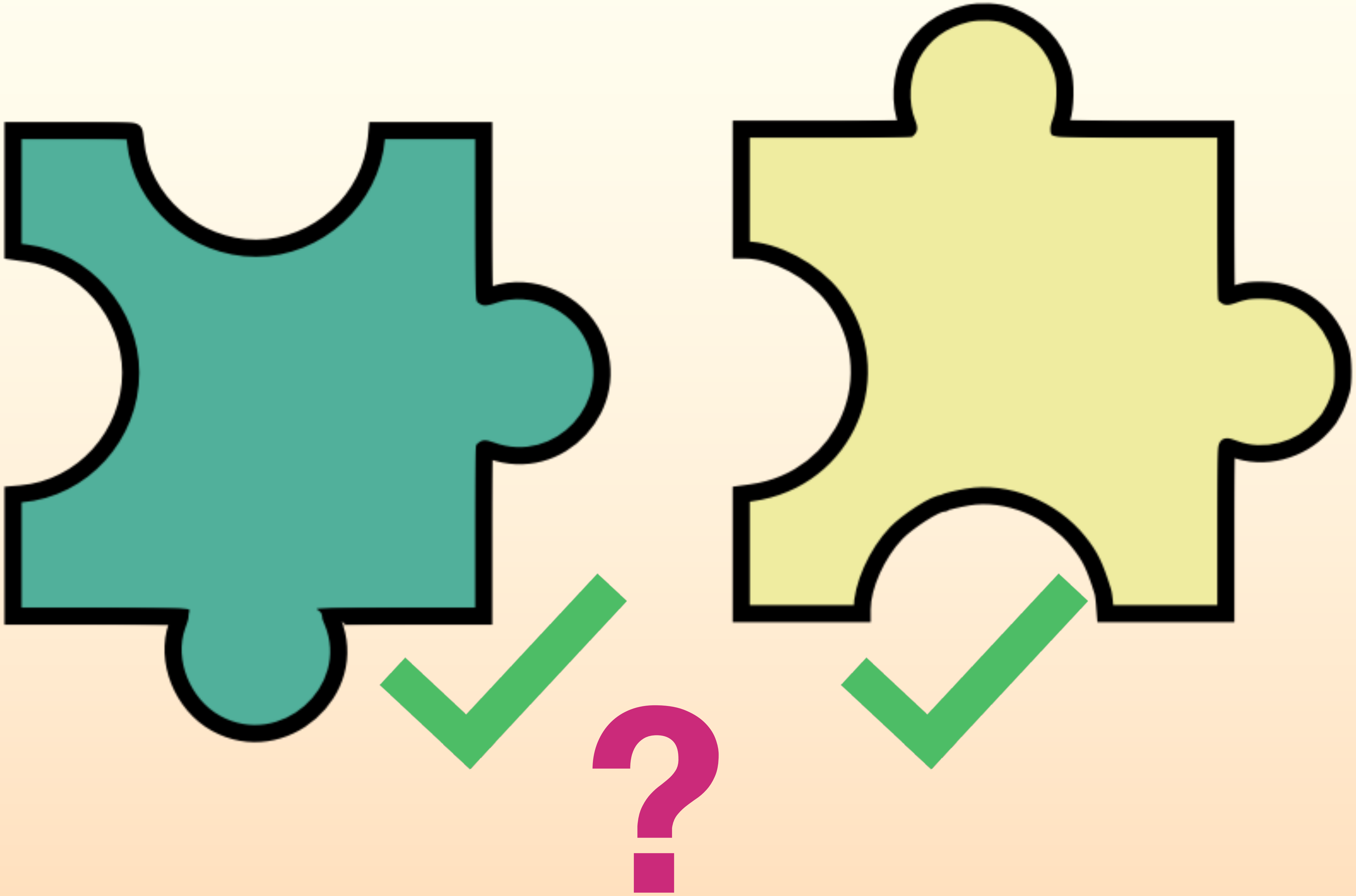
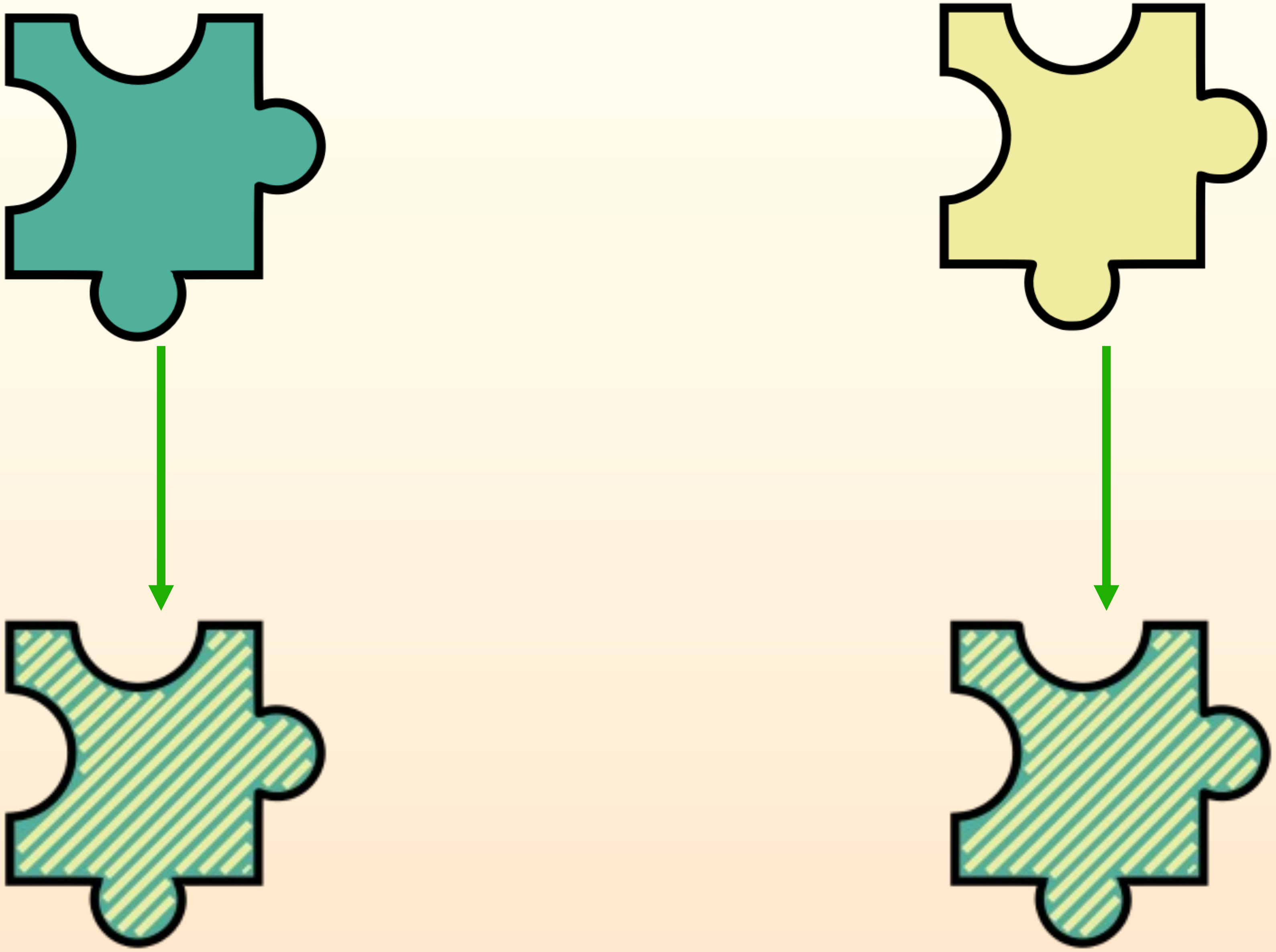
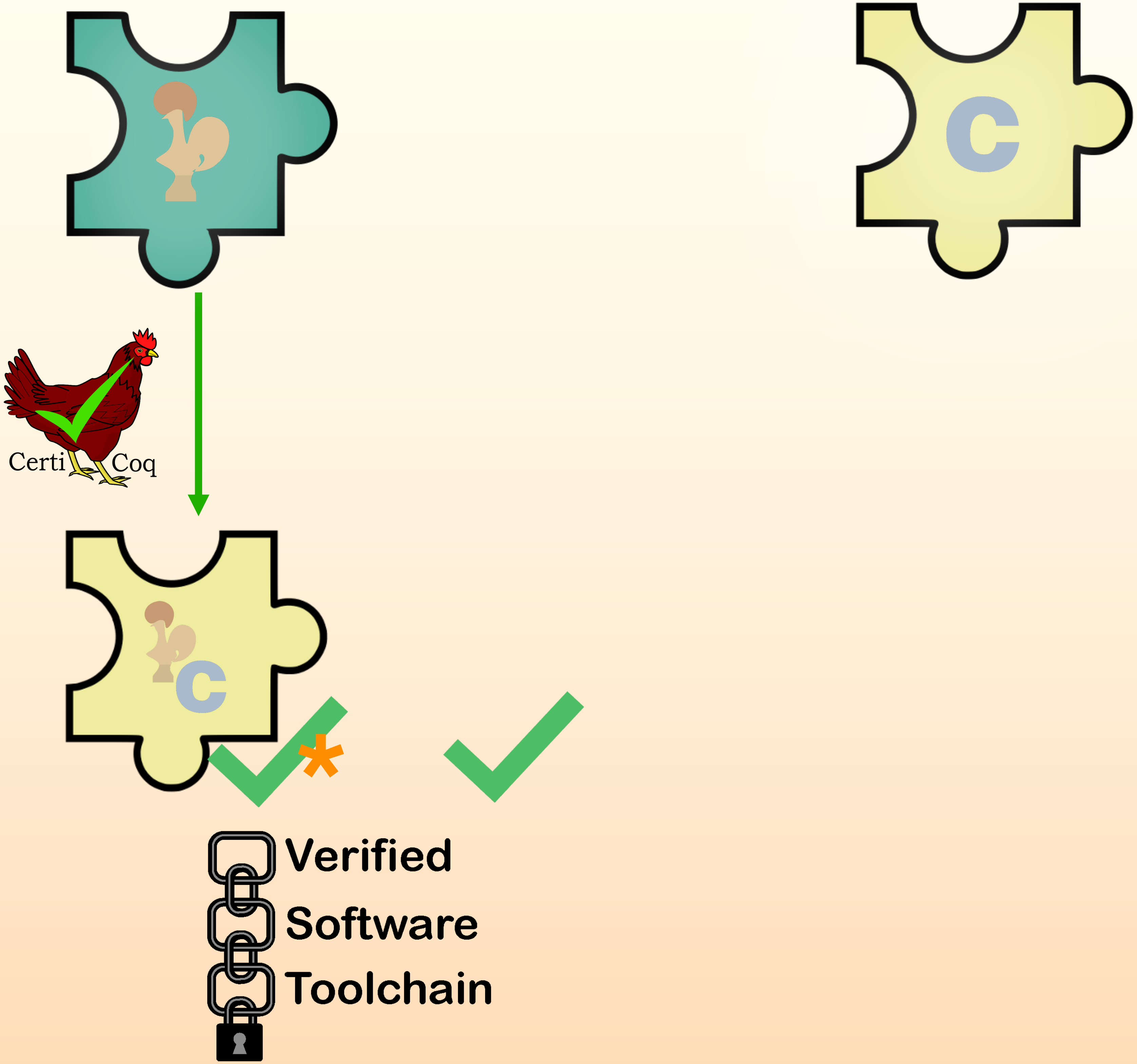
Foreign Function Verification Through Metaprogramming

