Formally Specifying Contract Optimizations with Bisimulations in Coq

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

Abstract

The efficacy of formal verification of smart contracts depends on being able to correctly specify and carry out the verification of optimized code. However, code optimized for performance is rarely optimized for intelligibility, which can make formally verifying optimized code difficult and costly. To address this issue, we present a formal tool for reasoning about an optimized contract in terms of its reference implementation. This tool reduces the work of formally verifying an optimized contract to proving behavioral equivalence to the reference implementation.

2012 ACM Subject Classification Theory of computation → Program verification

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

Digital Object Identifier 10.4230/OASIcs.FMBC.2025.11

Supplementary Material *Software (Source code)*: `https://github.com/dhsorens/FinCert/tree/FMBC-25` [13]; archived at `swh:1:dir:4373edebe9d83da3ce0c5452513515f60c7173aa`

1 Introduction

The efficacy of formal verification to prevent actual, critical contract vulnerabilities depends on the feasibility of applying formal verification to deployable contract code. However, deployable code is typically optimized for performance, which typically makes it more difficult to reason about formally [3]. Code highly optimized for performance thus risks vulnerability due to the difficulty of formal reasoning, while code written for ease of formal reasoning may not be efficient enough for the resource-scarce environment of smart contracts. Ideally, we would reason about contracts in a state optimized for formal reasoning while still deploying them in a state optimized for efficiency and gas consumption.

What is needed is a formal tool that enables us to reason about code in a format optimized for intelligibility, design and formal reasoning, whose results can be applied to a highly optimized and equivalent version of that code. As it stands no such framework exists for smart contracts. To mitigate this we introduce a formal framework of extensional equivalence between smart contracts in Coq, called *contract isomorphisms*. These equivalences will allow us to use a reference implementation as a specification of an optimized contract, as well as to port proofs between contracts that can be proved to be bisimilar.

We proceed as follows. In Section 2 we discuss related work. In Section 3 we define contract isomorphisms, our notion of formal, extensional equivalence which implements a bisimulation of contracts. In Section 4 we show that our definition of contract isomorphisms induces a strong form of equivalence between contracts, a *trace equivalence*. In Section 5, we give an example of a contract formally specified by equivalence to an existing contract and port proofs over a bisimulation. In Section 6 we conclude.

2 Related Work

Bisimulations are a core component of theoretical computer science. They primarily denote an equivalence of state transition systems [9, 12]. They are, for example, central in the study of process algebras, which rely on a notion of equivalence between processes in order to reason algebraically about the behavior of concurrent systems.

6th International Workshop on Formal Methods for Blockchains (FMBC 2025).
Editors: Diego Marmsoler and Meng Xu; Article No. 11; pp. 11:1–11:13
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

One critical role that bisimulations play is in equivalence checking [4, 5]. Equivalence checking is an approach to formal verification that consists in proving that two programs or models are related modulo some equivalence relation, or that one is included in the other modulo some preorder relation [4]. In this case, one uses a bisimulation to prove that a particular program meets its specification, where the specification is defined not in prose but as a program. Areas of formal reasoning and logic, including Hennessy-Milner logic, treat bisimulations as full equality and cannot distinguish between bisimilar processes [8, 9].

To our knowledge none of these techniques have been applied to smart contracts, but one can imagine that with a sufficient notion of contract bisimulation, we can mimic this process and use a contract formally verified and optimized for formal reasoning as a specification for a contract optimized for deployment. Proving the optimized contract correct then consists of producing a contract bisimulation.

One could also conceive of porting proofs over such an equivalence of contracts, *e.g.* in [2, 3, 15]. The strategy of porting proofs over equivalences is used in formal verification elsewhere. For example, work by Ringer *et al.* uses type equivalences to efficiently reuse proofs when updating a formally verified program [10]; work by Cohen *et al.* [2, 3] uses refinement types to optimize code in a proof-invariant way. Our work here is in a similar spirit, and may be applicable to future version of this work, but our equivalence in question is a contract bisimulation instead of a type equivalence.

