A Verified Foreign Function Interface Between Coq and C

JOOMY KORKUT, Princeton University, USA and Bloomberg L.P., USA
KATHRIN STARK, Heriot-Watt University, Scotland
ANDREW W. APPEL, Princeton University, USA

One can write dependently typed functional programs in Coq, and prove them correct in Coq; one can write low-level programs in C, and prove them correct with a C verification tool. We demonstrate how to write programs partly in Coq and partly in C, and interface the proofs together. The Verified Foreign Function Interface (VeriFFI) guarantees type safety and correctness of the combined program. It works by translating Coq function types (and constructor types) along with Coq functional models into VST function-specifications; if the user can prove in VST that the C functions satisfy those specs, then the C functions behave according to the user-specified functional models (even though the C implementation might be very different) and the proofs of Coq functions that call the C code can rely on that behavior. To achieve this translation, we employ a novel, hybrid deep/shallow description of Coq dependent types.

CCS Concepts: • **Software and its engineering → Software verification**; **Interoperability**; **Formal software verification**.

Additional Key Words and Phrases: foreign function interface, Coq, C

July 12, 2024

1 INTRODUCTION

We want to write functional programs, because proving those correct is simpler than proving imperative pointer programs. After we prove our programs correct, we want to compile and run them. One can prove programs using Coq, whose logic contains a pure functional programming language along with the proof theory for proving that those programs satisfy logical specifications. Then one can "extract" the programs to OCaml, and compile and run them. But the OCaml compiler (written in OCaml) is not proved correct; nor is the OCaml runtime system and garbage collector (written in C). We want *foundational* verification, in which the application program and all these tools can be proved correct in the same machine-checked logic, in theorems that compose together to make a single end-to-end correctness theorem.

For proved-correct compilation, one can use CertiCoq, a compiler from Coq to C that is verified in Coq. It composes with the CompCert verified C compiler and the CertiGC verified garbage collector (see section 2). Thus, a verified functional program in Coq compiled and executed with CertiCoq+CompCert+CertiGC can have the desired end-to-end correctness theorem in Coq.[1]

However, large programs are rarely written in a single language; additional languages are used for better performance or for capabilities that the primary language lacks. In particular, because Coq lacks primitive types,[2] mutation, and input/output actions, CertiCoq-compiled code must interact with another language to have those capabilities. Specifically (for the CertiCoq back-end targeting C), Coq code must be able to call C code and C code must be able to inspect and generate Coq data structures and call Coq code. There are already *foreign function interface* (FFI) systems to handle the *operational* interface between functional languages (ML, Haskell, etc.) and C [Blume 2001; Leroy 1999], or Java-like languages and C [Liang 1999]. Some of these provide APIs for the functional language to traverse C data structures, others provide APIs for C to traverse the functional language's data structures; and all provide APIs for the functional language to call C functions. In these systems, a type-directed "glue code generator" produces APIs and interface functions. Those FFIs make a *dynamic* (operational) connection between the high-level and low-level language; and some work has even addressed type safety [Tan et al. 2006].

But previous work has not addressed dependently typed high-level languages, and most importantly, has not shown how to connect *correctness proofs* of high-level client programs with correctness proofs of low-level primitives. When we prove a functional program correct in Coq's proof theory (the Calculus of Inductive Constructions) and we prove a C program correct in a program logic for C, how does the "glue code" work to connect these proofs together?

We provide a solution to that problem: **VeriFFI**, a **Veri**fied **F**oreign **F**unction **I**nterface between Coq and C (Figure 1). Coq program components are proved correct directly in Coq, C program components are locally proved correct using the Verified Software Toolchain (VST) [Appel et al. 2014], and the connection is made via VST function specifications that are generated by VeriFFI.

Compared to some other verified FFI systems (in section 13 we discuss related work), it's important that our high-level language is a higher-order dependently typed pure functional language embedded in a logic (i.e., Coq). "Functional" programming languages with mutation (such as OCaml) require separation logic for their reasoning on both sides of the FFI [Meijer 2014]; our approach limits separation logic only to the C side. And (unlike other verified FFI systems) our C language verification can be done using a powerful and general proof tool, the Verified Software Toolchain.

Contributions

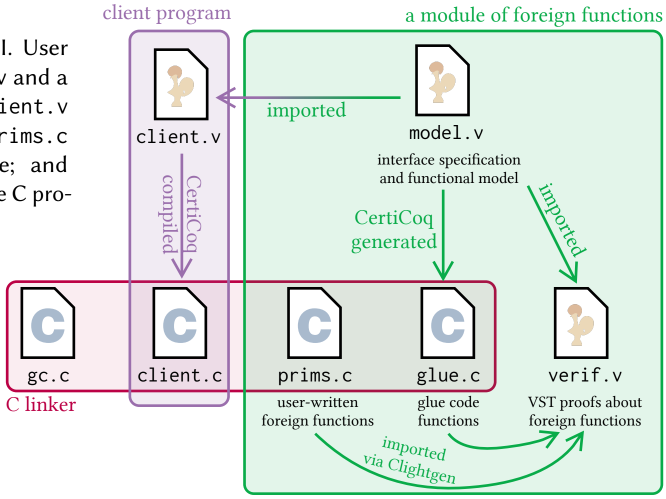
- VeriFFI guarantees both type safety and correctness (except for termination) of the foreign functions, and supports both data abstraction (C functions on types that are opaque to the Coq side) and data transparency (C functions on Coq inductive types).
- We achieve this by calculating C function specifications (pre/postconditions) from Coq dependent types; the user can use VST to prove that the C functions satisfy these specs.
- We calculate these specs using a novel hybrid deep/shallow description of Coq types that allows annotation on each component of a type; the annotations allow analysis and translation of Coq's dependent type structure for this and other applications in metaprogramming.
- Our semantic approach and our glue code generators provide language-local reasoning on the Coq side and the C side without the need for a multi-language semantics.

[1] Each of these components is verified in Coq to specifications that are consistent with each other, but CertiCoq's composed end-to-end correctness theorem has not yet been demonstrated. In fact, our work in this paper informs the statement of that theorem; see section 12.

[2] Or, to the extent that Coq supports primitive types such as 63-bit integers, the correctness of their implementation can be proved by considering their operations as foreign functions.

Fig. 1. Typical usage of VeriFFI. User writes an interface spec `model.v` and a proved-correct client program `client.v` in Coq; writes a C program `prims.c` that implements the interface; and proves in `verif.v` in Coq that the C program is correct.

(Clightgen is CompCert's front end that parses C into a Clight AST, to be verifiable by VST)



2 BACKGROUND

CertiCoq[3] compiles Coq functions by first reifying them into ASTs using MetaCoq [Sozeau et al. 2019], then translating to an untyped intermediate language λANF [Paraskevopoulou et al. 2021] and then to CompCert Clight, a high-level intermediate language of the CompCert verified C compiler [Leroy 2006]. From there, CompCert can compile to assembly language. Each of these languages—(reified) Coq, λANF, Clight, Assembly—has a formal operational semantics in Coq. Coq's formalization is part of MetaCoq, λANF's is part of CertiCoq's proof, and Clight's and Assembly's are part of the CompCert specification. Each of the translations (as well as each optimization pass from λANF to λANF) is proved correct (semantics-refining) with machine-checked proofs in Coq, with respect to the respective operational semantics.

The Verified Software Toolchain (VST). The first phase of CompCert translates C to Clight; Clight programs are readable as C programs, but Clight is an easier language for program verification as it has no side effects inside expressions. The Verified Software Toolchain [Appel et al. 2014] is a program logic and tool for proving functional correctness of Clight programs, and of C programs via their translation to Clight. VST has a formal soundness proof in Coq—that is, if you prove a property of a C program in VST, then that program running in the operational semantics of Clight will respect that property.

VST is used for the correctness proof of CertiCoq's garbage collector [Wang et al. 2019], which is written in C. the Clight code produced by CertiCoq allocates records (from the compilation of inductive data constructors) on a garbage-collected heap, and from time to time it must call the garbage collector (g.c.).

The proof of a C function in VST is with respect to a *function specification* (funspec), that gives the function precondition and function postcondition, all in higher-order impredicative separation logic, and all with respect to a set of quantified variables $\vec{x}$: if the program state before calling f satisfies $\mathbf{pre}(\vec{x})$ and if f terminates (in the Clight operational semantics), then the program state will satisfy $\mathbf{post}(\vec{x})$.

[3] There is no single citable work that describes all of CertiCoq. Separate papers describe different parts of the compiler and runtime: • the workshop paper announcing the beginning of the project [Anand et al. 2017] • CertiCoq's front end is MetaCoq via PCUIC [Sozeau et al. 2019] • the verified translation from MetaCoq to its λANF intermediate language [Paraskevopoulou and Grover 2021] • the verified shrink-reduction optimization phase [Savary Bélanger and Appel 2017] • the verified closure-conversion pass [Paraskevopoulou and Appel 2019] • the composition of all λANF phase verifications [Paraskevopoulou 2020] • the verified code generator [Savary Bélanger et al. 2019] • the CertiGC verified garbage collector [Wang et al. 2019].

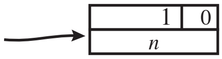
Combining Coq and C. Now, suppose a Coq function g calls a C function f; or more precisely, a Coq function g translated to a Clight function g_c calls a Clight function f. From the MetaCoq semantics of g and a CertiCoq correctness theorem for open programs as proposed in section 12, one would get a Coq proof about the behavior of g_c (subject to an assumption about f's behavior) in Clight operational semantics. Given some appropriate funspec for f in VST's logic, the user can interact with VST to prove correctness of f w.r.t. that funspec. Based on the semantic model of VST funspecs, that gives a Coq proof about the behavior of f in Clight's operational semantics.