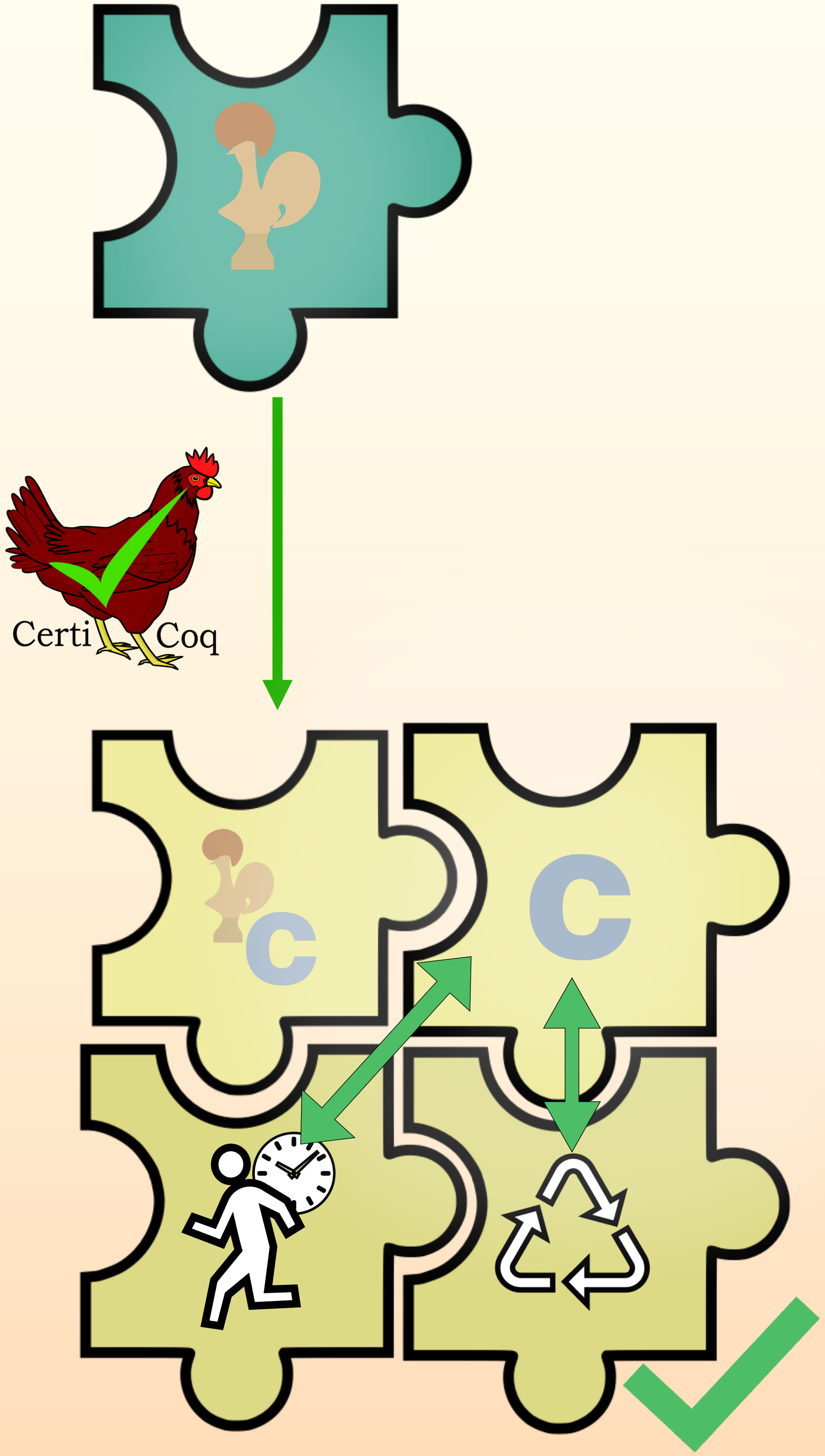
Joomy Korkut
Princeton University

Final Public Oral Examination
October 9th, 2024

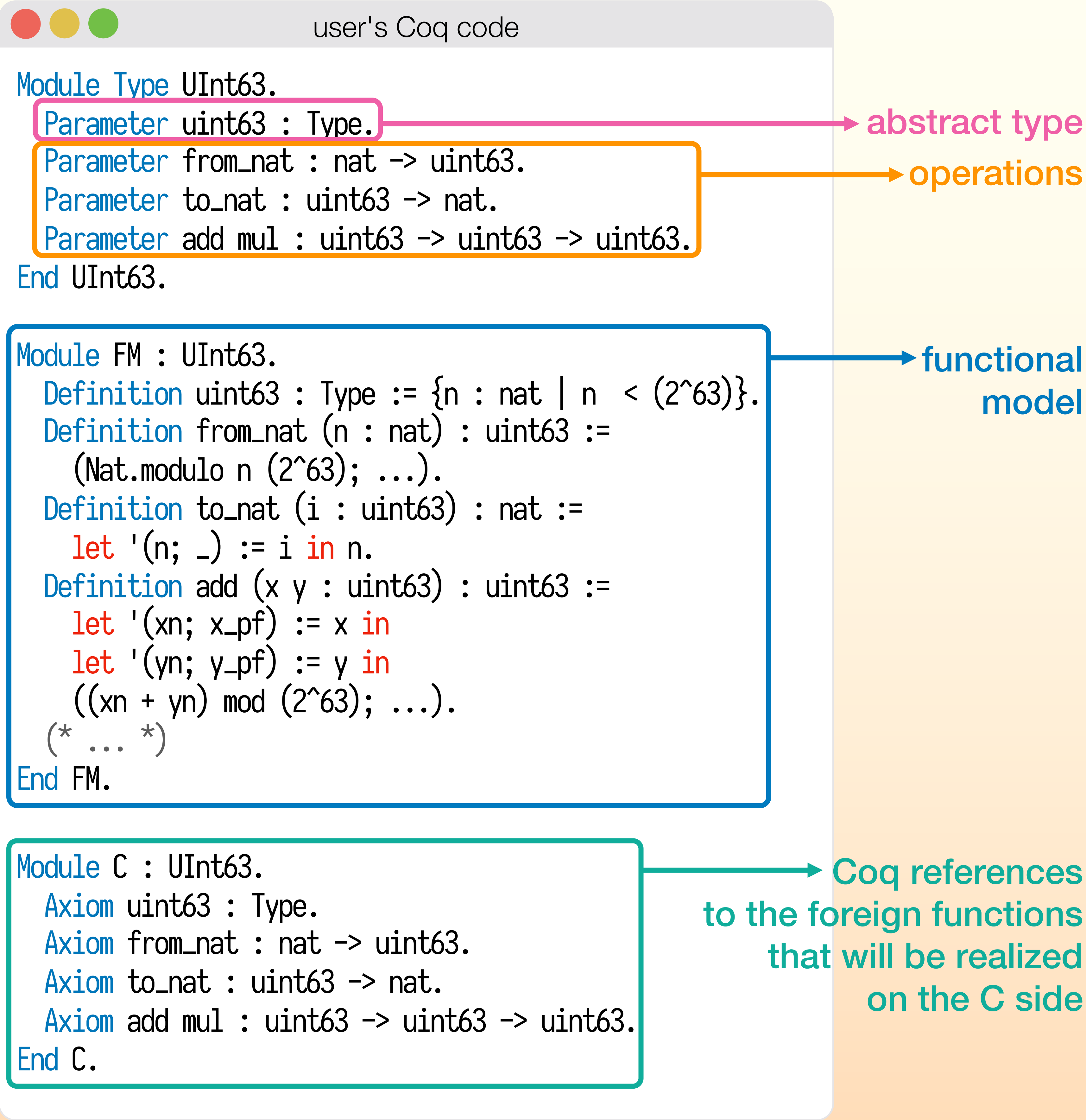





C
Certi Coq
C
*
Verified
Software
Toolchain



Wang et al.
"Certifying Graph-Manipulating C Programs via Localizations within Data Structures"
OOPSLA 2019



user's Coq code

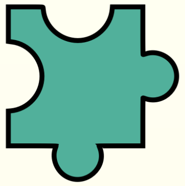
```
Module Type UInt63.
  Parameter uint63 : Type.
  Parameter from_nat : nat -> uint63.
  Parameter to_nat : uint63 -> nat.
  Parameter add mul : uint63 -> uint63 -> uint63.
End UInt63.

Module FM : UInt63.
  Definition uint63 : Type := {n : nat | n  < (2^63)}.
  Definition from_nat (n : nat) : uint63 :=
    (Nat.modulo n (2^63); ...).
  Definition to_nat (i : uint63) : nat :=
    let '(n; _) := i in n.
  Definition add (x y : uint63) : uint63 :=
    let '(xn; x_pf) := x in
    let '(yn; y_pf) := y in
    ((xn + yn) mod (2^63); ...).
  (* ... *)
End FM.

Module C : UInt63.
  Axiom uint63 : Type.
  Axiom from_nat : nat -> uint63.
  Axiom to_nat : uint63 -> nat.
  Axiom add mul : uint63 -> uint63 -> uint63.
End C.
```

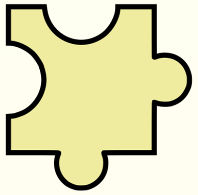
abstract type

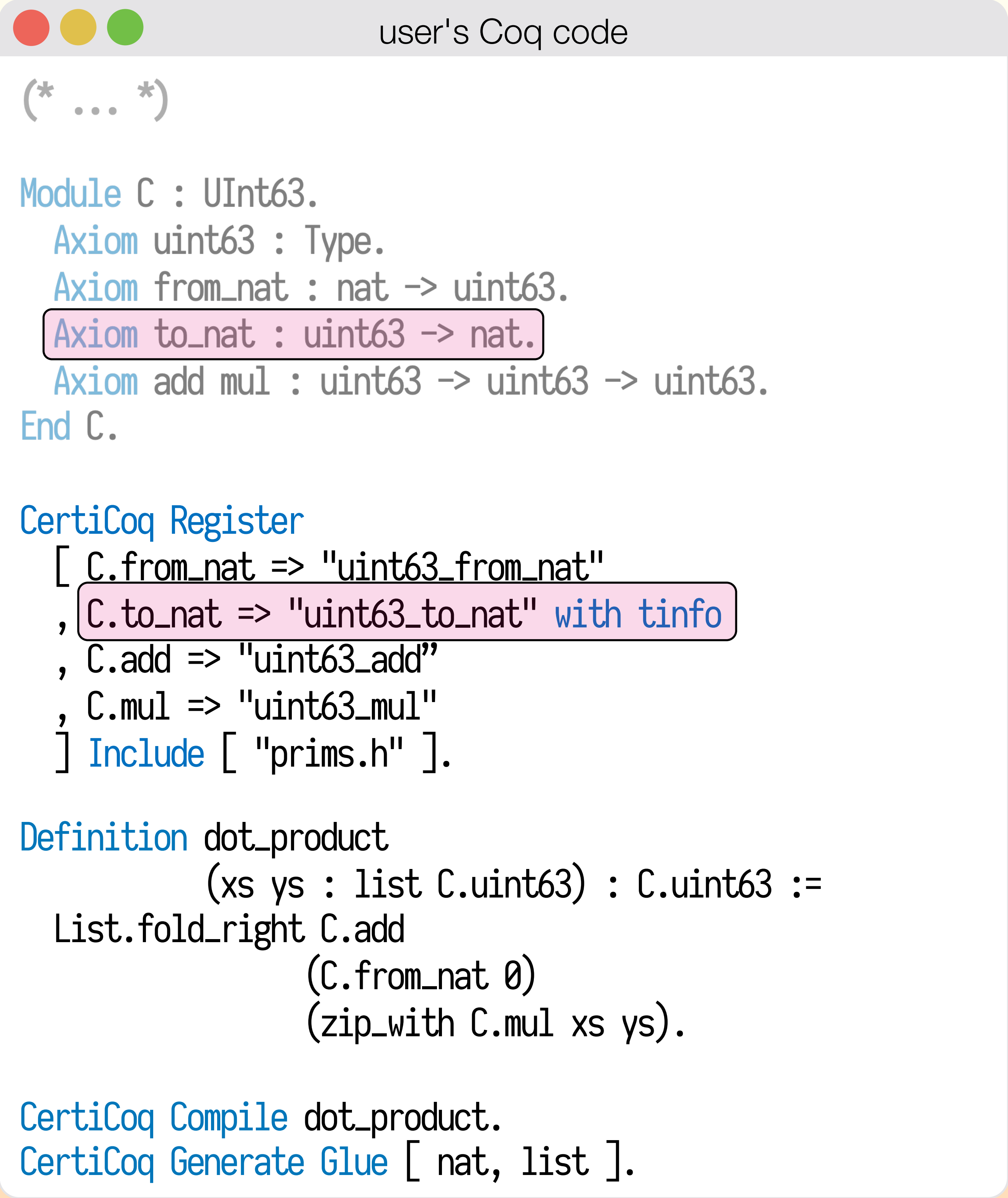
operations

functional model

Coq references to the foreign functions that will be realized on the C side

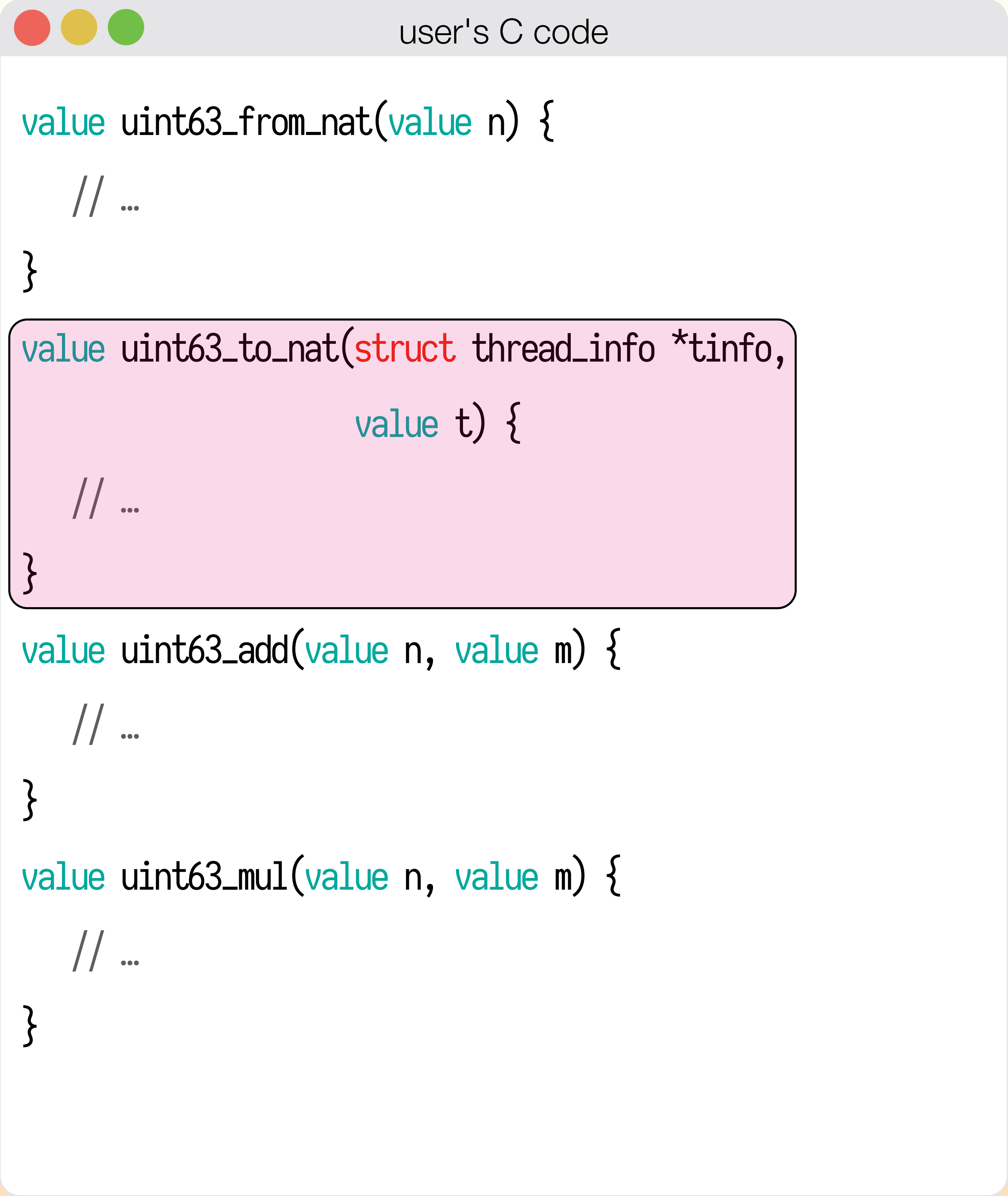






user's Coq code

```
(* ... *)

Module C : UInt63.
  Axiom uint63 : Type.
  Axiom from_nat : nat -> uint63.
  Axiom to_nat : uint63 -> nat.
  Axiom add mul : uint63 -> uint63 -> uint63.
End C.

CertiCoq Register
  [ C.from_nat => "uint63_from_nat"
  , C.to_nat => "uint63_to_nat" with tinfo
  , C.add => "uint63_add"
  , C.mul => "uint63_mul"
  ] Include [ "prims.h" ].

Definition dot_product
           (xs ys : list C.uint63) : C.uint63 :=
  List.fold_right C.add
                  (C.from_nat 0)
                  (zip_with C.mul xs ys).

CertiCoq Compile dot_product.
CertiCoq Generate Glue [ nat, list ].
```



user's C code

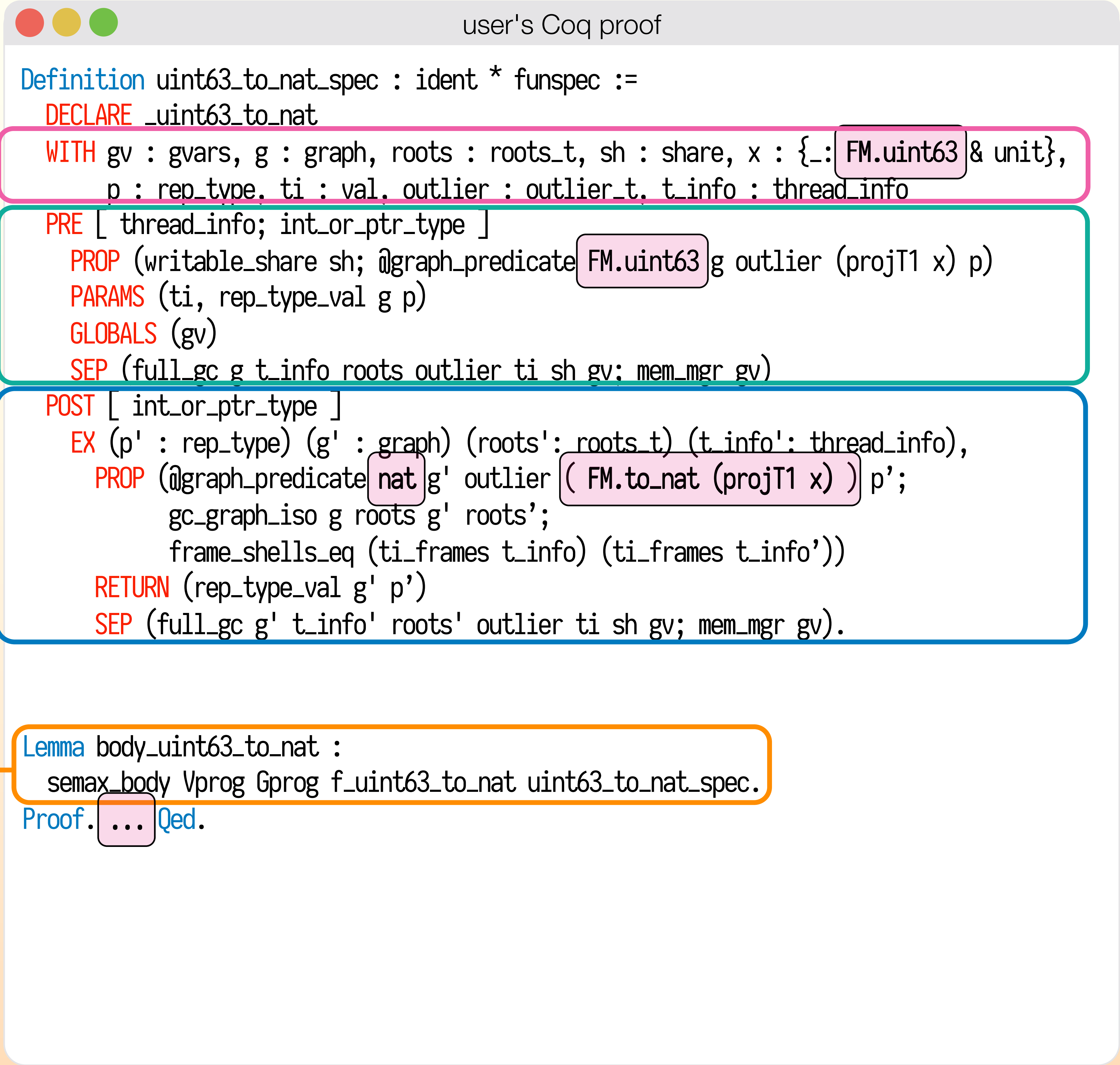
```
value uint63_from_nat(value n) {
   // …
}
value uint63_to_nat(struct thread_info *tinfo,
                    value t) {
   // …
}
value uint63_add(value n, value m) {
   // …
}
value uint63_mul(value n, value m) {
   // …
}
```

Given some runtime info, and an input in the functional model,

if the C function takes a value that is represented by the functional model input,

then the C function returns a value that is represented by the functional model output.