Previous work has used bisimulations to encode the notion of correct implementation on a UTXO-based blockchain [6], but to our knowledge the work here is the first application of bisimulations as a tool for formally verifying optimized contracts. We build off of previous work in Coq which introduced the notion of a *contract morphism*, the key tool used to construct contract bisimulations [14]. Our work here is a special use case of that theoretical tool. All of our work is built in ConCert [1], a Coq-based formal verification tool with verified extraction to high-level smart contract languages including Tezos's LIGO and Concordium's Oak.

3 Contract Isomorphisms

The fundamental contribution of this paper is a formal mechanism for proving equivalence (bisimilarity) between smart contracts in Coq, to be used in formal specification and verification. To present this mechanism, we first give a theoretical definition of contract isomorphisms as bisimulations between contracts (Section 3.1), moving onto the details of an implementation in Coq (Section 3.2).

3.1 Bisimilarity

Bisimilarity is a stable and natural concept that describes equivalence between processes [6, 11, 16]. A standard definition of bisimulation for labelled transition systems is as follows.

▶ **Definition 1** (Bisimulation). *Consider a labelled transition system $(S, \Lambda, \to)$, where S is a set of states, Λ is a set of labels, and $\to$ is a set of labelled transitions (a subset of $S \times \Lambda \times S$). A bisimulation is a binary relation $R \subseteq S \times S$ such that for every pair of states $(p, q) \in R$ and labels $\alpha, \beta \in \Lambda$,*

- *if $p \xrightarrow{\alpha} p'$, then there is $q \xrightarrow{\beta} q'$ such that $(p', q') \in R$*
- *if $q \xrightarrow{\beta} q'$, then there is $p \xrightarrow{\alpha} p'$ such that $(p', q') \in R$.*

A bisimulation defines equivalence between transition systems by defining a correspondence between states that is stable under transition: given two equivalent states and a transition on the first, there is a corresponding transition such that the output states are also equivalent.

As we will see in the following section, ConCert models smart contracts as pure functions. Since we wish to capture the notion of bisimulations of contracts by defining equivalences of states that are stable under transitions, our specialized definition of bisimulation is a *natural isomorphism* of pure functions. A natural isomorphism of two pure functions defines a correspondence of function inputs and outputs that is stable under application of the function. In the following definition, we consider the category of contracts defined in [14].

▶ **Definition 2** (Natural Isomorphism of Pure Functions)**.** *Consider functions $F : A \to B$ and $G : A' \to B'$. A natural isomorphism between F and G is a pair of isomorphisms, $\iota_A : A \cong A'$ and $\iota_B : B \cong B'$ such that the following square commutes:*

$$\begin{tikzcd}
A \arrow[r, leftrightarrow, "\iota_A", "\sim"'] \arrow[d, "F"'] & A' \arrow[d, "G"] \\
B \arrow[r, leftrightarrow, "\iota_B", "\sim"'] & B'
\end{tikzcd}$$

Smart contracts modeled as pure functions get their state and entrypoint calls as inputs and as output an updated state with the resulting transactions. Because contract calls result in transitions between contract states, we can consider contracts as state transition systems. The fact that the square commutes is precisely what makes it a bisimulation in this particular interpretation of a state transition system.

3.2 Bisimulations in ConCert

We now move on to give details of a specific implementation of contract bisimulations in ConCert.[1] In ConCert, smart contracts are formalized with a `Contract` type as a pair of pure 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 `C.(init)` and `C.(receive)` are given in Listing 1, where the types `Chain` and `ContractCallContext` are ConCert-specific types used to model the underlying blockchain and context.

Listing 1 Type signatures in ConCert of the `init` and `receive` functions of a contract.

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

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

Following the theory in Section 3.1, we formalize bisimulations in ConCert between a pair of contracts `C1` and `C2`,

```
C1 :  Contract Setup1 Msg1 State1 Error1

C2 :  Contract Setup2 Msg2 State2 Error2,
```

[1] The definitions and results of this section are available at theories/ContractMorphisms.v

$$\begin{tikzcd}
A_{\texttt{C1}} \arrow[r, leftrightarrow, "f_i"] \arrow[d, "\texttt{init}"'] & A'_{\texttt{C2}} \arrow[d, "\texttt{init}"] \\
B_{\texttt{C1}} \arrow[r, leftrightarrow, "f_o"] & B'_{\texttt{C2}}
\end{tikzcd}
\qquad
\begin{tikzcd}
A_{\texttt{C1}} \arrow[r, leftrightarrow, "f_i"] \arrow[d, "\texttt{receive}"'] & A'_{\texttt{C2}} \arrow[d, "\texttt{receive}"] \\
B_{\texttt{C1}} \arrow[r, leftrightarrow, "f_o"] & B'_{\texttt{C2}}
\end{tikzcd}$$

