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 λ_{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].

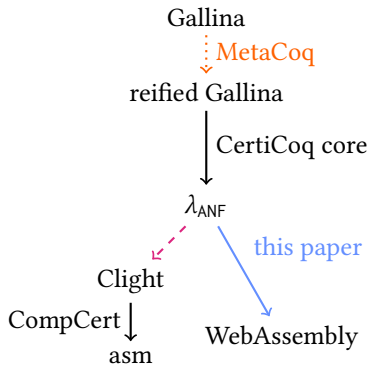

Figure 1. The CertiCoq pipeline. Proofs in progress are in dashed magenta, and MetaCoq [9] (which has to be trusted) is in dotted orange. Our verified contribution is in blue.

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. 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.

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.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$
\begin{array}{lll}
\text{(Variables)} & x, y, f \in \texttt{Var} & \\
\text{(Constructors)} & \texttt{C} & \in \texttt{Constr} \\
\text{(Function defs)} & fd & ::= (f(\bar{y}) = e) \\
\text{(Expressions)} & e & ::= \text{let } x = \texttt{C}(\bar{y}) \text{ in } e \\
 & & \mid \text{let } x = y.i \text{ in } e \\
 & & \mid \text{case } y \text{ of } [C_i \to e_i]_{i \in I} \\
 & & \mid \text{let } \overline{fd} \text{ in } e \\
 & & \mid \text{let } x = f\ \bar{y} \text{ in } e \\
 & & \mid f\ \bar{y} \\
 & & \mid \text{ret}(y) \\
\text{(Values)} & v & ::= (C, \bar{v}) \mid \left(\rho, \overline{fd}, x\right) \\
\text{(Environments)} & \rho & ::= \cdot \mid \rho, x \mapsto v
\end{array}
$$

Figure 2. Syntax of CertiCoq's λ_{ANF} intermediate language, primitive operations omitted

$$
\begin{array}{rl}
t ::= & \text{i32} \mid \text{i64} \mid \ldots \\
instr ::= & t.\text{const} \mid t.\text{add} \mid \ldots \mid t.\text{xor} \mid \ldots \mid \\
 & t.\text{load } memarg \mid t.\text{store } memarg \mid \\
 & \text{memory.grow} \\
 & \text{nop} \mid \text{if } bty\ instrs_1 \text{ else } instrs_2 \text{ end} \mid \\
 & \text{call } funcidx \mid \text{call\_indirect } tableidx\ funcidx
\end{array}
$$

Figure 3. Syntax of (parts of) WebAssembly

2 CertiCoq-Wasm

2.1 Correctness

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

$$
\begin{array}{l}
\left(\begin{array}{l} \cdot \vdash e \Downarrow v \wedge \mathit{compile}\ e = (\mathit{mod}, \ldots) \wedge \\ \mathit{instantiate}\ \mathit{mod} = (sr, \ldots) \end{array}\right) \Longrightarrow \\
\quad (sr, \ldots, [\text{call } idx_{\text{main}}]) \to^* (sr', \ldots, []) \wedge \\
\exists sr'. \left(v \simeq^{\text{val}}_{sr'} sr'.\text{globals}_{\text{res}} \vee sr'.\text{globals}_{\text{out\_of\_mem}} = 1\right)
\end{array}
$$

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 λ_{ANF} values to Wasm values via the relation in Figure 4. The simple semantics of Wasm makes the proof manageable. □

This only concerns compilation from CertiCoq-Wasm's λ_{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.

(VR_FUN)

$$
\frac{\begin{array}{c}(f(\bar{y}) = e) = \overline{fd}_{idx-4} \\ sr.\text{funcs}_{idx} = F \qquad F.\text{type} = (\text{i32}^{|\bar{y}|} \to []) \\ F.\text{body} = \text{codegen } e \qquad F.\text{locals} = \text{i32}^{|\text{bound\_vars}(e)|}\end{array}}{\left(\rho, \overline{fd}, f\right) \simeq^{\text{val}}_{sr} idx}
$$

(VR_CONSTR)

$$
\frac{\begin{array}{cc} sr.\text{mems}_0 = m & ptr + 4(|\bar{v}| + 1) \le sr.\text{globals}_{gmp} \\ m[ptr, ptr + 4] = C & \forall v_i \in \bar{v}.\, v_i \simeq^{\text{val}}_{sr} m \begin{bmatrix} ptr + 4(i+1), \\ ptr + 4(i+2) \end{bmatrix}\end{array}}{(C, \bar{v}) \simeq^{\text{val}}_{sr} ptr}
$$

Figure 4. Value relation, relating a λ_{ANF} value to a Wasm i32

We verify CertiCoq-Wasm down to the WebAssembly AST of WasmCert. The AST-to-string conversion to the `.wat` format (S-expressions) of WasmCert is not verified, but straightforward. WasmCert does not capture the binary `.wasm` format (which requires some light compilation, as implemented by `wat2wasm`).

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

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 λ_{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 can then generate efficient representations and code for. Our compiler does not support this yet.

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 Boulier, 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