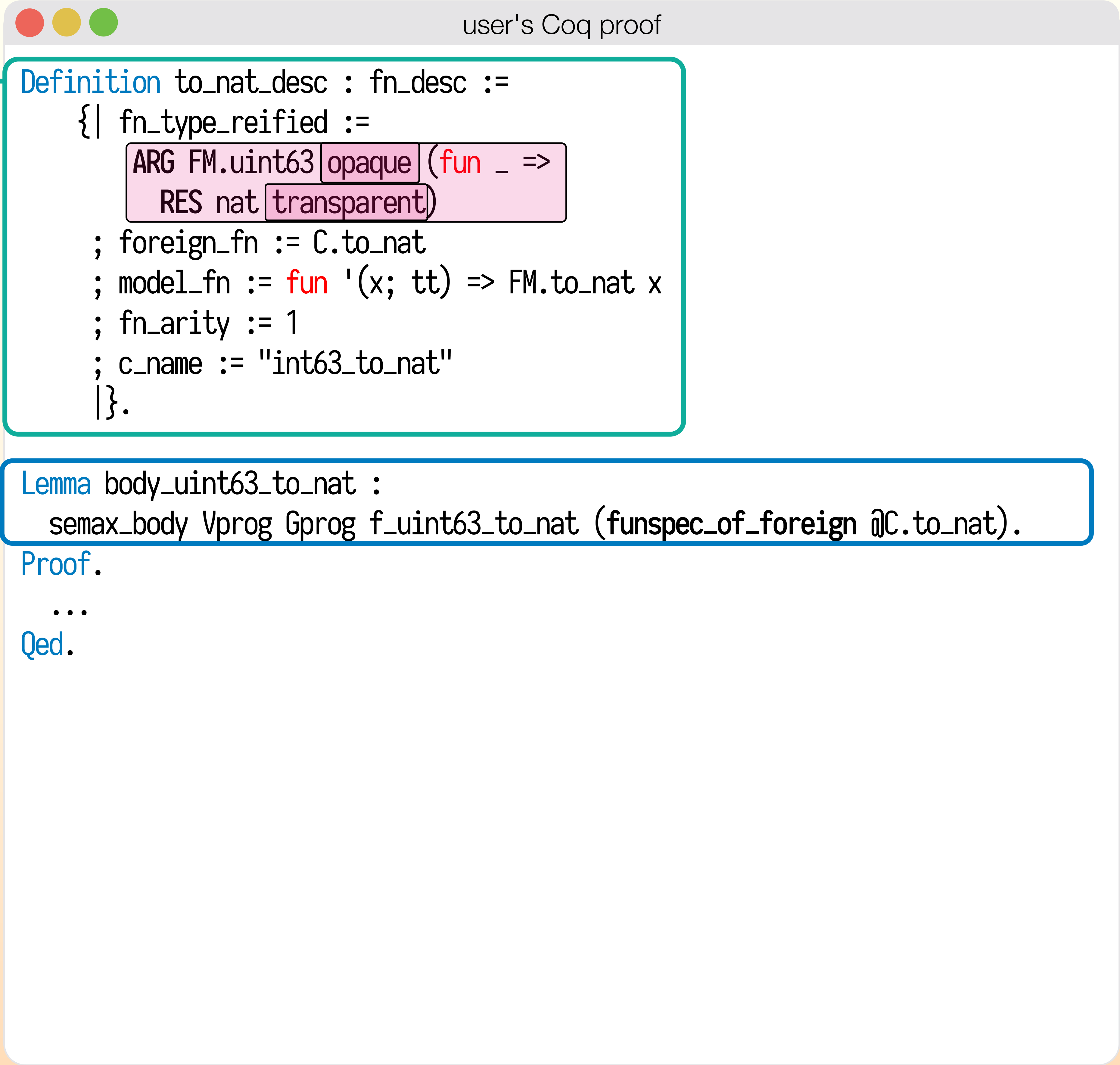
We claim that the function body satisfies this spec.



```
Definition uint63_to_nat_spec : ident * funspec :=
  DECLARE _uint63_to_nat
  WITH gv : gvars, g : graph, roots : roots_t, sh : share, x : {_: FM.uint63 & unit},
       p : rep_type, ti : val, outlier : outlier_t, t_info : thread_info
  PRE [ thread_info; int_or_ptr_type ]
    PROP (writable_share sh; @graph_predicate FM.uint63 g outlier (projT1 x) p)
    PARAMS (ti, rep_type_val g p)
    GLOBALS (gv)
    SEP (full_gc g t_info roots outlier ti sh gv; mem_mgr gv)
  POST [ int_or_ptr_type ]
    EX (p' : rep_type) (g' : graph) (roots': roots_t) (t_info': thread_info),
      PROP (@graph_predicate nat g' outlier ( FM.to_nat (projT1 x) ) p’;
            gc_graph_iso g roots g' roots’;
            frame_shells_eq (ti_frames t_info) (ti_frames t_info’))
      RETURN (rep_type_val g' p’)
      SEP (full_gc g' t_info' roots' outlier ti sh gv; mem_mgr gv).


Lemma body_uint63_to_nat :
  semax_body Vprog Gprog f_uint63_to_nat uint63_to_nat_spec.
Proof. ... Qed.
```

function description

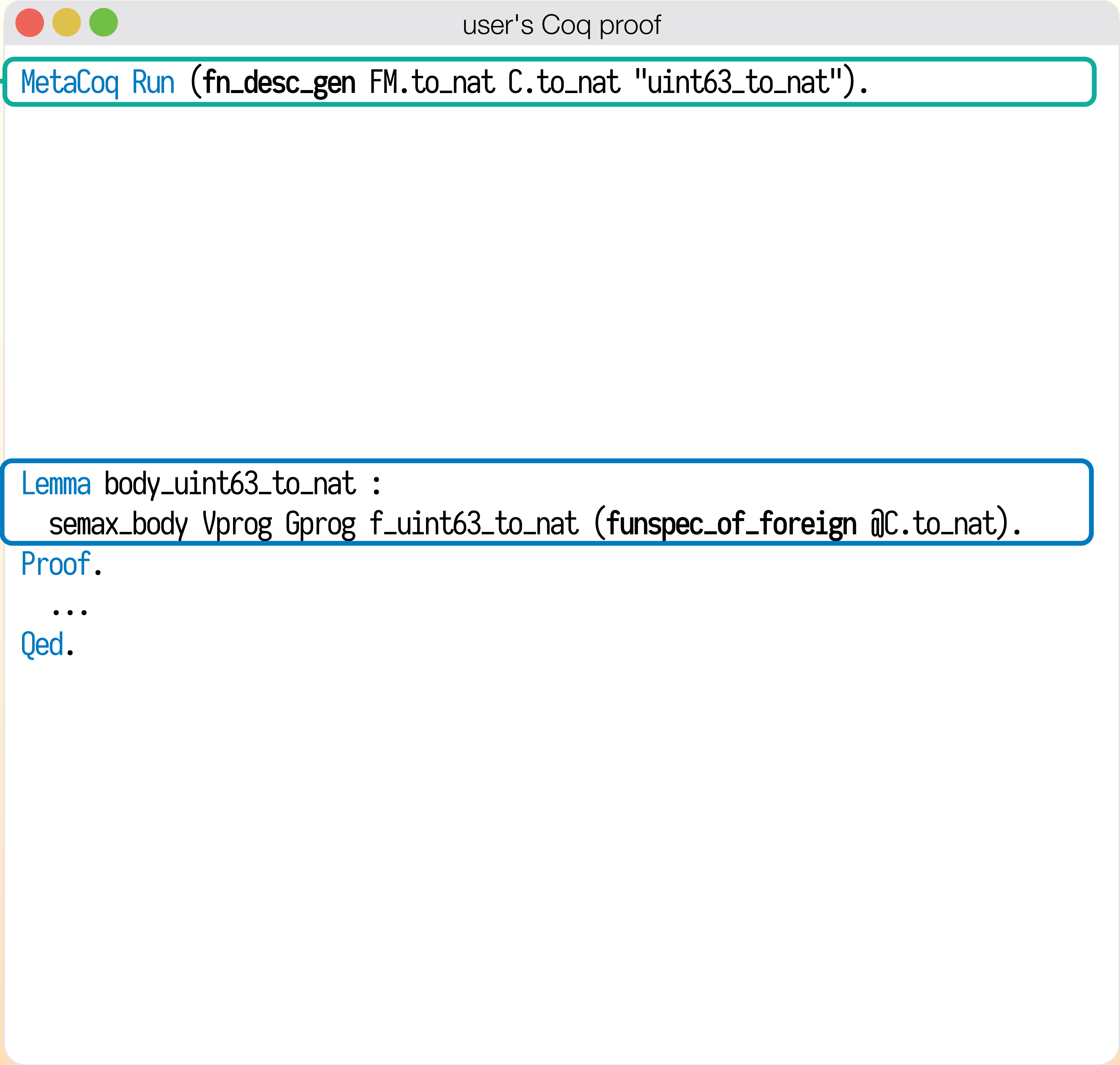
generate function specification



```
Definition to_nat_desc : fn_desc :=
    {| fn_type_reified :=
         ARG FM.uint63 opaque (fun _ =>
           RES nat transparent)
     ; foreign_fn := C.to_nat
     ; model_fn := fun '(x; tt) => FM.to_nat x
     ; fn_arity := 1
     ; c_name := "int63_to_nat"
     |}.

Lemma body_uint63_to_nat :
  semax_body Vprog Gprog f_uint63_to_nat (funspec_of_foreign @C.to_nat).
Proof.
  ...
Qed.
```

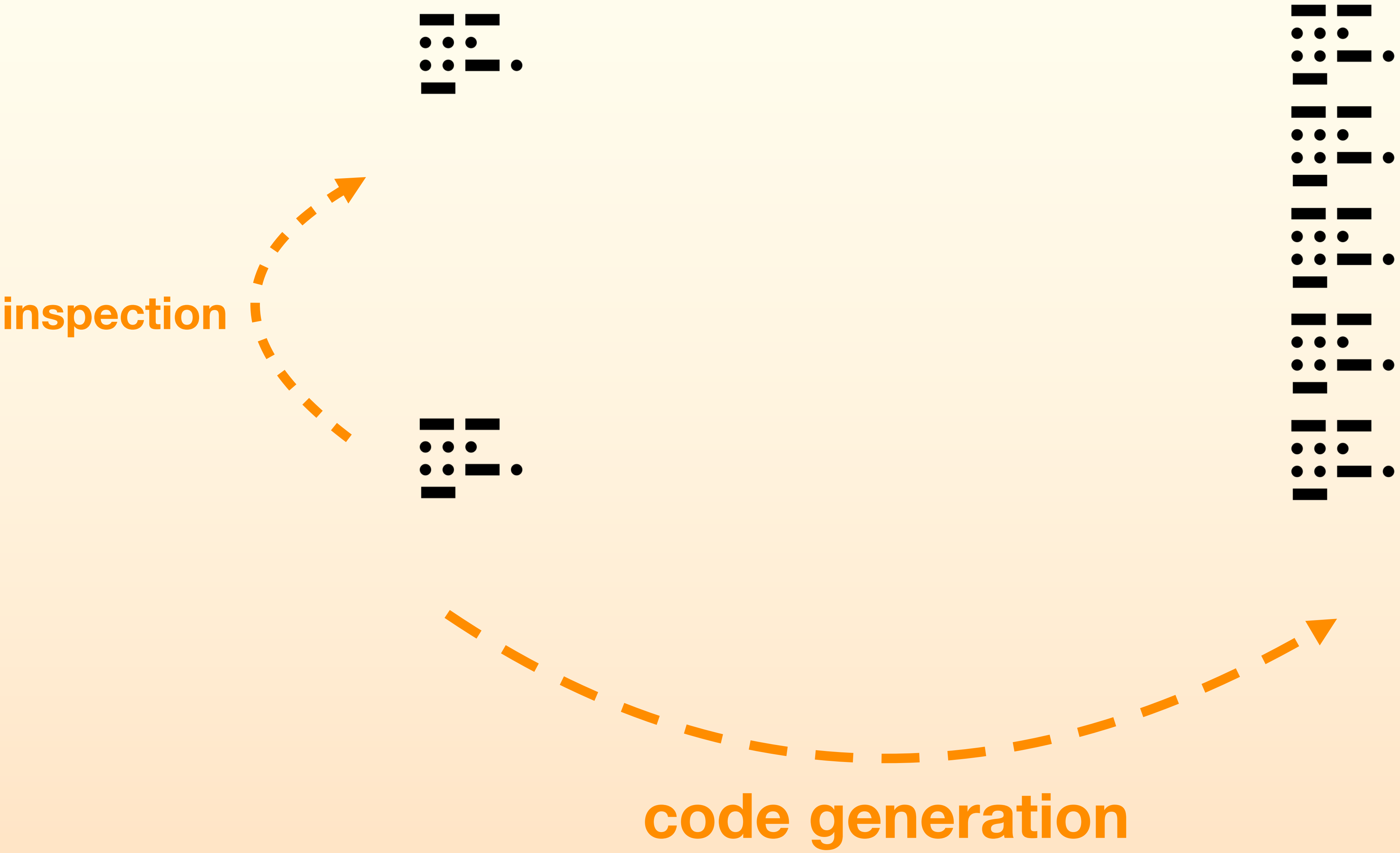
generate function description

generate function specification



```
MetaCoq Run (fn_desc_gen FM.to_nat C.to_nat "uint63_to_nat").









Lemma body_uint63_to_nat :
  semax_body Vprog Gprog f_uint63_to_nat (funspec_of_foreign @C.to_nat).
Proof.
  ...
Qed.
```

