This means that `C2` does not introduce any novel vulnerabilities relating to trades and balances not extant to `C1`. In particular, a proof of this fact would have prevented the attacks on Uranium Finance [8], NowSwap [4], and Nomad [7].

Moving on, note that `f` of Example 1 was directed from `C2` to `C1`. The coherence conditions of `f` forced all execution traces of `C2` to conform to a pattern set by `C1`, which is precisely what lets us make the claim that we haven't introduced any new behaviors regarding trading functionality to `C2` aside from the increase in precision. Morphisms directed in the opposite direction can also be used in specification. Rather than classifying all possible execution traces of the upgrade, in this case a morphism proves that certain desired behavior exists within the contract. We illustrate with an example of specifying backwards compatibility.

▶ **Example 2** (Backwards Compatibility)**.** Consider contracts `C1` and `C2`, where `C2` is again an upgrade of `C1`, and suppose that we wish to show that `C2` is backwards compatible with `C1`. The intent of this upgrade is that the full functionality of `C1` be present within `C2`. We show this by embedding `C1` into `C2` via an injective contract morphism.

We illustate with a simple example of a counter contract `C1` which keeps some `n : N` in storage and has one entrypoint `incr` that increments the natural number in storage by `1`. `C1` is upgraded to `C2`, which in addition to an entrypoint to increment the natural number in storage also includes a `decr` entrypoint to decrement the natural number in storage by `1`.

```
Inductive entrypoint1 := | incr (u : unit).
Inductive entrypoint2 := | incr' (u : unit) | decr (u : unit).
```

Listing 9 The entrypoint types of `C1` and `C2`, respectively.

We prove that `C2` is backwards compatible with `C1` by defining a contract morphism

$$\texttt{f : ContractMorphism C1 C2}$$

with the following component functions.

```
Definition msg_morph (e : entrypoint1) : entrypoint2 :=
    match e with | incr _ ⇒ incr' tt end.
Definition setup_morph : setup → setup  := id.
Definition state_morph : storage → storage := id.
Definition error_morph : error → error  := id.
```

These component functions do the obvious thing—send calls to the increment entrypoint of `C1` to the increment entrypoint of `C2` with the same payload, and do nothing otherwise. And `f` is an embedding since each of its component functions are manifestly injective, which we can formally prove.

```
Lemma f_is_embedding : is_inj_cm f.
```

Again, the meaning of `f` as a specification is in its coherence conditions. Any reachable state of `C1` necessarily has an analagous reachable state of `C2` which is fully structure preserving: if we were to only use the functionality of `C2` which it inherits from `C1`, we would get identical contract behavior to `C1`. We have a formal proof of this result.

```
Theorem injection_invariant bstate caddr
    (trace : ChainTrace empty_state bstate):
    env_contracts bstate caddr = Some (C1 : WeakContract) →
    (* Forall reachable states cstate of C1,
       there's a corresponding reachable state
       cstate' of C2, related by the injection *)
    exists (cstate' cstate : storage),
    contract_state bstate caddr = Some cstate ∧
    (* cstate' is a contract-reachable state of C2 *)
    cstate_reachable C2 cstate' ∧
    (* .. equal to cstate *)
    cstate' = cstate.
```

Listing 10 `C2` is backwards compatible with `C1` via the embedding `f`.

This is a toy example, but in practice specifying a new contract which is backwards compatible to the old in this strong sense may not be straightforward. Via contract embeddings, contract morphisms give us a way of formally specifying and verifying backwards compatibility.

5 Further Applications of Morphisms in Formal Verification

Contract morphisms establish a relationship between contracts which makes them suitable for specifying and verifying upgrades. For that same reason, contract morphisms may also have applications in proof reuse, or proof *transport*, more generally. The special case of contract *isomorphism* may also provide a stronger relationship between formal specification and proof on the associated contracts.

5.1 Hoare Properties and Contract Morphisms

First we consider properties that *transport* over a morphism, in particular those that we can pull back over a morphism. Hoare properties are a particularly strong example: they relate pre-conditions to post-conditions, which is relevant to morphisms because morphisms relate inputs and outputs of contract executions. As contracts are formalized in ConCert, constraints on on inputs amount to pre-conditions, and constraints on outputs amount to post-conditions. Thus for contracts `C1` and `C2` and a morphism `f : ContractMorphism C1 C2`, we might expect to be able to transport Hoare properties of one contract over `f` to the other.

Indeed, any Hoare property proved for `C2` will always have an analogous result on `C1`, mediated by `f`. We proved this in two results which relate all reachable states of `C1` to those of `C2`, and those of `C2` to those of `C1`, via the `state_morph` component of `f`. These results, `left_cm_induction` and `right_cm_induction`, are collectively called morphism induction, as they allow us to induct along the execution trace of one contract in relation to that of another.

In particular, morphism induction says that properties of the state of `C2` which are invariant over `state_morph` must hold for all states of `C1`.

As a toy example of this relationship, suppose that we can prove that if a certain boolean in the storage of `C2` is set at `true`, a given entrypoint `e2` of `C2` can be successfully called, and that it fails otherwise. Suppose further that the `msg_morph` component of `f` sends all calls to an entrypoint `e1` of `C1` to calls to `e2`, and that the `state_morph` component of `f` sends a state of `C1` with an analogous boolean set at `true` to one of `C2` with the boolean set at `false`, and visa versa. Then by morphism induction on the trace of `C1`, we get for free that calls to `e1` succeed only when the analogous boolean in the state of `C1` is set at `false`, rather than `true`. The relationship encoded by `f` between contracts `C1` and `C2` shows that `C1` and `C2` use opposing, but predictably related, logic for execution, which allows us to reuse proofs on `C2` to prove analogous results on `C1`.

