

# CertiCoq-Wasm: Verified compilation from Coq to WebAssembly

Wolfgang Meier  
womeier@posteo.de  
Aarhus University  
Aarhus, Denmark

Jean Pichon-Pharabod  
jean.pichon@cs.au.dk  
Aarhus University  
Aarhus, Denmark

Bas Spitters  
spitters@cs.au.dk  
Aarhus University  
Aarhus, Denmark

## Abstract

We describe CertiCoq-Wasm, a verified compiler from the Gallina programming language of the Coq theorem prover to WebAssembly. CertiCoq-Wasm is mechanised with respect to the WasmCert-Coq formalisation of the WebAssembly 1.0 standard, and produces WebAssembly programs with reasonable performance. Internally, CertiCoq-Wasm is based on the CertiCoq pipeline, and diverges only at the lowering from its minimal lambda calculus in administrative normal form.

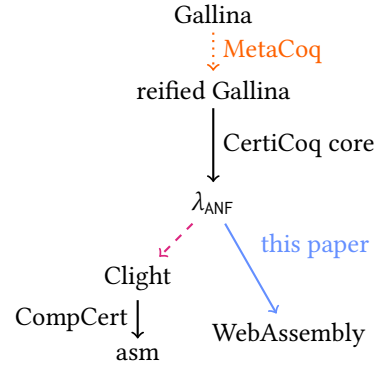
## 1 Introduction

Interactive theorem provers like Coq make it possible to develop a program and prove its properties in the same environment. In many cases, the program is also of interest outside of the theorem prover; therefore, the theorem prover makes it possible to *extract* such an internal program to an external file, possibly written in another language. However, this raises the question of how the extracted program relates to the internal program, especially if extraction involves non-trivial compilation. Coq provides both unverified extraction [5] and a recent certified extraction, CertiCoq [1]. CertiCoq targets Clight and depends on CompCert [4] to produce verified machine code. For applications such as the web and blockchains (‘web3’), WebAssembly (abbreviated Wasm) has emerged as the standard assembly language.

**Contribution.** We thus contribute CertiCoq-Wasm, an extraction mechanism from Gallina, the programming language of Coq, to WebAssembly. We implement CertiCoq-Wasm by replacing the back-end of CertiCoq [1] to produce WebAssembly from its  $\lambda_{\text{ANF}}$  language [6]. This greatly decreases the trusted code base over, for example, unverified compilation of Clight to WebAssembly. We prove CertiCoq-Wasm correct with respect to the official specification of WebAssembly 1.0 [7], as mechanised in WasmCert-Coq [10].

### 1.1 CertiCoq

CertiCoq is a compiler from Gallina, the programming language of the Coq proof assistant [3], to Clight [2], the dialect of the C programming language that CompCert [4] compiles.



**Figure 1.** The CertiCoq pipeline. Proofs in progress are in dashed magenta, and MetaCoq [9] (which has to be trusted) is in dotted orange. Our verified contribution is in blue. The *CertiCoq core* stage includes optimisations on  $\lambda_{\text{ANF}}$ .

(Variables)	$x, y, f \in \text{Var}$
(Constructors)	$C \in \text{Constr}$
(Function defs)	$fd ::= (f(\bar{y}) = e)$
(Expressions)	$e ::= \text{let } x = C(\bar{y}) \text{ in } e$ $\quad   \text{let } x = y.i \text{ in } e$ $\quad   \text{case } y \text{ of } [C_i \rightarrow e_i]_{i \in I}$ $\quad   \text{let } \bar{fd} \text{ in } e$ $\quad   \text{let } x = f \bar{y} \text{ in } e$ $\quad   f \bar{y}$ $\quad   \text{ret}(y)$
(Values)	$v ::= (C, \bar{v}) \mid (\rho, \bar{fd}, x)$
(Environments)	$\rho ::= \cdot \mid \rho, x \mapsto v$

**Figure 2.** Syntax of CertiCoq’s  $\lambda_{\text{ANF}}$  intermediate language, primitive operations omitted

It works as an alternative to the usual ‘extraction’ mechanism from Gallina to OCaml or Haskell.

