

A Verified Foreign Function Interface between Coq and C

Joomy Korkut, Princeton University & Bloomberg*

Kathrin Stark, Heriot-Watt University

Andrew W. Appel, Princeton University

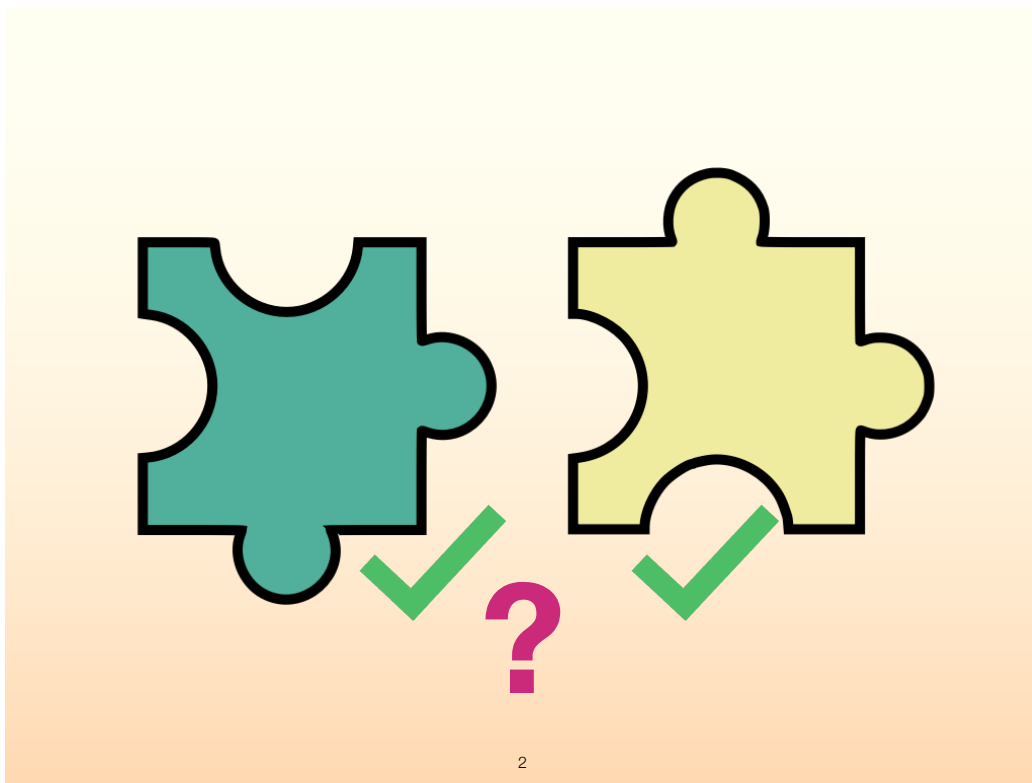
POPL 2025
January 22, 2025

Hi everyone!

I'm Joomy, a researcher at
Bloomberg.

Today I'm gonna talk about a verified
foreign function interface between
Coq and C.

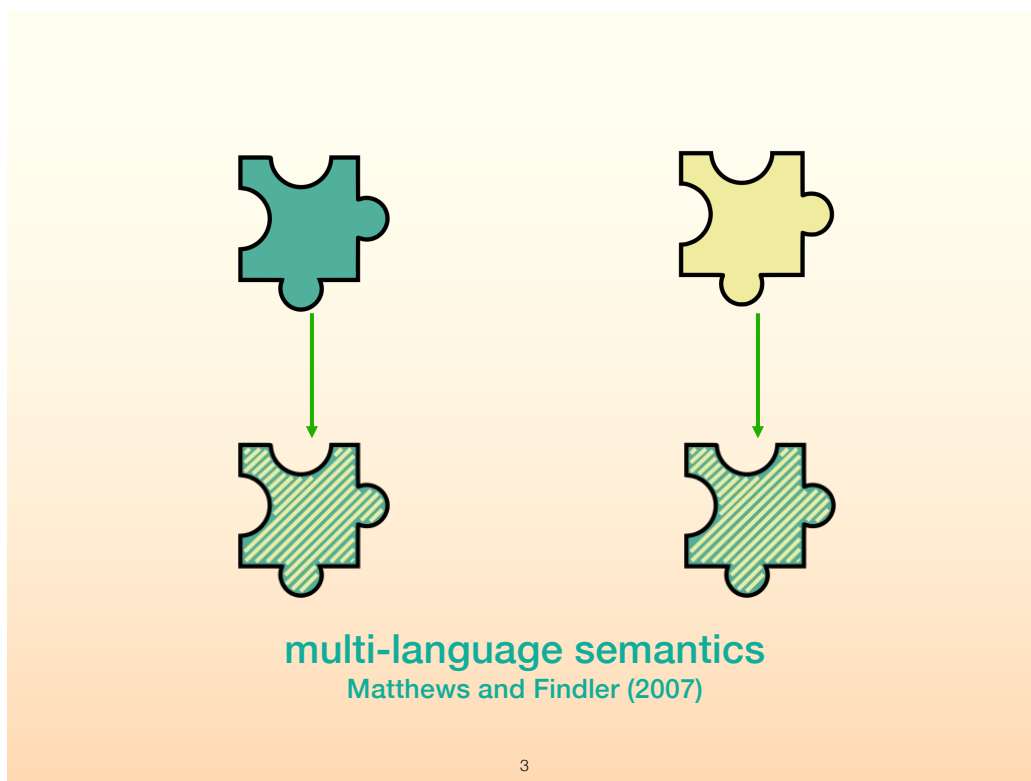
This is a paper based on my PhD
work. It is joint work with Kathrin Stark
and Andrew Appel.



