# Formally Specifying Contract Optimizations With Bisimulations in Coq

Derek Sorensen

University of Cambridge

FMBC 2025 4 May 2025

# Background and Setting

1. Contracts are difficult to formally specify correctly.
2. Performant code is harder to formally verify than naive code.
   - Binary `N` v Unary `nat`
   - `N.of_nat : nat -> N`
   - `N.to_nat : N -> nat`

# Background and Setting

1. Contracts are difficult to formally specify correctly.
2. Performant code is harder to formally verify than naive code.
   - Binary `N` v Unary `nat`
   - `N.of_nat : nat -> N`
   - `N.to_nat : N -> nat`

We would lke:

1. Reason about the reference implementation
2. (Safely) Deploy the performant implementation

# Goals

1. Define a notion of equivalences of smart contracts
2. Use it to reason about optimized contract code, **in terms of its reference implementation**.

# Goals

1. Define a notion of equivalences of smart contracts
2. Use it to reason about optimized contract code, **in terms of its reference implementation**.

**Introducing:** Contract Isomorphisms (Bisimulations)

# Contracts in ConCert

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

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

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

# Natural Isomorphism

## Definition (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:

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

# Contract Isomorphism

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

Figure: An isomorphism of contracts in ConCert is a natural isomorphism on `init` and `receive`.

`contracts_isomorphic C1 C2`

# Contract Isomorphisms are Bisimulations

Some formally proved results:

1. A contract morphism induces a morphism of contract traces
2. Composing morphisms results in composed trace morphisms
3. The identity morphism is the identity trace morphism

```
Corollary ciso_to_ctiso f g :
    is_iso_cm f g → is_iso_ctm (cm_to_ctm f) (cm_to_ctm g).
```

# Linked Lists vs Dynamic Arrays

C_arr

```
Inductive entrypoint :=
  | addOwner (a : N)
  | removeOwner (a : N)
  | swapOwners (a_fst a_snd : N).
```

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

C_ll

```
Inductive entrypoint :=
  | addOwner (a : N)
  | removeOwner (a : N)
  | swapOwners (a_fst a_snd : N).
```

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

# Linked Lists vs Dynamic Arrays

[] $\Rightarrow$ { SENTINEL : SENTINEL }.

[ a] $\Rightarrow$ { SENTINEL : a ; a : SENTINEL }.

[ a, b, c] $\Rightarrow$ { SENTINEL : a ; a : b ; b : c ; c : SENTINEL}

This correspondence results in functions:

```
Definition arr_to_ll (st : owners_arr) : owners_ll.
Definition ll_to_arr (st : owners_ll) : owners_arr.
```

## Linked Lists vs Dynamic Arrays

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

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

```
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.
```

`contracts_isomorphic C_arr C_ll`

# Porting Properties Over the Bisimulation

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

Isomorphism implies:

1. Correct functionality of the entry point function
2. Correct optimization of the list structure

# Porting Properties Over the Bisimulation

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

Isomorphism implies:

1. Correct functionality of the entry point function
2. Correct optimization of the list structure

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

# Revisting Goals

1. Define a notion of equivalences of smart contracts, with:
   - Type equivalences (modulo context)
   - Extensional equivalence of entry point functions
2. Provide a mechanism for:
   - Reasoning about optimized contract code
   - Specifying optimized code

Caveats:

1. Experimental/untested
2. Highly manual