Figure 1 A bisimulation of contracts in ConCert is a natural isomorphism of each of the component functions `init` and `receive`, which inductively constructs a contract bisimulation.

by constructing natural isomorphisms between the `init` and `receive` functions, respectively, of `C1` and `C2`. This is made by defining a correspondence of inputs and outputs to each of `init` and `receive` which is stable under contract initialization and contract calls. Constructing this equivalence consists of proving that the respective `init` functions are equivalent (the base case) and then show that each of the steps are also equivalent (the inductive step).

As we will see in Section 4, these natural isomorphisms induce a trace equivalence of contracts, or a bisimulation of contracts when considering them as state transition systems. This is an *extensional* equivalence of contracts.

3.2.1 Constructing Bisimulations via Contract Isomorphisms

We can encode these extensional equivalences in ConCert using *contract morphisms*, a category theoretic tool defined in ConCert for reasoning formally about one contract in terms of another [14].

A contract morphism is a formal, structural relationship between two contracts which formally relate the inputs, outputs, and state of both contracts in question. Much like the natural isomorphism from before, contract morphisms are encoded in ConCert as a *natural transformation* of contracts, which in diagram form differ from natural isomoprhisms only in that the horizontal arrows only point in one direction (*i.e.* they are not necessarily invertible).

$$\begin{tikzcd}
A_{\texttt{C1}} \arrow[r, "f_i"] \arrow[d, "\texttt{init}"'] & A'_{\texttt{C2}} \arrow[d, "\texttt{init}"] \\
B_{\texttt{C1}} \arrow[r, "f_o"] & B'_{\texttt{C2}}
\end{tikzcd}
\qquad
\begin{tikzcd}
A_{\texttt{C1}} \arrow[r, "f_i"] \arrow[d, "\texttt{receive}"'] & A'_{\texttt{C2}} \arrow[d, "\texttt{receive}"] \\
B_{\texttt{C1}} \arrow[r, "f_o"] & B'_{\texttt{C2}}
\end{tikzcd}$$

Figure 2 A contract morphism between contracts `C1` and `C1` is a natural transformation of `init` and `receive` functions, respectively.

The natural transformation depicted above in Figure 2 corresponds to a morphism `f` from contracts `C1` to `C2`, written either `f : C1 -> C2` or, more formally,

```
f : ContractMorphism(C1,C2).
```

Morphisms compose, and there is a canonical identity morphism [14]. These two facts give us everything we need to define an invertible pair of contract morphisms, or a contract *isomorphism*, which will take as our notion of contract bisimulation.

▶ **Definition 3** (Contract Isomorphism). *A contract isomorphism between contracts* `C1` *and* `C2` *is a pair of morphisms,*

```
f :  ContractMorphism C1 C2

g :  ContractMorphism C2 C1,
```

such that `f` *and* `g` *compose each way to the identity morphism.*

To state this as a formal proposition, we summarize this definition as a proposition in Coq.

Listing 2 Contract isomorphisms are defined as a pair of morphisms that compose each way to the identity morphism under the morphism composition function `compose_cm`.

```
Definition is_iso_cm
    (f : ContractMorphism C1 C2) (g : ContractMorphism C2 C1) : Prop :=
    compose_cm g f = id_cm C1 ∧
    compose_cm f g = id_cm C2.
```

4 Contract Bisimulations Induce Trace Equivalences in ConCert

Our task now is to prove that contract isomorphisms actually induce the desired, strong notion of equivalence between state transition systems. In fact, they induce an isomorphism of the generated trace graphs of the contracts in question [16]. In this section we give a formal proof in Coq that a contract isomorphism produces a trace equivalence of contracts. We define trace equivalence in 4.1. We formalize a trace equivalence between contracts in ConCert in Section 4.2, and then show that contract isomorphisms imply trace equivalence in Section 4.3.