VeriFFI's job will be to say what that funspec should be, and to provide the appropriate definitions and tools to make this connection. With VeriFFI, the foreign C function could be the garbage collector, a user-written C function, or a VeriFFI glue-code-generated C function. Any of these functions manipulate C data structures that are the CertiCoq translations of Coq data structures, as well as other C data structures that the C functions use internally. An important part of VeriFFI's job is to enable both concrete data types (C traversal and construction of Coq Inductive types) and abstract data types (whose representation is not known to the Coq client).

Data representations. CertiCoq represents Coq values in memory using the same low-level memory representations as OCaml [Minsky and Madhavapeddy 2022]. In this discussion, we assume a 64-bit word size. Unsigned integers n up to $2^{63} - 1$ are represented in memory as $2n + 1$. Since all pointers are word-aligned (and thus even numbers), this allows the garbage collector to distinguish pointers from nonpointers.

```
Inductive nat := O : nat | S : nat -> nat.
```

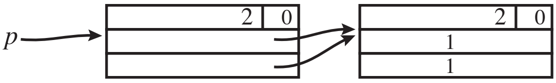
Inductive types such as `nat` are represented as follows. The `O` constructor, as the first constant constructor in this datatype, is represented by an *unboxed* (i.e., tagged as nonpointer) zero, $2 \cdot 0 + 1$. The value `S n` is represented by a *aligned* (even) pointer into a two-word record, where the *header* (at offset -1) contains a *length* (in this case, 1) and a *tag* (in this case 0, for the first *boxed* constructor).



The Coq heap in separation logic. We must describe Coq values in their C representations, using VST's separation logic. Trees in separation logic are typically represented as the separated conjunction of their subtrees, but that can't work for the usual implementation of an ML-like functional language. Consider a program with shared subtrees:

```
    Inductive tree := leaf : tree | node: tree -> tree -> tree.

    let x := node leaf leaf in
    let y := x in
    let p := node x y in ...
```



Because there is sharing between `x` and `y`, we cannot describe this in separation logic as $p \mapsto$ `node x y` $*$ $x \mapsto$ `node leaf leaf` $*$ $y \mapsto$ `node leaf leaf`.

We handle graph structures with sharing using the CertiGraph library [Wang et al. 2019], whose approach is to describe a graph `g` in the "propositional" part of Coq, as a mapping from vertex-numbers to edge-lists (and other information). It is a *labeled* graph, where each vertex-label includes the C address of the record representing that vertex (or, for vertices represented unboxed, the vertex-label has the unboxed value). Then the separation-logic resource (`graph_rep g`) describing this graph is the iterated separating conjunction (big-star) of all of its vertices.

CertiCoq uses a generational garbage collector, proved correct using VST [Wang et al. 2019]. That collector, or any collector, will need a *heap-management data structure* to keep track of memory not currently allocated but available for allocation.

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

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

CertiCoq Register [
  C.from_nat => "uint63_from_nat",
  C.to_nat => "uint63_to_nat",
  C.add => "uint63_add"
  ] Include [ "prims.h" ].
CertiCoq Generate Glue
      -file "glue" [ nat ].
```

(a) Coq side, model.v

```
Definition prog :=
  C.to_nat (C.add (C.from_nat 1)
                  (C.from_nat 2)).
CertiCoq Compile
      -file "client" prog.
```

(b) Coq side, client.v

```
value uint63_from_nat(
         struct thread_info *tinfo,
         value n) {
      value temp = n;
      uint64 i = 0;
      while (get_nat_tag(temp) == S) {
         i++;  temp = get_args(temp)[0];
      }
      return (value) ((i << 1) + 1);
    }

value uint63_to_nat (
         struct thread_info *tinfo,
         value t) {
  uint64 i = ((uint64)t)>>(uint64)1;
  value temp = make_nat_O();
  while (i) {
    if (tinfo->limit - tinfo->alloc < 2) {
      value roots[1]={temp};
      struct stack_frame fr =
             {roots+1,roots,tinfo->fp};
      tinfo->fp= &fr;
      tinfo->nalloc = 2;
      garbage_collect(tinfo);
      temp=roots[0]; tinfo->fp=fr.prev;
    }
    temp = alloc_make_nat_S(tinfo, temp);
    i--;
  }
  return temp;
 }

value uint63_add(
         struct thread_info *tinfo,
         value x, value y) {
  return (value) ((uint64)x+(uint64)y-1);
  }
```

(c) C side, prims.c

Fig. 2. Operational View of the FFI: Code in Coq (left) vs code in C (right).

In the VST proof of a C program that interacts with the CertiCoq garbage-collected heap, the separation-logic assertions will usually have these (separated) conjuncts: **heap**, described by the graph_rep predicate ∗ **thread_info predicate** comprising the heap-management data-structure and the stack of frames (a data structure keeping track of local variables pointing into the heap, following McCreight et al. [2010]) in the function-call stack ∗ **outliers**, data structures outside the garbage-collected heap, to which the heap may point.

3 VERIFFI IN A NUTSHELL

Foreign functions are useful when the C code can use better data structures than Coq's **Inductive**s, or can use mutable data structures, or can access special machine instructions such as cryptographic primitives; or when the program needs to do I/O. We illustrate how to write such programs with

VeriFFI using a simple example: 63-bit unsigned integers as a foreign type, with foreign functions to add (modulo 2^{63}) and convert from/to Coq's natural number type. We use 63-bit integers to leave space for the 1-bit tag that marks unboxed values for the garbage collector.

3.1 Operational

A typical use of VeriFFI is structured as shown in Figure 1. Coq file `model.v` specifies an interface (Coq inductive types, foreign abstract types, and foreign functions with their Coq functional models). Coq file `client.v` has a program that uses the foreign functions. Figure 2 shows an example that uses a C implementation of 63-bit unsigned integers. On the Coq side (Figure 2a, `model.v`), we define an API as a Coq module type `UInt63`, then make the claims in **`Module`** `C` that there are instantiations of type `C.t` and functions `C.from_nat`, `C.to_nat`, and `C.add`. The client can use this API in writing Coq functions (Figure 2b, `client.v`).

Coq's execution compiles the files `model.v` and `client.v` via the CertiCoq compiler, producing:

`glue.c` containing glue code for construction and traversal of Inductives used by the API;
`client.c` the compilation of the client program.

These components link at C level, with the garbage collector (`gc.c`) and with user-written `prims.c` (Figure 2c) which instantiates the axioms in **`Module`** `C`. The C foreign functions have this API:

```
/* prims.h */
#include <gc.h>
value uint63_from_nat(struct thread_info *tinfo, value z);
value uint63_to_nat(struct thread_info *tinfo, value t);
value uint63_add(struct thread_info *tinfo, value x, value y);
```

Each function's first parameter is a thread-info pointer, needed in case the function allocates on the heap. The remaining arguments correspond to the Coq arguments of the (uncurried) Coq function type. Each of these may be a concrete Coq type (such as `nat`) or an abstract Coq type (such as `C.t`). Either way, the C parameter type is just `value`, which is a typedef for `void*`.

The `thread_info` parameter describes (among other things) the location of the next allocable spot in the heap (`tinfo->alloc`); the end of the allocation space (`tinfo->limit`); and other data structures used only by the garbage collector. If `limit` minus `alloc` is less than the size of a new record (including header), then `garbage_collect` must be called.

All the operations on `C.t` are foreign functions—it is an abstract data type—so we are free to choose an efficient representation. Here, we implement `C.t` using C's unsigned 64-bit integers, and we represent `nat` as shown in section 2. The functions in `prims.c` are implemented as follows:

`uint63_from_nat`: To convert the low-level memory representation of `nat` to C's native 64-bit integer type, we have to count the number of `S` constructors in the data structure. The function getting the tag and the one accessing the arguments are *glue functions*, generated automatically in `glue.c`.

`uint63_to_nat`: uses the (automatically generated) glue function `alloc_make_nat_S` to allocate a *successor* constructor on the heap, as many times as called for by the input argument `t` (after its low-order tag bit is stripped off). It is a precondition of `alloc_make_nat_S` that enough space is available; to provably satisfy this precondition, we first test `limit - alloc`. In case a garbage collection is needed, the local variable `temp` is a *root* of the heap, so we need to push a frame on the stack of frames and copy `temp` into that frame; then after the collection, copy back the (possibly forwarded) `temp` and pop the stack.

`add`: To add two tagged integers (modulo 2^{63}), first shift each right to strip the tag; then add; then shift left and add 1. Or do it more efficiently, as shown.

3.2 Verification

Previous FFI systems have been able to "glue" at the operational level as described in subsection 3.1; but VeriFFI can connect specifications and proofs. We start by providing a *functional model* of (in this example) the UInt63 module type:

```
Module FM <: UInt63.
  Definition t : Type := {z : nat | z < 2 ^ 63}.
  Lemma mod63_ok: forall (n : nat), (n mod (2^63) < 2^63).
     Proof. intro. apply Nat.mod_upper_bound, Nat.pow_nonzero. auto. Qed.
  Definition from_nat (n : nat) : t :=  (n mod (2^63); mod63_ok _).
  Definition to_nat (z : t) : nat :=  let '(n; _) := z in n.
  Definition add (x y : t) : t :=
    let '(xn; x_pf) := x in let '(yn; y_pf) := y in ((xn + yn) mod (2^63); mod63_ok _).
End FM.
```

We model a 63-bit integer as a natural number n accompanied by a proof that $n < 2^{63}$. Then our definition of to_nat is trivial (just project out n), but in the definitions of of from_nat and add we must supply a proof that the result is in range, which we do using an auxiliary lemma mod63_ok. You can see in the functional model that the behavior of from_nat and add forces the results to be in range by explicitly doing a modulo operation, which models unsigned integer overflow.

VeriFFI guarantees that C.t and FM.t are isomorphic and that the operations (such as add) respect this isomorphism—provided that the user proves certain things about the C program as specified below. This is sufficient to prove the correctness of the client program. For example, we can prove that the prog in Figure 2 computes the number 3, or that add is associative.

In the next five sections, we will show how VeriFFI represents the reified types of foreign functions and Coq inductive constructors; how (based on these and on user-supplied functional models) VeriFFI generates VST funspecs that serve as theorem statements that the user must prove about the C program implementations. Then in section 9 we complete the UInt63 example:

- the funspec computed for uint63_to_nat;
- the proof that uint63_to_nat satisfies this funspec; and
- the functional correctness proof of the client program, relying on the fact that the foreign functions satisfy their specs.

4 GRAPH PREDICATES SYNTHESIZED FROM DESCRIPTIONS OF DEPENDENT TYPES

To make a *verified* FFI that connects proofs across the interface, we need a specification framework relating Coq data structures to heap-graph vertices. For each Coq inductive type, we must describe (parametrically) how each data constructor is represented as graph edges emanating from a graph vertex. To do that, we use the notion of a *graph predicate*, bundled with its invariants into a Coq type class:

```
Class InGraph (A : Type) : Type :=
  { graph_predicate : graph -> outlier_t -> A -> rep_type -> Prop
  ; has_v : . . .         (* CertiGraph-related property of graph_predicate *)
  ; is_monotone : . . .   (* graph_predicate preserved under heap allocation  *)
  ; gc_preserved : . . .  (* graph_predicate preserved under g.c.-isomorphism *)
  }.
```

A rep_type is a graph vertex, corresponding to the address of a boxed value or an unboxed (odd) integer. That is, graph_predicate g o x p says that value x of type A is represented at vertex p in graph g. *Outliers* (such as the o parameter here) are relevant when there are graph vertices outside the garbage-collected heap.

Of course, each different Coq type A has its own different data representations; hence graph_predicate is not a single fixed predicate, it is a Coq type class indexed by type A. The VeriFFI system automatically constructs instances of this type class, and proves automatically (for each instance) that graph_predicate satisfies the properties specified in the InGraph type class.

For example, consider the inductive type vec, polymorphic lists indexed by length:

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

The (type-indexed) graph predicate for this type is,

```
Instance InGraph_vec (A : Type) (InGraph_A : InGraph A) (n : nat) : InGraph (vec A n) :=
  let fix graph_predicate_vec (n : nat) (g : graph) (o : outlier_t)
                              (x : vec A n) (p : rep_type) {struct x} : Prop :=
    match x with
    | vnil => match p with repZ z => z = 0 | _ => False end
    | vcons arg0 arg1 arg2 =>
        exists p0 p1 p2 : rep_type,
          @graph_predicate nat InGraph_nat g o arg0 p0 /\
          @graph_predicate A InGraph_A g o arg1 p1 /\
          graph_predicate_vec arg0 g o arg2 p2 /\
          match p with
          | repNode v => compatible g v 0 (raw_fields v) [p0; p1; p2] /\
                         raw_mark v = false /\ raw_color v = 0 /\ raw_tag v = 0
          | _ => False
          end
    end in {| graph_predicate := (graph_predicate_vec n); has_v := . . . ; . . .  |}.
```

One can see that vnil is represented by a *constant* (z=0), and vcons n h t is represented as vertex p in graph g, such that p has three out-edges to vertices [p0; p1; p2] (ensured by the compatible predicate), and those also have (type-class-indexed) graph predicates. Importantly, the conjunctions are ordinary, not separating, which permits overlap between the graph structures of p0, p1, p2.

Building such instances is cumbersome and technical, whether or not they involve dependent types. To build such instances automatically, we implemented generators using MetaCoq. These generators inspect a particular inductive data type, identify the other inductive types used in that type, infer the InGraph instances for those types, generate them if they are missing, and prove the required lemmas about it via Ltac. (In the actual implementation, InGraph is split into two type classes: graph_predicate is in a separate type class from the lemmas to make it easier to automatically prove the lemmas for each instance.)

5 REIFIED DESCRIPTIONS WITH ANNOTATIONS

Graph predicates are the basic building blocks of the function specifications of generated glue code and foreign functions. We want to generate these function specifications automatically, but generating VST specifications *directly* from MetaCoq would be difficult. MetaCoq operates on the core language of Coq, and focuses on metatheory rather than easy code generation. The notation-heavy style of VST specifications also make it challenging to generate them from a fully deeply embedded description.

To get around these problems, we introduce an intermediate representation between MetaCoq and function specifications, tailored to the information we require to state a function specification – a *reified description*. We will use metaprogramming to obtain MetaCoq's representation of inductive types and constructors to convert them into our representation; then we can generate the specifications we need from our intermediate representation, in pure Gallina. This isolates metaprogramming to the first half of this conversion and simplifies the specification generation later.

This reified description is defined as:

```
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.
```

Our description type is parametrized by ann, an annotation type class, whose important instances will be *constructor annotation* and *foreign function annotation* (see section 6 and subsection 8.1). Thanks to ann, reified descriptions can carry extra information related to every component of the described type. The reified description type consists of 3 constructors:

- The TYPEPARAM constructor represents type parameters of a function or a constructor. It takes a higher-order function as an argument, where the function takes a Coq type A as an argument, along with a guarantee that there is an instance of the ann type class, and returns another reified description. This way the rest of the description has access to the type parameter and its annotation instance in the context.
- The ARG constructor represents dependent arguments of a function or a constructor. ARG takes the type of the argument, a witness that there is a type class instance for that type, and finally a higher-order function that takes an argument and returns a reified description. This argument allows us to express dependently typed arguments since the argument of the higher-order function can occur in the rest of the description.
- Finally, the RES constructor represents the result type of a function. RES takes the result type and a witness that there is an annotation instance for that type.

Our representation combines both *deep embedding* and *shallow embedding* techniques. The description that would solve our problems had to be traversable, therefore we defined it as an inductive type, like a deep embedding. In the arguments of each constructor, however, we see the Coq semantics of the respective concept: for a type parameter, we have a function that takes a type parameter, for an argument we have a function that takes an argument, resembling a shallow embedding.

This approach can be considered a special case of McBride [2010] or Prinz et al. [2022], except both object and host languages are Coq in our approach. This coincidence enables us to reuse more features of the host language than solely name binding; we can also annotate the components of a Coq type with Coq type class instances, we can interpret a Coq type description back to its corresponding Coq type without extra use of metaprogramming. This allows us to carry values satisfying a type description in a type-safe way, which we use in section 6 and subsection 8.1 to achieve reflection of constructors and foreign functions from their descriptions.

Using the reified type, we can now describe types of functions or constructors. For example, recall the vec type of section 4; its constructors vnil and vcons are described as:

```
(* vnil : forall (A: Type), vec A O *)
Definition vnil_reified : reified InGraph :=
  TYPEPARAM (fun (A : Type) (InGraph_A : InGraph A) =>
    RES (vec A O) (InGraph_vec A InGraph_A O).
```

```
(* 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)))))).
```

Not only inductive constructor types, but dependently typed foreign function types are described by `reified`. For example, the (non-foreign) function `length : forall {A : Type}, list A -> nat` can be described as:

```
Definition length_desc : reified InGraph :=
 TYPEPARAM (fun (A : Type) {InGraph_A : InGraph A} =>
  ARG (list A) (InGraph_list A InGraph_A) (fun (_ : list A) =>
   RES nat InGraph_nat)).
```

Consuming reified descriptions. We have many useful functions on `reified` descriptions, such as the one that calculates a `graph_predicate`. Here we show a simpler one, that calculates the (uncurried) argument type of a function, as a nested dependent tuple of the types of all type parameters and arguments in the description:

```
Fixpoint args {cls : Type -> Type} (r : reified cls) : Type :=
 match r with
 | TYPEPARAM k => {A : Type & {H : cls A & args (k A H)}}
 | ARG A H k => {a : A & args (k a)}
 | RES _ _ => unit
 end.
```

When we need to write a function that needs to quantify over all the arguments that a function or a constructor takes, we can use `args` of a `reified` description to achieve that. For the description of the `length` function, this would calculate:

```
args length_desc = {A : Type &  {_ : InGraph A & {_ : list A & unit}}}
```

We can also write a function that calculates the result type of a function, whose implementation is similar to `args`:

```
Fixpoint result {cls: Type -> Type} (r: reified cls) (xs: args r): {A: Type & cls A} := ...
```

Now, using `args` and `result`, we can write a function that gives us a type that is as close as possible to the original function or constructor type. In other words, we want to reflect the type description to an actual Coq type:

```
Definition reflect {cls : Type -> Type} (r : reified cls) : Type :=
 forall (P : args r), projT1 (result r P).
```

The type we obtain from this function is an uncurried version of the type of `length`. A function of type `reflect length_desc` would take a nested dependent tuple of all the arguments (and the annotations for type parameters) and return the same result type. Here is how that function would be implemented, where the nested tuple is pattern-matched in the parameter to **fun**:

```
Definition length_uncurried : reflect length_desc :=
 fun '(A; (_; (l; tt))) => @length A l.
```

The `reflect` function provides a type-safe way for us to go from the description into the original function. This will allow proofs by reflection, ensuring that the function we have fits the description we were provided.

Curried vs. uncurried. We have chosen to calculate the uncurried type of a multi-argument Coq function because the interface to C (and similar low-level languages) is more efficient and natural with all arguments at once in the uncurried style. Another reason for this choice is that the uncurried function type includes the annotation arguments, which are useful (for example) in calculating the `graph_predicate` instance from the `reified` description of a type. In this section we have instantiated the ann parameters with InGraph, but in the next sections we explain annotations useful for constructor types and for function types.

6 CONSTRUCTOR SPECIFICATIONS

To compose proofs of Coq programs that build and traverse data structures with proofs of C programs that build and traverse those same data structures, the VST separation logic function-specifications for construction and projection must be coherent with the Coq constructors. To accomplish that, we introduce a novel deep and shallow *constructor description*, derivable automatically from MetaCoq descriptions of inductive data types; and an interpretation of those constructor descriptions into VST function specifications.

Constructor descriptions. The glue code generator (section 7) builds C functions that construct Coq values, such as alloc_make_vec_vcons.

We calculate formal specifications of these functions in VST's specification language, from the reified description of the constructors. As usual, reified must be supplied with an appropriate annotation type. For data constructors, class ctor_ann contains the information we need:

