Œuf: Minimizing the Coq Extraction TCB

Eric Mullen
University of Washington, USA

Stuart Pernsteiner
University of Washington, USA

James R. Wilcox
University of Washington, USA

Zachary Tatlock
University of Washington, USA

Dan Grossman
University of Washington, USA

Abstract

Verifying systems by implementing them in the programming language of a proof assistant (*e.g.*, Gallina for Coq) lets us directly leverage the full power of the proof assistant for verifying the system. But, to execute such an implementation requires *extraction*, a large complicated process that is in the trusted computing base (TCB).

This paper presents Œuf, a verified compiler from a subset of Gallina to assembly. Œuf's correctness theorem ensures that compilation preserves the semantics of the source Gallina program. We describe how Œuf's specification can be used as a foreign function interface to reason about the interaction between compiled Gallina programs and surrounding shim code. Additionally, Œuf maintains a small TCB for its front-end by reflecting Gallina programs to Œuf source and automatically ensuring equivalence using computational denotation. This design enabled us to implement some early compiler passes (e.g., lambda lifting) in the untrusted reflection and ensure their correctness via translation validation. To evaluate Œuf, we compile Appel's SHA256 specification from Gallina to x86 and write a shim for the generated code, yielding a verified `sha256sum` implementation with a small TCB.

CCS Concepts • **Theory of computation → Logic and verification**; **Denotational semantics**; **Program verification**; *Interactive proof systems*; • **Software and its engineering → Software verification**; **Formal software verification**; *Formal language definitions*; *Compilers*;

Keywords Coq, Compilers, Formal Verification, Verified Systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CPP'18, January 8–9, 2018, Los Angeles, CA, USA

© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.

ACM ISBN 978-1-4503-5586-5/18/01…$15.00
https://doi.org/10.1145/3167089

ACM Reference Format:
Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. 2018. Œuf: Minimizing the Coq Extraction TCB. In *Proceedings of 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3167089

1 Introduction

In the past decade, a wide range of verified systems have been implemented in Gallina, Coq's functional programming language [8, 19, 20, 25, 26, 44, 45]. This approach eases verification because Coq provides extensive built-in support for reasoning about Gallina programs. Unfortunately, building a system directly in Gallina typically incurs a substantial trusted computing base (TCB) to actually extract, compile, and run the system and connect it to the effectful real world via untrusted shim code.

This paper presents Œuf, a methodology for verified compiler design which allows extraction of code written in Gallina to executable x86 code with minimal TCB. More precisely,

1. Œuf's compilation process uses computational denotation to automatically prove that the internal representation of the program is equivalent to the original Gallina;
2. Œuf guarantees that the generated assembly program is equivalent to the source program, thus the user can reason about the machine-level code in terms of the source;
3. Œuf-compiled executables depend on a much smaller unverified runtime and come with an interface theorem that enables reasoning about their interactions with C programs.

While our prototype compiler itself is currently limited, we believe the design ideas that enable these features could easily integrate into other tools.

Œuf assumes a smaller TCB than typical verified compilers using two key techniques. First, in the front end, Œuf avoids relying on an unverified parser or grammar definition by using translation validation [37, 38]: the Œuf compiler reflects Gallina programs into a deeply embedded expression language, then generates a proof that the computational denotation of the reflected expression is equivalent to the original function. This validation theorem, like that of CakeML [33], guarantees that the program Œuf compiles really is the program provided by the user. Second, at the back end, Œuf

provides a formal specification of its application binary interface (ABI) for invoking Œuf-compiled Gallina functions from supporting C code. Having a formal ABI definition allows reasoning about the behavior of the "shim" that connects the pure extracted Gallina code with the effectful real world, whereas current extraction techniques require the shim to be trusted.

The combination of translation-validated reflection, the ABI specification, and the Œuf compiler's main correctness theorem allows users to prove correctness of an overall system built with both Gallina and C-language components. Œuf is designed for use with shims written in C, and these shims themselves can be verified against the CompCert C semantics (e.g., using VST [4]) and then compiled with CompCert, a verified compiler. When reasoning about calls from the C shim into a component compiled from Gallina, the user can simply invoke Œuf's high-level guarantee that the machine code is a correct refinement of the source Gallina program, which is composed from the reflection, compiler correctness, and ABI specification components. Specifically, Œuf defines a relation between high- and low-level representations of values, and provides a proof that applying related functions to related arguments produces related results. The developer can then *reason about the machine code in terms of the source Gallina program* using all the usual conveniences of Coq.

In summary, Œuf contributes a methodology for verified compilation for systems code implemented in Coq, enabled by:

1. A technique to automatically validate the reflection of Gallina *a posteriori* using computational denotation, and related design decisions to simplify early compiler passes (e.g., lambda lifting) by extending this validation to also automatically establish correctness (Section 3);
2. A correctness theorem which supports reasoning about interactions between compiled Gallina code and the outside world (Section 5); and,
3. The prototype Œuf verified compiler which translates a small yet useful subset of Gallina to assembly via CompCert (Section 4), along with a case study applying Œuf to representative systems code (Section 7).

2 Overview

Figure 1 shows an overview of the Œuf compilation process. The user provides a Gallina function, such as the max function in Figure 2, which computes the maximum of two natural numbers. The user also writes C code as a "shim" to connect their pure Gallina functions with the effectful real world by performing I/O and other operations not possible in Gallina. Figure 3 shows C code for a program that reads two unsigned integers and computes their maximum by calling max. Given these two inputs, Œuf compiles both

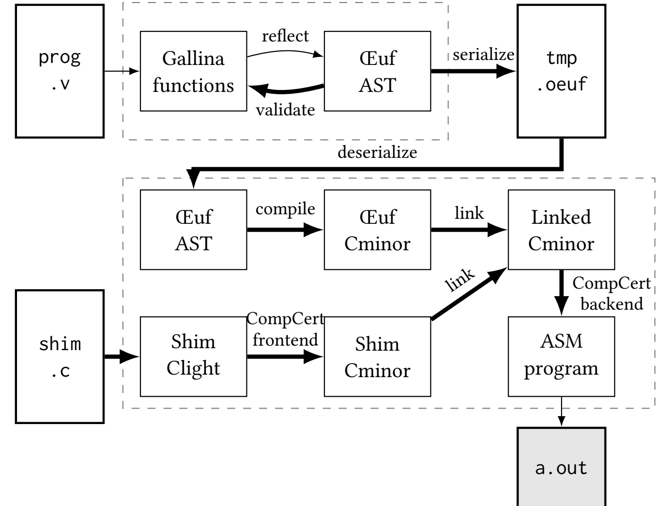

Figure 1. The Œuf compilation workflow. Arrows in bold indicate formally-verified operations. The output is a.out, indicated with gray.

```
Fixpoint max_orig (n m : nat) : nat :=
    match n with
    | 0     => m
    | S n' => match m with
              | 0     => n
              | S m' => S (max n' m')

Definition max (n m : nat) : nat :=
    nat_rect (fun _ => nat -> nat)
        (fun m => m)
        (fun n' rec m =>
            nat_rect (fun _ => nat)
                (S n')
                (fun m' _ => S (rec m'))
            m)
    n m
```

Figure 2. A Gallina function for computing the maximum of two natural numbers and its equivalent written in terms of nat_rect.

to a common representation, links them, and invokes the CompCert backend to produce an executable.

