Foreign Function Verification Through Metaprogramming

Joomy Korkut

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science

Adviser: Andrew W. Appel

November 2024

© Copyright by Joomy Korkut, 2024.
All rights reserved.

Abstract

CertiCoq is a compiler from Coq to C that is verified in Coq. Thanks to CertiCoq's mechanically checked proof of compiler correctness, users can be sure that programs they write and verify in Coq's rich type system output the same results when compiled to C (and to machine language, via the CompCert verified compiler). However, in practice, 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 user-defined primitive types, mutation, and input/output actions, CertiCoq-compiled code must interact with another language to have those capabilities. Specifically, Coq code must be able to call C code and C code must be able to inspect and generate Coq values and call Coq code. But what happens to the correctness proofs when these two languages interact?

A foreign C function has to be memory-safe, return the expected result, and have only the expected side effect. A specification that expresses these requirements must combine plain Coq, for the functional parts, and a program logic for C (such as the Verified Software Toolchain [VST]), for the verification of C functions. While VST is embedded in Coq, its specification language is quite different. VST proofs about C foreign functions do not directly translate to proofs about Coq primitives either. Bridging this gap by connecting plain Coq and VST theorems requires a system of techniques and methods, many of which are made feasible by using metaprogramming as a general methodological approach. In this dissertation, I describe these techniques and methods. Using these methods, the user can relate foreign functions and types to their functional models, generate VST specifications about the foreign C functions, and write plain Coq proofs about the Coq counterparts of the foreign functions. I also provide examples of Coq programs with primitive types, mutation, and I/O actions, along with specifications and proofs about these programs, to demonstrate VeriFFI, the verified foreign function interface for CertiCoq.

Contents

Abstract 3

1 Introduction 9
- 1.1 Contributions 11

2 Background 13
- 2.1 CertiCoq 13
- 2.2 Inductive Types 15
- 2.3 Metaprogramming Facilities 19
  - 2.3.1 MetaCoq 19
  - 2.3.2 Ltac 27

3 The Foreign Function Interface 30
- 3.1 An Example Use of the Interface 30
- 3.2 Generated Glue Code 35
  - 3.2.1 Constructing Coq Values 35
  - 3.2.2 Discriminating Coq Constructors 37
  - 3.2.3 Extracting Arguments of a Coq Constructor 39
  - 3.2.4 Printing Coq Values 39
  - 3.2.5 Calling Coq Closures 40

4 Representation Predicates 41
- 4.1 Memory Representation of Values 41
  - 4.1.1 Representation in the Heap Graph 44
- 4.2 Definition of Representation Predicates 45
- 4.3 Generation of Representation Predicates 49
  - 4.3.1 De Bruijn Notation Conversion 49
  - 4.3.2 Type Class Resolution 52
  - 4.3.3 Mutually Recursive Type Class Instances 55

5 Reification with Annotations **58**
5.1 Consuming Reified Descriptions . . . 63

6 Constructor Specifications **67**
6.1 Constructor Descriptions . . . 68
6.2 Generation of Constructor Descriptions . . . 71
6.3 Generation of Specifications for Glue Constructors . . . 75

7 Foreign Function Specifications **82**
7.1 Foreign Function Descriptions . . . 82
7.2 Consuming Foreign Function Descriptions . . . 87
7.3 Generation of Foreign Function Descriptions . . . 92
7.4 Generation of Specifications for Foreign Functions . . . 92
7.5 Rewriting Foreign Function Calls . . . 95

8 Examples **100**
8.1 Unsigned Integers . . . 100
8.1.1 The Coq Interface . . . 100
8.1.2 The C Implementation . . . 102
8.1.3 The Functional Model . . . 106
8.1.4 Model Proofs for Foreign Functions . . . 108
8.2 Bytestrings . . . 110
8.2.1 The Coq Interface . . . 111
8.2.2 The C Implementation . . . 112
8.2.3 The Functional Model . . . 116
8.3 Printing Bytestrings . . . 117
8.3.1 The Coq Interface . . . 119
8.3.2 The C Implementation . . . 121
8.3.3 The Functional Model . . . 123
8.3.4 Model Proofs for Foreign Functions . . . 124
8.4 Mutable Arrays . . . 126
8.4.1 The Coq Interface . . . 126
8.4.2 The C Implementation . . . 128
8.4.3 The Functional Model . . . 131
8.4.4 Model Proofs for Foreign Functions . . . 131

9 Related Work 133
9.1 Comparison with the Predecessors of Reified Descriptions 133
9.2 Comparison with Other Work on Safe Interoperation 135
9.3 Comparison with Other Compilers Work 139
9.4 Comparison with Other Effect Systems 140
9.5 Applications 141

10 Conclusion 142

Glossary 145

References 149

Acknowledgments

A Ph.D. can be a solitary endeavor, even more so when it is interrupted by a pandemic. I owe a great debt to the people who have helped remedy that. This page will unfortunately only have an incomplete list of them.

First and foremost, I want to thank my adviser, Andrew W. Appel, for his patience with me, and his foresight and wisdom in research. I gained great insight and an eclectic taste of research from all the times he answered my questions by referring me to different papers he wrote in the 1990s. He also gave me all the time and space I needed to dip my toes in different fields of computer science and quench my curiosity with many side projects, some of which have even led to papers. He provided extensive feedback on countless iterations of this dissertation, for which I am deeply grateful.

I would like to thank the members of my thesis committee, Zachary Kincaid and Matthieu Sozeau, for taking the time to read my dissertation and to give me feedback, and Aarti Gupta and David Walker, for their support and questions.

I would like to thank my colleague, Kathrin Stark, with whom I have discussed many of the ideas presented in this thesis. Without her contributions, the concepts I present in my thesis would be much less refined. I would like to thank Tim Carstens for the time he spent collaborating with us. His contributions to the garbage collector and his use of our project have illuminated our path forward.

I would also like to thank Anastasiya Kravchuk-Kirilyuk, Joseph W. Cutler, and the programming languages group at Princeton, for their academic and social support. I would like to thank Julien Vanegue for encouraging me to wrap up my thesis and also making it easier for me to do so.

I would like to thank my parents, Hülya and Birol Korkut, for providing me with all the resources I needed since I was a child with a burning interest in computers. This dissertation would not have been possible without their sacrifice and their belief in me.

Finally, I would like to thank my wife, Anna Blech, for the love and joy she brings to my life, and for her unwavering support throughout graduate school and beyond. Without her feedback, this dissertation would have been full of unnecessary commas, run-on sentences, and uninspired prose.

This material is based upon work supported by the National Science Foundation under Grant No. CCF-2005545. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

1

Introduction

A bird may love a fish but where would they build a home together?

Tevye, Fiddler on the Roof (play by Joseph Stein, 1964)

If you have ever visited a bilingual household, you may have heard people mix languages. For example, remarks like "*Oy vey*, I can't *schlep* all the way to New Jersey!" would not be unusual in a household where both English and Yiddish are spoken. The main reason for this switch of languages within a single sentence is the desire for higher expressive power. There is no good English translation for *oy vey*, and "have a tedious and exhausting trip" is a mouthful, while *schlep* is easy to say. By using phrases and words from another language, one can articulate ideas more effectively or efficiently!

A similar need arises in programming languages as well. When a single language is not sufficiently expressive or when it lacks certain capabilities, programmers turn to additional languages whose features can be used within the first language. Programs written in this way are referred to as *multilanguage programs* [71]. Another compelling reason to opt for multilanguage programs is code reuse. If you already have a substantial portion of your program written in another language, it is practical to simply call that code from your primary language, use its results, and call it a day. Consequently, almost all programs in the modern software world are multilanguage programs, as it helps save both time and money.

In contrast to how modern software is written, efforts to formally verify software predominantly concentrate on specifying and proving the correctness of programs composed in a single language. The question of how to verify multilanguage programs has attracted great theoretical interest [3, 79, 80, 82], but when I embarked on this dissertation these solutions had not yet been implemented in a real programming language setting. Since then, some projects have addressed this question for a setting where the higher-level language is restricted [31, 51]. In this dissertation, I will address this question for a setting where the higher-level language is a full, dependently typed, purely functional programming language called Coq, and the lower-level language is C. My dissertation does not constitute a complete answer to this question, but it provides a road map for one and introduces mechanisms that automate some steps in the process.

In this dissertation, I present a framework in which CertiCoq [4], a verified compiler from Coq to C, implemented and verified in Coq, can be used to implement a foreign function interface that allows verification of foreign functions. I will explain how we generate C glue code about Coq inductive types and functions, how we generate separation logic specifications about the glue code, how to use the generated glue code in our foreign functions, how we generate separation logic specifications about the foreign functions in relation to their functional model, and finally how we generate axioms that can rewrite propositions about opaque foreign functions. All the separation logic specifications and proofs in this dissertation are written in Coq, using the Verified Software Toolchain (VST) [11].

This dissertation, however, does not provide a mechanized proof that the axioms we generate about foreign functions are justified by the soundness of the VST system. This thesis also does not attempt to provide an overall proof of correctness for code generation that in-

cludes foreign functions [96]. These points are left as future work, and we believe these are achievable goals.

While the methods we discuss in this dissertation are described for a foreign function interface to C, the methods themselves are language-agnostic. One can apply the same methods to build a foreign function interface on top of a CertiCoq backend for a different language. One such backend already exists for WebAssembly [73], and others may be implemented in the future. Our glue code generation, constructor and foreign function description generation, and specification generation can be used for a future foreign function interface project for these backends.

1.1 Contributions

From a more technical standpoint, the main contributions of this dissertation consist of:

1. a foreign function interface that allows dependently typed foreign functions (Chapter 3)

2. type-specific representation predicates of Coq values for the CertiCoq compiler (Section 4.2)

3. techniques in MetaCoq that allow generation of representation predicates (Section 4.3)

4. a novel way to reify Coq types, which is a hybrid of deep and shallow embedding of Coq types, augmented by annotations of each component (Chapter 5)

5. a method for generating VST function specifications from constructor descriptions and foreign function descriptions (Chapter 6 and Chapter 7)

6. a method for generating rewrite principles about the foreign functions based on their functional models (Section 7.5)

7. examples of foreign functions and foreign types for Coq, including effectful programs, expressed through monads (Chapter 8).

If you want to compile, try, or use the code described in this dissertation, everything is available online. The verified compiler from Coq to C discussed in this thesis, CertiCoq, is available at `https://github.com/CertiCoq/certicoq`. The proof library about its foreign function interface, VeriFFI, is available at `https://github.com/CertiCoq/VeriFFI`.

2

Background

Here, we seek equipment to tame this gorgon's head with reflection.

CHAPMAN ET AL. [25]

2.1 CertiCoq

Traditionally, a compiler is a program that reads code in the *source language* from a file, translates it to the *target language*, and writes the resulting code to a file. The language the compiler itself is implemented in is called the *host language*. The host and source languages are often different languages. For example, the Elm and PureScript compilers are implemented in Haskell. For some compilers, the host and the source languages are the same, and they can compile themselves, these are called *self-hosting compilers*. The Glasgow Haskell Compiler [87, 68], the Rust compiler [69], and more recently the Idris 2 compiler [21] are examples of such compilers.

Let us take a look at how CertiCoq fits into this general picture. Colloquially CertiCoq is called "a compiler for Coq in Coq" [4], yet the input method and the pipeline of CertiCoq differ from traditional compilers. As opposed to most compilers, CertiCoq is not an independent program; it was implemented as a Coq plugin. It does not take a file as an input; it takes a definition name and runs as a command in a Coq session:

```
Definition list_sum : nat :=
  List.fold_left (fun x y => x + y) (List.repeat 1 100) 0.
```

```
list_sum is defined.
```

```
CertiCoq Compile list_sum.
```

```
list_sum.c is generated.
```

Notice how the user first creates a Coq definition, and then a special Vernacular command takes the name of this definition and compiles it. Can we call this compiling Coq? What does it mean to compile Coq anyway?

The part of Coq that we will consider consists of three languages: **Gallina**, the term language based on the Calculus of Inductive Constructions [36], **Ltac**, the domain-specific language for proofs and decision procedures, and **Vernacular**, the collection of commands through which we can send queries and requests to the Coq system. These languages can even appear in the same definition. Here is an example that creates the natural number 2, accompanied with a proof that 2 is less than 5:

```
Definition less_than_5 : sig (fun (n : nat) => n < 5).
Proof. exists 2. auto. Defined.
```

Here, `Definition` is a Vernacular command for making a new definition. The type of this definition is a function application term in Gallina, that states that the definition is a sigma type (i.e. a dependent pair) containing a `nat` and a proposition that `nat` is less than `5`. In the next line, that is followed by the `Proof` Vernacular command, which signals that the definition will be written in Ltac. Indeed, there are two Ltac tactics, the first one of which provides the witness of existence, namely `2`. The second one automatically satisfies the goal of type

`2 < 5`. When the Vernacular command **`Defined`** is run, the Ltac tactics construct a Gallina term that satisfies this type. The goal here is simple enough that the **`auto`** tactic suffices. For more complicated proofs, we would either have to use more tactics or use more complicated decision procedures. In order to see the proof term Ltac generates, we can ask Coq to print the definition:

```
Print less_than_5.
```

```
less_than_5 =
    exist (fun n : nat => n < 5) 2 (le_S 3 4 (le_S 3 3 (le_n 3)) : 2 < 5)
    : { n : nat | n < 5 }
```

This term could have been written by hand, but it would take longer, while here running the **`auto`** tactic generates it automatically, and it still works even if the numbers change. In other words, Ltac scripts are interpreted at compile time to generate Gallina terms. We will go deeper into the capabilities of Ltac in Subsection 2.3.2.

With this in mind, it would be more accurate to say CertiCoq compiles parsed and type-checked Gallina terms into C.

2.2 Inductive Types

Inductive types are a mechanism to define custom data types in Coq. They are a generalization of algebraic data types you find in mainstream typed functional programming languages like Haskell and ML.[1]

[1]Not to be confused with *generalized algebraic data types* [30] (GADTs), another generalization of algebraic data types influenced by inductive types, but a less expressive one. There is much larger literature on adding mechanisms to these languages to approximate inductive types, but that is not the topic of this dissertation.

Most values that we will inspect or generate in Coq will be values of inductive types. Their generality requires us to be precise about our terminology, therefore it will be helpful to revise the terms we use to refer to different parts of an inductive type. Here is the classic example of an inductive type, the vector type, which is a kind of list indexed by its length:

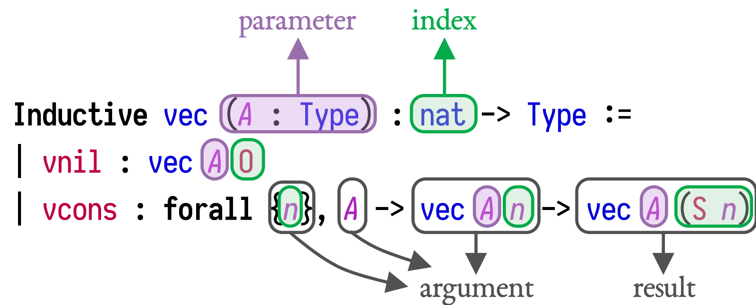


Figure 2.1: The definition of the `vec` type in Coq.

The code above defines a type `vec` with two *data constructors*: `vnil` and `vcons`. While `vec` is the name of the inductive type, it doesn't have the type `Type`, unlike a type like `nat`. This is because `vec` itself needs to take arguments. In other words, `vec` is a *type constructor*. When the word "constructor" is used alone, it refers to data constructors. The term "data constructor" is only used in contrast to type constructors.

The first of `vec`'s arguments is an example of a *parameter*, and the second is an example of an *index* [40]. Both parameters and indices are arguments to the return type of the inductive type. Parameters are written before the colon in the type signature of the inductive type being defined, while indices come after. Parameters must remain consistent across all constructors' return types, while indices can vary. It is worth noting that all the return types in `vec` have `A` as the first argument, but the second argument varies between `0` or `S n`. Thanks to this variation, the `vec` type can be indexed with different list lengths.

Notice that `vnil` takes no arguments, whereas `vcons` takes three. The first argument that `vcons` takes is named *n*, while the other two arguments remain unnamed. This argument is named because the return type depends on it, whereas no type depends on the other two arguments. The type of this argument, `nat`, can be omitted here since the type checker can infer it based on the type of `vec`.

Implicit and Explicit Arguments The curly braces around *n*, the first argument of `vcons`, tell the Coq type checker that this argument is *implicit*, which means this argument will be skipped in a normal application of `vcons`, and the Coq type checker will try to infer it depending on the context.

```
Check vcons.
```

```
vcons
     : ?A -> vec ?A ?n -> vec ?A (S ?n)
where
?A : [ |- Type]
?n : [ |- nat]
```

By adding `@` in front of a function or inductive type name, we can turn all implicit arguments in the type into explicit arguments:

```
Check @vcons.
```

```
@vcons
     : forall (A : Type) (n : nat), A -> vec A n -> vec A (S n)
```

Inspecting Inductive Values Values of inductive types can be inspected by **match** expressions. The programmer can write functions that contain **match** expressions (and ones that do not) using the **fun** keyword in Gallina or the **Definition** keyword in Vernacular, as we have seen in the `list_sum` example. Recursive functions, however, need to be defined using the **fix** keyword in Gallina or the **Fixpoint** keyword in Vernacular:

```
Fixpoint vlength {A : Type} {n : nat} (xs : vec A n) {struct xs} : nat :=
  match xs with
  | vnil => 0
  | vcons x xs' => S (vlength xs')
  end.

Definition vmap :=
  fix vmap {A B : Type} {n : nat}
           (f : A -> B) (xs : vec A n) {struct xs} : vec B n :=
    match xs with
    | vnil => vnil
    | vcons x xs' => vcons (f x) (vmap f xs')
    end.
```

Here, we define two recursive functions. The first function, `vlength` computes the length of a `vec`. The second function, `vmap` transforms all the elements of a `vec` by applying the higher-order function *f*.

In these examples, `vlength` is defined using **Fixpoint**, and `vmap` is defined using **fix**. In practice, top-level recursive functions are usually defined using **Fixpoint**. The **fix** syntax is usually reserved for when we have to write or generate Gallina terms, or when we have to define recursive auxiliary functions within other functions.

2.3 Metaprogramming Facilities

A simple definition of metaprogramming is the generation or analysis of programs by other programs. By this broad definition, C macros [104] are metaprogramming, JavaScript's `eval` is metaprogramming, even replacing text in code before compilation counts as metaprogramming. Metaprogramming comes in many different shapes and colors.

In this thesis, we will use MetaCoq, a homogeneous generative metaprogramming [19] system for Coq, and Ltac, a tactic language for Coq proofs. We will also later develop our own metaprogramming abstractions that are based on these two systems.

2.3.1 MetaCoq

MetaCoq is a project formalizing Coq in Coq, and it comes with a quoting mechanism for Coq [5, 103]. It provides primitives in Coq that can use and manipulate Gallina terms and types based on a deeply embedded description of Gallina terms, which can also be viewed as abstract syntax trees of Gallina terms within Gallina.

Here we see an example of a *quotation*, i.e. converting an expression into a deeply embedded description, i.e. the `term` type in MetaCoq.

```
MetaCoq Run (tmQuote true >>= tmPrint).
```

```
tConstruct
  {| inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"], "bool")
   ; inductive_ind := 0 |} 0 [] : term
```

The description of the term `true` above tells us that `true` is the zeroth constructor of the type `bool`, which is located in the `Coq.Init.Datatypes` module.

For top-level quotations, we often use the `<% … %>` syntax, defined as a simple Coq notation using the primitive tactic from MetaCoq:

```
Check <% andb %>.
```

```
tConst (MPfile ["Datatypes"; "Init"; "Coq"], "andb") []
  : term
```

The description of the term `andb` tells us that `andb` is a constant definition that is located in the `Coq.Init.Datatypes` module.

Here is an example of an *unquotation* (also called *antiquotation* or *splice* in the literature), converting a deeply embedded description into an expression. We have the `term` description of the variable referring to the logical "and" operator for Boolean values. By unquoting it, we can obtain the actual `andb` function.

```
MetaCoq Run (tmUnquoteTyped _
               (tConst (MPfile ["Datatypes"; "Init"; "Coq"], "andb") [])
             >>= tmPrint).
```

```
andb
```

For top-level unquotations, we often use the `~( … )` syntax, defined as a simple Coq notation using the primitive tactic from MetaCoq.

This is quite similar to systems like untyped[2] Template Haskell (TH) [99]. Both systems have a deeply embedded description type for expressions (`Exp` for TH and `term` for MetaCoq),

[2] A template metaprogramming system makes *code fragments* first-class values. An *untyped* template metaprogramming system would give these code fragments the same type, such as the `term` type in MetaCoq or the `Exp` type in Template Haskell. A *typed* template metaprogramming system would give code fragments types based on the code in the fragment, such as `.<1+2>. : int code` in MetaML [111].

a monadic interface for primitive actions (`Q` for TH and `TemplateMonad` for MetaCoq), and a language primitive to run them (`$(...)` splicing in TH and the **`MetaCoq Run`** Vernacular command in MetaCoq).

The primary features of MetaCoq we use are

1. generating terms
2. generating definitions and type class instances
3. generating proof obligations
4. inspecting inductive types

and we want to show examples of these in this section.

Generating Terms Let us start with generating terms. The classic example of generative metaprogramming systems is generating a specialized power function [98, 110, 37, 93, 106]. This example is useful for illustrating the capabilities of a typed metaprogramming system, and how well it can handle quasiquoting and scoping. As of the time this thesis is written, a dependently typed MetaML-style metaprogramming system is an active research area and an unsolved problem for a full language [56, 121].

In the most naive form, you would write the power function as such:

```
Fixpoint naive_pow (exponent : nat) (base : nat) : nat :=
  match exponent with
  | 0 => 1
  | S exponent' => base * naive_pow exponent' base
  end.
```

Observe that this function is recursive, but not tail recursive. One way we can use MetaCoq to optimize it to generate a power function specialized to its exponent. Such a function would allow us to have a non-recursive power function for a fixed exponent, such as

`fun x ⇒ x * (x * x)` that takes a number to its 3rd power. Here is a function that produces the deeply embedded description of such a function, given an exponent:

```
Definition make_specialized_pow (exponent : nat) : term :=
  let fix aux (exponent : nat) (base : term) : term :=
    match exponent with
    | 0 => <% 1 %>
    | S exponent' => tApp <% Nat.mul %> [base; aux exponent' base]
    end
  in tLambda {| binder_name := nNamed "n"; binder_relevance := Relevant |}
             <% nat %>
             (aux exponent (tRel 0)).
```

Now we can call the specialized power function generator with a specific exponent, unquote the result, and observe the generated function term:

```
Check ~(make_specialized_pow 3).
```

```
fun n : nat => n * (n * (n * 1)) : nat -> nat
```

Notice that this specialized power function cubes the input nonrecursively, unlike the `naive_pow` function. Therefore, our initial goal has been achieved.

Generating Definitions and Type Class Instances The function term generated by `make_specialized_pow` is not made into a definition yet. Ideally, the programmer would generate a definition specialized for a particular exponent and reuse it. Now, let us generate a definition using MetaCoq.

```
Definition generate_specialized_pow (exponent : nat) : TemplateMonad unit :=
  name <- tmFreshName (append "pow" (string_of_nat exponent)) ;;
  fn <- tmUnquoteTyped (nat -> nat) (make_specialized_pow exponent) ;;
  tmDefinition name fn ;;
  ret tt.
```

This function takes an exponent, such as 3, and generates a function named pow3, defined as the function term generated by make_specialized_pow above:

```
MetaCoq Run (generate_specialized_pow 3).
Compute (pow3 2).
```

```
8 : nat
```

In this thesis, we will use Coq's type classes [102] as a way of organizing generated data, therefore it is useful to see an example use of MetaCoq with type classes.

Type classes, as defined in Haskell literature, usually are indexed by other types, and they are used for overloading a single function name for different types. Type classes in Coq are built upon dependent records. Like inductive types, they can have any kind (not just types) of parameters. Here we can overload a single function name for different nats used as fixed exponents.

```
Class Pow (exponent : nat) : Type :=
  { pow : forall (base : nat), nat }.

Instance Pow2 : Pow 2 := {| pow := ~(make_specialized_pow 2) |}.

Definition special_pow (exponent : nat) `{Pow exponent} (base : nat) : nat :=
  @pow exponent _ base.
```

Here we define a type class that is parametrized by the exponent, which is a nat, and then we write instances of this type class for each fixed exponent we want to specialize for. Then we define the special_pow helper function, which makes the exponent argument explicit, but the type class instance implicit. Now we can call this function to get the square of a number:

```
Compute (special_pow 2 3).
```

```
9 : nat
```

Ideally, we would not have to write the instances of the `Pow` type class by hand, so let us utilize MetaCoq to generate them automatically up to a certain exponent:

```
Definition generate_pow_instance (exponent : nat) : TemplateMonad unit :=
  name <- tmFreshName (append "Pow" (string_of_nat exponent)) ;;
  fn <- tmUnquoteTyped (nat -> nat) (make_specialized_pow exponent) ;;
  @tmDefinition name (Pow exponent) {| pow := fn |} ;;
  mp <- tmCurrentModPath tt ;;
  tmExistingInstance export (ConstRef (mp, name)).

Fixpoint generate_pow_instances (exponent : nat) : TemplateMonad unit :=
  generate_pow_instance exponent ;;
  match exponent with
  | O => ret tt
  | S exponent' => generate_pow_instances exponent'
  end.
```

The first function generates an instance for a particular exponent, while the second one generates instances for all natural numbers less than or equal to the one it starts from. Here is an example use:

```
MetaCoq Run (generate_pow_instances 5).
Compute (special_pow 4 2).
```

```
16 : nat
```

Remember that this will only work if there is a `Pow` instance for the particular exponent we want to use. If there is no such instance, the `Pow` instance will remain an unsolved existential variable.

Generating Proof Obligations   As you can observe in the `make_specialized_pow` function, term generation with MetaCoq can be quite verbose. This is due to MetaCoq's design: MetaCoq reifies terms as they are in the core language of Coq, where all syntactic sugar has already disappeared. In the core language, there are no notations, records have become inductive types, and implicit arguments have become explicit. Therefore all generated code has to be verbose core-language code. This gets especially tricky when we are generating complicated proofs. At times like this, Ltac is a more useful tool for generating proofs. Therefore, we use the feature of MetaCoq that creates obligations.

