Pushing the Limits of Compiler Verification

Eric Mullen

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2018

Reading Committee:

Zachary Tatlock, Chair

Daniel Grossman, Chair

Xi Wang

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Pushing the Limits of Compiler Verification

Eric Mullen

Co-Chairs of the Supervisory Committee:
Assistant Professor Zachary Tatlock
Computer Science and Engineering

Professor Daniel Grossman
Computer Science and Engineering

Modern computer systems rely on the correctness of at least one compiler for correct operation. Formal verification is a powerful technique for constructing correct systems. While there have been many efforts to develop formally verified compilers, those compilers are still not widely used. In this thesis, I present two major systems developments and one case study which push the limits of compiler verification, towards more and better verified compilers.

Œuf: 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).

Here I present 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. I 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 my collaborators and me 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.

Using Œuf: Œuf was developed in order to allow verified systems to be developed and verified in Coq, compiled to executable code using Œuf, with all guarantees proven at the Gallina level preserved through compilation to the assembly level. In order to evaluate this goal, I built the WordFreq verified system in Coq, compiled it with Œuf, and preserve the correctness guarantee through to the generated assembly code.

Here I present the WordFreq verified system, its correctness guarantee, and the major parts of its correctness proof. I discuss the development of the system and its proof, as well as the axiomatic primitives necessary to tie it together.

Peek: Transformations over assembly code are common in many compilers. These transformations are also some of the most bug-dense compiler components. Such bugs could be eliminated by formally verifying the compiler, but state-of-the-art formally verified compilers like CompCert do not support assembly-level program transformations. Here I present Peek, a framework for expressing, verifying, and running meaning-preserving assembly-level program transformations in CompCert. Peek contributes four new components: a lower level semantics for CompCert x86 syntax, a liveness analysis, a library for expressing and verifying peephole optimizations, and a verified peephole optimization pass built into CompCert. Each of these is accompanied by a correctness proof in Coq against realistic assumptions about the calling convention and the system memory allocator.

Verifying peephole optimizations in Peek requires proving only a set of local properties, which my collaborators and I have proved are sufficient to ensure global transformation correctness. We have proven these local properties for 28 peephole transformations from the literature. Here I discuss the development of our new assembly semantics, liveness analysis, representation of program transformations, and execution engine; describe the verification challenges of each component; and detail techniques we applied to mitigate the proof burden.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

First and foremost, a sincere thank you to my advisors Zach and Dan. I couldn't have done any of this without you. Second, thank you to every teacher who has contributed to my education, in any form, big or small. And finally, thank you to my friends and family, who have supported me every step of the way.

DEDICATION

für meine Liebe, Katharina

Chapter 1
INTRODUCTION

Every day, our society moves towards greater dependence on software. From aerospace, and finance to transit and even dating, many facets of our lives are increasingly influenced by technology. While this technology brings impressive benefits, it is not without its drawbacks. Bugs in critical software can cause large financial loss [23], ground flights [10], or even cause loss of life [36]. Given our increasing reliance on critical software, we must meet this trend with increased reliability of our critical software.

No computer program is constructed in a vacuum. Every line of code is touched by large and complex pieces of software development infrastructure. If our goal is to create good software, we must first look to this infrastructure. In particular, the compiler (which translates human legible programs down to executable instructions) must not introduce errors in its translation, or no program has any hope of being error-free.

The compiler is a cornerstone of software development infrastructure. There exist formally verified compilers [59, 34] which guarantee that the meaning of any output program they generate will match the meaning of their input program. These guarantees have held up in practice with extensive testing [65, 31].

However, the remaining gap between existing verified compilers and the industrial language implementations used in practice leaves much to be desired. Specifically, existing verified compilers do not generate code that executes as fast as state of the art traditional compilers. As well, there are not many source languages supported by verified compilers.

In this thesis, I outline two projects which have pushed the limits of what is possible with verified compilers, both in terms of efficiency of verified code, as well as the number of languages supported. Both major projects build on the CompCert verified C compiler [34], a monumental project which has demonstrated that large scale verified software can be practical.

- Chapter 2 describes the architecture of the Œuf compiler, which enables verified compilation of verified systems built in Coq with a minimal trusted code base (TCB). This chapter is adapted from its original appearance in a paper published at CPP 2018 [45]. The Œuf compiler is open source, and available at `https://github.com/uwplse/oeuf`.

- Chapter 3 describes the ongoing efforts to build verified systems which can be compiled using the Œuf verified system development methodology. Sections 3.1 and 3.2 are adapted from their appearance in a paper published at CPP 2018 [45]. The remaining sections discuss the WordFreq system, developed for a separate paper, which originally appeared at ITP 2018 [28].

- Chapter 4 describes a lower level semantics for x86 assembly language within CompCert, which was originally created to allow verification of more peephole optimizations. This work originally appeared in PLDI 2016 [46], and has been slightly updated since publication.

- Chapter 5 describes a verified peephole optimizer for CompCert. This work originally appeared with the previous chapter in PLDI 2016 [46]. The Peek system is open source, and available at `https://github.com/uwplse/peek`.

1.1 CompCert

Both projects described in this thesis use the CompCert verified C compiler [34] in some core way. Thus understanding the work in this thesis is contingent on understanding the CompCert compiler. While CompCert is a C compiler, it is developed in Coq. This allows for the operational definition of program transformation from C to assembly to be accompanied by a proof of correctness. In CompCert's case, this proof is of the property that the output of CompCert must be a program which has the same meaning as the input program.

The statement of CompCert's correctness is a bisimulation. Specifically, it is a bisimulation between the input C program and the output assembly program. A bisimulation is the conjunction of a forward and a backward simulation. A forward simulation between two programs $\mathbb{A}$ and $\mathbb{B}$ means that any behavior that is exhibited by program $\mathbb{A}$ is also exhibited by program $\mathbb{B}$. A backwards simulation is similar, except any behavior that is exhibited by program $\mathbb{B}$ is also exhibited by program $\mathbb{A}$.

When we know that a bisimulation exists between two programs, we know that they both do the same thing. This is compiler correctness, and this is the property for which CompCert provides a proof.

CompCert is also a practically useful C compiler. It produces code with decent performance and runs with reasonable compilation time. However, it doesn't yet compete with state of the art non-verified C compilers such as `gcc` or `clang` in terms of the performance of generated code, nor in the number of targeted architectures. Back-ends currently exist for x86, ARM, PowerPC, RISC-V. Adding a back-end is non-trivial as one needs a correct semantics for the assembly language.

Chapter 2
ŒUF

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).

Here I present 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. I 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 my collaborators and me to implement some early compiler passes (e.g., lambda lifting) in the untrusted reflection and ensure their correctness via translation validation.

2.1 Œuf overview

Figure 2.1 shows an overview of the Œuf compilation process. The user provides a Gallina function, such as the `max` function in Figure 2.2, 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 2.3 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 to a common representation, links them, and invokes the CompCert backend to produce an

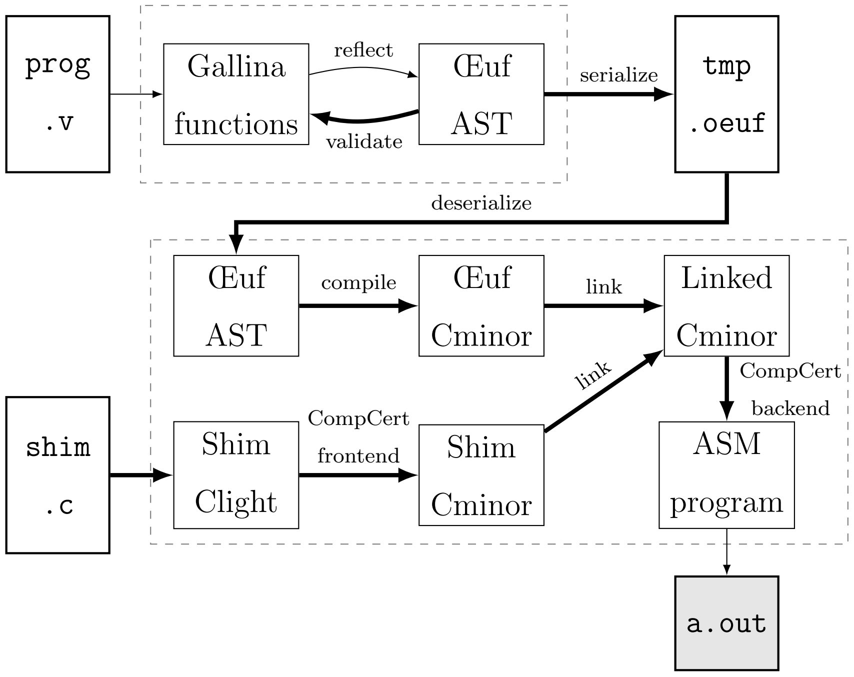

Figure 2.1: The Œuf compilation workflow. Arrows in bold indicate formally-verified operations. The output is `a.out`, indicated with gray.

```
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 2.2: A Gallina function for computing the maximum of two natural numbers and its equivalent written in terms of `nat_rect`.

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 2.2.3. 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 2.2.

The function `max` in Figure 2.2 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 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 2.4, 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 [53, 54]. 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 2.4; 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

```
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 2.3: A C program that calls the `max` function defined in Figure 2.2. Conversion between natural numbers and unsigned ints is provided by a library which is included with Œuf, but which is currently unverified.

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

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 2.1). The code for this step is shown in Figure 2.4.

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: $\forall x$, deserialize(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 2.3. 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.

2.1.1 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 $s \to s'$), 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.

A more complete discussion of Œuf's correctness proof occurs in Section 3.2.

2.2 Œuf Frontend

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.

2.2.1 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 trade offs, 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 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.

2.2.2 Syntax

Figure 2.5 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, 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.

2.2.3 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 that 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 [14]). 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 fig. 2.5 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.

2.2.4 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 ($[\![\cdot]\!]$): $\forall\ s\ s',\ \ s \rightarrow s' \Rightarrow [\![s]\!] = [\![s']\!]$. 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.

2.2.5 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.

2.3 Œuf 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 [34, 59].

2.3.1 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 2.2. 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.

2.3.2 Lowering towards Cminor

At this point the program is in a language that 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.

Return Statements The first step towards lower level programs is to add an explicit return statement to the end of each function. This simplifies the program somewhat, as function bodies are now simply a statement, instead of a statement and expression pair.

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 at this level), the correctness proof is subtle and lengthy due to the relatively complex relation of the unified, high-level heap memory to the split, lower-level heap and stack memories.

Malloc Introduction The next major step is turning memory allocation expressions into actual calls to the malloc function. This is relatively straightforward.

Backend At this point, the program is expressed in a language which is a subset of Cminor. It is trivial to emit it as a Cminor program, which we pass on to the linking stage.

2.4 Linking

Once the pure parts of the program are compiled from Gallina to Cminor, we link these definitions with the *shim*, a C program which can call these pure definitions, and can also perform effectful computation, such as I/O. The shim program is compiled to Cminor, at which point the pure definitions and the shim are linked into one whole Cminor program.