To begin the compilation process, the user must ensure that their function is written in a form that Œuf can reflect. The design of Œuf requires that the input function be written in an "ML-like" subset of Gallina, free of dependent types. This is required for compatibility with the computational denotation described in Section 3.3. Our prototype implementation further requires that all recursion and pattern matching be expressed using recursion schemes (also called *eliminators*). For a complete description of Œuf's input language and the rationale for its restrictions, see Section 3.

The function max in Figure 2 is already in the required format for reflection. A more idiomatic version of max, which an actual user would likely write first, is also given as max_orig in that figure. Since our prototype implementation does not automatically convert programs to use eliminators, the user

```
oeuf_function max;

int main() {
    unsigned int n, m, result;
    coq_nat *n_, *m_, *result_;
    scanf("%u %u", &n, &m);
    n_ = nat_of_uint(n);
    m_ = nat_of_uint(m);
    result_ = OEUF_CALL(max, n_, m_);
    result = uint_of_nat(result_);
    printf("%u\n", result);
    return 0;
}
```

Figure 3. A C program that calls the max function defined in Figure 2.

```
Oeuf Reflect max As max_ast.
Check max_ast : compilation_unit.

Lemma max_ast_validate : denote max_ast = max.
Proof. reflexivity. Qed.

Oeuf Eval lazy Then Write To File "max.oeuf"
    (compilation_unit.print max_ast).
```

Figure 4. The Coq Vernacular commands to reflect, denote, verify the roundtrip, and write the reflection to a file of the max function.

must manually perform the translation and prove that the converted function is equivalent to the original; as both are ordinary Gallina functions, this proof can be carried out in the standard fashion.

Given a Gallina function in the proper form, the first step in the Œuf compilation process is to reflect the function into an abstract syntax tree (AST) suitable for processing by the Œuf compiler. Œuf's reflection procedure is implemented as a Coq plugin. Invoking the reflection procedure, as shown in Figure 4, generates a value of type `compilation_unit` that contains a set of functions ready for processing by the rest of the Œuf compiler.

The Œuf reflection procedure need not be trusted: it is verified using translation validation [37, 38]. After constructing the Œuf compilation unit, the user invokes a script to generate a proof of equivalence between the denotation of the reflected function and the original function the user intended to compile. For simple functions, such as `max`, the proof is trivial, as shown in Figure 4; more complex functions require a more complex proof structure to work around performance problems in the Coq proof checker. The denotation function also appears in Œuf's correctness theorem, so the validation theorem helps connect the input Gallina function to the compiled code.

Up to this point, all steps of the Œuf workflow have taken place within an interactive Coq session. However, because the Œuf compiler integrates with CompCert, which relies on unverified OCaml code, the Œuf compiler must be executed as a separate binary, produced using Coq's existing extraction process. Using extraction also provides better performance than running inside the Coq interpreter for the Œuf compiler's translation passes.

To transfer the Œuf compilation unit from the interactive session to the Œuf compiler, the user invokes a serialization procedure to convert the compilation unit to a string, then writes the string to a temporary file (labeled `tmp.oeuf` in Figure 1). The code for this step is shown in Figure 4.

Once the compilation unit has been written to disk, the user can invoke the Œuf compiler binary to complete the compilation process. The compiler takes the compilation unit and shim source as input. It then deserializes the compilation unit. The deserializer is verified to be the inverse of the serializer: $\forall x, \text{deserialize}(\text{serialize}(x)) = x$. Next, the compiler translates the compilation unit to Cminor, a simplified C-like intermediate language used in CompCert, using the process described in Section 4. It similarly compiles the shim from its C source code to Cminor and links the two build products. The resulting complete Cminor program is translated to object code with the existing CompCert backend.

2.1 Guarantee

Œuf's correctness theorem guarantees that valid calls to Œuf-compiled functions will behave in a manner equivalent to the user's original Gallina implementation. A valid call is one that follows the Œuf application binary interface, which defines a relation $\sim$ between Gallina values and assembly-level memory representations of those values. Roughly stated, Œuf guarantees that "applying related functions to related arguments produces related results." That is, for any Gallina function f and argument x, and any assembly-level function f' and argument x', if $f \sim f'$ and $x \sim x'$, then $f(x) \sim f'(x')$.

This theorem is established in several steps.

1. First, when Œuf reflects the original Gallina functions to produce the input AST, it also generates a proof equating the denotation of the AST to the original function, as discussed above.
2. Second, Œuf comes with a proof that the denotational and operational semantics of its source language are compatible. If the operational semantics permit a step from state s to state s' (written $s \rightarrow s'$), then the denotations of the two states are equivalent ($[\![s]\!] = [\![s']\!]$).
3. Third, Œuf uses a verified serializer and deserializer when writing the input AST to the temporary file and reading it back. This ensures that the same AST that was reflected interactively is processed by the Œuf backend.
4. Fourth, the Œuf compiler comes with a proof of simulation between an operational semantics for the Œuf input language and the operational semantics of Cminor.
5. Finally, Œuf relies on CompCert's own simulation proof to connect the behavior of the Cminor program to that of the final assembly program.

The complete details of Œuf's correctness proof are covered in Section 5.

3 Front End

We begin with a discussion of the design constraints on Œuf's source language, emphasizing the requirement that the language have a computational denotational semantics in Coq. Then, we present the syntax of the source language, which is a lambda-lifted simply typed lambda calculus over a set of predetermined base types. Finally, we describe how to denote and reflect the source language, and emphasize how our design simplified early compiler passes by enabling us to implement them in the untrusted reflection and automatically ensure their correctness *a posteriori* using the computational denotation.

3.1 Design Constraints

Œuf's source language must satisfy several constraints, some of which are in tension.

Reflection. It must be possible to *reflect* Gallina terms into corresponding ASTs. Reflection is the first step of the compiler pipeline. However, we need not trust reflection directly, as we can check its result later using denotation.

Denotation. The language must support a *computational denotation*, *i.e.*, a function defined in Coq that converts the syntax of an expression into the corresponding Coq term. This is essential in order to formally connect verified Gallina programs with their representation as an AST. The denotation need not be trusted, since it is verified against the operational semantics of the input language.

Expressiveness. We would like to maximize the subset of Gallina programs we can compile. This is in direct tension with the requirement for a computational denotational semantics. While there exist many different possible tradeoffs, we believe the input language for Œuf is reasonably expressive.

Tractable Compilation. Lastly, it should be possible to write and verify (in Coq) a compiler from the source language to a low-level language.

Since our Œuf prototype compiler is a mainly intended as a proof-of-concept to explore validated reflection and ABI reasoning, its source language satisfies these constraints somewhat conservatively, sacrificing expressiveness for simplicity. For example, the choice to restrict base types to a predefined set means that the language definition and denotation need not handle user-defined types, simplifying the design and implementation at the cost of restricting user programs.

The process of translating a Gallina program into the supported language subset is a largely mechanical process, but it has not been automated in the current implementation of Œuf. In practice, these restrictions do not limit expressiveness for systems implemented in Gallina since users can easily prove the idiomatic version of their Gallina code equivalent to the corresponding eliminator-based version.

e	$\in$	*expr*	(Expressions)
e	::=	x	(variable)
	|	$e\ e$	(function application)
	|	$C\ e^*$	(data constructor)
	|	$E\ e^*\ e$	(data eliminator)
	|	$f\ e^*$	(closure creator)
C	$\in$	*constr*	(Constructors)
C	::=	true | cons	
	|	...	
τ	$\in$	*type*	(Types)
τ	::=	$b \mid \tau \to \tau$	
f	$\in$	*func*	(Function name)
b	$\in$	*base*	(Base types)
b	::=	bool	
	|	list b	
	|	...	
E	$\in$	*elim*	(Eliminators)
E	::=	bool_elim	
	|	list_elim	
	|	...	
g	$\in$	$f \to e$	(Global environment)

Figure 5. Syntax of the Œuf prototype compiler's source language.