To set up an example for this feature, let us revise the `Pow` type class and add a field stating that the specialized power function has to return the same result as the naive power function:

```
Class Pow (exponent : nat) : Type :=
  { pow : forall (base : nat), nat
  ; eq_naive : forall (base : nat), pow base = naive_pow exponent base
  }.
```

Now we will have to update the `generate_pow_instance` function to generate the `eq_naive` field as well. In order to avoid generating this proof with the core language terms with MetaCoq, we can discharge this term as an obligation and later use Ltac to satisfy that obligation.

```
Definition generate_pow_instance (exponent : nat) : TemplateMonad unit :=
  name <- tmFreshName (append "Pow" (string_of_nat exponent)) ;;
  fn <- tmUnquoteTyped (nat -> nat) (make_specialized_pow exponent) ;;
  eq_name <- tmFreshName (append "eq" (string_of_nat exponent)) ;;
  lem <- tmLemma eq_name _ ;;
  @tmDefinition name (Pow exponent) {| pow := fn ; eq_naive := lem |} ;;
  mp <- tmCurrentModPath tt ;;
  tmExistingInstance export (ConstRef (mp, name)).

Obligation Tactic := auto.
```

Here we used tmLemma to create the proof obligation of a certain type (left as _ in the code but inferred by the type checker), and later the auto tactic from Ltac solves the proof obligation.

Now we can call generate_pow_instances as usual and generate all the Pow instances we need.

Inspecting Inductive Types The final feature of MetaCoq that we use a lot in this thesis is inspecting an inductive type. Given a term, we can ask MetaCoq to give a detailed description of the inductive types that are used in the term:

```
MetaCoq Run (tmQuoteRec nat >>= tmPrint).
```

```
({| ...
   declarations :=
     [((MPfile ["Datatypes"; "Init"; "Coq"], "nat"),
       InductiveDecl {| ...
         ; ind_params := []
         ; ind_bodies :=
             [{| ind_name := "nat" ; ...
               ; ind_ctors :=
                   [{| cstr_name := "O"
                     ; cstr_args := [] ; cstr_indices := []
                     ; cstr_type := tRel 0 ; cstr_arity := 0 |}
                  ; {| cstr_name := "S"
                     ; cstr_args :=
                         [{| decl_name := {| binder_name := nAnon
                                           ; binder_relevance := Relevant |}
                           ; decl_body := None
                           ; decl_type := tRel 0 |}]
                     ; cstr_indices := []
                     ; cstr_type :=
                         tProd {| binder_name := nAnon
                                ; binder_relevance := Relevant |}
                          (tRel 0) (tRel 1)
                     ; cstr_arity := 1 |}]
               ; ind_projs := [] ; ind_relevance := Relevant |}] |})] |},
  tInd {| inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"], "nat")
        ; inductive_ind := 0 |} [])
```

Here we see the description MetaCoq gives, simplified for presentation. We see a mutually inductive type block, in which we only have one type. We can access the parameters of the block, types, arguments, names, indices of each constructor. This covers almost all the information from MetaCoq that we use in our code generation.

2.3.2 Ltac

Ltac is a domain-specific language for proofs and decision procedures in Coq. Ltac allows the user to manipulate the proof state imperatively, through commands called *tactics*. Tactics change the *proof state*, which includes the goals and the context. In an interactive Coq session, the user can step forward or backward to see exactly how a tactic changes the proof state. Tactics can add to or remove from the context. Tactics can create new goals. Tactics can satisfy goals. Ltac generates a proof term in Gallina when all the goals are satisfied.

Most of the attention in the metaprogramming literature is directed at homogeneous metaprogramming, where the object and meta languages are the same. Template Haskell [99] and MetaML [111] are the most prominent examples of this among typed functional languages. MetaCoq's quoting mechanism is the Coq equivalent of this tradition [5]. Ltac, although usually not classified as a metaprogramming language, deserves to be classified as such, only a heterogeneous one, where the object language is Gallina and the meta language is Ltac [18].

Ltac can inspect Gallina terms and generate new Gallina terms. It operates on the surface syntax of Gallina rather than a deeply embedded representation of Gallina. MetaCoq, on the other hand, operates on the deeply embedded representation of the core language. Here is an example that inspects a term and creates a new term, presented in both Ltac and MetaCoq for comparison:

```
Ltac single_app_tac t :=
  match t with
  | ?f _ => let x := (constr:(f t)) in idtac x
  | _ => fail
  end.

Definition single_app_fn A : Type (a : A) : TemplateMonad unit :=
  t <- tmQuote a ;;
  match t with
  | tApp f [_] => tmPrint (tApp f [t])
  | _ => tmFail ""
  end.
```

Both the Ltac tactic and the MetaCoq function check if a given Gallina term is an application. If it is, then the result is a new term that is the same function applied to the original term, and this result is printed. If the term is not an application, then the tactic or the function fails. This tactic can be used to turn `S O` into `S (S O)`, or to turn `negb true` into `negb (negb true)`.

Ltac allows pattern-matching not only on Gallina terms but also on the context of the proof. It allows checking if a given name, or a value whose type fits a certain pattern, is in the context. Similarly, Ltac can pattern-match on the goal as well:

```
Ltac destructs :=
  match goal with
  | [ x : _ * _ |- _ ] =>
      let p1 := fresh "p" in
      let p2 := fresh "p" in
      destruct x as [p1 p2]; destructs
  | [ |- _ * _ ] => split; destructs
  | _ => idtac
  end.
```

This tactic does two different things: If there are any pairs in the context, it `destruct`s them, i.e. adds the projections of the pair fields to the context (with fresh names) and removes the

original pair from the context, and recurses to handle any nested pairs that may occur in the projections. The second thing the tactic does is to check if the goal is a pair as well, and `split` that goal into two subgoals, one for each field of the pair, and then it recurses to handle any nested pairs. The tactic stops when there are no pairs in the context or the goal.

In the thesis, we will use Ltac's ability to match on goals to pass information from MetaCoq to Ltac. In Section 6.1, we will see that we can create a goal type that is a function call with an unused argument, where we can pattern match on the unused argument and use it in Ltac.

Ltac unfortunately does not let us inspect the constructors of an inductive type. While it is possible to use tactics like `case` or `induction` on a value of a given type, and that can give us some information about the arguments of a constructor, it is not possible to get a general look over all the constructors of an inductive type, since that would require the ability to consider multiple subgoals at the same time, which is not possible. However, Ltac provides the ability to apply a constructor that fits the type of the goal. Using tactics like `constructor` (or `unshelve econstructor` to deal with existential variables), the user can apply a constructor that fits the goal, without having to pattern-match on the goal type, or without having to know what the constructors of the goal type are.

3

The Foreign Function Interface

Since large programs grow from small ones, it is crucial that we develop an arsenal of standard program structures of whose correctness we have become sure—we call them idioms—and learn to combine them into larger structures using organizational techniques of proven value.

Alan J. Perlis [2]

In this chapter, we will showcase a simple use case of CertiCoq's foreign function interface and examine the glue code generated by CertiCoq.

For pedagogical purposes, this chapter discusses implementing in C what one could have also implemented in Coq (though less efficiently), but later chapters describe foreign functions that could not be implemented in Coq.

The Coq code described in this chapter uses Vernacular commands that are defined in the CertiCoq plugin, namely **`CertiCoq Register`**, **`CertiCoq Compile`**, and **`CertiCoq Generate Glue`** commands. In order to be able to run these Vernacular commands, you would have to have a CertiCoq installation on your machine or in your container.

3.1 An Example Use of the Interface

Suppose we want to have a hand-optimized implementation of a Coq function. Take the `div2` function from the Coq standard library. This function divides a natural number by 2.

```
Fixpoint div2 (n : nat) :=
  match n with
  | S (S n') => S (div2 n')
  | _ => 0
  end.
```

Notice that div2 is not tail recursive, and it allocates memory for the constructor S before every recursive call. Hence this function is not memory and stack-space efficient. It is possible to write a more efficient version of this function with tail recursion and no memory allocations:

```
Definition better_div2 (n : nat) : nat :=
  let fix aux (n m : nat) : nat :=
    match n , m with
    | S n', S (S m') => aux n' m'
    | S n', 1 => n'
    | _, _ => n
    end
  in aux n n.
```

To achieve optimal performance, CertiCoq compiles this function into a tail-recursive C function. This C function can then be further optimized into a loop by smart C compilers like CompCert, GCC, or Clang at higher optimization levels. In fact, both GCC and Clang can generate code that is comparable to hand-written C code for this function. However, we will show a different approach to compile more complex and compute-intensive functions, such as cryptographic primitives that could benefit from using C primitives such as arrays and bit-level arithmetic [7]. For simplicity, we will continue to use div2 as a pedagogical example.

To be able to call our function, we have to create an object on the Coq side. We follow Coq's traditional program extraction mechanism here and ask the user to declare the function as an axiom, which will be realized later when we compile our program.

```
Axiom best_div2 : nat -> nat.
```

Once we declare the axiom, we can tell the compiler that this axiom will be realized by a C function, and where that C function lives. This is achieved through the `CertiCoq Register` command:

```
CertiCoq Register [ best_div2 => "best_div2" ] Include [ "prims.h" ].
```

Now we can declare our C function in a C header file:

```
value best_div2(value);
```

CertiCoq represents Coq values in memory in a uniform way, as we will describe in greater detail in Section 4.1. We have the C type `value` that all Coq values, regardless of their types, live in. In other words, `value` is the C type of Coq values represented in C. The arguments and the result of our function, when realized in C, will all have the C type `value`.

Finally, we can implement our function in a C source file:

```
value best_div2(value a) {
  value tortoise = a, hare = a;
  /* All these function calls will be inlined by the compiler, of course. */
  while (get_Coq_Init_Datatypes_nat_tag(hare) == 1) {
    tortoise = get_args(tortoise)[0];
    hare = get_args(hare)[0];
    if (get_Coq_Init_Datatypes_nat_tag(hare) == 1) {
      hare = get_args(hare)[0];
    } else {
      break;
    }
  }
  return tortoise;
}
```

The C implementation of the function accepts a Coq value represented in C as input and returns a Coq value represented in C as output. Fortunately, users of CertiCoq's foreign function interface do not need to have a deep understanding of the internal workings of the `value` type or how the representation works. CertiCoq provides a set of functions, collectively known as *glue code*, that users can use to interact with `value`s, including the functions used in the above function. In the next section (Section 3.2), we will examine the details of all the functions automatically generated by CertiCoq in the glue code.

Now we can write a Coq program that uses `best_div2` and compile it.

```
Definition prog : nat := best_div2 5.
CertiCoq Compile -file "prog" prog.
CertiCoq Generate Glue -file "glue" [ nat ].
```

The **`CertiCoq Compile`** command compiles a program to C but does not print or use the result. The generated program, by default, does not print or use the result either. This differs from other functional languages like Haskell or OCaml. Haskell programs have an effectful program entry point called `main` with the type `IO ()`, which is executed when the program runs. OCaml programs can feature function calls outside of definitions, which run when the program is run, allowing effectful function invocations or an effectful program entry point. CertiCoq programs do not have these, therefore we need to write a C wrapper file to use the result from the Coq code compiled by CertiCoq. This C file serves as the program entry point, one that cannot be written directly in Coq:

```
#include "gc_stack.h"
#include "glue.h"
#include "prog.h"
```

```
int main() {
  struct thread_info *tinfo = make_tinfo();
  value result = body(tinfo);
  print_Coq_Init_Datatypes_nat(result);
  return 0;
}
```

Inside the `main` function, we first create the initial thread information object. Then we call the `body` function, which evaluates the compiled program, namely `prog` in our Coq file. We get the result back and then call a glue code function to print the result. In our case, this program will print `(S (S O))`.

The thread information object is a **`struct`** that contains the necessary information for the CertiCoq runtime to work. This includes the call stack, the location and bounds of the CertiCoq heap, and the location of the next allocatable spot in that heap [96, 97, 77]. We will refer to this object as the *thread info*, often abbreviated as `tinfo` when used as a function argument or a local definition.

Any function that potentially allocates memory in the CertiCoq heap must take the thread info as an argument. Our foreign C function, `best_div2`, only inspects Coq values; it does not create new ones. However, if we were to write a version of it that does allocate memory, we would need to declare that to the CertiCoq compiler when registering the foreign function:

```
CertiCoq Register [ best_div2 => "best_div2" with tinfo ] Include [ "prims.h" ].
```

Then the C function type of our foreign function could include the thread info as well:

```
value best_div2(struct thread_info *tinfo, value);
```

Now that we have reviewed how CertiCoq's foreign function interface is used, let us take a closer look at the glue functions generated by CertiCoq, and how they are implemented.

3.2 Generated Glue Code

The `CertiCoq Generate Glue` command takes a sequence of Coq inductive types and generates corresponding glue functions. For example, if a programmer intends to write foreign functions that interact with Coq `list`s and `nat`s, they would call this command:

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

When this vernacular command is run, the compiler generates a file that contains functions in C that programmers can use when writing foreign functions. Here we will explain every kind of function generated in detail, in separate subsections. The functions we will see below will have some syntactic sugar that your compiler output may not have[3], but they are the same functions otherwise.

Generated C functions that are specialized to a particular Coq type will use the fully qualified name of types. For a function that mentions the `list` type by name, the function name will contain `Coq.Init.Datatypes.list` (but with underscores instead of dots), so that name clashes are avoided.

3.2.1 Constructing Coq Values

VeriFFI automatically generates C functions that construct the values for each constructor in each Coq inductive type requested by the user. For instance, for the `list` type, VeriFFI generates these functions:

[3] This is because when CertiCoq generates C code, it produces Clight code, which is a syntactically restricted subset of C, and then prints it as C code.

```c
value make_Coq_Init_Datatypes_list_nil(void) {
  return (value) 1;
}

value make_Coq_Init_Datatypes_list_cons(value arg0, value arg1, value *argv) {
  argv[0] = (value) 2048;
  argv[1] = arg0;
  argv[2] = arg1;
  return argv + 1;
}

value alloc_make_Coq_Init_Datatypes_list_cons
      (struct thread_info *tinfo, value arg0, value arg1) {
  value *argv = tinfo->alloc;
  argv[0] = (value) 2048;
  argv[1] = arg0;
  argv[2] = arg1;
  tinfo->alloc = tinfo->alloc + 3;
  return argv + 1;
}
```

The generated glue function for a constructor can look differently depending on whether it has boxed or unboxed representation. We provide a more detailed explanation of these representations in Section 4.1. Unboxed constructors, like the `nil` constructor, are represented as integers at runtime. This can be observed in the `make_Coq_Init_Datatypes_list_nil` function. On the other hand, boxed constructors such as the `cons` constructor, 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, known as the *CertiCoq heap*, or outside of it in what we refer to as the *C heap*. For the `cons` constructor, the `make_Coq_Init_Datatypes_list_cons` function creates a value in the C heap, while the `alloc_make_Coq_Init_Datatypes_list_cons` function creates a value in the CertiCoq heap.

When creating a value on the CertiCoq heap with the `alloc_make_Coq_Init_Datatypes_list_cons` function, we have to pass the thread info object so that the glue function knows where to allocate memory.

When creating a value on the C heap with the `make_Coq_Init_Datatypes_list_cons` function, however, we need to pass a pointer to a part of memory, namely *argv*, where the glue function can build a value. The *argv* pointer must have enough available memory for the function to write on. It requires one word for each constructor, and a word right before the pointer to store header information. It is the programmer's responsibility to free this memory later or keep track of what can be freed and when. The CertiCoq garbage collector does not touch this allocated memory. When allocating on the C heap, one must take care of pointers from there into the garbage-collected heap. The CertiCoq garbage collector has mechanisms to manage that, but to avoid such complexity, in this thesis work we have focused mostly on allocation in the CertiCoq heap.

3.2.2 Discriminating Coq Constructors

For each Coq inductive type, VeriFFI generates a separate C function that allows the user to determine the constructor used to create a value of that type. For example, for the `list` type, the function would be:

```
unsigned long long get_Coq_Init_Datatypes_list_tag(value v) {
  if (is_ptr(v)) {
    switch (get_boxed_ordinal(v)) {
      case 0:
        return 1;
    }
  } else {
    switch (get_unboxed_ordinal(v)) {
```

```
      case 0:
        return 0;
    }
  }
  /* In the future, we can add the unreachable() annotation here,
     to improve the C compiler's ability to optimize this function. */
}
```

This function takes a Coq value and returns an integer that matches the word size of your machine, which we call the *tag* of the constructor.

A tag is an index (starting from `0`) denoting the order of the constructor that was used to create the value at hand. When the C representation of a Coq value is passed into this function, it returns `0` for `nil` and `1` for `cons`. For further ease of use, the programmer can define an `enum` type:

```
typedef enum { NIL, CONS } coq_list;
```

The programmer can then treat the result of this function as a value of type `coq_list`. However, CertiCoq does not generate this type automatically.

Internally, each constructor has an *ordinal*, the index of a constructor within the constructors of that inductive type with the same boxity. Counting the boxed and unboxed constructors separately is a trick that allows representing a larger number of constructors in the `value` type. However, it is important to note that ordinals are specific to the compiler's implementation and it is not recommended for FFI users to depend on them while writing C code. Instead, it is advisable to use tags and glue functions.

3.2.3 Extracting Arguments of a Coq Constructor

Given a Coq value of an inductive type, we would like to be able to access its constructor arguments. We have a single C function that works on values of any inductive type, as long as the value is boxed:

```
value *get_args(value v) {
  return (value *) v;
}
```

Calling the `get_args` function on unboxed values, closures, or type values will cause undefined behavior at runtime.

3.2.4 Printing Coq Values

For each Coq inductive type requested by the user, VeriFFI provides a C function that prints values of that type. For instance, for the `list` type, this would be:

```
void print_Coq_Init_Datatypes_list(value v, void print_param_A(value))
{
  unsigned int tag;
  void *args;
  tag = get_Coq_Init_Datatypes_list_tag(v);
  switch (tag) {
    case 0:
      printf(names_of_Coq_Init_Datatypes_list[tag]);
      break;
    case 1:
      args = get_args(v);
      printf(lparen_lit);
      printf(names_of_Coq_Init_Datatypes_list[tag]);
      printf(space_lit);
      print_param_A(get_args(args)[0]);
      printf(space_lit);
```

```
      print_Coq_Init_Datatypes_list(get_args(args)[1], print_param_A);
      printf(rparen_lit);
      break;
  }
}
```

Notice how since `list` is a parameterized type, the printing function takes a function pointer to deal with values of the parameter type.

3.2.5 Calling Coq Closures

The CertiCoq compiler represents Coq functions as closures at runtime, which consist of a function and an environment, but are not directly callable in C. 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);
}
```

When compiling a function definition with CertiCoq, programmers can use the `call` function to pass arguments to it. The `call` function also requires the thread info as an argument because functions within closures need the thread info to perform memory allocation.

4

Representation Predicates

One might say, by way of excuse, "but the language in which I program has the kind of address arithmetic that makes it impossible to know the bounds of an array." Yes, and the man who shot his mother and father threw himself upon the mercy of the court because he was an orphan.

ANDREW W. APPEL [6]

In code compiled from Coq to C by CertiCoq, Coq values are uniformly represented in memory. Foreign C functions called from Coq are expected to take inputs and give outputs in this representation, yet nothing stops them from flouting this expectation. A violation of this representation may cause a runtime error, but there is no way to catch it statically or prove that no such violation exists. In this chapter, I will describe the uniform memory representation of Coq values, I will introduce representation predicates, a mechanism to state the proposition that a Coq value in CertiCoq's heap memory truly represents that value and also respects this uniform representation, unless it is a value of a foreign type with a custom representation. I will also describe the automatic generation of representation predicates.

4.1 MEMORY REPRESENTATION OF VALUES

CertiCoq's memory representation of values is almost identical to OCaml's memory representation of values [67]. This similarity is no coincidence; it was a design choice that is meant to provide compatibility with OCaml programs, to have the option to use code written for

OCaml's backend (such as its garbage collector) [4], and eventually maybe even to use CertiCoq to compile OCaml code.

Since the representation predicates we present in this thesis are so closely tied to the memory representation of values, let us revisit it and understand how CertiCoq represents Coq values in memory.

A Coq value is a single word (32 bits or 64 bits depending on the machine) in memory that is either an unsigned integer or a pointer to another block of words in memory. We fit all Coq values in this format.

Coq has four kinds of values: types, constructors, function closures, and primitive values.

Types and values of sort `Prop` are erased from CertiCoq early in the compiler pipeline [4, 96, 77]. Any function that takes a type argument or a `Prop` value in Coq still takes that argument in the generated C code, but the erased value is computationally irrelevant: It is never inspected and is always represented by the integer `1` in CertiCoq's C runtime.

Constructors have either *boxed* or *unboxed* representation. Constructors that take arguments have boxed representation; they are represented by a pointer that points to a block of words:

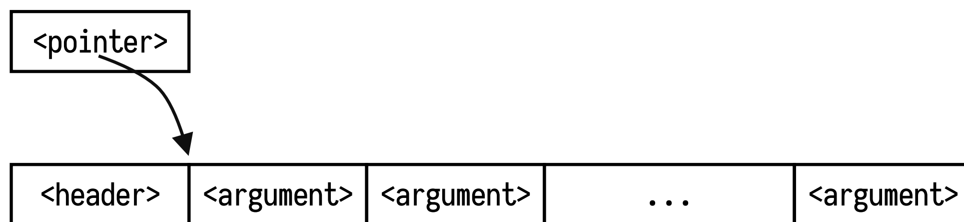


Figure 4.1: A boxed representation of a constructor.

A boxed value has a pointer address, which points to the second word of a block in memory where there is a header and the arguments of the constructor.

Here is what the header for a boxed constructor looks like when the word size is 64:

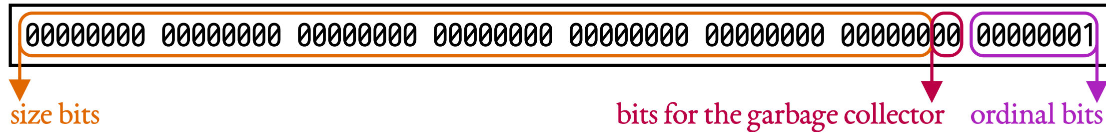


Figure 4.2: The header of a boxed constructor.

The first 22 or 54 bits (depending on the word size of the machine) of the header signify the size of the block (in words) that comes after the header. For a boxed inductive Coq constructor, this essentially means the number of arguments.[4]

Constructors that take no arguments have unboxed representation, so there is no header, and no block, and no pointer: instead there is an odd number.

Here is what an unboxed constructor looks like when the word size is 64:

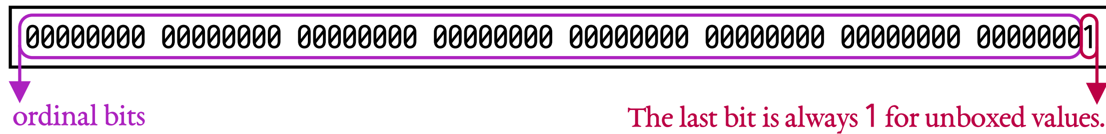


Figure 4.3: An unboxed representation of a constructor.

Function closures are represented as a pointer to a block of memory, where a function pointer and its environment live. The environment is represented as a linked list in runtime. However, the environment is not typeable in Coq, as the values in the environments can be of different types.

[4]For types that have parameters or indices, keep in mind that the parameter will not exist in runtime, but the index always will.

Primitive values have their own custom representations, which we will examine later in this thesis.

4.1.1 Representation in the Heap Graph

Based on the description above of CertiCoq's memory representation of values, one might assume that each Coq value has its own memory block. From a graph theory perspective, this assumption would have made the CertiCoq heap an *out-forest*. Each Coq value would then be represented in the graph as an *out-tree*. This, while elegant, is unfortunately not the case. DAGs in the heap graph can occur naturally: For terms such as `(x, x)` or `[x; x; x]`, the created object would be a DAG, not an out-tree. Even for Coq values that are created as out-trees initially, garbage collection can change the graph in a way that violates the tree properties, specifically because a path in the graph would not be necessarily unique anymore. The graph we arrive at is a *directed acyclic graph* (DAG) but not an out-forest. The heap graph below is an example of this change: The object on the left can turn into the object on the right when the garbage collector runs, if we are using a hash-consing garbage collector [10].

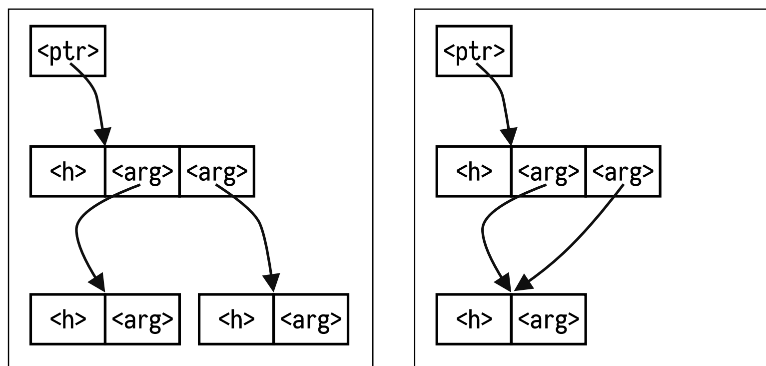


Figure 4.4: Two possible memory representations of the term `(S 0, S 0)`.

Here on both sides, we see the term (`S O`, `S O`). The root of the boxed constructor for pairs. Each field of the pair is a boxed constructor for the successor constructor `S`. In both of the successors, we see an unboxed constructor `O`. However, since the `S O` value occurs twice, the garbage collector can free the memory used for one of them and have both values point to the same memory block so that the representation is shared between two values.

Since memory blocks are shared for different values, we cannot simply depend on the $*$ connective in separation logic. At this point, we rely on previous work and use the CertiGraph library, which helps us reason about graph manipulating programs in VST [118, 117]. The CertiGraph project developed a mechanically verified garbage collector for the CertiCoq runtime. We use both their garbage collector implementation in the CertiCoq runtime and their library of proofs to write our proofs about foreign functions.

4.2 Definition of Representation Predicates

Correctness specifications of constructor glue functions and foreign functions will involve (among other things) the correct interpretation and production of Coq values in the uniform representation we described above. In this chapter, I will describe *representation predicate*s, a mechanism to assert that a Coq value in CertiCoq's heap memory respects this uniform representation, and is equal to a certain Coq value. Using representation predicates, we can express specifications about the correctness and type safety of glue functions and foreign functions.

The memory representation of values is uniform, but only for values of inductive types and closures. Primitive values and values of foreign types are allowed to have custom representations, therefore our system should be able to express custom representations. Therefore,

we use Coq's type classes [102] to define graph predicates; this allows us to overload a single function name with different definitions for uniform representations of different inductive types, and also custom representations of foreign types:

```
Class GraphPredicate (A : Type) : Type :=
  { graph_predicate : graph -> outlier_t -> A -> rep_type -> Prop }.
```

We define a graph predicate as a function that takes the heap graph, the outliers (i.e. the values that are used but live outside of the CertiCoq heap), an actual Coq value, and a heap representation, and produces a `Prop` that posits that the heap representation represents the given Coq value in the heap graph properly.

For most types, we generate and use the common instance that follows the uniform representation. However, the type class mechanism allows the user to write custom instances of the `GraphPredicate` type class, where the user can specify the layout of values of a particular type in the heap graph. As examples for custom representations, we will implement our own primitive integers in Section 8.1, and our own primitive strings in Section 8.2.

During our proofs, we will need some lemmas about our graph predicate function. These lemmas are automatically proven for the uniform representation of inductive types, while the user will have to prove them by hand for custom representations. Because of the intricacy of their generation (explained further in Section 4.3), these lemmas live in a separate type class:

```
Class InGraph (A : Type) : Type :=
  { in_graph_pred : GraphPredicate A
  ; has_v :
      forall (g : graph) (outliers : outlier_t) (n : A) (v : VType),
        graph_predicate g outliers n (repNode v) -> graph_has_v g v
  ; is_monotone :
      forall (g : graph) (outliers : outlier_t) (to : nat) (lb : raw_vertex_block)
             (e : list edge) (n : A) (p : rep_type),
```

```
      add_node_compatible g (new_copied_v g to) e ->
      graph_has_gen g to ->
      graph_predicate g outliers n p ->
      graph_predicate (add_node g to lb e) outliers n p
; outlier_compat :
    forall (g : graph) (outliers : outlier_t) (x : A) (p : GC_Pointer),
      outlier_compatible g outliers ->
      graph_predicate g outliers x (repOut p) ->
      In p outliers
}.
```

This type class contains

- the `GraphPredicate` instance,
- the `has_v` lemma, which states that if a value is represented in the graph as a node, then the vertex of the node is in the graph,
- the `is_monotone` lemma, which states that a Coq value that is represented in memory properly can still be represented in memory if a new compatible node is added to the heap graph,
- and the `outlier_compat` lemma, which states that if a Coq value is represented in the C heap (as an *outlier*), then that pointer will be in the outlier set.

The components of this type class are the work of Stark and Appel. This thesis does not require a deep understanding of these components because this thesis is mostly about writing foreign functions and their specifications, and not so much the proofs of these specifications. Therefore we will not go further in detail here, but the details of these lemmas can lemmas can be found in the tech report by Korkut, Stark, and Appel [59].

Since we now have a type class hierarchy of `GraphPredicate` and `InGraph`, we also define the following helper function that projects the `graph_predicate` field of the superclass `GraphPredicate`, from an instance of the subclass `InGraph`:

```
Definition is_in_graph {A : Type} `{InGraph A}
                       : graph -> outlier_t -> A -> rep_type -> Prop :=
  @graph_predicate A in_graph_pred.
```

Using these predicates, the users are empowered to write their own specifications and proofs for their foreign C functions. These predicates are also used to generate specifications and proofs for CertiCoq-generated glue functions, which can be used as building blocks for the foreign C functions and their correctness proofs.

For example, the `GraphPredicate` instance for the `vec` inductive type looks like this:

```
Instance GraphPredicate_vec
         (A : Type)
         (GraphPredicate_A : GraphPredicate A)
         (n : nat) : GraphPredicate (vec A n) :=
  let fix graph_predicate_vec
          (n : nat)
          (g : graph) (outliers : outlier_t) (x : vec A n) (p : rep_type)
          {struct x} : Prop :=
    match x with
    | vnil => graph_cRep g p (enum 0) []
    | vcons arg0 arg1 arg2 =>
        exists p0 p1 p2 : rep_type,
          @graph_predicate nat GraphPredicate_nat g outliers arg0 p0 /\
          @graph_predicate A GraphPredicate_A g outliers arg1 p1 /\
          graph_predicate_vec arg0 g outliers arg2 p2 /\
          graph_cRep g p (boxed 0 3) [p0; p1; p2]
    end
  in {| graph_predicate := (graph_predicate_vec n) |}.
```

4.3 Generation of Representation Predicates

We generate for the `GraphPredicate` and the `InGraph` type classes using MetaCoq.

The `GraphPredicate` type class only contains one field, the predicate function, and is generated entirely with MetaCoq. We made this design choice because the generated term has many moving parts: The instance has to quantify over the parameters and indices of the inductive type. If those parameters are types, then the instance has to quantify over the `GraphPredicate` instance of them. Parameters stay the same for the types in the same mutually inductive block, while indices can vary, which means the **`fix`** we generate has to quantify over the indices. The `GraphPredicate_vec` example above demonstrates all of these moving parts.

Instances of the `InGraph` type class, on the other hand, are generated using a combination of MetaCoq and Ltac. We use MetaCoq only for generating the types of the instances and invoking Ltac using `tmLemma`. While these instances still have quantified parameters and indices, the definitions inside an `InGraph` instance do not need to use a **`fix`** expression, so they are easier to discharge with Ltac than to build with MetaCoq.

In this section, we will examine the problems we encounter with generating instances for the `GraphPredicate` and `InGraph` type classes, and present techniques to solve these problems.

4.3.1 De Bruijn Notation Conversion

In MetaCoq, variables are represented using a *locally nameless* [15] approach[5]. This means that free variables are represented as names, while bound variables are represented as de Bruijn indices [38]. The `term` type in MetaCoq, which deeply embeds the core terms of Gallina,

[5] There is one caveat: While it is common to remove names from quantifiers in a locally nameless representation, MetaCoq retains them.

includes 4 constructors that are relevant to variables.

```
Inductive term : Type :=
    tRel : nat -> term
  | tVar : ident -> term
  | tEvar : nat -> list term -> term
  | tConst : kername -> Instance.t -> term
  ...
```

`tRel` represents variables referred to by a de Bruijn index, where the lowest number in `tRel` refers to the latest variable binding. `tVar` represents named variables introduced in Coq sections or interactive proofs [103]. However, we will use `tVar` for more than that soon. `tEvar` represents existential variables, which do not have locally nameless representation except in their sub`terms`. `tConst` represents constants (but not inductive types or constructors), i.e. free variables that have a fully qualified name.

Let us consider an example of a quoted term that can help illustrate:

```
MetaCoq Run (tmQuote (fun (A : Type) (a : A) => a) >>= tmPrint).
```

```
tLambda {| binder_name := nNamed "A"; binder_relevance := Relevant |}
   (tSort ...)
   (tLambda {| binder_name := nNamed "a"; binder_relevance := Relevant |}
            (tRel 0)
            (tRel 0))
```

Here we are quoting the polymorphic identity function, which takes one argument `A` that is a type, and one more argument `a` of the type `A`. However, notice that in the result of the quotation, the lambda body does not mention the type `A` by name; it is represented by a de Bruijn index, represented by `tRel 0`. This is observable in the second argument of the inner `tLambda`. The third argument of the inner `tLambda` construction represents the function body

in the original lambda, also happens to be `tRel 0`. The new zeroth de Bruijn index refers to the argument *a*, since the latest quantifier is *a* now.

As we can see from this term, having quoted terms in de Bruijn notation can be confusing when inspecting quoted terms; it is even more confusing and error-prone to generate terms to unquote. This is because the code we want to generate will often have to deal with varying numbers of indices or parameters of inductive types. Committing to generating code in de Bruijn notation will mean we have to keep track of how many of these indices and parameters we have at a given time. Considering the possibility of our generated code introducing new quantifiers and having complicated dependent types that themselves can contain de Bruijn indices, it becomes clear that this is unsustainable.

Our solution is to misuse the `tVar` constructor creatively to have a named representation for our terms. In this representation, all bound variables will be represented as `tVar`s instead of `tRel`s. We can define a function that converts between named and locally nameless representations. Any `term` we receive from `TemplateMonad` primitives or we pass into `TemplateMonad` primitives will have locally nameless representation. But for the remaining internal computation, we can use our named representation.

```
Definition named_term : Type := term.

Definition deBruijn (ctx : list name) (t : named_term) : TemplateMonad term :=
  ...
Definition closed_deBruijn (t : named_term) : TemplateMonad term :=
  deBruijn nil t.

Definition undeBruijn (ctx : list name)  (t : term) : TemplateMonad named_term :=
  ...
Definition closed_undeBruijn (t : term) : TemplateMonad named_term :=
  undeBruijn nil t.
```

The deBruijn and undeBruijn implementations are quite straightforward. They adhere to the de Bruijn notation conversion definitions commonly found in programming language textbooks. We only needed to consider the dependent type system of the Gallina core language and the fact that pattern matching in **match** expressions can alter the number of bindings in the branch body.

4.3.2 Type Class Resolution

MetaCoq does not provide a mechanism for type class resolution under a context. We need to find a way to make unquotable terms quotable (as we will explain) in order to use the primitive resolution mechanism. Our solution for this problem is lambda lifting [54].

As seen in the GraphPredicate_vec instance above, we need to call graph_predicate for the argument types of constructors. Given that graph_predicate is a method of the GraphPredicate type class, we have to make sure MetaCoq can figure out the type class instance to be used in the call. The initial solution most MetaCoq users would go for is to leave the instance argument as a hole, and for MetaCoq to figure out how to fill the hole during unquotation. Unfortunately, this solution is currently not reliable enough, therefore we will have to infer these type class instances manually. MetaCoq only provides the following mechanism for type class resolution:

```
Check tmInferInstance.
```

```
tmInferInstance
     : option reductionStrategy ->
       forall A : Type, TemplateMonad (option_instance A)
```

Observe that this TemplateMonad primitive takes a Type, and not a term. This is easy to call with an actual type class constraint:

```
MetaCoq Run (tmInferInstance (Some all)
                             (forall A `{GraphPredicate A} n,
                              GraphPredicate (vec A n))
             >>= tmPrint).
```

```
(my_Some _ GraphPredicate_vec)
```

However, when we inspect an inductive type, we usually work with the reified versions of types and constructors. In order to use the tmInferInstance primitive, we would have to unquote the term into the Coq value it represents. Unfortunately, unquoting can sometimes fail. Many intermediate forms we obtain while we compute results with terms are not suitable to unquoting. Consider the following function, one we actually use in the generation of GraphPredicate instances:

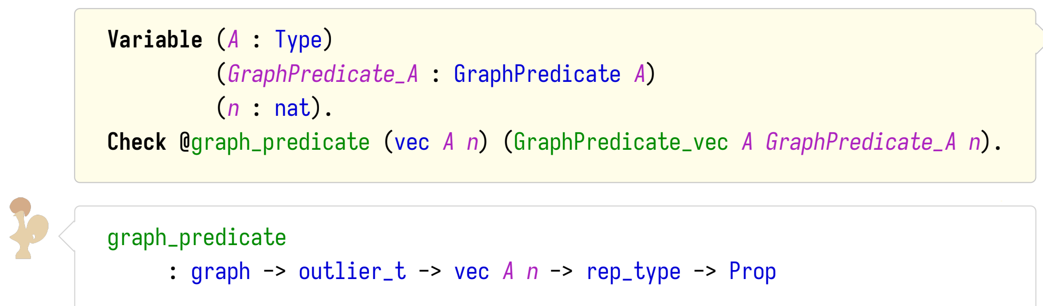
```
Fixpoint strip_pis (t : named_term) : named_term :=
  match t with
  | tProd _ _ rest => strip_pis rest
  | x => x
  end.
```

This function takes a full function or constructor type, and removes the surrounding function type binders (Π-types, hence the function name). If it is given a term representing `forall (A : Type), A`, it will return a term that represents `A`, i.e. the result type of the original function type.

This function generates unquotable terms for two reasons. One reason is that the stripped result type will contain free variables, therefore unquoting will fail. The other reason is that this function generates named_terms, where we have a named representation instead of a de

Bruijn notation, as described in Subsection 4.3.1. The former problem is handled by lambda lifting [54], and the latter problem is handled by de Bruijn notation conversion.

When we are generating the branches of the **match** expression, we want to be able to call `graph_predicate` on the arguments of constructors. However, we have to keep in mind that we are dealing with reified terms representing dependently typed arguments, in named representation. We will not be able to ask MetaCoq for an instance of `GraphPredicate (vec A n)` because since `A` and `n` are free variables, this type class is not even unquotable. To make it quotable, we have to turn it into a closed term by quantifying over the free variables, which we keep as the context, and convert it to de Bruijn notation. Then we can call the `tmInferInstance` primitive, as seen in the interaction above, and obtain the relevant `GraphPredicate` instance. However, the call to `graph_predicate` cannot solely pass `GraphPredicate_vec` as the instance, since the instance quantifies over the free variables even though the `graph_predicate` call does not. Therefore we have apply the context to the instance object:



```
Variable (A : Type)
         (GraphPredicate_A : GraphPredicate A)
         (n : nat).
Check @graph_predicate (vec A n) (GraphPredicate_vec A GraphPredicate_A n).
```

```
graph_predicate
     : graph -> outlier_t -> vec A n -> rep_type -> Prop
```

As seen in the arguments passed to `GraphPredicate_vec` here, applying the context makes the instance object type check.

4.3.3 Mutually Recursive Type Class Instances

Coq allows definitions of mutually inductive data types:

```
Inductive tree (A : Type) : Type :=
| tleaf : tree A
| tnode : A -> forest A -> tree A
with forest (A : Type) : Type :=
| fnil : forest A
| fcons : tree A -> forest A -> forest A.
```

Coq allows definitions of mutually recursive functions as well:

```
Fixpoint tree_size {A : Type} (t : tree A) : nat :=
  match t with
  | tleaf => 0
  | tnode _ f => 1 + forest_size f
  end
with forest_size {A : Type} (f : forest A) : nat :=
  match f with
  | fnil => 0
  | fcons t f => tree_size t + forest_size f
  end.
```

Notice that `Fixpoint` is a vernacular command that can create multiple, mutually recursive definitions.

However, MetaCoq does not provide a `TemplateMonad` primitive to create multiple definitions together. It only allows making a Gallina term into a single definition. In Gallina you cannot define both definitions at once, you have to define each function separately. Gallina's syntax for mutually recursive functions allows locally defining all functions but then exporting a single one of them at the end (with `for` followed by an identifier). Here is what these two functions would look like as stand-alone definitions:

```
Definition tree_size :=
  fix tree_size {A : Type} (t : tree A) : nat :=
    match t with
    | tleaf => 0
    | tnode _ f => 1 + forest_size f
    end
  with forest_size {A : Type} (f : forest A) : nat :=
    match f with
    | fnil => 0
    | fcons t f => tree_size t + forest_size f
    end.
  for tree_size.

Fixpoint forest_size {A : Type} (f : forest A) : nat :=
  match f with
  | fnil => 0
  | fcons t f => tree_size t + forest_size f
  end.
```

Notice that we had to repeat the body of the `forest_size` function. Given that it is a separate definition, any theorems we prove about the outer `forest_size` do not directly apply to the inner `forest_size`; we would have to unfold the definition to make sure that they are the same underneath.

Now, given these restrictions, let us reimagine type class instance generation for mutually inductive types.

When we define a `GraphPredicate` instance for a type in a mutually inductive type block, we want to check if we already have `GraphPredicate` instances for the other types in the block. If there are, then we do not have to generate `graph_predicate` functions in a **`fix`** block for them; we can simply call the existing instances if there are mutually recursive calls. If there are no instances for the types in the same mutually inductive block, then we generate all `graph_predicate` functions for the types in that block, place them in a **`fix`** expression, and return

only one of them. For example, the generated `GraphPredicate` instance for the `tree` type above would look like this:

```
Instance GraphPredicate_tree
         (A : Type)
         (GraphPredicate_A : GraphPredicate A) : GraphPredicate (tree A) :=
  let f :=
    fix graph_predicate_tree
        (g : graph) (outliers : outlier_t) (x : tree A) (p : rep_type)
        {struct x} : Prop :=
      match x with
      | tleaf => graph_cRep g p (enum 0) []
      | tnode arg0 arg1 =>
          exists p0 p1 : rep_type,
            @graph_predicate A GraphPredicate_A g outliers arg0 p0 /\
            graph_predicate_forest g outliers arg1 p1 /\
            graph_cRep g p (boxed 0 2) [p0; p1]
      end
    with graph_predicate_forest
         (g : graph) (outliers : outlier_t) (x : forest A) (p : rep_type)
         {struct x} : Prop :=
      match x with
      | fnil => graph_cRep g p (enum 0) []
      | fcons arg0 arg1 =>
          exists p0 p1 : rep_type,
            graph_predicate_tree g outliers arg0 p0 /\
            graph_predicate_forest g outliers arg1 p1 /\
            graph_cRep g p (boxed 0 2) [p0; p1]
      end
    for graph_predicate_tree
  in {| graph_predicate := f |}.
```

If we want to generate a `GraphPredicate` instance for the `forest` type as well, that can now use the `GraphPredicate_tree` instance.

5

Reification with Annotations

[T]he inherent risk of unintelligibility of reflexivity, has been noted throughout human history in countless paradoxes. For example, in a Greek myth, Narcissus became immobile as a result of staring at his own reflection in the water.

KUMIKO TANAKA-ISHII [112]

Obtaining a specification from a foreign function's type or a constructor's type requires the ability to traverse the components of the type. Such a traversal requires a method to inspect types (or at least constructors' types) as first-class constructs. In this chapter, we will examine why the existing methods fall short of serving our purposes, and we will introduce a new method that solves our problem.

Traversing types is necessary in many different kinds of homogeneous code generation, therefore the generic programming and metaprogramming literature has been preoccupied with it: The Generic Haskell literature achieved code generation at runtime through a representation of Haskell's algebraic data types by deconstructing them into a type that identifies its sums, products, and atomic components [53, 52]. The Template Haskell literature achieved code generation at compile time through a language primitive that supplies information about algebraic data types [99].

MetaCoq follows the Template Haskell tradition. MetaCoq's `tmQuoteRec` command in the `TemplateMonad` provides all the information about any inductive type and its constructors.

However, this data is not designed for quick and easy consumption, as it provides a deeply embedded description of the types, where all the terms are represented in the core language and they are not easily reflectable. The use of `tmQuoteRec` we have seen in Subsection 2.3.1 epitomizes the difficulty of parsing through the output in order to generate code quickly.

There are two main challenges contributing to the difficulty: Firstly, the MetaCoq representation requires explicit handling of names in binders, even after converting to named representation. Secondly, the `term`s obtained from MetaCoq cannot be reflected without their complete contexts. The solution we will present for our original problem of generating specifications for functions' and constructors' types, will have to address these challenges.

Handling names and binders has long been a persistent issue in programming languages. One popular solution that circumvents the issue is *higher-order abstract syntax* [88], which allows a language implementation to depend on the binding mechanism of the host language to implement binding for an object language. From a dependent types point of view, using the binders of the host language allows proper typing of all dependently typed arguments. This convenience makes higher-order abstract syntax a good starting point for our solution.

However, our requirements in this thesis are different from describing terms in any language. Unlike higher-order abstract syntax, we do not need to describe a full language. We only need to describe functions' or constructors' types, where every component is a type. We need to be able to traverse the components of the type, and to consider how a given component should affect the specification for that foreign function or constructor. This requires us to consume a type and produce a specification. While dependent type systems allow us to take a type as an argument, Coq does not allow us to inspect types at runtime.[6] Coq's mech-

[6] Unlike Idris 2, another dependently typed language, which has the *type-case* feature that allows inspection of types at runtime.

anism that allows us to inspect types and obtain different values accordingly is type classes, which are resolved at compile time. Since we are not allowed to look for type class instances at runtime, we should find them at compile time and annotate our representation with the appropriate type class instances.

Following the fundamental theorem of computer science[7], we introduce a new way to represent function types as data, an intermediate representation between MetaCoq and function specifications. We will use metaprogramming to obtain MetaCoq's representation of inductive types and constructors to convert them into our representation, and 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, and also allows us to reason about the specification generation if we ever decide to, though we leave that as future work.

At the heart of this intermediate representation is our description type, which is defined as such:

```
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.
```

- 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 type class instance in the context.

[7] "We can solve any problem by introducing an extra level of indirection", attributed to David J. Wheeler.

- 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 be 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 a type class instance for that type.

Our representation is a combination of *deep embedding* and *shallow embedding* techniques. Deep embedding is a syntactic description, where constructing an abstract syntax tree is the priority, while shallow embedding is a representation where the terms are represented directly in their semantics [49]. The description type 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 allows us to closely resemble the original function or constructor, while clearly identifying each part in the type.

Using the `reified` type, we can now describe types of functions or constructors. While we use it for generating specifications, the `reified` type is general-purpose in essence. The parameter *ann* allows descriptions to be annotated with instances of any type class that fits the type `Type -> Type`.

Before moving on to generating specifications from `reified` descriptions, let us examine a simpler use of this type. Let us define a type class that tells us how to convert values of a given

data type into strings, similar to Haskell's `Show` type class. We can define instances of this type class for some commonly used types, such as `nat`, `bool`, and `list`.

```
Class ToString (A : Type) := { to_string : A -> string }.
Instance ToString_nat : ToString nat := ...
Instance ToString_bool : ToString bool := ...
Instance ToString_list {A : Type} `{ToString A} : ToString (list A) := ...
```

Now, suppose we want to describe the type of this function:

```
length : forall {A : Type}, list A -> nat
```

The function type consists of a type parameter, an argument that follows it, and finally the result type. The `reified` description of it also follows the same order:

```
Definition length_desc : reified ToString :=
  TYPEPARAM (fun (A : Type) {H_A : ToString A} =>
    ARG (list A) (fun (_ : list A) =>
      RES nat)).
```

The annotation type we picked, namely `ToString`, is visible in the type of `length_desc`. In the definition, the `H_A` is visible, while the annotations in the `ARG` and `RES` constructors are implicit arguments, so they are hidden. Since the implicit arguments are type class instances, Coq's type class resolution system should be able to figure out the instance terms. However, we can choose to write the same description with explicit annotations:

```
Definition length_desc : reified ToString :=
  @TYPEPARAM _ (fun (A : Type) {H_A : ToString A} =>
    @ARG _ (list A) (@ToString_list A H_A) (fun (_ : list A) =>
      @RES _ nat ToString_nat)).
```

Now we have an example annotation, which makes it easier to demonstrate the functions that consume reified descriptions. One of these functions is args, which takes a reified description and generates a nested dependent tuple of the types of all type parameters and arguments in the description:

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

The type parameter case adds two fields to the tuple, one for the type, and one for the type class instance. The type class instance has to be there so that we can later call the function k, whose result we can use for a recursive call, which allows the traversal of the reified description to continue. The argument case adds one field to the tuple, the argument. The result (RES) case is the base case, it adds the unit type as a stop to the nested tuple.

Notice that args generates a Type at the end, which means we can have terms that have the type args of some description. For the description of the length function we had above, this term would look as such:

```
Definition length_args_example : args length_desc :=
  (bool; (ToString_bool; ([true; false]; tt))).
```

We should note that this term looks like an example call of the length function, namely @length bool [true; false], which should illustrate why we developed this mechanism: 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.

Let us write one such function. Since our description length_args_example is annotated by the ToString type class, we can design a function that would take the args for a length function as an argument, and build a string that contains all the arguments converted to a string. That function, generalized to any reified description, would look like this:

```
Equations append_descs (r : reified ToString) (xs : args r) : string :=
append_descs (@TYPEPARAM k) (A; (H; rest)) :=
  append_descs (k A H) rest ;
append_descs (@ARG A H k) (a; rest) :=
  to_string a ++ " | " ++ append_descs (k a) rest ;
append_descs (@RES _ _) tt := "".
```

This function uses dependent pattern matching, where the type of the second argument depends on the value of the first argument. While it is possible to write functions with dependent pattern matching in pure Gallina, we present these function definitions using the Equations plugin [101] for easier understanding. We will continue this presentation in other consumers of reified descriptions when necessary, to avoid convoluted Gallina expressions.

We can observe the dependent pattern matching in how the type of the second argument changes: In the TYPEPARAM case, the args object, is a nested dependent pair of A, the type, H_A, the ToString instance, and rest, the rest of the args object. In the ARG case, the args object is a dependent pair of a, which is the argument of type A, and rest, the rest of the args object. In the RES case, the args function arrives at its base case, therefore the args object is tt, the constructor for the unit type. This is a pattern we should expect in functions that take an args object as an argument.

Let us call the append_descs function with a reified description and an args object that fits that description, and observe the result we get:

```
Compute (append_descs length_desc length_args_example).
```

```
= "true :: false :: nil | "
: string
```

Now that we have seen how to write functions that consume `reified` descriptions, such as `args` and `append_descs`, we can write more functions like that. A remarkably useful one is `result`, that traverses the `reified` description of a function or constructor type and returns the result type of that function:

```
Equations result (r : reified ToString) (xs : args r) : Type :=
result (@TYPEPARAM k) (A; (H; rest)) := result (k A H) rest ;
result (@ARG A H k) (a; rest) := result (k a) rest;
result (@RES A H) tt := (A; H).
```

We traverse the `reified` description, by obtaining the rest of the description by calling `k` and recursing. In `RES`, our base case, we return the type that lives in the constructor field `A`. This is a good example of how we reap the benefits of our representation, which is only a valid Coq term because of all the context we preserved in the higher-order abstract syntax.

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 essentially the 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:

```
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. Using `reflect`, we can make sure that the function we have fits the description we were provided.

While it was feasible to derive the curried function type, which closely resembles the original constructor or foreign function type, we deliberately opted for the uncurried function in our implementation of `reflect`. This choice was driven by the necessity of uncurried functions in our function specifications outlined in Chapter 6 and Chapter 7. [8] The other reason for this choice is that the uncurried function type includes the annotation arguments, while the curried function does not. Therefore producing the curried one from the uncurried one makes more sense than producing the uncurried one from the curried one, since we do not have to make up bogus annotations in the former. Although we avoided the latter approach in the general scenario, a specific use case taking this route (traversal of a `reified` description without an `args` object) is presented in Chapter 7.

In Section 6.1 and Section 7.1, we explore detailed examples that showcase reified descriptions and their transformation into concrete Coq values. We leverage `reified` descriptions of Coq constructors and custom primitive functions to create VST specifications for the generated glue code and foreign functions in C.

