Oeuf

Eric Mullen, Stuart Pernsteiner, James Wilcox

The ability to build verified systems has brought unprecedented reliability to C compilers. CompCert, despite extensive testing by the Csmith and EMI projects, has yet to have a bug found in its verified portions. We can develop extremely reliable software by writing it in Coq, verifying it meets a spec, and running the extracted code. However, Coq's extraction system is not verified to produce semantically equivalent code. (Indeed, it is common to use custom extraction rules that can implement essentially arbitrary syntactic transformations on the code.) Furthermore, the extracted code must then be compiled using the target language's (unverified) compiler and run using its (unverified) run-time system. In some environments (eg, an OS kernel) this run-time requirement is not acceptable. For instance, the JitK project was forced to run in userspace because it required the OCaml runtime to execute CompCert.

We present Oeuf, a verified compiler from a subset of Gallina to x86. Oeuf eliminates both ad hoc extraction rules and unverified runtime systems from the TCB. We call the subset of Gallina supported by Oeuf "OeufML", since it supports roughly the features of a pure functional ML-like language. OeufML is carefully designed so that it can be deeply represented within Coq itself. The most striking aspect of this representation is a denotational semantics, *implemented as a total function in Coq,* which maps an OeufML term to the Gallina term it represents. Oeuf compiles OeufML to Cminor, a mid-level imperative intermediate language used in CompCert, then invokes CompCert itself to complete the compilation to x86.

The workflow for using Oeuf is shown in Figure 1. (1) The programmer implements their system as a Gallina program. (2) The programmer verifies that the system has the desired specification using standard Coq techniques. (3) If the system uses any features of Coq that are not supported by OeufML (e.g., dependent types, and, for the time being, parametric polymorphism), then the programmer manually reimplements the system to avoid those features, and proves this low-level implementation equivalent to the high-level original system. (4) The programmer *reflects* the low-level system using Oeuf's provided tactics to automatically obtain a deeply embedded representation of the low-level system as an OeufML expression. (5) The programmer runs a single `reflexivity` tactic to check that the denotation of the reflected low-level system is equal to the Gallina term representing the low-level system. (6) The programmer runs the Oeuf compiler on the reflected term, resulting in an x86 program.[1]

The end-to-end specification of Oeuf is that the resulting x86 program is equivalent to the original, high-level Gallina implementation of the system. More precisely, any possible behavior

[1] The current process to run the Oeuf compiler is less than ideal. The reflected term and Oeuf compiler (including all of CompCert) are extracted and compiled by OCaml to produce an executable that writes out the generated code. We hope to improve this in future work.

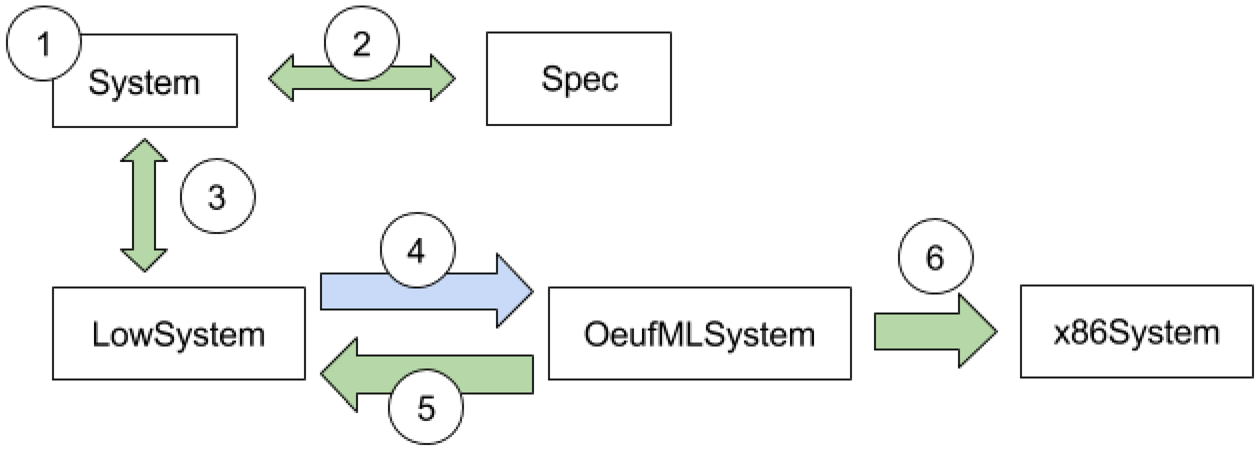

Figure 1: Workflow for using Oeuf. (1) Write a system in Gallina. (2) Verify the system satisfies its spec. (3) Manually translate away features not supported by OeufML and prove equivalence. (4) Reflect to OeufML system using provided tactics. (5) Use automatic check to ensure reflection denotes correctly. (6) Compile OeufML system with Oeuf. The resulting x86 system is equivalent to the original Gallina system.

of the x86 program under CompCert's operational semantics for x86 is also a possible behavior of the denotational semantics of the input OeufML expression. Since step (5) above guarantees that this expression is equivalent to the low-level Gallina system, and step (3) guarantees that the low-level system is equivalent to the high-level system, it follows that the x86 program faithfully implements the original Gallina program.

OeufML values and in-memory representations at the x86 level are very different beasts, so Oeuf's specification defines a (trusted) correspondence between the two. Part of this correspondence includes a specification of the assembly-level ABI for interacting with code generated by Oeuf. For example, if the programmer compiles the addition function on nats, the ABI specifies how to call the resulting closure and in what format it expects its arguments. The ABI for higher-order functions is more complex. For example, compiling the map function over lists of nats should allow an assembly-level caller to manually construct a closure, whose code is potentially not expressible in OeufML. Our work on formalizing this specification is ongoing.

The syntax of OeufML is described by the following grammar.

$$e \ ::= \ x \mid e\, e \mid \lambda x.e \mid C\, e^* \mid E\, e^*\, e$$

OeufML is a simply typed lambda calculus parametrized by a set of inductive base types. Variables, application, and lambda abstraction are standard. $C\, e^*$ represents a constructor applied to some number of arguments. $E\, e^*\, e$ represents an eliminator applied to some number of cases and a target. For simplicity, constructors and eliminators are syntactically restricted to be fully applied. For example, if S is the successor constructor for the natural numbers, then S itself is not a well-formed expression, but $S\, O$ is (where O is the zero constructor for the natural numbers). The successor *function* on natural numbers can be expressed by wrapping S with a lambda: $\lambda x.\, S\, x$.

The types for OeufML are just functions over the inductive base types. The typing judgement for OeufML is standard, except that it is parametrized over the base types, including a description of the number and types of arguments to each constructor and eliminator. The operational semantics are also standard, except that it has rule for eliminating a constructor which is parametric in the inductive type involved.

The Coq representation of the OeufML is designed primarily to enable the denotational semantics. It makes heavy use of dependent types to represent the type structure of the object program so that the type of the denotation function can be expressed in Coq’s type system. We have shown that this denotation function is preserved by the operational semantics.

Oeuf compiles OeufML to Cminor by way of 8 additional intermediate languages; see Figure 2. At a high-level, Oeuf must translate away all features of OeufML not supported by Cminor. In particular, lexically scoped anonymous functions are translated to heap-allocated closures over global function definitions and their free variables, constructors of inductive types are represented by tagged unions of pointer data structures, and eliminators are implemented by switching on the tag and recursing where necessary.

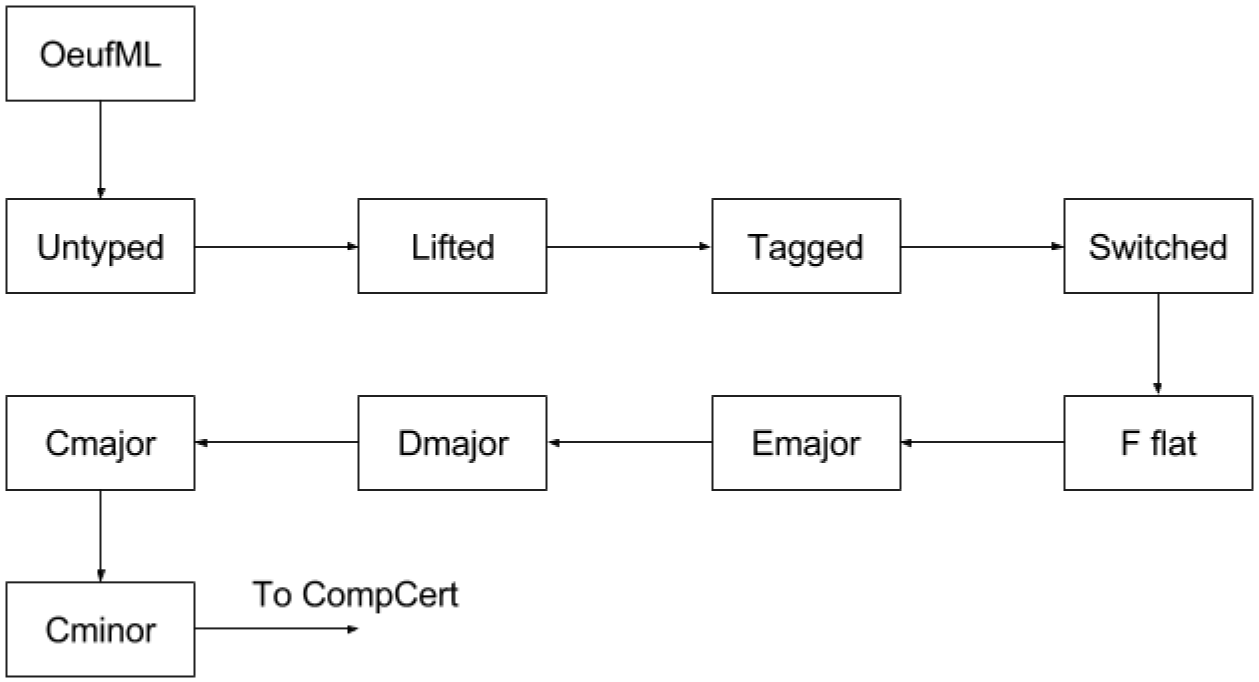

Figure 2: Intermediate languages of Oeuf.

The type information littered throughout OeufML makes it relatively inconvenient to work with, so the first pass discards all typing information in the object program, resulting in an otherwise-equivalent expression in the Untyped intermediate representation. Next, anonymous functions are lifted to global function definitions that explicitly close over their free variables, resulting in a Lifted program consisting of an environment of global function definitions and a main expression. Then, inductive types are converted to a tagged representation that uses a natural number to differentiate constructors instead of explicitly enumerating the possible constructors for each type. The result is a program in the Tagged representation. Next, eliminators are implemented by additional global functions using switch and recursion, resulting in a Switched program. Then, the program is transformed from a functional style to a Flattened imperative style, converting each complex expression to a sequence of statements. Once the program is flattened, the expressions are simplified even more until they are only variables and dereferences, giving us Emajor. The translation from Emajor to Dmajor is a large whole step: the semantics of Dmajor are written in terms of memory and pointers, whereas Emajor is written in terms of closures and constructors, and with no explicit memory. Here we implement the correspondence between high level values and low level values, for which the specification is trusted. From Dmajor to Cmajor is a relatively small pass, which only lowers a special `malloc` instruction down to an actual function call. Finally, translation from Cmajor to Cminor is trivial, as Cmajor is a strict subset of the CompCert Cminor language.

We have tested Oeuf on several simple examples involving basic arithmetic on nats from the standard library. Additionally, we have proved the first compiler pass correct. As future work, we will finish the verification and use Oeuf to compile real systems. We also plan to add support for builtin data types with custom representations, so that, for example, bitvectors and CompCert-style integers can be represented at the assembly level as machine words rather than heap-allocated values of inductive types.

One of the main challenges is dealing with dynamic memory allocation, which we solve by calling `malloc` as needed and then never deallocating any memory. This approach should be sufficient for batch-mode programs (such as CompCert), since all remaining memory will be freed by the operating system when the program exits. As future work, we may implement a verified garbage collector for use in long-running programs.

The benefits of a Gallina frontend to CompCert are many: we will be able to self-host CompCert, using CompCert to compile itself, and ideally cut the Ocaml runtime out of the TCB for CompCert. Further, we would gain the ability to write and verify programs in Coq, and compile and run them using the CompCert framework. This would allow for functions written in Gallina to be called from C code, and could allow for a better architecture for verified systems: Gallina for the purely functional code, combined with C (verified using VST) for any effectful computation and to call the Gallina functions. The ability to use Coq as a program logic instead of VST will allow us to verify much larger programs than would be feasible with VST.