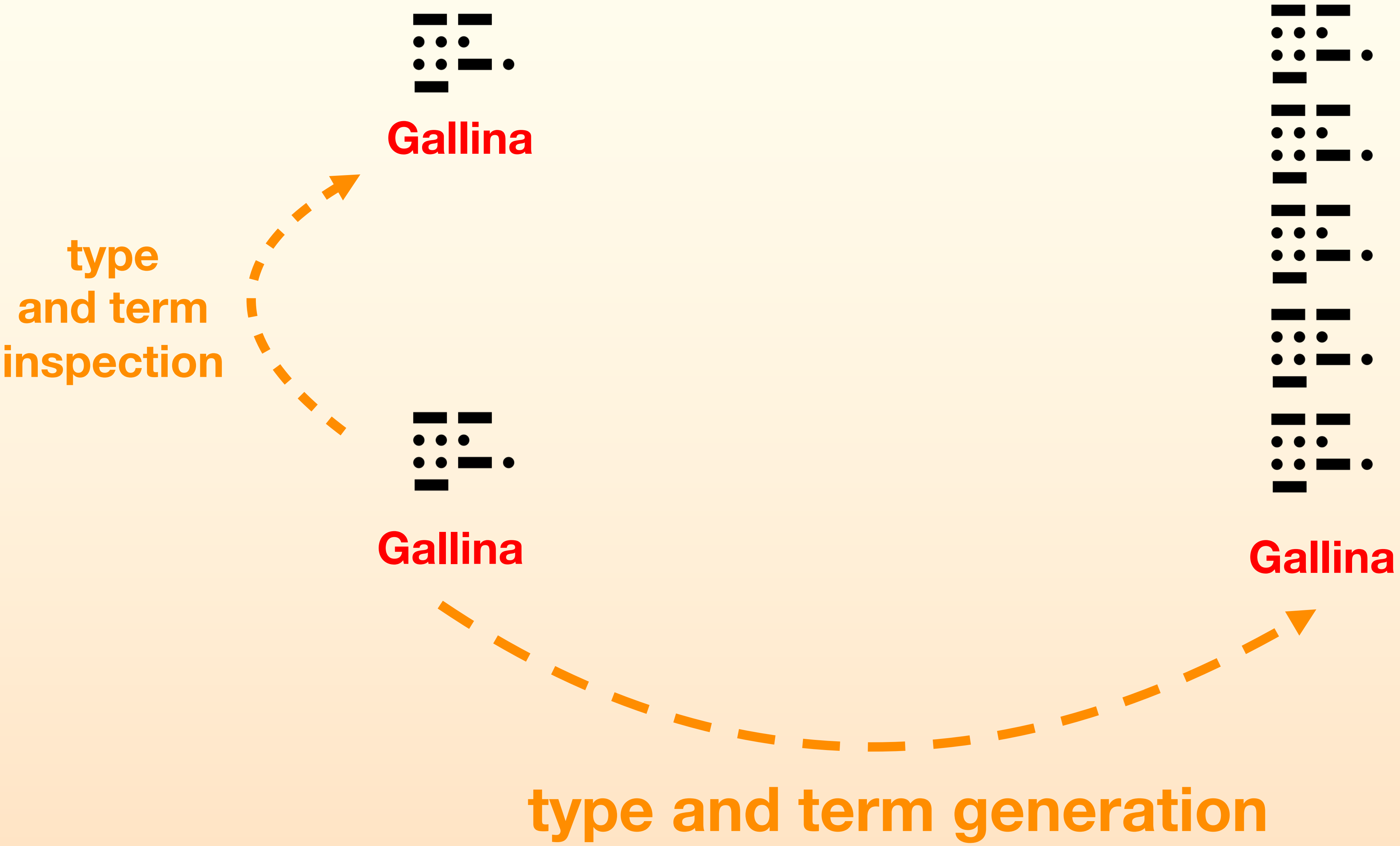
What is metaprogramming?



inspection

code generation

MetaCoq



Gallina

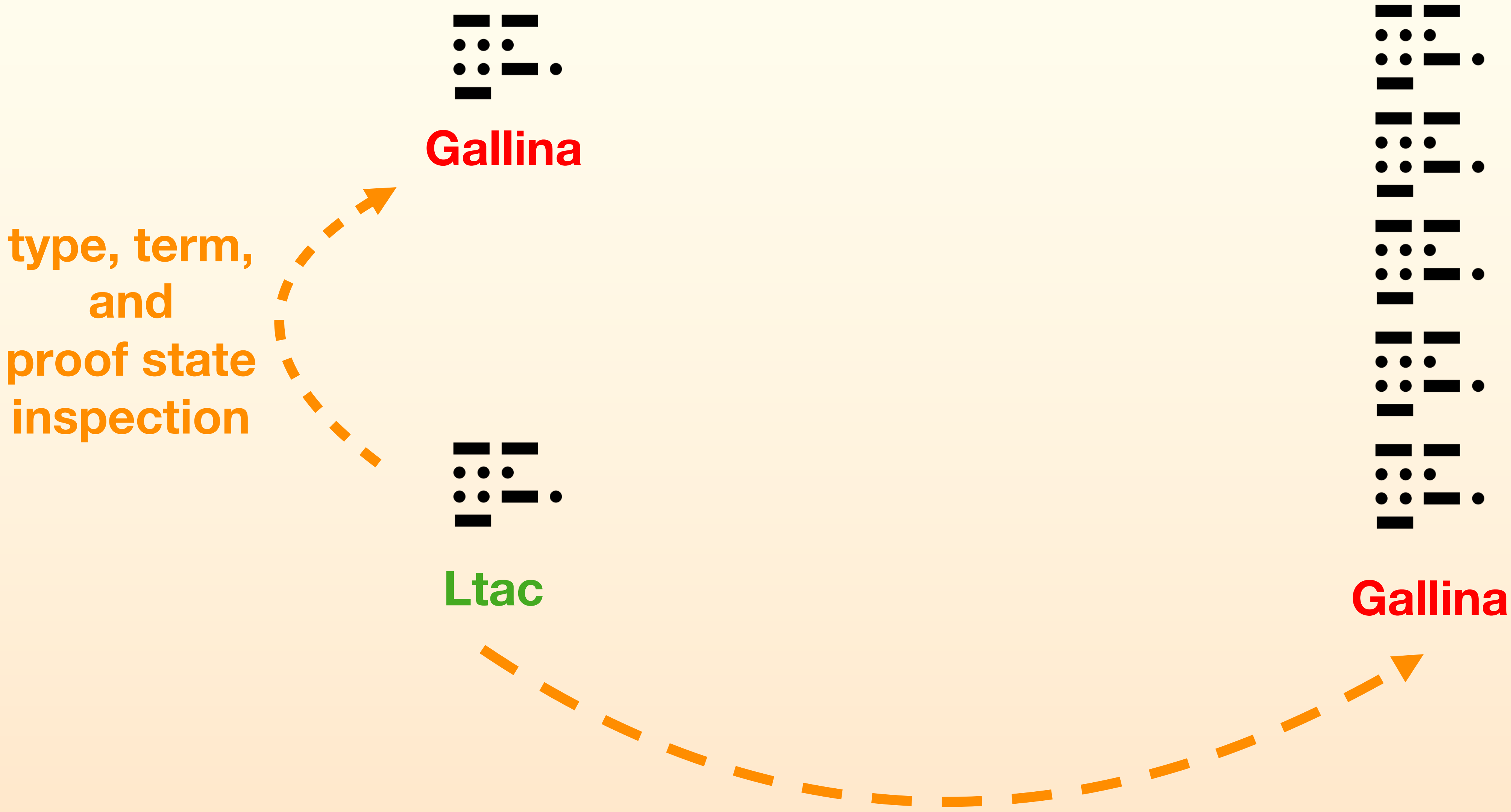
type
and term
inspection

Gallina

Gallina

type and term generation

Ltac



Gallina

type, term,
and
proof state
inspection

Ltac

Gallina

proof term generation

monolithic vs distilled generation

Problems

1. MetaCoq's representation of Coq terms is "**low level**" by design.

- Have to work with De Bruijn indices.
- Cannot have mutually recursive type class instances.
- Recursive calls have to refer to a specific `fix` expression.
- Type class inference has to resolve immediately.
- There is no easy inference based on a context.

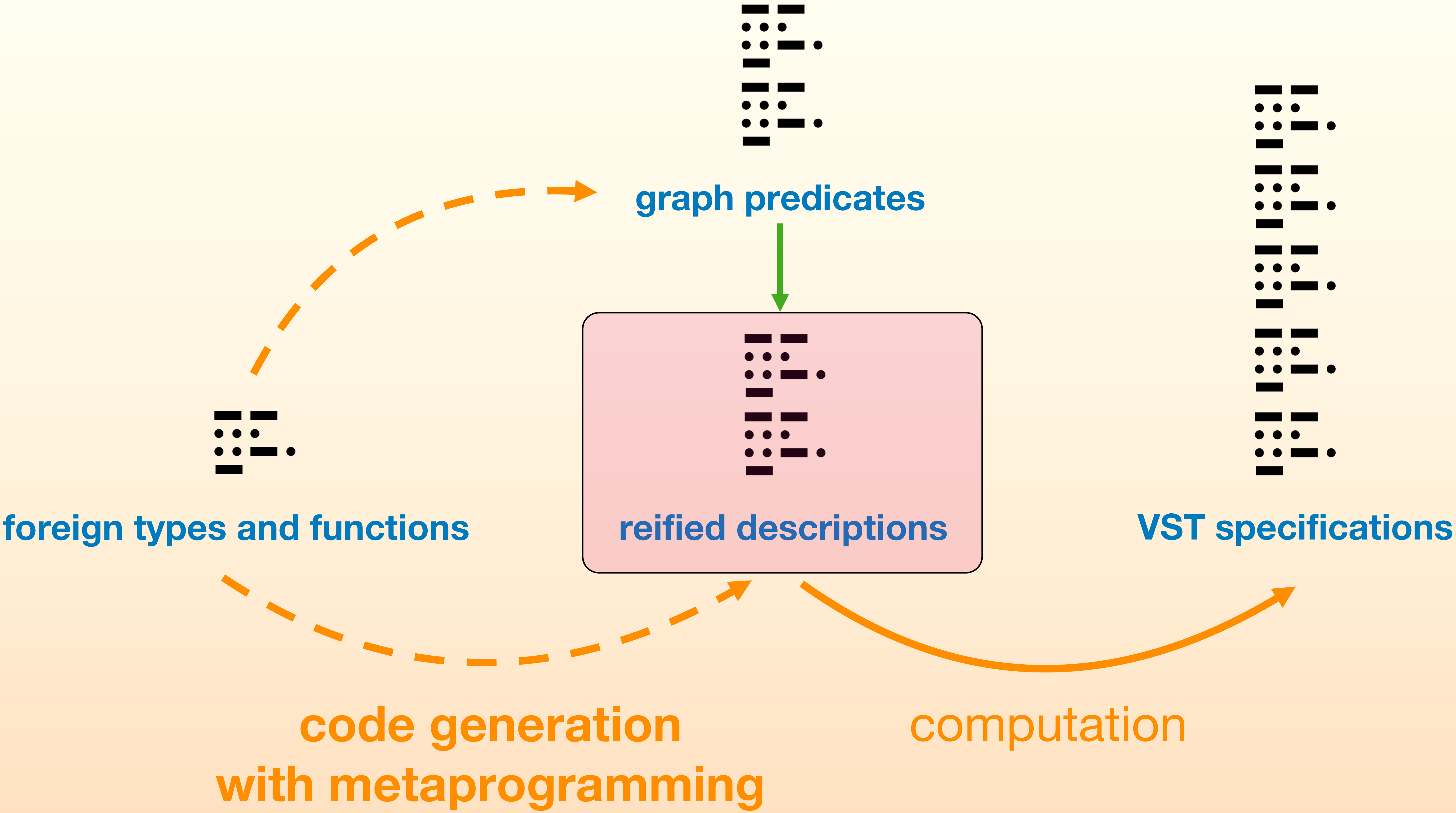
2. Metaprograms are **harder** to reason about!

