# Foreign Function Verification Through Metaprogramming

Joomy Korkut
Princeton University

Final Public Oral Examination
October 9th, 2024

1

Hi everyone! Thank you for coming to my FPO.

I'm Joomy, and today I'm gonna talk about a verified foreign function interface between Coq and C, and my contributions to this project, especially about how we used existing metaprogramming techniques, and devised new ones.
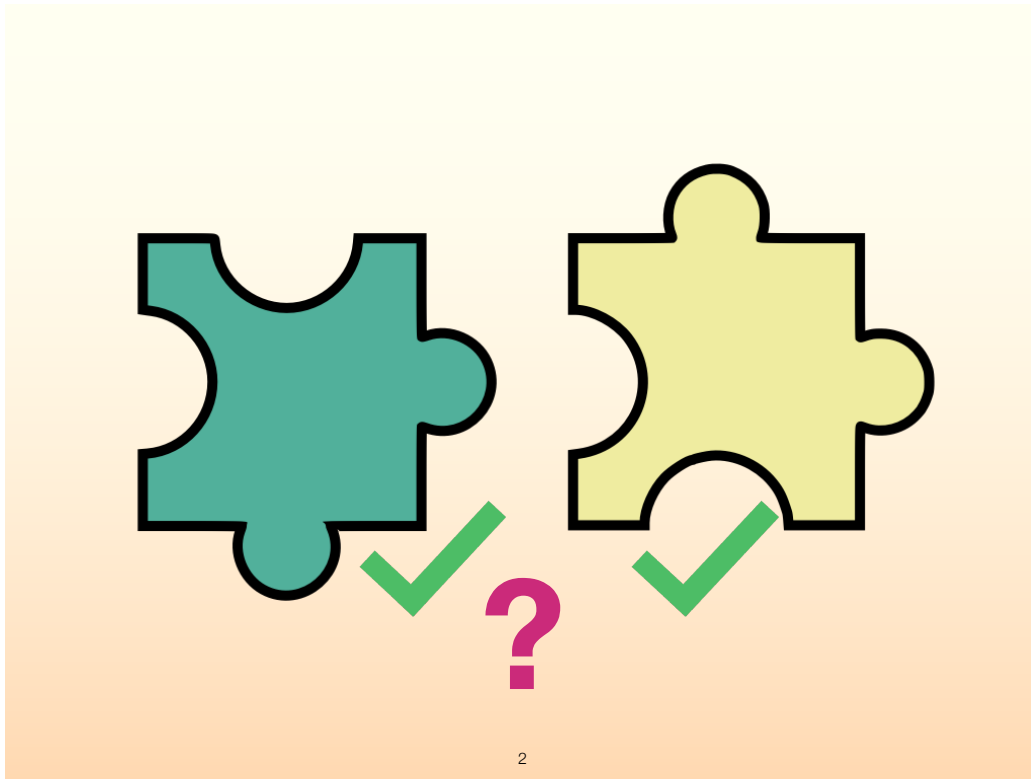
I want to start with a summary of the problem we are trying to solve with the verified FFI, and what our solution is.

In the real world, almost all programs are written in multiple languages. <click> and then linked together.

<click> Parts written in different languages can be verified separately,

<click> but how do we prove that when these parts are combined into one multilanguage program, that it still works correctly?
Many have studied this problem, and obviously there are many nuances here, but recently the common approach has looked something like this…
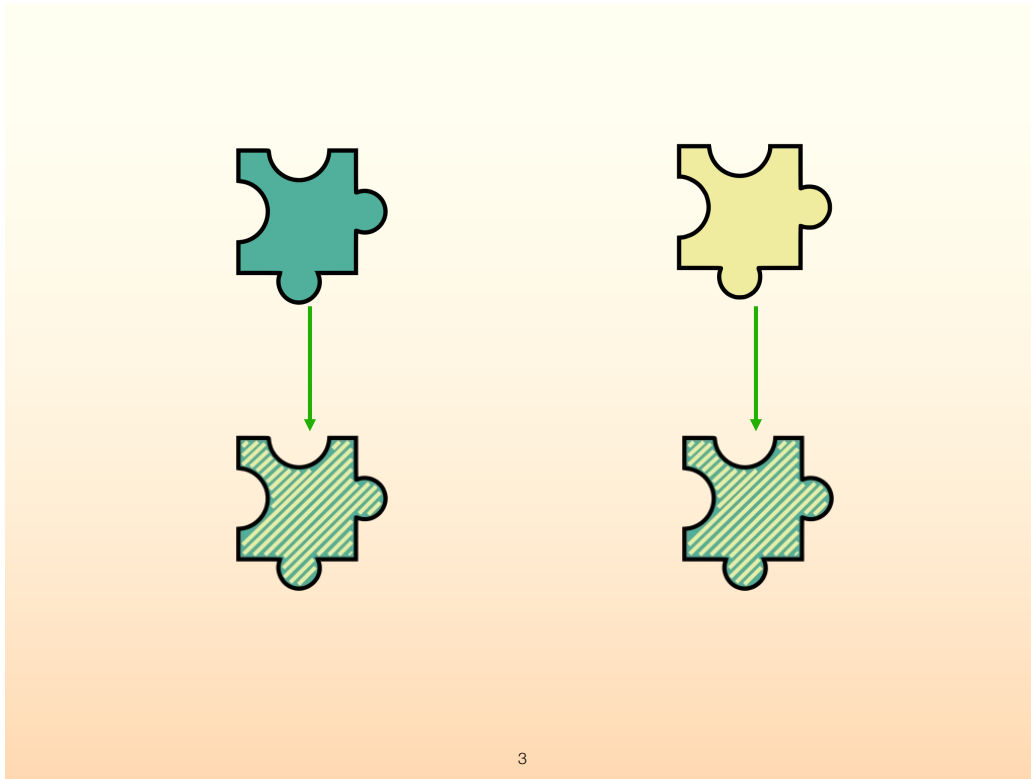
Here we have code in two different languages, we want to link these two.
<click> We define a combination of the two languages,
<click> and treat these programs as a program in the combined language.

Here, the combined language allows terms from one language to be embedded in the other language. This