3.2 Syntax

Figure 5 presents the syntax of the Œuf prototype compiler's source language, which is a lambda-lifted simply typed lambda calculus over a particular set of base types. Variables and function application are standard.

The next two expression forms are for manipulating data of base type. Base types consist of a pre-defined set of types, including common types in the Coq standard library, such as booleans, lists, natural numbers, and so on. The Œuf source language supports parameterized types, such as lists, by making the definition of base types recursive; however, since the recursion is not mutual with the definition of types, thus function types as arguments to type constructors are not supported. We believe this limitation could be lifted with a little engineering effort, but it simplifies the implementation.

In the expression grammar, a constructor C is a data constructor for one of the base types, *e.g.*, true for bool. A constructor has some number of argument subexpressions, written e^* in the figure.

An eliminator E represents a structural recursion principle for each base type. An eliminator for a particular base type takes a list of expressions representing "cases", one for each constructor of the base type, and then takes one last expression, which is the "target" of the elimination. The

Œuf prototype's source language encodes all recursive functions explicitly in terms of eliminators instead of a fixpoint construct. Eliminators were chosen instead of recursive functions due to their ease of denotation.

The last expression form is closure creation. A closure creation expression consists of a function name f and list of subexpressions, whose values will serve as the closure's environment. At run time, a closure creation expression evaluates each of its subexpressions and produces a closure value that contains the function name and the resulting values.

Since the Œuf prototype's source language has explicit closures, a top-level program is not simply an expression; instead, it is a global environment containing a mapping from function names to function definitions. Each function definition consists of a single expression that makes up the function's body. The body expression is not closed: it has one free variable corresponding to the function's sole argument, plus zero or more free variables provided by the closure environment.

3.3 Denotational Semantics

The key property of the source language is the ability to write a computational denotation function. The denotation function is a normal Coq function which takes the syntax of an expression and returns the corresponding Coq term. We use the standard technique of typed dependent de Bruijn indices to represent the syntax of binding (see, for example, Chlipala's CPDT [11]). In this style, the syntax of expressions carries the relevant typing information, which ensures that only well-typed expressions can be created. The syntax of types follows Figure 5 directly.

Expressions are represented as an inductive type family indexed on the types of free variables and the return type. Our representation of variables and application is standard and follows CPDT. We make modest extensions to standard techniques to support constructors, eliminators, and closures. We represent constructor names and eliminator names as elements of a type family indexed by their arguments and return types. Constructor and eliminator expressions use dependent types to enforce correct argument types.

A closure is represented with a reference to the body, and a list of expressions whose values will make up the closure's environment. An invariant of closure expressions is that the types of values in the environment match those of the free variables in the body of the closure.

These extensions to standard typed de Bruijn techniques pose no fundamental difficulties in the computational denotation function. First, each type is denoted to a corresponding Coq type; for example, if the syntax for the boolean type was `ty_bool`, the type denotation function would map this to Coq's `bool` type. Similarly, ASTs which represent function types denote to native Coq function types. Expressions are denoted by a function mapping expression syntax into the corresponding Coq term. The expression denotation function has a return type which uses the type denotation function, which captures the idea that an expression in the object language should correspond to a Coq term with a corresponding type. Expression denotation takes additional arguments to represent the environment in which to denote, including the available top-level functions and the values for free variables. The denotations of variables and application are standard. The cases for constructors and eliminators branch on the particular constructor or eliminator used and return the corresponding Coq constructor or eliminator.

The denotation of closures requires some care, since Gallina has no explicit notion of closures. We handle this by giving each denoted function an additional initial argument which packages up its free variables. Then, when constructing a closure, we partially apply the denoted function to a structure containing the denotation of the closure's environment expressions. To access a free variable, the body of a closure indexes into this structure with the corresponding variable name.

3.4 Operational Semantics

In addition to the denotational semantics, we also give an *operational* semantics for the source language. These and additional operational semantics for each intermediate language are used to verify semantics preservation later for each pass of the compiler. In order to connect the compiler guarantees to the original Coq term, we prove a theorem relating the denotational and operational semantics, namely that each step of the operational semantics ($\rightarrow$) preserves the denotation ($[\![\cdot]\!]$): $\forall\ s\ s',\quad s \rightarrow s' \Rightarrow [\![s]\!] = [\![s']\!]$. Together with the CompCert compiler's correctness theorem, this guarantees that values computed at the machine level correspond to the original Coq terms.

Our semantics handles local variables using an explicit local environment, which we chose because it simplifies later reasoning. This decision also makes the handling of closures fairly straightforward: calling a closure amounts to replacing the expression being evaluated with the body of the function and replacing the current local environment with a combination of the argument value and the closure's environment values. The only wrinkle is that retrieving a function body produces an expression that is valid in a restricted global environment, containing only the functions defined prior to the function of interest, whereas evaluation occurs only in the full environment. We handle this using a computational version of the standard weakening lemma, a function of type `expr G L ty -> expr (G' ++ G) L ty` that converts an expression into an equivalent one that is valid in an extended global static environment.

3.5 Reflection

Another important aspect of the source language is that it is possible to *reflect* a Coq term into the corresponding syntax. Of course, reflection will only succeed for Coq terms in the

image of the denotation function; Coq terms that use features not supported by the source language (*e.g.*, dependent types) will simply fail to reflect. We implemented the reflection procedure for the source language as a custom OCaml plugin to Coq. The reflection procedure is relatively straightforward and uses standard techniques.

The most important aspect of the reflection procedure is that we *need not trust it.* After reflection, the generated syntax can be checked by denoting it and ensuring that it is definitionally equal to the original Coq term. This is essentially a form of translation validation. This design decision significantly eases the proof burden for the early passes of the compiler. Performing lambda lifting during reflection allows us to avoid implementing a complex verified lambda-lifting pass in Coq. We are also able to extend the reflection with support for monomorphizing invocations of polymorphic functions, allowing our Œuf prototype compiler to support most instances of ML-style polymorphism with no modification to the source language and minimal verification effort.

The fact that reflection is not trusted is a key difference between our approach and related work, but it comes at the cost of supporting a restricted subset of Gallina. Even aside from the decisions we imposed to simplify the development of our prototype, the design of Œuf requires that the source language support computational denotation into Gallina imposes significant restrictions—in particular, it remains an open research question whether it is possible to denote a general Gallina AST into a Gallina term, all within Gallina.

However, we believe that with some engineering effort, it will be possible to extend the source language to include user defined datatypes, pattern matching, and recursion, which will capture typical verified systems implementations.

4 Compiler Internals

The bulk of our effort developing the Œuf prototype compiler was implementing and verifying 45 translation passes from our front end down to CompCert Cminor. To ease verification effort, we followed the standard practice of building many passes between closely related languages to simplify the refinement proof between each stage [25, 41].

4.1 From Gallina to Register Machine

Type Erasure Œuf first erases type information. The input language uses a dependently-typed program representation that admits only well-typed terms, which is necessary to write the denotation function used for validating the reflection procedure Section 3. Because the Œuf compiler includes simulation proofs for every translation pass, the untyped intermediate programs are guaranteed to behave identically to the well-typed input program.