4.1 Trace Equivalences

A trace equivalence between contracts modeled as pure functions is an equivalence of all possible execution trace graphs, or graphs where nodes are states and edges are state transitions labelled with emitted transactions. This is proved inductively with a mapping of nodes and edges which maps initial states to initial states (the base case), and that respects state transitions (the inductive step à la Definition 1).

4.2 Trace Equivalences in ConCert

To codify trace equivalences in ConCert, we formally define contract traces and morphisms betwen contract traces. With morphisms we can formalize equivalence via *trace isomoprhisms*. Similar to contract morhpisms [14], contract trace morphisms are a formal, structural relationship between the traces of two contracts. As we will see, an equivalence of contract traces is the strong form of extensional equivalence that we are looking for.[2]

4.2.1 Contract Traces

We begin by defining some key data types. First, a contract's trace is a chained list of contract states, connected by contract steps.

[2] The definitions and results of this section are available at theories/Bisimulation.v

```
Definition ContractTrace (C : Contract Setup Msg State Error) :=
    ChainedList State (ContractStep C).
```

Contract steps are a record type of the data for a successful contract call, or a call to the `receive` function, which links two contract states. The record contains data for a successful contract call such as the contract call context, the incoming message, the resulting actions, as well as a proof that the call to `receive` succeeds.

Listing 3 Contract steps are successful calls to the `receive` function.

```
Record ContractStep (C : Contract Setup Msg State Error)
      (prev_st : State) (next_st : State) := {
    (* data for a successful contract call *)
    seq_chain : Chain ;
    seq_ctx : ContractCallContext ;
    seq_msg : option Msg ;
    seq_new_acts : list ActionBody ;
    (* we can call receive successfully *)
    recv_some_step :
        receive C seq_chain seq_ctx prev_st seq_msg =
        Ok (next_st, seq_new_acts) ;
}.
```

Contract traces codify the trace of contracts as state transition systems.

4.2.2 Contract Trace Morphism

A *contract trace morphism*, analogous to a contract morphism of [14], encodes a formal, structural relationship between the traces of two contracts. For contracts

`C1:Contract Setup1 Msg1 State1 Error1`

`C2:Contract Setup2 Msg2 State2 Error2`,

a morphism of contract traces includes the following data:

- A function between contract state types, `ct_state_morph : State1 -> State2`.
- A proof that `ct_state_morph` sends valid initial states of `C1` to valid initial states of `C2`.
- A function `cstep_morph` that, for states `state1` and `state2` of `C1`, sends a contract step

  `step1 : ContractStep C1 state1 state2`,

  to a corresponding contract step of `C2` between the corresponding states

  `step2 : ContractStep C2 (ct_state_morph state1) (ct_state_morph state2)`.

Inductively, this data gives us a relationship between all reachable states: initial states of each contract are related via the function between state types, and from there, any contract step of `C1` is related to a contract step of `C2` that respects the function on states. We codify this with a type `f : ContractTraceMorphism C1 C2`.

▶ **Example 4** (The Identity Contract Trace Morphism). For any contract `C` we can define the identity morphism `id_ctm`, whose component functions are the identity and respective proofs are trivial, and which inhabits the type `ContractTraceMorphism C C`.

▶ **Example 5** (Contract Trace Morphism Composition). We can define composition of contract trace morphisms similar to composition of contract morphisms in [14], via a function `compose_ctm`, which takes morphisms

`f : ContractTraceMorphism C1 C2` and `g : ContractTraceMorphism C2 C3` and returns a morphism

```
compose_ctm g f :  ContractTraceMorphism C1 C3.
```

To compose contract morphisms, we simply compose their component functions. That composition is associative comes trivially. Similarly, it comes immediately that composition on either side with the identity is a trivial operation, and so composition and identity behave as we might expect in a well-defined category.

4.2.3 Contract Trace Isomorphisms

Contract trace isomorphisms are then defined analogously to contract isomorphisms (3.2.1).

▶ **Definition 6** (Contract Trace Isomorphism). *A contract trace isomorphism between contracts* `C1` *and* `C2` *is a pair of trace morphisms,*

```
f :  ContractTraceMorphism C1 C2

g :  ContractTraceMorphism C2 C1,
```

such that `f` *and* `g` *compose each way to the identity morphism* `id_ctm`*.*

To state this as a formal proposition, we summarize this definition in a type in Coq.

