

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
 - $N.\text{of_nat} : \text{nat} \rightarrow N$
 - $N.\text{to_nat} : N \rightarrow \text{nat}$

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
 - $N.\text{of_nat} : \text{nat} \rightarrow N$
 - $N.\text{to_nat} : N \rightarrow \text{nat}$

We would like:

- 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 $F : A \rightarrow B$ and $G : A' \rightarrow B'$. 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:

$$\begin{array}{ccc} A & \xleftarrow[\sim]{\iota_A} & A' \\ F \downarrow & & \downarrow G \\ B & \xleftarrow[\sim]{\iota_B} & B' \end{array}$$

Contract Isomorphism

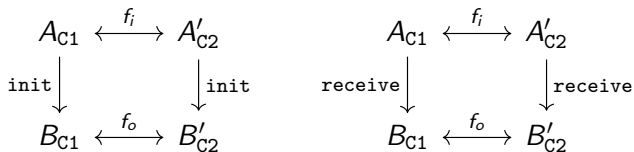


Figure: An isomorphism of contracts in ConCert is a natural isomorphism on `init` and `receive`.

`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

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

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

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

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

$$\text{addOwner } a := \{ | \text{owners_arr} := 1 | \} \Rightarrow \{ | \text{owners_arr} := a :: 1 | \}.$$

$$\begin{aligned} \text{addOwner } a := \{ | \text{owners_ll} := \{ \text{SENTINEL} : a' ; \dots \} | \} \Rightarrow \\ \{ | \text{owners_ll} := \text{SENTINEL} : a ; a : a' ; \dots | \}. \end{aligned}$$

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 \rightarrow 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 \rightarrow 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