```
Variant erasure := no_placeholder | has_placeholder | present.
Class ctor_ann (A : Type) : Type := {field_in_graph : InGraph A; is_erased : erasure}.
```

In section 4, we defined the InGraph type class, which consists of a graph predicate and lemmas about it for a given Coq type. The first field of the ctor_ann type class is an instance of InGraph for each field of the constructor we want to annotate. This allows us to specify how the values of the arguments are represented in the heap graph.

The second field, is_erased, tells us whether a constructor field is erased during compilation: In CertiCoq, *computationally irrelevant* values, such as values of type Type or values of kind Prop, are erased. When they are arguments to constructors or functions, their places are occupied by (unit) placeholders. Some values are entirely erased in the memory representation, such as parameters of inductive types.

Now that we have a ctor_ann type to annotate our reified descriptions with, VeriFFI defines a record that contains all the information we need about a constructor:

```
Record ctor_desc :=
  { ctor_name : string ; ctor_reified : reified ctor_ann
  ; ctor_reflected : reflect ctor_reified  ; ctor_tag : nat ; ctor_arity : nat  }.
```

Along with the name, tag, and arity of a constructor, we include the reified description of a constructor, in the ctor_reified field. Using dependently typed records, we include the field ctor_reflected, the reflected version of the reified description we just included in the record.

Here we can see some example ctor_desc values for the vnil and vcons constructors of the vec inductive type:

```
Definition vnil_desc : ctor_desc :=
  {| ctor_name := "vnil"
   ; ctor_reified := . . .  (* like vnil_reified but with ctor_ann annotations *)
   ; ctor_reflected := fun '(A; (_; tt)) => @vnil A
   ; ctor_tag := 0; ctor_arity := 0  |}.
```

```
Definition vcons_desc : ctor_desc :=
  {| ctor_name := "vcons"
   ; ctor_reified := . . .  (* like vcons_reified but with ctor_ann annotations *)
   ; ctor_reflected := fun '(A; (_; (n; (x; (xs; tt))))) => @vcons A n x xs
   ; ctor_tag := 1; ctor_arity := 3 |}.
```

VeriFFI's glue code generator defines a type class that allows easy transition from the real Coq constructor for an inductive type, into the ctor_desc for that constructor; and we can define instances for every constructor we generate descriptions for:

```
Class Desc {T : Type} (ctor_val : T) :=  { desc : ctor_desc }.
Instance Desc_vnil : Desc @nil := {| desc := vnil_desc |}.
Instance Desc_vcons : Desc @cons := {| desc := vcons_desc |}.
```

Desc does not come with a guarantee that the reified description matches the real Coq value. However, describing the wrong constructor in the Desc instance means the verification of the function specifications will fail later, so it can't lead to unsoundness.

Constructor descriptions are generated automatically; their generation is implemented mostly in MetaCoq and Ltac.

7 OPERATIONAL GLUE CODE GENERATION

CertiCoq Generate Glue generates C-language data-structure traversal and constructor functions for a user-specified set of Coq **Inductive** types:

```
CertiCoq Generate Glue [ vec , nat ].
```

For instance, for the vec type, VeriFFI generates these functions:

```
value make_vec_vnil(void) {  return (value) 1;  }

value alloc_make_vec_vcons
      (struct thread_info *tinfo, value arg0, value arg1, value arg2) {
  value *argv = tinfo->alloc;
  argv[0] = (value) 3072;  argv[1] = arg0;  argv[2] = arg1; argv[3] = arg2;
  tinfo->alloc = tinfo->alloc + 4;  return argv + 1;
}
```

Unboxed constructors, such as vnil, are represented as (odd) integers. Boxed constructors, such as the vcons, are represented as pointers to memory locations that store the constructor arguments. This memory can exist either within the CertiCoq runtime's garbage-collected memory region (the *CertiCoq heap*) or as "outliers" in the *C heap*. The alloc_make_vec_vcons function uses the thread-info to find the next unused word of the g.c. allocation space tinfo->alloc. It is a precondition of this function that at least 4 words of space are available (for the header, the length index, the head, and the tail); prior to calling the function, this precondition may be tested by a quick comparison, or established (if that fails) by calling the garbage collector.

VeriFFI formally specifies and verifies the C code of these glue functions. For each glue code function, VeriFFI generates a VST funspec from the constructor description (ctor_desc), and then automatically produces a VST correctness proof. We will not show the details of glue-code funspecs, but we explain VST funspecs for foreign functions in subsection 9.1.

Discriminating Coq constructors. For each Coq inductive type, VeriFFI generates a C function that allows the user to determine which Coq constructor had been used to create a given value. For example, for the vec type, the function would be:

```
size_t get_vec_tag(value v) {
  if (is_ptr(v)) /* that is, if v is an even number */
      switch (((size_t*)v)[-1]&255) {  /* fetch header, mask out all but constructor tag */
      case 0: return 1;  default: /* unreachable */;
      /* there would be more cases if more boxed constructors than vcons */   }
  else switch (v >> 1) {  /* strip off the tag bit */
      case 0: return 0;  default: /* unreachable */;
```

```
    /* there would be more cases if more unboxed constructors than vnil */  }
}
```

This function returns the *tag* of the constructor used to create this value, an index based on the order in which the Coq **Inductive** listed the constructor names.

Extracting arguments of a Coq constructor. Given a Coq value of an inductive type, to access its constructor arguments, we have a C function that works on values of any inductive boxed constructor:

```
value *get_args(value v) {   /* this function can always be inlined */
  return (value *) v;
}
```

Effectively this casts a pointer into an array of values, so the arguments of an arity-n constructor can be accessed with get_args(v)[0], ..., get_args(v)[n-1].

Calling Coq closures. The CertiCoq compiler represents Coq functions as closures at runtime, which consist of a function-pointer and an environment-pointer. To call these from C, one must fetch the code-pointer, fetch the environment pointer, and pass the environment as one of the arguments to the code-pointer function. We have a C function that implements this protocol:

```
value call(struct thread_info *tinfo, value clo, value arg) {
  value f = ((struct closure *) clo)->func;
  value envi = ((struct closure *) clo)->env;
  return ((value (*)(struct thread_info *, value, value)) f) (tinfo, envi, arg);
}
```

8 FOREIGN FUNCTION SPECIFICATIONS

When proving correctness of a Coq program that calls functions implemented in C and proved correct in VST, the VST function specification must be coherent with an appropriate Coq functional model. In this chapter we show how to generate a coherent VST function specification from a reified function description. Coherence on the Coq side is assured by reflection. Coherence on the C side is assured by a Coq proof using VST's program logic.

8.1 Foreign function descriptions

Foreign functions may use Coq inductive types and also user-defined foreign types such as 63-bit integers or packed strings that are not (efficiently) expressible in Coq inductive types. Since reified descriptions allow us to annotate every component of a function type, we can define an annotation type that contains additional information about the foreign types we may need to use.

A user of our system will define their foreign types and foreign functions as axioms in Coq. In section 3 we showed axioms stating the existence of a C representation and operations on 63-bit unsigned integers; and the corresponding functional model FM:

```
Module C : UInt63.                    Module FM <: UInt63.
 Axiom t : Type.                       Definition t : Type := {n : nat | n < 2^63}.
 Axiom from_nat : nat -> t.            Definition from_nat (n : nat) : t := ...
 Axiom to_nat : t -> nat.              Definition to_nat (x : t) : nat := ...
 Axiom add : t -> t -> t.              Definition add (x y : t) : t := ...
End C.                                End FM.
```

The C module contains Coq axioms for the foreign types and Coq axioms for foreign functions that may use these foreign types. These functions will be realized by C functions through the FFI.

The user must justify all these axioms by defining a type C.t and Coq functions C.from_nat (etc.) such that an isomorphism between modules C and FM can be proved.

To connect the functional model to the C type in a reified description of a foreign function such as add, we provide an *annotation* to reified. For constructor descriptions we instantiate the ann parameter with ctor_ann, and for functions, with foreign_ann:

```
Class foreign_ann (model : Type) : Type :=
  { foreign : Type
  ; foreign_in_graph : ForeignInGraph model foreign
  ; foreign_iso : Isomorphism model foreign
  }.
```

This provides a link between the model type and the foreign type, as well as the graph_predicate representation of the foreign type and an isomorphism between the two types. This isomorphism is needed for user-level proofs about the behavior of the foreign function, which acts on the foreign type *as if* it were acting on the model type.

The foreign type (such as C.t) has a graph_predicate that's (typically) a single vertex, in contrast to the graph predicate for the functional model FM.t which is (in our example) a Peano chain of unary constructor graph vertices. To connect these, in a way that the Coq type class system can properly instantiate the foreign_in_graph component of a foreign_ann, we use the following type class:

```
Class ForeignInGraph (model foreign : Type) : Type :=  model_in_graph : InGraph model.
```

Here, the model_in_graph field is the (single-vertex) graph_predicate of the foreign (representation) type, masquerading as a graph_predicate of the model type; ?? hints at how this is useful.

The isomorphism class is standard:

```
Class Isomorphism (A B : Type) : Type :=
  { from : A -> B
  ; to : B -> A
  ; from_to : forall (x : A), to (from x) = x
  ; to_from : forall (x : B), from (to x) = x
  }.
```

For nonforeign (transparent) types, we define the isomorphism transparently as the identity, so the user can use this fact in Coq correctness proofs. For abstract (opaque) types, we cannot let the user assume that the type is interchangeable with its functional model, only isomorphic:

```
Definition transparent {A : Type} `{IG_A : InGraph A} : foreign_ann A :=
  {| foreign := A; foreign_in_graph := IG_A; foreign_iso := Isomorphism_refl |}.

