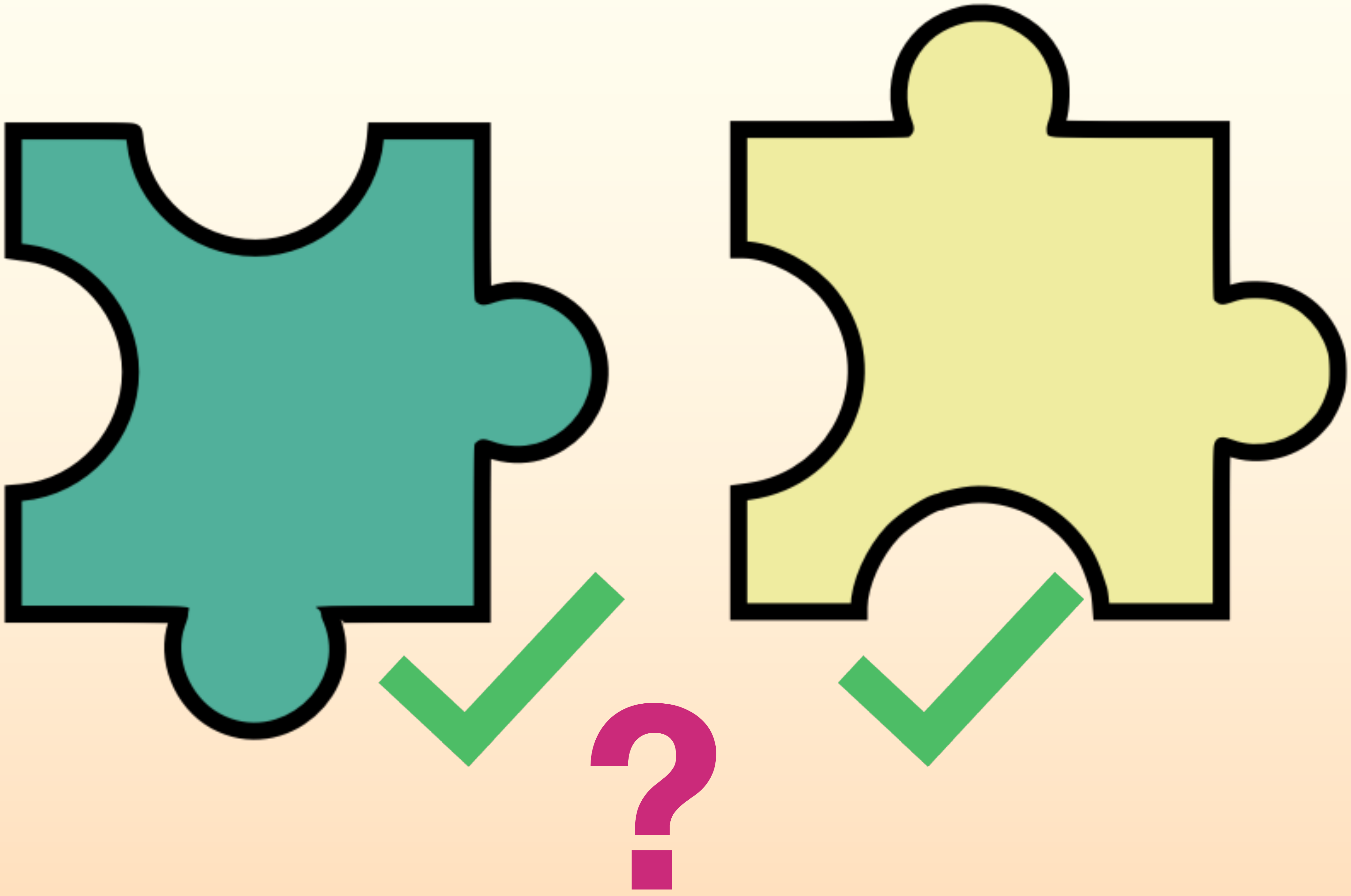
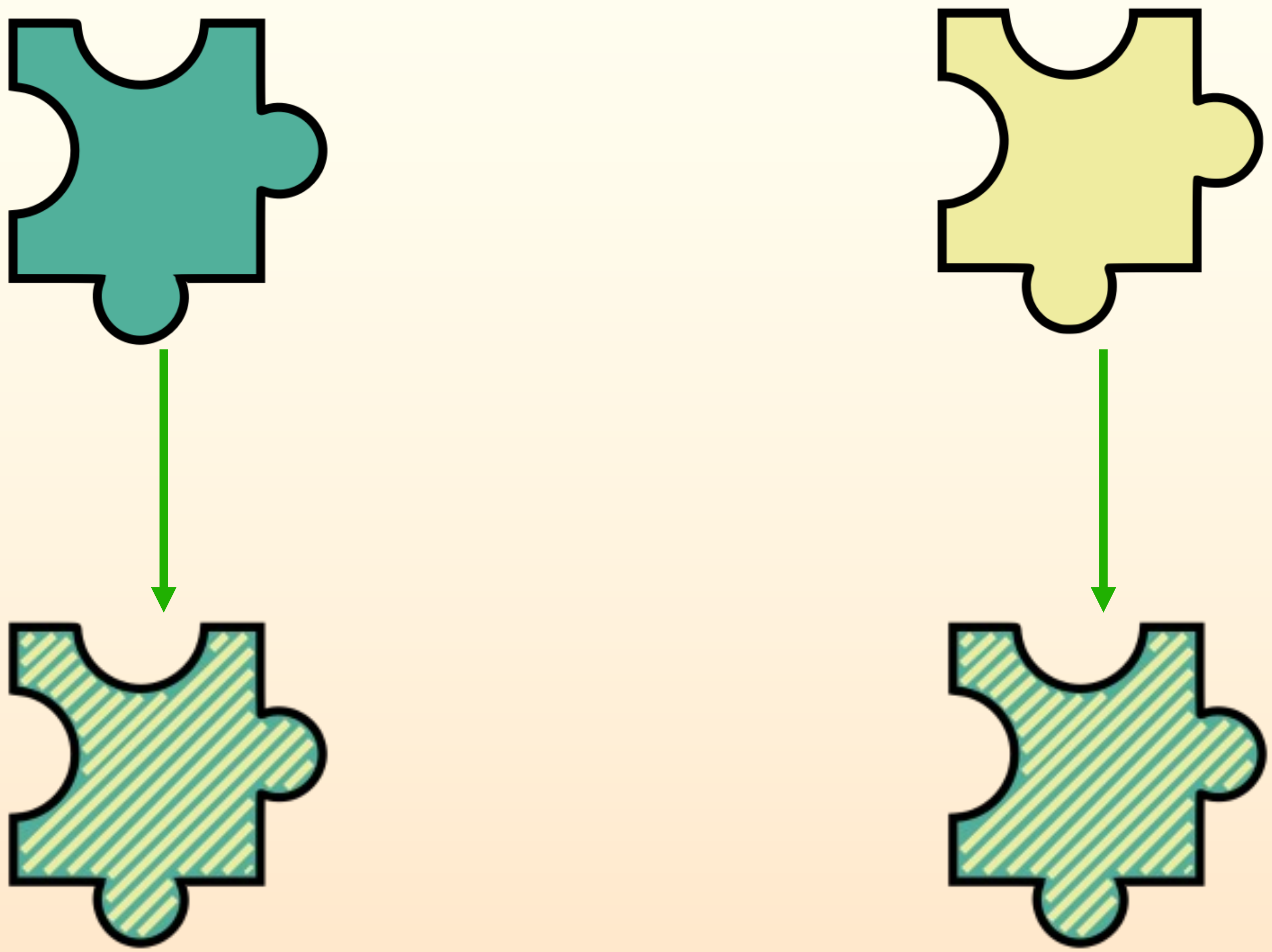
A Verified Foreign Function Interface between Coq and C