[8] VST's `WITH` clauses have an easier time with a nested dependent tuple of arguments than many dependent arguments. Also, the language of our foreign function, C, has naturally uncurried functions.

6

Constructor Specifications

A good programmer builds a working vocabulary.

Guy L. Steele Jr. [105]

In order 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.

Functional programmers from the ML and Haskell tradition often start their program development by defining inductive types and data constructors. This approach offers a top-down design strategy, outlining the required data structures, functions, and potential I/O operations. Should a user opt to implement certain functions as foreign functions, these functions will likely need to manipulate constructors of the involved inductive types. To assist with this, a glue code generator is available (see Section 3.2), aiding users in writing their foreign functions in C by offering building blocks for the operations themselves. However, when it comes to proving properties about these foreign functions, users will require a different set of building blocks – ones suitable for proofs.

In this section, we present a framework to generate specifications for glue functions. Further work not covered in this chapter also proves these specifications about glue functions, which allows these proofs to be leveraged as building blocks in larger proofs about foreign functions [59].

6.1 Constructor Descriptions

The glue code generator we presented in Section 3.2 generates glue functions in C that construct Coq values, such as `make_Coq_Init_Datatypes_list_nil` or `alloc_make_Coq_Init_Datatypes_list_cons`.

We need to produce formal specifications of these constructor-types in VST's specification language. We will do that with the assistance of our `reified` semi-deep-embedded description language. As usual, `reified` must be supplied with an appropriate annotation type. For data constructors we will use the following type class that contains all 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 4.2, 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. Having `InGraph` instances for all arguments of a constructor will later allow 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. *Computationally irrelevant* values, such as values of type Type, or all values of kind Prop are erased by CertiCoq's compilation pipeline. When they are arguments to constructors or functions, these values are represented as the unit value[9] as a placeholder. This replacement would be easy to bake into the InGraph type class: The graph_predicate function would simply check if the memory representation is the number 1. While those values are erased, their places are kept and occupied by their placeholders. Some values are entirely erased in the memory representation, such as parameters of inductive types. For those cases, we want to track whether a constructor argument is erased, which is what the second field of the ctor_ann type class does. If an argument to a constructor is not meant to be erased, however, we say the value is present.

Now that we have a ctor_ann type to annotate our reified descriptions with, we can define 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 one more field, ctor_reflected, which is the reflected version of the reified description we just included in the record.

[9]The unit value is represented as the number 1 in memory, which is the tag 0 shifted once to the left, and the last bit set to 1.

Here we can see some example `ctor_desc` values for the `nil` and `cons` constructors of the `list` inductive type:

```
Definition nil_desc : ctor_desc :=
  {| ctor_name := "nil"
   ; ctor_reified :=
       @TYPEPARAM _ (fun (A : Type) (H_A : ctor_ann A) =>
         @RES _ (list A) ({| field_in_graph := InGraph_list A (field_in_graph H_A)
                           ; is_erased := present |}))
   ; ctor_reflected := fun '(A; (_; tt)) => @nil A
   ; ctor_tag := 0
   ; ctor_arity := 0
   |}.

Definition cons_desc : ctor_desc :=
  {| ctor_name := "cons"
   ; ctor_reified :=
       @TYPEPARAM _ (fun (A : Type) (H_A : ctor_ann A) =>
         @ARG _ A ({| field_in_graph := H_A
                    ; is_erased := present |}) (fun (x : A) =>
           @ARG _ (list A) ({| field_in_graph := InGraph_list A (field_in_graph H_A)
                             ; is_erased := present |}) (fun (xs : list A) =>
             @RES _ (list A) ({| field_in_graph := InGraph_list A (field_in_graph H_A)
                               ; is_erased := present |}))))
   ; ctor_reflected := fun '(A; (_; (x; (xs; tt)))) => @cons A x xs
   ; ctor_tag := 1
   ; ctor_arity := 2
   |}.
```

We can also define a type class that allows easy transition from the real Coq constructor for an inductive type, into the `ctor_desc` for that constructor:

```
Class Desc {T : Type} (ctor_val : T) :=
  { desc : ctor_desc }.
```

We can define instances of `Desc` for every constructor we generate descriptions for:

```
Instance Desc_nil : Desc @nil := {| desc := nil_desc |}.
Instance Desc_cons : Desc @cons := {| desc := cons_desc |}.
```

This way, a user of our proof interface would only have to call `desc @nil` or `desc @cons` to reach the descriptions of the `nil` and `cons` constructors.

It might be worth noting that our `Desc` type class does not come with a guarantee that the `reified` description matches the real Coq value. It would, however, be possible to add a new field to the `Desc` type class that checks if the `ctor_val` value is equal to the reflection of the `reified` description. We have not done this for two reasons. The first reason is that this requires careful handling of the curried and uncurried versions of the constructor, and that comes with proof engineering challenges. The second reason is that describing the wrong constructor in the `Desc` instance means the verification of the function specifications will fail later, which makes this guarantee less necessary for our use case.

6.2 Generation of Constructor Descriptions

Generation of constructor descriptions is implemented mostly in MetaCoq, with one part discharged to Ltac as a proof obligation, which is automatically solved by a tactic we provide. There are two interesting parts to generating a `ctor_desc`: generating the `reified` description, for the `ctor_reified` field of the `ctor_desc` record, and generating the reflected version of that reified description, for the `ctor_reflected` field of the `ctor_desc` record.

We achieve the generation of `reified` descriptions of constructors' type through MetaCoq. MetaCoq's `tmQuoteRec` primitive provides all the information needed about the constructors of an inductive type. A traversal of the types of constructors can generate the necessary

TYPEPARAM, ARG, and RES constructors. We will, however, have to generate ctor_ann instances for each of the components. A ctor_ann instance, as we have seen earlier, consists of an InGraph instance, and a field that indicates the erasure of the component in runtime. The InGraph instance can be generated by the method described in Section 4.3, the constructor description generator would only have to invoke that method. The remaining field of the ctor_ann instance, is_erased, can be generated based on the other metadata provided by the tmQuoteRec primitive.

We achieve the generation of a term for the ctor_reified field of a ctor_desc record through a clever Ltac trick, invoked by MetaCoq. We use the MetaCoq primitive tmLemma, which can create a proof obligation of a given type. When we are generating a ctor_desc, we use tmLemma to create a proof obligation of the type of ctor_reflected, which is reflect ctor_reified, where ctor_reified is the value of an earlier field in the record. Then our MetaCoq program calls our Ltac program to prove that obligation, which is much easier than build the proof term directly in MetaCoq.

Let us consider the nil constructor as an example to see what the type of ctor_reified should be for a description of nil:[10]

```
Compute (reflect (ctor_reified nil_desc)).
```

```
= forall (P : {A : Type & {_ : ctor_ann A & unit}}),
    list (projT1 P)
: Type
```

[10]The result of Compute is presented with some syntactic sugar in these examples.

Similarly, let us examine the type of the ctor_reflected field for the cons constructor:

```
Compute (reflect (ctor_reified cons_desc)).
```

```
= forall
    (P : {A : Type & {_ : ctor_ann A & {_ : A & {_ : list A & unit}}}}),
    list (projT1 P)
: Type
```

The implementations of the ctor_reflected fields have to abide by this type. They have to provide an implementation of the original constructor, just in the uncurried type that the type of the field requires. As seen in nil_desc and ctor_desc, the ctor_reflected field in both returns the original constructor at the end. Now, if we are creating a proof obligation of the type reflect ctor_reified, how do we recover the original constructor when creating the reflected version in Ltac?

This is where it helps to remember that Ltac is a metaprogramming facility that can pattern-match on surface-level syntax. Ltac's **match** is more powerful[11] than the **match** expression of Gallina, which can only pattern-match on the value of expressions. Therefore, if we create a proof obligation that contains whatever information we want to pass onto Ltac, but still evaluates to the original proof goal we wanted to have, then we can use Ltac's **match** to recover that information and utilize it in the Ltac script to fill the proof obligation. In other words, we need a version of reflect that takes the extra information we need to pass onto Ltac as an argument, and then ignores it:

```
Definition reflector {cls : Type -> Type} (r : reified cls)
                     {k : Type} (x : k) : Type :=
  reflect r.
```

[11] Ltac's **match** and **eval** can do what Gallina's match does.

Notice that the `cls` and `r` arguments are the same as `reflect`, but this new function has two extra arguments, namely `k` and `x`. The first argument `k` stands for *kind*, a term commonly used to refer to types of types in languages with ML-like type systems. While Coq does not distinguish types and terms as sternly, intuition should help us understand why we need the extra argument `k`. The other argument `x` is meant to be used to pass the original constructor onto Ltac. Since we want to pass the constructors before any application, they will have different types. (Or kinds, according to the intuition we discussed.) Seeing the types of bare constructors should clarify that:

```
Check @nil.
```

```
@nil
     : forall A : Type, list A
```

```
Check @cons.
```

```
@cons
     : forall A : Type, A -> list A -> list A
```

The types of bare constructors would be passed as the `k` argument.

Using `k` and `x`, we can create proof obligations such as `@reflector _ ctor_reified _ @nil` or `@reflector _ ctor_reified _ @cons`, where `ctor_reified` would be a `reified` object held in a `ctor_desc` record. Then our Ltac tactic can pattern-match on this goal and extract the original constructor:

```
Ltac reflecting :=
  match goal with
  | [ |- @reflector _ _ _ ?C ] => hnf; reflecting_aux C
  end.
```

The `reflecting_aux` tactic, which takes the original constructor term as an argument, can then be implemented.

```
Ltac build_ctor C :=
  match goal with
  | [P : {_ : ctor_ann _ & _} |- _ ] =>
    destruct P; build_ctor C
  | [P : {_ : _ & _} |- _ ] =>
    let a := fresh "a" in destruct P as [a P];
    build_ctor constr:(C a)
  | [P : unit |- _ ] => exact C
  end.

Ltac reflecting_aux C :=
  let P := fresh "P" in intro P; simpl in P; build_ctor C.
```

The `reflecting_aux` tactic creates a function that takes an `args` object of the `reified` description and invokes `build_ctor`, which traverses the `args` object by finding the earliest argument to the described constructor and applying that argument to the original constructor as necessary, but skipping the annotations in the `args` object. At the end of the traversal, the entire tactic sequence produces the uncurried version of the original constructor, which satisfies the type of the `ctor_reflected` field.

6.3 Generation of Specifications for Glue Constructors

In Section 3.2, we introduced CertiCoq's glue code generator. The generated C functions are meant to be used in foreign function implementations, and they can construct, inspect, or call Coq values. We want to allow users of CertiCoq's FFI to write foreign functions easily in C, and also write separation logic specifications and proofs about these functions, using

the Verifiable C program logic in the Verified Software Toolchain (VST) [11]. However, the methods we discuss here are independent of the language of the foreign functions.

When we prove properties of foreign functions that use glue code, we do not want to write specifications and proofs for the glue functions themselves over and over again. Ideally, we would have generators for these specifications. Generated specifications and generated proofs would serve as building blocks for foreign function proofs, similar to how generated glue functions serve as building blocks for foreign function implementations.

In this section, we describe the generation of specifications for the glue functions that construct Coq values in CertiCoq heap, such as `alloc_make_Coq_Init_Datatypes_list_cons`. Our generator can generate the specification for such a function for any constructor that has a constructor description, i.e. a `ctor_desc` record, such as `cons_desc`.

Before we start the generation of specifications, we should clarify what we want to specify about the glue function that constructs a Coq value. We already discussed `alloc_make_Coq_Init_Datatypes_list_cons` in detail in Section 3.2, therefore that is a good example to follow here as well:

```
Definition alloc_make_cons_spec (A : Type) `(InGraph A) : funspec :=
  WITH gv : globals, g : graph, x_p : rep_type, xs_p : rep_type,
       x : A, xs : list A, roots : roots_t, sh : share,
       ti : val, outlier : outlier_t, t_info : GCGraph.thread_info
  PRE [[ [ thread_info ; int_or_ptr_type ; int_or_ptr_type ] ]]
    PROP (is_in_graph g outlier x x_p ;
          is_in_graph g outlier xs xs_p ;
          2 < headroom t_info ;
          writable_share sh)
    PARAMS (ti ; rep_type_val g x_p ; rep_type_val g xs_p)
    GLOBALS (gv)
    SEP (full_gc g t_info roots outlier ti sh gv)
```

```
POST [ int_or_ptr_type ]
  EX (p' : rep_type) (g' : graph) (t_info' : GCGraph.thread_info),
    PROP (is_in_graph g' outlier (@cons A x xs) p' ;
          headroom t_info' = headroom t_info - 3 ;
          gc_graph_iso g roots g' roots ;
          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).
```

This dissertation does not include a background section on VST specifications [12], therefore let us write down what this specification says in plain prose:

> Assume these preconditions hold for `alloc_make_Coq_Init_Datatypes_list_cons`:
>
> - This function takes three arguments (**PARAMS**), one for the thread info and two Coq values, whose C representations can be recovered from `x_p` and `xs_p`.
> - The first value `x_p` represents the head of the list, `x : A`, in the CertiCoq heap graph, as stated by `is_in_graph g outlier x x_p` in the propositional part (**PROP**) of the precondition (**PRE**).
> - The second value `xs_p` represents the tail of the list `xs : list A`, in the CertiCoq heap graph, as stated by `is_in_graph g outlier xs xs_p` in the propositional part of the precondition.
> - There are at least two words of memory (`headroom`) left in the thread info object `t_info`, which contains all the runtime information we need.
> - The entire graph `g` is represented as structs and pointers in the C program's memory, as stated by the separation logic part (**SEP**) of the precondition.
>
> Assuming these preconditions, the following postconditions should hold for `alloc_make_Coq_Init_Datatypes_list_cons`:
>
> - This function returns a C representation `p'`, which represents the value `cons x xs` in the CertiCoq heap graph.
> - The number of words available (`headroom`) in the thread info object `t_info'` has decreased by 3, compared to the old thread info object `t_info`.
> - Anything reachable in the heap graph `g` before the function call is reachable after the function call in graph `g'`.

[12] See Appel et al. [11] for a formal description of VST. For an up-to-date reference manual, see Appel et al. [13]. For a introductory book, see Appel et al. [12].

- Pointers into the heap graph, that are local variables of C functions on the stack that called this one, have not been changed in the memory of those frames, states as the equality of the `ti_frames` fields of the thread info objects before and after the function call.
- The new graph `g'` is represented in the C program memory, as stated by the separation logic part of the postcondition.

The specification we present above may look complicated to unfamiliar eyes, but its layers of complexity are already hidden in abstractions. Verifiable C uses VST's separation logic and CompCert's C syntax, both embedded in Coq, with custom syntax through clever uses of Coq's notation system. VeriFFI uses Verifiable C and CertiGraph to reason about the CertiCoq heap graph, and MetaCoq to generate representation predicates about types. These predicates are used through function calls in the specification above.

The next step we have to take is to generalize this specification to any constructor with a constructor description. To achieve this generalization, we have to

1. quantify over all arguments of the constructor, and their C representations,
2. generate the C function type based on the constructor arity,
3. specify in the precondition that all arguments of the constructor are represented by the corresponding C representation, according to our memory representation predicates,
4. specify in the postcondition that the function result is a C representation of the correct Coq value, the constructor we want with the arguments we have.

We solved the first problem with the `args` mechanism, which provides a nested tuple collection of a constructor's arguments. The second problem is an easy problem that only requires us to repeat the C type of Coq values a given number of times. We solved the fourth problem

with the ctor_reflected mechanism, which allows us to recover the original constructor from a constructor description. We only lack a solution for the third problem. To remedy that, we can implement a function that takes a collection of arguments and invokes the memory predicate for each argument.

This function would need to pattern-match on the reified description of a constructor and the args collection that depends on that description, therefore it has to be dependently typed. While it is possible to write functions with dependent pattern matching in pure Gallina, like we did for append_descs or result, we present this function definition using the Equations plugin [101] for easier understanding:

```
Equations in_graphs
          (g : graph) (outliers : outlier_t)
          (r : reified ctor_ann) (xs : args r) (ps : list rep_type) : Prop :=
in_graphs g outliers (@TYPEPARAM k) '(A; (H; xs)) ps :=
  in_graphs g outliers (k A H) xs ps ;
in_graphs g outliers (@ARG _ _ _) _ [] := False ;
in_graphs g outliers (@ARG A H k) '(a; xs) (p :: ps) :=
  match is_erased H with
  | no_placeholder => True
  | _ => field_in_graph g outliers a p /\ in_graphs g outliers (k a) xs ps'
  end ;
in_graphs g outliers (@RES _ _) _ ps := ps = [].
```

Here this function traverses the args object. Type parameters of constructors are not represented in memory, therefore that case only has a recursive call. Arguments are only represented if they are not erased. The list of C representations we start with have to match the arguments we have in the args object, violating this invariant means having False as a precondition in the constructor specification. The result case specifies that there are no more C representations passed as an argument.

Now we have all the mechanisms we need to define the constructor specification for any constructor description:

```
Definition alloc_make_spec_general (c : ctor_desc) : funspec :=
  WITH gv : globals, g : graph, ps : list rep_type,
       xs : args (ctor_reified c), roots : roots_t, sh : share,
       ti : val, outlier : outlier_t, t_info : GCGraph.thread_info
  PRE' (thread_info :: repeat int_or_ptr_type (ctor_arity c))
    PROP (in_graphs g outlier (ctor_reified c) xs ps ;
          Z.of_nat (ctor_arity c) < headroom t_info ;
          writable_share sh)
     (PARAMSx (ti :: map (rep_type_val g) ps)
      (GLOBALSx [ gv ]
        (SEPx [ full_gc g t_info roots outlier ti sh gv ])))
  POST [ int_or_ptr_type ]
    EX (p' : rep_type) (g' : graph) (t_info' : GCGraph.thread_info),
      PROP (let r := result (ctor_reified c) xs in
            @is_in_graph r.1 (@field_in_graph r.1 r.2)
                         g' outlier (ctor_reflected c xs) p' ;
            headroom t_info' = headroom t_info - Z.of_nat (S (ctor_arity c)) ;
            gc_graph_iso g roots g' roots ;
            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).
```

In plain prose, here is what this specification says:

> For any constructor description `c`, assume these preconditions hold for the `alloc_make_...` glue function for this constructor:
>
> - This function takes the number of arguments specified as the arity in `c`, preceded by an extra argument for the thread info, as seen in the `PRE'` clause. The C representations for these arguments are stored in the list `ps`.
> - The C representations stored in `ps` represent the arguments of the constructor stored in the `args` object in the CertiCoq heap graph `g`, as stated by the call to `in_graphs`.
> - There are enough words of memory (`headroom`) left in the thread info object `t_info`, which contains all the runtime information we need. The necessary number of words is one more than the constructor arity.

- The entire graph `g` is represented as structs and pointers in the C program's memory, as stated by the separation logic part (**SEP**) of the precondition.

Assuming these preconditions, the following postconditions should hold for the `alloc_make_...` glue function for this constructor:

- This function returns a C representation `p'`, which represents the actual constructor value from the `ctor_reflected` applied to the arguments in the `args` object. This value is represented in the CertiCoq heap graph according to the memory predicate for the type this constructor returns.
- The number of words available (`headroom`) in the thread info object `t_info'` has decreased by one more than the arity in `c`, compared to the old thread info object `t_info`. The arity is for the arguments stored in memory and the additional one is for the header.
- Anything reachable in the heap graph `g` before the function call is reachable after the function call in graph `g'`.
- Pointers into the heap graph, that are local variables of C functions on the stack that called this one, have not been changed in the memory of those frames, states as the equality of the `ti_frames` fields of the thread info objects before and after the function call.
- The new graph `g'` is represented in the C program memory, as stated by the separation logic part of the postcondition.

In this chapter, we described constructor descriptions, their generation, and generation of specifications for glue functions based on the descriptions. We have proofs for these specifications as well but the proofs are currently not automatically generated. Further proof engineering should make that an achievable goal since the required mechanisms are now in place.

7

Foreign Function Specifications

If he was forced to look at the light itself, wouldn't it hurt his eyes?

Wouldn't he turn away, and run back to the things he could see?

PLATO, ON THE CAVE ALLEGORY [90]

When proving correctness of a Coq program that calls functions implemented in C and proved correct in VST, the VST *function specification* (funspec) must be coherent with an appropriate Coq functional model. In this chapter we show how to generate a coherent VST funspec 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. Coherence of the funspec generation could be proved, in principle, from VST's semantic model and soundness proof, but that is beyond the scope of this thesis.

7.1 Foreign Function Descriptions

Similarly to how we use `reified` descriptions to write down (and eventually generate) descriptions of constructors, we also want to use them to describe foreign functions.

Our foreign function interface allows not only foreign functions that use Coq inductive types, but also user-defined foreign types, and foreign functions that use these foreign types. In fact, a lot of the interesting cases where we might want to use the foreign function interface will involve some foreign types and some functions that construct, modify, or observe values

of these foreign types, and some Coq inductive types.

The `reified` descriptions we introduced in Chapter 5 suit this purpose as well. 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. Then, the only remaining question is, what additional information is necessary for our purposes? To understand this question better, we need to familiarize ourselves with the workflow of writing foreign types and foreign functions.

A user of our system will define their foreign types and foreign functions as axioms in Coq, preferably in a separate module. Here we define a type for 63-bit unsigned integers and some function that use this type, which we will explain in more detail in Section 8.1:

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

These axioms are the foreign types and foreign functions that our client programs will use. In order to reason about these types and functions we defined here, we want to describe the relationship between these axioms and their functional models. Here we define a model type and some model functions for the axioms above:

```
Module FM.
  Definition uint63 : Type := {n : nat | n < (2 ^ 63)}.
  Definition from_nat (n : nat) : uint63 :=
    (n mod (2 ^ 63) ; ...).
  Definition to_nat (x : uint63) : nat :=
    let '(n; _) := x in n.
```

```
  Definition add (x y : uint63) : uint63 :=
    let '(n; _) := x in
    let '(m; _) := y in
    ((n + m) mod (2 ^ 63) ; ...).
End FM.
```

Notice that these are also defined in a separate module, and the names of the functional model definitions match the foreign type and foreign function names. The `C` module will contain Coq axioms for the foreign types, and Coq axioms for foreign functions that may use these foreign types. These functions will be realized later by C functions through the FFI, as we have seen in Chapter 3. The `FM` module, on the other hand, will contain the functional model definitions. These definitions are written solely in Coq, and we can prove properties about these definitions using the usual reasoning methods of Coq.

When we write a `reified` description of the foreign function `C.from_nat`, we want to denote that the types of `C.uint63` and `FM.uint63` are related to each other. The type of `C.uint63` will contain components that are foreign types, the type of `FM.uint63` will have the same type but the foreign type components will be replaced with the functional model counterparts.

A `reified` description annotates every component of a type with an annotation type `ann : Type → Type`. The annotation type is parametrized or indexed by a single type, as opposed to two types at the same time. This still works for our purposes, but we have to consider one of the types the primary one. We consider the functional model type to be the primary type and the foreign function type to be the secondary type in our definitions, but there is no fundamental reason for this choice, only convenience.

We need two more definitions before we define the annotation type for foreign functions. The first definition is a new type class, that allows us to express which model type component and which foreign type component are related:

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

This type class is helpful because we need to define an `InGraph` record for the functional model type, which is represented as a plain Coq type, as opposed to the foreign type, which is represented as a Coq axiom. This is because the `InGraph` record needs to be able to inspect values of this type, and values of an axiomatized type will be uninspectable.

Defining `InGraph` record as an instance, however, creates a different problem: In the case that the type used as a functional model will be used in computation later for a different type, we would have multiple `InGraph` instances for the same type, which breaks type class inference. For example, in Section 8.2, we use the `string` type in Coq as the functional model type for bytestrings. Bytestrings and naive Coq strings will have different memory representations, even though they will both have `InGraph` instances indexed by the same type (up to normalization). In order to get around this problem, we should avoid defining an `InGraph` instance for the foreign case, and use a `ForeignInGraph` instance for such cases. Since the axiomatized foreign type is not reused, this will rectify the type class inference for our system.

There is one missing piece in the larger puzzle of foreign function annotations: an isomorphism abstraction that allows us to express that the model type and the foreign type are isomorphic to each other. We can use the common definition:

```
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
  }.
```

Now we have all the pieces to define the annotation type:

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

We intend this annotation type to be used in two different ways:

1. The first and easy way is for nonforeign types. If we want to use a type that is defined in plain Coq, the primary and secondary type are the same, which means the isomorphism between them is trivial. We define a function `transparent` to make it easier to generate this annotation:

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

2. The second way is for foreign types. In this case, the `primary` type is the functional model type, and the `secondary` type will be the foreign type. We will have an `InGraph` record for the functional model type. This goes against usual type class usage, where there is only one instance expected for a given type. The `InGraph` record has to be parametrized by the functional model instead of the foreign type because the `graph_predicate` function taking the functional model type as an input allows us to specify how different foreign values are represented in memory. (We will see examples of this in Chapter 8.) We define a function `opaque` to make it easier to generate this kind of annotation:

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

Bogus Annotations At the end of Chapter 5, we discussed how the nested higher-order abstract syntax style of our descriptions can require bogus annotations when the annotations cannot be obtained somewhere else. Here, we define a bogus `foreign_ann` annotation `foreign_ann_any`, where the memory representation predicate requires proving `False`, hence is impossible to prove:

```
Definition GraphPredicate_any {A : Type} : GraphPredicate A :=
  {| graph_predicate g outliers x p := False |}.

Definition InGraph_any {A : Type} : InGraph A.
Proof. ... Defined.

Definition foreign_ann_any {A : Type} : foreign_ann A :=
  @transparent A InGraph_any.
```

This annotation is not defined as an instance since we do not want Coq's type class resolution system to select it by default, but it is a useful annotation when we want to traverse `reified` data structures easily, and we will utilize it in the next section.