This Cminor program is then compiled to an executable using any of CompCert's backends. Further discussion of this occurs in Section 3.2.

2.5 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" [60]: 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.

2.6 Related Work

Verified Compilation. Work on verified compilers stretches almost as far as the invention of compilers themselves [40] and continued throughout the second half of the twentieth century [43]. (See [17] for further references.) But the breakthrough work of [34] 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 [24]. Even more importantly, a wide array of projects use CompCert as a subcomponent [63, 5, 16, 20, 21]. Œuf similarly builds on top of CompCert.

Œuf is most similar in purpose and design to CertiCoq, a verified Gallina compiler [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 non-determinism [7, 12, 13, 52, 18]. Perhaps most closely related is the CakeML verified ML compiler [29, 59]. 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 2.5. 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 [47]. 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 lan-

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

guage 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 [26, 58] and elsewhere [48, 62]. 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* [50, 1]. 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 [19], which uses a denotational semantics to translation validate LLVM optimization passes. Œuf's reflection procedure is similar in spirit to Template Coq [38], 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 [5], 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.

Other Verified Systems. The primary way to use Coq to build systems today is to use extraction [35]. 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 [15]. 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 [27]. 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 [41] 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.

$$
\begin{array}{rcll}
e & \in & \mathit{expr} & \text{(Expressions)} \\
e & ::= & x & \text{(variable)} \\
 & | & e\ e & \text{(function application)} \\
 & | & C\ e^{*} & \text{(data constructor)} \\
 & | & E\ e^{*}\ e & \text{(data eliminator)} \\
 & | & f\ e^{*} & \text{(closure creator)} \\
 & & & \\
C & \in & \mathit{constr} & \text{(Constructors)} \\
C & ::= & \mathrm{true} \mid \mathrm{cons} & \\
 & | & \ldots & \\
 & & & \\
\tau & \in & \mathit{type} & \text{(Types)} \\
\tau & ::= & b \mid \tau \to \tau & \\
 & & & \\
f & \in & \mathit{func} & \text{(Function name)} \\
 & & & \\
b & \in & \mathit{base} & \text{(Base types)} \\
b & ::= & \mathrm{bool} & \\
 & | & \mathrm{list}\ b & \\
 & | & \ldots & \\
 & & & \\
E & \in & \mathit{elim} & \text{(Eliminators)} \\
E & ::= & \mathrm{bool\_elim} & \\
 & | & \mathrm{list\_elim} & \\
 & | & \ldots & \\
 & & & \\
g & \in & f \to e & \text{(Global environment)}
\end{array}
$$

Figure 2.5: Syntax of the Œuf prototype compiler's source language.

Chapter 3
USING Œuf

Œuf was developed in order to allow verified systems to be developed and verified in Coq, compiled to executable code using Œuf, with all guarantees proven at the Gallina level preserved through compilation to the assembly level. In this section, I will detail our experiences with using Œuf to develop verified systems.

Sections 3.1 and 3.2 have previously appeared in CPP 2018 [45]. Section 3.1 details a few small systems we compile with Œuf. Section 3.2 details the ABI for interaction with Gallina code compiled with Œuf. The remaining sections discuss in greater depth the process of linking a shim with Œuf compiled Gallina definitions (3.3), how to formally reason about a linked program (3.4), an overview of the WordFreq verified system (3.5), and the Trusted Code Base (TCB) of the WordFreq verified system developed with Œuf (3.6).

3.1 Mini Case studies: SHA256 and list_max

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.

3.1.1 Implementation

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:

```
Fixpoint list_max (xs : list nat) : nat :=
    match xs with
    | [] => 0
    | x :: xs => max x (list_max xs)
    end.
```

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:

```
Definition max' (n m : nat) : nat :=
    nat_rect (fun _ => nat -> unit -> nat)
        (fun m dummy => m)
        (fun n' IHn m dummy =>
            nat_rect (fun _ => unit -> nat)
                (fun dummy => S n')
                (fun m' IHm dummy => S (IHn m' dummy))
            m dummy)
    n m tt.


Definition list_max' (xs : list nat) : nat :=
    list_rect (fun _ => nat)
        (0)
        (fun x xs IHxs => max' x IHxs)
    xs.
```

```
Lemma max'_correct:
  forall n m, max' n m = max n m.
Proof.
  induction n; destruct m; simpl; try reflexivity.
  - rewrite <- IHn. reflexivity.
Qed.

Lemma list_max'_correct:
  forall xs, list_max' xs = list_max xs.
Proof.
  induction xs; simpl; try reflexivity.
  - rewrite IHxs. apply max'_correct.
Qed.
```

We next reflect these definitions into deeply embedded ASTs, and prove that they denote back to the originals. Next we serialize the ASTs out to a file.

```
Oeuf Reflect list_max' As list_max_cu.

Lemma list_max_cu_validate:
  hhead (genv_denote (exprs list_max_cu)) hnil = list_max'.
Proof.
  reflexivity.
Qed.

Oeuf Eval lazy Then Write To File "list_max.oeuf"
  (Pretty.compilation_unit.print list_max_cu).
```

Finally, we need a shim (piece of C code) to call the Gallina defined function, and perform any necessary I/O.

```
#include <stdio.h>
#include "shim.h"

/* included from "shim.h" */
union nat {
    int tag;
    struct {
        int tag;
    } O;
    struct {
        int tag;
        union nat* n;
    } S;
};

#define TAG_nat_O       0
#define TAG_nat_S       1

union list {
    int tag;
    struct {
        int tag;
    } nil;
    struct {
        int tag;
        void* head;
        union list* tail;
    } cons;
};

#define TAG_list_nil        0
#define TAG_list_cons       1

typedef void* oeuf_function(void*, void*);
/* End content from shim.h */

oeuf_function list_max;

union list* read_nat_list() {
    union list* head = NULL;
    union list** tail = &head;
    unsigned n;
    union nat* n_;
    while (scanf("%u", &n) == 1) {
        n_ = nat_of_uint(n);
        *tail = make_cons(n_, NULL);
        tail = &(*tail)->cons.tail;
    }
    *tail = make_nil();
    return head;
}

int main() {
    union list* input_ = read_nat_list();
    unsigned result;
    union nat* result_;
    result_ = OEUF_CALL(list_max, input_);
    result = uint_of_nat(result_);
    printf("max = %u\n", result);
    return 0;
}
```

The shim for `list_max` (shown above) reads integers from standard input and builds an Œuf list containing the equivalent `nat`s, using the data representation described in Section 3.2.2. 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 implementation of the SHA-256 hash function, originally developed by Appel [4] 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

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 3.1: 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.

the proofs relating them is shown in Figure 3.1.

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.

3.1.2 Performance

Performance results for the two example programs are shown in Figure 3.2. All programs compile and run correctly, but as expected, they are quite slow and use a large amount of

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 3.2: 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.

memory. As shown in the third column of Figure 3.2, 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 [9], 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 3.2.

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.

3.2 Œuf ABI

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 [5]) 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 3.2.1 describes Œuf's primary correctness theorem, which is designed for reasoning about calls spanning the boundary between the shim and the compiled code Section 3.2.2 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 3.2.3 describes the assumptions currently relied on by Œuf's proofs.

3.2.1 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:

Theorem 3.2.1. *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 3.3. 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

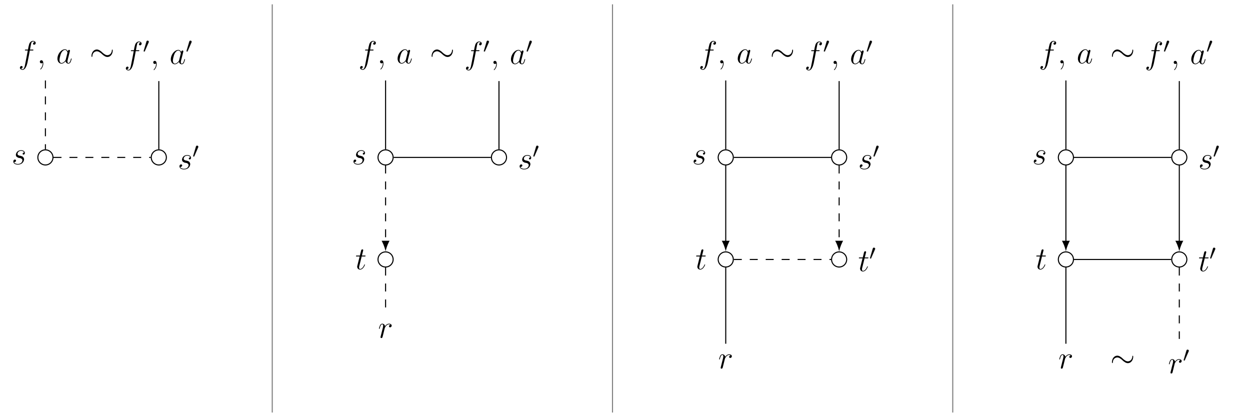

Figure 3.3: The structure of the Œuf compiler correctness proof, from left to right. Given Gallina values f, a matching Cminor values f', a' and Cminor callstate s' for f' and a', we prove that (1) there exists a Gallina callstate s for f and a that matches s'; (2) s steps to some Gallina state t that 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.

simulation. We have proved the first, third, and fourth parts in Coq; see section 3.2.3 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.

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 [46].

3.2.2 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 3.4 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

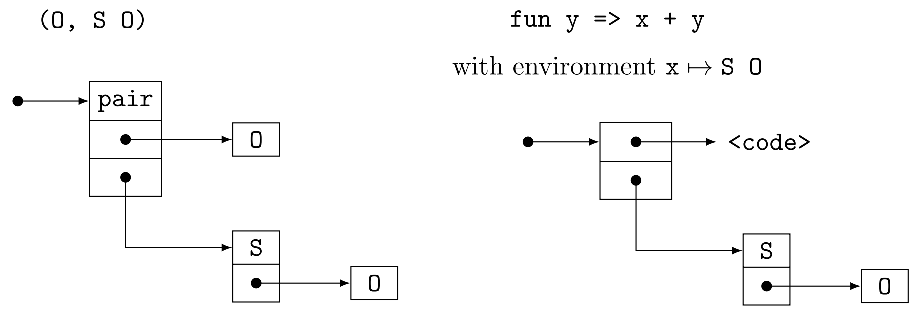

Figure 3.4: Example Cminor-level memory representations of Gallina values. The left shows the representation of the pair `(0, 1)` (represented in unary). The right shows the representation for a closure whose environment contains a single value (for the free variable `x`).

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.

3.2.3 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 3.2.1). 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 [57].

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.

3.3 Linking

Once a system is written and verified at the Gallina level, and its shim is written, it can be compiled to Assembly. For the Gallina code, it is compiled using the process described in Section 2.2 and Section 2.3, down to a Cminor program. The shim code is compiled to Cminor as well, using the CompCert frontend. The programs are then linked together into a single Cminor program using a linker written for Œuf.