Joomy Korkut, Princeton University & Bloomberg*
Kathrin Stark, Heriot-Watt University
Andrew W. Appel, Princeton University

POPL 2025
January 22, 2025

* Ph.D. work done before joining Bloomberg

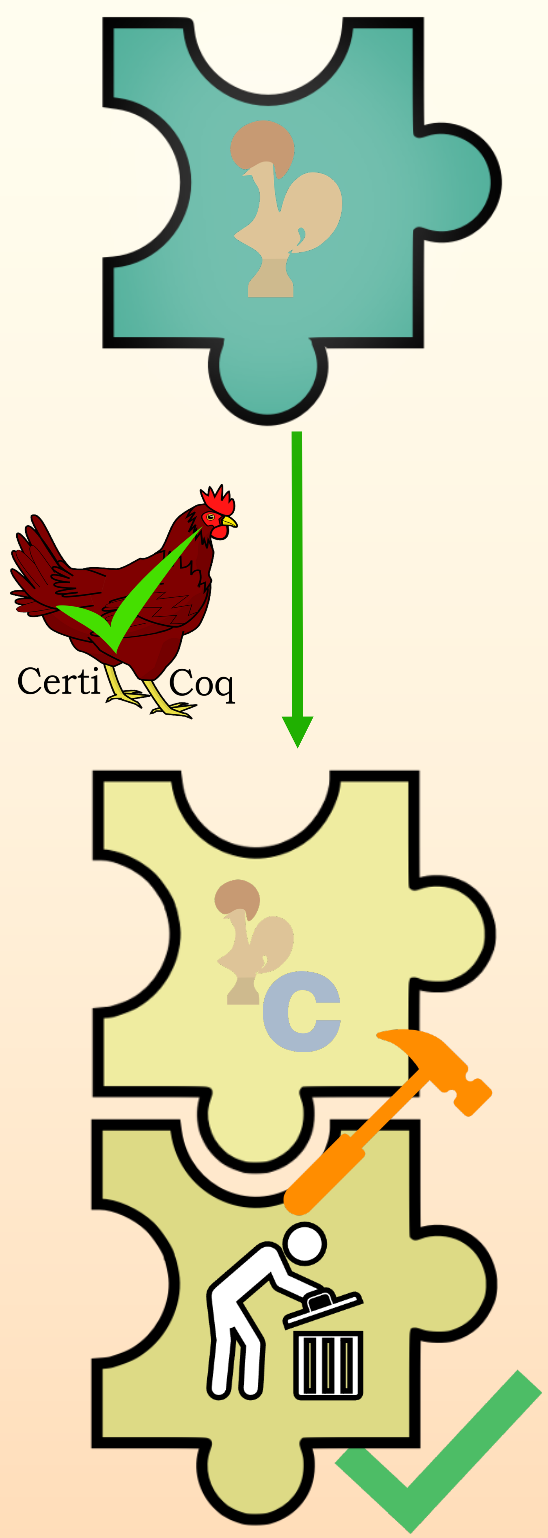


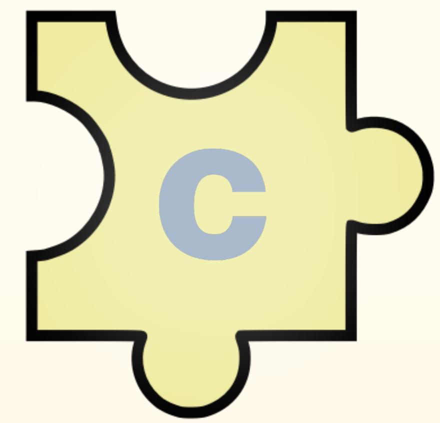


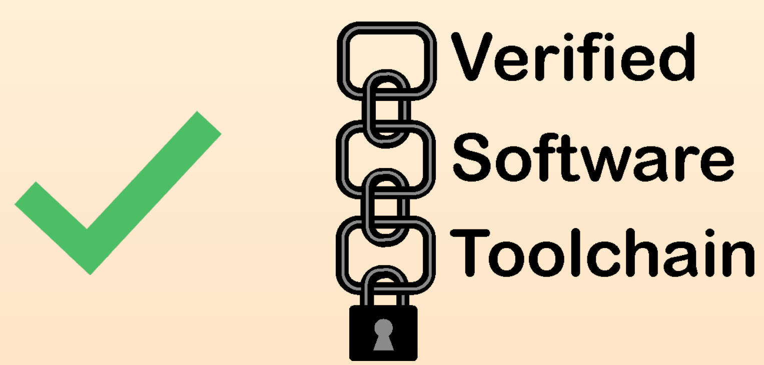
multi-language semantics

Matthews and Findler (2007)