is an idea from Matthews and Findler. We dared to think that we can avoid this formula because of a particular coincidence.
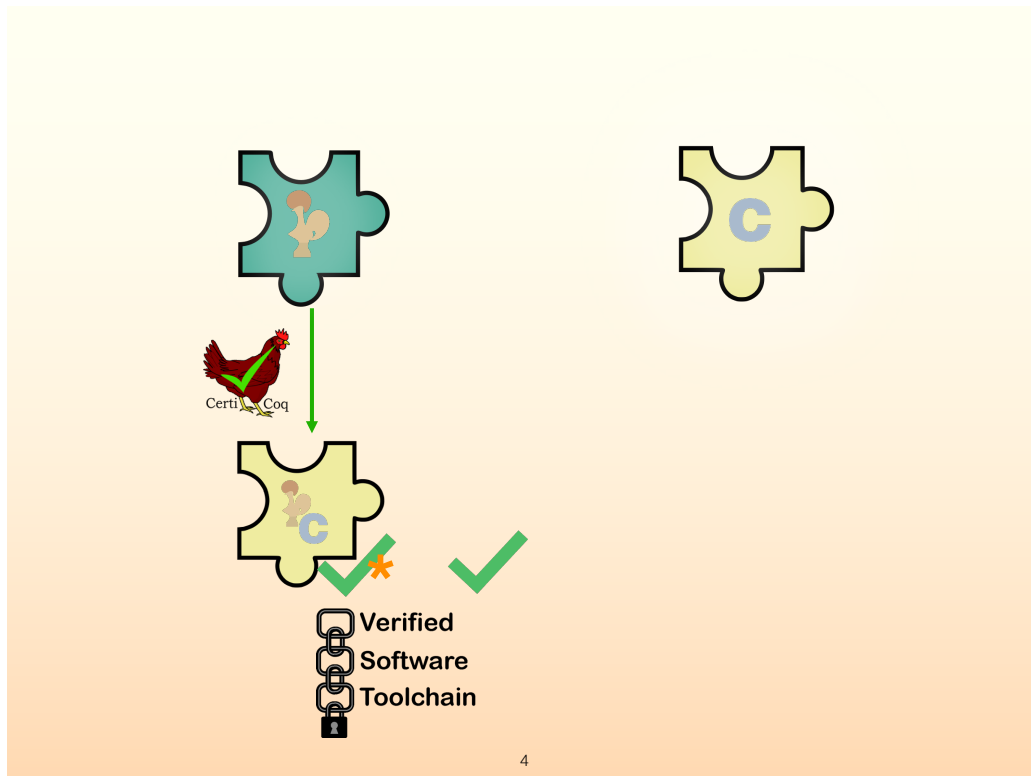
That coincidence goes like this: We have some Coq code and some C code that we want to link together.
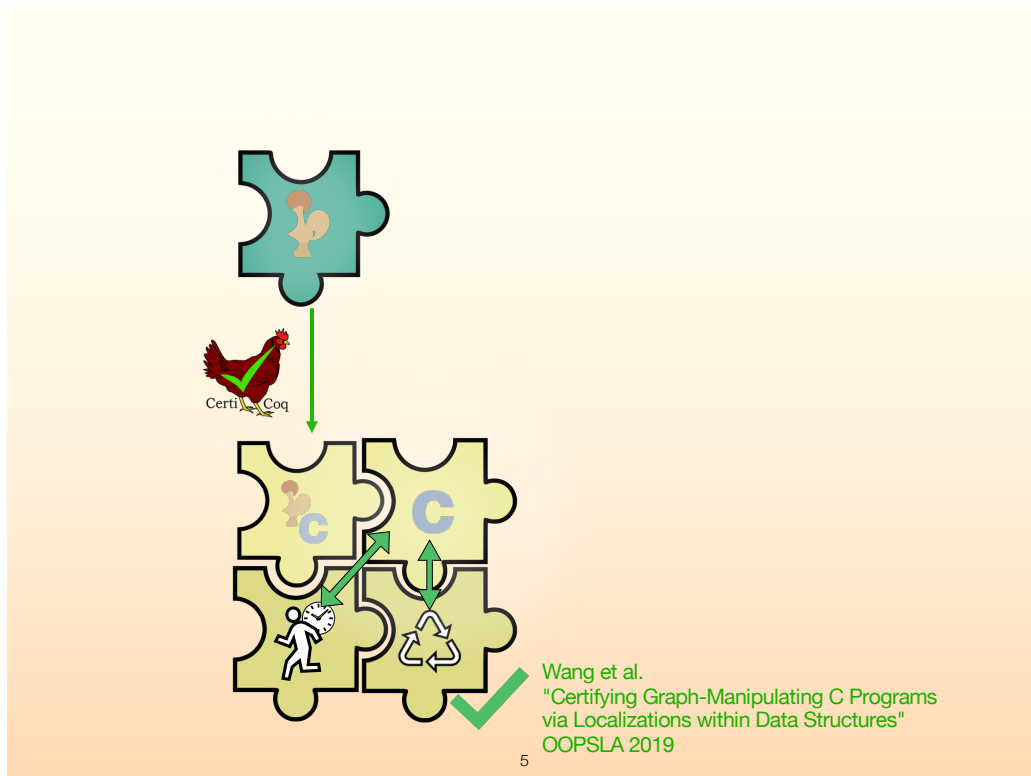
<click> But we also have a verified compiler from Coq to C, so we can compile our Coq code to C code. This is the CertiCoq project that has been in the works for 10 years or so. <click> Now that we have the C

version of our Coq program, we can link that with our C program and reason about the combined program, using the Verified Software Toolchain (VST), which includes a program logic for C, based on separation logic.

Here's the crucial observation about the verified FFI project: **Our language of reasoning and the source language of our compiler are the same; we use Coq for both. Our language of foreign functions and the target language of our compiler are the same; we use C for both. This setup helps us avoid the traditional approach to multi-language semantics, where you have to combine two languages.**

<click> Though I must note that the end-to-end compiler correctness proof of CertiCoq is under construction. We had an incomplete proof for closed programs but due to some recent changes that proof is now out of date. We discuss in our tech report with Kathrin Stark and Andrew Appel, how our work can help us state the theorem for open programs, and how that theorem can connect to VST.

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

In reality, a complete program in our system looks more like this, where you have a Coq program compiled to C, foreign functions in C, and also a runtime and a garbage collector.
<click> In fact, we have a verified garbage collector implementation, for a garbage collector that operates on the CertiCoq runtime.
<click> In the VeriFFI project, we

provide the necessary mechanisms for foreign functions to be verified with respect to the CertiCoq runtime and garbage collector.

Instead of enumerating my contributions, I want to take you through an example of what a user of our system experiences. I will point out my contributions as we go along, and we can delve deeper into the necessary details later.
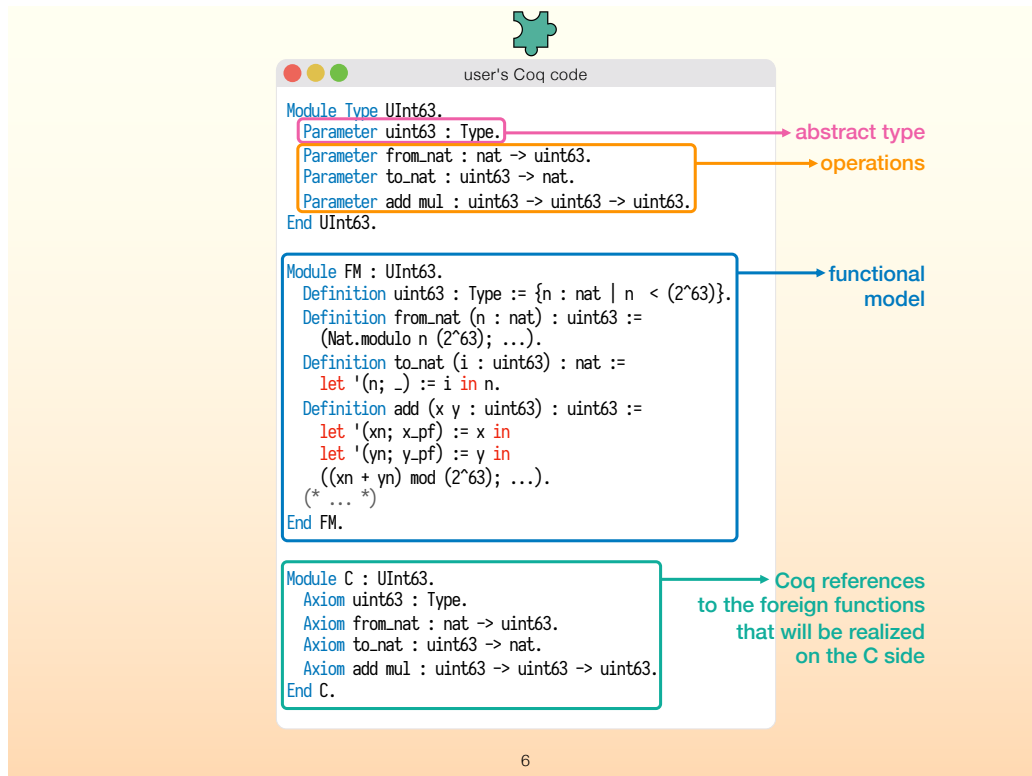
Integers are the most common data type, so suppose we want to write a program that uses integers. In Coq, we already have the inductive representation of integers, so yes we can use them, but they are quite wasteful with space, we have to do a lot of allocations to create such values, and a lot of pointer dereferences to traverse them. We

really want to have faster integers, the single machine word integers we know and love. With our system, implementing and using primitive single machine word integers is possible.

Let's start by defining an interface for unsigned 63-bit integers, as a module type in Coq, which is like a module signature in OCaml / Standard ML. <click> We have an abstract type, and some operations on it.

Now we need to provide implementations of this module type.

<click> One possible implementation is a purely functional one. In order to stay as close as possible to machine integers, we

can define integers as bounded natural numbers with modulo wrapping. This is going to have terrible performance, but that is okay!

<click> What we really want to use is the primitive one. Here we declare the operations on integers as axioms, in order to tell Coq that they don't have a plain Coq implementation. These functions will be realized when we compile to C and link with the foreign functions written in C.