In practice, this linker does little more than put the two program fragments next to each other. However, it is verified with a crucial semantic property.

Suppose we compile some Gallina definitions G to a Cminor program fragment G_{cm}. We then compile its corresponding shim S to a Cminor program S_{cm}. We then link S_{cm} and G_{cm} together to form the whole program W_{cm}. The correctness theorem of the linker states that any series of steps from any state s to any other state s' which occur within the G_{cm} program will still occur within program W_{cm}. This is precisely what is necessary, as the Œuf correctness theorem gives us an execution trace through program G_{cm}. Linker correctness allows for transformation of this execution trace into an execution trace through the whole W_{cm} program.

Theorem 3.3.1. *Let G_{cm} be some program in Cminor. Let S_{cm} another Cminor program, and W_{cm} be the result of successfully linking G_{cm} and S_{cm}. Let st and st' be program states, and t be a trace of externally visible events, such that st takes some number of steps to st' within G_{cm}, producing externally visible events t. Let lst be some program state such that it matches st (i.e. lst is identical to st, except perhaps more defined in cases where st was undefined). Given these things, we know that there exists some state lst' such that lst takes some number of steps to lst' within program W_{cm}, producing externally visible events t, and we know that st' matches lst' (i.e. lst' is identical to st', except perhaps more defined in cases where st' was undefined).*

3.4 Œuf Shim Reasoning

Given a correctly linked program, constructed from Cminor compiled from Gallina definitions as well as a shim, total system correctness is now close at hand. We must now prove that this whole program behaves as desired, but we get to reason about calls to functions defined in Gallina at the Gallina level, even within the context of a Cminor program. Once we prove that the linked Cminor program performs the desired behavior, one more lemma translates that guarantee to the assembly level.

3.4.1 Linked Program Specification

In order to verify the linked program, we first need to write down what we would like to prove about it. While, for Gallina definitions, it's fairly natural to find some relation for the input and output of functions, at the Cminor level it's a bit more subtle how one goes about specifying the desired behavior of a program. In Œuf, what we have settled on is a constructive form of correctness: informally, given a program state at the beginning of the program, that state takes a finite series of steps to a final state at the end of the program, and the execution trace produced along the way is subject to an arbitrary correctness predicate. Consider an example where the shim simply reads input, passes the input to a function defined in Gallina, then prints out the result, then the specification would state that the trace contains an input event, and an output event containing the Gallina function applied to the input. Our ability to precisely reason about input and output is limited by the CompCert representation of traces, which (in the version of CompCert on which Œuf is based) don't even contain types which would allow for input or output of truly arbitrary strings. In practice, we've axiomatized opaque *input* and *output* predicates over traces, as well as a `read_stdin` function and a `write_stdout` function, along with the fact that calls to `read_stdin` create an *input* trace, and calls to `write_stdout` create an *output* trace. This allows us to describe the behavior of our linked program in terms of these axiomatized opaque predicates, allowing us to shoehorn better support for string input and output into a system which doesn't natively support such concepts.

3.4.2 Linked Program Reasoning

Once the linked program is specified, our work is just begun. Now we must also prove that the program meets the specification. The basic principle for these proofs is simple: we construct the steps of execution one at a time. In most cases it's simple, and one tactic discharges the step. When the shim program calls `read_stdin` or `write_stdout`, the proof is a single call to an axiom. Assuming that the shim code didn't do anything too complex, we will soon reach a point in our proof where the shim calls a function defined in Gallina.

At this point we stop manually constructing the steps ourselves. We construct the preconditions to the Œuf top level theorem, which are mainly facts relating our arguments (Cminor values) to their Gallina level equivalents. Once we have the preconditions satisfied, the entire series of steps through the Œuf compiled Gallina code is simply one application of the top level Œuf correctness theorem. This leaves us in a program state immediately after returning from the call into Œuf compiled Gallina code. There is a bit of necessary unpacking of invariants in order to establish the correct shape of values once the call returns, but then the proof about execution can proceed.

The proof about execution after a call into Œuf compiled code proceeds almost identically to the proof prior to the call.

3.4.3 Assumptions and Limitations

While Œuf does expose the ability to reason about a linked program, shim code included, the support for this feature is in its infancy. There are several limitations which keep this approach from being applied to large systems. One is the axiomatic approach to input and output. In order to better support arbitrary string I/O, we would need CompCert to support it as well. Another limitation is simply the awkward nature of constructing steps by hand, and its resulting proof size. However, perhaps one of the biggest limitations is the lack of more stringent relation of the memory state before and after a call into Œuf compiled Gallina code. The current top level Œuf theorem states that the result of that call will be returned, with memory containing the correct values in such a way that they correspond to the Gallina level result of the call. However, the relation makes no guarantee about anything else present in memory before the call into Œuf compiled code. In practice, Œuf compiled code will never modify anything in place in memory prior to it being called. However, this fact has not been formally proven, and is not able to be taken advantage of during linked program verification.

3.5 WordFreq overview

The goal of Œuf was to enable development of verified systems, with the guarantees about them preserved from Gallina all the way to the generated assembly code. In order to evaluate this goal, I built the WordFreq verified system in Coq, compiled it with Oeuf, and preserved the correctness guarantee through to the generated assembly code.

Here I present the WordFreq verified system, its correctness guarantee, and the major parts of its correctness proof. I discuss the development of the system and its proof, as well as the axiomatic primitives necessary to tie it together.

3.5.1 WordFreq

The intended behavior of the WordFreq system is a simple one. When run, the program WordFreq will take, as input from standard in, a stream of text. It will then count the number of occurrences of every whitespace separated word in the stream of text, with all whitespace stripped. The program then prints these counts to counts to standard out, one per line.

3.5.2 Gallina Implementation

The Gallina implementation of the pure portion of the WordFreq system is relatively simple. The main developmental challenge is expressing every computation in terms of primitive recursors (eliminators), as opposed to using recursion. The strategy used for WordFreq is one of parallel development: first, the recursive version of a function is written, then an eliminator version, and the two are proven equivalent. While this strategy leads to what feels like a large amount of duplicated code, the entire file which contains the WordFreq system is still under 800 lines long. The main Gallina function which performs the computation for WordFreq is `calculate_frequencies_top`, which has type `list ascii` $\to$ `list ascii`.

3.5.3 Formal Specification

In order to verify the WordFreq program, we must have a specification. In this case we use two separate properties, and prove both about the Gallina implementation. These two properties are:

1. The set of words in the input is the same as the set of words in the output, i.e. a word is contained in the output list of frequencies if and only if it is contained in the input

list of words (after stripping whitespace).

2. If a word and corresponding count of occurrences occur in the output, that count is equal to the number of times that word occurs in the input.

Both these properties are proved in Coq about the `calculate_frequencies` function. The `calculate_frequencies_top` function is a thin wrapper, which calls `calculate_frequencies`, then calls a simple function to format the output into a single string.

Normally, this is where the verification process for a verified system would stop. However, Œuf allows arbitrary properties about Coq code to be formally preserved from the Gallina level down to the assembly level. This process will be detailed in later sections.

3.5.4 Shim Code

Almost all of the functional complexity of the WordFreq system is handled in Gallina. However, Gallina can only express pure computation. Our system requires effects: specifically reading from standard in, and printing to standard out. This is where the shim comes in: below is the WordFreq shim.

```
int main() {
    //read input from stdin, and convert to a string format Coq can use
    oeuf_string_t input = to_coq_string(read_stdin());

    //create a closure object through which to call into Coq code
    oeuf_closure_t calc_freq_closure = malloc(sizeof(struct closure));
    calc_freq_closure->f = calculate_frequencies_top;

    //actually call our Coq code
    oeuf_string_t freqs = calc_freq_closure->f(calc_freq_closure,input);
    //print the result to stdout
    write_stdout(of_coq_string(freqs));
    return 0;
}
```

The shim does only 5 things:

1. Read input from stdin
2. Convert that input into the format Coq expects
3. Call `calculate_frequencies_top`, compiled from Coq
4. Convert the result of the call back into a C string
5. Print the converted string to stdout

The minimal nature of the shim is no accident. As the goal is total verification of every piece of running code, we quickly encounter the reality that verification of Gallina code is immensely easier than the verification of C code.

3.5.5 Shim Verification

Using the techniques discussed in Section 3.4, I have constructed a proof that the linked program (Gallina and shim code) reads input, converts it to Coq format, calculates the word frequencies in that input, converts the resulting output frequencies back to C format, and prints that to the screen. This reasoning relies on 4 axiomatized primitives: `read_input` and `write_output` discussed earlier, as well as `to_coq_string` and `of_coq_string`. The latter two simply convert between the string format used in C, and the string format expected by Gallina compiled Coq programs. My hope in the future is to use ongoing work into Native Types in order to allow for Gallina computation over C strings directly.

3.6 Axioms and Trust

The goal of Œuf is to minimize the TCB of critical systems developed in Coq. It does just that: it leaves systems such as WordFreq with a much smaller TCB than they would have were they to be extracted using traditional Coq extraction. Gone is the trust in the OCaml or Haskell compiler. Gone is not just the trust in the Ocaml or Haskell runtime system, but the presence of a runtime at all. Gone is the trust in the semantic match up between Gallina and another similar looking functional language.

However small, a TCB still remains. As always, we rely on the soundness of the Coq proof checker, just as before. Just like any verified system, we also rely on the specification we've verified against. No verified system is protected against a buggy specification. We also rely on the unverified system assembler, to get us from assembly to machine code. Further, we rely on the semantic match up between the CompCert model of assembly, and the actual function of a microprocessor. While this semantic gap is much smaller than a whole compiler, it is still a clear and present mismatch where further problems likely lurk. We still also rely on the axiomatized I/O primitives as discussed in Subsection 3.5.5.

Chapter 4
BIT LEVEL POINTERS FOR COMPCERT

In CompCert, many correct peephole optimizations are not provably correct due to arithmetic operations which are undefined (or partially defined) over pointer values. Peephole optimizations must be correct regardless of the type of value a register contains at runtime. To support peepholes, we add a new CompCert assembly semantics where pointers are represented as concrete 32-bit integers so more operations will be defined over pointers.

This chapter motivates the need for this change, defines the new language, axiomatizes a correct memory allocator that can fail due to memory exhaustion, and describes the proof that our semantics is equivalent to the existing semantics on any execution where memory allocation always succeeds. We refer to the "existing" semantics with respect to infinite memory as $\text{ASM}_{\mathbb{Z}}$ and our "new" semantics as $\text{ASM}_{\mathbb{Z}32}$ since the change is to give pointers a bit vector representation.