Definition opaque {A B : Type} `{IG_A : ForeignInGraph A B} `{Iso : Isomorphism A B}
                                                   : foreign_ann A :=
  {| foreign := B; foreign_in_graph := IG_A; foreign_iso := Iso |}.
```

In practice, we typically instantiate foreign_iso with the identity isomorphism, Isomorphism_refl. That's because the isomorphism is there more to enforce opaqueness than to relate two different representations. Relating different representations is done in VST funspecs by graph_predicate instances, as section 9 will explain.

As we will show, from a reified description, we can produce a VST funspec that specifies the correctness of the C function with respect to the functional model (e.g., FM.add) operating on the InGraph representations. Therefore, every *type* **Axiom** is justified by an InGraph representation predicate, and every *foreign function* **Axiom** is justified by a VST funspec and proof.

9 A VERIFIED FOREIGN FUNCTION INTERFACE

Using these reified descriptions of constructor types and of foreign-function types with functional models, VeriFFI sets up the framework for combining C code and Coq code. To relieve the user from boilerplate, it automatically generates the header file prims.h (containing C function prototypes) that informs prims.c (written by the user, containing C functions).

For each Coq inductive type, VeriFFI generates "glue" operations that allow C code to construct and traverse it. On the verification level, VeriFFI defines predicates for the representation of Coq data types, as well as proofs of general operations on these datatypes. It hence helps the user to preserve an abstraction barrier allowing mostly language-local reasoning.

Primitive functions that do not use the CertiCoq heap—such as uint63_add—are straightforward to specify and prove in VST. It is standard in VST (independent of VeriFFI) that the user may supply (for each abstract type) a *representation relation* that relates the functional model of a type (such as Coq nat) to its layout in the C program's data-structure memory. In VeriFFI's use of VST, this representation relation takes the form of a custom graph_predicate show how, for example, 63-bit integers or packed bytestrings are represented as single vertices in the graph; the purpose of foreign_in_graph (subsection 3 is to correctly index Coq's type-class resolution to select that graph_predicate.

But VST function-specifications and proofs get more complicated once we have to refer to the CertiCoq heap: we have to ensure that certain invariants are kept. A key contribution of this paper is in both stating these invariants in an abstract way and ensuring that reasoning is independent of the implementation of these invariants.

The conditions of the garbage-collected heap will typically appear as the separation logic predicate full_gc g t_info roots outlier ti sh gv, describing the current state of the data graph g, a thread info t_info, the roots, a set of outliers, the address of the thread info ti, permissions, and the global variables gv. It further comes with a whole list of consistency conditions.

To reason about the graph, we will use propositional[4] statements on the existence of certain Coq values in the graph; for example, graph_predicate g o n p states that the natural number n is represented in the graph g with outliers o at position p : rep_type. This graph_predicate statement is different for each Coq datatype, using type classes that VeriFFI generates automatically as explained in section 4. To be able to use this statement in the presence of garbage collection, it must be invariant under graph isomorphism, so we use the gc_preserved component of InGraph.

As long as we stay at this abstraction level, the proofs work straightforwardly in VST. For example, the proof of uint63_to_nat_spec proceeds by stepping through the propositions while keeping certain invariants about the graph g (see subsection 9.2). During each step of the loop, the garbage collector might run, producing new graph g', proved isomorphic to g by the specification of the garbage_collect function.

9.1 VST function specification

Based on the reified description of the type and functional model of uint63_to_nat, VeriFFI computes a VST function specification:

```
Definition uint63_to_nat_spec : ident * funspec := fn_desc_to_funspec uint63_to_nat_desc.
```

That is, this aspect of glue code generation is not simply a "script" in Python or Ltac, it can be calculated and reasoned about within the logic. With a bit of automatic simplification, this particular funspec comes out as shown in Figure 3. As in any VST funspec, the **WITH** clause quantifies over all the logical (Coq) variables to be shared between precondition and postcondition. If the caller of

[4] Recall that vertex-in-graph is a "pure propositional" predicate, while graph-in-heap is a separation-logic predicate.

```
Definition uint63_to_nat_spec :  ident * funspec :=
  DECLARE _uint63_to_nat
  WITH gv : gvars, g : graph, roots : roots_t, sh : share, x : {_: FM.t & 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.t 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.
   (* this theorem states that the function body satisfies its spec *)
Proof. ... Qed.
```

Fig. 3. Specification and proof of uint63_to_nat: Most parts will be identical in any specification interacting with Coq data structures; only the highlighted parts are specific to uint63_to_nat. The type of x in the **WITH** clause (which is isomorphic to FM.t) comes from the args function's trivial uncurrying of a 1-argument function. FM.t is the dependent product {n : nat | n < 2^63}; therefore the **PROP** part of the precondition ensures the upper bound on n. The graph_predicate (at the typeclass instance for FM.t) ensures that p is actually an unboxed integer, i.e., it chooses that constructor of the rep_type inductive datatype. Therefore rep_type_val g p is a 64-bit integer value. In the postcondition, the graph_predicate instance for nat ensures that p' is a pointer to a Peano natural number in the graph g'.

this function can find any instantiation of the **WITH** variables for which the precondition is satisfied, then the function will guarantee to satisfy the postcondition with the same instantiation.

In this case, the **WITH** variables are (gv) the C program's static global data addresses; (g) the graph; (roots) all the pointers in the stack-of-frames, i.e., live local heap-pointer variables of currently stacked function calls; (sh) the permission-share for reading/writing the heap; (x) the functional model of the input to the function; (ti) the pointer to the heap-management data-structure; (outlier) a description of all pointers not managed by the garbage collector; and (t_info) a description of the *contents* of the heap-management data structure.

We choose C.t to be equal to FM.t; that is, we instantiate foreign_iso with the identity isomorphism. The purpose of having an opaque isomorphism was just to prevent clients from performing uint63 operations on nat values, or vice versa.

The C function uint63_to_nat converts between two very different number representations in the C memory. This difference in representations is manifest in the function-spec by the choice of two different instances of the InGraph typeclass, that is, graph_predicate instantiations for FM.t and nat respectively in precondition and postcondition. When these definitions are unfolded, it gives the VST specification of a function that must convert a 63-bit integer (with tag bit) to a chain of Peano S constructors.

In detail, this function's precondition says,

PRE [thread_info; int_or_ptr_type] the C function takes two arguments: a pointer to a thread-info data structure, and a heap-value (a word that may be either an odd integer or a word-aligned pointer).

PROP(. . .) the permission-share does indeed give read/write permission on the heap, and the input-argument graph-vertex p corresponds to the input functional-model value x as described above.

PARAMS(ti, rep_type_val g p) The *values* of the C function parameters are the address of the thread-info struct and the C representation of the graph-vertex p. A rep_type such as p can be one of three things: a boxed vertex in the graph (repNode v), an outlier, or an unboxed vertex in the graph (repZ z). The function rep_type_val translates this representation to a C value; in this case the graph_predicate instance in the PROP part of the precondition has forced p to be a repZ, a C integer value.

SEP(full_gc g . . . , mem_mgr . . .) The graph is indeed represented in memory as a separation-logic "resource," with the garbage-collector's heap-management data structure. Separately, the malloc/free memory manager (mem_mgr) is also in the heap, in case the C program needs to use it for non-Coq data.

The postcondition says,

POST [int_or_ptr_type] the C function returns a heap-value.

EX p' g' roots' t_info' there will exist some graph vertex p' representing the newly created nat, a new graph g' (resulting from possibly garbage-collecting the graph g as well as adding the new vertex p'), and new roots and thread-info (since garbage collection may have forwarded the old roots).

PROP(. . .) the new vertex p' is the root of a data structure in graph g' representing the new nat; the new graph g' contains an isomorphism of the old graph g; and the stack of frames is the same (modulo forwarding of root-pointers by the g.c.).

RETURN(rep_type_val g' p') the C function's return value is the address in memory for graph vertex p'.

SEP(. . .) the new graph g' is represented in memory, along with the representation of the updated g.c. management data t_info' and the malloc-free memory manager.

The VST funspec that VeriFFI generates says, "the C function implements its functional model." You can see the functional model in the PROP clause of the postcondition; in this case, it is FM.to_nat. That is, the new graph vertex p' is supposed to be a representation (in the g.c. graph) of the functional model applied to the input argument x. (The projT1 applied to x is an artifact of the degenerate uncurrying of a 1-argument function.)

Most of the predicates used here may be found in any function specification that interacts with CertiCoq/VeriFFI data structures: C globals, a graph, roots of the graph in the C "stack of frames", permission-share for the heap, the address and contents of the thread-info structure. Whenever we interact with (CertiCoq-compiled) Coq code we require full_gc, ensuring wellformedness of the current state. In more detail, full_gc g t_info roots outlier ti sh gv contains

- the spatial representation of the outliers, the thread info, and the graph;
- C global variables used by the collector;
- several wellformedness and compability conditions on the graph, e.g. that there are no backwards pointers and the graph is coherent with the roots and outliers.

9.2 An example proof

We repeat here from Figure 2 the C implementation of `uint63_to_nat` that constructs a Peano natural number by wrapping *n* heap-allocated `S` constructors around an `O` constructor:

```
value uint63_to_nat (struct thread_info *tinfo, value t) {
  uint64 i = ((uint64)t)>>(uint64)1; /* strip off the tag */
  value temp = make_nat_O();  /* create the base case */
  while (i) {
    if (tinfo->limit - tinfo->alloc < 2) {  /* test whether we need to garbage-collect */
      value roots[1]={temp};      /* register the root-pointer temp */
      struct stack_frame fr = {roots+1,roots,tinfo->fp};
      tinfo->fp= &fr;
      tinfo->nalloc = 2;   /* state the need for 2 words */
      garbage_collect(tinfo);
      temp=roots[0]; tinfo->fp=fr.prev; /* fetch temp back and pop the frame stack */
    }
    temp = alloc_make_nat_S(tinfo, temp);  /* wrap an S constructor around temp */
    i--;
  }
  return temp;
 }
```

The user must then use VST to prove that this C function (whose abstract syntax in Coq we call `f_uint63_to_nat`) satisfies the `uint63_to_nat_spec`—a lemma of the form,

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

Proof. We start by proving that the initial value of `temp` contains a representation in graph `g` of the natural number `0`, that is, `graph_predicate g outlier 0 p`. Calling `make_nat_O` provides us with a vertex `p` satisfying this condition. Behind the scenes, `p` will simply be a leaf in the graph, represented by `repZ`; this information is abstracted from the user.

For the while loop, we require a loop invariant. This one states that there exists `v : rep_type`, `m : nat`, `g' : graph`, thread info `t_info'`, and a set of `roots'` such that `m≤n`, `v` is the `nat` representation of `n` in graph `g'` (`graph_predicate g' outlier m v`), the new graph and forwarded roots are isomorphic to the old graph and original roots (`gc_graph_iso g roots g' roots'`), and all the g.c. invariants hold on the new state (`full_gc g' t_info' roots' outlier ti sh gv`).

Before the while loop, the loop invariant is easily satisfied by using the postcondition of the first two commands (assigning `i` and `temp`) and reflexivity of graph isomorphism. Similarly, it is very straightforward in VST to prove that the loop postcondition implies the function postcondition.

In the loop body, we first check whether we still have enough space on the heap and call the garbage collector if we do not. The correctness proof for CertiGC's `garbage_collect` handles, among other things, the stack of root-frames starting at `tinfo->fp` and the establishment of an isomorphic graph `g'` with enough headroom, which still satisfies `full_gc g' t_info' roots' outlier ti sh gv` and in which constructions that were saved in the roots are preserved. In this case, those constructions include `temp`, which was saved in the topmost frame.

In the loop body after the g.c. test, a new `S` constructor is allocated by calling `alloc_make_nat_S`, whose precondition is that `tinfo->limit - tinfo->alloc` ≥ 2, which has been established by the if-statement. Afterwards, `i` is decreased by 1. The loop invariant is then reestablished, using the postcondition of the funspec for `alloc_make_nat_S` and transitivity of the isomorphism predicate.

VeriFFI Support. To assist with proofs such as the one shown here, VeriFFI provides a library of g.c.-graph isomorphism properties for use in proofs of foreign functions and provides VST-Floyd [Cao et al. 2018] tactical provers for common patterns such as those used here.

Automatically generated glue functions (that construct and traverse Inductive types) are proved fully automatically by a tactic that uses some of the same techniques as shown here. Note that, different to this section, these glue functions go below the abstraction barriers – and hence the proofs have to go below these abstraction barriers and to technical graph manipulations as well: For example, to prove `alloc_make_nat_S` correct, it has to be proven that the newly generated graph still satisfies all the wellformedness conditions in `full_gc g' t_info' roots' outlier ti sh gv`.

10 PROVING CLIENT PROGRAMS CORRECT USING FUNCTIONAL MODELS

VeriFFI uses functional models in Coq as specifications of (foreign) functions (operating on foreign abstract types). Recall that the functional model and the actual C representation are connected by isomorphism (section 8). In our example, the functional model of 63-bit int is a range-bounded Peano natural number (a dependent product type), and the functional models of the operations are Coq functions on that type. Proofs of correctness properties of client programs can make use of these functional models. For example, one can easily prove that the `prog` of Figure 2, which converts 1 and 2 to `C.t`, then adds them, then converts back, results in 3.

Here we show how, using functional models in a client-side proof, one can show that `C.add` is associative:

```
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.
  (* Step 1: VeriFFI tactic to unpack isomorphisms between C representation and FM *)
  props to_nat_spec.  props add_spec.  props from_nat_spec. prim_rewrites.
  (* Proof goal is now,
    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))  *)
  (* Step 2: an ordinary Coq proof about the functional model *)
  unfold FM.add, FM.from_nat, FM.to_nat.
  unfold proj1_sig.
  rewrite <- !(Nat.Div0.add_mod y z), <- !(Nat.Div0.add_mod x y), <- !(Nat.Div0.add_mod).
  f_equal; apply Nat.add_assoc.
  all: apply Nat.pow_nonzero; auto.
Qed.
```

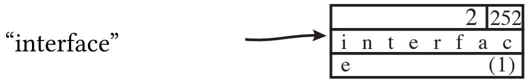
11 A SECOND EXAMPLE: PACKED BYTESTRINGS

In this section, we give another example of a foreign function that manipulates an abstract type. The Coq `string` type is defined as a list of `ascii`, each of which is record of 8 booleans:

```
Inductive ascii := Ascii : bool->bool->bool->bool->bool->bool->bool->bool->ascii.
Inductive string := EmptyString : string | String : ascii -> string -> string.
```

Each `String` constructor is represented as three 64-bit words (a header and two pointers); each `Ascii` constructor is nine words, in which each boolean is an unboxed constructor, with 1 representing true and 3 representing false. In all, 96 bytes per ASCII character.

As a foreign type with foreign functions, we can provide a packed *bytestring* representation, in which each character occupies one byte, as in OCaml [Minsky and Madhavapeddy 2022, Chapter 23, "string values"]. The header tells the number of 8-byte words, and the last byte of the last word tells how many bytes in that word are meaningful. The special tag 252 indicates that none of the words in the record are pointers—none should be traversed by the garbage collector—so they don't need to use the last bit of each word to distinguish pointers from integers.



The CertiCoq code generator cannot manipulate the contents of a packed string, because it is not built using ordinary inductive data types. Instead, we can implement it as an abstract datatype, with operations implemented in C and specified using VeriFFI.

11.1 Description of the pack function

With the type `bytestring`, the user has chosen a functional model for that type, as a vehicle for describing the functional models of its operations: `FM.bytestring := string`. Now the user provides a Coq type for `pack : string -> C.bytestring`, as well as a functional model `FM.pack`. The type of `FM.pack` is completely determined by the type of `pack`, as `string -> FM.bytestring`, which is to say `string -> string`; but what function of that type should it be? Since bytestrings are intended to be an isomorphic (but more efficient) representation of strings, the most straightforward specification choice is the identity function: `FM.pack (x : string) : FM.bytestring := x`. Unlike `C.bytestring` and `C.pack`, which are opaque to the Coq-side client, `FM.bytestring` and `FM.pack` are transparent definitions so the client-side proofs can reason about behavior.

With all these components user-specified, VeriFFI automatically generates the reified description:

```
Definition pack_desc : fn_desc :=
  {| type_desc := ARG string _ (fun _ : string => RES FM.bytestring _)
   ; foreign_fn := C.pack
   ; model_fn := fun '(s; _) => FM.pack s
   ; f_arity := 1
   ; c_name := "pack"
  |}.
```

Using VeriFFI's `args` function (presented in section 5), we can compute the argument type of `pack`. That is, `args pack_desc = {_ : string & unit}`. This is isomorphic to `string`, as we would expect. And therefore, `model_fn pack_desc` (shown as a field of `pack_desc`) simply applies `FM.pack` to its argument, modulo the type isomorphism.

So, the functional model `FM.bytestring` is simply `string`, and the functional model `FM.pack` is simply the identity function. In proofs of the Coq client functions that call *pack* returning results of type `C.bytestring`, one can use the functional model as a reasoning principle by isomorphism between `C.bytestring` and `FM.bytestring`, but the Coq-side client does not know whether type `C.bytestring` is *equal* to `FM.bytestring` (and hence it cannot possibly know whether `C.pack=FM.pack`). As we will explain, in the VST proofs, we do choose `C.bytestring := FM.bytestring` and `C.pack := FM.pack`.

11.2 Implementation of the pack function

We have a hand-written C implementation of the `pack` function that works as follows:

(1) Traverse the string to calculate its length *len*.
(2) Test that at least $n = 1 + \lceil (len + 1)/8 \rceil$ words are available in the g.c. "nursery".
(3) (If not, save live pointers into the stack-of-frames, call the garbage collector, fetch live pointers from the stack-of-frames.)
(4) Reserve n words of space in the nursery (by adjusting the *heap management data structure*).

(5) Traverse the string again, translating records-of-8-booleans into bytes, and storing those bytes into the new space.
(6) Store the header word and trailer bytes (as in the "interface" example).

This function is a bit tricky, because during the traversal at step 5, the heap-management data structure is not coherent with the graph (because one record has been removed from the former but not yet added to the latter). No native Coq function would ever read from the graph during such an incoherence. The proof takes care to accommodate this slightly relaxed invariant.

11.3 Specification of the pack function

As usual, VeriFFI computes the VST funspec for `pack` from the `reified` description and functional model, producing something equivalent to the following:

```
Definition pack_spec : ident * funspec :=
  DECLARE _pack
  WITH gv : gvars, g : graph, roots : roots_t, sh : share, x : {_: string & 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 string 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 bytestring g' outlier (FM.pack (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).
```

This funspec is much like the one described in subsection 9.1, and only the highlighted parts differ: the abstract type is `bytestring` rather than `C.t`, and the functional model is `string` rather than `nat`. And as in that example, although `bytestring` has a very different representation than `string`, this difference is not reflected in the `foreign_iso` component of the `InGraph` class, which is just an identity isomorphism. The difference in representations is accomplished by using different type-class instances for `graph_predicate` in the precondition (where it is for `string`) and in the postcondition (where it is for `bytestring`). Recall that `graph_predicate` describes how a Coq type is laid out in the graph; the `bytestring` instance uses just a single graph-vertex containing all the bytes of data, whereas the `string` instance uses a chain of `Ascii` constructors. And even though `bytestring` is convertible with `string`, typeclass resolution is by name, not by value.

The user must then use VST to prove that the hand-written C function satisfies `pack_spec`. The proof is hundreds of lines long. The remark above that "this function is a bit tricky" translates to extra work proving that this trickiness is done correctly.

12 SOUNDNESS

Assuming that the CertiCoq compiler is correct, then the VeriFFI system is sound. In this section we explain the basis for that claim, and how (in future work) it could be proved.

We rely primarily on the verified-in-Coq soundness of the Verified Software Toolchain [Appel et al. 2014]. Here we explain the VST soundness theorem informally. Suppose one has a set of functions named $i_0, i_1, \ldots, i_{n-1}$ with function-bodies (including headers) $f_0, \ldots, f_{n-1}$ and funspecs $s_0, \ldots, s_{n-1}$. We collect the funspecs into a *context* $\Gamma = [(i_0, s_0); \ldots; (i_{n-1}, s_{n-1})]$. Suppose we prove the correctness of each function individually:

$$\texttt{semax\_body}\ \Gamma\ f_j\ (i_j, s_j) \qquad \text{that is,} \qquad \Gamma \vdash \{\text{pre}(s_j)\} f_j \{\text{post}(s_j)\}$$

such as the lemma body_pack mentioned in subsection 9.2. Whenever a function-body f_j calls some function named i_k, the correctness proof can assume the specification (i_k, s_k) for that function found in assumption Γ (even if $j = k$, i.e., recursion is supported by VST's step-indexed semantic model). Suppose the initial state when calling function f_0 satisfies the precondition pre(s_0). **Then** executing function f_0 in the operational semantics of CompCert C will not crash, and if it terminates, the resulting state will satisfy the postcondition post(s_0).

When using VeriFFI, we have:

- The top-level Coq function g_0 that has internal functions $g_{n+1}, \ldots, g_{n+m}$, all of which are clients of foreign functions named $i_1, \ldots, i_n$; that is, the $i_1, \ldots, i_n$ are free variables of g_0. Function g_0 is compiled by CertiCoq into a C function f_0 with auxiliary functions $f_{n+1}, \ldots, f_{n+m'}$, which are not internal to f_0 because they have been hoisted to top level after closure conversion; and m' may differ from m because of optimizations and transformations by the CertiCoq compiler.
- C functions $f_1, \ldots, f_n$ with names $i_1, \ldots, i_n$. These functions have functional models $g_1, \ldots, g_n$ from which VeriFFI automatically generates funspecs $s_1, \ldots, s_n$ as described in subsection 9.1.
- One garbage_collect function written in C, and the auxiliary functions it calls.

We generate the funspecs of the $f_1, \ldots, f_n$ from their functional models $g_1, \ldots, g_n$, using fn_desc_to_funspec (for user-written C functions) or in a related way for glue-code-generated C functions that allocate data constructors. We generate a funspec s_0 from function g_0 using fn_desc_to_funspec. We do not need funspecs for $f_{n+1}, \ldots, f_{n+m}$ because the proof of correctness of CertiCoq relates those functions to f_0 using direct operational-semantic methods.

From the CertiCoq compiler-correctness claim, we hypothesize, semax_body Γ f_0 (i_0, s_0). The CertiCoq team has not yet completed this correctness proof: the entire front-end is proved correct in Coq [Sozeau et al. 2019], the entire λANF back-end is proved correct in Coq [Paraskevopoulou 2020; Paraskevopoulou et al. 2021], the code generator is proved correct, but the composed end-to-end theorem is still under construction.

In fact, CertiCoq's end-to-end compiler correctness theorem *for open programs (i.e., with foreign functions as free variables),* has not yet even been stated. Our work here provides the framework for doing so, and suggests that the proof should follow VST's semantic method for stapling together a collection of semax_body proofs of mutually recursive higher-order functions. That is, the CertiCoq theorem should relate the Coq function g_0 to the operational behavior of the generated code f_0, subject to assumptions about the operational behavior of the N primops. We can talk about operational behavior in this way because VST's semax_body predicate is a shallow-embedded definition stating properties of a CompCert Clight operational-semantic execution—not, for example, an inductive definition which could only be proved by a certain set of Hoare-logic proof rules.

Now, for every one of these functions we need a semax_body proof of its correctness w.r.t. its funspec.

- The garbage_collect function was proved correct by Wang et al. [2019] using VST.
- The proof of semax_body Γ f_0 (i_0, s_0), relying on related proofs for $f_{n+1}, \ldots, f_{n+m'}$, will be a consequence of CertiCoq compiler correctness, as described above.
- The semax_body proofs of $f_1, \ldots, f_n$ are done using the Verified Software Toolchain's VST-Floyd proof automation system. For those of the f_j that allocate or discriminate data constructors, whose C functions were generated fully automatically by VeriFFI glue code, VeriFFI generates these VST-Floyd proofs automatically using an Ltac script. For the f_j functions whose C

functions are written by hand by the user, the VST-Floyd proofs are done interactively by the user, with assistance from VST-Floyd.

All these `semax_body` proofs can be tied together using VST's `semax_func` constructor lemmas [Appel et al. 2014, page 207] into a single program-correctness proof.

Total correctness vs. partial correctness. The source function g_0 provably terminates (because it type-checks in Coq); and the functional models $g_1, \ldots, g_n$ are total functions (because they are expressed as functions in Coq). However, the VeriFFI+CertiCoq proof of the whole program will not guarantee termination, because VST's program logic is a Separation Hoare logic of partial correctness. This is not a defect of VeriFFI; it is inherent in compiling Coq to any computer architecture with a fixed number of address bits (e.g., 64-bit addresses). The Coq function that computes Ackermann's function on Peano natural numbers is a total function, but compiled to Risc-V it will inevitably run out of memory even on smallish inputs.

13 RELATED WORK

13.1 Verified FFI systems

Melocoton [Guéneau et al. 2023] allows users to write programs in a toy subset of OCaml and a toy subset of C and reason about both sides and their interactions. Users can verify their OCaml code in an OCaml program logic, and their C code in a C program logic, where both program logics are defined on top of Iris, a separation logic framework embedded in Coq. Following the conventional way of verifying interoperability through a combination of languages [Matthews and Findler 2007; Perconti and Ahmed 2014], Melocoton defines operational semantics and program logics for C, OCaml, and their combination, a "multi-language semantics". The user does not have to interact with the combined language and its program logic, but the combined program logic is essential to tie the separate parts together. Melocoton does not include a verified garbage collector, but it has reasoning based on a nondeterministic model of a garbage collector.

In contrast to Melocoton, VeriFFI allows users to write programs in all of Gallina and almost all of C. The user can reason about their Coq programs directly in Coq, which is already a logic and proof assistant and therefore easier to reason in, and about their C programs in Coq via the Verified Software Toolchain [Cao et al. 2018], a separation logic framework embedded in Coq.

For VeriFFI we did not have to develop a combined language and a combined program logic for two languages; it has a simpler architecture than Melocoton because of the languages it is based on: Coq is both our language of reasoning, and the source and implementation language of our compiler. On the other end of the spectrum, C is both the target language of our compiler and the language of our foreign functions. This coincidence means our multi-language programs can just be "plugged together," as both the compiler output of our Coq code and our foreign functions are in C. Hence, all of our reasoning about foreign functions can be achieved within the Verifiable C program logic. VeriFFI is also based on a verified garbage collector, CertiGC, whose heap graph representation is essential in how VeriFFI reasons about the representation of Coq values in memory, and whose implementation can be linked to compiled to Coq programs.

Cogent [Cheung et al. 2022] allows one to write functional programs in the HOL logic, that type-check in HOL and can be proved correct in HOL; but that *also* type-check in a much more restrictive first-order linear type system—that is, no higher-order functions, no sharing of data structures. These first-order linear programs are compiled to C code that (because linear) can use `malloc`/`free`, and do not require a garbage collector. Although that is a reasonable trade-off to make, it severely restricts the expressiveness of the functional language.

CakeML [Kumar et al. 2014] is a compiler for a subset of Standard ML, verified in the HOL4 proof assistant. Guéneau et al. [2017] integrate Characteristic Formulae, a separation logic for stateful

ML programs, into CakeML. This system supports foreign functions as well, but ultimately this system reasons about ML, the higher-level side of the two languages interacting via the FFI. Hence, it is possible to write specifications on how the foreign function is used in ML, but there is no mechanism to verify that the foreign function is implemented correctly. In comparison, VeriFFI allows both reasoning about the higher-level side, since it is just Coq code, and the lower-level side, since VST's separation logic and C program logic is available.

Œuf [Mullen et al. 2018] is another verified compiler project from Coq to C. Œuf can compile a subset of Gallina, with no user-defined types, dependent types, fixpoints, or pattern matching. In comparison, CertiCoq can compile all of Gallina. Œuf's compiler correctness theorem allows the shim (wrapper code in C that executes the compiled Coq program) to be verified using VST, but it does not have a story about how Coq programs can call C programs, or regarding the specified/verified attachment of a garbage collector.

13.2 Other compilers and FFI systems

Foreign function interfaces achieve interoperability by having one language mimic the calling conventions of the other language [Matthews 2008]. For FFI systems where the lower-level language is C, having C types exposed to the higher-level language is common. While exposing base C types like `int`, `void`, and pointers suffices for most cases, it is possible to encode more complicated C types such as **`struct`**s and **`union`**s into the higher-level language. Blume [2001] presents an example of this for Standard ML and Yallop et al. [2018] demonstrate a different design for OCaml. In these approaches, the glue code generators catch discrepancies between the types of foreign functions and their higher-level representations. VeriFFI does not expose the C types to Coq, but it would be possible to implement a library that does so, and prove properties about it.