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, recently
the common approach has looked
vaguely something like this...



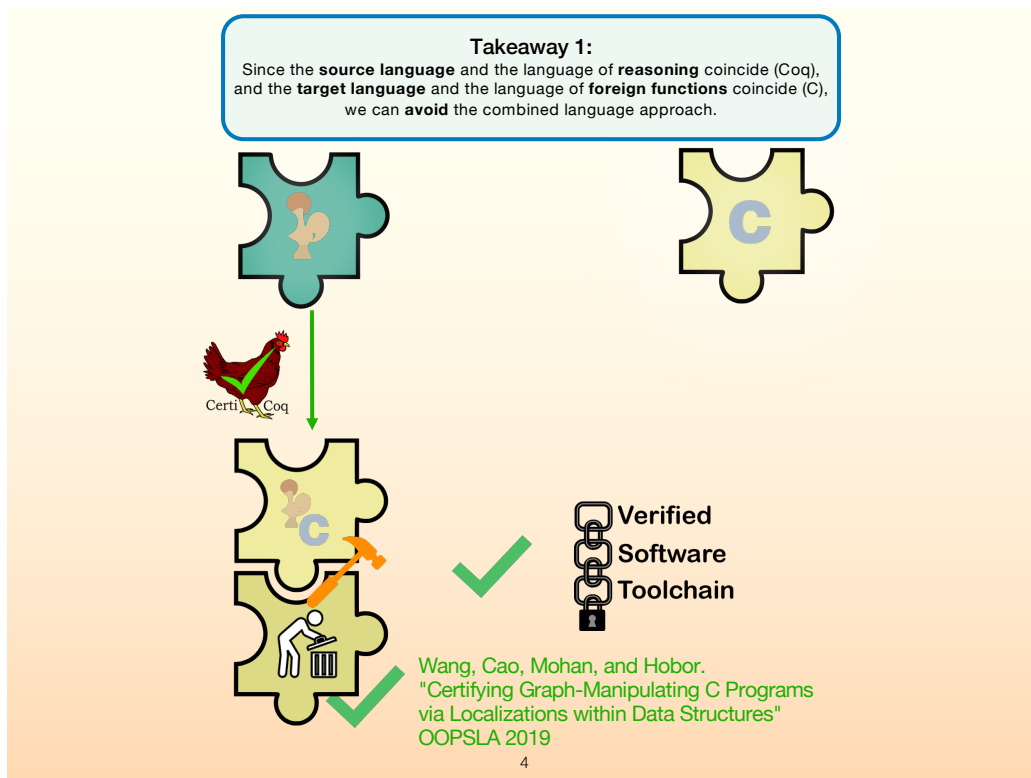
We have code in two different languages we want to link.

<click> We define a combination of these two languages, <click> and treat these programs as programs in the combined language.

This is an idea from Matthews and Findler, and it's often referred to as the multi-language semantics

approach. This is brilliant and necessary for the general case, but it is also a clunky way to reason about multi-language programs. It requires extra indirection, duplicated proof efforts, etc.

But what if we didn't have to cover the general case? Can we avoid this formula then? We think we can because the languages we choose have a particular overlap.



That overlap 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. 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.

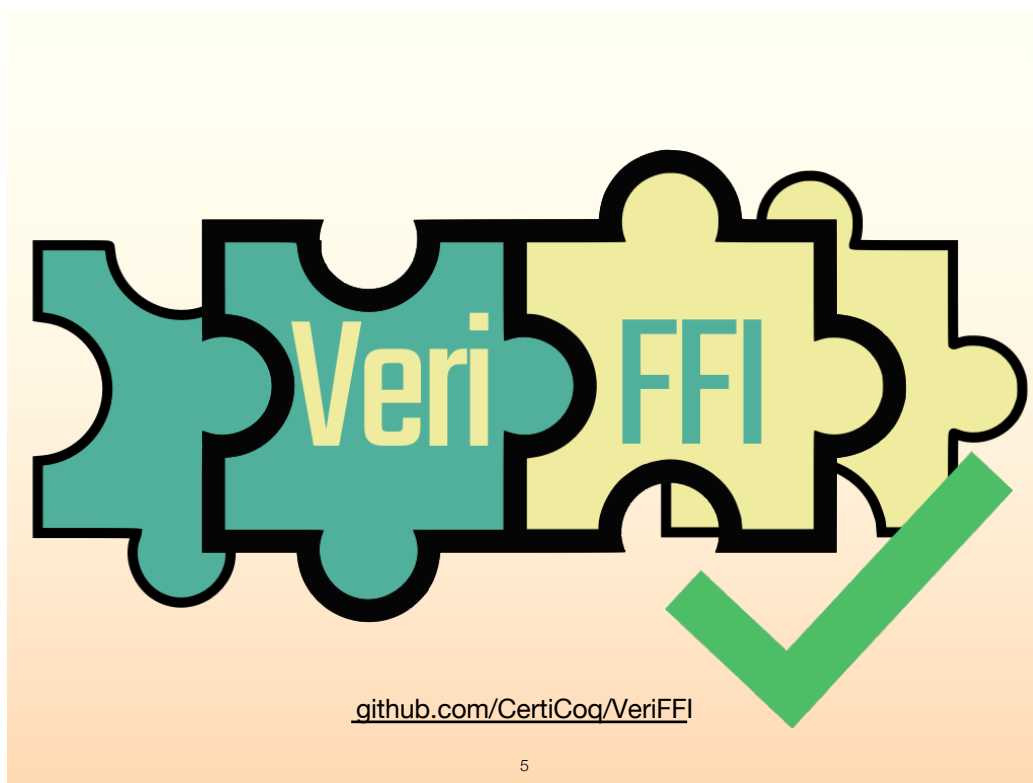
So I want this to be takeaway one:

<click> Since the source language of our compiler and our language of reasoning coincide (they are both Coq), and the target language of our compiler and our language of foreign functions coincide (they are both C), we can avoid the traditional approach to multi-language semantics.