Takeaway 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.







Wang, Cao, Mohan, and Hobor.
"Certifying Graph-Manipulating C Programs via Localizations within Data Structures"
OOPSLA 2019

github.com/CertiCoq/VeriFFI

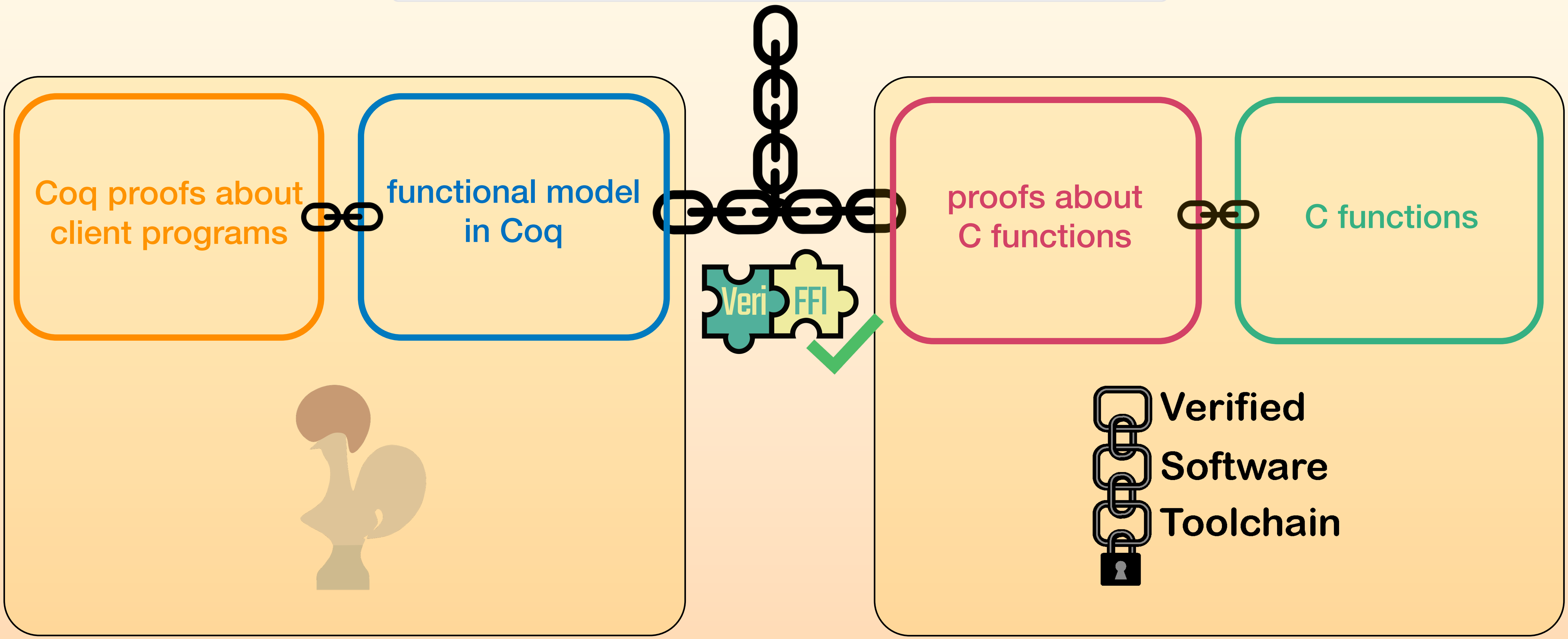
Takeaway 2:
VeriFFI allows the user to reason **conventionally**
in Coq and VST separately and connects these proofs together.