Looking at this slide, you can probably sense here that we want to prove that these two module implementations are equivalent, that they behave the same way! This is stuff we teach to undergrads in COS 321!
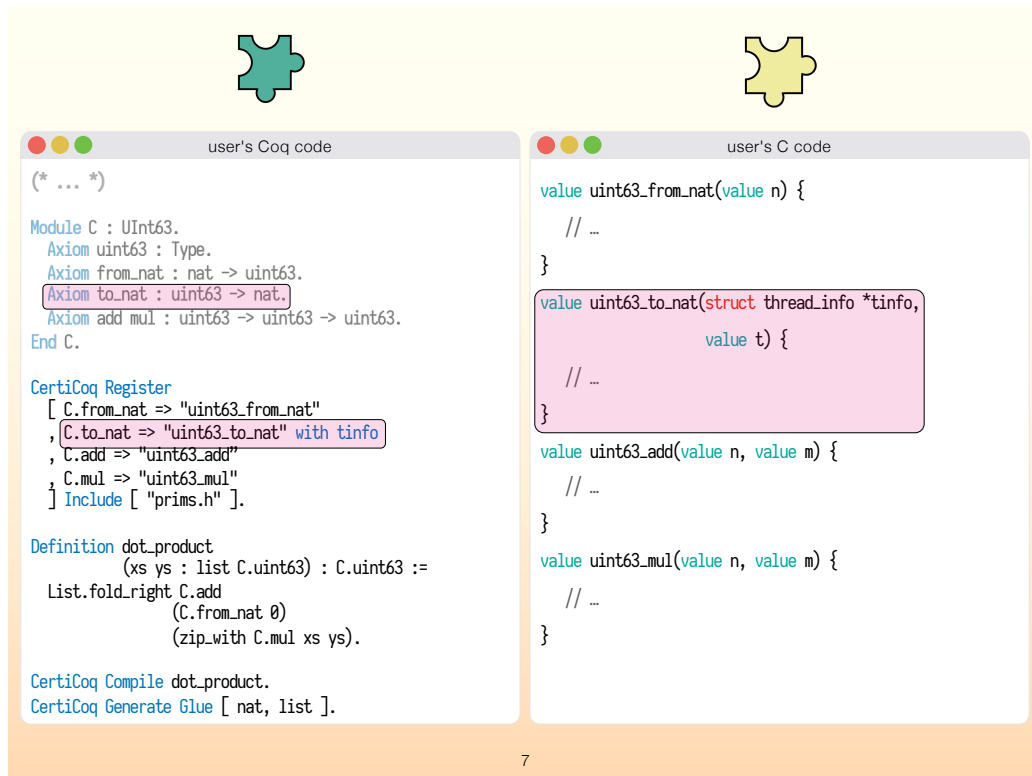
But first, let's finish the operational side. Now all we need to do is to register these references with Coq, and actually provide the C implementation for these functions. I am not showing the implementation here, that's not scientifically novel, but it's in the thesis.

Once we do that, … <click> we are

free to write our own functions that use primitive integers. Like this dot product function on lists of primitive integers. We can then compile this function to C using CertiCoq, and call it from any C program.

Now, let's talk about the correctness of these functions. What we want to do is to state as the specification of each of these functions, the functional model function, and the C implementation counterpart, do the same thing. VST is a great tool for such proofs!
<click> Let's zoom in on the to_nat function, which converts a primitive integer to a Coq natural number, and let's talk about its VST specification.

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.

```
● ● ●                    user's Coq proof
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.
```

8

If we were to write by hand, here's what that specification would look like. There's a lot here, and you don't have to follow the details. Very very roughly, what we are saying here is this: <click> Given some runtime info, and an input to the functional model, <click> if the C function takes a value that is represented by that functional model input,

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

There are a lot of details about how the memory heap is represented by a graph, and how garbage collection can change this heap graph but the new graph is isomorphic modulo changes in this function, etc. We don't have time for this right now but our tech report explains these details more clearly.

This is just the specification.
<click> We then claim that the C function body follows this specification and write the proof by hand.
The cool part is, if we have a complete proof of this, that means our foreign

function is
1) type-safe
2) correct with respect to the functional model.
(Though we do not have a proof of type-safety since it requires reasoning across meta-levels)

I know this slide is overwhelming. Thankfully only certain parts of it vary from function to function.
<click> and as long as we can account for those variations, we can actually generate this spec automatically.
And making that possible is one of my contributions in my thesis (joint work with Stark and Appel).
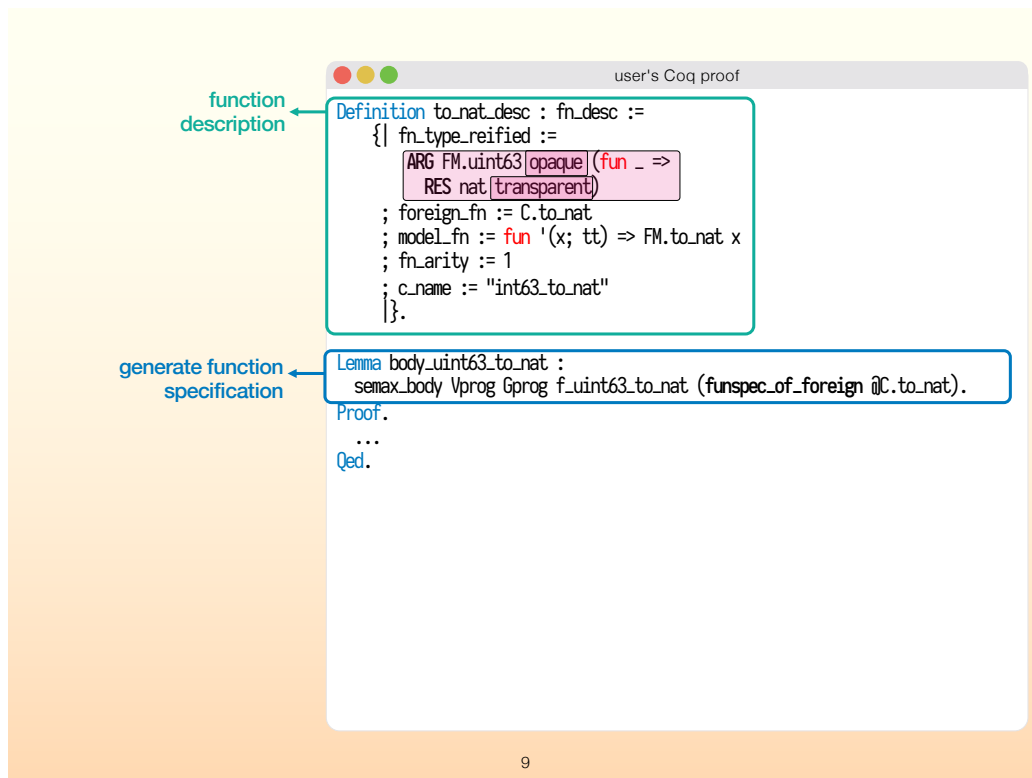
Here's what generating this spec looks like.

<click> We have a function description, which includes everything we need to know about this function. Most importantly, it has
<click> a reified description of the function type. Thanks to this description, we can ensure that the

foreign function and the model function in this record actually abide by the same type.

Another important use of the reified description is the
<click> annotation of each component of the type with a type class instance. In a function description, we use these instances to hold information about how these types are represented in memory. Here you can see that we say the input to to_nat is opaque (i.e. a primitive type) and its output is transparent (a plain Coq type).

Once we finish the function description, we can then
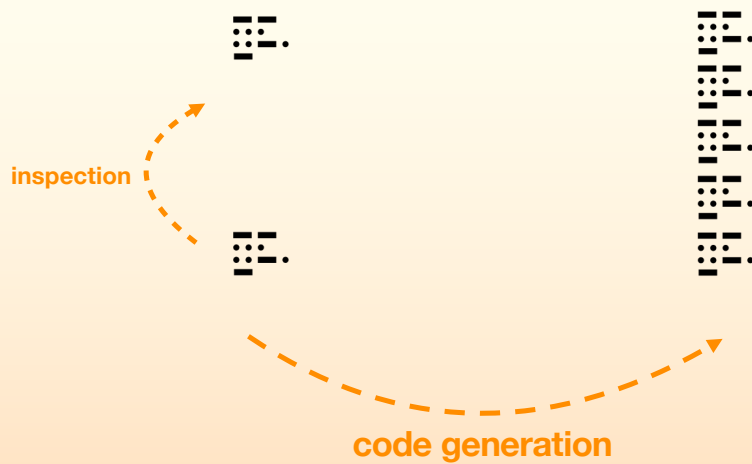<click> compute a VST specification from