4.1 Motivation

In CompCert C, the (non-aggregate) values are floats of various sizes, integers of various sizes, pointers, or the undefined value. The dynamic semantics of every intermediate language in CompCert is defined to compute over these same values, with each value containing a (runtime) type tag. These type tags are included even in $\text{ASM}_{\mathbb{Z}}$ (and retained in $\text{ASM}_{\mathbb{Z}32}$) even though an actual x86 implementation does not have them: on actual hardware, there are only bit vectors without the corresponding type tags. Thus $\text{ASM}_{\mathbb{Z}32}$ guarantees that, at runtime, no values with tag `Vptr` are present in registers or memory, all memory addresses are represented using `Vint`.

This semantic gap then propagates to the semantics of instructions. With CompCert's type tags, the instruction definition often branches on the tag of an argument's value with some branches producing the special "undefined" value `Vundef`. An example of such a semantics is for the integer-subtraction instruction:

```
Definition sub (v1 v2: val): val :=
 match v1, v2 with
  | Vint n1, Vint n2 =>
      Vint(Int.sub n1 n2)
  | Vptr b1 ofs1, Vint n2 =>
      Vptr b1 (Int.sub ofs1 n2)
  | Vptr b1 ofs1, Vptr b2 ofs2 =>
      if eq_block b1 b2
      then Vint(Int.sub ofs1 ofs2)
      else Vundef
  | _, _ => Vundef end.
```

In actual x86, this subtraction instruction always performs subtraction on bit vectors. In the definition above, several cases instead produce `Vundef`, such as if an operand is a float or if the two operands are pointers into different blocks. The $\mathrm{ASM}_{\mathbb{Z}}$ semantics is sufficient because (1) legal C source programs will not depend on undefined-values[1] and (2) when undefined-values are not produced, the semantics aligns with actual x86 just with the addition of type tags and with a higher-level representation of pointers using a block and an offset (see Section 4.2).

Now consider this peephole optimization, also used in Section 5.1, that uses bitwise negation to replace subtraction.

```
sub %eax, %ecx           notl %eax
mov %ecx, %eax   =====>  add %ecx, %eax
dec %eax
```

[1]The subtraction example in particular matches C's semantics: Subtraction of two pointers returns a defined value only if the two pointers are into the same block of memory.

The semantics for taking the bitwise-negation of a pointer is `Vundef` since there is no reasonable way to bit-negate a pair of a block (represented as a $\mathbb{Z}$) and an offset (represented as a $\mathbb{Z}_{32}$), nor is the operation defined on C pointers. But now under this semantics, the peephole optimization is *not* meaning-preserving due to program states where an operand may be a pointer. A peephole optimization must be correct for all (types of) operands since by this stage in the compiler, static type information has been erased. In this case, if type information had not been erased, the information available in CompCert still cannot distinguish between pointers and integers. Hence the proof of correctness for a common integer optimization, for example, must include a case for registers that contain floating point values, and registers with undefined values, and so on.

After pursuing several design alternatives, we concluded that the best small-but-sufficient design change was to introduce a lower-level semantics that still has type tags but also gives pointers a bit vector representation. Two computations over pointer values are equivalent as long as they produce the same bit vector for every input. As described in the remainder of this chapter, we axiomatize how a memory allocator must behave in terms of the bit vectors it returns as pointers in order for $\text{ASM}_{\mathbb{Z}32}$ to be equivalent to $\text{ASM}_{\mathbb{Z}}$. We need to take particular care not to introduce a false axiom because there are finitely many 32-bit bit vectors whereas $\text{ASM}_{\mathbb{Z}}$'s semantics assumes an infinite memory, so any claim of a bijection between the two representations of pointers is logically inconsistent. Moreover, our finite range of pointer values necessitates allowing for memory exhaustion (i.e., failed allocation), a possibility not previously considered in CompCert.

4.2 Converting Between Memory Addresses and Pointer Values

In $\text{ASM}_{\mathbb{Z}}$ and $\text{ASM}_{\mathbb{Z}32}$, a *memory address* is a `block` $\times$ `offset` pair, where a `block` is a positive integer, unbounded in size, and an offset is a 32-bit unsigned integer. In $\text{ASM}_{\mathbb{Z}}$, a *pointer value* is a memory address, but in $\text{ASM}_{\mathbb{Z}32}$ a pointer value is a 32-bit integer. The two semantics use the same memory model with an infinite memory of blocks, thus simplifying the translation from $\text{ASM}_{\mathbb{Z}}$ to $\text{ASM}_{\mathbb{Z}32}$.

Performing memory operations in ASM$_{\mathbb{Z}32}$ requires conversion between 32-bit integers and memory addresses, i.e. a memory allocator. We use `pinj` ("pointer inject") to map from a memory address to a pointer value and `psur` ("pointer surject") to map from a pointer value to a memory address. As discussed below, these functions are defined as opaque axioms subject to constraints. They have these signatures:

$$\texttt{pinj} : \texttt{AS} \rightarrow \texttt{block} \rightarrow \texttt{int}_{32} \rightarrow \texttt{option int}_{32}$$

$$\texttt{psur} : \texttt{AS} \rightarrow \texttt{int}_{32} \rightarrow \texttt{option (block} * \texttt{int}_{32}\texttt{)}$$

The option in `pinj`'s return type models memory exhaustion and is crucial to avoid axiomatizing false by claiming that an infinite set can map injectively to a finite set. `psur` has a return type of option as well, which lets us axiomatize that any successful `psur` implies a successful `pinj`. To model these mappings changing over the execution of the program (as pointer locations are reused via malloc and free), we add an additional argument to `pinj` and `psur`, representing the current allocator state (`AS`). This allocator state is kept opaque, and is manipulated by `malloc` and `free` opaque functions to produce modified allocator state.

We axiomatize the propagation of the opaque allocator state. A single object of this type represents an initial (empty) allocator state, `init : AS`. We also assume the initialized allocator state corresponds to the initial memory when `main` begins to execute. We axiomatize functions representing allocate and free actions which transform the allocator state. $\texttt{alloc} : \texttt{AS} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \texttt{block} \rightarrow \texttt{AS}$ and $\texttt{free} : \texttt{AS} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \texttt{block} \rightarrow \texttt{AS}$. In CompCert, allocate and free operate on ranges of offsets within blocks. In order to support this, `alloc` and `free` each take two $\mathbb{Z}$ arguments, which represent the range within the block. Likewise, we axiomatize `as_ec` and `as_ec'`, which record external calls in the allocator state (corresponding to CompCert's `external_call` and `external_call'` respectively).

Further, we define what it means for a memory and an allocator state to *match*. Specifically, the empty memory and `init` match. A new memory produced from an allocate action matches a new `AS` produced from an `alloc` action (with the same arguments and results), provided the previous memory and `AS` matched. A new memory produced from a free action matches a new `AS` produced from a `free` action (with the same arguments), provided the previous memory and `AS` matched. A new memory produced from some action that does not allocate or free (e.g., store) matches any `AS` the previous memory matched. Again, matching allocator state and memory is similarly defined over external calls. Finally, we say that a particular `AS` *extends* another `AS` if one was produced in some number of steps from the other.

To reason about these opaque functions, which map memory addresses to $\text{ASM}_{\mathbb{Z}32}$ pointer values, we use a set of axioms about `pinj` and `psur` Figure 4.1.

The semantics for memory-load and -store instructions uses `pinj` and `psur` to manipulate registers holding bit vectors that describe memory addresses. Additionally, in $\text{ASM}_{\mathbb{Z}}$, a new memory allocation generates a pointer to a new block of memory in a simple way: a global counter of blocks is updated, and the new fresh block is used to generate pointers into the memory. In $\text{ASM}_{\mathbb{Z}32}$, the same approach is used, but the result pointers are injected to bits using `pinj`.

In $\text{ASM}_{\mathbb{Z}}$, arithmetic operations produce fewer undefined results for 32-bit integers than for pointers. That is, if an operation does not produce `Vundef` for pointer operands, then it does not produce `Vundef` for int operands. Therefore, $\text{ASM}_{\mathbb{Z}32}$ is strictly more defined than $\text{ASM}_{\mathbb{Z}}$, as everywhere $\text{ASM}_{\mathbb{Z}}$ will not produce `Vundef`, $\text{ASM}_{\mathbb{Z}32}$ will not produce `Vundef`. However, $\text{ASM}_{\mathbb{Z}32}$ will not produce `Vundef` in additional cases such as bitwise-negation of a pointer value, which is the entire purpose of $\text{ASM}_{\mathbb{Z}32}$.

4.3 Verification

4.3.1 Axiomatizing Memory Allocation

Conceptually, the system memory allocator (e.g., `malloc`, `alloca`, or function stack frames) is an instantiation of `pinj` and `psur`: Allocating a new memory block amounts to assigning it a pointer-value (its position in the virtual address space). We do not define our semantics

1. `pinj` always remembers a mapping:

   $\forall\ as, \forall\ as', \forall\ b, \forall\ o, \forall\ bits,$

   $\texttt{pinj}\ as\ b\ o = \texttt{Some}\ bits \rightarrow \texttt{extends}\ as\ as' \rightarrow$

   $\texttt{pinj}\ as'\ b\ o = \texttt{Some}\ bits.$

2. `pinj` allows for pointer arithmetic:

   $\forall\ as, \forall\ b, \forall\ o, \forall\ bits, \texttt{pinj}\ as\ b\ o = \texttt{Some}\ bits \rightarrow$

   $\forall x, \texttt{pinj}\ as\ b\ (o + x) = \texttt{Some}\ (bits + x).$

3. Null is always invalid:

   $\forall\ as, \forall\ m, \texttt{match}\ as\ m \rightarrow$

   $\forall\ b, \forall\ o, \texttt{pinj}\ as\ b\ o = \texttt{Some NULL} \rightarrow$

   $\texttt{valid}\ m\ b\ o = \texttt{false} \wedge \texttt{valid}\ m\ b\ (o - 1) = \texttt{false}.$

4. `psur` is equivalent to `pinj` and weakly valid:

   $\forall\ as, \forall\ m, \texttt{match}\ as\ m \rightarrow \forall\ b, \forall\ o, \forall\ bits,$

   $\texttt{psur}\ as\ bits = \texttt{Some}\ (b,o) \longleftrightarrow$

   $(\texttt{pinj}\ as\ b\ o = \texttt{Some}\ bits \wedge \texttt{weakvalid}\ m\ b\ o = \texttt{true}).$

5. `pinj` supports pointer comparison:

   $\forall\ as, \forall\ b, \forall\ o1, \forall\ o2, \forall\ bits1, \forall\ bits2,$

   $\texttt{pinj}\ as\ b\ o1 = \texttt{Some}\ bits1 \rightarrow \texttt{weakvalid}\ m\ b\ o1 = \texttt{true} \rightarrow$

   $\texttt{pinj}\ as\ b\ o2 = \texttt{Some}\ bits2 \rightarrow \texttt{weakvalid}\ m\ b\ o2 = \texttt{true} \rightarrow$

   $(o1 < o2 \longleftrightarrow bits1 < bits2).$

Figure 4.1: All of the axioms constraining `pinj` and `psur`, except those related to external calls, which are used to verify the forward simulation from $\text{ASM}_{\mathbb{Z}}$ to $\text{ASM}_{\mathbb{Z}32}$.

in terms of an actual instantiation of `pinj` and `psur`, but instead we define our required specification of the allocator. (And since this is only the semantic definition, we do not actually need one. Like in CompCert prior to our work, the compiled binary is linked against the ordinary standard (unverified) library for C.)