Listing 4 Contract trace isomorphisms are defined as a pair of morphisms that compose each way to the identity morphism.

```
Definition is_iso_ctm
(m1 : ContractTraceMorphism C1 C2) (m2 : ContractTraceMorphism C2 C1) :=
compose_ctm m2 m1 = id_ctm C1 ∧
compose_ctm m1 m2 = id_ctm C2.
```

By definition, if two contracts are related by a contract trace isomorphism, then there is a one-to-one correspondence between all possible contract states; furthermore, this correspondence respects initial states. Thus extensionally, contracts which are trace isomorphic have identical behavior up to their state isomoprhisms. When considered as a labelled transition system, their execution graphs are necessarily isomorphic. The behavior of a contract is fully defined by its initial state and the steps it can take from there, and so contract trace isomorphisms give us the strong form of extensional equivalence we are looking for.

4.3 Contract Morphisms to Contract Trace Morphisms

The final result of this section is that contract bisimulations induce contract trace isomorphisms. We prove this result by defining a function `cm_to_ctm`, which takes a contract morphism

```
f :  ContractMorphism C1 C2
```

and returns a contract trace morphism

```
cm_to_ctm f :  ContractTraceMorphism C1 C2,
```

which respects identity and compositions. Contract morphisms and contract trace morphisms define a category whose objects are contracts in ConCert, so `cm_to_ctm` is a functor.

To define `cm_to_ctm` for a contract morphism `f : C1 -> C2`, we need a function between the state types of `C1` and `C2` which respects initial states and state transitions. The obvious candidate is, of course, the component function of `f` of contract states, `f.(state_morph)`, which respects initial states state transitions by the coherence conditions of its definition [14].

Furthermore, the identity contract morphism induces the identity contract trace morphism, and compositions of contract morphisms induce compositions of contract trace morphisms.

Listing 5 Identity induces the identity.

```
Theorem cm_to_ctm_id : cm_to_ctm (id_cm C1) = id_ctm C1.
```

Listing 6 Compositions induce compositions.

```
Theorem cm_to_ctm_compose (g : ContractMorphism C2 C3) (f : ContractMorphism C1 C2) :
    (* the image of the composition = ... *)
    cm_to_ctm (compose_cm g f) =
    (* composing the image morphisms *)
    compose_ctm (cm_to_ctm g) (cm_to_ctm f).
```

Since contract isomorphisms and contract trace isomorphisms are both defined as respective morphism pairs which compose each way to the identity, a bisimulation of contracts induces a trace equivalence. We have our desired result.

5 Using Bisimulation as a Tool for Formal Specification

Contract isomorphisms (bisimulations) could be considered as a tool in at least two ways: first, to reuse proofs on a different contract version by porting them over the isomorphism and achieve those results on the target contract, *e.g.* as in [14]; and second, to use a contract *as a specification*. To show this, in this section we give an example of a contract whose specification is another contract, *e.g.* a reference implementation, and explore the ways in which proofs transport over a contract bisimulation. This is an example where a common optimization makes a contract more difficult to reason about, and we use a contract bisimulation to formally specify the optimized contract with the intelligible contract.[3]

5.1 Linked Lists and Dynamic Arrays

Consider a simple contract `C_arr` that manages an array of owners, *e.g.* for access control, each identified by a natural number. It has functionality to add owners, remove owners, and swap owners. Consider also a second implementation `C_ll` that does the same, except that it stores owner IDs as a linked list instead of a dynamic array, a common contract optimization strategy over arrays in Solidity which introduces nontrivial challenges to verification [7]. The correctness criteria for the second, optimized implementation are that it behave identically to the reference implementation from an extensional standpoint, precisely because the linked list is supposed to emulate a dynamic array (though more efficiently at the bytecode level).

The two contracts `C_arr` and `C_ll` share setup and entrypoint types, but differ in their storage types and the implementation of their entrypoint functions. Both contracts must maintain a set of owners with no duplicates.