it and start writing our proof.
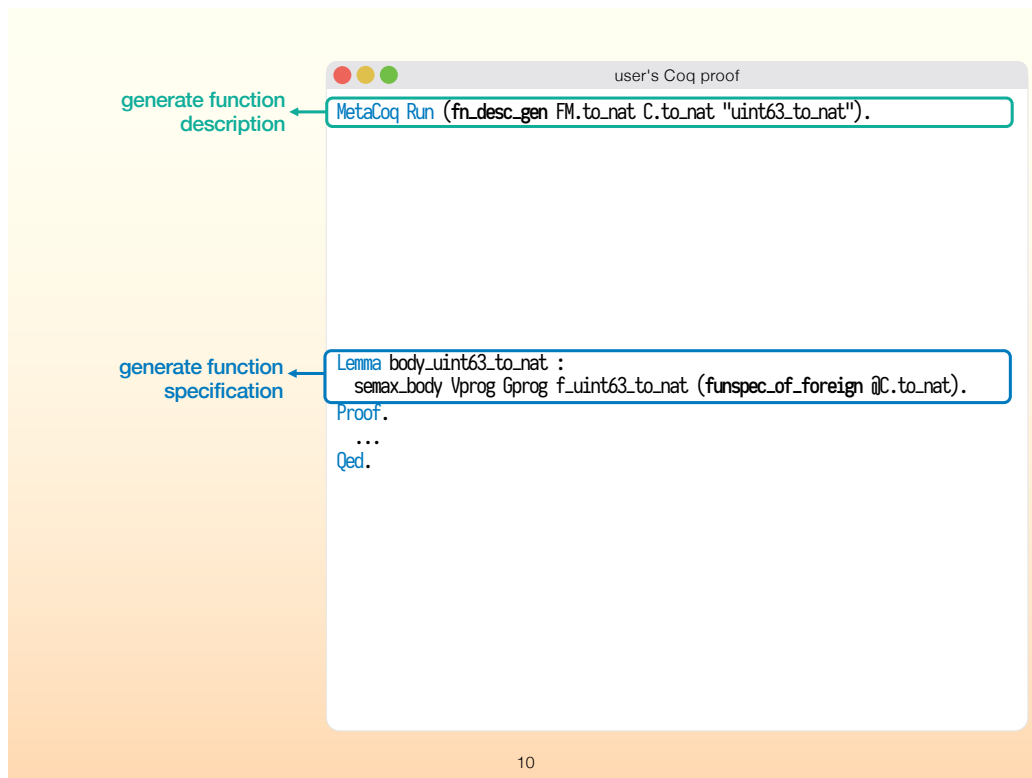
Not to sound like an infomercial, but there's more! We can also generate the function description automatically using the generator we wrote with MetaCoq.

Well, how is that possible? Through a lot of metaprogramming efforts.

What is metaprogramming?

inspection

code generation

11

Well, what do I even mean by metaprogramming?

Metaprogramming is simply programs generating or inspecting other programs.

It comes in all shapes and colors: C macros are metaprogramming. `eval` in JavaScript is metaprogramming. Replacing text in your code before every compilation is

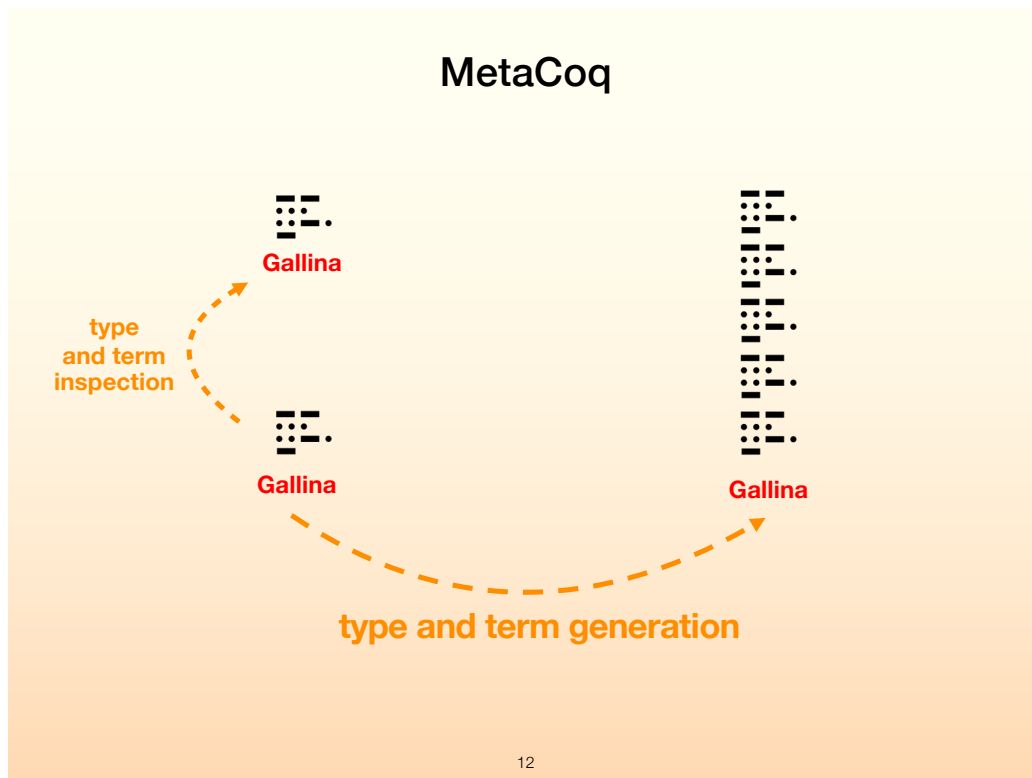metaprogramming. Template Haskell is metaprogramming.

There are a few different axes we can categorize metaprogramming:
- compile-time vs run-time, based on when the generation or introspection happens. C macros or Template Haskell are compile-time, JavaScript's `eval` is run-time.
- homogeneous vs heterogenous, based on which language is generating or inspecting which. If a language does it to itself, it's homogeneous. Template Haskell, once again, homogeneous. JavaScript's `eval`, homogeneous. C macros, I'd argue are heterogeneous, the macro directives appear inside C code but they have their own language

with its own if statements and
definitions and namespaces.
- text-based vs term-based, how
 bindings are handled, etc.

Okay, so which kind do we care about
here? Well, in the VeriFFI project, we use
MetaCoq and Ltac for metaprogramming.

MetaCoq is a project formalizing Coq in Coq, but it also comes with a "Template Haskell"-like system. This system lets you express metaprograms in what is called a "template monad". At compile time, you can run these programs, and these programs can inspect existing terms and types, they can generate new types and terms. They can add

new definitions, new type class instances etc.

Both the language of metaprograms, and the language our metaprograms manipulate are Gallina. Therefore, this is a homogeneous compile-time metaprogramming system.
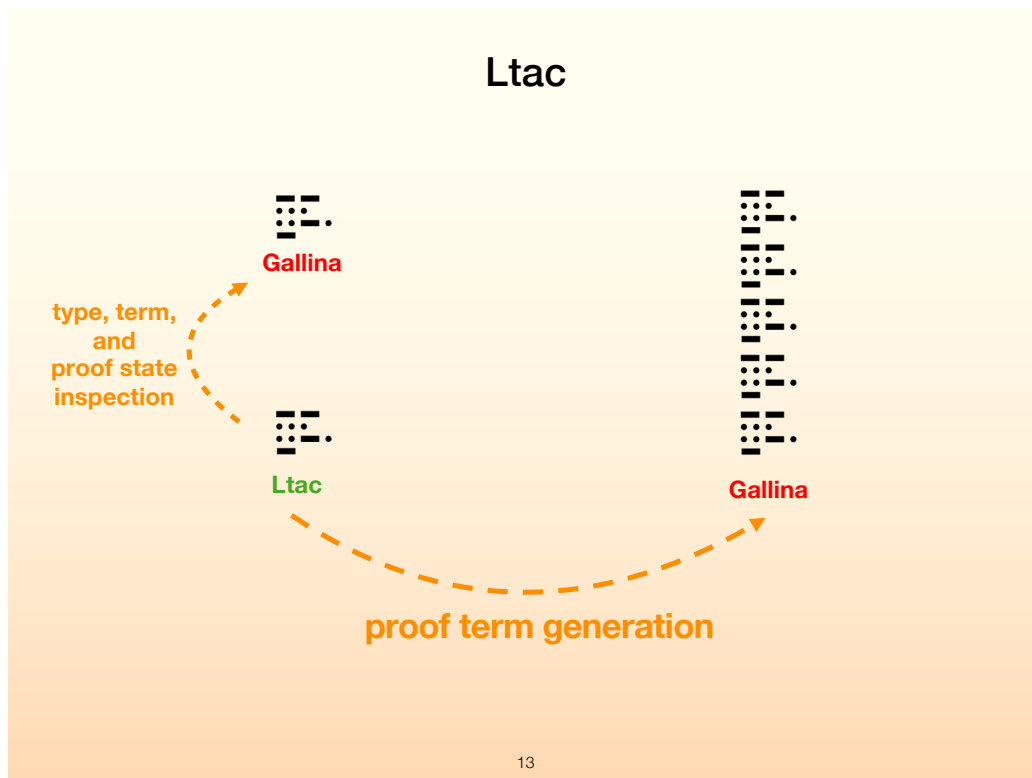
The other tool we use for metaprogramming is Ltac. Ltac is Coq's tactic system. It is not commonly thought as a metaprogramming system but all it does it letting the user inspect types, terms, and proof states. When a definition is finished, Ltac generates a proof term, so it does proof term generation as well.