5.2 Isomorphisms and Propositional Indistinguishability

This relationship between contracts strengthens when we have a pair of morphisms

`f : ContractMorphism C1 C2` and `g : ContractMorphism C2 C1`

such that `compose_cm g f = id_cm C1` and `compose_cm f g = id_cm C2`. This is an *isomorphism* of contracts. Isomorphisms of contracts are particularly strong; the component functions are equivalences of types and they induce a bisimulation of contracts in ConCert.

Since bisimulation is a strong and mathematically stable notion of equivalence [17], future work could investigate proof transport over contract isomorphisms, building on recent work in Coq-based formal methods. For example, we may wish to prove results on a contract optimized for formal reasoning, and transport those onto a bisimlar, performant contract, similar to the work of Cohen *et al.* [5]. This might include altering certain data types while maintaining an equivalence; chosen data types have a strong influence on the structure of proofs and can be nontrivial to transport [11, 15, 20].

6 Conclusion

Our goal in this paper was to provide a formal framework for formally specifying and verifying smart contract upgrades in Coq. To do so we introduced the notion of a contract morphism, which encodes a formal relationship between execution traces of two contracts. We argued that this was a suitable, formal notion with which to reason about contract upgrades and provided examples of contract upgrades which can be specified and verified with contract morphisms. To our knowledge, this is the first time that the intent of an upgrade has been articulated explicitly in formal specification, and is the first formal attempt at reasoning explicitly about contract upgrades in a formal setting.

This work is intended to be a preliminary framework for reasoning about contract upgrades in Coq. As such, there are practical questions to be asked, such as whether these tools are even feasible on gas-optimized code, which can be difficult to formally reason about. Even so we are optimistic, as the previously-mentioned work by Ringer *et al.* in proof repair is practically useful and resembles our framework from a theoretical standpoint. Since the status quo is to simply update the formal specification of a previous version into the specification of the new, we hope that contract morphisms will be a strong start to efficient and rigorous verification of contract upgrades.

References

1. Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. Extracting smart contracts tested and verified in Coq. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, pages 105–121, New York, NY, USA, January 2021. Association for Computing Machinery. `doi:10.1145/3437992.3439934`.
2. Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. ConCert: A smart contract certification framework in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 215–228, New York, NY, USA, January 2020. Association for Computing Machinery. `doi:10.1145/3372885.3373829`.
3. Pedro Antonino, Juliandson Ferreira, Augusto Sampaio, and A. W. Roscoe. Specification is Law: Safe Creation and Upgrade of Ethereum Smart Contracts. In *Software Engineering and Formal Methods: 20th International Conference, SEFM 2022, Berlin, Germany, September 26–30, 2022, Proceedings*, pages 227–243. Springer, 2022.
4. Rob Behnke. Explained: The NowSwap Protocol Hack. https://halborn.com/explained-the-nowswap-protocol-hack-september-2021/, September 2021. Accessed January 2024.
5. Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *International Conference on Certified Programs and Proofs*, pages 147–162. Springer, 2013.
6. Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, Vikram Saraph, and Eric Koskinen. Proof-Carrying Smart Contracts. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 325–338, Berlin, Heidelberg, 2019. Springer. `doi:10.1007/978-3-662-58820-8\_22`.
7. etherscan.io. Nomad Bridge Exploit. Transaction 0xa5fe9d044e4f3e5aa5bc4c0709333cd2190cba0f4e7f16bcf73f49f83e4a5460, 2022.
8. Uranium Finance. Uranium Finance Exploit. https://uraniumfinance.medium.com/exploit-d3a88921531c, April 2021. Accessed January 2024.
9. Immunefi. Hack Analysis: Nomad Bridge, August 2022. https://medium.com/immunefi/hack-analysis-nomad-bridge-august-2022-5aa63d53814a, January 2023.
10. Oussama Jebbar, Ferhat Khendek, and Maria Toeroe. Upgrade of highly available systems: Formal methods at the rescue. In *2017 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 270–274. IEEE, 2017.
11. Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. In *International Workshop on Types for Proofs and Programs*, pages 181–196. Springer, 2000.
12. Stephen McCamant and Michael D Ernst. Predicting problems caused by component upgrades. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 287–296, 2003.
13. Nick Mudge. EIP-2535: Diamonds, Multi-Facet Proxy. https://eips.ethereum.org/EIPS/eip-2535. Accessed January 2024.
14. Talia Ringer. *Proof Repair*. University of Washington, 2021.
15. Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. Proof repair across type equivalences. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 112–127, 2021.
16. Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 115–129, 2018.
17. Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5):447–479, October 1998. `doi:10.1017/S0960129598002527`.
18. Amritraj Singh, Reza M Parizi, Qi Zhang, Kim-Kwang Raymond Choo, and Ali Dehghantanha. Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. *Computers & Security*, 88:101654, 2020.

19 Derek Sorensen. (In)Correct Smart Contract Specifications. *IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2024.

20 Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. Equivalences for free: univalent parametricity for effective transport. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–29, 2018.

21 Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 154–165, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2854065.2854081.

22 Jiahua Xu, Krzysztof Paruch, Simon Cousaert, and Yebo Feng. SoK: Decentralized Exchanges (DEX) with Automated Market Maker (AMM) Protocols. *ACM Computing Surveys*, 55(11):1–50, 2023.