Eliminators Next the Œuf compiler contains 6 passes to translate inductive datatype eliminators to recursive functions. These passes collectively replace each eliminator expression with a call to a new top-level function that performs a `switch` on the argument's constructor tag and executes the appropriate arm. If a constructor contains further values of the same argument type, the eliminator function will generate recursive calls to itself, executing the same pattern-matching code on the structurally smaller values.

Stack Machine The Œuf compiler then converts nested expressions to flat sequences of stack machine operations. We use this stack machine language as an intermediate step, allowing us to separate flattening of the program from assigning of names to temporary values. There is generally one stack machine operation for each expression type, whose behavior is to pop values corresponding to any subexpressions, perform the same operation as the original expression, and push the result back onto the stack. Execution of this stack machine representation directly mirrors the order of evaluation of the original expression, but makes explicit the creation and use of intermediate values.

Register Machine Once manipulation of intermediate values has been made explicit, the Œuf compiler converts the program to a high-level register machine program that gives an explicit name to each value. The set of operations for the register machine is nearly identical to that of the stack machine, except that each operation now explicitly reads and writes named registers instead of manipulating an implicit stack. The pattern of register accesses corresponds closely to C-level local variable accesses, except that programs still manipulate abstract "closure" and "constructed" values rather than machine-level integers and pointers.

4.2 Lowering towards Cminor

At this point the program is in a language which computes using closures and constructors. However, to meet with any language from the CompCert pipeline, the program must use integers and pointers, i.e. C level values.

Switch Statements To transform a program from a high level functional language to a lower level procedural language, the Œuf compiler generates C-style `switch` statements, with explicit breaks and implicit fall through. In earlier phases, a single step takes a switch statement into the corresponding correct case; in this phase the transition may take multiple steps in order to "step over" all non-chosen cases. While this is conceptually straightforward, the proof was quite subtle, due to the arbitrary number of steps the target program may take, as well as the relatively strong invariants which must be true about the current program continuation during all of these steps.

Heap Introduction The Œuf compiler next transforms programs to use heap-allocated memory, which requires

introducing memory into the program state. In this phase, explicit allocations and stores are inserted to construct well formed closure and constructor values. The simulation relation for this phase maps higher level values to their lower level memory representation: where the higher level program would produce a value, the lower level program produces a pointer whose contents correspond to the high level value.

Stack Introduction The next major step is stack introduction, where allocations and frees of runtime stack blocks are introduced. While the compilation strategy is straightforward (since stack blocks are never read nor written), the correctness proof is subtle and long due to the relatively complex relation of the unified, high-level heap memory to the split, lower-level heap and stack memories.

Backend Once the heap and stack are introduced, only minor changes are necessary to compile the program to a subset of Cminor. Afterward, the Œuf compiler relies on existing CompCert passes to compile the program from Cminor down to any of CompCert's assembly backends.

5 Shim

Œuf provides verified compilation of pure Gallina functions to assembly code. But real-world applications are not completely pure: they read inputs from disk, communicate over the network, or print results to the terminal. For any extraction or compilation mechanism, including Œuf's, to be useful, it must allow integrating the resulting code with a *shim* that connects the pure functions with impure side effects. For instance, one might verify and compile a pure function for computing the maximum of two numbers, then link that with a shim that reads two numbers from the terminal, invokes the compiled function, and outputs the result. When using Coq's built-in extraction mechanism, the shim is an OCaml or Haskell program; with Œuf, it is written in C.

Most verification projects do not bother verifying the shim code. Indeed, there is limited value in verifying a program whose main purpose is to call into code generated by unverified extraction and which itself will be compiled with an unverified compiler. With Œuf, however, stronger guarantees are possible. Œuf's compilation process is verified, and thus it is possible to establish strong correctness properties about the compiled code. Furthermore, Œuf is designed for use with shims written in C, which can be verified against the CompCert C semantics (e.g., using VST [4]) and then compiled with CompCert.

The remainder of this section describes aspects of Œuf's design that enable verification of complete programs, consisting of compiled code together with a C shim. Section 5.1 describes Œuf's primary correctness theorem, which is designed for reasoning about calls spanning the boundary between the shim and the compiled code Section 5.2 gives the complete definition of the Œuf application binary interface (ABI), including the definitions of several relations that are mentioned in the correctness theorem. Finally, Section 5.3 describes the assumptions currently relied on by Œuf's proofs.

5.1 Correctness Theorem

Œuf's correctness theorem is the primary interface for users who are verifying a shim program. Like most compiler correctness theorems, it relates the behavior of the high-level input source program (in this case Gallina) to the low-level output target program. However, we have been careful to state the theorem in terms that will be meaningful and relevant to the shim verifier. In particular, we state the theorem using Cminor as the low-level language, rather than any of CompCert's assembly languages, as we expect users will prefer verifying their shims at this higher level. If needed, one could directly compose Œuf's correctness theorem with the CompCert backend correctness theorem to obtain the equivalent theorem for CompCert assembly.

The correctness theorem relies on three Œuf-specific relations, called *value-matching*, *callstate*, and *returnstate*. The details of these relations define the ABI by which shims may interact with Œuf-compiled code. We describe each relation briefly, leaving their full definitions to the next subsection.

The value-matching relation $v \sim v'$ relates a high-level Gallina value v to its low-level memory representation v'. This is unlike the design of CompCert, which uses the same value representation at all levels and thus can use simple equality or "more-defined-than" to relate values in the input and output programs. Here, the high-level values are built from inductive data type constructors, such as the natural number `S (S (S O))` (representing 3), while low-level values are concrete integers and pointers in a particular machine-level memory layout.

The *callstate* and *returnstate* relations describe a particular subset of program states that stand in relation to values. We say that a program state is a *callstate* for closure f and argument a if it represents the point in execution "during the function call" of f applied to a, when control has transferred to the callee's code, but none of that code has executed yet. A program state is a *returnstate* carrying value r if it represents the point in execution "during the function return", with r as the value being returned. The precise details of these definitions depend on the language in question, and in fact, each intermediate language in the Œuf compiler has its own definition of callstates and returnstates. This is necessary because the structure of functions, calls, and returns changes as the program is transformed from an expression tree down to low-level sequential code. The correctness theorem refers to callstates and returnstates at the Cminor level, where these relations describe the calling convention of Œuf-compiled code. The statement of Œuf's correctness theorem is as follows:

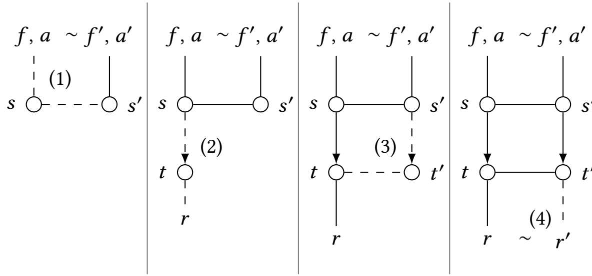

Figure 6. The structure of the Œuf compiler correctness proof. Given Gallina values f, a matching Cminor values f', a' and Cminor callstate s' for f' and a', we prove that (1) there exists a Gallina callstate s for f and a that matches s'; (2) s steps to some Gallina state t that is a returnstate for $r = f(a)$; (3) s' steps to a unique Cminor state t' such that t' matches t; and (4) t' is a returnstate for some r' that matches r.