Just to get a better sense of the big picture,

<click> we also have a verified garbage

collector implementation, thanks to Wang et al. We build our work upon their graph library.



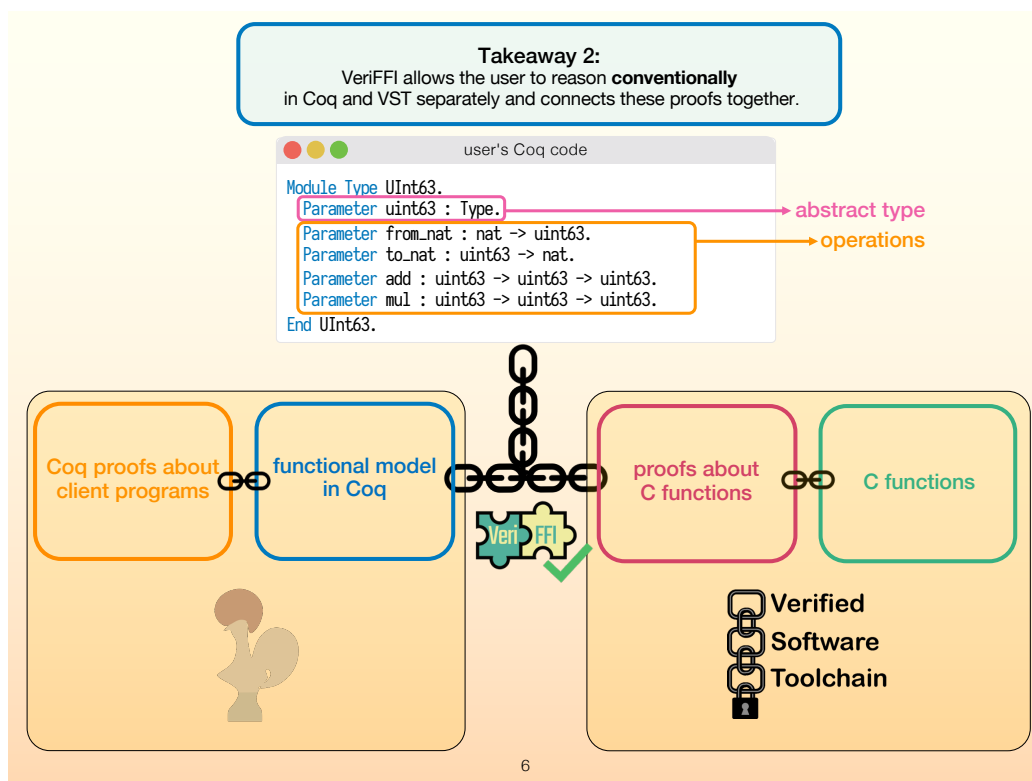
And this setup constitutes our project, VeriFFI, which is a Verified Foreign Function Interface between Coq and C, where you can call C functions from Coq.

Coq program components are proved correct directly in Coq.

C program components are locally proved correct using the Verified

Software Toolchain (VST).

The connection is made via VST function specifications that are generated by our system.



Let's dig a little deeper into our setup.

Suppose we want to write a program that uses machine integers. Until recently, that wasn't possible, you'd have to use an inductive type for integers in your program. There are hacks you can do in extraction but those can disregard the hard-earned guarantees you got, through the

proofs you finished through blood, sweat and tears. We don't want that! Here is the setup we propose.

We define an API for unsigned 63-bit integers in the standard way: as a module type in Coq, which is like a module signature in ML.

<click> We have an abstract type, and some operations defined on that abstract type.

<click> We want to write proofs about this interface, so we give a purely functional definition of this interface, and we can reason about client programs of this interface using that functional model.

<click> and all that reasoning is plain, conventional Coq!