foreign types and functions

VST specifications

code generation
with metaprogramming

monolithic vs **distilled** generation



graph predicates

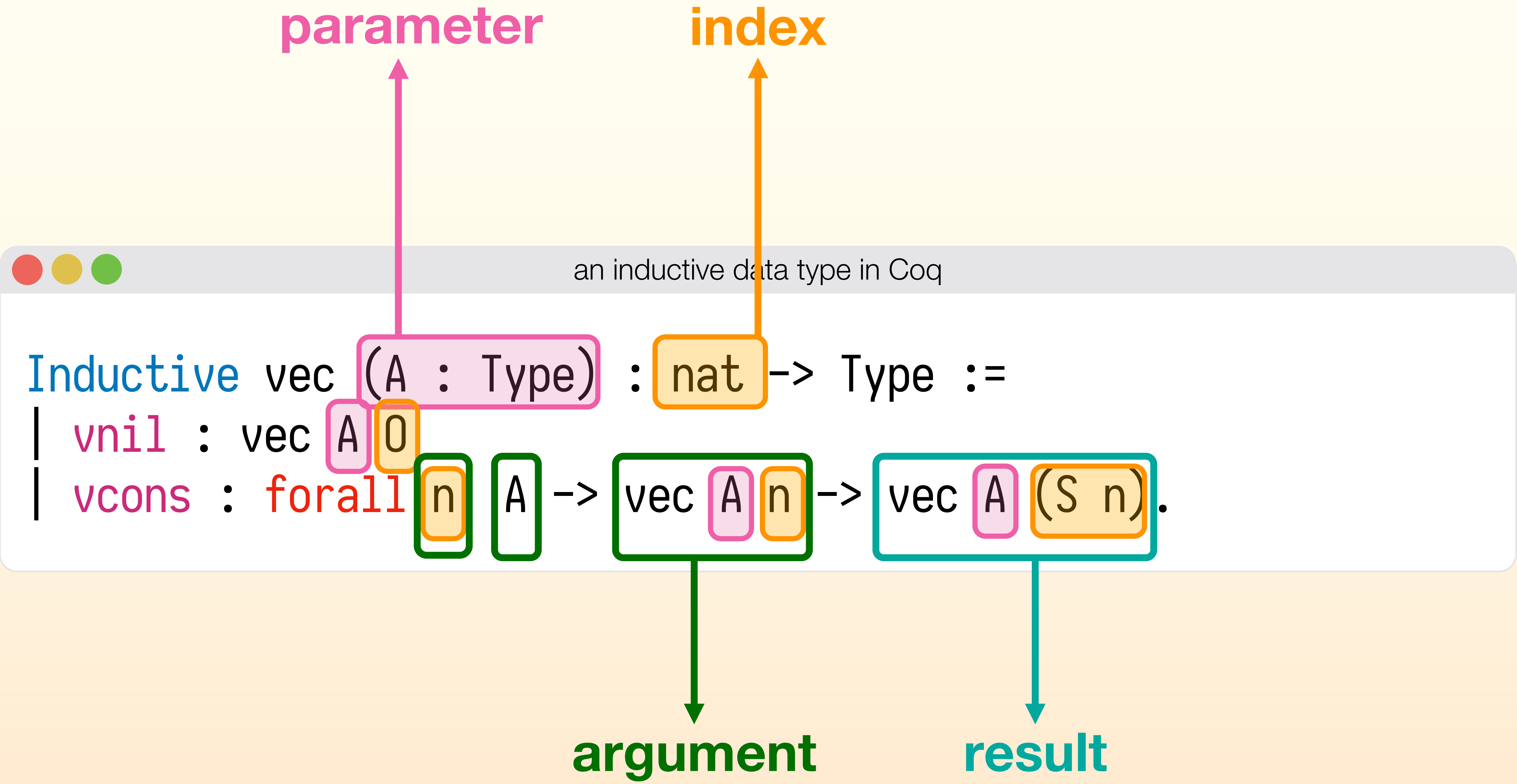
foreign types and functions

reified descriptions

VST specifications

code generation with metaprogramming

computation



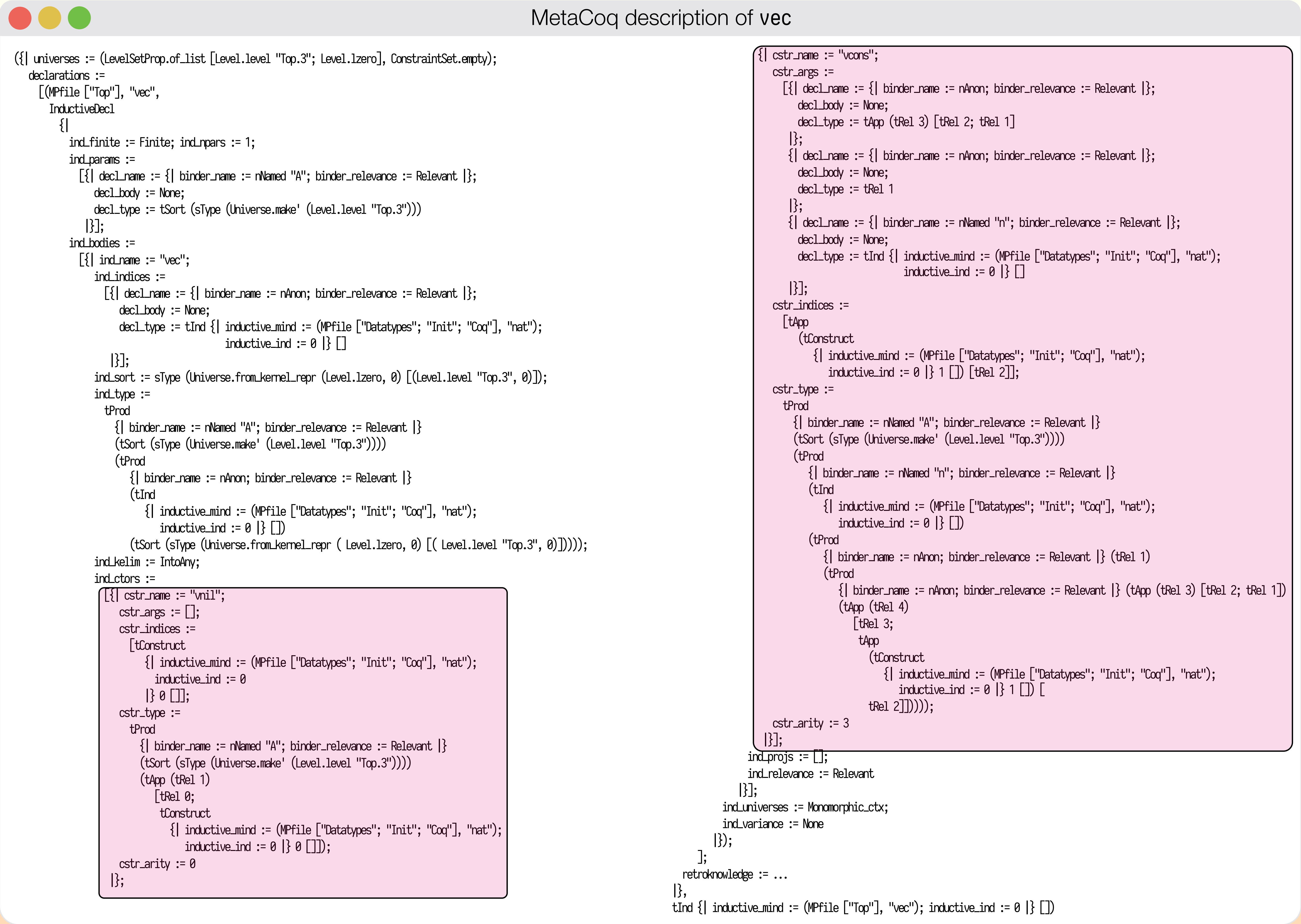
parameter

index

```
Inductive vec (A : Type) : nat -> Type :=
| vnil : vec A O
| vcons : forall n A -> vec A n -> vec A (S n).
```

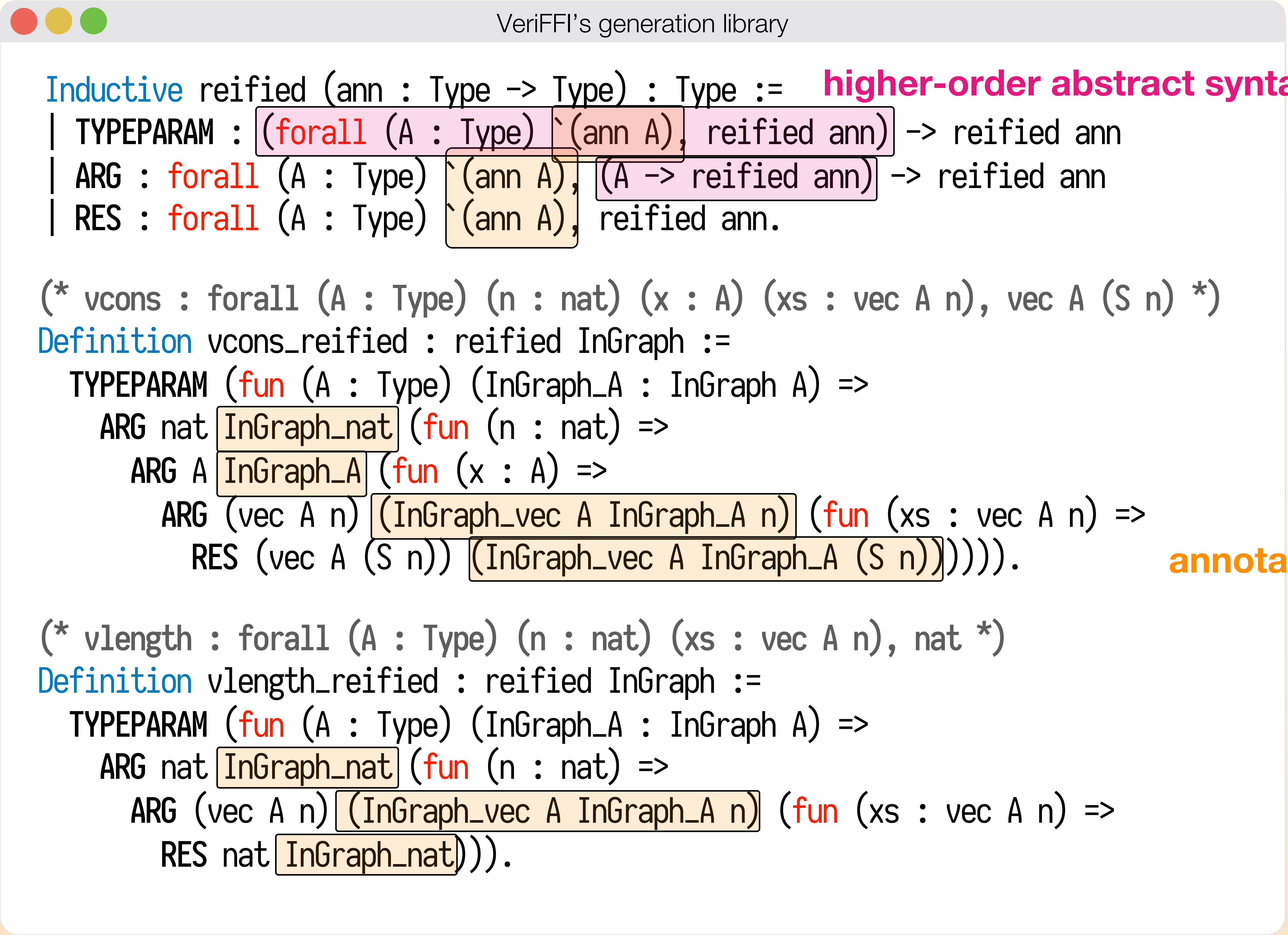
argument

result



MetaCoq description of **vec**

```
({| universes := (LevelSetProp.of_list [Level.level "Top.3"; Level.lzero], ConstraintSet.empty);
  declarations :=
    [(MPfile ["Top"], "vec",
      InductiveDecl
        {|
          ind_finite := Finite; ind_npars := 1;
          ind_params :=
            [{| decl_name := {| binder_name := nNamed "A"; binder_relevance := Relevant |};
                decl_body := None;
                decl_type := tSort (sType (Universe.make' (Level.level "Top.3")))
              |}];
          ind_bodies :=
            [{| ind_name := "vec";
                ind_indices :=
                  [{| decl_name := {| binder_name := nAnon; binder_relevance := Relevant |};
                      decl_body := None;
                      decl_type := tInd {| inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"], "nat");
                                           inductive_ind := 0 |} []
                    |}];
                ind_sort := sType (Universe.from_kernel_repr (Level.lzero, 0) [(Level.level "Top.3", 0)]);
                ind_type :=
                  tProd
                    {| binder_name := nNamed "A"; binder_relevance := Relevant |}
                    (tSort (sType (Universe.make' (Level.level "Top.3"))))
                    (tProd
                       {| binder_name := nAnon; binder_relevance := Relevant |}
                       (tInd
                          {| inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"], "nat");
                             inductive_ind := 0 |} [])
                       (tSort (sType (Universe.from_kernel_repr ( Level.lzero, 0) [( Level.level "Top.3", 0)]))));
                ind_kelim := IntoAny;
                ind_ctors :=
                  [{| cstr_name := "vnil";
                      cstr_args := [];
                      cstr_indices :=
                        [tConstruct
                           {| inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"], "nat");
                              inductive_ind := 0
                           |} 0 []];
                      cstr_type :=
                        tProd
                          {| binder_name := nNamed "A"; binder_relevance := Relevant |}
                          (tSort (sType (Universe.make' (Level.level "Top.3"))))
                          (tApp (tRel 1)
                             [tRel 0;
                              tConstruct
                                {| inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"], "nat");
                                   inductive_ind := 0 |} 0 []]);
                      cstr_arity := 0
                    |};
                   {| cstr_name := "vcons";
                      cstr_args :=
                        [{| decl_name := {| binder_name := nAnon; binder_relevance := Relevant |};
                            decl_body := None;
                            decl_type := tApp (tRel 3) [tRel 2; tRel 1]
                          |};
                          {| decl_name := {| binder_name := nAnon; binder_relevance := Relevant |};
                            decl_body := None;
                            decl_type := tRel 1
                          |};
                          {| decl_name := {| binder_name := nNamed "n"; binder_relevance := Relevant |};
                            decl_body := None;
                            decl_type := tInd {| inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"], "nat");
                                                 inductive_ind := 0 |} []
                          |}];
                      cstr_indices :=
                        [tApp
                           (tConstruct
                              {| inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"], "nat");
                                 inductive_ind := 0 |} 1 []) [tRel 2]];
                      cstr_type :=
                        tProd
                          {| binder_name := nNamed "A"; binder_relevance := Relevant |}
                          (tSort (sType (Universe.make' (Level.level "Top.3"))))
                          (tProd
                             {| binder_name := nNamed "n"; binder_relevance := Relevant |}
                             (tInd
                                {| inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"], "nat");
                                   inductive_ind := 0 |} [])
                             (tProd
                                {| binder_name := nAnon; binder_relevance := Relevant |} (tRel 1)
                                (tProd
                                   {| binder_name := nAnon; binder_relevance := Relevant |} (tApp (tRel 3) [tRel 2; tRel 1])
                                   (tApp (tRel 4)
                                      [tRel 3;
                                       tApp
                                         (tConstruct
                                            {| inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"], "nat");
                                               inductive_ind := 0 |} 1 []) [
                                         tRel 2]])))));
                      cstr_arity := 3
                    |}];
                ind_projs := [];
                ind_relevance := Relevant
              |}];
          ind_universes := Monomorphic_ctx;
          ind_variance := None
        |});
    ];
  retroknowledge := ...
|},
tInd {| inductive_mind := (MPfile ["Top"], "vec"); inductive_ind := 0 |} [])
```



```
Inductive reified (ann : Type -> Type) : Type :=
| TYPEPARAM : (forall (A : Type) `(ann A), reified ann) -> reified ann
| ARG : forall (A : Type) `(ann A), (A -> reified ann) -> reified ann
| RES : forall (A : Type) `(ann A), reified ann.

(* vcons : forall (A : Type) (n : nat) (x : A) (xs : vec A n), vec A (S n) *)
Definition vcons_reified : reified InGraph :=
  TYPEPARAM (fun (A : Type) (InGraph_A : InGraph A) =>
    ARG nat InGraph_nat (fun (n : nat) =>
      ARG A InGraph_A (fun (x : A) =>
        ARG (vec A n) (InGraph_vec A InGraph_A n) (fun (xs : vec A n) =>
          RES (vec A (S n)) (InGraph_vec A InGraph_A (S n)))))).

(* vlength : forall (A : Type) (n : nat) (xs : vec A n), nat *)
Definition vlength_reified : reified InGraph :=
  TYPEPARAM (fun (A : Type) (InGraph_A : InGraph A) =>
    ARG nat InGraph_nat (fun (n : nat) =>
      ARG (vec A n) (InGraph_vec A InGraph_A n) (fun (xs : vec A n) =>
        RES nat InGraph_nat))).
```