[3] The contracts and bisimulations of this section are available at optimization2.v

Listing 7 The entrypoint type shared by both C_arr and C_ll.

```
Inductive entrypoint :=
    | addOwner (a : N) (* to add a as an owner ID *)
    | removeOwner (a : N) (* to remove a as an owner ID *)
    | swapOwners (a_fst a_snd : N). (* to swap a_fst for a_snd as owners *)
```

The first contract, C_arr, keeps track of owners in an array in its storage type storage_arr.

```
Record storage_arr := { owners_arr : list N }.
```

The optimized contract C_ll keeps track of owners in a linked list implemented (somewhat unconventionally in Coq) via a finite mapping.

```
Record storage_ll := { owners_ll : FMap N N }.
```

The mapping emulates an array as follows: Using a global constant SENTINEL : N, the empty list is emulated as the mapping which points SENTINEL to SENTINEL.

```
arr_to_ll := [] ⇒ { SENTINEL : SENTINEL }.
```

From here, to insert an element a into the mapping, we point SENTINEL to a, and a to whatever SENTINEL used to point to (SENTINEL if a is the first element of the list).

```
arr_to_ll := [a] ⇒ { SENTINEL : a ; a : SENTINEL }.
```

This pattern continues such that in the mapping SENTINEL always points to the most recently-added element, and elements form a chain until the last points back to SENTINEL. So for list of the form [a, b, c], the corresponding mapping points SENTINEL to a, and a to b, b to c, and c back to SENTINEL.

```
arr_to_ll := [a, b, c] ⇒ { SENTINEL : a ; a : b ; b : c ; c : SENTINEL}
```

The three entrypoints behave analogously for their respective data structures. For our array contract, C_arr, calling (addOwner a) simply appends a to the list of owners (provided a is not already an owner).

```
addOwner a := {| owners_arr := l |} ⇒ {| owners_arr := a :: l |}.
```

For our linked list contract, C_ll, calling (addOwner a) inserts the owner into the linked list.

```
addOwner a := {| owners_ll := { SENTINEL : a' ; ...  } |} ⇒
    {| owners_ll := SENTINEL : a ; a : a' ; ...  |}.
```

Removing an owner behaves similarly: for C_arr, (removeOwner a) removes a from the array,

```
removeOwner a := {| owners_arr := [ ..., b, a, b',  ...  ] |} ⇒
    {| owners_arr := [ ...,  b, b',  ...  ] |}.
```

while for C_ll, (removeOwner a) updates the pointers in the mapping to excise a.

```
removeOwner a := {| owners_ll := { ... ; b : a ; a : b' ; ...  } |} ⇒
    {| owners_ll := ... ; b : b' ; ...  |}.
```

Finally, to swap owners in C_arr, (swapOwners a a') replaces a with a',

```
swapOwners a a' := {| owners_arr := [ ...,  b, a, b',  ...  ] |} ⇒
    {| owners_arr := [ ...,  b, a', b',  ...  ] |}.
```

and C_ll, does the analogous operation by updating its pointers.

```
swapOwners a a’ := {| owners_ll := { ... ; b : a ; a : b’ ; ...  } |} ⇒
    {| owners_ll := ... ; b : a’ ; a’ : b’ ; ...  |}.
```

5.2 The Bisimulation

We now explore the consequences of a bisimulation, or a contract isomoprhism, between our reference implementation C_arr and its counterpart C_ll.

```
Theorem bisim_arr_ll : contracts_isomorphic C_arr C_ll.
```

A witness of the proposition contracts_isomorphic C_arr C_ll is a contract isomorphism between C_arr and C_ll. We first explore how a bisimulation between C_arr and C_ll lets us use code as a specification (5.2.1), and then explore how the specification of each ports over the bisimulation (5.2.2). Note that in the following example we assume some key properties about array and map operations and their properties.

5.2.1 Contract as a Specification

The purpose of any contract optimization is to improve the performance of the code without changing its behavior within some semantic domain. That domain is, at least in principle, the domain of a formal specification. This almost always means that changes can be made *intentionally*, affecting the inner workings of the contract, but *extensional* behavior – behavior from an outside or semantic perspective – should remain the same. In the case of our contracts C_arr and C_ll, we expect C_ll to behave identically to C_arr up to an equivalence of data structures. That precise equivalence, of expected behavior of data structures and contract entrypoints, is exactly the data held in the contract isomorphism.