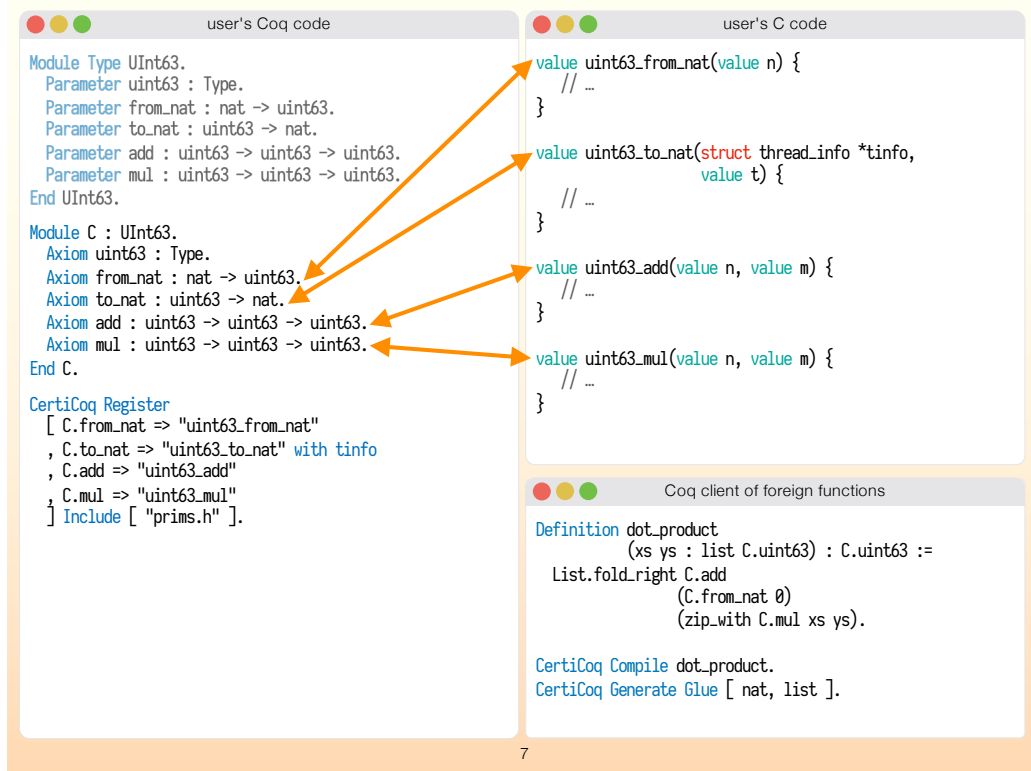
<click> On the other hand, we write foreign functions that satisfy this interface, and we prove that these functions play nicely with the Coq program compiled to C via CertiCoq, and that they play nicely with the functional model.

<click> and all that reasoning is a conventional VST proof!

<click> All the required mechanisms in the middle are generated by our system, VeriFFI.

This is takeaway two:

VeriFFI allows you to do conventional reasoning in Coq and VST separately, and connects these proofs together.



Let's start with the operational side.
<click> We declare an axiom for the type itself and the operations on it, 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. This is standard practice in Coq when you want to define foreign functions as well.

<click> We register these references with Coq, and actually provide the C implementations of these functions.

<click> Now we are free to write our own functions that use integers. Like this dot product function on lists of integers. We can then compile this function to C using CertiCoq. This is enough for the operational part of the foreign function interface! We're now conceptually at the state of the art for 1995!

Now, let's talk about the correctness of these functions. How do we know that the C functions we wrote are 1) safe, and 2) functionally correct? Our idea is to write and prove VST specifications about these C functions that express that. Let us start with the functional correctness.

```
user's Coq code

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

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

CertiCoq Register [ (* ... *) ] Include [ "prims.h" ].

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.
```

functional model

To reason about the functional correctness of the C function, we must write a purely functional model in plain Coq, of what these C functions actually do. This can be a module that implements the module type we had before.

In this module we define the integer type to be the inductive natural

number type with a bound. We define its operations to respect modulo wrapping, just like machine integers. If we were to actually run this, it would have terrible performance, but that is okay, since this is only for the proofs, and it is an easier interface to write proofs for!

```
user's Coq code
Module Type UInt63.
  Parameter uint63 : Type.
  Parameter from_nat : nat -> uint63.
  Parameter to_nat : uint63 -> nat.
  Parameter add : uint63 -> uint63 -> uint63.
  Parameter mul : uint63 -> uint63 -> uint63.
End UInt63.

Module C : UInt63.
  Axiom uint63 : Type.
  Axiom from_nat : nat -> uint63.
  Axiom to_nat : uint63 -> nat.
  Axiom add : uint63 -> uint63 -> uint63.
  Axiom 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" ].

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) {
  // ...
}

Coq client of foreign functions
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 ].
```

Let's attempt such a proof, we can take `to_nat` here as an example, which is a function that converts a machine integer to a Coq natural number.

We want to state as a specification that the C implementation of `to_nat` does the same thing as our functional model definition of `to_nat`. Thankfully, VST is a great tool for such proofs!