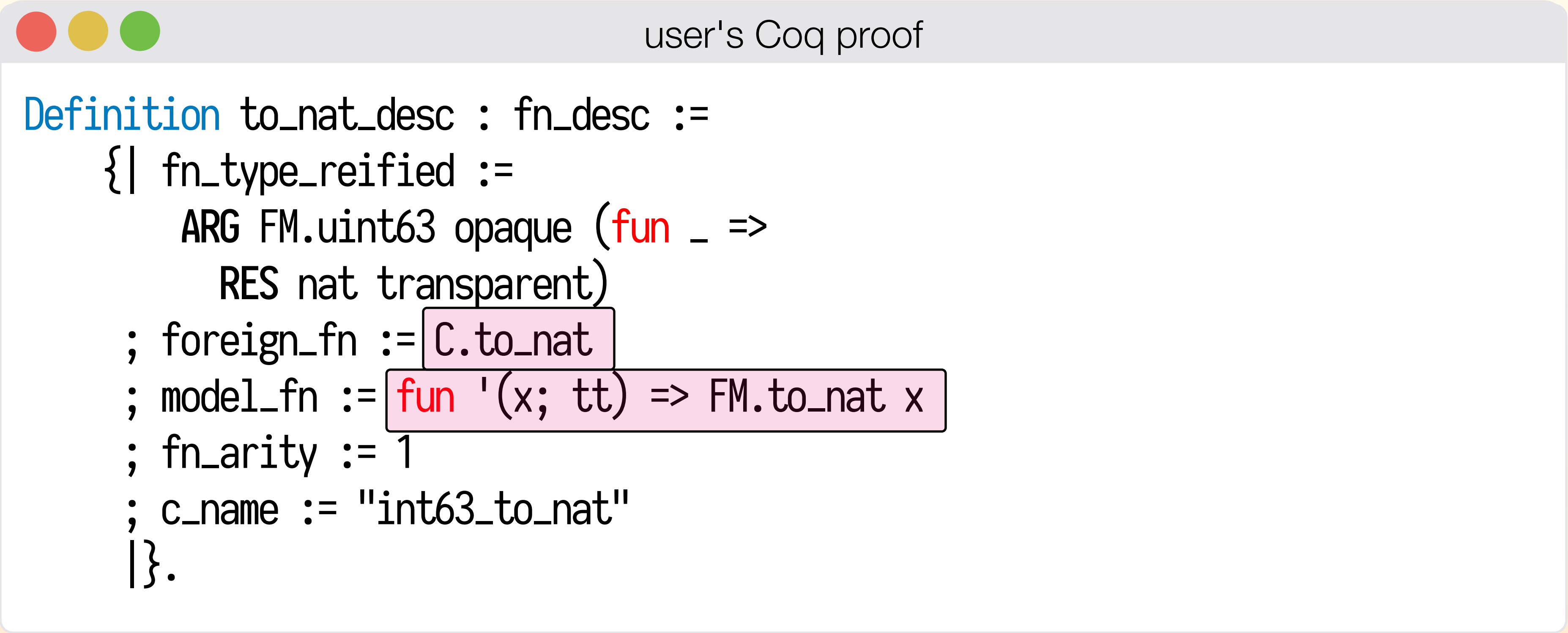
higher-order abstract syntax-ish

annotations

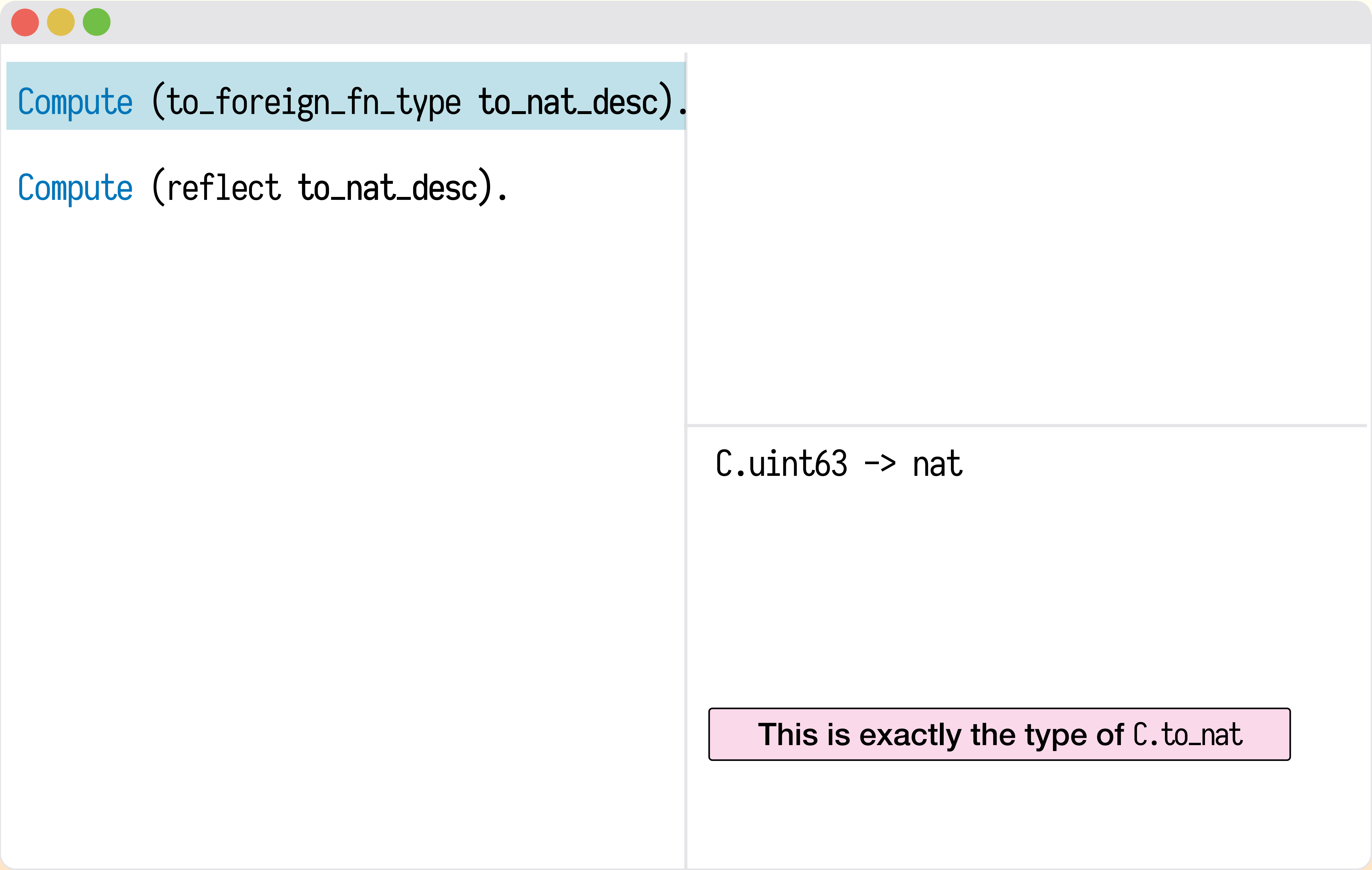
For other mixes of deep and shallow embeddings, see:
“Outrageous But Meaningful Coincidences: Dependent Type-Safe Syntax and Evaluation”. McBride. 2010.
"Deeper Shallow Embeddings". Prinz, Kavvos, Lampropoulos. 2022.

What do reified descriptions buy us?

1. type safety



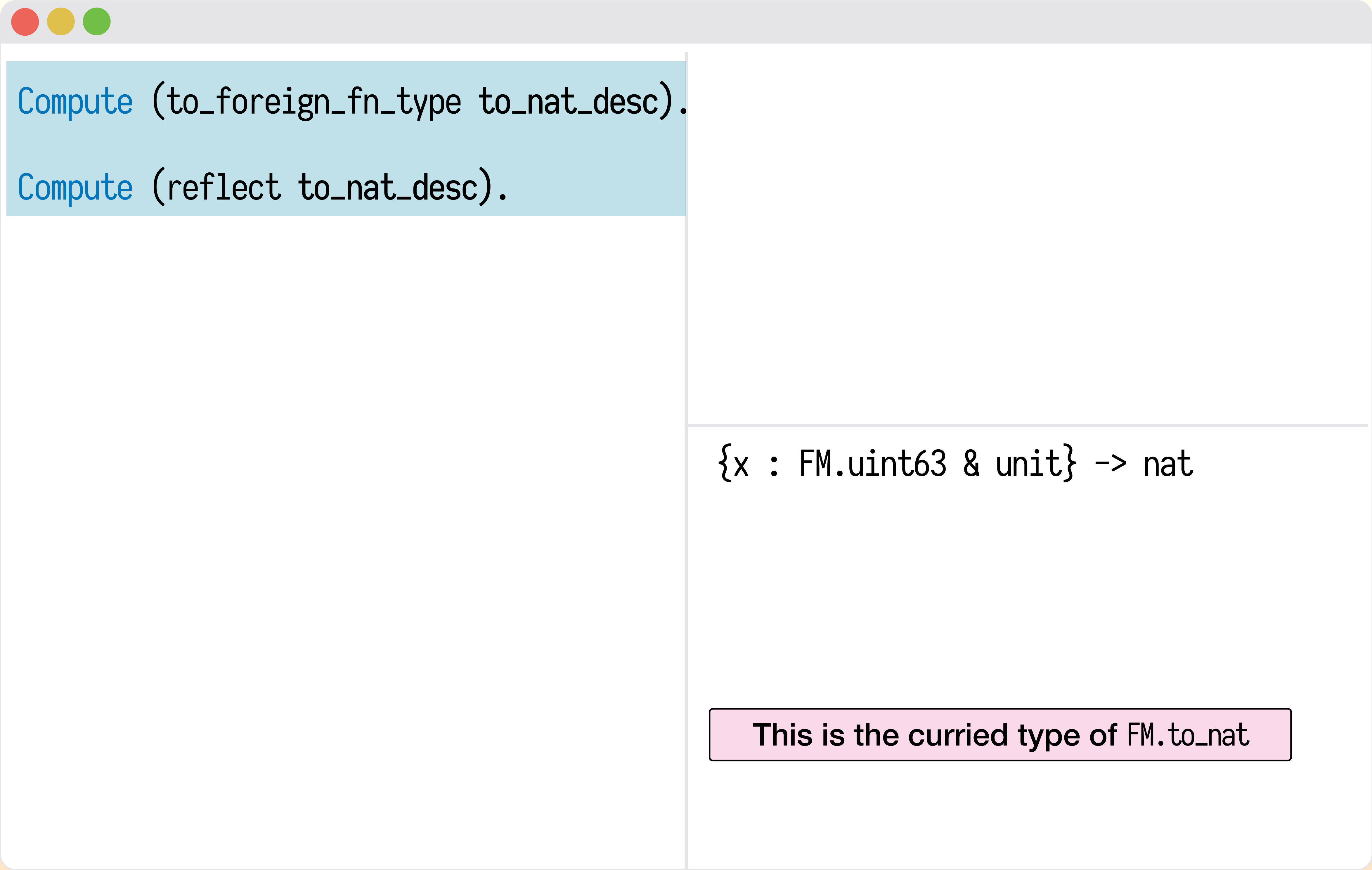
```
Definition to_nat_desc : fn_desc :=
    {| fn_type_reified :=
        ARG FM.uint63 opaque (fun _ =>
          RES nat transparent)
     ; foreign_fn := C.to_nat
     ; model_fn := fun '(x; tt) => FM.to_nat x
     ; fn_arity := 1
     ; c_name := "int63_to_nat"
     |}.
```



```
Compute (to_foreign_fn_type to_nat_desc).

Compute (reflect to_nat_desc).
```

```
C.uint63 -> nat
```

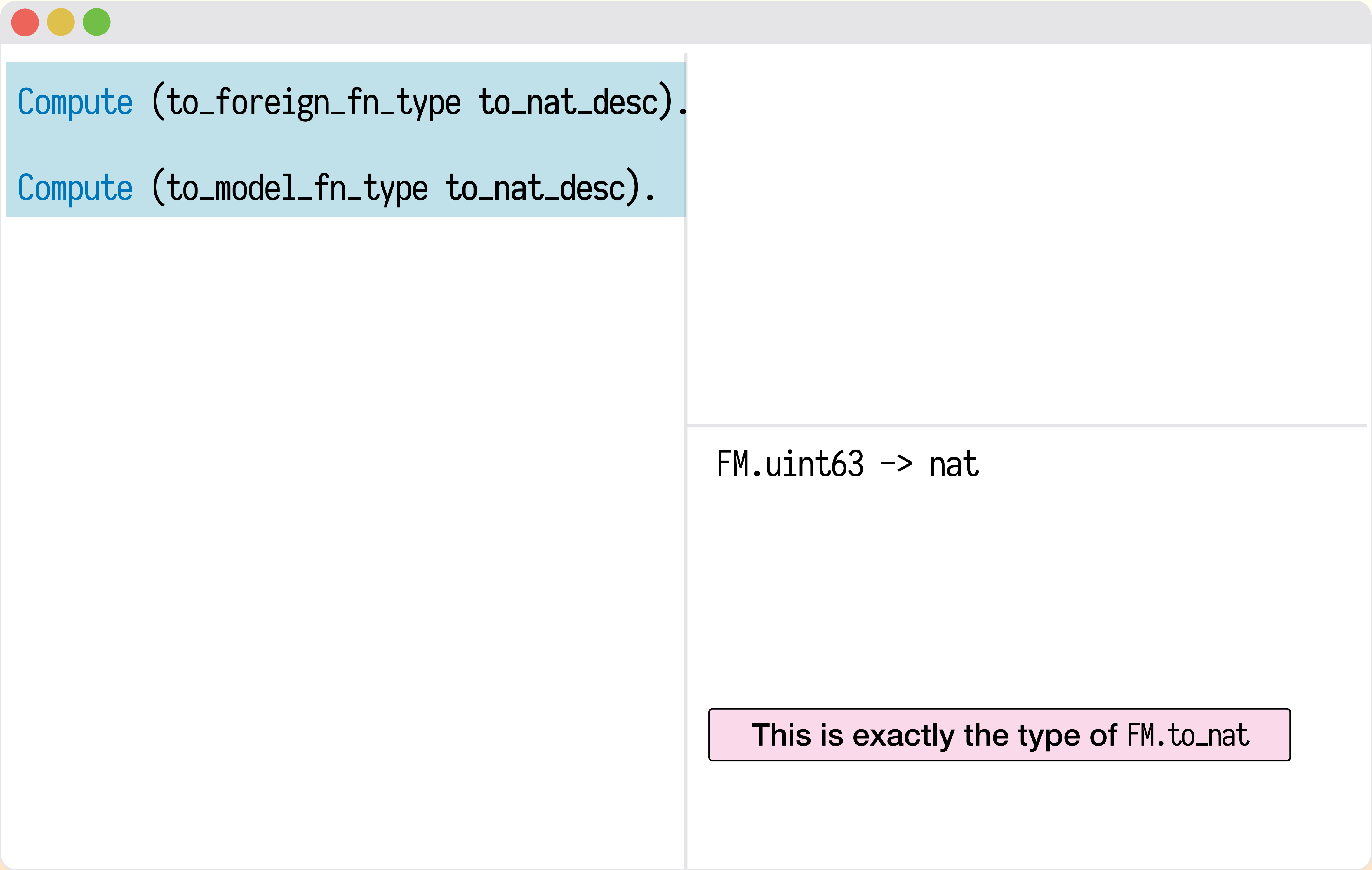
This is exactly the type of `C.to_nat`



```
Compute (to_foreign_fn_type to_nat_desc).

Compute (reflect to_nat_desc).
```

```
{x : FM.uint63 & unit} -> nat
```

This is the curried type of `FM.to_nat`

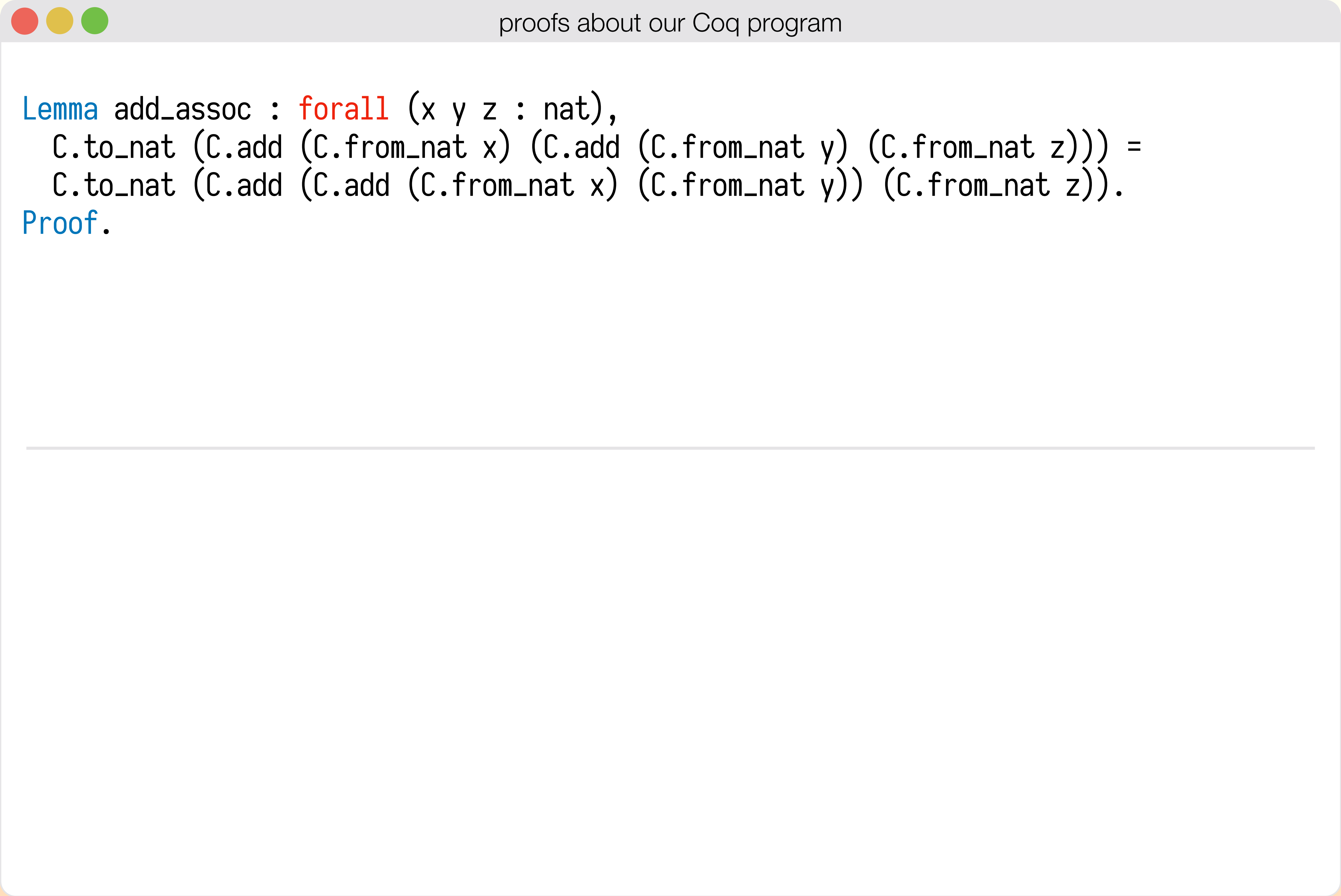


```
Compute (to_foreign_fn_type to_nat_desc).

Compute (to_model_fn_type to_nat_desc).
```

```
FM.uint63 -> nat
```