To illustrate this point, we construct the contract morphism. To do so we need functions between entrypoint, state, error, and setup types [14]. Because C_arr and C_ll differ only in their entrypoint type, these functions are the identity on all but the entrypoint type; and for the entrypoint type, these are the functions arr_to_ll and ll_to_arr specified above in Section 5.1.

Listing 8 The component functions of morphisms between C_arr and C_ll.

```
(* msg, setup, and error morphisms are all identity *)
Definition msg_morph : entrypoint → entrypoint := id.
Definition setup_morph : setup → setup := id.
Definition error_morph : error → error := id.

(* storage morphisms *)
Definition state_morph : owners_arr → owners_ll := arr_to_ll.
Definition state_morph_inv : owners_ll → owners_arr := ll_to_arr.
```

With these component functions we can prove the corresponding coherence conditions, and we get morphisms:

```
f : ContractMorphism C_arr C_ll   and   f_inv : ContractMorphism C_ll C_arr
```

The key point of data held in this pair of functions, which form a bisimulation, is in the way that they codify the relationship in functionality between storage and entrypoints in each contract. This is precisely the data of the argument we made in Section 5.1 that C_ll was indeed an alternative representation of C_arr.

Consider in particular the behavior of calling (addOwner a). We know from Section 5.1 that in C_arr this appends a to the list of owners, while in C_ll this inserts a into the implemented linked list. We have a formal proof of this correspondence in the following two lemmas. The functions add_owner_arr and add_owner_ll are, respectively, the functions that implement the addOwner entrypoint in each of C_arr and C_ll.

Listing 9 Two coherence results which show the correspondence of the addOwner entrypoint between C_arr and C_ll.

```
Lemma add_owner_coh : forall a st st' acts,
    add_owner_arr a st = Ok (st', acts) →
    add_owner_ll a (state_morph st) = Ok (state_morph st', acts).
```

```
Lemma add_owner_coh' : forall a st e,
    add_owner_arr a st = Err e →
    add_owner_ll a (state_morph st) = Err e.
```

These are coherence results à la Figure 2: adding a to the state of C_arr and then transforming the state to a linked list is the same as transforming the state to a linked list first and then adding a to the state of C_ll, and vice versa. They constitute a formal proof that the behavior of the two contracts is the same up to the equivalence of their data structures for the addOwner entrypoint.

We have analogous proofs for each of the remaining two entrypoints of C_arr and C_ll. That they give us a bisimulation of contracts tells us that the behavior of the two contracts is the same up to the equivalence of their data structures for each entrypoint – and the equivalence of their data structures is precisely a formal description of how arrays are emulated as linked lists in the state of C_ll. How could you possibly be more precise in formally specifying C_ll as an optimization of C_arr than by a formal proof like this that the two contracts are extensionally equivalent?

5.2.2 Porting Properties Over the Bisimulation

Standard practice for comparing an optimized contract to its reference implementation would be to apply the same test suites or formal specification to the new contract and ensure that it passes all tests and still conforms to the formal specification. If the formal specification includes details of the inner workings of the contract, then relevant alterations are made to the formal specification to accommodate the new setting. This is a translation effort, which can be prone to mistranslation and resulting errrors by underspecification, so instead we would rather see if we can port previously-proved results over a bisimulation.

Indeed, we can and we will do so here with a key property for both contracts: that there be no duplicate owner IDs in storage. This property is important not only because of the intended contract functionality of C_arr, but also in the optimization of C_arr into C_ll. Due to the implementation of C_ll as a linked list via a mapping, being able to add a "duplicate" would actually compromise the integrity of the linked list as a model of an array: the mapping only allows for an owner ID to point to one other owner, so adding a "duplicate" would mean altering the pointers and unlinking the data structure. That there not be duplicates is thus an important property both from the perspective of high-level functionality (with respect to contract permissions and control flow) as well as from the perspective of low-level implementation correctness (linked list implementation emulating an array).

We first formally verify the reference implementation, C_arr, by proving that all reachable contract states are free of duplicates, codified in the following result.

Listing 10 All reachable states of `C_arr` have no duplicate owners in storage.

```
Theorem no_dup_arr (st : owners_arr) :
    reachable C_arr st → no_duplicates_arr st.
```

Using the bisimulation, we can now prove the analogous result about `C_ll` using `morphism_induction`, a proof technique that leverages contract morphisms to compare the reachable states of contracts related by contract morphisms [14].