Ltac has its own language for tactics, but it generates a different language, Gallina, so we can categorize it as a heterogeneous compile-time metaprogramming system.

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**

14
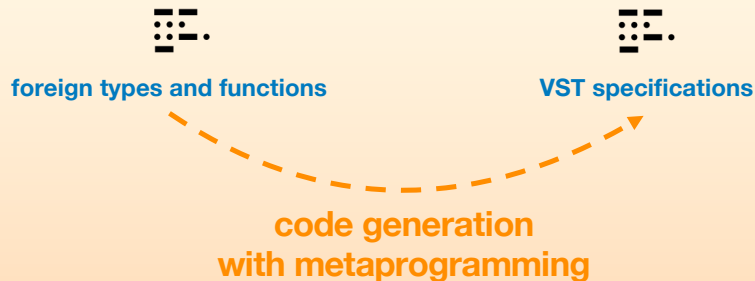
So, why am I talking about this? Well, it's because I had to make a choice about how we generate things.
I could choose to take foreign types and functions (the module implementations I showed you before), and generate a scary VST specification directly from that, using MetaCoq or Ltac. That's still a doable thing (at least with MetaCoq), but that

would require writing a colossal metaprogram. We can call that approach "monolithic generation".

There are two problems with this approach, supposing we use MetaCoq: <click>

1. MetaCoq's representation of Coq terms is "low level" by design. They kept the core language to a minimum, which made it easier to write proofs about, which is MetaCoq's primary goal, but this made it harder to generate code in the core language. It's unavoidable to run into certain issues. The graph predicate generation part of our development is more monolithic, so I did run into these issues and had to come up with solutions to get around them:

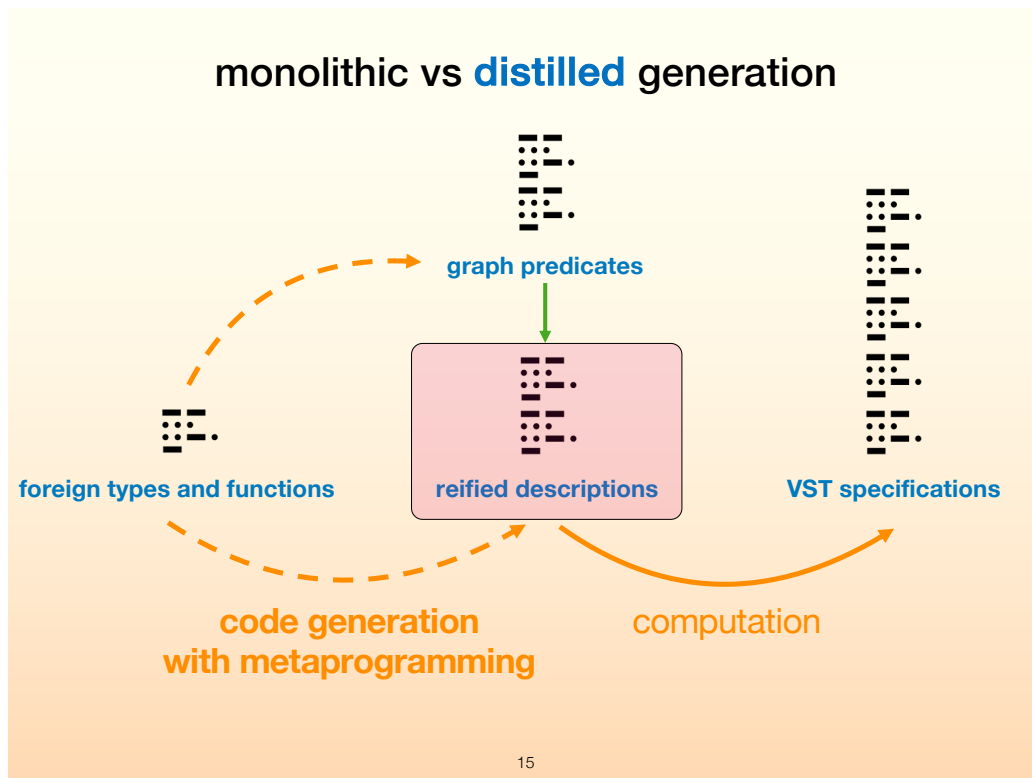• MetaCoq terms use De Bruijn indices for

bound variables.

- MetaCoq doesn't allow mutually recursive type class instances so you can only define one at a time.
- Recursive calls have to refer to a specific fix expression, which means you cannot absentmindedly do a recursive call on constructor arguments and let type class resolution handle where the call goes.
- Type class inference has to resolve immediately, not when all the definitions are finished or anything like that. You cannot hope that everything will be fine at the end and avoid rigor.
- There is no easy inference based on a context. You have to create a context yourself and then run the inference primitive in MetaCoq, essentially doing lambda lifting.

I describe these solutions in my thesis.

The other important reason to avoid monolithic generation is this:

2. Metaprograms are harder to reason about! It is harder for us to tell if our metaprogram generates the right thing, and that it always works. Reasoning about metaprograms requires reasoning from a meta-level above, which is like *chopping veggies with oven mitts on.*

monolithic vs **distilled** generation

graph predicates

foreign types and functions   reified descriptions   VST specifications

**code generation
with metaprogramming**   computation

15

Here's what I suggest instead.
<click> We come up with an
intermediate representation. In my
case, this is the reified description
type.
<click> We can generate these
descriptions using metaprogramming.
<click> But for the rest, we do not
need compile-time metaprogramming.
We can write a Gallina function that

takes this description and computes a VST specification.

This way we isolate the metaprogram to the first half of generation. We generate the VST spec, by first distilling into a multi-purpose description, and from there we can do a lot of things as we wish.

<click> In our verified FFI project, we use MetaCoq to generate the graph predicates and reified descriptions. These are separate generations. The lemmas about our graph predicates are generated via Ltac. These predicates and lemmas are then used in the reified descriptions.

<click> Since I believe this is one of the scientific contributions of my thesis, I

want to focus a bit more on the reified descriptions.
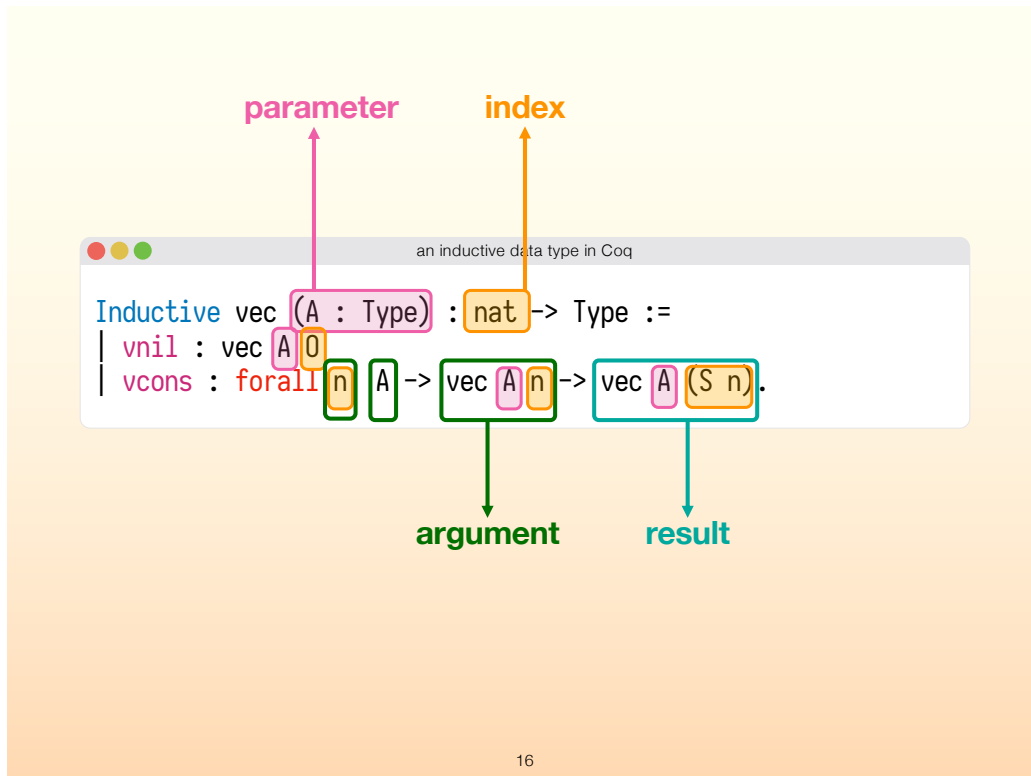
Before that, we should have a quick refresher of what an inductive data type can look like in Coq. This is dependent types land, so there might be some people unfamiliar in the audience.

(For those who have seen the ICFP bingo, the vector type has appeared, you can mark it on your card!)