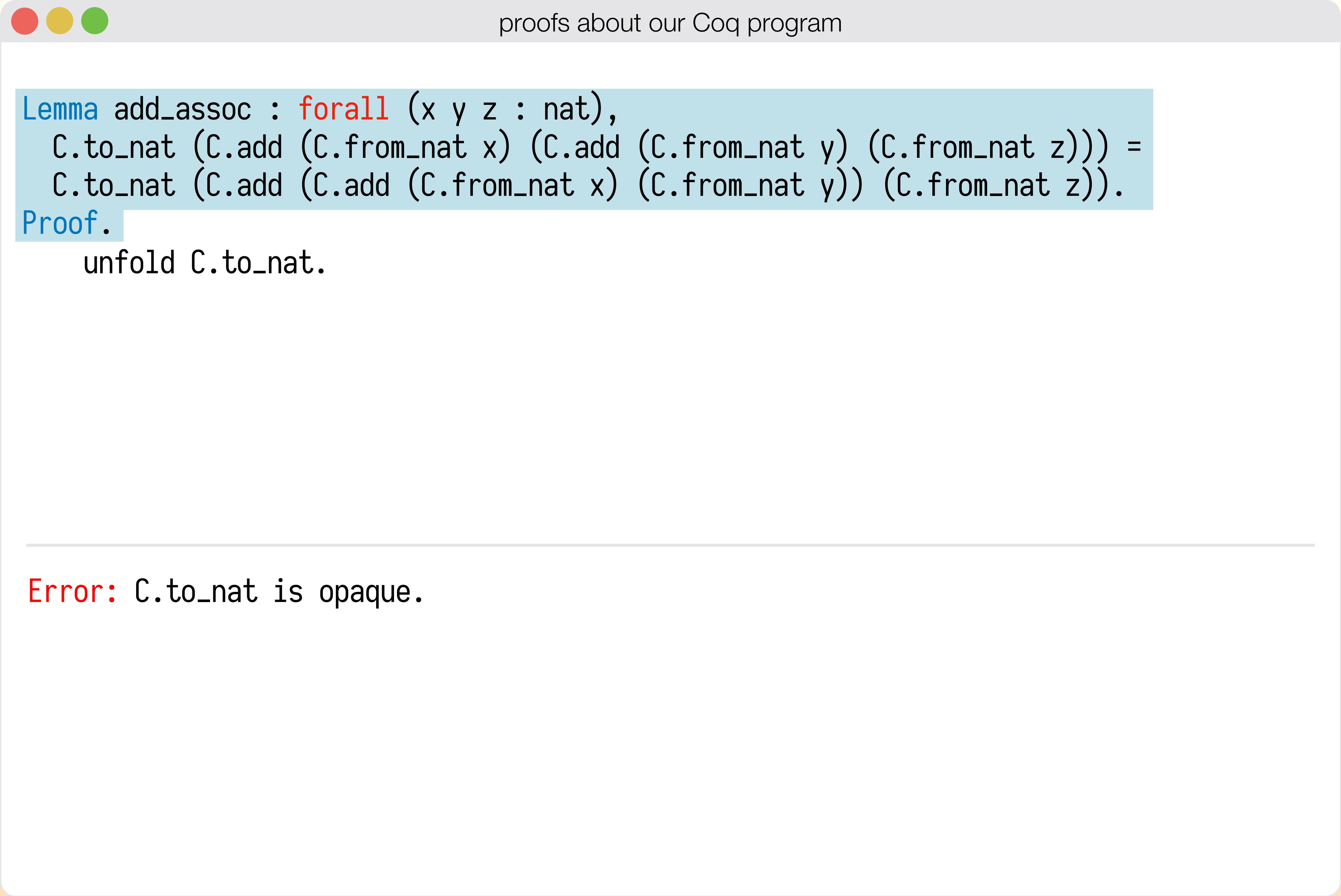
This is exactly the type of `FM.to_nat`

2. rewrites of primitives to models



```
Lemma add_assoc : forall (x y z : nat),
  C.to_nat (C.add (C.from_nat x) (C.add (C.from_nat y) (C.from_nat z))) =
  C.to_nat (C.add (C.add (C.from_nat x) (C.from_nat y)) (C.from_nat z)).
Proof.
```

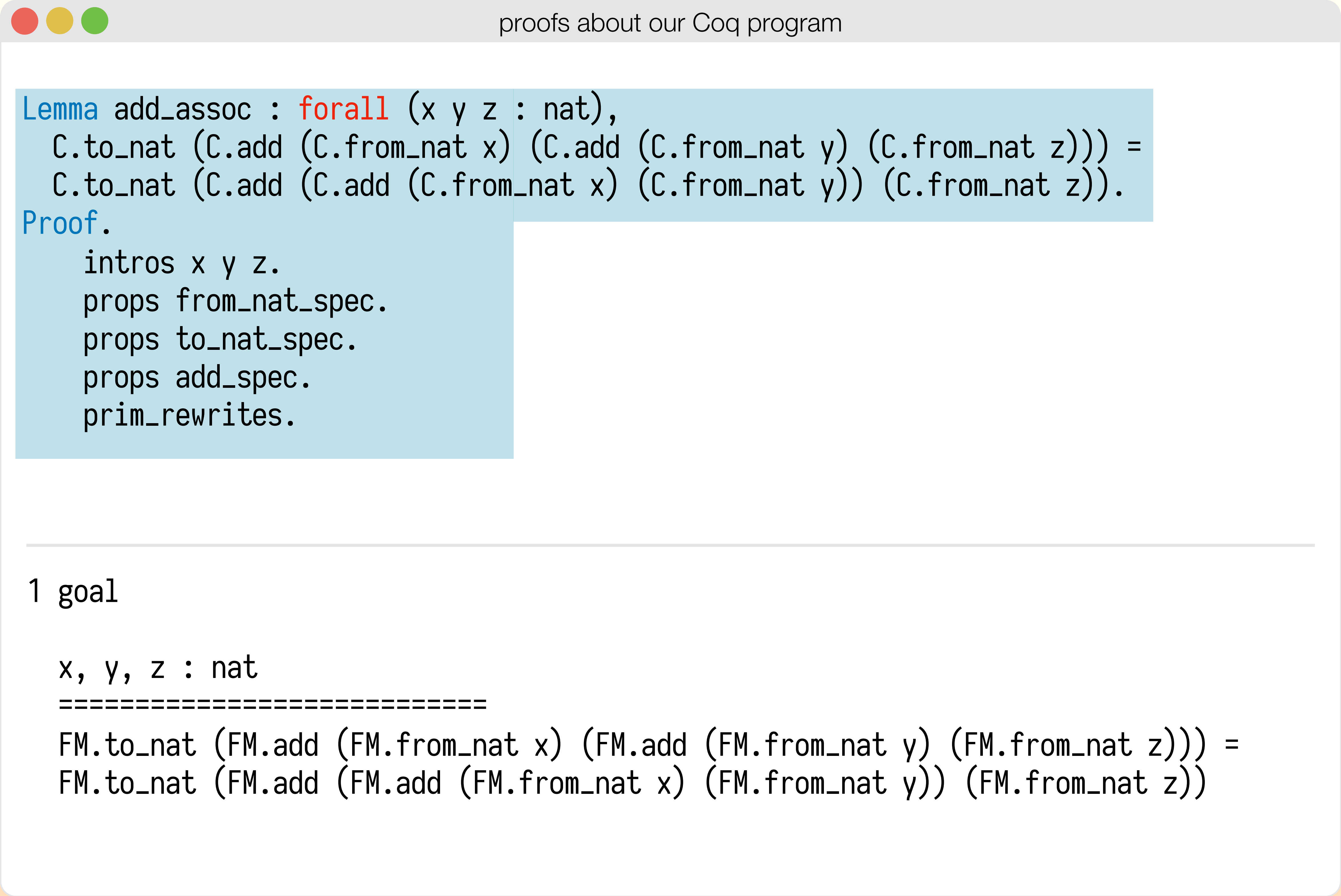
2. rewrites of primitives to models



```
Lemma add_assoc : forall (x y z : nat),
  C.to_nat (C.add (C.from_nat x) (C.add (C.from_nat y) (C.from_nat z))) =
  C.to_nat (C.add (C.add (C.from_nat x) (C.from_nat y)) (C.from_nat z)).
Proof.
    unfold C.to_nat.
```

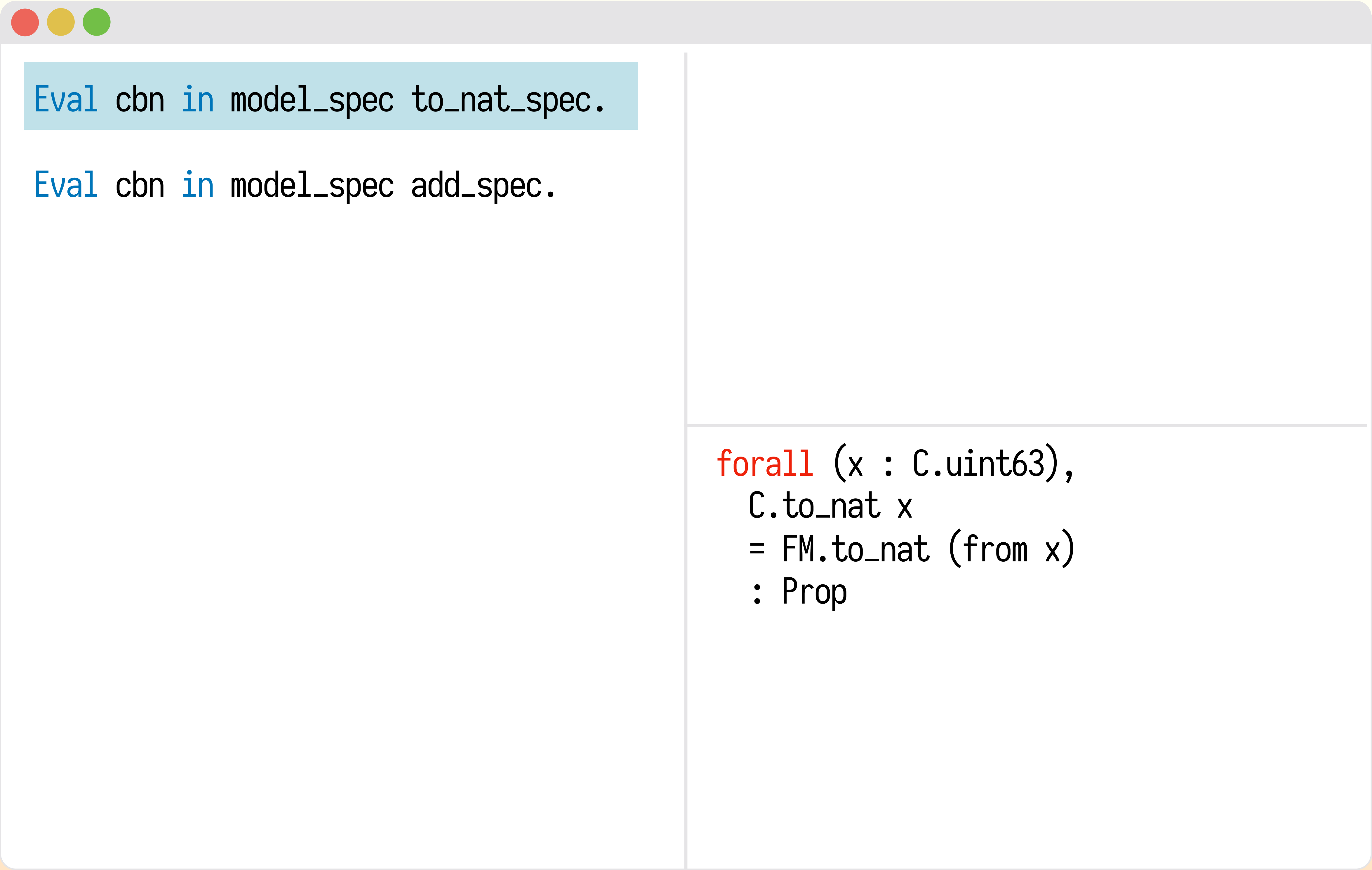
```
Error: C.to_nat is opaque.
```

2. rewrites of primitives to models



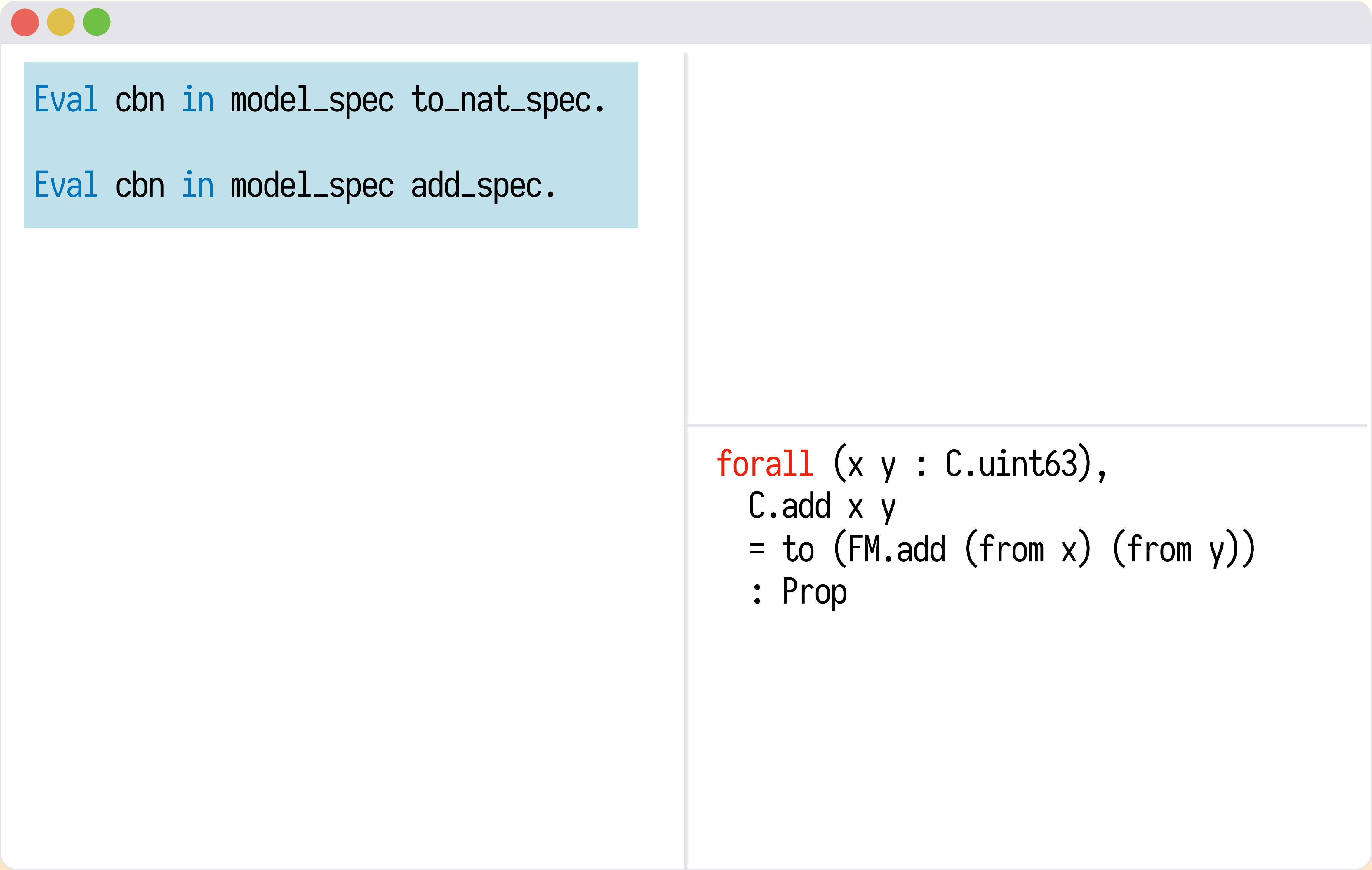
```
Lemma add_assoc : forall (x y z : nat),
  C.to_nat (C.add (C.from_nat x) (C.add (C.from_nat y) (C.from_nat z))) =
  C.to_nat (C.add (C.add (C.from_nat x) (C.from_nat y)) (C.from_nat z)).
Proof.
    intros x y z.
    props from_nat_spec.
    props to_nat_spec.
    props add_spec.
    prim_rewrites.
```

```
1 goal

  x, y, z : nat
  ============================
  FM.to_nat (FM.add (FM.from_nat x) (FM.add (FM.from_nat y) (FM.from_nat z))) =
  FM.to_nat (FM.add (FM.add (FM.from_nat x) (FM.from_nat y)) (FM.from_nat z))
```



```
Eval cbn in model_spec to_nat_spec.

Eval cbn in model_spec add_spec.
```