▶ **Lemma 7** (Morphism Induction). *Consider contracts* `C1` *and* `C2` *and a contract morphism* `f : ContractMorphism C1 C2`*. Then every reachable state* `st_1` *of* `C1` *corresponds to a reachable state* `st_2` *of* `C2`*, related by the state morphism component of* `f` *such that*

```
st_2 == f.(state_morph) st_1.
```

This lemma is codified as `left_cm_induction` in FinCert.[4]

Because we have a bisimulation, not only do we know that the states of `C_arr` and `C_ll` are related by `state_morph` described in Section 5.2.1, but we know that `state_morph` has an inverse. Thus using the details of that morphism we can prove that `C_arr` has duplicates in storage if and only if `C_ll` has been unlinked, the analogous property for duplicates in a linked list. By morphism induction, then, we have the analogous result on `C_ll`.

Listing 11 The desired result that all reachable states of `C_ll` have no duplicate owners in storage.

```
Theorem no_dup_ll (st : owners_ll) :
    reachable C_ll st → no_duplicates_ll st.
```

6 Conclusion

The efficacy of formal verification on smart contracts depends on being able to correctly specify and carry out the verification of optimized code. However, code optimized for performance is rarely optimized for intelligibility, which can make formally verifying optimized code difficult and costly. To remedy this, we introduced contract isomorphisms, a formal tool that establishes a structural equivalence between smart contracts, and we proved that contract isomorphisms give us full trace equivalences of contracts. We then demonstrated how contract isomorphisms can be used to formally specify and verify an optimized smart contract by proving it extensionally equivalent to its reference implementation. Our example illustrates the practical application of this framework to a common optimization technique in smart contract development. It shows how formal proofs of correctness can be ported over a bisimulation and how a bisimulation enables the use of a contract as a specification. We hope that this work paves the way for more robust and reliable smart contract verification, enabling practitioners to more easily reason about optimized contracts in terms of their more intelligible reference implementations.

References

1 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`.

[4] theories/ContractMorphisms.v

2 Cyril Cohen, Enzo Crance, and Assia Mahboubi. Trocq: proof transfer for free, with or without univalence. In *European Symposium on Programming*, pages 239–268. Springer, 2024. doi:10.1007/978-3-031-57262-3_10.

3 Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *International Conference on Certified Programs and Proofs*, pages 147–162. Springer, 2013. doi:10.1007/978-3-319-03545-1_10.

4 Hubert Garavel and Frédéric Lang. Equivalence checking 40 years after: A review of bisimulation tools. *A Journey from Process Algebra via Timed Automata to Model Learning: Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*, pages 213–265, 2022. doi:10.1007/978-3-031-15629-8_13.

5 Osman Hasan and Sofiene Tahar. Formal verification methods. In *Encyclopedia of Information Science and Technology, Third Edition*, pages 7162–7170. IGI global, 2015.

6 Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, January 1985. doi:10.1145/2455.2460.

7 Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Aleksandar Nanevski, and Mooly Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 756–772, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-39799-8_53.

8 Kim G Larsen. Proof systems for satisfiability in Hennessy-Milner logic with recursion. *Theoretical Computer Science*, 72(2-3):265–288, 1990. doi:10.1016/0304-3975(90)90038-J.

9 Robin Milner. Communication and concurrency. *Prentice Hall International*, 13, 1989.

10 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. doi:10.1145/3453483.3454033.

11 Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5):447–479, October 1998. doi:10.1017/S0960129598002527.

12 Davide Sangiorgi. *Introduction to bisimulation and coinduction*. Cambridge University Press, 2011.

13 Derek Sorensen. FinCert. Software (visited on 2025-05-08). URL: https://github.com/dhsorens/FinCert/tree/FMBC-25, doi:10.4230/artifacts.23081.

14 Derek Sorensen. Towards Formally Specifying and Verifying Smart Contract Upgrades in Coq. *5th International Workshop on Formal Methods for Blockchains (FMBC 2024)*, 2024.

15 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. doi:10.1145/3236787.

16 Johan Van Benthem and Jan Bergstra. Logic of transition systems. *Journal of Logic, Language and Information*, 3:247–283, 1994. doi:10.1007/BF01160018.