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,
       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 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 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.

```

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

if the C function takes a value that corresponds to the functional model input, then the C function returns a value that corresponds to the functional model output.

We claim that the function body satisfies this spec.

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 corresponds to the functional model input,

<click> then the C function returns a value that corresponds to the functional model output.

There are a lot of details about how heap graphs and their isomorphisms. See our paper for the explanation.

<click> So far this is just the specification, so 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

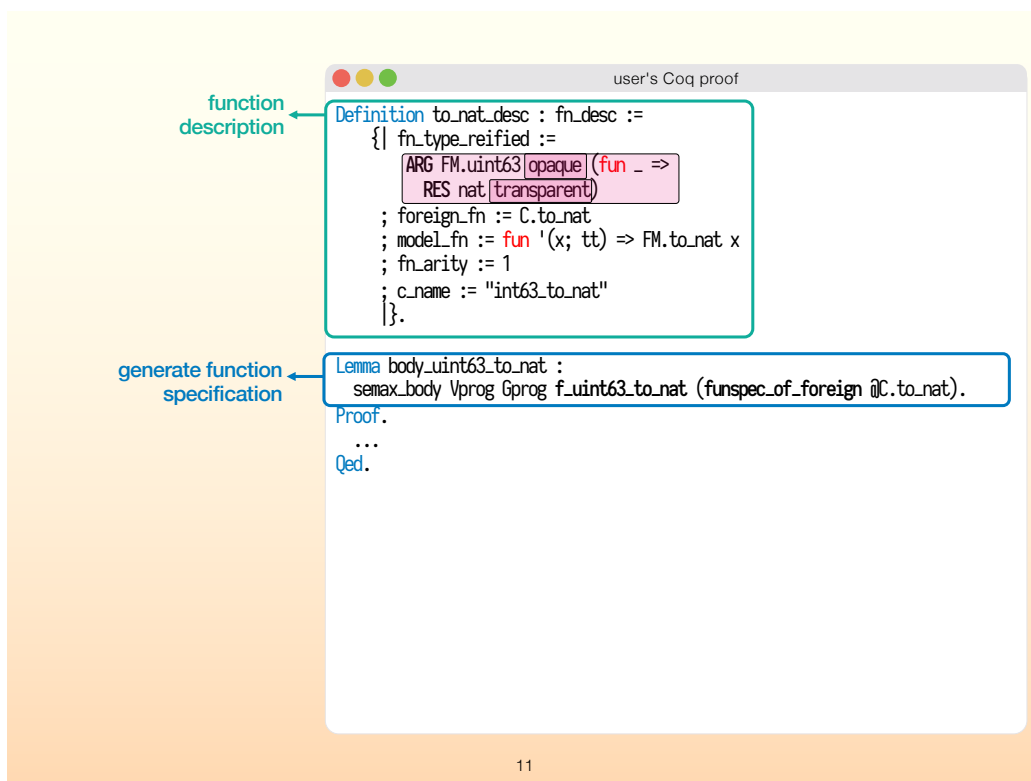
- 1) type-safe
- 2) correct with respect to the functional model.

(Though we do not have a proof of type-

safety since it requires reasoning across meta-levels)

I know this spec is overwhelming.

<click> Thankfully only certain parts of it vary from function to function. Maybe we can find a way to account for these variations. One idea is to keep them in a record.



Here's what that looks like.

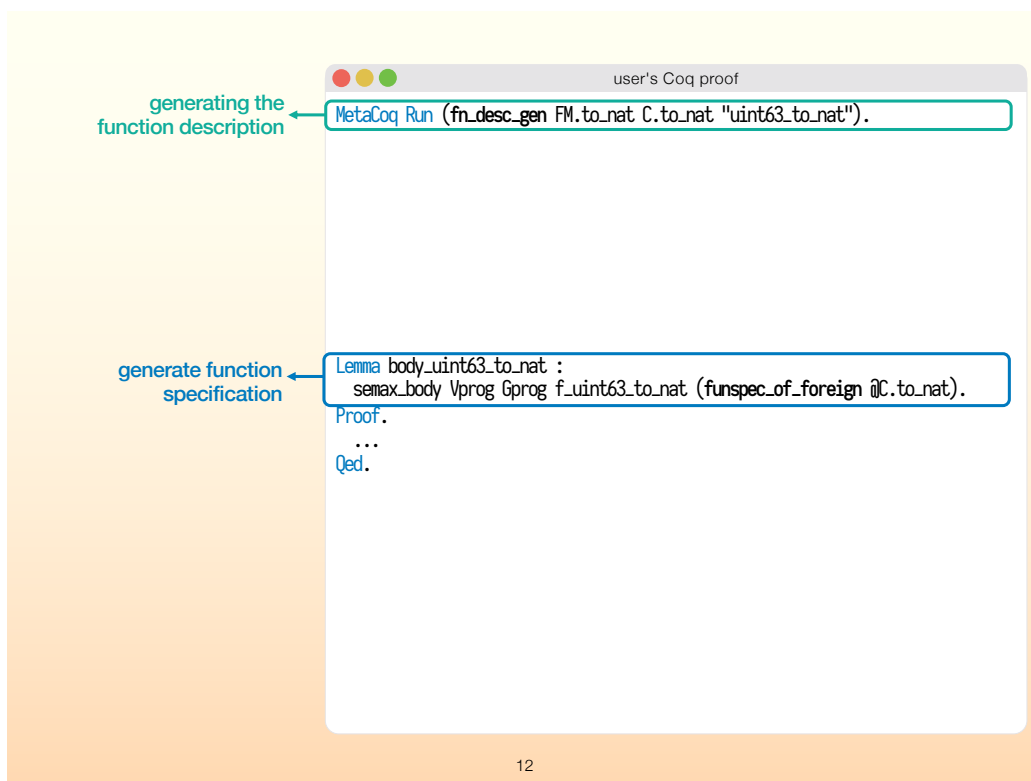
We have a function description, which includes everything we have to know about this function. Most importantly, it has

<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 type.

In the reified description, we have `<click>` annotations of each component of the type with a type class instance. In a function description, we use these instances to hold information about type-specific memory representations.

`<click>` Once we finish the function description, we can then **compute** a VST specification from it and start writing our proof.



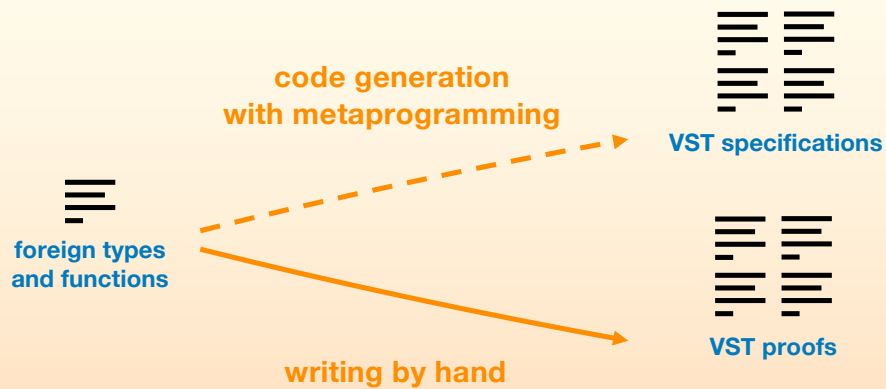
Not to sound like an infomercial, but there's even more! We can also generate the function description automatically using the generator we wrote with MetaCoq.