As often in compilation of a functional language to an imperative one, CertiCoq’s pipeline involves a low intermediate language, which is where CertiCoq-Wasm inserts itself.

```

t ::= i32 | i64 | ...
instr ::= t.const | t.add | ... | t.xor | ... |
        t.load memarg | t.store memarg |
        memory.grow
        nop | if bty instrs1 else instrs2 end |
        call funcidx | call_indirect tableidx funcidx

```

**Figure 3.** Syntax of (parts of) WebAssembly

## 1.2 WebAssembly

CertiCoq-Wasm is mechanised with respect to the WasmCert-Coq formalisation of the WebAssembly 1.0 standard. WebAssembly is a simple but detailed stack language (Figure 3). The main feature we rely on is that each WebAssembly module is equipped with a *linear memory*, a growable array of bytes, accessed with loads and stores that take integers indices (as opposed to the complex pointers of C). We also take advantage of `call_indirect`, which takes a function index into a table of functions.

## 2 CertiCoq-Wasm

### 2.1 Correctness

**Theorem 2.1** (Correctness of lowering). *For any closed well-formed  $\lambda_{\text{ANF}}$  program  $e$  with globally unique bound variables,*

$$\left( \begin{array}{l} \cdot \vdash e \Downarrow v \wedge \text{compile } e = (\text{mod}, \dots) \wedge \\ \text{instantiate mod} = (sr, \dots) \end{array} \right) \Longrightarrow \exists sr'. \left( v \simeq_{sr'}^{\text{val}} sr'.\text{globals}_{\text{res}} \vee sr'.\text{globals}_{\text{out\_of\_mem}} = 1 \right)$$

where ‘*compile*’ is our extraction to Wasm, which generates a Wasm module *mod*, which is then instantiated (see §4), inducing a ‘*store*’ *sr*.

*Proof.* The proof is roughly as per [8, §4.3], and relates  $\lambda_{\text{ANF}}$  values to Wasm values via the relation in Figure 4. The simple semantics of Wasm makes the proof manageable.  $\square$

This only concerns compilation from CertiCoq-Wasm’s  $\lambda_{\text{ANF}}$  to WebAssembly. However, we can combine it with CertiCoq’s internal correctness to get a Gallina-to-Wasm result.

## 3 TCB

Because CertiCoq-Wasm builds on the CertiCoq front-end and middle-end, it inherits all of CertiCoq’s “upper” assumptions, in particular MetaCoq’s correctness.

We verify CertiCoq-Wasm down to the WebAssembly AST of WasmCert. The conversion of the AST to the `.wasm` binary format is part of WasmCert but not yet verified.

Our diff to CertiCoq is some OCaml in a single file to insert our backend; the rest of CertiCoq-Wasm is Gallina code.

$$\text{(VR\_FUN)} \quad \frac{\begin{array}{l} (f(\bar{y}) = e) = \overline{fd}_{idx-4} \\ sr.\text{funcs}_{idx} = F \quad F.\text{type} = (i32^{|\bar{y}|} \rightarrow []) \\ F.\text{body} = \text{codegen } e \quad F.\text{locals} = i32^{|\text{bound\_vars}(e)|} \end{array}}{(\rho, \overline{fd}, f) \simeq_{sr}^{\text{val}} idx}$$

$$\text{(VR\_CONSTR)} \quad \frac{\begin{array}{l} sr.\text{mems}_0 = m \quad ptr + 4(|\bar{v}| + 1) \leq sr.\text{globals}_{gmp} \\ m[ptr, ptr + 4] = C \quad \forall v_i \in \bar{v}. v_i \simeq_{sr}^{\text{val}} m \left[ \begin{array}{l} ptr + 4(i + 1), \\ ptr + 4(i + 2) \end{array} \right] \end{array}}{(C, \bar{v}) \simeq_{sr}^{\text{val}} ptr}$$