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.
```

abstract type

operations



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 ].
```

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

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 ].
```

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.

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.
```

function description

generate function specification

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"
     |}.

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

generating the function description

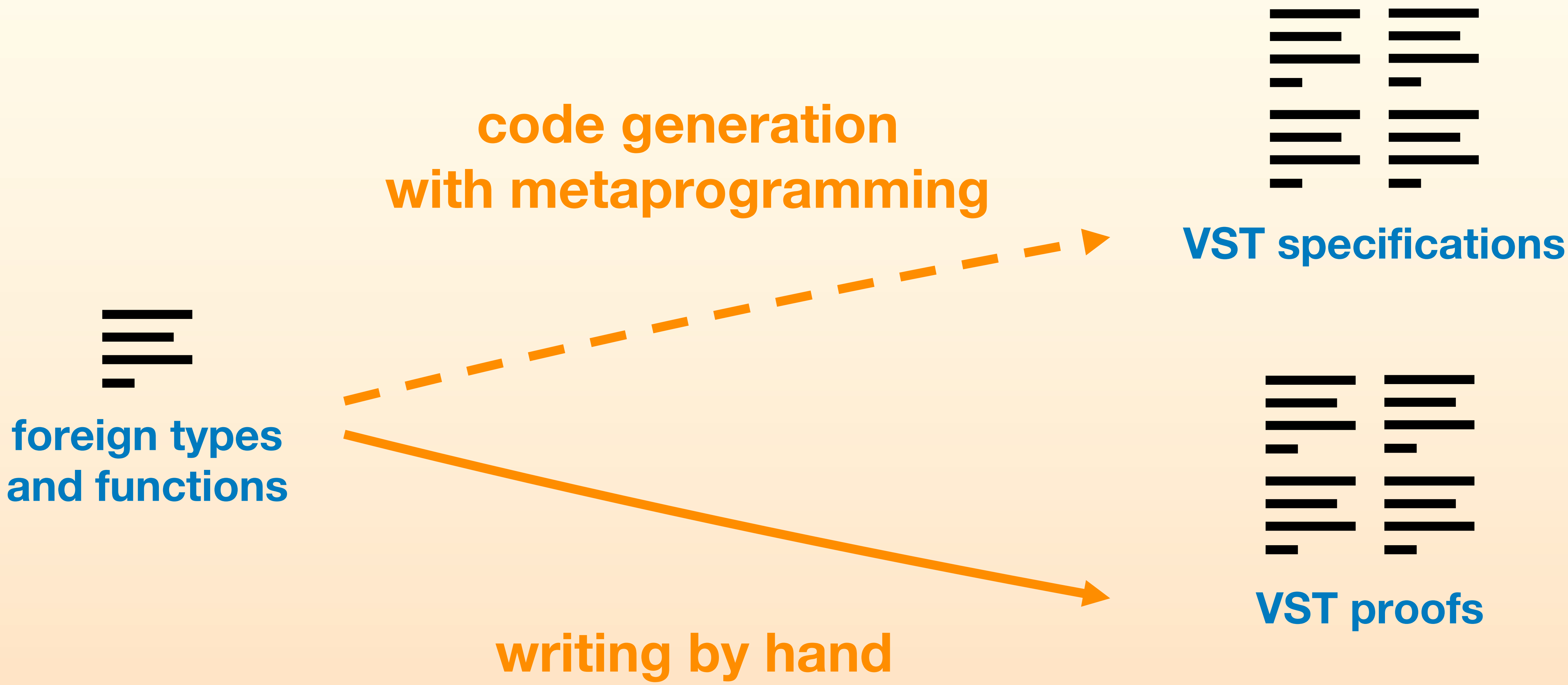
generate function specification

user's Coq proof

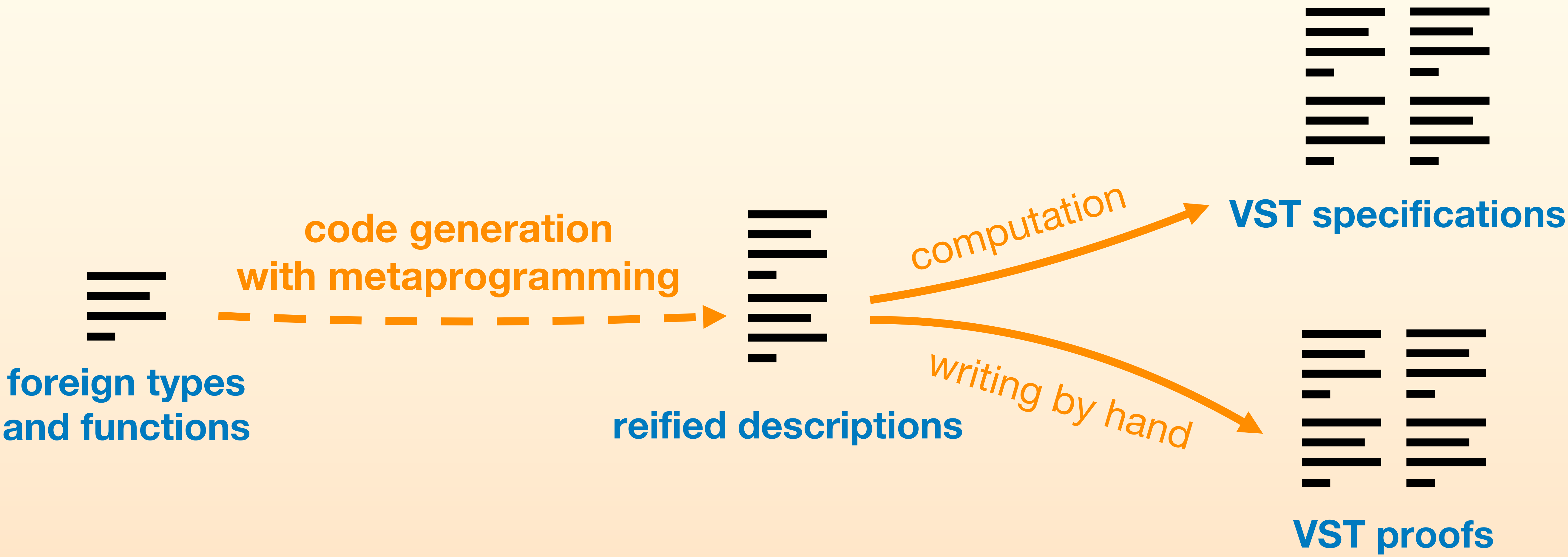
```
MetaCoq Run (fn_desc_gen FM.to_nat C.to_nat "uint63_to_nat").

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

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.



monolithic vs. **distilled** generation



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.

What do reified descriptions buy us?

1. type safety

```
Compute (to_foreign_fn_type to_nat_desc).

Compute (to_model_fn_type to_nat_desc).
```

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

This is exactly the type of `C.to_nat`

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.
```

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.
    unfold C.to_nat.
```

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

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))
```

user's Coq code

```
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" ].
```

user's C code

```
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).
```

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)
```

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.

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

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)