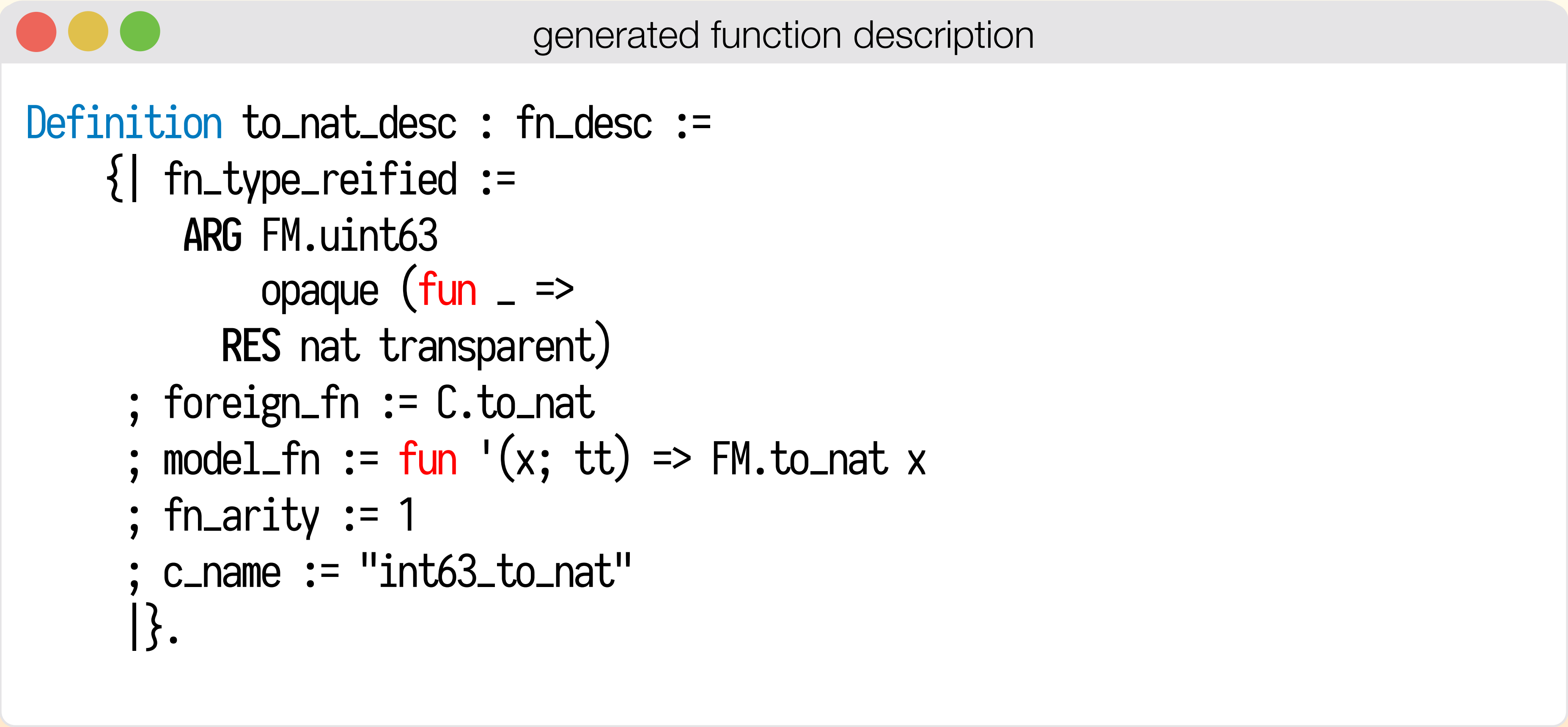
```
forall (x : C.uint63),
  C.to_nat x
  = FM.to_nat (from x)
  : Prop
```



```
Eval cbn in model_spec to_nat_spec.

Eval cbn in model_spec add_spec.
```

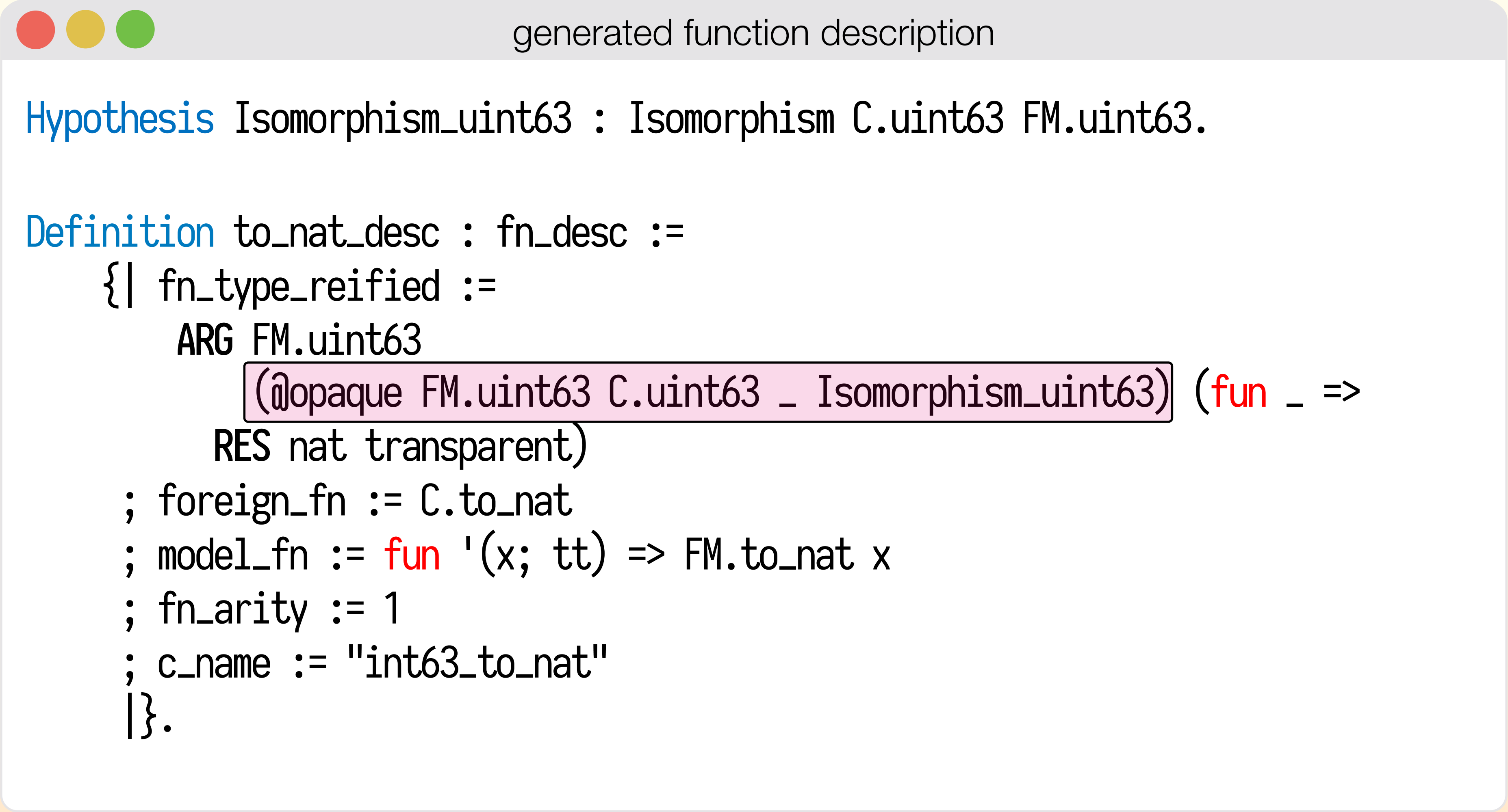
```
forall (x y : C.uint63),
  C.add x y
  = to (FM.add (from x) (from y))
  : Prop
```

An isomorphism
between **the foreign type** and **the model type**



```
Definition to_nat_desc : fn_desc :=
    {| fn_type_reified :=
        ARG FM.uint63
            opaque (fun _ =>
          RES nat transparent)
     ; foreign_fn := C.to_nat
     ; model_fn := fun '(x; tt) => FM.to_nat x
     ; fn_arity := 1
     ; c_name := "int63_to_nat"
     |}.
```

An isomorphism
between **the foreign type** and **the model type**



```
Hypothesis Isomorphism_uint63 : Isomorphism C.uint63 FM.uint63.

Definition to_nat_desc : fn_desc :=
    {| fn_type_reified :=
        ARG FM.uint63
            (@opaque FM.uint63 C.uint63 _ Isomorphism_uint63) (fun _ =>
          RES nat transparent)
     ; foreign_fn := C.to_nat
     ; model_fn := fun '(x; tt) => FM.to_nat x
     ; fn_arity := 1
     ; c_name := "int63_to_nat"
     |}.
```

Comparison with other verified compilers / FFIs

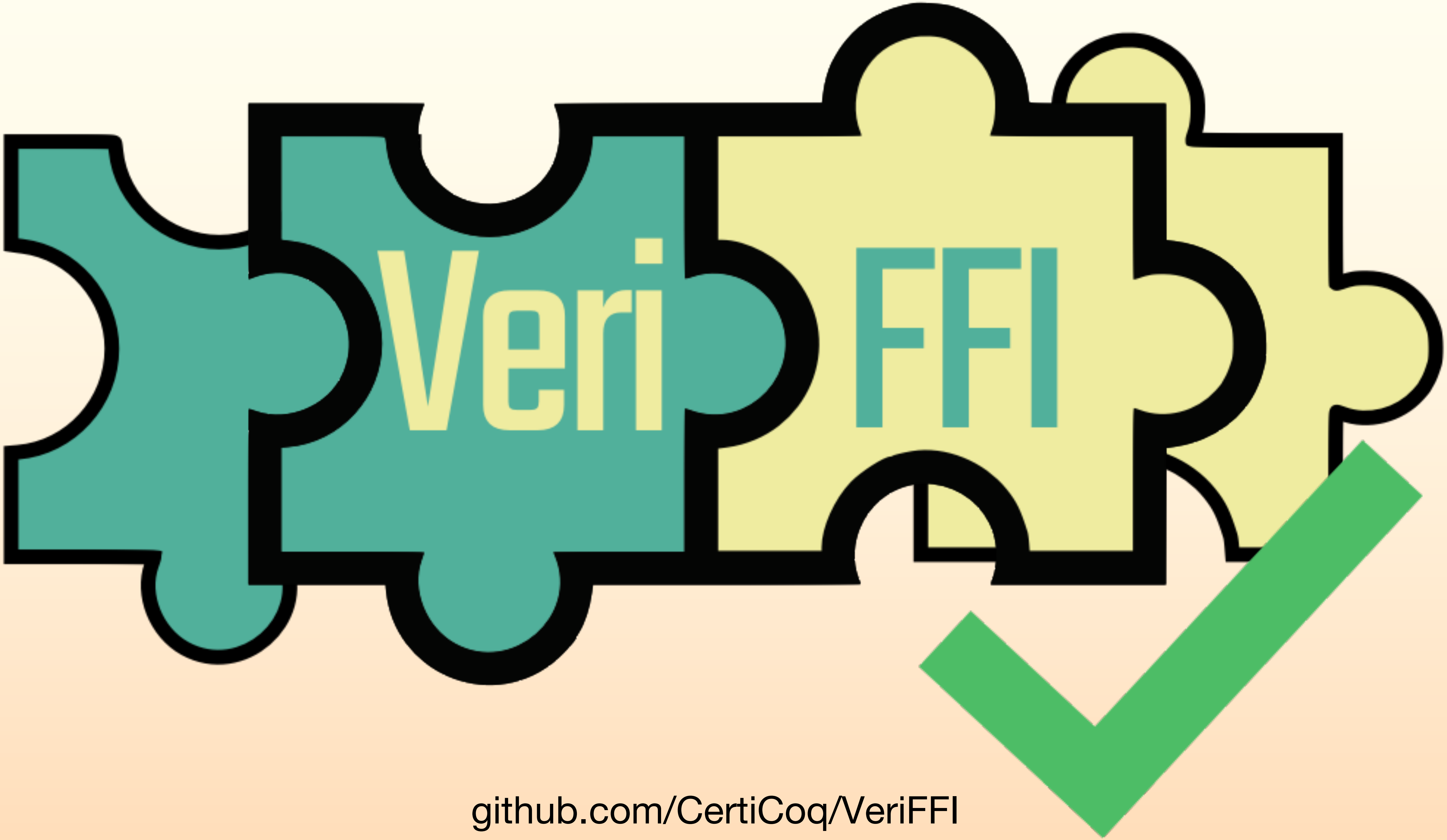
	Œuf (2018)	**Cogent** (2016-2022)	**CakeML** (2014-2019)	**Melocoton** (2023)	**VeriFFI** (2017-2024)
project	verified compiler	*certifying* compiler + verifiable FFI	verified compiler + FFI	verifiable FFI	verified compiler + verifiable FFI
language pair	subset of Coq and C	Cogent and C	ML and C	toy subset of OCaml and toy subset of C	Coq and CompCert C
FFI aims for	-	safety	correctness + safety	correctness + safety	correctness + safety
mechanism	-	-	not a program logic but an oracle about FFIs	Iris's separation logic for multi-language semantics	VST's separation logic
garbage collection	optional external GC	no (unnecessary)	yes (verified)	has a nondeterministic model	yes (verified)

The important scientific contributions of my dissertation are

- Reified descriptions can describe and annotate function types in a concise and type-safe way.
- Given a reified description, we can calculate separation logic specifications about foreign functions that talk about their correctness and safety.
- We can assume an isomorphism between the foreign type and the model type if there’s a module equivalence.

See my dissertation for

- Details of glue code, reified descriptions, function descriptions, constructor descriptions, rewrite principles, and their generation
- Examples, such as primitive bytestrings, I/O and mutable arrays



github.com/CertiCoq/VeriFFI