**Figure 4.** Value relation, relating a  $\lambda_{\text{ANF}}$  value to a Wasm i32

## 4 Limitations

CertiCoq-Wasm has a full proof of correctness, but is still work-in-progress. Some of the limitations are:

- CertiCoq implicitly utilises the correctness of CompCert to extract assembly (Figure 1), yielding a Gallina-to-assembly pipeline. We are not aware of any verified WebAssembly compilers, so our result stops at WebAssembly, and is thus weaker than that of CertiCoq.
- Some Gallina programs are ‘too big’ to be directly represented in WebAssembly, for example, if they have more than  $2^{32}$  constructors.
- Unavoidably, some compiled Gallina programs consume too much memory, resulting in the weak clause of the correctness statement. However, our compiler does not take steps to circumvent that: it grows the memory as needed to allocate new objects, but never garbage collects old objects.
- Our backend only supports  $\lambda_{\text{ANF}}$  tailcalls, non-tailcalls are not supported. The absence of non-tailcalls can be enforced with CertiCoq’s `-cps` flag.
- Even though all major WebAssembly runtimes support them, tailcalls are not part of the 1.0 spec of WebAssembly. We generate standard calls, and a script replaces them with tailcalls in the binary.
- WebAssembly is run in two phases: (1) it is first *instantiated*, which involves type-checking and sets up the WebAssembly ‘store’, a sort of evaluation context; (2) it is then run proper from its start function. Our theorems assume that the module produced by our compiler gets successfully instantiated.
- CertiCoq can be made to use ‘native types’ like `i32` that it then generates efficient representations and code for. Our compiler does not support this yet.
- CertiCoq-Wasm only supports the compilation of closed programs, it is currently not straightforward to link the generated Wasm module with other Wasm code.

## Acknowledgments

We thank Zoe Paraskevopoulou for suggesting the project and providing advice. This work is supported by an Aarhus University DIGIT small collaboration grant.

## References

- [1] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, , and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *CoqPL'17: The Third International Workshop on Coq for Programming Languages*. <http://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf>
- [2] Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *J. Autom. Reason.* 43, 3 (2009), 263–288. <https://doi.org/10.1007/s10817-009-9148-3>
- [3] Coq development team. 2023. The Gallina specification language. <https://coq.inria.fr/doc/V8.18.0/refman/language/gallina-specification-language.html>
- [4] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. <https://doi.org/10.1145/1538788.1538814>
- [5] Pierre Letouzey. 2002. A New Extraction for Coq.. In *TYPES (Lecture Notes in Computer Science, Vol. 2646)*. Springer, 200–219.
- [6] Zoe Paraskevopoulou, John M. Li, and Andrew W. Appel. 2021. Compositional optimizations for CertiCoq. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. <https://doi.org/10.1145/3473591>
- [7] Andreas Rossberg. 2019. *WebAssembly Core Specification W3C Recommendation*. Technical Report. W3C. <https://www.w3.org/TR/wasm-core-1/>
- [8] Olivier Savary Bélanger. 2019. *Verified Extraction for Coq*. Ph.D. Dissertation. Princeton University. <https://dataspace.princeton.edu/handle/88435/dsp01zw12z817f>
- [9] Matthieu Sozeau, Abhishek Anand, Simon Boulter, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *J. Autom. Reason.* 64, 5 (2020), 947–999. <https://doi.org/10.1007/s10817-019-09540-0>
- [10] Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. 2021. Two Mechanisations of WebAssembly 1.0. In *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13047)*, Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan (Eds.). Springer, 61–79. [https://doi.org/10.1007/978-3-030-90870-6\\_4](https://doi.org/10.1007/978-3-030-90870-6_4)