I want to talk a bit more about reified descriptions, since I believe it's one of our scientific contributions.

monolithic vs. distilled generation

Problems

1. MetaCoq is "low level" by design.
2. Metaprograms are **harder** to reason about!
3. Requires a much deeper understanding of the system.



13

This contribution comes from our realization that we have a choice about how we generate things. We could choose to take foreign types and functions and generate a scary VST specification directly from that, with a colossal metaprogram. Or for parts where we write VST proofs by hand, we would write a long proof that's hard to follow.

We can call that approach “monolithic generation”.

But that’s not ideal: <click>

1. We use MetaCoq for code generation, which includes a Coq plugin for compile-time metaprogramming, like Template

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

Their main goal is to reason about

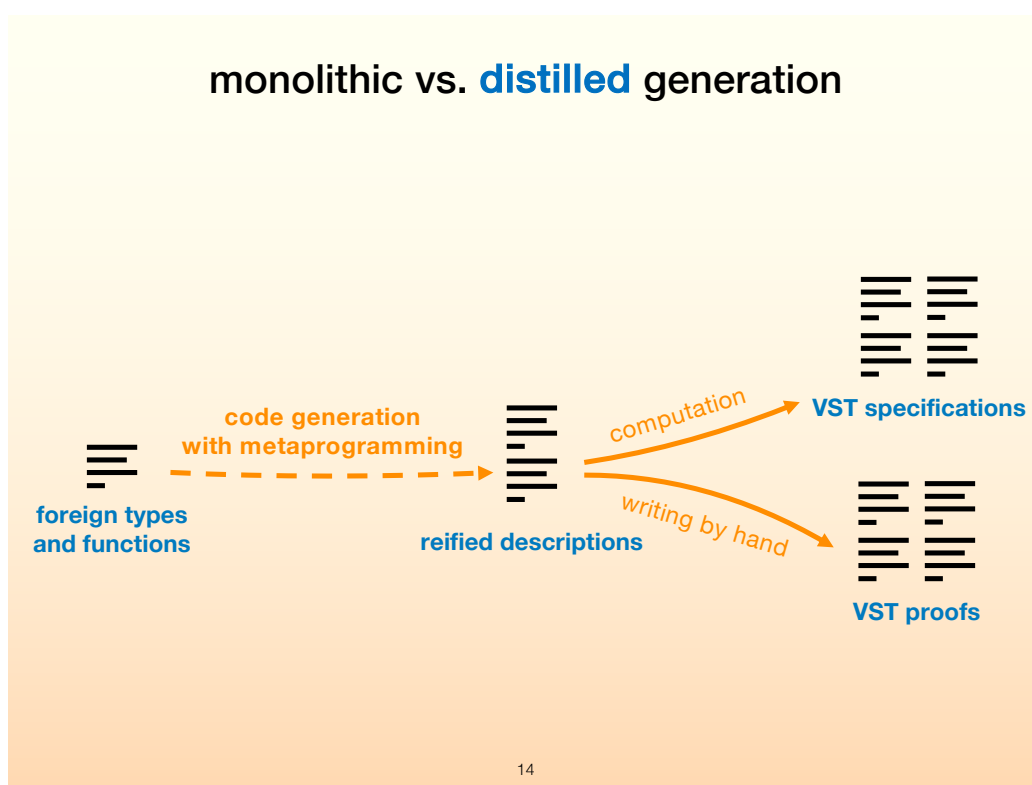
Coq’s metatheory so that’s

understandable, but this means it’s a bit more cumbersome to use for code generation.

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 from a meta-level above
can get clunky.

3. Writing the proofs this way requires
a much deeper understanding of how
everything works, which would render
our system unusable for most users.



Here's what I suggest instead. We come up with an **intermediate representation, that is, reified descriptions.**

<click> We can generate these descriptions using compile-time metaprogramming.

<click> But for the rest, we do not need compile-time metaprogramming. We can write Coq functions that take

these descriptions and compute whatever we need. This way we isolate the metaprogram to the first half of generation. Or we can write VST proofs with these abstractions, which makes them easier to write.

Okay, now we're ready to see what a reified description is.

Takeaway 3:

By making the **describer** and **describee** the same language (Coq), and using higher-order abstract syntax, we can handle dependent types and annotate each component in a **concise** and **type-safe** way.