7.2 Consuming Foreign Function Descriptions

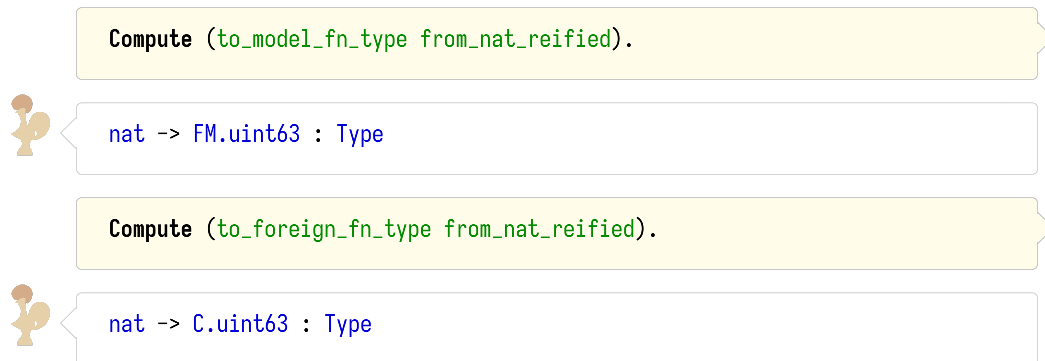
Since the annotation type `foreign_ann` contains both the type of the functional model, which is the primary type, and the type of the foreign function, which is the secondary type, we can traverse the `reified` annotation and reproduce the function types of the foreign function and

its functional model. Suppose we have a reified description for from_nat from earlier in this chapter:

```
Definition from_nat_reified : reified foreign_ann :=
  @ARG _ nat (@transparent _ InGraph_nat) (fun _ =>
   (@RES _ FM.uint63 (@opaque FM.uint63 C.uint63 ForeignInGraph_uint63 Iso_uint63))).
```

Notice that this description prioritizes the functional model type, as the components of it are passed to ARG and RES, while the components of the foreign function type are passed only to the annotation.

Now, we want to have two functions, to_model_fn_type and to_foreign_fn_type, that use the reified description with annotated by foreign_ann, to compute the type of the functional model function and the foreign function, like this:



Notice how the former computes a type that matches FM.from_nat's, and the latter computes a type that matches C.from_nat's.

These functions can be defined as traversals of reified descriptions, even though they need bogus annotations to go all the way down the description. The to_model_fn_type function that computes the functional model type is easier, since the reified description primarily describes the functional model type:

```
Fixpoint to_model_fn_type (r : reified foreign_ann) : Type :=
  match r with
  | TYPEPARAM k =>
      forall (A : Type), to_model_fn_type (k A foreign_ann_any)
  | ARG primary ann k =>
      forall (p : primary), to_model_fn_type (k p)
  | RES primary ann => primary
  end.
```

The `to_foreign_fn_type` function that computes the foreign function type is a bit trickier:

```
Fixpoint to_foreign_fn_type (r : reified foreign_ann) : Type :=
  match r with
  | TYPEPARAM k =>
      forall (A : Type), to_foreign_fn_type (k A foreign_ann_any)
  | ARG primary ann k =>
      forall (m : secondary),
        to_foreign_fn_type (k (@to primary secondary foreign_iso m))
  | RES _ ann => secondary
  end.
```

In the `ARG` case, the `primary` argument is the functional model component type, and `k` expects a value of type `primary`. When we compute the foreign function type, however, we quantify over the `secondary` type. Therefore, we have to call `to` in the isomorphism between the primary and secondary types. Through this call, we obtain a value of the type `primary`, and are able to continue with recursion.

Now that we can compute the type of the foreign function from the `reified` description, we can put the description and the actual functions in the same dependently-typed record:

```
Record fn_desc :=
  { fn_type_reified : reified foreign_ann
  ; foreign_fn : to_foreign_fn_type fn_type_reified
  ; model_fn : reflect fn_type_reified
  ; fn_arity : nat
  ; c_name : string
  }.
```

The function description record contains:

- the reified description of the function, namely fn_type_reified,
- the actual foreign function, namely foreign_fn,
- the actual (but uncurried) foreign function, namely model_fn,
- the arity of the function, namely fn_arity,
- and the name of the foreign function in C, namely c_name.

Currying the Uncurried   Given that we defined to_model_fn_type and to_foreign_fn_type before, we should explain why we did not use to_model_fn_type here, which would have allowed us to pass the actual foreign function into this record. Instead, we used the reflect function to obtain the uncurried version of the functional model type described in fn_type_reified. This was a design decision to make the VST specification generation easier[13]. Also, we can still compute the curried function from the reified description and a function that has the reflected type of that description:

[13] WITH clause of VST funspecs is more pleasant to work with when the parameters in the clause are not dependent on each other. Uncurrying packs all the arguments into one nested tuple (thanks to args), which no other parameter in the WITH clause depends on.

```
Equations curry_model_fn
          (r : reified foreign_ann) (mt : reflect r) : to_model_fn_type r :=
curry_model_fn (TYPEPARAM k) mt :=
  fun (A : Type) => curry_model_fn (k A foreign_ann_any)
                                   (fun xs => mt (A; (foreign_ann_any; xs))) ;
curry_model_fn (ARG A H k) mt :=
  fun (a : A) => curry_model_fn (k a) (fun xs => mt (a; xs)) ;
curry_model_fn (RES A H) mt := mt tt.
```

Understanding this function is not important for this chapter, but it is interesting from a metaprogramming perspective, so we should explain the intuition behind this function: The function traverses the `reified` description to construct a new functional model function, effectively re-currying the uncurried `reflect`ed function `mt`. It recreates the curried version by creating a Coq lambda term for type parameters and arguments, in which the recursive call carries the remaining `reflect`ed value. At every step, we apply the arguments we have seen so far to the `reflect`ed value. However, since we need to recreate the nested tuple of all arguments, we add the seen arguments to the `args` object `xs`, which allows us to create the required nested tuple within the lambda.

Now with this function, it is almost as if we had a field with the functional model type of the foreign function, because we can obtain it with a mere function call in plain Coq:

```
Definition curried_model_fn (d : fn_desc) : to_model_fn_type (fn_type_reified d) :=
  curry_model_fn (fn_type_reified d) (model_fn d).
```

This example once again demonstrates the usefulness of our `reified` description mechanism, since it isolates the metaprogramming to the creation of the description.

7.3 Generation of Foreign Function Descriptions

The mechanisms we outlined regarding the generation of constructor descriptions (in Section 6.2) apply to the generation of foreign function descriptions. This time we do not have to traverse the inductive type descriptions we obtain from MetaCoq, we only have to traverse the functional model and foreign function types. There are no new techniques involved in this generation; we merely traverse two types at the same time, as opposed to the one type we traversed for constructor descriptions in Section 6.2.

7.4 Generation of Specifications for Foreign Functions

Generating VST specifications for foreign functions is very similar to how we generated VST specifications for constructor glue functions in Section 6.3. There we generated specifications using `reified` descriptions with `ctor_ann` annotations; now we will use `reified` descriptions with `foreign_ann` annotations.

To understand what we are generating, it might be helpful to see an example foreign function and how we would write its VST specification by hand. Our running example, `from_nat`, is a good candidate. Its specialized VST specification would look like this:

```
Definition from_nat_spec : ident * funspec :=
  DECLARE _from_nat
  WITH gv : globals, g : graph, n_p : rep_type,
       n : nat, roots : roots_t, sh : share,
       ti : val, outlier : outlier_t, t_info : GCGraph.thread_info
  PRE [[ [ thread_info ; int_or_ptr_type ] ]]
    PROP (is_in_graph g outlier n n_p ;
          writable_share sh)
    PARAMS (ti ; rep_type_val g x_p ; rep_type_val g xs_p)
    GLOBALS (gv)
    SEP (full_gc g t_info roots outlier ti sh gv)
```

```
POST [ int_or_ptr_type ]
  EX (p' : rep_type) (g' : graph) (t_info' : GCGraph.thread_info),
    PROP (is_in_graph g' outlier (FM.from_natn) p' ;
          gc_graph_iso g roots g' roots)
    RETURN (rep_type_val g' p')
    SEP (full_gc g' t_info' roots outlier ti sh gv).
```

Following the example of the previous chapter, let us write down what this specification says in plain prose:

> Assume these preconditions hold for the C function backing `C.from_nat`:
>
> - This function takes two arguments (**PARAMS**), one for the thread info and one for the Coq value, whose C representation can be recovered from `n_p`
> - The first value `n_p` represents the natural number, `n : nat`, in the CertiCoq heap graph, as stated by `is_in_graph g outlier n n_p` in the propositional part (**PROP**) of the precondition (**PRE**).
> - The entire graph `g` is represented as structs and pointers in the C program's memory, as stated by the separation logic part (**SEP**) of the precondition.
>
> Assuming these preconditions, the following postconditions should hold for the C function backing `C.from_nat`:
>
> - This function returns a C representation `p'`, which represents the value `FM.from_nat n` in the CertiCoq heap graph.
> - Anything reachable in the heap graph `g` before the function call is reachable after the function call in graph `g'`.
> - The new graph `g'` is represented in the C program memory, as stated by the separation logic part of the postcondition.

The next step we have to take is to generalize this specification to any foreign function with a `fn_desc`. To achieve this generalization, we have to

1. quantify over all arguments of the foreign function, and their C representations,
2. generate the C function type based on the foreign function arity,

3. specify in the precondition that all arguments of the foreign function are represented by the corresponding C representation, according to our memory representation predicates,

4. specify in the postcondition that the function result is a C representation of the correct Coq value, related to the functional model we want with the arguments we have.

We have solved all of these problems in Section 6.3. Therefore, we follow the same solutions or adapt the solutions described there (such as the annotation type of `in_graphs`) to work for this case. With these solutions, we can now define the foreign function specification for any foreign function description:

```
Definition fn_desc_to_funspec (f : fn_desc) : ident * funspec :=
  DECLARE (ident_of_string (c_name d))
  WITH gv : globals, g : graph, roots : GCGraph.roots_t, sh : share,
       xs : args (fn_type_reified d), ps : list rep_type, ti : val,
       outlier : GCGraph.outlier_t, t_info : GCGraph.thread_info
   PRE' (thread_info :: repeat int_or_ptr_type (fn_arity d))
     PROP (writable_share sh ;
           in_graphs g outlier (fn_type_reified d) xs ps)
     (PARAMSx (ti :: map (rep_type_val g) ps)
       (GLOBALSx [ gv ]
         (SEPx [ full_gc g t_info roots outlier ti sh gv ; library.mem_mgr gv ] )))
   POST [ int_or_ptr_type ]
     EX (p' : rep_type) (g' : graph)
        (roots': GCGraph.roots_t) (t_info' : GCGraph.thread_info),
       PROP (let r := result (fn_type_reified d) xs in
             @is_in_graph r.1 (@foreign_in_graph r.1 r.2)
                          g' outlier (model_fn d xs) p' ;
             gc_graph_iso g roots g' roots')
       RETURN (rep_type_val g' p')
       SEP (full_gc g' t_info' roots' outlier ti sh gv ; library.mem_mgr gv).
```

Following the previous examples, let us explain what this specification says in plain prose:

> Assume these preconditions hold for the foreign function described by `d`:
>
> - This function takes the number of arguments specified as the arity in `d`, preceded by an extra argument for the thread info, as seen in the `PRE'` clause. The C representations for these arguments are stored in the list `ps`.
> - The C representations stored in `ps` represent the arguments to the foreign function stored in the `args` object, either in the CertiCoq heap graph `g` or the outlier set, as stated by the call to `in_graphs`.
> - The entire graph `g` is represented as structs and pointers in the C program's memory, as stated by the separation logic part (`SEP`) of the precondition.
>
> Assuming these preconditions, the following postconditions should hold for the foreign function described by `d`:
>
> - This function returns a C representation `p'`, which represents the return value of the functional model when it is called with all the arguments. This representation is according to the possible foreign type representation of the return type.
> - Anything reachable in the heap graph `g` before the function call is reachable after the function call in graph `g'`.
> - The new graph `g'` is represented in the C program memory, as stated by the separation logic part of the postcondition.

The proofs of these specifications are the work of Stark and Appel and therefore are out of the scope of this dissertation. They are covered in the tech report by Korkut, Stark, and Appel [59].

7.5 Rewriting Foreign Function Calls

In Coq, proofs about programs are often written by unfolding definitions and reducing the resulting terms. However, our FFI workflow requires users to declare all foreign types and foreign functions as axioms. Since axioms cannot be unfolded, writing proofs about a program that includes references to these axioms becomes very difficult.

Although these axioms are realized when the compiled program is linked to the foreign functions on the C side, proofs about this program are checked before compilation and therefore are unavailable at this stage. Without a solution to this problem, users of our FFI cannot prove properties about foreign types and foreign functions.

Our solution to this problem is to introduce a mechanism in which occurrences of these foreign types and foreign functions can be rewritten into their functional models. Since the annotation type `foreign_ann` includes a field that defines and proves an `Isomorphism` between the components of the foreign function type and the functional model type, we can use this isomorphism to generate an equality between these two types. We will then assume these equalities and use them to rewrite the foreign parts into their familiar functional models.[14]

Now let us define a function that relates the foreign function and its functional model:

We define a function `model_spec_aux` that traverses the `reified` description `r`, the foreign function type `pt`, and the functional model type `mt` at the same time, and generates a type that expresses the equality we described earlier:

```
Equations model_spec_aux
          (r : reified foreign_ann)
          (pt : to_foreign_fn_type r)
          (mt : to_model_fn_type r) : Prop :=
model_spec_aux (@TYPEPARAM k) pt mt :=
  forall (A : Type), model_spec_aux (k A foreign_ann_any) (pt A) (mt A) ;
model_spec_aux (@ARG primary ann k) pt mt :=
  forall (x : secondary),
    model_spec_aux (k (to x)) (pt x) (mt (to x)) ;
model_spec_aux (@RES primary ann) pt mt :=
  pt = (@from _ _ foreign_iso mt).
```

[14]For the sake of comparison, the functional model is like the shadow of the real objects outside of the cave, hence the epigraph from Plato at the beginning of this chapter.

For easier use, we define another function `model_spec` that uses a `fn_desc` to generate the same type:

```
Definition model_spec (d : fn_desc) : Prop :=
  model_spec_aux (fn_type_reified d) (foreign_fn d) (curried_model_fn d).
```

This function generates a type that quantifies over all the arguments of the foreign function, and asserts that these two functions give the same result for the same inputs, up to the isomorphisms in each annotation.

In order to demonstrate exactly what `model_spec` generates, let us extend the examples we used earlier in this chapter. Here are the foreign function descriptions for those examples:

```
Definition from_nat_desc : fn_desc :=
  {| fn_type_reified :=
      @ARG _ nat transparent (fun _ =>
        @RES _ FM.uint63 opaque)
   ; foreign_fn := C.from_nat
   ; model_fn := fun '(x; tt) => FM.from_nat x
   ; fn_arity := 1
   ; c_name := "int63_from_nat"
  |}.

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

```
Definition add_desc : fn_desc :=
  {| fn_type_reified :=
      @ARG _ FM.uint63 opaque (fun _ =>
        @ARG _ FM.uint63 opaque (fun _ =>
          @RES _ FM.uint63 opaque))
   ; foreign_fn := C.add
   ; model_fn := fun '(x; (y; tt)) => FM.add x y
   ; fn_arity := 2
   ; c_name := "int63_add"
   |}.
```

The results of calls to `model_spec` with those descriptions will help illustrate how `model_spec` can be used:

```
Eval cbn in (model_spec from_nat_desc).
```

```
forall x : nat,
  C.from_nat x = to (FM.from_nat x)
  : Prop
```

```
Eval cbn in (model_spec to_nat_desc).
```

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

```
Eval cbn in (model_spec add_desc).
```

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

We must make it clear here that the types `model_spec` returns are currently not provable within our framework. In order to use these equalities, the user would have to treat them as preconditions. One possible way to do so is this:

```
Section UIntProofs.
  Variable from_nat_spec : model_spec from_nat_desc.
  Variable to_nat_spec : model_spec to_nat_desc.
  Variable add_spec : model_spec add_desc.
  ...
End UIntProofs.
```

This way we can presume these equalities generated by `model_spec` only for the proofs about programs using foreign functions and foreign types. Within those proofs, however, we can rewrite calls to those foreign functions into calls to the functions from the functional model, which increases our capability of writing programs that both use the FFI and take advantage of dependent types.

8

Examples

Men become builders by building and lyre players by playing the lyre...

Aristotle [14]

In this chapter, we will investigate examples uses of the foreign function interface, and explore how we can write functional models, specifications, and proofs about them. This chapter will not feature the VST specifications and proofs for the C function in this chapter, as the VST funspecs will be automatically generated using the methods we described in Section 7.4, while the VST proofs are outside of the scope of this dissertation.

8.1 Unsigned Integers

In this section, we will implement 63-bit unsigned integers as a foreign type and functions to convert from/to the natural number type, and to add unsigned integers as foreign functions. The VST proofs for these functions are written by Stark and Appel, and are covered in the tech report by Korkut, Stark, and Appel [59].

8.1.1 The Coq Interface

In Section 7.1, we had defined primitive integers as an example for `reified` descriptions. Our implementation here will follow that example.

The ideal way to organize a module that uses foreign types and foreign functions, is *module types*. By providing a collection of types and functions on those types, we can define our foreign types as abstract data types [66]. Later we can provide different implementations of this abstract data type; the foreign implementation and the functional model are just two of such implementations.

The module type for unsigned 63-bit integers can be defined as such:

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

The foreign implementation in C is computationally opaque from the Coq side. All the foreign types and foreign functions are defined as `Axiom`s in Coq:[15]

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

The foreign functions have to be mapped to their C function names, along with the information of whether the C function to implement a foreign function needs to use the `thread_info`. The system also needs a C header file of the foreign functions, which will be `include`d in the compiled code.

[15] Although it might be nicer to define the entire `C` module as a single axiom of module type `UInt63`, we cannot do so because of a limitation in CertiCoq's treatment of (external) primitive values as free variables.

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

If a C function merely inspects Coq values and doesn't need to create any boxed Coq values, then `thread_info` argument is unnecessary. In these examples, `C.from_nat` inspects and traverses a possibly boxed Coq value, but creates an unboxed Coq value. `C.to_nat` inspects an unboxed Coq value, and creates a possibly boxed Coq value. `C.add` inspects unboxed Coq values and creates an unboxed Coq value.

8.1.2 The C Implementation

Since the glue functions that discriminate Coq values (such as `get_Coq_Init_Datatypes_nat_tag` return an `int` value, we can define a C **`enum`** to make the correspondence between a constructor and its tag clearer. Here, we use the constructor names and order from the original Coq definition of `nat`s to define such an **`enum`**:

```
typedef enum { O, S } nat;
```

We can now define a foreign function `uint63_from_nat` that traverses a Coq `nat` value and creates a value of the foreign type for unsigned integers. (Recall that a `nat` value is a chain of boxed `S` constructors in memory.)

```
value uint63_from_nat (value n) {
  value temp = n;
  uint64_t i = 0;

  while (get_Coq_Init_Datatypes_nat_tag(temp) == S) {
    i++;
```

```
    temp = get_args(temp)[0];
  }
  return (value) ((i << 1) + 1);
}
```

The `uint63_from_nat` function keeps a variable (`temp`) for the remaining Coq `nat` value, and another variable (`i`) for the intermediate result. The function keeps trying to increase `i` until there are no more successors (`S`) left in the `nat` value.

The memory representation of unsigned integers will be unboxed. In a 64-bit word, we set the last bit to `1` to mark that this value is unboxed. We use the remaining 63 bits for the integer itself. This commonly used [67, 22] trick eliminates the pointer indirection and memory allocation and header requirements of boxed values at the cost of one bit[16]. In foreign functions dealing with 63-bit integers, we will see a lot of bit shifting to convert back and forth from this representation.

Implementing `uint63_to_nat` function is less straightforward. For a given integer n, we have to allocate memory for `S`, the successor constructor, n times:

```
value uint63_to_nat (struct thread_info *tinfo, value t) {
  uint64_t i = ((uint64) t) >> ((uint64_t) 1);
  value temp = make_Coq_Init_Datatypes_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;
```

[16]If a user needs that one bit, they could also implement 64-bit integers in this system with a boxed representation.

```
    }
    temp = alloc_make_Coq_Init_Datatypes_nat_S(tinfo, temp);
    i--;
  }
  return temp;
}
```

To convert a 63-bit integer into a Coq `nat`, we have to discard the last bit of the integer and loop over the remaining number. As long as the number is greater than 0, we keep building new successor values, using the glue function `alloc_make_Coq_Init_Datatypes_nat_S`. However, glue functions that build Coq values *do not* check if there is enough space in the CertiCoq heap beforehand; that is assumed as a precondition of these glue functions. Checking if there is enough space and calling the garbage collector if necessary is left to the programmer. This is a deliberate choice that allows the programmer to decide when to do this; the programmer may decide to check at every step for each constructor, or to precalculate how much space will be needed and check if that is available at once.

In this particular function, we choose to check if there is enough space every time we want to allocate an `S` value. We call the garbage collector if there is not enough space left in the CertiCoq heap, but we have to make sure `temp`, our temporary Coq `nat` value is not discarded in this process. We achieve that by declaring `temp` a *root* of the heap; we push a frame on the stack of frames and copy `temp` into that frame. After the collection, we copy back the (possibly forwarded) `temp`, pop the stack, and then we can call the glue function to make a Coq `S` value, since we now know that we have enough space in the CertiCoq heap.

A curious reader might wonder why we chose to check if there is enough space inside the loop instead of outside the loop. Precalculating the necessary space and checking once for the entire result would seem like a reasonable optimization. However, the garbage collector

attempts to guarantee that the nursery (generation 0 in a generational garbage collector) has at least `tinfo->nalloc` available slots. If the space we need is larger than the nursery size, garbage collection will fail.[17]

Macros to Help with Memory Checks It can be tedious to write, in every function that allocates constructors, the code that checks if there is enough space, saves intermediate values in a stack frame, and calls the garbage collector. We hide that repetition behind C macros. We introduce **BEGINFRAME**, **GC_SAVE**k, and **ENDFRAME**. **BEGINFRAME** takes k, the number of values that will be saved, as an argument. **GC_SAVE**k takes the number of available slots in the CertiCoq heap we need. **ENDFRAME** merely closes delimiters opened by **BEGINFRAME**. We also introduce a macro **LIVEPOINTERS**k that we can wrap around function calls that might possibly include memory checks and garbage collector calls.

With these macros, we can define the `uint63_to_nat` function in a simpler way:

```
value uint63_to_nat(struct thread_info *tinfo, value t) {
  uint64_t i = (uint64_t) (((uint64_t) t) >> 1);
  value save0 = make_Coq_Init_Datatypes_nat_O();
  BEGINFRAME(tinfo, 1)
  while (i) {
    GC_SAVE1(2)
    save0 = alloc_make_Coq_Init_Datatypes_nat_S(tinfo, save0);
    i--;
  }
  return save0;
  ENDFRAME
}
```

[17] We could, of course, check the nursery size and set `tinfo->nalloc` accordingly, as few times as possible. However, that is a more complicated implementation and VST proof.

Finally, we can also implement `uint63_add`, which adds two 63-bit unsigned integers:

```
value uint63_add(value x, value y) {
  return (value) ((((((uint64_t) x) >> 1) + (((uint64_t) y) >> 1)) << 1) + 1);
}
```

This function first converts the `value`s into 64-bit C integers by discarding the last bit, followed by the addition, after which the result is converted back to the old representation by the last bit being set to `1`.

That concludes the C implementation of 63-bit unsigned integers. Along with the Coq interface, this part suffices for the operational use of VeriFFI. The next subsection is only relevant for users who want to prove properties about their foreign functions.

8.1.3 The Functional Model

The Verified Software Toolchain [13, 12] allows the user to prove that a C function implements a functional model, that is, a mathematical model of the C function we want to prove correct. Coq is merely the language in which we express mathematics.

VST does not impose any requirements about the type of the functional model. As long as the arguments are quantified in the resulting VST function specification, the C type of the C function and the Coq type of the functional model do not need to line up exactly. (The VST user provides a *representation relation* characterizing the correspondence.) Our recipe for verifying foreign functions will be stricter; we wish both the C side and the functional model to be compatible with the same **`Module Type`**, namely `UInt63` in this case.

`FM`, the module that contains the functional model, is simply a module that implements[18] the same **`Module Type`** as `C`, the module that contains the axiomatized references to the foreign

[18] We use <: instead of : when we ascribe the module type to the module, since we want the definitions in the module to be transparent to the outside. This allows us to write proofs about the functional model later.

functions. For a meaningful mathematical model of foreign types and foreign functions, the `FM` module should consist of concrete definitions in plain Coq:

```
Module FM <: UInt63.
  Definition uint63 : Type := {n : nat | n < (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) : uint63 :=
    (Nat.modulo n (2 ^ 63); mod63_ok _).

  Definition to_nat (i : uint63) : nat :=
    let '(n; _) := i in n.

  Definition add (x y : uint63) : uint63 :=
    let '(xn; x_pf) := x in
    let '(yn; y_pf) := y in
    let n := (xn + yn) mod (2 ^ 63) in
    (n; mod63_ok _).
End FM.
```

Machine integers are bounded, therefore we have to model the lower and upper bounds of integers. For unsigned 63-bit integers, the bounds are $[0, 2^{63})$. The `nat` type has a lower bound of 0 by construction, therefore we only have to model the upper bound, which we do using the Σ-type to express (constructive) existential quantification, as seen above in the definition of `FM.uint63`, the functional model of our foreign type.

We define the `FM.from_nat`, using the modulo operation to model integer overflow. Similarly, `FM.to_nat` unpacks the existential value and returns it. Finally, `FM.add` unpacks both inputs, adds the `nat`s and takes the modulo of the sum.

We use the lemma `mod63_ok`, which ensures that the modulo of a number with our upper bound is less than our upper bound, to satisfy the propositional part of the existential.

8.1.4 Model Proofs for Foreign Functions

We now have two implementations of unsigned 63-bit integers: The first one lives in the `C` module; it looks like axioms on the Coq side but when compiled to C, it is backed by foreign functions in C. The second one lives in the `FM` module; it is a functional model that is implemented in plain Coq. The first one is meant to be executed, while the second one is meant to be used in proofs. In Section 7.5, we set up the mechanisms for us to freely rewrite calls to the functions in `C` to calls to the functions in `FM`. With these rewrites, we can prove theorem statements that involve the functions in `C`.

```
Section UInt63Proofs.
```

We can start by opening a new **`Section`**, which allows us to assume equalities and isomorphisms without having to make them into actual Coq axioms.

```
Declare Instance Isomorphism_uint63 : Isomorphism FM.uint63 C.uint63.
```

We assume there is an isomorphism between the foreign type and its functional model. (We will later justify this assumption by instantiating `C.uint63 := FM.uint63`; here the isomorphism is just to prevent the user from breaking the abstraction.) A foreign value and the mathematical object that models that foreign value are not same entity, but this (constructive) isomorphism makes it is possible to convert between them.

```
Instance GraphPredicate_uint63 : GraphPredicate FM.uint63 :=
  {| graph_predicate (g : graph) (o : outlier_t) (x : FM.uint63) (p : rep_type) :=
      let '(n; _) := x in
      match p with
      | repZ i => i = Z.of_nat n
      | _ => False
      end |}.
```

```
Instance ForeignInGraph_uint63 : ForeignInGraph FM.uint63.
Proof. ... Qed.
```

We define `GraphPredicate` and `ForeignInGraph` instances for the foreign type and its functional model. Notice that the `graph_predicate` definition takes the functional model value as an input, as the foreign type in the `C` module is completely opaque, therefore values of that type cannot be inspected in plain Coq. This predicate expresses that the integer is represented as an unboxed vertex (`repZ`) in the CertiCoq heap graph.

```
(* After generating from_nat_desc, to_nat_desc, and add_desc. *)
Variable from_nat_spec : model_spec from_nat_desc.
Variable to_nat_spec : model_spec to_nat_desc.
Variable add_spec : model_spec add_desc.
```

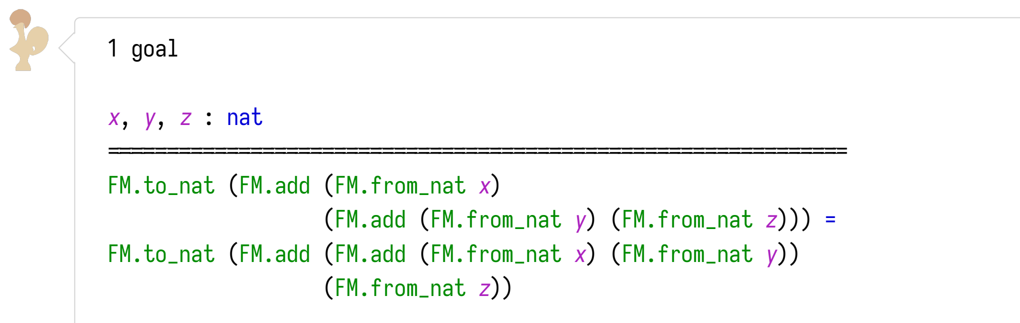
Now we get to use the mechanisms in Section 7.4 to generate descriptions of foreign functions, and from them, using `model_spec` mechanism from Section 7.5, we can generate the equalities we will use to rewrite functions in the `C` module to their counterparts in the `FM` module.

```
Theorem add_assoc : forall (x y z : nat),
  C.to_nat (C.add (C.from_nat x) (C.add (C.from_nat y) (C.from_nat z))) =
  C.to_nat (C.add (C.add (C.from_nat x) (C.from_nat y)) (C.from_nat z)).
Proof.
  intros x y z.
  props to_nat_spec.
  props add_spec.
  props from_nat_spec.
  foreign_rewrites.
  (* The rest is just a proof about the functional model! *)
  unfold FM.add, FM.from_nat, FM.to_nat.
  unfold proj1_sig.
  rewrite <- !(Nat.Div0.add_mod y z).
```

```
  rewrite <- !(Nat.Div0.add_mod x y).
  rewrite <- !(Nat.Div0.add_mod).
  f_equal.
  apply Nat.add_assoc.
  all: apply Nat.pow_nonzero; auto.
Qed.

End UInt63Proofs.
```

Using these assumptions, we are now able to prove the associativity of 63-bit unsigned machine integers in our system. The first half of our proof consists of rewrites based on the `model_spec` assumptions. After those rewrites, the proof goal looks like this:



```
1 goal

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

At this point in the proof, we can proceed as if the proof is only about the functional model, as in can be seen in the second half of `add_assoc`.

8.2 Bytestrings

The Coq `string` type is defined as a linked list of `ascii`, each of which is record of 8 `bool`s:

```
Inductive ascii :=
| Ascii : bool -> bool -> bool -> bool -> bool -> bool -> bool -> bool -> ascii.

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

As we have seen in Section 4.1, each `String` constructor is represented as three words (a header and two pointers), and each `Ascii` constructor is nine words, in which each `bool` is unboxed. This means when a value of the Coq type `string` is compiled to C, our system uses 96 bytes (on a 64-bit system) per ASCII character, which clearly is inefficient. Instead, we can introduce a foreign type for bytestrings, where each character would occupy one byte (as they do in OCaml [67, Chapter 23, "string values"]), and provide foreign functions to convert between this foreign type and the Coq `string`s, and operations on this foreign type.

In this section, we implement such a foreign type and its foreign functions. The VST proofs for these functions are written by Stark and Appel, and are covered in the tech report by Korkut, Stark, and Appel [59].

8.2.1 The Coq Interface

We start by defining a **`Module Type`**, `Bytestring`, that contains the foreign types and foreign functions we need to implement bytestrings.

```
Module Type Bytestring.
  Parameter bytestring : Type.
  Parameter pack : string -> bytestring.
  Parameter unpack : bytestring -> string.
  Parameter append : bytestring -> bytestring -> bytestring.
End Bytestring.
```

We proceed by defining a module, `C` that satisfies the `Bytestring` interface. This module states that all values in the `Bytestring` interface will be foreign types and foreign functions.

```
Module C : Bytestring.
  Axiom bytestring : Type.
  Axiom pack : string -> bytestring.
  Axiom unpack : bytestring -> string.
  Axiom append : bytestring -> bytestring -> bytestring.
End C.
```

We have to register with CertiCoq the foreign functions and the names of C functions that will realize those foreign functions when a client program is compiled to C.

```
CertiCoq Register
  [ C.pack => "pack" with tinfo,
  , C.unpack => "unpack" with tinfo,
  , C.append => "append" with tinfo,
  ] Include [ "prims.h" ].
```

8.2.2 The C Implementation

For brevity, let us explain only the C implementation of the `C.pack` function out of these foreign functions. This function converts a Coq `string` into our foreign type `C.bytestring`. To understand how this function would be implemented in C, we have to grasp the memory representations of these types.

We have already discussed the memory representation of the `string` type, as it is just another inductive type. Once we add `string` to the list of types to generate glue code about, we will have the `string` counterparts of all the helper functions we went over in Section 3.2.

The memory representation of `C.bytestring` will be more complicated. For further compatibility with the OCaml ecosystem, we use the same bytestring representation as OCaml. We would have the following memory representation (in a 64-bit setting) for the bytestring `"interface"`:

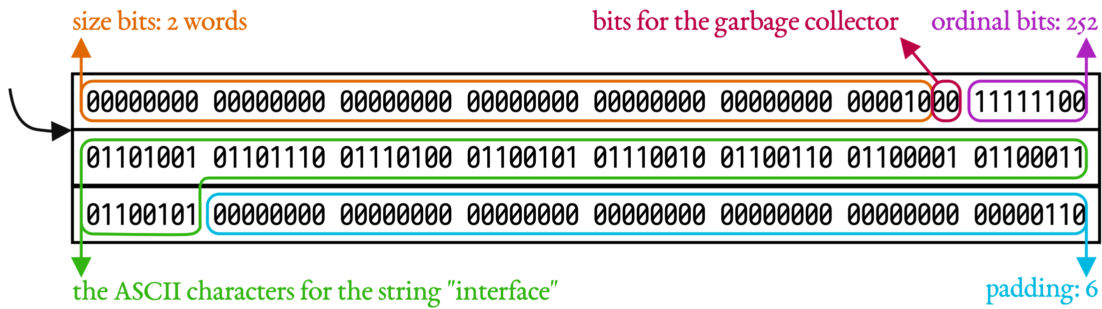


Figure 8.1: The representation of the bytestring `"interface"` in memory.

This string's representation consists of 3 words, one for the header and the remaining two for the characters and padding. After the header, we use one byte per ASCII character, terminated by a null byte. However, our runtime system assumes that values consist of words in memory, and our number of character bytes might not line up with the word size. We alleviate this discrepancy by filling the remaining bytes in the word with a padding. This padding consists of null bytes, with the last byte containing how many extra null bytes had to be added. If the characters and the terminating null byte perfectly fit in words, the last byte is a null byte, which serves both as the terminating byte and also the number of extra null bytes that had to be added.

The header contains the size of the string content, which is 2 words in this example. The last part of the header contains the number 252, which is a special tag for bytestrings in OCaml. This tag 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.

Finally, just as it is the case for all other values in our memory representation, the header is one word behind where the pointer points to.

Now that we understand how bytestrings are supposed to be represented in memory, we can start the C implementation. We can start with defining an `enum` for the possible constructors of the Coq type `string`:

```
typedef enum { EMPTYSTRING, STRING } string;
```

Here is what the implementation for `C.pack` looks like:

```
value pack(struct thread_info *tinfo, value save0) {
  BEGINFRAME(tinfo, 1)

  // Allocate enough memory
  value temp = save0;
  size_t i, len = 0;
  while (get_Coq_Strings_String_string_tag(temp) == STRING) {
    len++;
    temp = get_args(temp)[1];
  }
  size_t mod = len % sizeof(value);
  size_t pad_length = sizeof(value) - (len % sizeof(value));
  size_t nalloc = (len + pad_length) / sizeof(value) + 1ULL;
  GC_SAVE1(nalloc)

  value *argv = tinfo->alloc;
  argv[0LLU] = (value)(((nalloc - 1) << 10) + 252LLU); // string tag

  // Make the characters
  char *ptr = (char *) (argv + 1LLU);
  temp = save0;
  while (get_Coq_Strings_String_string_tag(temp) == 1) {
    *ptr = ascii_to_char(get_args(temp)[0]);
    ptr++;
    temp = get_args(temp)[1];
  }
```

```
  // Make the padding
  for (i = 0; i < pad_length - 1; i++) {
    ptr[i] = 0;
  }
  ptr[i] = i;

  tinfo->alloc += nalloc;
  return (value) (argv + 1LLU);
  ENDFRAME
}
```

This function performs the following steps:

1. traverses the Coq `string` once to compute its length, stored in `len`
2. makes sure there are $n = 1 + \lceil (\texttt{len} + 1)/8 \rceil$ words in the CertiCoq heap, where it can place the bytestring
3. makes the bytestring header using the special tag and the number of words needed for the bytestring content
4. makes the character by traversing the Coq `string` and converting each C `char` to a Coq `ascii`, by calling the `ascii_to_char` function, which we have implemented by hand but omit here
5. makes the padding by adding null bytes, followed by how many extra null bytes had to be added
6. returns a pointer to the first character byte.

Our implementations of `C.unpack` and `C.append` are similar to the implementation of `C.pack` that we presented here. To summarize, `C.unpack` traverses all the character bytes in the bytestring

and creates `ascii` and `string` values, using the glue code. `C.append` computes the lengths of both of its inputs, allocates enough words in the CertiCoq heap, copies characters, and adds padding.

8.2.3 The Functional Model

The simplest way to model bytestrings in Coq is to define them as Coq `string`s. This makes the functional models of all the functions that operate of our foreign type trivial:

```
Module FM.
  Definition bytestring : Type := string.
  Definition pack (x : string) : bytestring := x.
  Definition unpack (x : bytestring) : string := x.
  Definition append (x y : bytestring) : bytestring := append x y.
End FM.
```

A curious reader might wonder why we did not address the length of the bytestring in our functional model. This would be a fair question, given that we *have* addressed a similar restriction in the functional model for integers, in Section 8.1. Our answer in this case is practicality. In the memory representation above, we showed that the bytestring header contains 54 bits (in a 64-bit setting) for the size of the string, in words. Thus, the largest bytestring we can create will have $(2^{54}) \cdot 8$, or 2^{57} bytes, which is 128 petabytes. For `C.pack`, we would need to start with a Coq `string`. In order to reach a bytestring that is 128 petabytes, at 2 words per record, we would need $2 \cdot 8 \cdot 2^{57} = 2^{61}$ bytes of memory, i.e. ≈ 2.3 exabytes.[19] Here we are assuming we are not going to run Coq functions on memories that large, but it is also possible to design a functional model for bytestrings in a way that would account for the maximum

[19] This question and its answer was observed by Appel.

length.[20]

We will not present any proofs about the models for the foreign functions on `bytestring`s since our model is trivial and all proofs will simply be about Coq `string`s.

8.3 Printing Bytestrings

So far, we have only implemented pure foreign functions. Neither integers nor bytestrings have any side effects; functions that operate on them are guaranteed to return the same result every time they use the same arguments,[21] with no observable difference from the outside. Real programs, however, often need to have side effects, therefore we need to find a way to write effectful programs if we want to be able to express them.

Two factors require us to keep the language pure even when expressing effectful programs. The first factor is that Coq does not have a default evaluation order, since its underlying type theory enjoys strong normalization and confluence [35, 119]. The second factor is that we use our functions in proofs and types, and we do not want proofs and types to depend on the outside world.

Thankfully, we are not the first to face this conundrum. The Haskell community has encountered the same question from the beginning; in their case, Haskell's lazy evaluation made it difficult to predict when a function's side effect would be executed. Their solution was to forbid functions from having side effects[22] and to exile all side effects into the `IO` monad. Effectful programs are then expressed as terms with an `IO` type, parametrized by the result type

[20]In such a design, the `pack` function would have decide how to treat the inputs that would create bytestrings larger than what the bytestring representation could handle. The `pack` function could choose to return only the part that fits in memory, or it could choose to return an `option` value, etc.

[21]As long as there is enough space in the CertiCoq heap. If not, garbage collection will fail with an error.

[22]As noted in the famous XKCD comic:
"Code written in Haskell is guaranteed to have no side effects." "... because no one will ever run it?" [75]

of the computation. For example, here are the Haskell terms that we use for printing to and scanning from the console:

```
putStrLn :: String -> IO String
getLine :: IO String
```

Neither of these terms have side effects by themselves; they are merely *action*s that we can use to build a program. The user composes these actions using the monadic bind to build larger programs. Here is an example Haskell program, written as a chain of monadic actions (without syntactic sugar, i.e. **do** notation):

```
main :: IO ()
main = getLine >>= (\x -> putStrLn ("Hello " ++ x))
```

This program, when executed, reads a line of user input from the console, prepends the input string with another string, and prints the new string. What triggers the execution, however, is the runtime: Haskell's runtime is hard-coded to execute the `main` action when the program starts. The runtime has an interpreter for actions, which executes the left side of the bind, plugs the result in the function on the right side of the bind, and keeps interpreting. This design allows us to express effectful programs purely (since actions are values that do not have side effects before execution), provides explicit sequencing for effectful computations, and makes the execution order of effects predictable.

We want to express effectful programs the same way in Coq, for which we will have to write an interpreter for actions in C.

In this section, we will extend our foreign bytestring implementation from the previous section with a monad that has effectful actions to print bytestrings into a stream and to scan (read) bytestrings from a stream. The VST proofs for these functions will be addressed in future work.

8.3.1 The Coq Interface

Our interface will have to contain a monadic type `M`, a function `pure` to inject pure values into our monad (also called returning), and a function `bind` to express an effect followed by more computation or effects. We also expect functions in the interface that express effects, such as `print` for printing and `get_line` for taking inputs. Here is what that interface looks like as a **`Module Type`**:

```
Module Type Bytestring.
  (* Definitions from the previous version of Bytestring *)
  Parameter M : Type -> Type.
  Parameter pure : forall {A : Type}, A -> M A.
  Parameter bind : forall {A B : Type}, M A -> (A -> M B) -> M B.
  Parameter print : bytestring -> M unit.
  Parameter get_line : M bytestring.
  Parameter same_behavior : forall {A : Type}, M A -> M A -> Prop.
End Bytestring.
```

Notice that we have an additional element in this module type, called `same_behavior`. We will use this relation later in Subsection 8.3.4 to reason about actions, to assert that two actions have the same behavior.

Following the recipe we demonstrated in the previous examples, we want to implement a `C` module of the module type `Bytestring`. Until now, the `C` modules we have seen have consisted of **`Axiom`**s. These axioms were only realized when a Coq program that uses them is compiled to C and the compiled program is linked to the C file containing the foreign functions. We will, however, diverge from this recipe now.

Just like how we would design a abstract syntax tree (AST) in Coq for an interpreter in Coq, we will design an AST in Coq for an interpreter in C. The user will write their programs using this AST, and compile them. The interpreter for this AST, which is written by the

user of our foreign function interface, will be called from the C wrapper file described in Chapter 3.

```
Module C <: Bytestring.
  (* Definitions from the previous version of C for bytestrings *)

  Inductive MI : Type -> Type :=
  | pureI : forall {A : Type}, A -> MI A
  | bindI : forall {A B : Type}, MI A -> (A -> MI B) -> MI B
  | printI : bytestring -> MI unit
  | get_lineI : MI bytestring.

  Definition M : Type -> Type := MI.
  Definition pure : forall {A : Type}, A -> M A := @pureI.
  Definition bind : forall {A B : Type}, M A -> (A -> M B) -> M B := @bindI.
  Definition print : bytestring -> M unit := @printI.
  Definition get_line : M bytestring := @get_lineI.

  Axiom same_behavior : forall {A : Type}, M A -> M A -> Prop.
End C.
```

We define here an inductive type[23] that has constructors for both the monadic components (such as returning and binding) and the effectful foreign functions. We then provide aliases for these constructors, because Coq modules do not allow us to use constructors to satisfy the requirements of a `Module Type`. Finally, we axiomatize `same_behavior`, which will only make sense when we have a functional model for it.

With this interface, we can write the same Haskell function now in Coq:

```
Definition main : C.M unit :=
  C.bind C.get_line (fun x => C.print x).
```

[23]If we wanted to express possibly nonterminating programs, we could have described the free monad as a *coinductive* type. Other than having to force the thunk explicitly from the C side, not much would have changed operationally. Abel et al. [1] explore a coinductive `IO` monad for Agda, which we could base a similar Coq system on.

8.3.2 The C Implementation

Another way we have diverged from the usual recipe is the lack[24] of the use of **CertiCoq Register** to register any of our effectful functions. This is a deliberate choice on our part; we will implement a function `runM` that executes an action, and `runM` will be called in the C wrapper file after computing the program result. We define `runM` as such:

```
typedef enum { PURE, BIND, PRINT, GETLINE } M;

value runM(struct thread_info *tinfo, value action) {
  BEGINFRAME(tinfo, 2)
  switch (get_prog_C_MI_tag(action)) {
    case PURE:
      return get_args(action)[1];
    case BIND: {
      value arg0 = get_args(action)[2];
      value arg1 = get_args(action)[3];
      value temp = LIVEPOINTERS1(tinfo, runM(tinfo, arg0), arg1);
      temp = call(tinfo, arg1, temp);
      return runM(tinfo, temp);
    }
    case PRINT: {
      value arg0 = get_args(action)[0];
      print_bytestring(s);
      return make_Coq_Init_Datatypes_unit_tt();
    }
    case GETLINE:
      return get_line_bytestring(tinfo);
    default:
      return 0;
  }
  ENDFRAME
}
```

[24] We still use **CertiCoq Register** for `C.pack`, `C.unpack`, and `C.append` on bytestrings.

We start by defining a C `enum` that makes it easier to understand what constructor of `C.MI` is handled, and then move on to the `runM` function. `runM` traverses the action AST that was built on the Coq side and returns the value inside the monad. The `PURE` case is simple; we just return the argument in the constructor.[25] The `PRINT` and `GETLINE` cases are simple to interpret, although here we elide the `print_bytestring` and `get_line_bytestring` functions, where the complexity of these cases is hidden.

The `BIND` case is at the crux of our interpreter for monadic actions. We collect the monadic action and the continuation function from the `MI.bindI` constructor. Then, we have to execute the monadic action to obtain the input of the continuation function, which we achieve with a recursive call to `runM`. When executing this monadic action, we have to keep in mind that `arg1` is not in the roots, and it should be protected if our recursive call ultimately calls the garbage collector. Once we know what input to call the continuation function with, we can `call` it. The continuation function returns another monadic action, which we can pass to another recursive call to `runM`.

This function currently uses the C call stack to handle recursion. To avoid stack overflow errors, we can choose to implement `runM` with an explicit call stack, or we can try to rewrite the action input of `runM` using the associativity law of monads [115], where `C.bind (C.bind m f) g` can be rewritten to `C.bind m (fun x => C.bind (f x) g)`. Repeated applications of such rewrites can ensure that the action argument to `C.bind` is atomic, after which we only need tail recursion to execute the remaining action. We leave this implementation detail to the users of our foreign function interface.

[25] Note that the `0`th index in the array returned by `get_args` will contain the type argument for `MI.pureI`.

8.3.3 The Functional Model

Following the Haskell tradition [86], we define the functional model of our effectful monads as a function from the state to a pair of a result and the new state. For our particular set of effects, where we only write to the console and read from the console, we model the program state as the input stream to the program and the output stream from the program. The rest of our functional model definitions mirror the purely functional state monad [55].

```
Module FM <: Bytestring.
  (* Definitions from the previous version of FM for bytestrings *)

  Definition state : Type :=
    (string * string). (* the input stream and the output stream *)
  Definition M (A : Type) : Type := state -> A * state.
  Definition pure {A : Type} (a : A) : M A := fun s => (a, s).
  Definition bind {A B : Type} (m : M A) (f : A -> M B) : M B :=
    fun s => let '(a, s') := m s in f a s'.

  Definition print (x : bytestring) : M unit :=
    fun '(input, output) => (tt, (input, append output x)).

  Definition get_line : M bytestring :=
    fun '(input, output) => (substring 0 n input,
        (substring n (length input) input, output)).

  Definition same_behavior A (a1 a2 : M A) : Prop :=
    forall (s : state), a1 s = a2 s.
End FM.
```

Before we move on to the proofs about our functional model, let us observe our `same_behavior` definition above. We left its definition as an axiom in the `C` module with no intention of realizing its definition on the C side, but we are actually providing a Coq definition in the functional model. We do not need to provide a definition in the `C` module; `same_behavior`

returns a Prop, which is erased by CertiCoq. Therefore same_behavior itself never has computationally relevant content. In FM, however, we need a real definition so that we can later rewrite the uses of C.same_behavior into FM.same_behavior and proceed with the proof.

8.3.4 Model Proofs for Foreign Functions

In the previous examples of the foreign function interface, we have used Coq's equality type = to claim equalities between values of a foreign type. Unfortunately, this approach does not make sense for effectful actions; these values are ASTs of actions and the equality of the AST is not correlated with the equality of the output when the actions are executed. It is possible for two action ASTs to be different and have the same output when executed, just as it is possible for two ASTs to be the same and have different outputs. Therefore, we will use the same_behavior relation that we included in our interface.

```
Section BytestringProofs.

(* Generation of foreign function descriptions. *)
Parameter pure_spec.
Parameter bind_spec.
Parameter print_spec.
Parameter get_line_spec.
Parameter same_behavior_spec.
```

We use the mechanisms in Section 7.4 and Section 7.5 to set ourselves up to rewrite functions from C to their FM counterparts. We are now ready to prove a simple result about printing bytestrings, following a similar idea to that of Swierstra and Altenkirch [109], who provide a functional semantics for the "awkward squad" to reason about impure code:

```
Theorem print_steps :
  forall (a b : C.bytestring),
    C.same_behavior (C.bind (C.print a) (fun _ => C.print b))
                    (C.print (C.append a b)).
Proof.
  intros a b.

  eapply (transport id). symmetry.
  eapply (same_behavior_spec unit); simpl.
  foreign_rewrites.
  unfold FM.same_behavior.

  props bind_spec.
  foreign_rewrites.
  unfold FM.bind.

  props print_spec.
  foreign_rewrites.
  unfold FM.print.

  props append_spec.
  foreign_rewrites.
  unfold FM.append.

  intros [instream outstream].
  rewrite append_assoc; auto.
Qed.

End BytestringProofs.
```

We provide the type argument `unit` to `same_behavior_spec`, since our example program has the type `C.M unit`. Then we rewrite `C` terms to their `FM` counterparts, unfold the `FM` definitions, and finish the proof.

Unlike the model proof in Section 8.1, our model proof on monadic effects does not let us rewrite a theorem statement about functions from `C` into a theorem statement about func-

tions from `FM` all at once. Since the rest of the monadic action depends on the result of the continuation, we can only rewrite and unfold one step at a time. Once the functional model definition of the continuation is evaluated, we can rewrite and unfold it once again.

8.4 Mutable Arrays

Purely functional data structures are easier to reason about than imperative data structures. However, they are inherently inefficient for some use cases [91, 17, 89, 76], therefore we inevitably need mutable data structures.

Mutable data structures à la OCaml break purity, which is why we only want to allow them in a controlled way. One way to implement them without breaking purity is, of course, to implement them as an effect, similar to how we printed bytestrings in Section 8.3. However, this may be more restricting than we need. As Launchbury and Peyton Jones [61] have demonstrated for the `ST` monad in Haskell, it is possible to have local mutation that is externally pure. That is, a client of our function with local mutation cannot tell if our function has mutation in it; its type is pure and it returns the same value for the same inputs.

In this section, we will implement a monad to use mutable arrays as a foreign type in Coq, and functions to set, get, and run the monad as foreign functions. The VST proofs for these functions will be addressed in future work.

8.4.1 The Coq Interface

Here we will follow a similar approach to how we handled effects in Section 8.3, but we can now expose `runM` to the Coq side, since we can model the mutation in a purely functional way and it does not break purity.

```
Definition elt := nat.

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

Once again, we have monadic components for returning and binding, as well as "effectful" operations on arrays, such as getting and setting. The main difference is the `runM` function, which takes the length of the initial array and the initial value at unset indices of the array.

Our approach here is different from that of Launchbury and Peyton Jones in that we do not apply their rank-2 types trick for the `ST` monad, which restricts references from escaping the monad. Instead, we never expose a reference to a mutable variable (or the entire array) from the interface. The user will have only one array in each instance of the monad `M`.[26] While our interface here looks similar to that of Sakaguchi [94], our approach here is simpler and less type-safe.[27] This is a deliberate design choice to make the proof simpler to present.

```
Module C <: Array.
  Inductive MI : Type -> Type :=
  | pureI : forall {A : Type}, A -> MI A
  | bindI : forall {A B : Type}, MI A -> (A -> MI B) -> MI B
  | setI : nat -> elt -> MI unit
  | getI : nat -> MI elt.

  Definition M := MI.
  Definition pure : forall {A : Type}, A -> M A := @pureI.
```

[26] Users can have multiple arrays using a monad stack.

[27] For `C.set`, we ignore the operation if the index is out of bounds. For `C.get`, we return the default element if the index is out of bounds.

```
  Definition bind : forall {A B : Type}, M A -> (A -> M B) -> M B := @bindI.
  Definition set : nat -> elt -> M unit := @setI.
  Definition get : nat -> M elt := @getI.
  Axiom runM : forall A (len : nat) (init : elt), M A -> A.
End C.
```

Similar to Section 8.3, we implement operations in the mutable array monad as an abstract syntax tree of actions. We will still have a `runM` function in C to interpret these actions, but this time, we will expose it to the Coq side by declaring it an **`Axiom`** and telling the compiler that this axiom is realized by a particular C function.

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

8.4.2 The C Implementation

We implement the C side of the mutable array monad interpreter with two functions: `runM` allocates the necessary space on the CertiCoq heap and initializes the array with the initial value at each index. `executeM` traverses the monadic action and executes it.

```
value runM(struct thread_info *tinfo, value a, value len,
                 value save0, value save1) {
  BEGINFRAME(tinfo, 2)
  size_t size = nat_to_size_t(len);
  nalloc = size + 1; GC_SAVE2
  value *arr = tinfo->alloc;
  arr[0LLU] = size << 10;
  arr = arr + 1LLU;
  for (size_t i = 0; i < size; i++) {
    arr[i] = save0;
  }
  tinfo->alloc += nalloc;
  return executeM(tinfo, size, save0, (value) arr, save1);
  ENDFRAME
}
```

The function first converts[28] the array length *len*, which is a Coq `nat`, to a C `size_t`. This number determines how much space we need to allocate on the CertiCoq heap. We treat the array like a Coq constructor with that many arguments, we allocate one word for the header and one word per array index. Of course, while we perform a memory check and possibly call the garbage collector, we have to make sure the initial value for the array (*save0*) and the monadic action (*save1*) are not collected by accident, therefore we save them in a stack frame using the `GC_SAVE` macro. Once we have enough space, we initialize the array with the initial value *save0*, and call `executeM` to execute the monadic action.

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

value executeM(struct thread_info *tinfo, size_t size,
           value init, value save0, value save1) {
  BEGINFRAME(tinfo, 2)
  switch (get_prog_C_MI_tag(save1)) {
    // Cases for PURE and BIND as before
    case SET: {
      value arg0 = get_args(save1)[0];
      size_t i = nat_to_size_t(arg0);
      if (i < size) {
        nalloc = 1; GC_SAVE2
        value arg1 = get_args(save1)[1];
        certicoq_modify(tinfo, (value *) save0 + i, arg1);
      }
      return make_Coq_Init_Datatypes_unit_tt();
    }
    case GET: {
      value arg0 = get_args(save1)[0];
      size_t i = nat_to_size_t(arg0);
      if (i < size) {
        return get_args(save0)[i];
      } else {
```

[28] The implementation of `nat_to_size_t` is almost the same as `uint63_from_nat`, just without the bit shifting to make the last bit `1`.

```
        return init;
      }
    }
  }
  ENDFRAME
}
```

Our executeM implementation is similar to the runM implementation in Subsection 8.3.2; it is a free monad interpreter written in C. We omit the PURE and BIND cases here since they are almost the same as the previous interpreter. The SET and GET cases, however, are interesting.

When we see a SET action, we want to find the right slot in the array and assign the new value in that slot. However, we have to notify the garbage collector that we have a mutable reference, because the garbage collector normally operates on the assumption that older values never point on the newer values, which allows it to move values around freely and erase unused values [44]. By creating a mutable reference, we break that assumption, therefore we have to keep a remembered set of old values pointing to new values, so that the garbage collector can update these references when new values are moved [95]. We achieve that with a call to certicoq_modify[29], which is a write barrier that saves the mutable reference to the bottom end of the CertiCoq heap and therefore keeps a remembered set.

When we see a GET action, we simply access the index in the array if it is within bounds, or return the default value of all indices if the index is out of bounds.

Reuse in Persistent Arrays While the design of our array interface and its foreign functions is for mutable arrays, it is possible to use the same memory representation of arrays for persistent arrays, which are recently introduced to the Coq standard library [33, 39].

[29] certicoq_modify was implemented by Tim Carstens and Appel.

8.4.3 The Functional Model

The functional model for our mutable array monad is similar to our functional model for effects in Section 8.3, except we also provide an implementation for runM, which interprets the abstract syntax trees of actions in a purely functional way.

```
Module FM <: Array.
  Definition state : Type :=
    (list elt * elt). (* the internal list and the default element *)
  Definition M (A : Type) : Type := state -> A * state.
  Definition pure {A : Type} (a : A) : M A := fun s => (a, s).
  Definition bind {A B : Type} (m : M A) (f : A -> M B) : M B :=
    fun s => let '(a, s') := m s in f a s'.
  Definition set (index : nat) (x : elt) : M unit :=
    fun '(l, init) => (tt, (replace_nth index l x, init)).
  Definition get (index : nat) : M elt :=
    fun '(l, init) => (nth index l init, (l, init)).
  Definition runM {A : Type} (len : nat) (init : elt) (m : M A) : A :=
    fst (m (repeat init len, init)).
End FM.
```

8.4.4 Model Proofs for Foreign Functions

In Subsection 8.3.4, we argued that using Coq's equality type = does not make sense for effectful actions, since the output of the execution depends on the outside world. In our mutable array monad, however, the output does not depend on the outside world, and we do not break purity by calling runM to execute the monadic action. Therefore, we can call C.runM on both sides of the equality, and as long as the array size and default value are the same, we can argue that two different actions have the same result.

In the example below, we prove that first setting an index of an array to a value and then getting the value at that index, gives the same final result (but not the same state). Our proof

once again resembles that of Swierstra and Altenkirch [109], since we use the functional semantics in `FM` to reason about mutable state:

```
Lemma set_get :
  forall (n len : nat) (bound : n < len) (init : elt) (to_set : elt),
    (C.runM len init (C.bind (C.set n to_set) (fun _ => C.get n)))
      =
    (C.runM len init (C.pure to_set)).
Proof.
  intros n len bound init to_set.

  props runM_spec.
  foreign_rewrites.
  unfold FM.runM.

  props bind_spec.
  props pure_spec.
  foreign_rewrites.
  unfold FM.bind, FM.pure.

  props set_spec.
  props get_spec.
  foreign_rewrites.