A parameter is a part of the type that doesn't change among the subterms. The type A here is a parameter. If your vector is a vector of booleans, all the subvectors are gonna be vectors of booleans.

An index is a part of the type that CAN change among the subterms. Nat here is an index, indicating the length of the vector. If your vector is of length 3, the immediate subterm will have length 2 and so on.

An argument is a part of a constructor that holds some value. The cons constructor has 3 arguments, for example.

A result is the what the constructor returns at the end.

Now that we agree on a terminology to

talk about inductive types, let's talk about how to describe constructor and function types.

MetaCoq description of **vec**

If we use MetaCoq's representation for constructor types directly to generate things, we are going to have to deal with this monster.

\<click\> These are the parts that describe the nil and cons constructors for indexed vectors.

While this is a nice, extensive deeply embedded description, it is too far from real Coq values. We want a distilled version of this. And that is the reified description mechanism we developed.

```
                              VeriFFI's generation library

Inductive reified (ann : Type -> Type) : Type :=   higher-order abstract syntax-ish
| 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))))))).   annotations

(* 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))).
```

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.

18

As a metaprogramming term, reifying means representing a language construct into an explicit object in a language. Here are we trying to define an inductive type in Coq, that describes different parts of a Coq constructor type or a Coq function type.

Essentially, we have a different constructor for each component. Type parameters, arguments, and the return type.

<click> Notice that all except the return type take a function as an argument.

This might seem familiar to you from the work on higher-order abstract syntax, and it is indeed inspired from that.

The difference here is that we have a way of telling apart what part of a type we are looking at, and we can add extra information about the types we are dealing with. The annotation argument (written here as ann) carries

the extra information.
Here we are mixing deep and shallow embeddings: We use the deep embedding part to distinguish the components of a type, but we are using the shallow part to annotate these components with a type class instance.

We already have seen an example reified description for the to_nat function, but here's another example:

<click> We want to describe the type of the cons constructor for indexed vectors. Notice that the type class instances are playing nicely with

dependent types.

<click> We can also describe the type of a function on indexed vectors.

These descriptions can be automatically generated from the function types, using our generators based on MetaCoq.

There are other approaches that try to combine deep and shallow embeddings, namely by McBride, and also by Prinz et al. Their work is more general than ours, but our specific setup of languages allows us to avoid some of their more complicated mechanisms involving the universe

pattern. We can simply <click> annotate the components with type class instances thanks to this coincidence, since the language we are describing is a part of Coq, a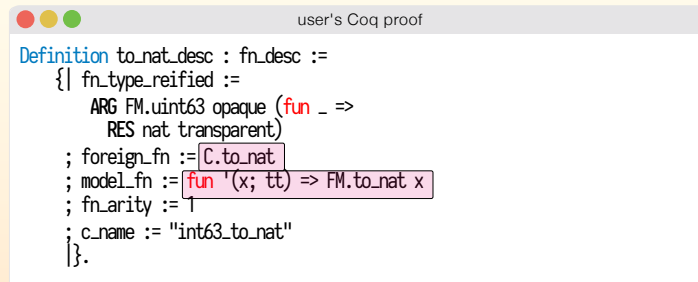nd we are annotating with Coq type classes. **Here's the gist of the reified descriptions in one sentence: By making the describer and describee the same language, and using higher-order abstract syntax, we can handle dependent types and annotate each component in a concise and type-safe way.**

The annotation we have in this example here talks about how values

of a Coq type are represented in the heap graph when the program is compiled to C, but we can annotate descriptions with anything we want.

What do reified descriptions buy us?

1. type safety


```
user's Coq proof

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"
    |}.
```
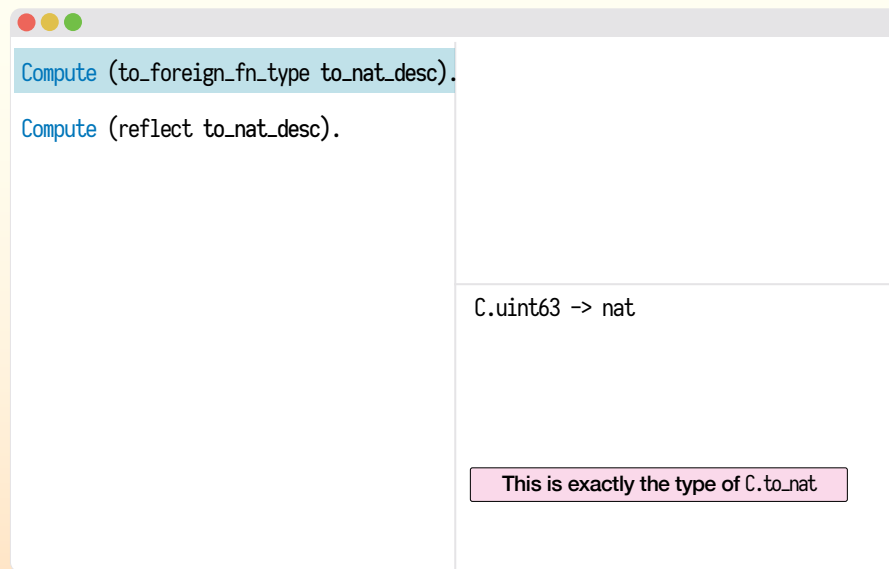
19

Well, why did we go through all this trouble? What do reified descriptions buy us? The most important aspect is type safety. Here we see the description of the to_nat function, once again. We have a reified description inside, and thanks to that…

<click> when we provide the Coq references to the C implementation,

and the functional model of this particular foreign function, we know they have the right type. How do we know that?...
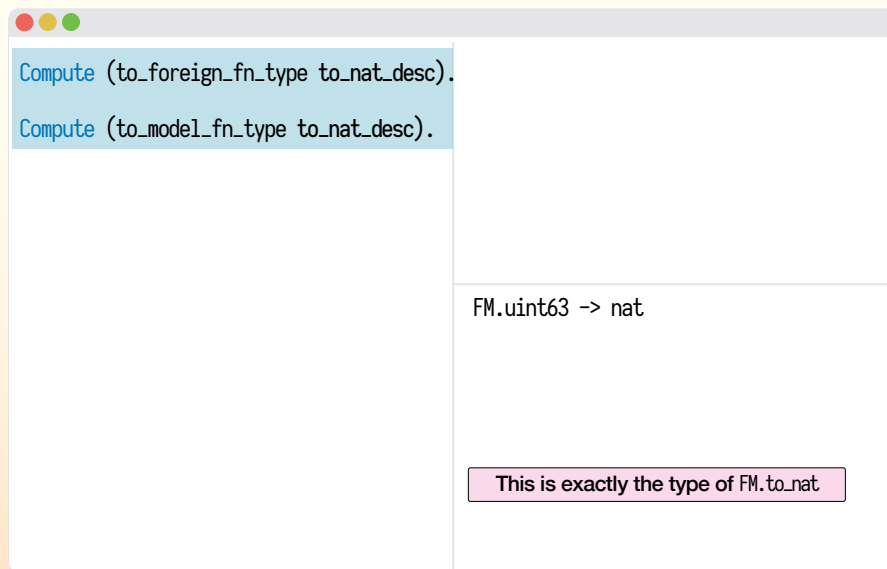
```
Compute (to_foreign_fn_type to_nat_desc).

Compute (reflect to_nat_desc).
```