```
VeriFFI's generation library

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.
```

annotated with
type class instances

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.

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

Those components are the type parameters, the arguments, and the return type. We have a constructor for each of these, which gives us the benefits of deep embedding.

<click> But notice how these two cases take a function as an argument. Thanks to this higher-order abstract syntax-like approach, we get the benefits of a shallow embedding. Most importantly, we get to annotate the description with type class instances that are actually indexed by the type of the component.

<click> These annotations have

access to the same binder context as the type component! This allows us to describe even complicated dependent types.

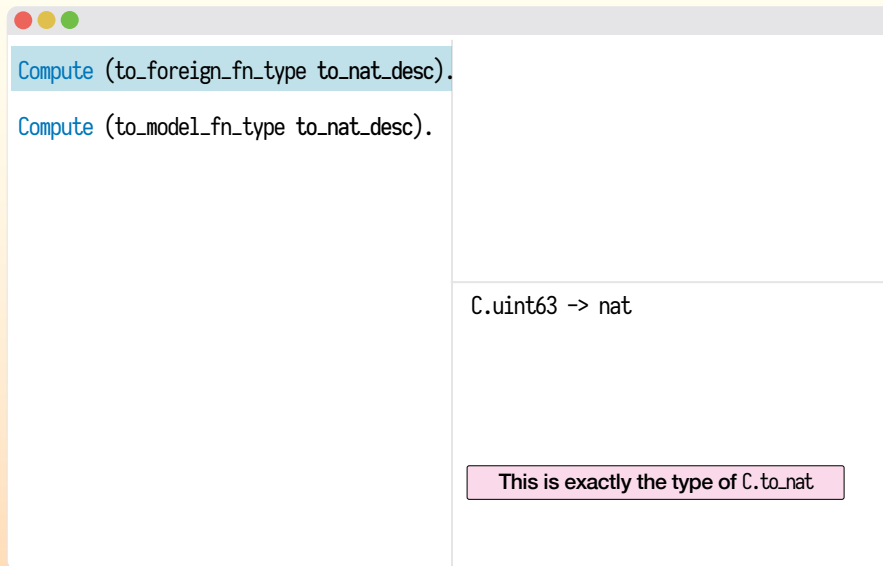
There are other approaches that try to combine deep and shallow embeddings. Their work is more general than ours, which requires complicated mechanisms like the universe pattern. By not being general, we avoid that complexity once again. Here we are describing a part of Coq within Coq, so we can annotate these components with Coq type class instances.

**And that is the other takeaway:
<click> By making the describer and
describee the same language (they
are both Coq), and using HOAS, we
can handle dependent types and
annotate each component in a
concise and type-safe way.**

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

What do reified descriptions buy us?

1. type safety



The screenshot shows a theorem prover interface with a command window on the left and a result window on the right. The command window contains two lines of code: `Compute (to_foreign_fn_type to_nat_desc).` and `Compute (to_model_fn_type to_nat_desc).`. The result window displays the type `C.uint63 -> nat`. A pink callout box at the bottom of the result window contains the text: "This is exactly the type of C.to_nat".

16

Well, why did we go through all this trouble? What do reified descriptions buy us? The most important aspect is type-safety of our specifications.

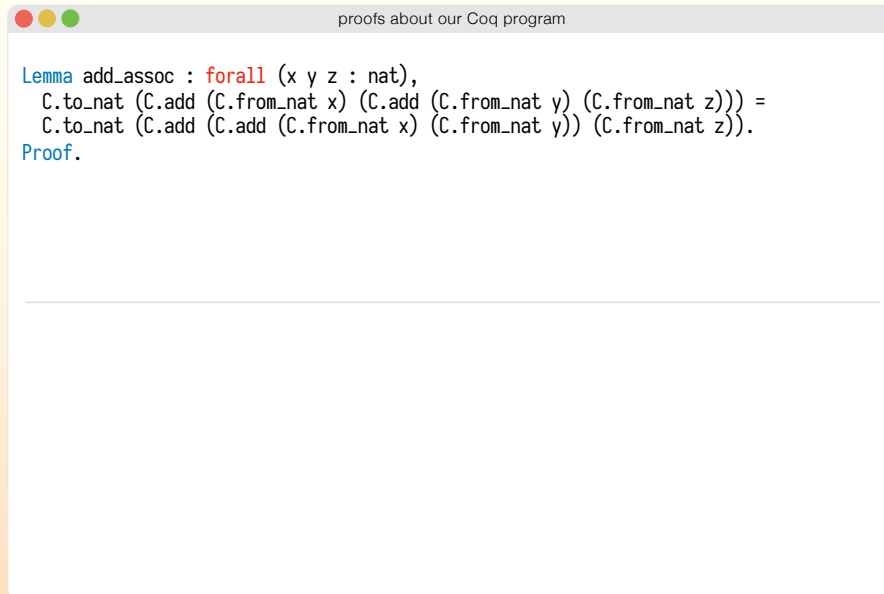
If we have a description of a particular function, like `to_nat`, we can recover the original types from that.

Here we recompute the type of the `to_nat` foreign function from its types'

description. We can do the same thing for the functional model type as well.

Thanks to this, we can build a larger, dependently typed record that has a type description and the functions that actually **are** of that type.

2. rewrites of foreign function calls 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.
```

17

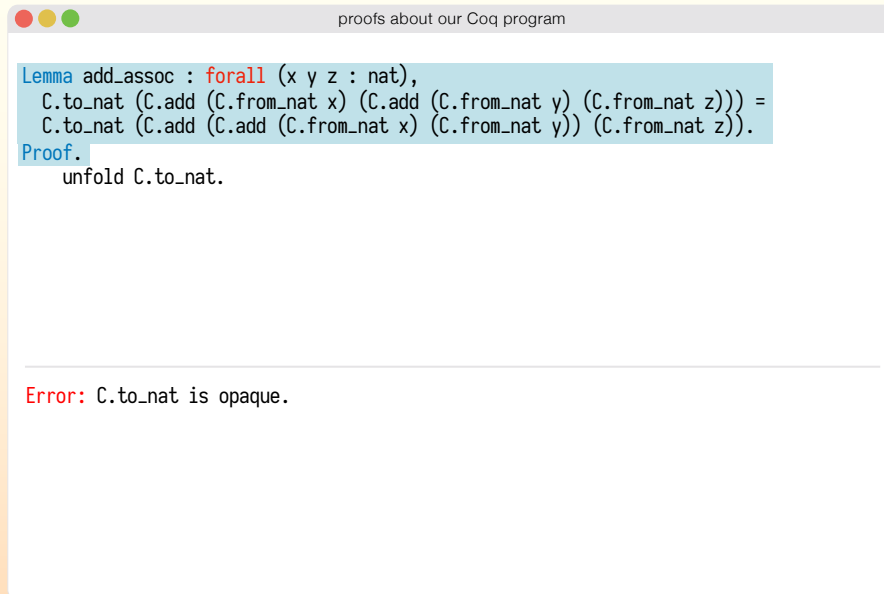
The other thing we use reified descriptions for is rewrites of foreign function calls to functional model calls.