Additionally, ASM$_{\mathbb{Z}32}$ includes the axiom that any external call behavior is duplicated when all pointer values in the registers and memory have been converted via `pinj` to bits, and that the allocator state produced by these two calls is equal.

These axioms comprise the specification of a memory allocator, as shown in Figure 4.1. Any allocator must satisfy all of the properties above, and any allocator satisfying those properties is a valid system allocator for running assembly programs. For example, since pointer arithmetic exists at the assembly level, the allocator must preserve the behavior of that arithmetic (2); the allocator must not use the `NULL` address for anything, as programs rely on that address being invalid (4); and `pinj` and `psur` must be inverses of each other, as it is essential for the semantics to correctly pack/unpack pointer values (5).

Memory deallocation (i.e, `free`) deserves brief discussion. Because CompCert's correctness guarantee is only for legal C programs and programs executing dangling-pointer dereferences are illegal, their behavior is not directly relevant. In ASM$_{\mathbb{Z}}$ and ASM$_{\mathbb{Z}32}$, `free` marks a block of memory as unusable and subsequent access produces `Vundef`. More interesting, and unconsidered in ASM$_{\mathbb{Z}}$ due to infinite memory, is needing to *reuse* pointer values. Our explicit handling of allocator state has allowed us to build, to our knowledge, the first formally verified system with a specification of a memory allocator which allows for memory reuse.

4.3.2 Equivalence of ASM$_{\mathbb{Z}}$ and ASM$_{\mathbb{Z}32}$

ASM$_{\mathbb{Z}32}$ is a new semantics over the same syntax for which ASM$_{\mathbb{Z}}$ is defined. To maintain CompCert's correctness guarantee, a bisimulation proof between ASM$_{\mathbb{Z}}$ and ASM$_{\mathbb{Z}32}$ is required. We have constructed this proof, *assuming that pointers produced in the program by ASM$_{\mathbb{Z}}$ are injectable to bits:* i.e., do not return `None` when passed to `pinj`. If `None` is produced, this corresponds to a memory-allocation failure, which is allowed in C but not modeled by $\mathrm{ASM}_{\mathbb{Z}}$ or any of CompCert's higher-level languages. More formally, we assume that any pointer to the first byte of all allocated blocks in any program state reachable (in some finite number of steps) from the initial state does not return `None` when passed to `pinj`.

We only guarantee forward simulation in cases where memory allocation doesn't fail: a guarantee morally equivalent to the existing CompCert guarantee. Our approach to injecting pointers does not, in essence, add axioms to CompCert so much as it makes explicit certain assumptions which existed implicitly in the semantic gap between CompCert's x86 backend semantics, and the actual behavior of an x86 chip.

In order to argue forward simulation from $\mathrm{ASM}_{\mathbb{Z}}$ to $\mathrm{ASM}_{\mathbb{Z}32}$, the definition of matching register sets is straightforward: If a register contains a value other than a pointer or undefined in $\mathrm{ASM}_{\mathbb{Z}}$, it contains that same value in $\mathrm{ASM}_{\mathbb{Z}32}$. If a register contained a pointer in $\mathrm{ASM}_{\mathbb{Z}}$, it contains the corresponding integer in $\mathrm{ASM}_{\mathbb{Z}32}$. If a register contained the undefined value in $\mathrm{ASM}_{\mathbb{Z}}$, it could contain any value (other than a pointer) in $\mathrm{ASM}_{\mathbb{Z}32}$.

4.4 Related work

Infinite to Finite Memory Model Translation. We are not the first to formalize the translation of pointers from high-level infinite memory models to low-level finite memory models. Kang et. al. built and verified what they call a quasi-concrete memory model, which addresses the gap between infinite and finite memory models [25]. Their model injects high level pointers to low level bits on demand, as those pointers are cast to integers. If a pointer is never cast, it is never injected. Further, they give semantics to their injection (or as they call it, concretization), whereas we leave the specification for the allocator opaque. Recent work to develop a concrete memory allocator for CompCert [8] verifies translations against a simple, conservative memory allocator which lacks the ability to reuse memory. Our memory allocator interface allows for memory allocators that can reuse previously freed memory. The CompCertTSO [61] project built finite memory into every level of CompCert, dealing with many similar issues to us, but never explicitly translating from infinite to finite memory. This translation required our novel memory-allocator axiomatization and an equivalence theorem up to the first allocation failure.

Chapter 5

PEEK

Transformations over assembly code are common in many compilers. These transformations are also some of the most bug-dense compiler components. Such bugs could be eliminated by formally verifying the compiler, but state-of-the-art formally verified compilers like CompCert do not support assembly-level program transformations. Here I present Peek, a framework for expressing, verifying, and running meaning-preserving assembly-level program transformations in CompCert. Peek contributes three new components: a liveness analysis, a library for expressing and verifying peephole optimizations, and a verified peephole optimization pass built into CompCert. Each of these is accompanied by a correctness proof in Coq against realistic assumptions about the calling convention and the system memory allocator, and leverage the lower level semantics presented in the previous chapter.

Verifying peephole optimizations in Peek requires proving only a set of local properties, which my collaborators and I have proved are sufficient to ensure global transformation correctness. We have proven these local properties for 28 peephole transformations from the literature. Here I discuss the development of our new assembly semantics, liveness analysis, representation of program transformations, and execution engine; describe the verification challenges of each component; and detail techniques we applied to mitigate the proof burden.

5.1 Overview

Background and Example The peephole optimizations we consider operate over compiler-generated assembly code, i.e., at the lowest-level language used in the compiler. These transformations can take advantage of machine-specific information and can clean up inefficiencies introduced during the translation to assembly.

As a simple-but-clever example, consider this rewrite for x86 assembly[1] from the literature [6]:

```
sub %eax, %ecx              notl %eax
mov %ecx, %eax   =====>     add %ecx, %eax
dec %eax
```

We call the code on the left the *find-pattern* and the code on the right the *replace-pattern*. This transformation corresponds to replacing the expression `x = y - x - 1` with `x = y + ~x` where `x` and `y` are twos-complement 32-bit integers initially in registers `%eax` and `%ecx` respectively and `~` is bitwise-negation. While it could be applied in any intermediate representation (that has bitwise negation), doing so misses opportunities since the pattern can arise in generated code for operations like indexing into memory structures—code not explicit in higher-level representations. Other examples of peephole optimizations may use assembly-language features (e.g., vector instructions) unavailable at higher levels.

To perform a peephole optimization, a compiler scans the assembly code for the find-pattern (allowing various permutations of registers) and replaces it with the replace-pattern. Compilers typically have many different peephole optimizations, with an interface for expressing new ones. An execution engine performs the scanning-and-replacing for any peephole optimization.

Our peephole-optimization framework for CompCert follows this typical architecture, with proofs that (1) each peephole optimization is correct and (2) the full system is correct if each peephole optimization is correct.

Local Correctness and the Role of Liveness Informally, we reason that a peephole optimization is correct if the find-pattern and the replace-pattern act identically on the machine-state: for all states, applying either instruction sequence results in the same state.[2]

[1]For the duration of the thesis, we will use GAS syntax.

[2]As a technical detail, this clearly cannot hold if the program text is part of the state, which is relevant in the presence of self-modifying code, programs that observe the size of the binary or the exact value of

However, requiring precisely identical state is too strong: registers in an assembly program are often *dead*, meaning they are not read before next being written. An optimization need not preserve the values in such registers. In our example above, the transformation does not preserve the value in `%ecx`, so it is valid only in contexts where `%ecx` is dead.

In unverified compilers, peephole optimizations can often assume certain registers are dead, using knowledge about compiler-specific code-generation details. Examples include knowing that certain registers may only be used for temporary values, or flag registers are never live across a branch. Our verified system has no such luxury, so each peephole optimization lists the set of registers that must be dead for it to apply. Furthermore, our framework includes a verified intraprocedural static liveness analysis over the assembly code to produce a sound approximation of which registers must be dead at each program point. The execution engine uses information from this liveness analysis to apply a peephole optimization only when the registers it requires to be dead are, in fact, dead.

A Lower-Level Assembly Semantics Every intermediate language in CompCert has a dynamic semantics. The compiler does not execute these semantics; it only generates assembly code to be run later. The semantics are instead used to prove that each optimization or translation to a lower-level language preserves meaning. The semantics for the highest- (C) and lowest-level language (x86) are particularly important because they are *trusted* to adequately represent the C language definition and the x86 language definition.

As explained in Chapter 4, there is a semantic gap between CompCert's definition of 32-bit x86 assembly and reality. CompCert's x86 semantics represents pointers as a pair of a memory block and an offset, not as the 32-bit integers used in reality. Blocks are represented by mathematical integers, and memory is assumed to be infinite.

CompCert's x86 semantics correspondingly produces undefined values for bitwise (and other) operations on pointers. With such a semantics, the example transformation above

the program counter, etc. Compilers can ignore these issues because such techniques for inspecting the binary are unavailable to well-defined source programs.

is invalid when `%eax` holds a pointer. But in reality, this restriction is unnecessary: the optimization is correct for any register contents, even if the bits happen to represent a pointer. Therefore, we wish to prove such optimizations correct without concern for the type of value in the register.

To achieve this we use the new lower-level x86 semantics ($\text{ASM}_{\mathbb{Z}32}$) described in Chapter 4, in which pointer values are 32-bit integers. This allows more peephole optimizations to be proven correct.

Global Correctness To maintain CompCert's compiler-correctness proof, we must show that applying any (locally) correct peephole-optimization preserves *global* program meaning. This proof contains three orthogonal components, each discussed in detail Section 5.3:

- The liveness analysis is sound: any register identified as dead at a program point must actually be dead whenever execution reaches that program point.
- If a peephole optimization is locally correct, then any use of it at any place in the program would preserve program meaning.
- The execution engine performs peephole optimizations correctly: it performs a rewrite as specified and only at program points where the optimization is valid.

Each of these proofs uses the $\text{ASM}_{\mathbb{Z}32}$ x86 assembly semantics, as discussed in Chapter 4.

Verification and Assumptions. All our proofs are in Coq, using the existing CompCert framework. Our liveness analysis assumes (but does not verify) that all code obeys the C calling convention (see Section 5.3.3 for details). Our semantics for memory allocation axiomatizes assumptions about the memory allocator rather than verifying an actual memory allocator. We believe these assumptions about the calling convention and memory allocator could be discharged independently of our work.

5.2 Verifying Individual Peepholes

Peek eases the proof burden for formally verifying peephole optimizations by identifying a set of *local properties* that are practical to prove for an optimization yet sufficient to establish the global correctness of applying any optimization. This section details the three parts of such records: *Peephole Data*, which specifies the transformation, *Local Correctness*, which proves the transformation satisfies the local correctness properties, and *Side Conditions*, which establish auxiliary properties used in the global correctness proof.

5.2.1 Peephole Data

A *concrete* peephole transformation is defined by a list ℓ of x86 instructions to find, and a list r of x86 instructions to replace them with, and 3 sets of registers: U, D, and T. U lists all registers that ℓ and r read, D lists all registers that ℓ and r update (with equal values), and T lists all registers that ℓ and r may leave with different contents (i.e. scratch registers the peephole can trash). This set is necessary for peepholes for which the r pattern uses fewer registers than the ℓ pattern, such as converting 3 move instructions to an exchange. Thus, a register that contains an input value to the peephole and also later contains an output value, could be in both U and D, and any register that is written by either ℓ or r must be in D or T. The registers in U and D could be computed with static analysis, but Peek currently requires the peephole writer to specify them, which we have found easy to do in practice.

The problem with concrete transformations is that we would need an almost-identical transformation for every combination of registers. Instead, we want *parameterized* peephole transformations that can be instantiated with actual registers to produce concrete transformations. A parameterized peephole transformation is a function from register names to a concrete peephole transformation. Verifying a parameterized peephole transformation requires proving that, for any instantiation of register names, the resulting concrete peephole transformation will be correct as described the following subsections. As described later in Section 5.3, rewriting with a parameterized peephole transformation requires computing a substitution to instantiate the register name parameters in the rewrite.

5.2.2 Local Correctness

In Peek, a concrete peephole transformation is correct if ℓ and r always terminate and make equal updates to all live locations, assuming they begin in states that agree on all live locations. Control is implicitly preserved by this definition, because the program counter is always live. *Local correctness* establishes that ℓ and r make equal updates to all registers in D and equal updates to all memory locations, assuming that they begin in states that agree on all registers in U. Let P be the set of registers that do not appear in U, D, or T; these are registers that neither ℓ nor r read or write and are thus preserved by the peephole. The Peek execution engine "ties the knot" by computing the sets of live locations L_{entry}, L_{exit} at the entry and exit of rewritten locations respectively and checking that both $U \subseteq L_{entry}$ and $L_{exit} \subseteq D \cup P$. Note that because local correctness requires equal memory updates from ℓ and r, we do not compute liveness for memory locations.

Let $\sigma \overset{c}{\rightsquigarrow} \sigma'$ hold whenever a state σ executes through code fragment c (within a larger program) to yield state σ' and let $\sigma_1 \overset{L}{\sim} \sigma_2$ hold when σ_1 and σ_2 agree (as defined in Section 5.3) on all locations in L. Then local correctness can be proved by showing (1) *local simulation* that holds for any larger program context ℓ and r may appear in:

$$\forall\, \sigma_\ell\, \sigma'_\ell\, \sigma_r,\ \sigma_\ell \overset{U}{\sim} \sigma_r \wedge \sigma_\ell \overset{\ell}{\rightsquigarrow} \sigma'_\ell \implies$$
$$\exists\, \sigma'_r,\ \sigma'_\ell \overset{D}{\sim} \sigma'_r \wedge \sigma_r \overset{r}{\rightsquigarrow} \sigma'_r$$

and (2) *ℓ-normalization* by providing a measure m from machine states to natural numbers satisfying:

$$\forall\, \sigma_\ell\, \sigma'_\ell,\ \sigma_\ell \overset{\ell}{\rightsquigarrow} \sigma'_\ell \implies m(\sigma'_\ell) < m(\sigma_\ell)$$

For intuition, *ℓ-normalization* is simply a proof that any execution which enters the ℓ pattern will not diverge within it.

Consider proving local simulation for the example in Section 5.1: σ_l and σ_r have the same initial values in `%eax` and `%ecx`, call them a_0 and c_0. For σ_l, the final value in `%eax` is $c_0 - a_0 - 1$. For σ_r, the final value in `%eax` is $c_0 + \tilde{}a_0$. As D contains only `%eax`, local simulation amounts to proving $c_0 - a_0 - 1 = c_0 + \tilde{}a_0$, which holds for 32-bit ints.

Because we require only a local simulation, peephole optimizations cannot rely on any properties of the rest of the program, including position in the program, invariants about values in particular registers, etc. This is the essence of what makes an optimization a *peephole* optimization. This interface provides modularity by allowing the rewrite to be carried out in many contexts. Section 5.3 details our proof showing that for any context, local simulation relations are sufficient to establish global optimization correctness.

Regarding ℓ-normalization, we currently support the common case where ℓ contains only forward jumps (or no jumps). Peek provides a default measure (difference between PC and end of the rewrite), and a proof that this measure implies normalization. Supporting rewrites that contain backwards jumps would require manipulating a measure construction provided by the optimization implementer.

Note that there is no explicit proof that the r pattern normalizes. However, the local simulation relation already accomplishes this, since if the ℓ pattern normalizes, then there is some finite series of steps from the beginning to the end of the ℓ pattern. The local simulation relation relates those steps to a finite series of steps from the beginning to the end of the r pattern. Thus the r pattern also normalizes.

5.2.3 Side Conditions

The global proof of correctness (Section 5.3) assumes rewrites satisfy the following side conditions: the length of ℓ is the length of r, [3] and that the final instruction in ℓ is not a label.[4] Furthermore, Peek requires that ℓ and r do not contain any call or return instructions, which implies that they do not modify the global program trace by making a system call. In

[3]This restriction could be relaxed with a pre and post pass to insert and/or remove `NOP` instructions. We have not verified `NOP` insertion or deletion, but we have defined the CompCert pretty printer to print `NOP` instructions as the empty string, effectively implementing `NOP` removal.

[4]This last detail arises from CompCert's choice to make labels themselves instructions and to model a jump to a label as transferring control to immediately after the label.

practice, this condition could be relaxed but no peephole transformation we found violated the constraint and this condition eases the global correctness proof. All the above conditions are decidable, and Peek includes decision procedures or tactic support for automating their proofs.

5.2.4 Local Peephole Verification

As described above, the framework requires both a termination measure and a local simulation. In practice, simply using the number of instructions remaining in a peephole suffices for the measure of all peepholes we have verified. Proofs of local simulation are, on the other hand, very specific to each peephole, but with some common elements. They naturally decompose into two parts: (1) proving the right side can step (given the steps on the left), and (2) proving the resulting values computed by the right and left are equivalent.

To ease proving (1), we developed lemmas and custom Ltac tactic support. Without this support, naively constructing a single step on the right side can take between 60 and 100 lines of Ltac and exhausted available memory on some optimizations as short as 4 instructions. By developing tactics specifically related to correlating the resulting left and right program states, a single tactic often suffices for each program step, typically taking under a second and imposing reasonable memory overhead. Furthermore, this tactic support can typically automatically prove that the right-hand-side can step, allowing the optimization writer to focus on (2) and show that the optimization is meaning preserving.

5.3 Verifying Peek

Section 5.2 discussed the local properties sufficient for a correct peephole transformation. This section connects these local properties to global program execution. As noted in previous sections, liveness plays a key role in verifying Peek. The high level approach to the main correctness result is implementing a liveness analysis that computes the set of live registers at every program point, proving that any transformation that preserves the values of these registers also preserves global program meaning, and then proving that any application of the execution engine to apply a locally correct peephole will preserve live locations and therefore global program meaning. We detail each of these components and their proofs below.

5.3.1 Liveness

A peephole's local correctness properties imply global correctness only when the liveness information at a rewritten program location guarantees that all the inputs (registers containing values used in the peephole) are live and that all live locations at the output that are updated by the find pattern are refined by the replace pattern as detailed below.

Our liveness analysis computes a function `live` mapping program locations to a set containing all live registers at that location. We prove that a program's behavior depends only on the values stored in registers marked as live. We say y *refines* x if either x is undefined or $x = y$. Further, if for each register r in L, the value contained in register r in σ_2 refines the value contained in r in σ_1, we write $\sigma_1 \overset{L}{\sim} \sigma_2$. If $\sigma_1 \overset{L}{\sim} \sigma_2$, then for any correct liveness set L, program executions from these states will generate equivalent traces of externally visible events. Let $\sigma(r)$ denote the value of register r in state σ. We show that any transformation preserving liveness also preserves program behavior by proving this simulation relation:

$$\begin{aligned} &\forall\, \sigma_1\, \sigma_1'\, \sigma_2, \quad (\sigma_1 \overset{\texttt{live}(\sigma_1(\texttt{PC}))}{\sim} \sigma_2 \quad \wedge \quad \sigma_1 \leadsto \sigma_1') \implies \\ &\quad \exists\, \sigma_2', \quad \sigma_1' \overset{\texttt{live}(\sigma_1'(\texttt{PC}))}{\sim} \sigma_2' \quad \wedge \quad \sigma_2 \leadsto \sigma_2' \end{aligned}$$

5.3.2 Liveness Implementation

Peek's liveness analysis is an iterative dataflow analysis. The set of live registers is initialized to $\emptyset$ at each program point, and an update function is iteratively applied using a standard worklist algorithm until the liveness information reaches a fixed point with respect to the update function. The liveness at return instructions is fixed to callee-saved registers and the register containing the return result. Likewise, the liveness at function call sites is fixed to caller-saved registers.

The proof that the calculated liveness information meets the above specification naturally decomposes into two parts. First, we prove that if iteration of the update function halts, then the result is a fixed point of the update function. Specifically, the calculated liveness information is invariant to future applications of our update function. Second, we prove that liveness information that is at a fixed point with respect to the update function is a simulation relation (as above). To prove this, we show that our static approximation to control flow is sound, and that our static approximation to which registers each instruction uses and defines is sound. These two facts compose nicely into a simulation relation.

5.3.3 Calling Convention

The correctness of the liveness analysis depends on five facts that comprise the x86 calling convention that CompCert uses. They are true for CompCert-generated code, but we have not verified them. These are necessary for proving the intraprocedural analysis is interprocedurally correct, as the given CompCert semantics for call and return instructions are defined quite liberally, and do not provide many guarantees.

1. Any step that executes a call instruction steps to the beginning of a block (i.e., the beginning of a function).

2. Any step that returns from a function steps to an instruction right after some call.

3. All non-callee-save registers are dead when executing a call instruction. (CompCert passes all function parameters on the stack.)

4. All registers except the callee-save registers and the return-value register are dead when executing a return.

5. When returning, the location of the return value assumed by the returning function matches the location of the return value assumed by the caller.

5.3.4 Global Peephole Optimization Correctness

The liveness analysis and local peephole properties come together to form a correctness proof of global program transformation. The proof of global correctness takes a local peephole simulation relation and forms a global program simulation relation with input from the liveness analysis. We use CompCert's approach for proving program transformations correct. In CompCert, a transformation is correct if there is a bisimulation between the original and transformed programs. In practice in CompCert, we prove only a forward simulation for each transformation and use the fact that each intermediate language's semantics is deterministic to construct a backward simulation. Therefore, given two programs p_1 and p_2, we need to show that all possible behaviors of p_1 are possible behaviors of p_2.

A forward simulation shows that if state σ_l in the original program matches some state σ_r in the transformed program, and σ_l can take a step to σ_l', then there is some state σ_r' that σ_r can step to, and the resulting states σ_l' and σ_r' match. This relation can be transitively composed to prove that any executions of arbitrary length agree between the original and transformed program.

For proving peephole rewrites correct globally, we partition program states as follows. A state σ is *outside* a rewrite if the program counter points to an instruction that was not rewritten by the transformation. A state σ is *at entry* if the program counter points to an instruction that is the first instruction that was rewritten by the transformation. A state σ is *inside* if its program counter puts it at an instruction that was rewritten, but is not the first instruction in the rewrite, and there is a series of steps from some other state σ_0 to σ, where σ_0 is *at entry*, and those steps occur within the rewritten region. (See Figure 5.2.)

It remains to define this matching relation between σ_l and σ_r to show that any locally correct peephole can be applied to produce a new program that (forward) simulates the original program. σ_l and σ_r are in the relation if any of the following hold:

- *match out*: σ_l and σ_r are both at the same program location, that program location is *outside* as defined above, σ_l and σ_r contain the same values in all live registers, and σ_l

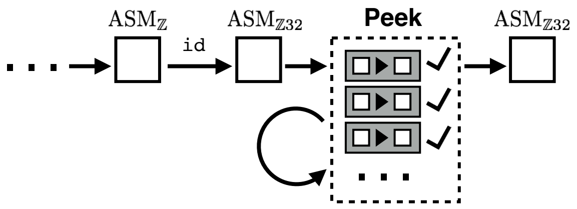
$\mathrm{ASM}_{\mathbb{Z}}$
id
$\mathrm{ASM}_{\mathbb{Z}32}$
Peek
$\mathrm{ASM}_{\mathbb{Z}32}$

Figure 5.1: **Peek Backend** CompCert produces code in $\mathrm{ASM}_{\mathbb{Z}}$. We prove a forward simulation to establish that a program will have equivalent behavior under $\mathrm{ASM}_{\mathbb{Z}32}$. Peek repeatedly applies peephole optimizations from a set of verified rewrites to the code.

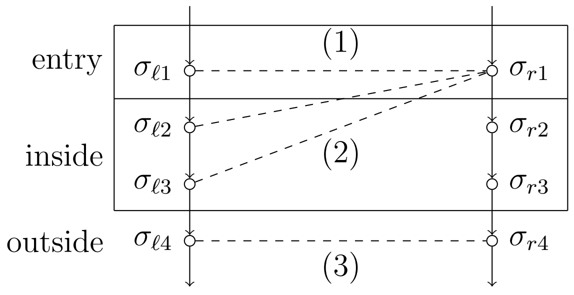

Figure 5.2: Match States Cases: At the beginning of the rewritten region (1), states match when at the same code location. As σ_ℓ steps through the left side of the rewritten region, it repeatedly matches σ_{r1} (2). Once σ_ℓ exits the rewritten region, it again matches at identical code locations (3).

and σ_r have identical memory.

- *match entry*: σ_l and σ_r are both at the same program location, that program location is *at entry* as defined above, σ_l and σ_r contain the same values in all live registers, and identical memory.

- *match in*: σ_l is *inside* as defined above, σ_0 and σ_r contain the same values in live registers, and the same memory.

The entry and outside cases, while similar, are kept distinct to allow for the **Step Cases** lemma below. To apply a peephole transformation, we also need to prove that control flow enters the rewritten region only at the beginning of the rewrite and exits only at the end. Without this property, very few classical peephole transformations are valid. Proving this fact is surprisingly subtle in x86, as some instructions have semantics that simply store the value of a register into the program counter and thus could jump to arbitrary locations.

We prove this control-flow property via two lemmas: *single entry* states that any execution step that would enter the rewritten region always steps to the first instruction, and *single exit* states that any execution step that would exit the rewritten region steps to the instruction immediately following the rewritten region. Using these lemmas, we characterize the scenarios a program can be in after taking a step:

Step Cases Lemma: If $\sigma \rightsquigarrow \sigma'$, then (1) if σ is *outside*, then σ' is *outside* or *at entry*, and (2) if σ is *at entry* or *inside*, then σ' is *inside* or *outside*.

This lemma then allows the simulation proof to proceed by handling the corresponding cases. The following theorem is the proof of global correctness.

Theorem: If σ_l matches σ_r, and $\sigma_l \rightsquigarrow \sigma_l'$, then either there is some σ_r' such that $\sigma_r \rightsquigarrow \sigma_r'$ and σ_l' matches σ_r', or σ_l' matches σ_r and a measure decreases from σ_l to σ_l'.

Proof Sketch:

- If σ_l is *outside*, and σ_l' is *outside* or *at entry*, the proof follows from the correctness of liveness.

- If σ_l is *at entry* or *inside*, and σ_l' is *inside*, the proof shows that σ_l' matches σ_r and the measure decreases. The proof of the measure decreasing comes from local properties in the verified rewrite rule.

- If σ_l is *at entry* or *inside*, and σ_l' is *outside*, we first appeal to the *single exit* property to show that σ_l' is precisely at the end of the rewritten region. Next, we unfold the definition of *inside*, collect the series of steps from our original σ_0 to σ_l', and use that series of steps to leverage the correctness of the rewrite.

5.3.5 Parameterized Rewrites

Recall parameterized rewrites work for any set of appropriate registers whereas concrete rewrites specify exact assembly registers. We use higher-order functions to get the expressiveness of parameterized rewrites with the verification burden of concrete rewrites as follows: The optimization writer actually writes a function that given an instruction sequence (by the optimization-execution engine) determines if registers can be assigned to produce a verified concrete transformation that matches the instruction sequence. If so, the particular correct concrete transformation corresponding to the code found in the program is produced. The proofs of correctness of these transformations are done once and for all, with abstract registers and constraints over them. Once a concrete rewrite is needed, the particular proof is simply instantiated with the given concrete registers and appropriate proofs of register constraints. Note the function that attempts to produce the concrete transformation need not be verified itself — if it wrongly misses an opportunity to produce a concrete transformation no unsoundness results, and the type system guarantees that the optimization writer cannot write a matcher that ever successfully produces a concrete transformation that is not correct. In practice, writing a parameterized transformation (which is what peephole transformations in conventional compilers actually are) is hardly more difficult than writing a concrete transformation: one simply pattern-matches on an instruction sequence and checks register-name constraints. We include tactic support to further minimize the effort.

5.4 Evaluation

Peek is implemented and verified as an extension of CompCert version 2.4. We discuss our implementation and evaluation of it in these terms:

- The complexity of Peek's implementation, proof, and trusted computing base
- The complexity of proving particular peephole optimizations correct, including the variety of optimizations we have proven such as several challenging ones from the recent superoptimization literature
- Compiling CompCert benchmark programs with our peephole optimizations: how many programs have peephole optimizations occur and a preliminary investigation of performance impact

5.4.1 Implementing Peek

The peephole framework that we added to CompCert comprises approximately 30,000 lines of Coq code and proof lines. Its main components are the lower level x86 semantics, liveness analysis, and the peephole execution engine and their respective proofs of correctness. We also provide a library to make proving peephole optimizations correct easier. The implementation sizes of these components are shown in Figure 5.3.

The trusted computing base (TCB) for Peek is similar to the TCB for CompCert. Just like CompCert, we trust the Coq proof checker, the Ocaml compiler and runtime, and that the C front-end models the semantics of C. Unlike CompCert, we trust the $\text{ASM}_{\mathbb{Z}32}$ backend instead of the $\text{ASM}_{\mathbb{Z}}$ backend to model x86 execution faithfully, but as explained earlier, we proved $\text{ASM}_{\mathbb{Z}32}$ and $\text{ASM}_{\mathbb{Z}}$ equivalent. Furthermore, $\text{ASM}_{\mathbb{Z}32}$ is closer to real program execution than $\text{ASM}_{\mathbb{Z}}$. We further trust that certain axioms are true, in particular that all generated code obeys the standard calling convention. Note that adding a peephole optimization incurs no increase in our trusted computing base.

	Spec	Proof	Total
Peek Total	**6,000**	**10,000**	**16,000**
Liveness	1,300	2,300	3,600
Peephole Exec	3,300	6,600	9,900
Libraries	1,100	900	2,000
Parameterization	40	163	203
ASM$_{\mathbb{Z}32}$	**3,300**	**5,500**	**8,800**
Peephole Lib	**2,000**	**3,100**	**5,100**
Total	**11,300**	**18,600**	**29,900**

Figure 5.3: Approx. lines of code according to `coqwc` tool.

5.4.2 Peephole Success

CompCert ships with a collection of small benchmarks, some of which are implementations of various hash functions in C. Some of these hash functions frequently left-rotate 64 bit values. For some of the left-rotate operations, CompCert currently generates 3 instructions including a left shift, a right shift, and an or. However, a single extended shift instruction is bitwise equivalent to these three. We implemented and verified two peepholes of this style, but with different orders of shifts. With these peephole transformations enabled, the `siphash24` benchmark in the CompCert benchmark suite sped up by 4%, and the `sha3` benchmark saw a speedup of almost 1%.

5.4.3 Peephole Variety

We proved a total of 28 peephole optimizations correct, using a range of different assembly computations. Some perform arithmetic operations (e.g., shift instead of multiply), others remove unnecessary jumps, eliminate redundant loads, or improve instruction-level parallelism. The appendix contains all the peephole transformations we have verified. Our purpose is to exercise a variety of assembly instructions to show that our framework is expressive enough to reason about semantic equivalence for useful transformations. Moreover, proof burden, while clearly non-trivial, is minor compared to proving the framework correct. The average proof size for a peephole optimization was 72 lines. Our approach for parameterization over registers does not increase this proof size.

Six of our verified peephole optimizations are all six of the assembly transformations presented as discovered by the stochastic superoptimizer in Bansal et al.'s recent work [6].[5] We took these rewrites as "challenge problems" that perform surprising transformations where equivalence is not obvious. Note the contribution of Bansal et al. is a system for automatically discovering them; we verified their correctness.

Qualitatively, the limiting factor in implementing more peephole optimizations, particularly those that would improve performance, is that the current $\mathrm{ASM}_{\mathbb{Z}}$ and $\mathrm{ASM}_{\mathbb{Z}32}$ semantics covers only the subset of x86 behavior needed by the existing code generator. It is conservative in two respects: (1) not including unused instructions or addressing modes and (2) often leaving values in flag registers undefined even when the instruction behavior for x86 specifies them. This conservatism is a common and wise engineering decision in formal verification — specify only the subset the system needs — but it directly limits the expressiveness of peephole optimizations we can verify. To date, we have been loathe to make additions to $\mathrm{ASM}_{\mathbb{Z}}$ (and correspondingly $\mathrm{ASM}_{\mathbb{Z}32}$), but future work can relax this approach with appropriate (unverifiable) care that our additions model x86 properly.

5.4.4 Benchmark Results

While our focus has been on verification and expressiveness, we also compared the output of CompCert with and without our optimizations on a standard set of small benchmarks shipped with CompCert. On 15 of the 23 benchmarks, none of our optimizations apply, so identical assembly is generated. For the remaining, see Figure 5.4 for the number of

[5] In one case, we changed the optimizations to use a `test` instruction rather than subtraction because CompCert's semantics for subtraction is conservative with respect to how flags are set.

Benchmark	# of Rewrites	Speedup
`binarytrees`	1	-
`chomp`	6	-
`fannkuch`	3	-
`fftw`	1	-
`knucleotide`	10	-
`sha1`	4	-
`sha3`	23	0.7%
`siphash24`	33	4.0%

Figure 5.4: Performance results of Peek.

optimizations that occur (static count), and the percent speedup.

Performance improvements so far are less encouraging. We see improvements of 4.0% and 0.7% on benchmarks `siphash24` and `sha3` respectively. For other benchmarks, run-time does not change to a statistically significant extent.

These benchmarks were run on an Intel i7-4790K at 4.00GHz with 16 gigabytes of RAM, running Ubuntu 15.04.

As discussed above, the limited expressiveness of $\text{ASM}_{\mathbb{Z}}$ holds back additional improvements. We have identified peephole optimizations that would speed up additional benchmarks, but verifying them would require extending $\text{ASM}_{\mathbb{Z}}$ to include new instructions and addressing modes.

5.4.5 Verified Benchmarks

One of the big advantages of low marginal cost verified optimization, is that it can be applied to speed up already verified code while retaining the guarantees already provided by this verified code. We took the C code that the VST [4] project verified was a correct implementation of SHA-256, and used it as an additional benchmark. We have developed verified peepholes which fire on this benchmark, and managed a 3.9% speedup (arithmetic average over 9 trials).

5.5 Related Work

Peek builds on previous research in peephole optimization, extensible compilers, and formal compiler verification.

Peephole Optimizers. Peephole optimizations are well-studied in the compiler literature [42], particularly in the context of superoptimization [39, 6, 56, 55, 51, 11]. However, to our knowledge, Peek is the first peephole optimization framework within a fully formally verified compiler. In particular, Peek shows how to extend CompCert's memory model to support verifying common peephole optimizations and provides a framework to formally prove peepholes with reasonable proof overhead.

Extensible Compilers. Frameworks like Gospel [64], Broadway [22], Cobalt [32], Rhodium [33], and PEC [30] all enable the programmer to express optimizations in a domain-specific language (DSL). Allowing programmers to develop optimizations in a DSL eases the effort to write an optimization and makes it easier for the framework to analyze and run the optimization. Of these frameworks, Cobalt, Rhodium, and PEC all exploit the restricted language of the DSL to automatically prove the correctness of optimizations. In these systems correctness is checked fully automatically using a solver like Z3, but both the reduction from the optimization correctness problem to Z3 queries and the execution engine are trusted to be correct without proof.

Verified Compilers. There is a long history on compiler verification, from early projects like Piton verified in ACL2 [43] to more recent work on project's like Chlipala's Lambda Tamer [13], Leroy's CompCert [34], and CompCertTSO [61]. These project all develop techniques to provide machine-checkable proofs of the compiler's correctness. Formalizing assembly languages represents some of the most relevant work from this area. Sewell et. al.'s work on formalizing x86 semantics [49] and Morrisett's x86 semantics for the NaCl SFI checker [44] serve as examples of realistic x86 formalizations. However, none of these systems provide support for extensible optimization passes; in particular, they do not support proving and adding peephole optimizations in the context of a realistic C compiler like CompCert.

The XCert tool was designed for automatically formally verifying optimizations checked by an SMT-based optimization proof tool. Unlike Peek, XCert worked in the compiler middle-end (RTL) which greatly eased implementation, reasoning, and verification of XCert. By operating at the RTL level, XCert can simply "link in" new code to avoid invalidating dataflow facts about other locations in the program. Peek has no such luxury at the assembly level where code is represented as a linear sequence of instructions in memory. More importantly, XCert is not fully formally verified. As a solver-aided tool, XCert has a much larger TCB than Peek because XCert assumes the correctness of an SMT solver like Z3. Thus XCert's TCB is bigger because it includes an SMT solver and XCert cannot verify assembly level transformations. Peek's primary goal is to make it easy to add new peephole optimizations to CompCert without increasing the compiler's trusted computing base. We believe that Peek's general approach could be applied in other verified compiler contexts without major changes.

Similar to CompCert, the Vellvm framework [66, 67] strives to provide a highly reliable suite of verified compilation tools. Alive [37] is a tool for verifying peephole optimizations in LLVM. Using Alive, a peephole writer can express and automatically prove their optimizations correct and extract the optimization to efficient C++ code which runs in the LLVM framework. The authors applied Alive to identify several bugs in LLVM's existing peephole optimizations. Unlike Alive, Peek requires peephole writers to manually verify their optimizations (though Peek provides extensive tactic support to make the proof burden reasonable). Peek is also proven correct in Coq and does not include the correctness of any constraint solvers in its TCB.

Chapter 6
CONCLUSION

Verified compilers are an important cornerstone on which we can build more reliable, sophisticated, and correct software. In this thesis I have demonstrated 2 projects where I pushed the limits of verified compilers.

Œuf: The Œuf project brings verified compiler support to a domain where it is particularly relevant, namely to a subset of the Gallina language, allowing for verified compilation of a language popular for creating verified systems. While the performance of code generated by Œuf is not competitive with existing compilers, it has reduced the TCB minimization problem to a compiler optimization problem. In the future, I hope to see a compiler with the trusted code base of Œuf, but with competitive performance for its generated code.

Peek: The Peek project provides architecture specific assembly optimizations to CompCert, and tackles key challenges with bridging the semantic gap between high level languages and computer hardware. While it is a step in the right direction, the remaining semantic gap between the assembly semantics for CompCert and the actual behavior of the microprocessor leaves much to be desired. I hope that we will one day see a processor and a compiler formally connected, with no semantic gap in between.

BIBLIOGRAPHY

[1] Amal Ahmed. Verified compilers for a multi-language world. In *1st Summit on Advances in Programming Languages, SNAPL 2015*, pages 15–31, 2015.

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

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

[4] Andrew Appel. Verification of a cryptographic primitive: SHA-256. *ACM Transactions on Programming Languages and Systems*, 37(2):7:1–7:31, April 2015.

[5] Andrew W. Appel. 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, 2011.

[6] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. ASPLOS '06, pages 394–403.

[7] Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 97–108, 2009.

[8] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. A concrete memory model for CompCert. ITP '15, pages 67–83.

[9] H Boehm and M Weiser. Garbage collection in an uncooperative environment. In *Software Practice and Experience*, 1988.

[10] Brakkton Booker. Delta air lines cancels nearly 700 additional flights. `https://n.pr/2aOV2mN`, 2016.

[11] Sebastian Buchwald. Optgen: A generator for local optimizations. In *Proceedings of the 24th International Conference on Compiler Construction*, pages 171–189, April 2015.

[12] Adam Chlipala. 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*, pages 54–65, 2007.

[13] Adam Chlipala. A verified compiler for an impure functional language. *SIGPLAN Not.*, 45(1):93–106, January 2010.

[14] Adam Chlipala. *Certified Programming with Dependent Types.* MIT Press, December 2013.

[15] Guillaume Claret. Coq.io. `http://coq.io/`, 2016.

[16] David Costanzo, Zhong Shao, and Ronghui Gu. 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, pages 648–664, 2016.

[17] Maulik A. Dave. Compiler verification: A bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.

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

[19] Paul Govereau. *Denotational Translation Validation.* PhD thesis, Harvard University, 2012.

[20] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 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*, pages 595–608, 2015.

[21] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 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.*, pages 653–669, 2016.

[22] Samuel Z. Guyer and Calvin Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93(2):342–357, February 2005.

[23] Rebecca Hersher. Amazon and the $150 million typo. `https://n.pr/2lHshek`, 2017.

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

[25] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A formal C memory model supporting integer-pointer casts. PLDI '15, pages 326–335.

[26] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. In *Proceedings of the 43rd Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16, pages 178–190, 2016.

[27] 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. sel4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09. ACM, 2009.

[28] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. Software verification with ITPs should use binary code extraction to reduce the TCB (short paper). In *ITP '18*.

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

[30] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. PLDI '09.

[31] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14. ACM, 2014.

[32] Sorin Lerner, Todd Millstein, and Craig Chambers. Cobalt: A language for writing provably-sound compiler optimizations. *Electronic Notes in Theoretical Compututer Science*, 132:5–17, 2005.

[33] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. POPL '05, pages 364–377.

[34] Xavier Leroy. 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, 2006.

[35] Pierre Letouzey. Extraction in Coq: An overview. In *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369, 2008.

[36] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[37] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with Alive. PLDI '15, pages 22–32.

[38] Gregory Malecha. Template Coq plugin. https://github.com/gmalecha/template-coq, 2015.

[39] Henry Massalin. Superoptimizer: A look at the smallest program. ASPLOS '87, pages 122–126.

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

[41] Andrew McCreight, Tim Chevalier, and Andrew Tolmach. A certified framework for compiling and executing garbage-collected languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 273–284, 2010.

[42] W. M. McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, July 1965.

[43] J. Strother Moore. A mechanically verified language implementation. *J. Autom. Reasoning*, 5(4):461–492, 1989.

[44] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. Rocksalt: Better, faster, stronger SFI for the x86. PLDI '12, pages 395–404.

[45] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. Œuf: Minimizing the Coq extraction TCB. In *CPP '18*, pages 172–185.

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

[47] Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ml from higher-order logic. ICFP '12, 2012.

[48] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 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*, pages 166–178, 2015.

[49] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: X86-TSO. TPHOLs '09, pages 391–407.

[50] 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*, pages 128–148, 2014.

[51] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. PLDI '14, pages 396–407.

[52] Clément Pit-Claudel. Compilation using correct-by-construction program synthesis. Master's thesis, Massachusetts Institute of Technology, August 2016.

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

[54] Hanan Samet. Proving the correctness of heuristically optimized code. *Commun. ACM*, 1978.

[55] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. PLDI '14, pages 53–64.

[56] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. ASPLOS '13, pages 305–316.

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

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

[59] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for CakeML. ICFP '16, 2016.

[60] Ken Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8), August 1984.

[61] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3), June 2013.

[62] Peng Wang, Santiago Cuellar, and Adam Chlipala. 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, pages 675–690, 2014.

[63] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14. USENIX Association, 2014.

[64] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, November 1997.

[65] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11. ACM, 2011.

[66] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. PLDI '13, pages 175–186.

[67] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. POPL '12, pages 427–440.