C.uint63 -> nat

This is exactly the type of C.to_nat

Here's how. We can recompute the type of the foreign function from that description. Here we see that converting back to the foreign function type gives us exactly the type of C.to_nat. It takes the C version of our abstract type, and gives back a Coq nat.
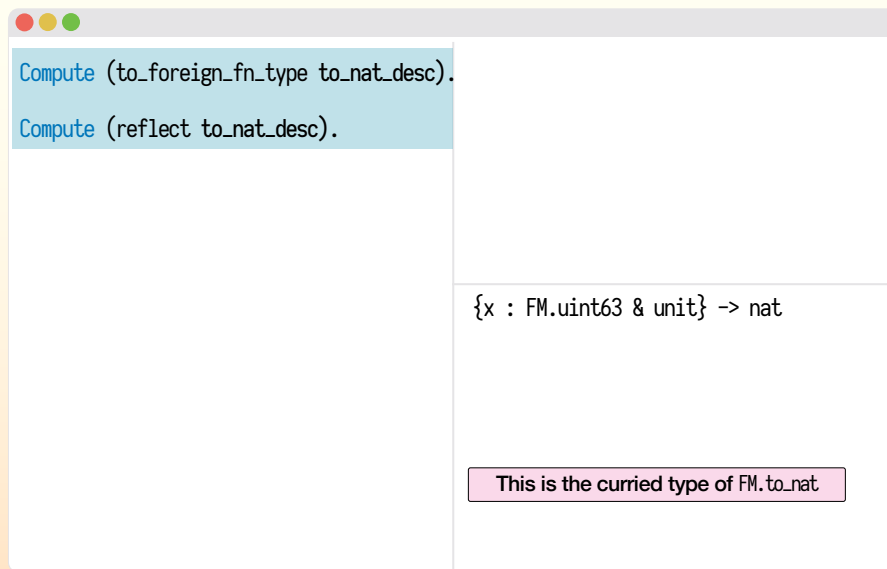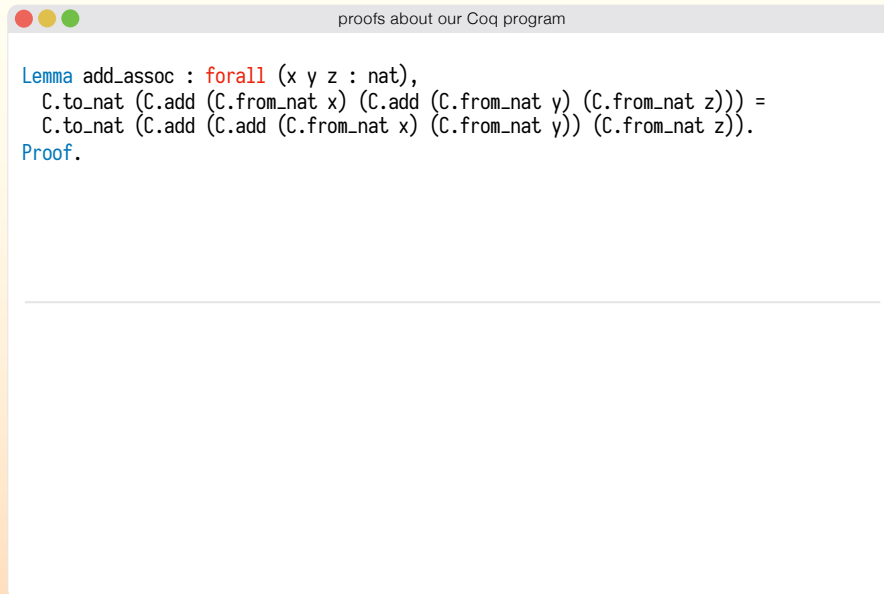
We can do the same thing for the functional model. Here we get the type of the uncurried version of the FM.to_nat function. This is just more convenient for VST specs, but we could have easily computed the curried type…

```
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`

…like this. This function takes the functional model version of our abstract type, and returns a normal Coq nat, as it should.
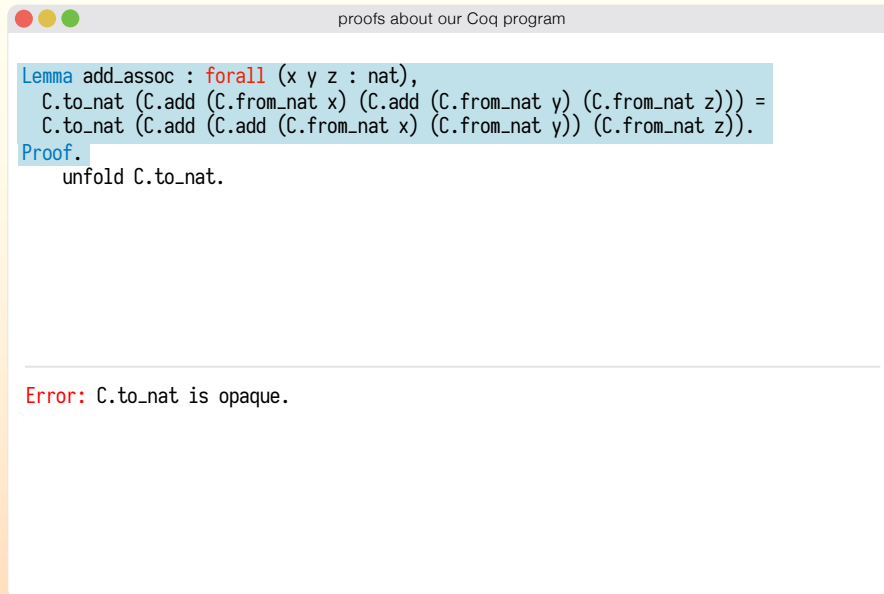
2. rewrites of primitives to models

```
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.
```

Here's another thing this representation buys us:

We know dependent type checking involves evaluation! Our foreign functions, on the other hand, do not evaluate. So if we try to prove a lemma such as this, where we want to prove the associativity of addition, for the primitive addition operation, we have a problem: We cannot unfold the

definitions of to_nat, from_nat, and add
and continue our proof…

2. rewrites of primitives to models

Since they are axioms on the Coq side, they get stuck! These foreign functions evaluate when we compile the program to C, because then they get realized by C functions, but that's not good enough for proofs about the Coq references of foreign functions.

So, if we wanted to write a lemma like this, how can we do that if our foreign

functions do not evaluate at compile time?

2. rewrites of primitives to models

proofs about our Coq program

```
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))
```
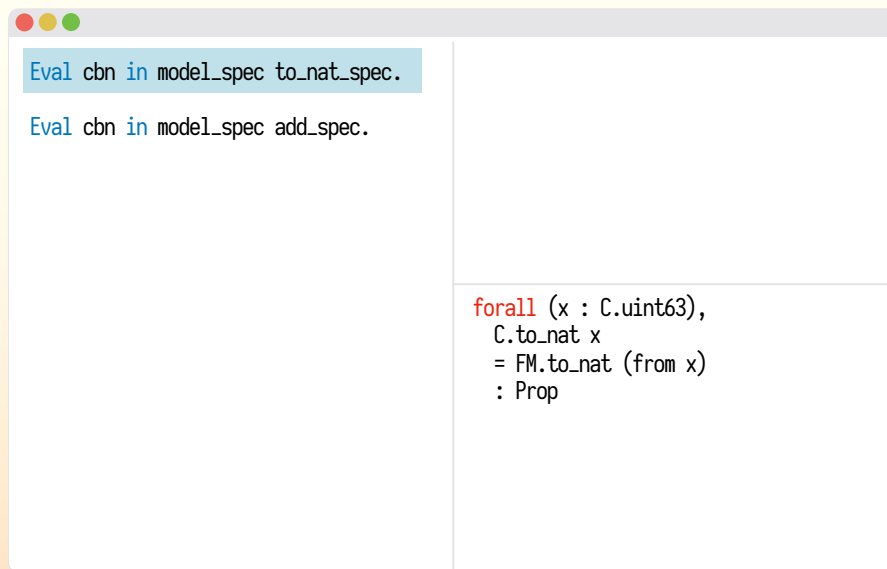
25

Our solution to this is a rewrite mechanism. We derive a way to rewrite calls to the foreign functions into calls to the functional model. If you have proofs for the VST specifications we generated earlier, using these rewrite principles becomes fair game.

Here we use our rewrite tactic. Notice

how our goal is now entirely about the functional model, and from there it's straightforward to prove this goal.

Under the hood, these tactics depend on rewriting principles we generate from the reified descriptions, which look like this…
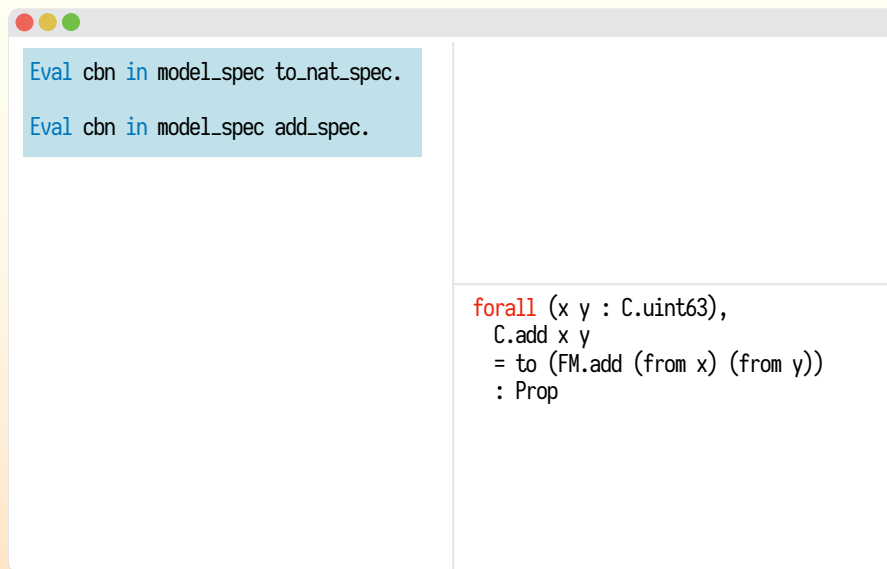
```
Eval cbn in model_spec to_nat_spec.