Theorem 5.1. *Let s' be a Cminor callstate, representing an application of the Cminor closure value f' to the Cminor value a'. Suppose there exist Gallina values f and a such that $f \sim f'$ and $a \sim a'$ and the application $f(a)$ is well-typed in Gallina. Then there exists a unique Cminor returnstate t' carrying a value r' such that $f(a) \sim r'$ and the Cminor state s' will always reach t'.*

This theorem establishes total correctness of Œuf-compiled functions. Once the program begins performing the call to a compiled function, it is guaranteed that the callee will eventually return, barring exceptional conditions such as memory exhaustion. This is possible because all Gallina functions are terminating, and compiled functions behave identically to their original Gallina counterparts.

The proof of Œuf's correctness theorem is structured in five parts, illustrated in Figure 6. First, callstate matching: since s' is a Cminor callstate for closure f' and argument a', $f \sim f'$, and $a \sim a'$, there must exist a Gallina callstate s for f and a, which matches s' using a state-matching (or simulation) relation of the type commonly used in compiler correctness proofs. Second, termination: since all Gallina functions are terminating, there must exist a result value $r = f(a)$ and a Gallina returnstate t carrying r, where s takes zero or more steps to reach t. Third, forward simulation: since s is related to s' and s steps to t, there must exist a Cminor state t' matching t, where s' takes zero or more steps to reach t'. Fourth, returnstate matching: since t is a Gallina returnstate carrying r and t' is a Cminor state matching t, t' must be a Cminor returnstate carrying a value r', where $r \sim r'$. Fifth, backwards simulation via determinacy: since Cminor is deterministic, backwards simulation follows from forwards simulation. We have proved the first, third, and fourth parts in Coq; see Section 5.3 for details on our assumptions in the second and fifth parts.

Aside from termination, which is relevant only at the Gallina level, we prove the necessary lemmas by composition. Each transformation pass in the compiler includes proofs of callstate matching, forward simulation, and returnstate matching between its input and output languages. These separate proofs are composed to establish the end-to-end properties relating Gallina to Cminor.

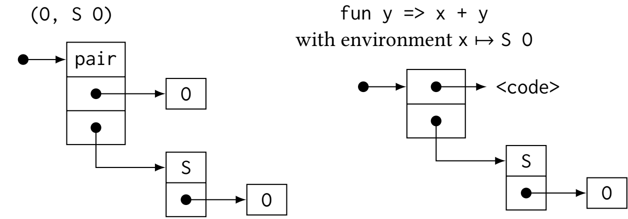

Figure 7. Example Cminor-level memory representations of Gallina values. The left shows the representation of the pair `(0, 1)` (represented in unary). The right shows the representation for a closure whose environment contains a single value (for the free variable `x`).

The theorems above ensure that an Œuf-generated Cminor function f' can always simulate the behavior of the corresponding input Gallina function f, and furthermore, all such behaviors will satisfy any theorems proved about f. To ensure that f' never exhibits any "extra" behaviors not possible for f, we adopt the approach used in CompCert and prove our target language, in this case Cminor, is deterministic. Note that (like CompCert), we assume that no program will exhaust memory. For a more in depth exploration of this assumption, we suggest prior work [32].

5.2 The Œuf ABI

The Œuf application binary interface (ABI) describes the means by which shim code written in C can call into Œuf-compiled functions and preserve any guarantees established at the source Gallina level. The ABI consists of a Cminor-level representation of Gallina values and a calling convention for invoking Œuf closures. These aspects of the ABI are captured in the definitions of the value-matching, callstate, and returnstate relations used in the compiler's primary correctness theorem. The theorem itself provides guarantees about the results of valid calls made according to the Œuf ABI.

Value Representation There are two kinds of values in the subset of Gallina supported by the prototype compiler: values of inductive data types, which we call "data", and closures. Our prototype uses a pointer to a sequence of machine words as the low-level value representation in both cases, so all Œuf values are one machine word in size when stored in registers, on the stack, or within another data structure. For data values, the first word of the allocation is an integer tag, indicating which constructor was used to build the value, and each remaining word is another Œuf value, comprising the arguments to the constructor. For closure values, the first word of the sequence is a function pointer, and the remaining words make up the environment of the closure. Figure 7 shows examples of the memory representations for some simple values.

Given these definitions, shims for use with our prototype are permitted two ways of constructing Œuf values. First, the shim can create a data value from a constructor tag and a sequence of argument values. It does so by allocating $1 + n$ words, where n is the number of arguments to the constructor, and filling the allocated storage with the constructor tag followed by the values. (Recall that all Œuf values are one word in size.)

Second, the shim can create a closure value for a function f. This is permitted only when f has no free variables—in other words, it must be a top-level input to the compiler, not a lifted lambda. Since the environment of the closure is always empty, the shim allocates exactly one word and writes the function pointer into the allocated memory. There is nothing strictly preventing the shim from building closures for lifted lambdas with nonempty environments, but the Œuf prototype compiler preserves the names only of top-level functions, so it is difficult to refer directly to a specific lifted lambda.

The shim, being an arbitrary C program, could of course perform other operations on Œuf values. Most notably, the shim might construct a closure whose code pointer refers to a C function defined by the shim itself. However, such a closure would not be related to any Gallina value under the value-matching relation, so the compiler's correctness theorem would provide no guarantees for operations that use such a value. This property captures the fact that some C functions, particularly those that use side effects, do not admit any Gallina implementation. Similarly, if the shim were to construct a cyclic data value at the Cminor level, it would not relate to any Gallina value (as Gallina inductive data must be finite), and the result of passing the illegal value to any Œuf function would be unspecified. The burden of reasoning about such unsupported manipulations of Œuf values is left to the user.

Calling Convention The shim can invoke Gallina functions compiled with the Œuf prototype compiler using a straightforward calling convention: given a closure `f` and an argument `a`, the shim should dereference `f` to obtain a function pointer `p`, then call `p(f, a)`. Since C supports only bare function pointers, not closures, the caller must provide `f` explicitly so the closure code can access the environment. The result of applying the closure to the argument is returned directly from the call to `p`.

There are two additional points to note regarding the practical use of this calling convention. First, this convention supports only a single argument for each function. This is in line with the definition of the Œuf prototype compiler's input language, which currently supports only single-argument lambdas. Multiple-argument functions can be implemented using currying, and they can be invoked by performing multiple individual call steps following this convention. Second, for uniformity, the prototype compiler only supports calls made through a closure object. If the shim needs to call a top-level function directly (as opposed to applying a closure obtained from some previous call), it must construct a closure object for the function as described above, then apply the closure as normal. We believe it would be straightforward to lift these restrictions when developing a more feature-complete compiler.

5.3 Verification Assumptions

To verify a complete program, including the interaction between Œuf-compiled code and the surrounding Cminor shim requires the following two additional facts.

1. Œuf-compiled functions will never modify any values in previously allocated memory.
2. The source Gallina program passed to Œuf terminates.
3. Forward simulation plus determinacy implies backwards simulation.

Fact (1) is necessary for carrying any properties *across* calls into Œuf-compiled code; otherwise, verifiers would be forced to assume that any properties established before the call could become invalid after the call. We are confident that this is true because Œuf-generated code only writes to small constant offsets into freshly allocated memory blocks. However, we have not yet formally proved this fact.