Some FFI systems expose the value representation of the higher-level language to the lower-level language. OCaml's values are represented in C with the `value` type [Leroy 1999], which CertiCoq and VeriFFI reuse. Similarly, Java's JNI [Liang 1999] achieves interoperability by exposing the higher-level language's values to the lower-level language, where the user has access to C types such as `jstring` and `jobject` for Java strings and objects.

Furr and Foster [2005] explore static checks to ensure that foreign functions do not violate type safety in OCaml, and in later work, Java's JNI [Furr and Foster 2006, 2008]. Their work involves automatic inference of higher-level language types from foreign function implementations in C, and therefore is easier than VeriFFI to apply in larger codebases. In comparison, VeriFFI guarantees type safety as a corollary of correctness. In a similar line of work, Tan et al. [2006] add static and dynamic checks to ensure that foreign code does not violate memory safety or Java's type safety.

VeriFFI can be used to implement particular data types more efficiently and bring compiler optimizations on a case-by-case basis to CertiCoq compiled code. For example, Baudon et al. [2023] and Elsman [2024] present a technique called "bit-stealing" to represent algebraic data types using less space, and implement a compiler that uses this technique in all data types. While CertiCoq does not use this technique in its representation of Coq values, it is possible to implement a foreign type that makes this optimization for a particular type, and prove it correct using VeriFFI. One useful example would be an integer type that has one constructor that carries a 63-bit integer and another constructor that carries a big integer. Since constructor payloads differ in their boxities, we do not need boxed constructors and constructor headers to distinguish between the machine and big integers.

14 FUTURE WORK

14.1 State and IO monads, higher-order functions

For Coq's language of pure functions to interoperate with C code that can use state or perform I/O, we have designed and implemented a monadic interface. The C code behaves as a monadic interpreter, which uses the `call` function (section 7) to call back to the Coq client. Our glue code generator generates operational code that works in demonstrations such as a constant-time-access array (implemented as an array in C). We specify each monad by giving a functional model of its operations, and we can prove the correctness of the client code using this model. VeriFFI generates the VST funspecs in its normal way. In future work, we will prove that the C code (for the array monad interpreter) satisfies these funspecs.

14.2 End-to-end soundness

We plan to work with the CertiCoq team to specify and verify the correctness of *open* Coq expressions, that is, programs that call external functions—following the methodology explained in section 12. Based on that—using the VST semantic model of the function specification in terms of the Clight operational semantics—it should be straightforward to build a machine-checked proof of soundness for VeriFFI.

14.3 Retargeting

The Coq formalizations described in sections 4, 5, 6, and 8 are entirely independent of the target language, so this work could be retargeted to CertiCoq's WebAssembly back end [Meier et al. 2024].

15 CONCLUSION

For a (dependently) typed functional language to interact with a low-level language, at least one language must be taught how to traverse the data structures of the other, and master the calling conventions of the other; and, for verification, to *reason* over the gap. Our glue code generator allows C to traverse (and build) Coq data structures; allows C functions to support Coq calling conventions; and allows C functions to be proved correct with respect to Coq functional models. Our program logics on both sides are very rich and expressive: Coq (the CiC logic) is a widely used and well-established logic for reasoning about functional programs written in that logic; our system for specifying and verifying across interfaces permits both concrete data types (C traversal/construction of Coq inductive constructors) and abstract data types (C representations unachievable in pure Coq).

Proofs in VST that foreign functions satisfy their funspecs are often long and tedious. If only there were a way to automatically synthesize proofs of C functions from their Coq functional models! But there *is* such a way: it's called CertiCoq, which compiles Coq functions into certified-correct C code. This is the right way to do it for most functions. But for functions (such as `pack`) operating on data types whose representations cannot be described efficiently by Coq constructors, or whose algorithms cannot be efficient enough as functional programs on such constructors, we need a way to write highly tuned C programs by hand and prove them correct using VST's powerful program logic. And that way is VeriFFI, the Verified Foreign Function Interface.

ACKNOWLEDGMENTS

This work was supported by NSF Grant CCF-2005545 and by a Princeton University Presidential Postdoctoral Fellowship.

REFERENCES

Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *CoqPL: Third International Workshop on Coq for Programming Languages*. 2 pages. https://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers*. Cambridge.

Thaïs Baudon, Gabriel Radanne, and Laure Gonnord. 2023. Bit-Stealing Made Legal: Compilation for Custom Memory Representations of Algebraic Data Types. *Proc. ACM Program. Lang.* 7, ICFP, Article 216 (aug 2023), 34 pages. https://doi.org/10.1145/3607858

Matthias Blume. 2001. No-Longer-Foreign: Teaching an ML compiler to speak C "natively". *Electronic Notes in Theoretical Computer Science* 59, 1 (2001), 36–52. https://doi.org/10.1016/S1571-0661(05)80452-9 BABEL'01, First International Workshop on Multi-Language Infrastructure and Interoperability.

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reason.* 61, 1-4 (June 2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

Louis Cheung, Liam O'Connor, and Christine Rizkallah. 2022. Overcoming Restraint: Composing Verification of Foreign Functions with Cogent. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Philadelphia, PA, USA) *(CPP 2022)*. Association for Computing Machinery, New York, NY, USA, 13–26. https://doi.org/10.1145/3497775.3503686

Martin Elsman. 2024. Double-Ended Bit-Stealing for Algebraic Data Types. ICFP (2024).

Michael Furr and Jeffrey S. Foster. 2005. Checking type safety of foreign function calls. (2005), 62–72. https://doi.org/10.1145/1065010.1065019

Michael Furr and Jeffrey S Foster. 2006. Polymorphic type inference for the JNI. In *European Symposium on Programming*. Springer, 309–324. https://doi.org/10.1007/11693024_21

Michael Furr and Jeffrey S. Foster. 2008. Checking type safety of foreign function calls. *ACM Trans. Program. Lang. Syst.* 30, 4, Article 18 (aug 2008), 63 pages. https://doi.org/10.1145/1377492.1377493

Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 247 (oct 2023), 29 pages. https://doi.org/10.1145/3622823

Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. 2017. Verified Characteristic Formulae for CakeML. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 584–610. https://doi.org/10.1007/978-3-662-54434-1_22

Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. *ACM SIGPLAN Notices* 49, 1 (2014), 179–191. https://doi.org/10.1145/2535838.2535841

Xavier Leroy. 1999. CamlIDL user's manual. https://caml.inria.fr/pub/old_caml_site/camlidl/htmlman/

X. Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL'06*. 42–54. http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf

Sheng Liang. 1999. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman.

Jacob Matthews and Robert Bruce Findler. 2007. Operational semantics for multi-language programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 3–10. https://doi.org/10.1145/1190216.1190220

Jacob Burton Matthews. 2008. *The Meaning of Multilanguage Programs*. Ph. D. Dissertation. The University of Chicago. https://web.archive.org/web/20240331001311/https://plt.cs.northwestern.edu/matthews-phd.pdf

Conor McBride. 2010. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming* (Baltimore, Maryland, USA) *(WGP '10)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1863495.1863497

Andrew McCreight, Tim Chevalier, and Andrew Tolmach. 2010. A certified framework for compiling and executing garbage-collected languages. In *ICFP'10: 15th ACM SIGPLAN International Conference on Functional programming*. 273–284. https://doi.org/10.1145/1863543.1863584

Wolfgang Meier, Jean Pichon-Pharabod, and Bas Spitters. 2024. CertiCoq-Wasm: Verified compilation from Coq to WebAssembly. In *CoqPL'24: The Tenth International Workshop on Coq for Programming Languages*. 3 pages. https://popl24.sigplan.org/details/CoqPL-2024-papers/3/CertiCoq-Wasm-Verified-compilation-from-Coq-to-WebAssembly

Erik Meijer. 2014. The Curse of the Excluded Middle: Mostly functional programming does not work. *Queue* 12, 4 (apr 2014), 20–29. https://doi.org/10.1145/2611429.2611829

Yaron Minsky and Anil Madhavapeddy. 2022. *Real World OCaml*. Cambridge University Press. https://doi.org/10.1017/9781009129220

Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. 2018. Œuf: Minimizing the Coq Extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 172–185. https://doi.org/10.1145/3167089

Zoe Paraskevopoulou. 2020. *Verified Optimizations for Functional Languages.* Ph. D. Dissertation. Princeton University. https://www.cs.princeton.edu/techreports/2020/006.pdf

Zoe Paraskevopoulou and Andrew W. Appel. 2019. Closure conversion is safe for space. *Proc. ACM Program. Lang.* 3, ICFP, Article 83 (July 2019), 29 pages. https://doi.org/10.1145/3341687

Zoe Paraskevopoulou and Anvay Grover. 2021. Compiling with continuations, correctly. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 114 (Oct. 2021), 29 pages. https://doi.org/10.1145/3485491

Zoe Paraskevopoulou, John M. Li, and Andrew W. Appel. 2021. Compositional optimizations for CertiCoq. *Proc. ACM Program. Lang.* 5, ICFP, Article 86 (aug 2021), 30 pages. https://doi.org/10.1145/3473591

James T Perconti and Amal Ahmed. 2014. Verifying an open compiler using multi-language semantics. In *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings 23*. Springer, 128–148.

Jacob Prinz, GA Kavvos, and Leonidas Lampropoulos. 2022. Deeper Shallow Embeddings. In *International Conference on Interactive Theorem Proving*. https://doi.org/10.4230/LIPIcs.ITP.2022.28

Olivier Savary Bélanger and Andrew W. Appel. 2017. Shrink Fast Correctly!. In *PPDP'17: International Symposium on Principles and Practice of Declarative Programming*. 49–60. https://doi.org/10.1145/3131851.3131859

Olivier Savary Bélanger, Matthew Z. Weaver, and Andrew W. Appel. 2019. Certified Code Generation from CPS to C. , 13 pages. https://www.cs.princeton.edu/~appel/papers/CPStoC.pdf

Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2019. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 8 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371076

Gang Tan, Andrew W Appel, Srimat Chakradhar, Anand Raghunathan, Srivaths Ravi, and Daniel Wang. 2006. Safe Java native interface. In *IEEE International Symposium on Secure Software Engineering*, Vol. 97. 106–119.

Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. 2019. Certifying Graph-Manipulating C Programs via Localizations within Data Structures. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 171 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360597

Jeremy Yallop, David Sheets, and Anil Madhavapeddy. 2018. A modular foreign function interface. *Science of Computer Programming* 164 (2018), 82–97. https://doi.org/10.1016/j.scico.2017.04.002