Here's the problem:

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 involving foreign function calls, we have a problem: We cannot unfold the definitions and continue our proof...

2. rewrites of foreign function calls 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.

Since these functions are axioms on the Coq side, they get stuck! These functions evaluate in a compiled program, because then they are realized by C functions, but that's not good enough for compile-time evaluation of them! What do we do, then?

2. rewrites of foreign function calls 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.
  foreign_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))
```

19

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.


```
Module Type Array.
Parameter M : Type -> Type.
Parameter pure : forall {A}, A -> M A.
Parameter bind : forall {A B}, M A -> (A -> M B) -> M B.
Parameter runM :
  forall {A} (len : nat) (init : elt), M A -> A.
Parameter set : nat -> elt -> M unit.
Parameter get : nat -> M elt.
End Array.

Module C <: Array.
Inductive M : Type -> Type :=
| pure : forall {A}, A -> M A
| bind : forall {A B}, M A -> (A -> M B) -> M B
| set : nat -> elt -> M unit
| get : nat -> M elt.

Axiom runM :
  forall {A} (len : nat) (init : elt), M A -> A.
End C.

CertiCoq Register
[ C.runM => "array_runM" with tinfo
] Include [ "prims.h" ].
```

```
typedef enum { PURE, BIND, SET, GET } m;

value array_runM(struct thread_info *tinfo,
                 value a, value len, value init,
                 value action) {
  // ...

  switch (get_prog_C_MI_tag(action)) {
  case PURE: { /* ... */ }
  case BIND: { /* ... */ }
  case SET: { /* ... */ }
  case GET: { /* ... */ }
  }

  // ...
}
```

```
Coq client of foreign functions

Definition incr (i : nat) : C.M unit :=
v <- C.get i ;;
C.set i (1 + v).
```

Before I finish the talk, I want to show an example with side effects. Here is a mutable array example.

We can define the operational side with a free monad, and write an interpreter for it in C. Though we haven't verified the interpreter in VST.

```
proofs about our Coq program

Lemma set_get :
  forall (n len : nat) (bound : n < len) (init : elt) (to_set : elt),
    (C.runM len init (C.bind (C.set n to_set) (fun _ => C.get n)))
    =
    (C.runM len init (C.pure to_set)).
Proof.
  intros n len bound init to_set.
  props runM_spec. foreign_rewrites.
  props bind_spec. props pure_spec. foreign_rewrites.
  props set_spec. props get_spec. foreign_rewrites.

1 goal

n, len : nat
bound : n < len
init, to_set : elt
=====
FM.runM len init (FM.bind (to (FM.M unit) (C.M unit) (C.set n to_set))
                          (fun _ => to (FM.M elt) (C.M elt) (C.get n)))
= FM.runM len init (FM.pure to_set)
```

We can also prove properties about client programs using the rewrite principles.

Takeaways

1. Since the **source language** and the language of **reasoning** coincide (Coq), and the **target language** and the language of **foreign functions** coincide (C), we can **avoid** the combined language approach to multi-language semantics.
2. VeriFFI allows the user to reason **conventionally** in Coq and VST separately and connects these proofs together.
3. By making the **describer** and **describee** the same language (Coq), and using HOAS, we can handle dependent types and annotate each component in a **concise** and **type-safe** way.

22

So, to quickly repeat the main takeaways:

- 1) We have a carefully chosen pair of languages that helps us avoid the combined language approach to multi-language semantics.
- 2) VeriFFI allows the user to reason conventionally in Coq and VST separately, and connects these proofs together.

3) Once again, since we're reasoning about Coq values within Coq, when we describe Coq values, our describer and describee are the same, which allows concise and type-safe annotations.

See our paper**“A Verified Foreign Function Interface between Coq and C” for**

- how exactly are the VST specifications are computed
- generated glue code, and its VST specifications
- more examples, such as
 - primitive bytestrings and the correctness proofs of their operations
 - I/O and mutable arrays

See my dissertation**“Foreign Function Verification Through Metaprogramming” for**

- the metaprogramming details

Future work / work in progress

- End-to-end compiler correctness proof of CertiCoq for open programs, and how it connects to VST
- VST correctness proofs for I/O and mutable arrays operations

That is all I have time for today. For details about the specifications, our paper is the best source. For details about metaprogramming, my dissertation is the best source. Further details of mutations and side effects are also discussed in both.

There is some future work we want to do.

1) Most importantly, we need to complete the end-to-end compiler correctness proof of CertiCoq for open programs, and state how that connects to VST. We have an incomplete, but mostly finished compiler correctness proof for CertiCoq for closed programs but that is now out of date. We discuss in the paper how this work can help us state the theorem for open programs.

2) We want to finish the VST proofs for programs with side effects and mutation.

Comparison with other verified compilers / FFIs

	Œuf (2018)	Cogent (2016-2022)	CakeML (2014-2019)	Melocoton (2023)	VeriFFI (2017-2024)
project	verified compiler	<i>certifying</i> 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)

24

I want to leave you with this final slide of comparisons with similar projects, that I have a hunch that most of the questions will be about. Thank you very much.

DON'T READ THIS, JUST AS A REMINDER:

<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 has 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.