Fact (2) is true metatheoretically, but we need to manifest it inside of Coq. This amounts to proving normalization of our operational semantics for the compiler's reflected source language. More formally, given any initial source program state s, we assume there exists a source program state t such that s steps to t in zero or more steps and t is a final state, meaning no further evaluation is possible. We use this fact in our proof of Œuf's primary correctness theorem (Theorem 5.1). We have already proved a progress lemma (every non-final state can take a step), so the only remaining doubt is that we may have defined the semantics in a way that admits infinite sequences of steps through non-final states. We believe this is unlikely because our definition of the semantics largely follows standard practice, but we have not proved that no such infinite sequence exists. The proof could likely be conducted using the standard technique of logical relations [39].

Fact (3) is also true metatheoretically, and in fact CompCert proves it as a lemma for their semantic framework. However, Œuf's semantics cannot directly use this lemma because they do not fit into CompCert's semantic framework.

In the future, Œuf implementations may provide libraries proving these facts; however, in our current prototype we assume the shim implementer dispatches these obligations.

6 Trust

This section takes a more thorough look at the TCB (trusted computing base) of programs compiled with Œuf. In particular, we carefully consider how our reliance on the (unverified)

OCaml compiler and (unverified) runtime differs from the conventional approach to executing Gallina programs.

At first blush, the situation is fairly straightforward: Because Œuf is itself verified in Coq (as is the CompCert compiler whose back-end we reuse), we have a machine-checked guarantee that the x86 assembly Œuf generates is equivalent to the Gallina source code that Œuf consumes, up to the usual assumption that CompCert's model of x86 assembly is adequate. Given this equivalence, any Coq proofs about the source program apply also to the binary, provided shim code follows our specified ABI.

Remaining in the TCB are components that are typical in the development of formally verified systems: (1) the consistency of Coq's logic and the correctness of Coq proof checking; (2) the shim code and any libraries it uses (e.g., the C standard library); (3) the underlying execution environment (e.g., the operating system and hardware).

The first of these is the most interesting because Coq itself is implemented in OCaml and so the proof-checker's creation relied on the OCaml compiler and the proof-checker's execution relies on the OCaml run-time system. Therefore, in this sense, Œuf-compiled code still relies on OCaml's implementation, but the situation compared to extraction-of-Coq-to-OCaml is qualitatively different: A compiler or run-time bug cannot be exploited on Œuf-compiled code unless it causes the Œuf compiler itself to be incorrect in such a way that compilation appears to succeed but in fact is incorrect. That is, a security hole would require OCaml miscompiling the Œuf-compiler source-code in such a way that Œuf then miscompiled another Coq program in an exploitable way.

Would it help to compile the Œuf compiler with a verified compiler too, perhaps Œuf itself (i.e., bootstrapping our compiler)? In a practical sense, yes, since each level of a verified compiler compiling another verified compiler would make a false claim about the eventual program compiled by Œuf that much less likely. But on a theoretical level, such a sequence of verified compilers is exactly the focus of Thompson's famous lecture on, "Trusting Trust" [42]: In principle every compiler in the sequence of compiler bootstrapping steps back to the dawn of computing is in the TCB. In theory, Œuf is not immune to this classic argument.

In contrast, most prior work on verified systems relied *directly* on Coq's built-in extraction, despite the extraction procedure being complex, unverified, and known to contain bugs, including some that can cause incorrect code generation. Our Œuf compiler does not directly suffer from these problems unless a Coq-extraction or OCaml implementation bug causes the Œuf compiler itself to be miscompiled. Any Coq/OCaml bugs that do not affect the compilation of Œuf are irrelevant to Œuf's correctness. As for programmers' proofs about code that Œuf compiles, Coq's extraction mechanism is not relevant, but the implementation of Coq proof-checking in OCaml remains trusted.

Program	Component	Size (LOC)
`list_max`	Original	5
	Adapted	16
	Proofs	10
	Total	31
`SHA256`	Original	200
	Adapt (N)	230
	Proofs (N)	1400
	Adapt (elim)	700
	Proofs (elim)	650
	Total	3180

Figure 8. Approximate line counts for the Gallina components of our test programs. "Original" is the size of the original implementation, written in idiomatic Gallina. "Adapt" is the size of the code after adaptation for compatibility with the Œuf prototype compiler, and "proofs" is the size of the proofs of equivalence between the original and adapted versions. We adapted `SHA256` code in two phases, first converting it to use the `N` natural number type, and later converting it to use eliminators in place of explicit recursion. Totals include both original and adapted implementations and equivalence proofs because reasoning about calling code typically refers to all components.

7 Case Study

To judge the initial feasibility of our approach, we used the Œuf prototype to compile and run two example programs. Our goal in doing so was to show that the techniques used in Œuf are sufficient for compiling and running nontrivial Gallina functions. Demonstrating good performance was an explicit non-goal—we have thus far focused on the front-end and shim interfaces, and put minimal effort into optimization. As expected, Œuf-compiled functions use orders of magnitude more time and space than manually-written C.

7.1 Test Cases

We implemented two programs and compiled them using the Œuf prototype to test our approach. Each program consists of a pure functional component written in Gallina and a shim written in C. The shim reads from standard input, invokes the Œuf-compiled version of the Gallina function, and writes the result to standard output.

`list_max`. Our first test case is a simple program for finding the largest number in a list. The Gallina component is a function `list_max : list nat -> nat`, which we implemented in idiomatic Coq style. We then adapted the function and all of its dependencies to use eliminators, as required by Œuf's input language, and proved the adapted version equivalent to the original. The size of each is shown in Figure 8.

The shim for `list_max` reads integers from standard input and builds an Œuf list containing the equivalent `nat`s, using the data representation described in Section 5.2 It then invokes the compiled `list_max` function, converts the resulting `nat` to a C integer, and prints it.

`SHA256`. Our second test case is a SHA-256 hashing utility, similar to `sha256sum`. The Gallina component is a complete

Program	Input	Default	Boehm	Slab	OCaml
`list_max`	100 items	0.03 s	0.04 s	0.01 s	0.00 s
	1000 items	(OOM)	34.63 s	11.31 s	0.02 s
`SHA256`	55 bytes	2.22 s	3.12 s	1.31 s	0.07 s
	500 bytes	(OOM)	24.44 s	10.75 s	0.58 s
	5000 bytes	(OOM)	246.94 s	107.06 s	5.85 s

Figure 9. Performance results for our test programs. The first three timing columns show the time taken when running Œuf-compiled code with the default shim and allocator, with the default shim and the Boehm GC, and with the slab-based shim and allocator. The final column shows the time taken by the same program run via unverified extraction to OCaml. Trials marked "OOM" failed after exhausting the 4GB of address space.

implementation of the SHA-256 hash function, originally developed by Appel [5] for their verification of the OpenSSL SHA-256 code. `SHA256` is a nontrivial function, comprising about 200 lines of code, not including the numerous list and integer library functions that it relies upon.

We adapted `SHA256` for compilation with Œuf in two steps. First, we converted all uses of CompCert's `int` type to equivalent operations using Coq's built-in `N` natural number type, which (unlike `nat`) represents each number as a linked list of bits. CompCert implements the `int` type as a dependent pair of a mathematical integer $x \in \mathbb{Z}$ and a proof that $0 \leq x < 2^{32}$, but our prototype compiler cannot handle types that include proof terms. Removing uses of `int` was largely a straightforward replacement of `int` and operations with equivalent functions over `N`s. Second, we converted uses of fixpoints and pattern matching to explicit invocations of eliminators, as we did for `list_max`. In most cases this transformation was completely mechanical, but two functions required more significant adjustment due to more complex recursive calls. The sizes of these variants (including adapted library functions) and the proofs relating them is shown in Figure 8.