  eapply nth_replace_nth.
  rewrite repeat_length.
  auto.
Qed.
```

Similar to Subsection 8.3.4, we have to rewrite and unfold the `C` functions one at a time, since the second argument of `C.bind`, the continuation, hides more uses of functions from `C` inside. Once the functional model definition of the continuation is evaluated, we continue rewriting and unfolding as necessary and finally finish the proof by reasoning about the functional model definitions `FM.set` and `FM.get`.

9

Related Work

Comparison is the thief of all joy.

Theodore Roosevelt, in a letter to W.S. Bigelow, 1898

Some of the main contributions of this dissertation, as well as the examples presented in Chapter 8, build upon preexisting work in the literature. In this section, we review the precursors and inspirations for these ideas and compare them to our approach.

9.1 Comparison with the Predecessors of Reified Descriptions

In Chapter 5, we introduced `reified`, a Coq inductive data type that can describe and annotate Coq types. This representation is a combination of deep and shallow embeddings. We used `reified` to describe constructors (in Chapter 6) and foreign functions (in Chapter 7), and to generate VST specifications.

Ours was not the first combination of deep and shallow embeddings. Others have also attempted to create similar representations and observed that such approaches are easy to interpret into other types. We can categorize these attempts as general approaches and domain-specific approaches.

On the general side, McBride (2010) [72] presents a deep/shallow embedding for a dependent type theory, where they use mutual induction and induction-recursion [41] to define the syntax and typing judgments for object language, as well as its interpretation to the host

language at the same time. This approach allows the user to take advantage of the host language's type equality, instead of defining a semantics for the object language. Prinz et al. (2022) [92] present another way of combining deep and shallow embeddings without using induction-recursion. This method indexes the object language definition by the shallow embedding of every term. While this is a less natural way to define an object language, it requires less from the host language, making it easier to achieve syntactic transformations like renaming and substitution.

In comparison to these general approaches, our approach of `reified` description with annotations is not as general; it can be considered a special case of McBride [72] or Prinz et al. [92], except where both the object language and the host language is Coq. This coincidence enables us to reuse more features of the host language than name binding and normalization; we can also annotate the components of a Coq type with Coq type class instances, and 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 6.1 and Section 7.1 to achieve reflection of constructors and foreign functions from their descriptions.

On the domain-specific side, Svenningsson and Axelsson (2013) [107] present a design pattern in Haskell, where an embedded domain-specific language is presented in a deeply embedded way, augmented with a type class that allows expressing a correspondence between the deep embedding and its shallowly embedded counterpart. In the same tradition of research, Gibbons and Wu (2014) [49] explore the algebraic connection between deep and shallow embeddings of domain-specific languages and how folds are critical for this connection, even though the paper does not present a new technique to combine these embeddings.

While superficially related, our approach in `reified` descriptions is quite different. Svenningsson and Axelsson's encoding is a solution to the *expression problem* [116], while our approach is focuses on making traversals of function types easier.

9.2 Comparison with Other Work on Safe Interoperation

VeriFFI provides a roadmap to verify the correctness of foreign functions and the safety of multilingual interoperation. In this section, we discuss other systems that attempt a similar goal. The section is organized mostly in chronological order, with occasional out-of-order citations for relevant tangents.

Blume (2001) [20] presents an FFI system between Standard ML and C but aims to allow users to write their low-level functions in Standard ML, by providing a method to represent C types in ML. CertiCoq's FFI aims for the opposite; it exposes Coq types into C (not as C types but as glue functions) because we want users to write their foreign functions in C so that we can take advantage of VST's separation logic to reason about them. In possible future work, it would be possible to implement a similar approach to Blume's as a library for VeriFFI, where the C types would be reified by CompCert's Clightgen and then handled by MetaCoq to generate Coq types.

Furr and Foster (2005) [46] explore static checks to ensure that foreign functions do not violate type safety in OCaml, and in later work, Java's JNI [47, 48]. Their work presents a multilingual type system that captures how OCaml values are represented in memory, and uses data-flow analysis to check foreign function calls do not introduce type and memory safety violations. This approach falls short of our needs in two aspects: One aspect is that their approach has problems with polymorphism, yet VeriFFI deals with full dependent types. The

other aspect is that VeriFFI can express more than the type safety of the calls; users can verify that a foreign function returns the same results as its functional model. However, 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.

CakeML (2014) [60] is a compiler for a subset of Standard ML, verified in the HOL4 proof assistant. Guéneau et al. (2017) [50] 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 are available.

Œuf (2018) [74] 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.

Cogent (2022) [31] 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 nested higher-order functions, no sharing of data structures. These first-order linear programs are compiled to C code that (because they are 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.

Turcotte et al. (2019) [113] propose a framework for defining the whole language semantics of FFIs without modeling the semantics of the guest language. This framework allows them to prove the conditional soundness of the host language, stating that any unexpected runtime errors must originate from foreign functions. They also provide an implementation of such an FFI system for Lua as the host language and C as the guest language.

Patterson et al. (2022) [81] provide not a system but a recipe (following the theoretical work of Matthews and Findler (2007) [70] and Perconti and Ahmed (2014) [84]) for sound language interoperability. Their approach involves augmenting their higher-level language with a construct to embed lower-level language terms, and the lower-level language with a construct to embed higher-level language terms. These terms are given a type in the language they are embedded. Their framework stipulates a convertibility relation between the higher-level language types and lower-level language types. While not exactly the same concept, this relation resembles the heap graph predicates we presented in Chapter 4. Our `GraphPredicate` type class instances are defined over Coq types, and they can relate Coq values of a specific type to their memory representations in C.

Melocoton (2023) [51], perhaps the closest project to VeriFFI in the FFI literature, 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 [70, 84], 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 within Coq, which is already a logic and proof assistant and therefore easier to reason in, and their C programs in Coq, via the Verifiable C program logic in the Verified Software Toolchain [23], 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 multilanguage 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 [118], 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.

9.3 Comparison with Other Compilers Work

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) [16] and Elsman (2024) [43] 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 with one constructor with a 63-bit integer and another with 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.

Chataing et al. (2024) [28] present a more fine-grained method of representing algebraic data types more efficiently, allowing users to mark which constructors to represent in an unboxed way. This approach makes interoperation between user-defined types and foreign types easier, and is easier to reason about without knowing the compiler internals. While VeriFFI does not provide this feature, it is (once again) possible to implement the algebraic data types with custom representations presented in the paper as foreign types in VeriFFI.

A similar argument can be made about packed representations of algebraic data types. Chen et al. (2023) [29] present a compiler that allows the user to specify custom memory layouts for algebraic data types, and Singhal et al. (2024) [100] present another compiler that picks the most optimal representation (i.e. with minimal pointer chasing) for data types.

9.4 Comparison with Other Effect Systems

In the effectful examples in Section 8.3 and Section 8.4, we discuss the necessity of defining effects as a free monad [108] in Coq and an interpreter for it in C. This idea derives from the long-standing tradition of representing effects as monads in lazy languages, as proposed by Peyton Jones and Wadler (1993) [86] and later explained in further detail in Peyton Jones (2001) [85]. We follow the same tradition and its successors in Coq.

FreeSpec (2020, 2021) [63, 64] is a Coq framework that allows the modeling and implementation (through extraction) of effectful programs. It has a monadic representation, but it diverges from the Haskell tradition in that different kinds of effects are represented by different "interfaces," which can be composed to write a program that can have various kinds of effects at the same time, providing modularity. Interaction trees (2019) [120], another Coq framework for modeling and implementing effectful programs, equipped with an equational reasoning toolkit that FreeSpec lacks, follows the same approach (though they call different effects "events").

The way we defined monadic effects in Chapter 8 is simpler than that. What we lose in modularity, we gain in conciseness and performance. If we were to use interaction trees, our compiled program using a composition of effects would require many levels of pointer indirection for each binary sum of effects it has to unwrap. We want to avoid that, since it affects the performance of the resulting program and also does not provide the most compelling example in a dissertation. Carstens (2022) [24], for example, provides a non-modular but flat collection of effects, which eliminates this indirection.

It is possible to use interaction trees to write effectful programs with CertiCoq and VeriFFI. In fact, this approach meshes well with the VeriFFI infrastructure, as there is already prior work on using interaction trees and VST to verify effectful C programs, such as a web server [58].

9.5 Applications

The PhD thesis of Frank (2024) [45] uses CertiCoq to compile Linux kernel module component formalizations from Coq to C. His work suggests changes to CertiCoq and VeriFFI, particularly in making the compiler output and generated glue code compatible with kernel space.

Paykin (2024) [83] uses CertiCoq's FFI in the opposite direction from what this dissertation commonly uses. Instead of running C code within Coq programs, she compiles a verified set data structure backed by red-black trees into C and uses it from C++ code.

As CertiCoq currently does not produce binaries and requires a wrapper program to be written, and writing wrappers that use the Coq program's result inevitably requires the use of generated glue code, we can expect more people to use CertiCoq's FFI system in the near future.

10

Conclusion

[T]he C language continues to be a medium for much of the world's working software

—to the continued regret of many researchers.

Stephen Kell [57]

Every approach to formal verification has a different challenge that limits its adoption and production readiness. The main challenge in model checking is dealing with state-space explosion [32], while automatic program verification à la Dafny [62] can struggle with convincing the SMT solver that a logical formula is valid.

Using interactive theorem proving to verify software has challenges too. One tradition is to derive the software from the proof, as done by Coq's (or originally Nuprl's) traditional code extraction method [65, 34], or CertiCoq [4]. For programs derived in this manner, performance is the main challenge. The OCaml extraction might not reach the desired performance in performance-critical domains, since it translates the data structures in Coq directly to OCaml, and those data structures are often not built with performance considerations. When we try to use more efficient data structures, we can cause bugs in the extracted code, since we might map Coq data types onto OCaml data types in a way that breaks our carefully constructed proofs; the new data structures are unverified and they might interact with our code in unexpected ways. We do, however, get to take advantage of the optimizations in the OCaml compiler and gain performance that way.[30] On the other hand, CertiCoq,

[30] The extraction mechanism can also extract to Haskell and Scheme, for which there are mature compilers

the verified compiler from Coq to C that this dissertation is based on, is not yet as performant as other functional programming language compilers [77], as only a limited number of optimizations have been implemented so far. This is partly due to the inherent overhead of having to formally verify the correctness of each new optimization pass, which can be a significant research undertaking in itself [114, 77, 78]. Alternatively, interactive theorem provers can verify code written in other languages, rather than generating code from Coq programs. Verifiable C (in VST) [11], for verifying C code in Coq, and CFML [26, 27], for verifying OCaml code in Coq, are examples of this approach. These projects start from the original software, i.e. no software is derived from the proof, so performance of the derived software is not an issue, neither do these tools lose any guarantees later in the process. The approach we present in this dissertation is a hybrid of these two traditions that are solutions to the challenges of interactive theorem proving. With verified foreign functions, a user of our system can

- write their program in Coq in a functional style,
- prove properties about their program in Coq,
- identify performance bottlenecks of their program, such as the underlying data structures and other computationally expensive functions,
- implement these data structures and other expensive functions more efficiently in C, and use them as foreign types and foreign functions in their Coq programs,

---

like Glasgow Haskell Compiler [68] and Chez Scheme [42], so the same arguments apply.

- prove properties with VST about these foreign types and foreign functions using the automatically generated specifications, where they use their original functional implementation as the functional model for the proofs,
- and preserve the proofs about their larger program since they depend on the functional model, which the user can prove to be equivalent to the foreign implementation using the proof interface.

A user of our system can keep refining their program until they are happy with the performance of the resulting program, and they can still reason about their larger program as if it is written in a functional style, while isolating the lower-level reasoning to foreign types and foreign functions. This approach not only recuperates potential performance dips caused by the inadequacy of compiler optimizations, but also helps with theoretical limits associated with purely functional data structures [91, 17, 89, 76].

Our system makes this hybrid workflow possible by facilitating

- the implementations of foreign types and foreign functions by generating glue code,
- proofs about the foreign functions by generating function specifications and memory predicates about Coq types,
- and the preservation of proofs about the larger program by providing a mechanism to rewrite proof obligations about programs using foreign functions into equivalent programs using the functional model.

Glossary

boxed A value that is represented in memory with indirection, such as a pointer into a separate data structure that holds information. 36, 38, 39, 42, 43, 45, 102, 103, 145

boxity Whether a value is *boxed* or *unboxed*. 38, 139, 147

C heap The region in memory outside of the CertiCoq heap. Memory management of this region is the user's responsibility. 37, 47

CertiCoq heap The region in memory the CertiCoq runtime primarily operates on, and is garbage collected by the CertiCoq runtime. 34, 36, 37, 46, 76–78, 80, 81, 93, 95, 104, 105, 109, 115–117, 128–130, 138, 145, 147

closure The value form of a function in a functional programming language. It consists of a function and an environment, where the environment contains values for the free variables in the function body. 39, 40, 42, 43, 45

data constructor The only way of creating a value of an inductive type. Also simply referred to as "constructor". 11, 16, 27, 37, 39, 42, 43, 45, 58–61, 63, 65–72, 75, 76, 78–81, 92, 102, 114, 120, 122, 133, 134

deep embedding A description of an object language where the syntax tree of the object language is data in the host language. 59, 61, 133, 134

foreign function A function that is implemented in a different language than the primary programming language. 3, 10–12, 45, 58, 59, 66–68, 75, 76, 82–84, 87–97, 99–103, 106–109, 111, 112, 117, 119, 120, 126, 130, 133–137, 143–145, 147

foreign function interface (FFI) A multilanguage programming style in which one language imitates the calling conventions of the other languages so that the two languages can call each other. Consists of foreign functions and foreign types. 3, 10–12, 75, 82, 84, 95, 96, 99, 120, 122, 124, 135–137, 141

foreign type A type whose values are defined in a different language than the primary programming language. 3, 12, 41, 45, 46, 82–86, 95, 96, 99–102, 107–109, 111, 112, 116, 118, 124, 126, 139, 143–145, 147

functional model A program written in Coq that implements the same function we want to prove algorithms about. It is often easier to use a functional model as an intermediate step in proving that an imperative program satisfies a high-level specification. We can prove separately that the functional model satisfies high-level specification, and that the imperative program implements the same thing as the functional model [9, 8]. 3, 10, 12, 82–92, 94–96, 99, 101, 106–110, 116, 120, 123, 126, 131, 132, 136, 146

Gallina The language of terms in Coq. Also see Ltac and Vernacular. 14, 15, 18, 19, 27, 28, 49, 52, 55, 60, 64, 73, 79, 136, 138, 146, 148

glue code Functions that help programmers inspect, construct, or call one language's values from another language. For CertiCoq's generated glue code, see Section 3.2. 10, 11, 33, 45, 75, 76, 112, 116, 135

header (memory) An extra field where metadata about whatever comes afterwards can live. 43

host language The language the compiler is implemented in. 13, 59, 134, 146

index An argument to the inductive type constructor that can vary between data constructors of that type. 16, 27, 43, 49, 51, 84

inductive type A generalized version of algebraic data types, used for defining custom data types in Coq. 15, 16, 18, 23, 27, 35, 37–39, 45, 46, 51, 53, 61, 67, 69–71, 82, 83, 112, 120

locally nameless A representation for variables where free variables are represented as names, while bound variables are represented as de Bruijn indices. 49

Ltac The language of tactics in Coq. Explained in more detail in Subsection 2.3.2. Also see Gallina and Vernacular. 14, 15, 19, 25–29, 49, 71–74, 146, 148

meta language For a compiler context, see host language. For a metaprogramming system: the language the user would write code in, as opposed to the object language of the generated code. 27

metaprogramming Writing programs that inspect existing programs or generate new programs. 3, 19–21, 27, 58, 60, 73, 91, 134, 146

object language A language the compiler implements. Both the source and target languages are object languages in a compiler. 27, 59, 134, 146

ordinal The index of a constructor within the constructors of that inductive type with the same boxity. Not the same thing as a tag. 38, 147

parameter An argument to the inductive type constructor that cannot vary between data constructors of that type. 16, 23, 27, 43, 49, 51, 61, 69, 84, 86, 117

primitive The simplest building block of a programming language, not defined in terms of other types or values in the same language. We consider foreign functions and foreign types primitives of a language as well. 3, 19, 42, 44–46, 51–55, 58, 66, 71, 72, 147

quotation Converting a Coq expression into a syntax tree representing the expression. It is a method of reification. 19, 147

reflection The act of converting data, or a first-class object, into code. Unquotation is an example of reflection. 13, 65, 71, 134, 147, 148

reification The act of representing a piece of code as data, or as a first-class object. Quotation is an example of reification. 11, 135, 147

shallow embedding A description of an object language that uses the semantics of the host language to implement corresponding features. 61, 133, 134

source language The language a compiler translates from. 13, 147

tag The index of a constructor in the inductive type definition. Not the same thing as an ordinal. 38, 69, 102, 147

target language The language a compiler translates to. 13, 147

thread info A C `struct` that holds the necessary information for the CertiCoq runtime to work, such as the call stack, the location of the CertiCoq heap, and the location of the next allocatable spot in that heap. Often shortened to `tinfo` when used in code. 34, 36, 37, 40, 77, 78, 80, 81, 93, 95, 103–105, 114, 115, 121, 128, 129, 147

type constructor The name of a type, with no arguments applied, such as `vec` for the vector type. 16

unboxed A value that is represented without a memory indirection, as a nonpointer. 36, 38, 39, 42, 43, 45, 102, 103, 111, 139, 145

unquotation Converting a syntax tree representing the Coq expression into the actual Coq expression represented by the tree. It is a method of reflection. 20, 22, 52, 53, 147

Vernacular The language of commands in Coq. Also see Gallina and Ltac. 14, 15, 18, 21, 30, 146

Bibliography

[1] Andreas Abel, Stephan Adelsberger, and Anton Setzer. Interactive Programming in Agda–Objects and Graphical User Interfaces. *Journal of Functional Programming*, 27:e8, 2017. doi: 10.1017/S0956796816000319. URL https://doi.org/10.1017/S0956796816000319. (cited in page 120)

[2] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 2nd edition, 1996. ISBN 978-0262510875. (cited in page 30)

[3] Amal Ahmed. Verified Compilers for a Multi-Language World. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2015. URL https://web.archive.org/web/20240401021538/https://www.khoury.northeastern.edu/home/amal/papers/verifcomp.pdf. (cited in page 10)

[4] Abhishek Anand, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Weaver. CertiCoq: A Verified Compiler for Coq. In *the 3rd International Workshop on Coq for Programming Languages (CoqPL)*, 2017. URL https://web.archive.org/web/20221117172213/https://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf. (cited in pages 10, 13, 42, and 142)

[5] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards Certified Meta-Programming with Typed Template-Coq. In *International Conference on Interactive Theorem Proving*, pages 20–39. Springer, 2018. URL https://doi.org/10.1007/978-3-319-94821-8_2. (cited in pages 19 and 27)

[6] Andrew W. Appel. *Modern Compiler Implementation in ML*. Addison-Wesley, 1998. ISBN 978-0521582742. (cited in page 41)

[7] Andrew W. Appel. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, 37(2), apr 2015. ISSN 0164-0925. doi: 10.1145/2701415. URL https://doi.org/10.1145/2701415. (cited in page 31)

[8] Andrew W. Appel. Coq's Vibrant Ecosystem for Verification Engineering (invited talk). In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 2–11, 2022. URL https://doi.org/10.1145/3497775.3503951. (cited in page 146)

[9] Andrew W. Appel and Yves Bertot. C-Language Floating-Point Proofs Layered with VST and Flocq. *Journal of Formalized Reasoning*, 13(1):1–16, 2020. URL https://doi.org/10.6092/issn.1972-5787/11442. (cited in page 146)

[10] Andrew W. Appel and Marcelo J. R. Gonçalves. Hash-Consing Garbage Collection. 1993. URL https://web.archive.org/web/20220520152504/https://www.cs.princeton.edu/~appel/papers/hashgc.pdf. (cited in page 44)

[11] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, USA, 2014. ISBN 110704801X. (cited in pages 10, 76, 77, and 143)

[12] Andrew W. Appel, Lennart Beringer, and Qinxiang Cao. Verifiable C. In Benjamin C. Pierce, editor, *Software Foundations*, chapter 5. 2023. URL https://softwarefoundations.cis.upenn.edu/vc-current/index.html. (cited in pages 77 and 106)

[13] Andrew W. Appel, Lennart Beringer, Qinxiang Cao, and Josiah Dodds. *Verifiable C - Applying the Verified Software Toolchain to C Programs*, January 2023. URL https://github.com/PrincetonUniversity/VST/raw/master/doc/VC.pdf. (cited in pages 77 and 106)

[14] Aristotle and W. D. Ross. *Nicomachean Ethics*. 2016. ISBN 978-1539025580. (cited in page 100)

[15] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering Formal Metatheory. *SIGPLAN Not.*, 43(1):3–15, Jan 2008. ISSN 0362-1340. doi: 10.1145/1328897.1328443. URL https://doi.org/10.1145/1328897.1328443. (cited in page 49)

[16] Thaïs Baudon, Gabriel Radanne, and Laure Gonnord. Bit-Stealing Made Legal: Compilation for Custom Memory Representations of Algebraic Data Types. *Proc. ACM Program. Lang.*, 7(ICFP), aug 2023. doi: 10.1145/3607858. URL https://doi.org/10.1145/3607858. (cited in page 139)

[17] Amir M. Ben-Amram and Zvi Galil. On Pointers Versus Addresses. *J. ACM*, 39(3): 617–648, jul 1992. ISSN 0004-5411. doi: 10.1145/146637.146666. URL https://doi.org/10.1145/146637.146666. (cited in pages 126 and 144)

[18] Martin Berger. Foundations of Meta-Programming. https://www.cl.cam.ac.uk/events/metaprog/2016/metaprogramming-martin-berger.pdf, 2016. Accessed: 2022–03-22. (cited in page 27)

[19] Martin Berger, Laurence Tratt, and Christian Urban. Modelling Homogeneous Generative Meta-Programming. 2017. doi: 10.48550/arXiv.1602.06568. URL https://doi.org/10.48550/arXiv.1602.06568. (cited in page 19)

[20] Matthias Blume. No-Longer-Foreign: Teaching an ML Compiler to Speak C "Natively". *Electronic Notes in Theoretical Computer Science*, 59(1):36–52, 2001. ISSN 1571-0661. URL https://doi.org/10.1016/S1571-0661(05)80452-9. BABEL'01, First International Workshop on Multi-Language Infrastructure and Interoperability (Satellite Event of PLI 2001). (cited in page 135)

[21] Edwin Brady. Idris 2: Quantitative Type Theory in Practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:26, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-190-0. doi: 10.4230/LIPIcs.ECOOP.2021.9. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2021.9. (cited in page 13)

[22] Vladimir Brankov. What is Gained and Lost with 63-bit Integers? URL https://web.archive.org/web/20240228085028/https://blog.janestreet.com/what-is-gained-and-lost-with-63-bit-integers/. Accessed: 2024-08-12. (cited in page 103)

[23] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reason.*, 61(1-4):367–422, June 2018. ISSN 0168-7433. doi: 10.1007/s10817-018-9457-5. URL https://doi.org/10.1007/s10817-018-9457-5. (cited in page 138)

[24] Tim Carstens. uvrooster: A High-Performance Runtime for the CertiCoq Compiler, Built on Coq and libuv. https://github.com/intoverflow/uvrooster, 2022. (cited in page 140)

[25] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The Gentle Art of Levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, page 3–14, New York, NY, USA,

2010. Association for Computing Machinery. ISBN 9781605587943. doi: 10.1145/1863543.1863547. URL https://doi.org/10.1145/1863543.1863547. (cited in page 13)

[26] Arthur Charguéraud. Program Verification Through Characteristic Formulae. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, page 321–332, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605587943. doi: 10.1145/1863543.1863590. URL https://doi.org/10.1145/1863543.1863590. (cited in page 143)

[27] Arthur Charguéraud. Characteristic Formulae for the Verification of Imperative Programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 418–430, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450308656. doi: 10.1145/2034773.2034828. URL https://doi.org/10.1145/2034773.2034828. (cited in page 143)

[28] Nicolas Chataing, Stephen Dolan, Gabriel Scherer, and Jeremy Yallop. Unboxed Data Constructors: Or, How cpp Decides a Halting Problem. *Proc. ACM Program. Lang.*, 8(POPL), jan 2024. doi: 10.1145/3632893. URL https://doi.org/10.1145/3632893. (cited in page 139)

[29] Zilin Chen, Ambroise Lafont, Liam O'Connor, Gabriele Keller, Craig McLaughlin, Vincent Jackson, and Christine Rizkallah. Dargent: A Silver Bullet for Verified Data Layout Refinement. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi: 10.1145/3571240. URL https://doi.org/10.1145/3571240. (cited in page 139)

[30] James Cheney and Ralf Hinze. First-Class Phantom Types. Technical report, Cornell University, 2003. URL https://hdl.handle.net/1813/5614. (cited in page 15)

[31] Louis Cheung, Liam O'Connor, and Christine Rizkallah. Overcoming Restraint: Composing Verification of Foreign Functions with Cogent. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, page 13–26, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391825. doi: 10.1145/3497775.3503686. URL https://doi.org/10.1145/3497775.3503686. (cited in pages 10 and 136)

[32] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer, Berlin, Heidelberg, 2012. ISBN 978-3-642-35746-6. doi: 10.1007/978-3-642-35746-6_1. URL https://doi.org/10.1007/978-3-642-35746-6_1. (cited in page 142)

[33] Sylvain Conchon and Jean-Christophe Filliâtre. A Persistent Union-Find Data Structure. In *Proceedings of the 2007 Workshop on Workshop on ML*, ML '07, page 37–46, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936769. doi: 10.1145/1292535.1292541. URL https://doi.org/10.1145/1292535.1292541. (cited in page 130)

[34] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., USA, 1986. ISBN 0134518322. (cited in page 142)

[35] Thierry Coquand. *Une Théorie des Constructions*. PhD thesis, Université Paris 7, 1985. (cited in page 117)

[36] Thierry Coquand and Christine Paulin. Inductively Defined Types. In *International Conference on Computer Logic*, pages 50–66. Springer, 1988. URL https://doi.org/10.1007/3-540-52335-9_47. (cited in page 14)

[37] Duncan Coutts. Partial Evaluation for Domain-Specific Embedded Languages in a Higher Order Typed Language. 2004. URL https://web.archive.org/web/20170830054633/http://www.cs.ox.ac.uk/people/duncan.coutts/papers/transfer_dissertation.pdf. (cited in page 21)

[38] Nicolaas Govert de Bruijn. Lambda Calculus Notation with Nameless Dummies, A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972. URL https://doi.org/10.1016/1385-7258(72)90034-0. (cited in page 49)

[39] Maxime Dénès. Towards Primitive Data Types for Coq: 63-bits Integers and Persistent Arrays. In *Coq-5, the Coq Workshop 2013*, 2013. URL https://web.archive.org/web/20240522104209/https://coq.inria.fr/files/coq5_submission_2.pdf. (cited in page 130)

[40] Peter Dybjer. Inductive Families. *Form. Asp. Comput.*, 6(4):440–465, jul 1994. ISSN 0934-5043. doi: 10.1007/BF01211308. URL https://doi.org/10.1007/BF01211308. (cited in page 16)

[41] Peter Dybjer. A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *Journal of Symbolic Logic*, 65(2):525–549, 2000. doi: 10.2307/2586554. URL https://doi.org/10.2307/2586554. (cited in page 133)

[42] R. Kent Dybvig. The Development of Chez Scheme. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, page 1–12, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933093. doi: 10.1145/1159803.1159805. URL https://doi.org/10.1145/1159803.1159805. (cited in page 143)

[43] Martin Elsman. Double-Ended Bit-Stealing for Algebraic Data Types. *Proc. ACM Program. Lang.*, 8(ICFP), aug 2024. doi: 10.1145/3674628. URL https://doi.org/10.1145/3674628. (cited in page 139)

[44] Robert R. Fenichel and Jerome C. Yochelson. A LISP Garbage-Collector for Virtual-Memory Computer Systems. *Commun. ACM*, 12(11):611–612, nov 1969. ISSN 0001-0782. doi: 10.1145/363269.363280. URL https://doi.org/10.1145/363269.363280. (cited in page 130)

[45] Mario Frank. *On Synthesising Linux Kernel Module Components from Coq Formalisations*. PhD thesis, Universität Potsdam, 2024. URL https://doi.org/10.25932/publishup-64255. (cited in page 141)

[46] Michael Furr and Jeffrey S. Foster. Checking Type Safety of Foreign Function Calls. page 62–72, 2005. doi: 10.1145/1065010.1065019. URL https://doi.org/10.1145/1065010.1065019. (cited in page 135)

[47] Michael Furr and Jeffrey S Foster. Polymorphic Type Inference for the JNI. In *European Symposium on Programming*, pages 309–324. Springer, 2006. doi: 10.1007/11693024_21. URL https://doi.org/10.1007/11693024_21. (cited in page 135)

[48] Michael Furr and Jeffrey S. Foster. Checking Type Safety of Foreign Function Calls. *ACM Trans. Program. Lang. Syst.*, 30(4), aug 2008. ISSN 0164-0925. doi: 10.1145/1377492.1377493. URL https://doi.org/10.1145/1377492.1377493. (cited in page 135)

[49] Jeremy Gibbons and Nicolas Wu. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 339–347, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328739. doi: 10.1145/2628136.2628138. URL https://doi.org/10.1145/2628136.2628138. (cited in pages 61 and 134)

[50] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified Characteristic Formulae for CakeML. In *Programming Languages and Systems:*

26th European Symposium on Programming, ESOP 2017, April 22–29, 2017, Proceedings, pages 584–610, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. ISBN 978-3-662-54434-1. doi: 10.1007/978-3-662-54434-1_22. URL https://doi.org/10.1007/978-3-662-54434-1_22. (cited in page 136)

[51] Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C. *Proc. ACM Program. Lang.*, 7(OOPSLA2), oct 2023. doi: 10.1145/3622823. URL https://doi.org/10.1145/3622823. (cited in pages 10 and 137)

[52] Ralf Hinze and Johan Jeuring. Generic Haskell: Applications. *Generic Programming: Advanced Lectures*, pages 57–96, 2003. URL https://doi.org/10.1007/978-3-540-45191-4_2. (cited in page 58)

[53] Patrik Jansson and Johan Jeuring. PolyP—a Polytypic Programming Language Extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 470–482, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897918533. doi: 10.1145/263699.263763. URL https://doi.org/10.1145/263699.263763. (cited in page 58)

[54] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 190–203, Berlin, Heidelberg, 1985. Springer. ISBN 978-3-540-39677-2. URL https://doi.org/10.1007/3-540-15975-4_37. (cited in pages 52 and 54)

[55] Mark P Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text 1*, pages 97–136. Springer, 1995. doi: 10.1007/3-540-59451-5_4. URL https://doi.org/10.1007/3-540-59451-5_4. (cited in page 123)

[56] Akira Kawata and Atsushi Igarashi. A Dependently Typed Multi-Stage Calculus. In *Asian Symposium on Programming Languages and Systems*, pages 53–72. Springer, 2019. URL https://doi.org/10.1007/978-3-030-34175-6_4. (cited in page 21)

[57] Stephen Kell. Some Were Meant for C: The Endurance of an Unmanageable Language. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, page 229–245, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450355308. doi: 10.1145/3133850.3133867. URL https://doi.org/10.1145/3133850.3133867. (cited in page 142)

[58] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, page 234–248, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362221. doi: 10.1145/3293880.3294106. URL https://doi.org/10.1145/3293880.3294106. (cited in page 141)

[59] Joomy Korkut, Kathrin Stark, and Andrew W. Appel. A Verified Foreign Function Interface Between Coq and C. July 2024. URL https://github.com/CertiCoq/VeriFFI/blob/62535e8e239197d09563cf5161ff1457718021e6/doc/veriffi_techreport.pdf. (cited in pages 47, 68, 95, 100, and 111)

[60] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. CakeML: A Verified Implementation of ML. *ACM SIGPLAN Notices*, 49(1):179–191, 2014. doi: 10.1145/2578855.2535841. URL https://doi.org/10.1145/2578855.2535841. (cited in page 136)

[61] John Launchbury and Simon L. Peyton Jones. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 24–35, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 089791662X. doi: 10.1145/178243.178246. URL https://doi.org/10.1145/178243.178246. (cited in pages 126 and 127)

[62] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer. ISBN 978-3-642-17511-4. URL https://doi.org/10.1007/978-3-642-17511-4_20. (cited in page 142)

[63] Thomas Letan and Yann Régis-Gianas. FreeSpec: Specifying, Verifying, and Executing Impure Computations in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 32–46, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370974. doi: 10.1145/3372885.3373812. URL https://doi.org/10.1145/3372885.3373812. (cited in page 140)

[64] Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular Verification of Programs with Effects and Effects Handlers. *Form. Asp. Comput.*, 33 (1):127–150, jan 2021. ISSN 0934-5043. doi: 10.1007/s00165-020-00523-2. URL https://doi.org/10.1007/s00165-020-00523-2. (cited in page 140)

[65] Pierre Letouzey. Extraction in Coq: An Overview. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms*, pages 359–369, Berlin, Heidelberg, 2008. Springer. ISBN 978-3-540-69407-6. URL https://doi.org/10.1007/978-3-540-69407-6_39#citeas. (cited in page 142)

[66] Barbara Liskov and Stephen Zilles. Programming with Abstract Data Types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, page 50–59, New York, NY, USA, 1974. Association for Computing Machinery. ISBN 9781450378840. doi: 10.1145/800233.807045. URL https://doi.org/10.1145/800233.807045. (cited in page 101)

[67] Anil Madhavapeddy and Yaron Minsky. *Real World OCaml: Functional Programming for the Masses*. Cambridge University Press, 2nd edition, 2022. ISBN 978-1009125802. (cited in pages 41, 103, and 111)

[68] Simon Marlow and Simon Peyton Jones. The Glasgow Haskell Compiler. In *The Architecture of Open Source Applications*, volume 2. 2004. URL https://web.archive.org/web/20221212192523/https://simon.peytonjones.org/assets/pdfs/glasgow-haskell-compiler.pdf. (cited in pages 13 and 143)

[69] Nicholas D. Matsakis and Felix S. Klock. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450332170. doi: 10.1145/2663171.2663188. URL https://doi.org/10.1145/2663171.2663188. (cited in page 13)

[70] Jacob Matthews and Robert Bruce Findler. Operational Semantics for Multi-language Programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, page 3–10, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 1595935754. doi: 10.1145/1190216.1190220. URL https://doi.org/10.1145/1190216.1190220. (cited in page 137)

[71] Jacob Burton Matthews. *The Meaning of Multilanguage Programs*. PhD thesis, The University of Chicago, 2008. URL https://web.archive.org/web/20240331001311/https://plt.cs.northwestern.edu/matthews-phd.pdf. (cited in page 9)

[72] Conor McBride. Outrageous but Meaningful Coincidences: Dependent Type-Safe Syntax and Evaluation. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*, WGP '10, page 1–12, New York, NY, USA, 2010. Association

for Computing Machinery. ISBN 9781450302517. doi: 10.1145/1863495.1863497. URL https://doi.org/10.1145/1863495.1863497. (cited in pages 133 and 134)

[73] Wolfgang Meier, Jean Pichon-Pharabod, and Bas Spitters. CertiCoq-Wasm: Verified Compilation from Coq to WebAssembly. In *the 10th International Workshop on Coq for Programming Languages (CoqPL)*, 2023. URL https://web.archive.org/web/20240401020120/https://womeier.de/files/certicoqwasm-coqpl24-abstract.pdf. (cited in page 11)

[74] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. Œuf: Minimizing the Coq Extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 172–185, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355865. doi: 10.1145/3167089. URL https://doi.org/10.1145/3167089. (cited in page 136)

[75] Randall Munroe. XKCD: Haskell. URL https://web.archive.org/web/20240727051624/https://xkcd.com/1312/. Online; accessed 25-August-2024. (cited in page 117)

[76] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999. ISBN 978-0521663502. (cited in pages 126 and 144)

[77] Zoe Paraskevopoulou. *Verified Optimizations for Functional Languages*. PhD thesis, Princeton University, 2020. URL https://web.archive.org/web/20230706171148/https://www.cs.princeton.edu/techreports/2020/006.pdf. (cited in pages 34, 42, and 143)

[78] Zoe Paraskevopoulou, John M. Li, and Andrew W. Appel. Compositional Optimizations for CertiCoq. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021. doi: 10.1145/3473591. URL https://doi.org/10.1145/3473591. (cited in page 143)

[79] Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. FunTAL: Reasonably Mixing a Functional Language with Assembly. *SIGPLAN Not.*, 52(6): 495–509, jun 2017. ISSN 0362-1340. doi: 10.1145/3140587.3062347. URL https://doi.org/10.1145/3140587.3062347. (cited in page 10)

[80] Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. Semantic Soundness for Language Interoperability. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 609–624, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523703. URL https://doi.org/10.1145/3519939.3523703. (cited in page 10)

[81] Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. Semantic Soundness for Language Interoperability. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 609–624, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523703. URL https://doi.org/10.1145/3519939.3523703. (cited in page 137)

[82] Daniel Patterson, Andrew Wagner, and Amal Ahmed. Semantic Encapsulation Using Linking Types. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2023, page 14–28, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702990. doi: 10.1145/3609027.3609405. URL https://doi.org/10.1145/3609027.3609405. (cited in page 10)

[83] Jennifer Paykin. certicoq-set-library: Using CertiCoq to implement a library for formally verified set in C++. https://github.com/jpaykin/certicoq-set-library, 2024. (cited in page 141)

[84] James T Perconti and Amal Ahmed. 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*, pages 128–148. Springer, 2014. doi: 10.1007/978-3-642-54833-8_8. URL https://doi.org/10.1007/978-3-642-54833-8_8. (cited in page 137)

[85] Simon Peyton Jones. Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-language Calls in Haskell. *NATO Science Series Sub Series III Computer and Systems Sciences*, 180:47–96, 2001. URL https://web.archive.org/web/20221212192523/https://simon.peytonjones.org/assets/pdfs/tackling-awkward-squad.pdf. (cited in page 140)

[86] Simon L. Peyton Jones and Philip Wadler. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, page 71–84, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915607. doi: 10.1145/158511.158524. URL https://doi.org/10.1145/158511.158524. (cited in pages 123 and 140)

[87] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993. URL https://web.archive.org/web/20220121100953/https://www.microsoft.com/en-us/research/wp-content/uploads/1993/03/grasp-jfit.pdf. (cited in page 13)

[88] Frank Pfenning and Conal Elliott. Higher-Order Abstract Syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, page 199–208, New York, NY, USA, 1988. Association for Computing Machinery. ISBN 0897912691. doi: 10.1145/53990.54010. URL https://doi.org/10.1145/53990.54010. (cited in page 59)

[89] Nicholas Pippenger. Pure versus Impure Lisp. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 104–109, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917693. doi: 10.1145/237721.237741. URL https://doi.org/10.1145/237721.237741. (cited in pages 126 and 144)

[90] Plato. *The Republic*. Cambridge Texts in the History of Political Thought. Cambridge University Press, Cambridge, England, September 2000. ISBN 978-0521484435. (cited in page 82)

[91] Carl G. Ponder, Patrick McGeer, and Anthony P-C. Ng. Are Applicative Languages Inefficient? *SIGPLAN Not.*, 23(6):135–139, jun 1988. ISSN 0362-1340. doi: 10.1145/44546.44559. URL https://doi.org/10.1145/44546.44559. (cited in pages 126 and 144)

[92] Jacob Prinz, G. A. Kavvos, and Leonidas Lampropoulos. Deeper Shallow Embeddings. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-252-5. URL https://doi.org/10.4230/LIPIcs.ITP.2022.28. (cited in page 134)

[93] Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, page 127–136, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450301541. doi: 10.1145/1868294.1868314. URL https://doi.org/10.1145/1868294.1868314. (cited in page 21)

[94] Kazuhiko Sakaguchi. Program Extraction for Mutable Arrays. *Science of Computer Programming*, 191:102372, 2020. ISSN 0167-6423. doi: 10.1016/j.scico.2019.102372. URL https://doi.org/10.1016/j.scico.2019.102372. (cited in page 127)

[95] Patrick M. Sansom and Simon L. Peyton Jones. Generational Garbage Collection for Haskell. In *Proceedings of the Conference on Functional Programming Languages*

and Computer Architecture, FPCA '93, page 106–116, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 089791595X. doi: 10.1145/165180.165195. URL https://doi.org/10.1145/165180.165195. (cited in page 130)

[96] Olivier Savary Bélanger. *Verified Extraction for Coq*. PhD thesis, Princeton University, 2019. URL https://web.archive.org/web/20220521073059/https://www.cs.princeton.edu/techreports/2019/011.pdf. (cited in pages 11, 34, and 42)

[97] Olivier Savary Bélanger, Matthew Z Weaver, and Andrew W. Appel. Certified Code Generation from CPS to C. 2019. URL https://web.archive.org/web/20230322153846/https://www.cs.princeton.edu/~appel/papers/CPStoC.pdf. (cited in page 34)

[98] Tim Sheard. Accomplishments and Research Challenges in Meta-Programming. *SAIG*, 2196(3):2–44, 2001. URL https://doi.org/10.1007/3-540-44806-3_2. (cited in page 21)

[99] Tim Sheard and Simon Peyton Jones. Template Meta-Programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, Dec 2002. ISSN 0362-1340. doi: 10.1145/636517.636528. URL https://doi.org/10.1145/636517.636528. (cited in pages 20, 27, and 58)

[100] Vidush Singhal, Chaitanya Koparkar, Joseph Zullo, Artem Pelenitsyn, Michael Vollmer, Mike Rainey, Ryan Newton, and Milind Kulkarni. Optimizing Layout of Recursive Datatypes with Marmoset: Or, Algorithms {+} Data Layouts {=} Efficient Programs. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:28, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-341-6. doi: 10.4230/LIPIcs.ECOOP.2024.38. URL https://doi.org/10.4230/LIPIcs.ECOOP.2024.38. (cited in page 139)

[101] Matthieu Sozeau and Cyprien Mangin. Equations Reloaded: High-Level Dependently-Typed Functional Programming and Proving in Coq. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019. doi: 10.1145/3341690. URL https://doi.org/10.1145/3341690. (cited in pages 64 and 79)

[102] Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In *International Conference on Theorem Proving in Higher Order Logics*, pages 278–293. Springer, 2008. URL https://doi.org/10.1007/978-3-540-71067-7_23. (cited in pages 23 and 46)

[103] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The

MetaCoq Project. *Journal of Automated Reasoning*, 64(5):947–999, 2020. URL https://doi.org/10.1007/s10817-019-09540-0. (cited in pages 19 and 50)

[104] Richard M Stallman and Zachary Weinberg. The C Preprocessor. *Free Software Foundation*, page 16, 1987. URL https://web.archive.org/web/20240210213047/https://gcc.gnu.org/onlinedocs/cpp.pdf. (cited in page 19)

[105] Guy L. Steele. Growing a Language. In *Addendum to the 1998 Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (Addendum)*, OOPSLA '98 Addendum, page 0.01–A1, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 1581132867. doi: 10.1145/346852.346922. URL https://doi.org/10.1145/346852.346922. (cited in page 67)

[106] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. A Practical Unification Of Multi-Stage Programming and Macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2018, page 14–27, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360456. doi: 10.1145/3278122.3278139. URL https://doi.org/10.1145/3278122.3278139. (cited in page 21)

[107] Josef Svenningsson and Emil Axelsson. Combining Deep and Shallow Embedding for EDSL. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming*, pages 21–36, Berlin, Heidelberg, 2013. Springer. ISBN 978-3-642-40447-4. URL https://doi.org/10.1007/978-3-642-40447-4_2. (cited in pages 134 and 135)

[108] Wouter Swierstra. Data Types à la Carte. *Journal of Functional Programming*, 18(4): 423–436, 2008. (cited in page 140)

[109] Wouter Swierstra and Thorsten Altenkirch. Beauty in the Beast. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, page 25–36, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936745. doi: 10.1145/1291201.1291206. URL https://doi.org/10.1145/1291201.1291206. (cited in pages 124 and 132)

[110] Walid Taha. A Gentle Introduction to Multi-Stage Programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, Berlin, Heidelberg, Berlin, Heidelberg, 2004. URL https://doi.org/10.1007/978-3-540-25935-0_3. (cited in page 21)

[111] Walid Taha and Tim Sheard. Multi-Stage Programming with Explicit Annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '97, page 203–217, New York, NY,

USA, 1997. Association for Computing Machinery. ISBN 0897919173. doi: 10.1145/258993.259019. URL https://doi.org/10.1145/258993.259019. (cited in pages 20 and 27)

[112] Kumiko Tanaka-Ishii. *Semiotics of Programming*. Cambridge University Press, 2010. ISBN 978-0521736275. (cited in page 58)

[113] Alexi Turcotte, Ellen Arteca, and Gregor Richards. Reasoning About Foreign Function Interfaces Without Modelling the Foreign Language. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:32, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-111-5. URL https://doi.org/10.4230/LIPIcs.ECOOP.2019.16. (cited in page 137)

[114] Katja Vassilev. Specification of the Dead Parameter Elimination Optimization of the CertiCoq Compiler. Senior thesis, Princeton University, 2019. URL http://arks.princeton.edu/ark:/88435/dsp015m60qv74k. (cited in page 143)

[115] Philip Wadler. The Essence of Functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, page 1–14, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 0897914538. doi: 10.1145/143165.143169. URL https://doi.org/10.1145/143165.143169. (cited in page 122)

[116] Philip Wadler. The Expression Problem, November 1998. URL https://web.archive.org/web/20240510200911/http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt. Accessed: 2024-08-11. (cited in page 135)

[117] Shengyi Wang. *Mechanized Verification of Graph-Manipulating Programs*. PhD thesis, National University of Singapore, 2020. URL https://web.archive.org/web/20231212164913/https://www.cs.princeton.edu/~shengyiw/resource/thesis.pdf. (cited in page 45)

[118] Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. Certifying Graph-Manipulating C Programs via Localizations within Data Structures. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019. doi: 10.1145/3360597. URL https://doi.org/10.1145/3360597. (cited in pages 45 and 138)

[119] Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris 7, 1994. (cited in page 117)

[120] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi: 10.1145/3371119. URL https://doi.org/10.1145/3371119. (cited in page 140)

[121] Ningning Xie, Matthew Pickering, Andres Löh, Nicolas Wu, Jeremy Yallop, and Meng Wang. Staging with Class: A Specification for Typed Template Haskell. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022. doi: 10.1145/3498723. URL https://doi.org/10.1145/3498723. (cited in page 21)