Eval cbn in model_spec add_spec.
```

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

Here we say, assuming there's an isomorphism between the C version and the functional model version of the abstract type then you can say,

if you have a primitive integer, converting it to a nat with the C.to_nat function is the same as converting it to a nat with FM.to_nat from the functional model, where the input to

the functional model one is isomorphic to the starting primitive integer.

```
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
```
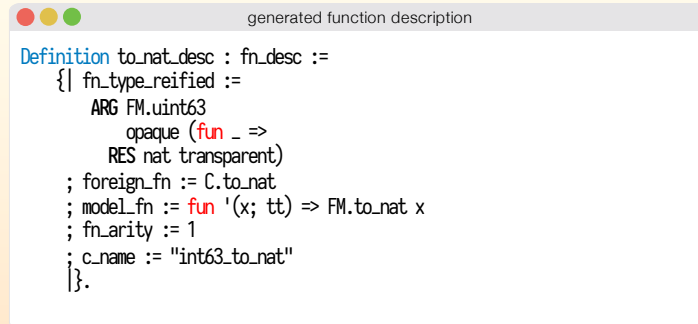
The rewriting principle we'll have for addition says,
if you have two primitive integers,
adding them via the C implementation,
will give the same result as adding
them in the functional model, where
the inputs to the foreign and functional
model versions are isomorphic.

Well, where does this isomorphism

come from?

An isomorphism
between **the foreign type** and **the model type**
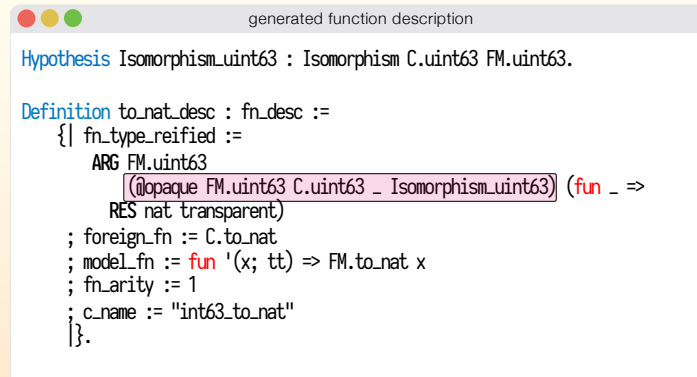
```
●●●                    generated function description

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"
    |}.
```

28

Let's take a closer look at this function description. When we said that the input is opaque, we actually store an isomorphism in this description…

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

```
generated function description

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"
    |}.
```

29

That isomorphism looks like this. We assume on the proof level that we can convert back and forth between the foreign type and model type. Based on this assumption, we can write proofs about the Coq references to foreign functions, like the associativity of addition that we just saw.

This is an assumption that is only

available at the proof level, but it is a safe assumption to make if we have VST proofs for all the foreign functions in the module. Those VST proofs amount to equivalence of the C and FM modules, therefore we can say that the types are isomorphic. Once again, this is stuff from COS 321!

We do not, however, have a Coq proof of this isomorphism yet. It's not clear to me how such a proof would go. I suspect it might require a proof about MetaCoq's reification and reflection system, or Coq modules and their power of abstraction, among other things. I think this should be future work.

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)

Now that we all have a better sense of the VeriFFI project, I want to compare VeriFFI with existing work on verified compilers and verified FFIs.

<click> Oeuf is a verified compiler for a subset of Coq with no user-defined types, dependent types, fixpoints, or pattern matching. It doesn't feature an FFI, but it allows verifying the wrapper C program to be verified via VST. Oeuf

allows plugging in a garbage collector if you want to, but it's unverified.

<click> Cogent is a restricted functional language with a *certifying* (translation validation) compiler. The language has no general recursion or nested higher-order functions, but it features a uniqueness type system that makes garbage collection unnecessary. It allows users to check if their C foreign functions satisfy this type system and provides safety that way.

<click> CakeML is a verified compiler for ML. It allows C foreign functions and accounts for the correctness of the foreign functions in the compiler's correctness theorem, but it doesn't have a program logic in which the user can prove foreign functions correct. It has an oracle about the behavior of foreign functions that the correctness theorems depend on. And CakeML does have a verified garbage collector.

<click> Melocoton is a verified FFI project that allows programs written in a toy subset of OCaml and a toy subset of C to interact. Users can prove the correctness

and safety of their programs using Iris's separation logic. While Melocoton uses the multi-language semantics based on a combined language, it tries to isolate users from that and enable language-local reasoning for code in OCaml or C. It uses a model of a garbage collector to reason about multilanguage programs.

<click> In comparison, our work, VeriFFI, is built upon on verified compiler, CertiCoq. It allows reasoning about both correctness and safety of programs written in Gallina and CompCert C. One can use VST's separation logic to reason about C foreign functions, and it features a real, verified garbage collector.

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

31

Before I finish my talk, I want to summarize what I think my scientific contributions are in this thesis.

<click> Reified descriptions can describe and annotate function types in a concise and type-safe way, thanks to higher-order abstract syntax and making the describer and describee the same language.

<click> Given a reified description, we can calculate (instead of generate) separation logic specifications about foreign functions that talk about their correctness and safety.

<click> We can assume an isomorphism between the foreign type and the model type if there's a module equivalence.

There's a lot more that I didn't have time for today
<click> You can read my dissertation to learn more about glue code, reified descriptions, function and constructor descriptions, rewrite principles, and generation of all these. I also feature more examples in the thesis, such as primitive bytestrings, programs with input/output, and mutable arrays.

We just got the news that our POPL submission about the VeriFFI project was

conditionally accepted, so hopefully you can also see our paper there.
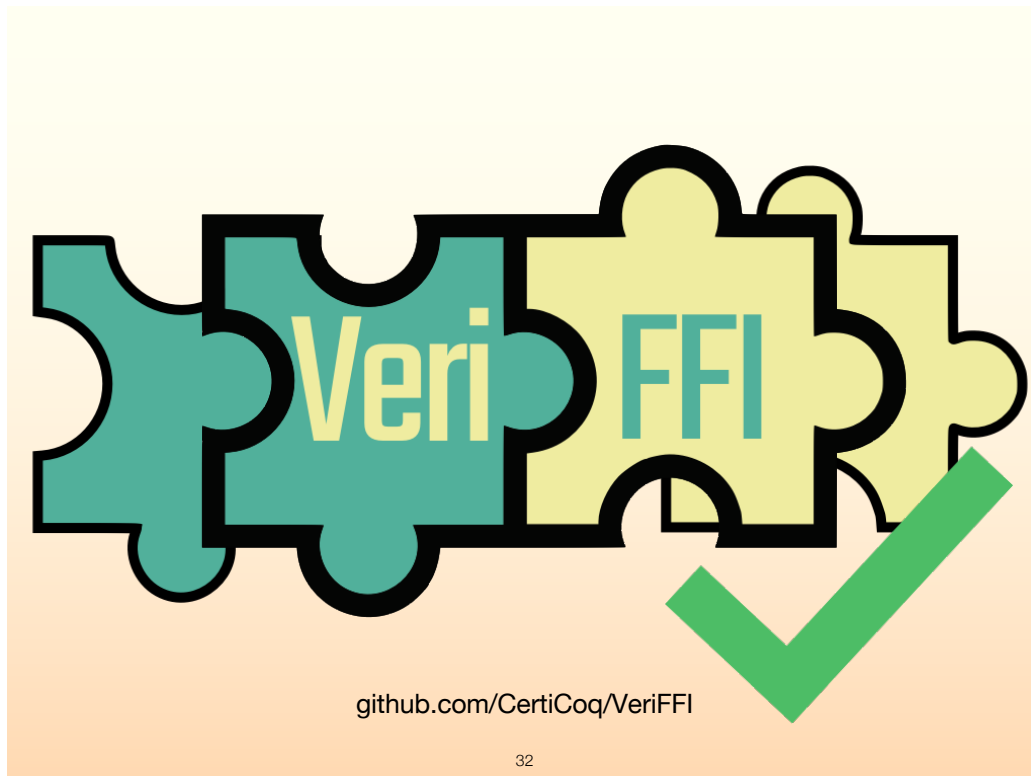
Thank you for listening. This was VeriFFI,<click> a Verified Foreign Function Interface between Coq and C. Coq program components are proved correct directly in Coq, C program components are locally proved correct using the Verified Software Toolchain (VST), and the connection is made via VST function specifications that are generated by

VeriFFI.

Thank you.