The `SHA256` shim reads bytes from standard input until EOF, building an Œuf list containing each byte represented as an `N`. It passes the list to the compiled `SHA256` function, obtaining another list of bytes (again stored as `N`s in the range 0–255) representing the hash. Finally, it prints each byte of the hash in hexadecimal, as is standard for hashing utilities.

7.2 Performance

Performance results for the two example programs are shown in Figure 9. All programs compile and run correctly, but as expected, they are quite slow and use a large amount of memory. As shown in the third column of Figure 9, running Œuf-compiled code with the default memory allocator results in an out-of-memory condition on all but the smallest inputs. This is caused by the naïve memory allocation strategy used in the current implementation. Due to the difficulty of correctly implementing and verifying automatic memory management, the Œuf prototype compiler generates code that never deallocates. Temporary objects are leaked once they become unused, and the size of the heap quickly exceeds the 4GB address space limit for 32-bit processes.

To get some idea of our test programs' performance on larger inputs, we ran them with two other memory allocation strategies. These options reduce the overall memory footprint, but at the cost of increasing the TCB. The fourth column of the table shows the results of running the test programs with the Boehm garbage collector [7], which automatically frees memory that is no longer reachable. The fifth column shows the results with a modified shim that uses slab allocation. Slab allocation is an allocation strategy where new memory is allocated from various slabs, and it is possible to free an entire slab at a time. This is useful when calling Œuf-compiled code, as the result of each call can be copied out, and the slab used to make the call can be freed. The slab-allocated variants are modified such that iteration over the input sequence is handled in C instead of in Gallina. The body of the loop is the same as in the original implementation: a call to a Gallina function that processes the next chunk of data. Aside from this change, the shim also uses a simple slab allocator (under 100 lines of code) that wraps the system `malloc`. After each iteration, the shim reads out the result, then invokes an allocator routine to discard all other data that was allocated during the iteration. Though verifying either a garbage collector or a slab allocator would require significant additional work, doing so would improve Œuf-compiled code performance along the lines of Figure 9.

We believe that another major factor in the performance of our test programs is the representation of numbers used in these programs. Like most Coq programs, both of our examples use the numeric types from the standard library: either the `nat` algebraic data type, a unary representation that stores x as a chain of $O(x)$ constructors, or the `N` and `Z` types, which use a binary representation requiring $O(\log x)$ constructors. As a result, all operations on numbers take time linear or logarithmic in the values of their inputs. This problem could be addressed by providing Œuf-compiled programs with access to a type that compiles down to machine integers, along with operations that have the same wrapping behavior as native arithmetic instructions, but we leave the verified implementation of such to future work.

Finally, we compared Œuf-compiled code with code extracted using Coq's built-in unverified extraction mechanism and then compiled with the standard OCaml compiler. For this comparison, we extracted the Œuf-adapted version of each test program's Gallina component, so the exact same code and data structures were used for both Œuf- and OCaml-compiled variants. Even using the same sub-optimal numeric representation, the OCaml-compiled version of `SHA256` is about 20 times faster than the version produced by the Œuf prototype compiler. We believe this difference is most likely caused by the prototype compiler's unoptimized handling of closures. It currently allocates a closure object for every call, including every partial application within a call to a curried function, and makes no attempt to remove unused variables from closure environments. Furthermore, the generated code

always calls using the closure's code pointer, even when the target is statically known, which prevents any inlining that might normally be performed by CompCert. Improving the prototype compiler's code generation to address these issues is left to future work.

8 Related Work

Verified Compilation. Work on verified compilers stretches almost as far as the invention of compilers themselves [29] and continued throughout the second half of the twentieth century [31]. (See Dave [14] for further references.) But the breakthrough work of Leroy [25] fundamentally altered the landscape by showing that it was possible to carry out the full development in a proof assistant, showing that the assembly output is guaranteed to be equivalent to the input C program. Over the last decade, CompCert has been extended in a variety of directions, *e.g.*, to make guarantees about its parser [21]. Even more importantly, a wide array of projects use CompCert as a subcomponent [4, 13, 17, 18, 44]. Œuf similarly builds on top of CompCert.

Œuf is most similar in purpose and design to CertiCoq, a verified Gallina compiler developed by Anand et al. [3], CertiCoq uses the template-coq reflection mechanism to reflect Gallina functions into input ASTs, compiles the program down to CompCert's Clight language using a series of verified transformation passes, and runs the CompCert backend to produce executable code. Œuf can compile only a limited subset of Gallina programs[1], while CertiCoq can compile any Gallina program, but Œuf provides stronger guarantees about the programs in that limited subset. The key difference is that Œuf's restriction of the input language allows for translation validation of the reflection procedure, via a verified denotation. The resulting proof of correspondence between the original and reflected terms, together with the compiler's own correctness theorem, enables users of Œuf to reason soundly about compiled code in terms of its original high-level Gallina implementation. Œuf also exposes its correctness theorem to the shim, which allows reasoning about code that calls Œuf-compiled functions.

Other work has also investigated verifying compilers for functional programming languages more generally, including features such as general recursion, side effects, and nondeterminism [6, 9, 10, 15, 36]. Perhaps most closely related is the CakeML verified ML compiler [24, 41]. CakeML supports a rich input language including mutually recursive functions, pattern matching, ML modules, and mutable arrays. CakeML's design was a major inspiration for the design of Œuf's internal passes: both compilers use many small transformation passes between closely related languages. One technical difference is that CakeML is able to bootstrap by directly evaluating the compiler inside HOL4. This avoids adding that "phase" of trust to the TCB, as discussed in Section 6. When given shallowly embedded HOL functions, the CakeML compiler uses a form of translation validation to guarantee their input arrives at the frontend of the compiler unchanged [33]. While CakeML syntactically derives a certificate that the input program evaluates to the correct answer in the given big step semantics, Œuf uses denotational semantics to denote the program back into Gallina. Either approach results in similar levels of trust, however the complexity of denotation has been a limitation preventing Œuf from supporting more features, and thus the approach from CakeML may improve the Œuf compiler.

The Œuf correctness theorem is useful for reasoning about programs in the target language that interact with Œuf-compiled code. One special case of such reasoning is *separate compilation*, which previous work has considered both in the context of CompCert [22, 40] and elsewhere [34, 43]. Œuf also does not assume that the entire program is in Gallina, instead linking with a shim during compilation.

Another approach to reasoning about the interaction between the shim and the source program is to use a *multi-language semantics* [1, 35]. Such a semantics gives a unified account of the behavior of the shim, the source program, and the compiled program, which allows clean reasoning.

Cogent [2] is a language for systems code whose compiler produces not only C code, but also a high-level specification in Isabelle/HOL and proof of refinement. Users can reason about the code in terms of its high-level specification, similar to how Œuf users can reason in terms of the Gallina code.

Œuf's use of the computational denotation to translation validate reflection and lambda lifting is broadly similar to Govereau [16], which uses a denotational semantics to translation validate LLVM optimization passes. Œuf's reflection procedure is similar in spirit to Template Coq [28], which can reflect *all* of Gallina into syntax represented by an inductive datatype in Coq. On the other hand, Template Coq does not support *denoting* all programs in this syntax back into Coq.

Verified Software Toolchain. Languages such as Gallina are designed from the ground up for integration into a proof assistant. In contrast, languages not designed with such goals in mind require additional tooling to enable verification. One such example is VST [4], a framework for reasoning about C programs. It provides the ability for users to prove properties of their programs, then appeal to a mechanically verified theorem that those properties are preserved through compilation. VST provides a deeply embedded separation logic over a C-like language, while Œuf lets users prove properties of their programs using Coq's built-in support for reasoning about Gallina. While C programs tend to have orders of magnitude better performance than their Œuf-compiled counterparts, we believe that the proof effort saved puts Œuf at a useful point in the design space.

[1] The CertiCoq implementation is not yet publicly available, so we have not been able to make direct comparisons.

Other Verified Systems. The primary way to use Coq to build systems today is to use extraction [27]. While convenient, extraction provides no formal guarantee that the resulting program is semantically equivalent to the original. Another way to build systems in Coq is to use Coq.io, which provides monadic I/O primitives [12]. The user writes a Gallina program using these primitives for side-effectful operations, then reasons about it using theorems that describe the behavior of those monadic primitives. Executing the code still requires extraction and compilation with the unverified OCaml compiler. In principle, we could provide similar I/O facilities by extending Œuf with monadic primitives and implementing an interpreter in the C shim.

The seL4 operating system was one of the first major successes in formal verification of large systems [23]. They spent 20 person years on verification, which was largely refinement proofs. By building verified compilers we can mitigate some of this burden for future projects of this scale.

GCminor [30] is a language similar to the CompCert Cminor IR, except containing garbage collection primitives. From this language, there is a verified translation to Cminor, along with a Cminor specification and implementation of a garbage collector. In the future, Œuf could benefit from a verified garbage collector.

9 Conclusion

We presented Œuf, a verified compiler from Gallina to assembly language. Along with standard compiler verification techniques, Œuf uses computational denotation of its input language to perform translation validation of its reflection procedure and lambda-lifting pass, eliminating both from the TCB. To enable verification of interactions between Œuf-compiled functions and their C shims, we expressed Œuf's primary correctness theorem in terms of CompCert Cminor values and program states. Finally, we compiled and ran Appel's Gallina specification of SHA256 with Œuf to ensure that Œuf can handle nontrivial input programs.

We are working towards a future where implementers can consider Gallina a full-fledged systems programming language that requires no special effort to run verified code. This paper's key contribution is a detailed exploration of the trade-offs for designing the *interfaces* future implementers may program against.

Acknowledgments

The authors acknowledge Andrew Appel, Clément Pit–Claudel, and Magnus O. Myreen for useful discussion, and the anonymous reviewers for their helpful feedback.

Zachary Tatlock's research was supported in part by a generous gift from Google. This material is based upon work supported by the National Science Foundation under Grant Nos. DGE-1256082 and 1219172. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Amal Ahmed. 2015. Verified Compilers for a Multi-Language World. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*. 15–31.

[2] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, and others. 2016. Cogent: Verifying high-assurance file system implementations. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 175–188.

[3] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *CoqPL Workshop (CoqPL '17)*.

[4] Andrew W. Appel. 2011. Verified Software Toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software (ESOP'11/ETAPS'11)*. Springer-Verlag.

[5] Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* 37, 2 (April 2015).

[6] Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, Step-indexing and Compiler Correctness. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. 97–108.

[7] H Boehm and M Weiser. 1988. Garbage Collection in an Uncooperative Environment. In *Software Practice and Experience*.

[8] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM.

[9] Adam Chlipala. 2007. A certified type-preserving compiler from lambda calculus to assembly language. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 54–65.

[10] Adam Chlipala. 2010. A Verified Compiler for an Impure Functional Language. *SIGPLAN Not.* 45, 1 (Jan. 2010), 93–106.

[11] Adam Chlipala. 2013. *Certified Programming with Dependent Types*. MIT Press.

[12] Guillaume Claret. 2016. Coq.io. (2016). http://coq.io/

[13] David Costanzo, Zhong Shao, and Ronghui Gu. 2016. End-to-end Verification of Information-flow Security for C and Assembly Programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. 648–664.

[14] Maulik A. Dave. 2003. Compiler Verification: A Bibliography. *SIGSOFT Softw. Eng. Notes* 28, 6 (2003), 2–2.

[15] Maxime Dénès and Xavier Leroy. 2015. Coqonut: A verified JIT compiler for Coq. http://www.maximedenes.fr/download/coqonut.pdf. (January 2015).

[16] Paul Govereau. 2012. *Denotational Translation Validation*. Ph.D. Dissertation. Harvard University.

[17] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 595–608.

[18] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 653–669.

[19] Arjun Guha, Mark Reitblatt, and Nate Foster. 2013. Machine-verified Network Controllers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM.

[20] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2012. Establishing Browser Security Guarantees Through Formal Shim Verification. In *21st USENIX Conference on Security Symposium (SECURITY '12)*. USENIX Association.

[21] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating $LR(1)$ Parsers. In *Proceedings of the 21st European Symposium on Programming (ESOP 2012) (Lecture Notes in Computer Science)*, Vol. 7211. Springer, 397–416.

[22] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. 178–190.

[23] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM.

[24] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM.

[25] Xavier Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM.

[26] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: Certified Causally Consistent Distributed Key-value Stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM.

[27] Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008 (Lecture Notes in Computer Science)*, Vol. 5028. 359–369.

[28] Gregory Malecha. 2015. Template Coq Plugin. https://github.com/gmalecha/template-coq. (2015).

[29] John McCarthy and James Painter. 1967. Correctness of a compiler for arithmetic expressions. *Mathematical aspects of computer science* 1 (1967).

[30] Andrew McCreight, Tim Chevalier, and Andrew Tolmach. 2010. A Certified Framework for Compiling and Executing Garbage-collected Languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. 273–284.

[31] J. Strother Moore. 1989. A Mechanically Verified Language Implementation. *J. Autom. Reasoning* 5, 4 (1989), 461–492.

[32] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. 2016. Verified Peephole Optimizations for CompCert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*.

[33] Magnus O. Myreen and Scott Owens. 2012. Proof-Producing Synthesis of ML from Higher-Order Logic *(ICFP '12)*.

[34] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: a compositionally verified compiler for a higher-order imperative language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. 166–178.

[35] James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. 128–148.

[36] Clément Pit-Claudel. 2016. *Compilation Using Correct-by-Construction Program Synthesis*. Master's thesis. Massachusetts Institute of Technology. http://pit-claudel.fr/clement/MSc/

[37] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '98)*. Springer-Verlag.

[38] Hanan Samet. 1978. Proving the Correctness of Heuristically Optimized Code. *Commun. ACM* (1978).

[39] R. Statman. 1985. Logical relations and the typed lambda-calculus. *Information and Control* 65, 2 (1985), 85 – 97.

[40] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. 275–287.

[41] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2016. A New Verified Compiler Backend for CakeML *(ICFP '16)*.

[42] Ken Thompson. 1984. Reflections on Trusting Trust. *Commun. ACM* 27, 8 (Aug. 1984).

[43] Peng Wang, Santiago Cuellar, and Adam Chlipala. 2014. Compiler verification meets cross-language linking via data abstraction. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 675–690.

[44] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. 2014. Jitk: A Trustworthy In-Kernel Interpreter Infrastructure. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. USENIX Association.

[45] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM.