CertiCoq-Wasm: A Verified WebAssembly Backend for CertiCoq

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

Martin Jensen
mkarup@post.au.dk
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 contribute CertiCoq-Wasm, a verified WebAssembly backend for CertiCoq. CertiCoq-Wasm is implemented and verified in the Coq proof assistant, and is mechanised with respect to the WasmCert-Coq formalisation of the WebAssembly standard. CertiCoq-Wasm works from CertiCoq's minimal lambda calculus in administrative normal form (ANF), and produces WebAssembly programs with reasonable performance. It implements Coq's primitive integer operations as efficient WebAssembly instructions, identifying a corner case in their implementation that led to unsoundness. We compare CertiCoq-Wasm against other, partially verified extraction mechanisms from Coq to WebAssembly, benchmarking running time and program size. We demonstrate the practical usability of CertiCoq-Wasm with two case studies: we extract and run a Gallina program on the web, and a ConCert smart contract on the Concordium blockchain.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Formal software verification**.

Keywords: Compilers, Formal verification, Coq, WebAssembly

ACM Reference Format:
Wolfgang Meier, Martin Jensen, Jean Pichon-Pharabod, and Bas Spitters. 2025. CertiCoq-Wasm: A Verified WebAssembly Backend for CertiCoq. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '25), January 20–21, 2025, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3703595.3705879

An early version of this work was presented at CoqPL'24 [29].

CPP '25, January 20–21, 2025, Denver, CO, USA

ACM ISBN 979-8-4007-1347-7/25/01
https://doi.org/10.1145/3703595.3705879

1 Introduction

Interactive theorem provers like Coq [46], Isabelle [33], Lean [14], Minlog [7], and NuPRL [13] 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. For that reason, these theorem provers make it possible to *extract* such an internal program to an external program, often written in another, more mainstream language. While this is useful, it raises the question of how the extracted program relates to the internal program, especially if extraction involves non-trivial compilation. Coq has, almost since its inception, provided extraction to OCaml by exploiting the closeness between the source and target languages [36–38]. Despite the similarity between the languages, verifying this extraction has been a long-standing challenge [19, 20, 26, 27], and has only recently been satisfactorily met [18]. To widen the applicability of extraction by targeting a more mainstream language, a more recent project, CertiCoq [1], implements an extraction to C; it produces machine code when combined with the verified CompCert compiler [25]. While C underpins much of our computing infrastructure, WebAssembly [21, 39] (abbreviated Wasm) has emerged as the standard language to deploy client-side apps on the web. WebAssembly is also increasingly used on the server-side, for example in edge computing and on the Internet of Things. Moreover, its strict execution model also makes it suitable to run smart contracts on the blockchain [12]. With the promise of near-native performance [30], this makes WebAssembly an attractive target for compilation. Together with a clearly defined semantics, this makes it an attractive target for verified compilation.

Contribution. We thus contribute CertiCoq-Wasm, a verified WebAssembly backend for CertiCoq. CertiCoq-Wasm works as an alternative to the original Clight backend of CertiCoq, producing instead WebAssembly from CertiCoq's λ_{ANF} intermediate language [35, §3.1], as illustrated in Figure 1. 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 specification of

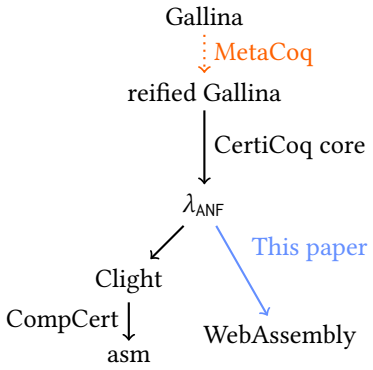

Figure 1. The CertiCoq pipeline. MetaCoq [42], which has to be trusted, is in dotted orange. Our verified contribution is in blue. The *CertiCoq core* stage includes optimisations on λ_{ANF}.

WebAssembly 2.0 with the tail call extension [40], as mechanised in WasmCert-Coq [49]. For portability, CertiCoq-Wasm can be made to generate Wasm binaries that only use the subset of Wasm defined in the WebAssembly 1.0 standard [39]. For performance comparison, we also develop an (unverified) extension of CertiCoq-Wasm which uses the (work-in-progress) garbage collection extension of WebAssembly. For performance, CertiCoq-Wasm supports translating Coq's 63-bit 'primitive' integers and the operations on these to efficient Wasm instructions, making it Coq's first verified extraction mechanism to support them. During the verification of the primitive integer operations, we identified a corner case which led to an unsoundness in Coq's *vm_compute*[1], and to bugs in CertiCoq's (not yet verified) implementation of primitive operations, as well as an incorrect modelling of shifts in WasmCert[2]. We compare CertiCoq-Wasm against other extraction mechanisms from Coq to WebAssembly, benchmarking running time and program size. We demonstrate the practical usability of CertiCoq-Wasm with two case studies: we extract and run a Gallina program on the web, and a ConCert smart contract on the Concordium blockchain.

1.1 CertiCoq

CertiCoq is a compiler from Gallina, the programming language of the Coq proof assistant [15], to Clight [8], the dialect of the C programming language that CompCert [25] compiles. It works as an alternative to the usual extraction mechanism from Gallina to OCaml, Haskell, or Scheme. Several stages of CertiCoq are verified, including the frontend, optimisations, and there is an ongoing effort to verify the C backend. However, CertiCoq does not yet have an end-to-end correctness statement [24, §13].

As often in compilation of a functional language to an imperative one, CertiCoq's pipeline involves a low intermediate language, λ_{ANF}, which is where CertiCoq-Wasm inserts itself. λ_{ANF} is an untyped lambda-calculus in A-normal form [34, §3]. In addition to common constructs, it features 'primitive' 63-bit integer values and corresponding operations, mirroring those available in Coq [46, §2.1.13]. We give the syntax of λ_{ANF} in Figure 2, and its operational semantics in Figure 3.

$$
\begin{array}{lllll}
\textit{(Variable)} & x, y, f & \in & \texttt{Var} & \\
\textit{(Constructor)} & C & \in & \texttt{Constr} & \\
\textit{(Primitive val)} & p & \in & \texttt{PrimVal} & \\
\textit{(Primitive op)} & op & \in & \texttt{PrimOp} & \\
\textit{(Function def)} & fd & ::= & (f(\overline{y}) = e) & \\
\\
\textit{(Expression)} & e & ::= & \text{let } x = C(\overline{y}) \text{ in } e & \\
 & & \mid & \text{let } x = y.i \text{ in } e & \\
 & & \mid & \text{case } y \text{ of } [C_i \to e_i]_{i \in I} & \\
 & & \mid & \text{let } x = f\ \overline{y} \text{ in } e & \\
 & & \mid & f\ \overline{y} & \\
 & & \mid & \text{let } x = p \text{ in } e & \\
 & & \mid & \text{let } x = op\ \overline{y} \text{ in } e & \\
 & & \mid & \text{let } \overline{fd} \text{ in } e & \\
 & & \mid & \text{halt } y & \\
\\
\textit{(Value)} & v & ::= & (C, \overline{v}) \mid \left(\rho, \overline{fd}, f\right) \mid p & \\
\textit{(Environment)} & \rho & ::= & \cdot \mid \rho; x \mapsto v &
\end{array}
$$

Figure 2. Syntax of CertiCoq's λ_{ANF} intermediate language.

We require that λ_{ANF} expressions satisfy the following restrictions:

1. Both locally bound variables and function variables are internally represented by *globally unique* integers, and a mapping provides meaningful names.
2. Function definitions are only allowed at the top-level.
3. The number of constructors, function definitions, etc., are representable as Wasm i32s.

The first two restrictions are ensured by an earlier pipeline stage. CertiCoq-Wasm checks the third restriction, and refuses to compile a λ_{ANF} expression if it is not satisfied. Similar restrictions have to hold for the original C backend.

1.2 WebAssembly

CertiCoq-Wasm is mechanised with respect to the WasmCert-Coq formalisation of the WebAssembly standard. Wasm is a simple but detailed stack language, whose syntax we sketch in Figure 4. Each WebAssembly module can be equipped with a *linear memory*, a growable array of bytes, accessed with load and store instructions that take integer indices as addresses (as opposed to the complex pointers of C). We use this memory to lay the tree of constructors out. We also take advantage of the indirect functional call instruction, call_indirect, which takes as argument a function *index* into a table of functions. One unusual feature of WebAssembly is

[1] https://github.com/coq/coq/issues/19402
[2] https://github.com/WasmCert/WasmCert-Coq/issues/44

$$\frac{\rho(\overline{y}) = \overline{w} \qquad \rho; x \mapsto (C, \overline{w}) \vdash e \Downarrow_k v}{\rho \vdash \mathsf{let}\ x = C(\overline{y})\ \mathsf{in}\ e \ \Downarrow_k v}\ (\text{E\_CONSTR})$$

$$\frac{\rho(y) = (C, \overline{w}) \qquad \rho; x \mapsto w_i \vdash e \Downarrow_k v}{\rho \vdash \mathsf{let}\ x = y.i\ \mathsf{in}\ e \ \Downarrow_k v}\ (\text{E\_PROJ})$$

$$\frac{\rho(y) = (C, \overline{w}) \qquad (C \Rightarrow e) \in \overline{b} \qquad \rho \vdash e \Downarrow_k v}{\rho \vdash \mathsf{case}\ y\ \mathsf{of}\ \overline{b} \ \Downarrow_k v}\ (\text{E\_CASE})$$

$$\frac{\begin{array}{c}\rho(f) = (\rho', \overline{fd}, f) \\ (f(\overline{x}) = e) \in \overline{fd} \qquad names(\overline{fd}) = \{f_1, \ldots, f_n\} \\ \rho'; f_i \mapsto (\rho', \overline{fd}, f_i); \overline{x} \mapsto \rho(\overline{y}) \vdash e \Downarrow_k v\end{array}}{\rho \vdash f\ \overline{y} \ \Downarrow_{k+1} v}\ (\text{E\_TAILCALL})$$

$$\frac{\begin{array}{c}\rho(f) = (\rho', \overline{fd}, f) \\ (f(\overline{x}) = e) \in \overline{fd} \qquad names(\overline{fd}) = \{f_1, \ldots, f_n\} \\ \rho'; f_i \mapsto (\rho', \overline{fd}, f_i); \overline{x} \mapsto \rho(\overline{y}) \vdash e \Downarrow_k w \\ \rho; x \mapsto w \vdash e \Downarrow_k v\end{array}}{\rho \vdash \mathsf{let}\ x = f\ \overline{y}\ \mathsf{in}\ e \ \Downarrow_{k+1} v}\ (\text{E\_CALL})$$

$$\frac{\rho(y) = v}{\rho \vdash \mathsf{halt}\ y \ \Downarrow_k v}\ (\text{E\_HALT}) \qquad \frac{\rho; x \mapsto p \vdash e \Downarrow_k v}{\rho \vdash \mathsf{let}\ x = p\ \mathsf{in}\ e \ \Downarrow_k v}\ (\text{E\_PRIMVAL})$$

$$\frac{\rho(\overline{y}) = \overline{v} \qquad op\ \overline{v} = w \qquad \rho; x \mapsto w \vdash e \Downarrow_k v}{\rho \vdash \mathsf{let}\ x = op\ \overline{y}\ \mathsf{in}\ e \ \Downarrow_k v}\ (\text{E\_PRIMOP})$$

$$\frac{\begin{array}{c}\rho; f_1 \mapsto (\rho, \overline{fd}, f_1); \ldots; f_n \mapsto (\rho, \overline{fd}, f_n) \vdash e \Downarrow_k v \\ names(\overline{fd}) = \{f_1, \ldots, f_n\}\end{array}}{\rho \vdash \mathsf{let}\ \overline{fd}\ \mathsf{in}\ e \ \Downarrow_k v}\ (\text{E\_FUNDEF})$$

$$names(\overline{fd}) := \{f \mid (f(x) = e) \in \overline{fd}\}$$

Figure 3. Big-step operational semantics of λ_{ANF}. See [41, Figure 3.1] and [34, §3].

that it only features structured control flow with if-then-else and loops, but no goto. This means that compilation from languages with non-structured control requires a 'relooping' stage [51]. However, this does not pose problems when generating code from λ_{ANF}.

CertiCoq-Wasm is mechanised with respect to the WasmCert formalisation of WebAssembly 2.0 with the tail call extension [40]. In verifying the integer primitives, we identified a corner case in the shift operation of WasmCert, which was treating the shift like a C shift, which assumes there is a bound on how much shifting is allowed, whereas in WebAssembly, there is no bound, and the amount of shift is instead treated modulo. The generated binary only uses features from WebAssembly 1.0 and tail call instructions; for compatibility, CertiCoq-Wasm can also be made to generate strict WebAssembly 1.0 binaries.

$$\begin{array}{lll}
\textit{(Value Type)} & t ::= & \mathsf{i32} \mid \mathsf{i64} \mid \ldots \\
\textit{(Instruction)} & \textit{instr} ::= & t.\mathsf{const} \mid t.\mathsf{add} \mid \ldots \mid t.\mathsf{xor} \mid \ldots \mid \\
 & & t.\mathsf{load}\ \textit{memarg} \mid t.\mathsf{store}\ \textit{memarg} \mid \\
 & & \mathsf{memory.grow} \mid \\
 & & \mathsf{nop} \mid \mathsf{if}\ \textit{bty}\ \textit{instrs}_1\ \mathsf{else}\ \textit{instrs}_2\ \mathsf{end} \mid \\
 & & \mathsf{call}\ \textit{funcidx} \mid \\
 & & \mathsf{call_indirect}\ \textit{tableidx}\ \textit{funcidx} \\
\textit{(Module)} & m ::= & \left\{ \begin{array}{l} \mathsf{funcs} : \mathsf{list}\ \textit{func}, \\ \mathsf{mem} : \mathsf{option}\ \textit{mem}, \\ \mathsf{start} : \mathsf{option}\ \textit{funcidx}, \\ \mathsf{tables} : \mathsf{list}\ \textit{table} \\ \ldots \end{array} \right\}
\end{array}$$

Figure 4. Syntax of WebAssembly (excerpts). The type of modules is simplified for presentation purposes.

2 CertiCoq-Wasm

2.1 Code Generation

CertiCoq-Wasm generates a Wasm module with a `main` function that can be invoked from a Wasm runtime host, like Chrome's V8 and Firefox's SpiderMonkey in the browser, or Node.js and Wasmtime on the server side. CertiCoq-Wasm proper takes a λ_{ANF} expression and generates a WebAssembly module. When combined with the earlier CertiCoq pipeline, as in Figure 1, it takes a Gallina expression.

Functions & Globals. The generated module has a `main` function, whose body is the compiled main λ_{ANF} expression. When called, it writes its result to the `result` variable, or sets the global `out_of_mem` to 1.

The translated λ_{ANF} functions follow the main function: they have the type $\mathtt{i32}^n \to []$, where n is the arity of the original λ_{ANF} function. Just like the main function, these functions return their result via the global `result`.

Moreover, functions maintain a common 'high water mark' of memory consumption in the global `mem_ptr`, abbreviated as `gmp`, which we explain in Section 2.1.1.

Table. The module contains a table that is initialised with an identity mapping to support indirect function calls. WebAssembly 1.0 does not have native reference types or pointers, so a function's i32 index identifies a function. An indirect call instruction then takes the index and calls the function specified in the table.

Linear Memory. The generated Wasm module contains a linear memory, in which we store boxed constructor values and primitive values. It is grown dynamically as required using the `memory.grow` operation, up to a maximum size of 1920 MB.

2.1.1 Memory Management.

Binaries generated by CertiCoq-Wasm use naive memory management, without garbage collection. Once allocated, values remain in the linear memory until the end of the execution. To ensure that values are never overwritten, the module maintains a global `gmp` 'high water mark' which always points to the next free space in the linear memory. Whenever a new segment of a certain size is allocated, the `gmp` is increased by that size; it is never decreased.

Before every allocation, a check is performed whether the linear memory is large enough or has to be grown dynamically. CertiCoq-Wasm keeps track of available memory statically and only inserts dynamic checks when necessary. If a dynamic check fails to provide enough memory, execution terminates with the global `out_of_mem` set to 1.

Prototype with Garbage Collection. There are several ways to refine this approach with garbage collection. One can, like CertiCoq's C backend [41, §4.1.6][48], adjust CertiCoq-Wasm to generate bindings to a GC written and verified in the target language.

Alternatively, one can rely on the target language: in our case, make use of Wasm's recent *WasmGC* proposal, which extends WebAssembly with heap-allocated memory for which the runtime performs garbage collection. This proposal is standardised and available in most (but not yet all) important Wasm runtimes[3].

We follow the second approach, as we presume that a GC implemented in Wasm would likely be slower, since it would have to maintain a shadow-stack, as Wasm programs can't access the call stack to scan for live roots.

We thus develop a prototype[4] of CertiCoq-Wasm targeting *WasmGC*, which we include in the performance evaluation but don't describe further here. The prototype is not verified, since WasmCert does not (yet) support the *WasmGC* proposal, but we expect it would be straightforward to adapt our correctness proof.

This GC proposal might not be appropriate for all applications, for example in the blockchain context where execution is typically required to be deterministic. However, for such applications, 2 GB of linear memory is typically enough and not a limitation in practice.

2.1.2 Representing λ_{ANF} Values in Wasm.

The code generation function of CertiCoq-Wasm translates a λ_{ANF} expression to a list of Wasm instructions with the same behaviour. However, since λ_{ANF} and Wasm have different execution models and different representations — λ_{ANF} works with ADTs, while Wasm works with bytes — we need to bridge that gap. We capture what it means for a Wasm program to evaluate to the same value as a λ_{ANF} expression with the *value relation* of Figure 5, which relates a λ_{ANF} value to a Wasm i32 value.

[3] https://webassembly.org/features/
[4] https://github.com/womeier/certicoqwasm/tree/wasmgc

(VR_FUN)

$$\frac{(f(\overline{y}) = e) = \overline{fd}_{idx-1} \quad sr.\text{funcs}_{idx} = F \quad F.\text{type} = (\text{i32}^{|\overline{y}|} \to []) \quad F.\text{body} = \mathit{codegen}\ e \quad F.\text{locals} = \text{i32}^{|\text{bound\_vars}(e)|}}{\left(\rho, \overline{fd}, f\right) \simeq^{\text{val}}_{sr} idx}$$

(VR_CONSTR_BOXED)

$$\frac{\overline{v} \neq \text{nil} \quad \exists n, ptr = 2 \cdot n \quad sr.\text{mems}_0 = m \quad ptr + 4(|\overline{v}| + 1) \leq sr.\text{globals}_{gmp} \quad m[ptr, ptr+4] = C \quad \forall v_i \in \overline{v}.\, v_i \simeq^{\text{val}}_{sr} m \begin{bmatrix} ptr + 4(i+1), \\ ptr + 4(i+2) \end{bmatrix}}{(C, \overline{v}) \simeq^{\text{val}}_{sr} ptr}$$

(VR_CONSTR_UNBOXED)

$$\frac{\overline{v} = \text{nil}}{(C, \overline{v}) \simeq^{\text{val}}_{sr} (2 \cdot C + 1)}$$

(VR_PRIM_VAL)

$$\frac{sr.\text{mems}_0 = m \quad ptr + 8 \leq sr.\text{globals}_{gmp} \quad m[ptr, ptr+8] = p}{p \simeq^{\text{val}}_{sr} ptr}$$

Figure 5. Value relation. Relates a λ_{ANF} value to its representation as a Wasm i32 value.

Function Value. A λ_{ANF} function value $(\rho, \overline{fd}, f)$ is related to the i32 *idx* identifying a Wasm function F. F takes arguments of type i32 according to the arity of the λ_{ANF} function. Recall that all functions return their result via the global variable `result`.

Constructor Value. CertiCoq-Wasm's representation of a λ_{ANF} constructor value $C(\bar{v})$ is based on the one used by CertiCoq's C backend [41, §4.1], which in turn is based on the one used by the OCaml compiler. For efficiency, we represent nullary constructor values as *unboxed* i32 values (using 31 bits), while non-nullary constructor values are *boxed*: they are represented by an address starting from which linear memory contains the code of the constructor, followed by the arguments. The least significant bit of the i32 is used to distinguish the two.

Primitive Value. We represent Coq's 63-bit integers in Wasm with an i32 pointing to an i64 in the linear memory. For every integer primitive operation, the arguments are loaded and the result stored. We could instead trade time for space and use i64s to represent each λ_{ANF} value, which would avoid this indirection, but doing that would double memory usage for programs without primitives. We do not implement float and array primitives.

2.1.3 Translation of λ_{ANF} Expressions.

The function *translate_body* compiles a λ_{ANF} expression to a list of WebAssembly instructions that simulate the behaviour of the source expression. It is called on every λ_{ANF} expression: the main expression, and the body of every function definition. In the correctness proof, we use the — more convenient — relational version of this function, which we call the *codegen relation*. It depends on multiple mappings, in particular `lenv`, which maps the let-bound λ_{ANF} variables to Wasm locals, and `fenv`, which maps λ_{ANF} function variables to their function index in Wasm. We omit these mappings in the following when they are unambiguous.

2.2 Correctness

The correctness statement of CertiCoq-Wasm concerns compilation from CertiCoq's λ_{ANF} to WebAssembly. Once CertiCoq has an end-to-end correctness theorem linking its verified stages, we can combine our result with CertiCoq's internal correctness to get a Gallina-to-Wasm result.

CertiCoq-Wasm's correctness guarantee is given in Theorem 2.1. It states that running the main function of the generated Wasm module results in a store sr', where the `result` variable contains a value related to the corresponding λ_{ANF} value (or the execution ran out of memory).

The proof goes by forward simulation over the operational semantics of λ_{ANF} and Wasm. For every evaluation step of a λ_{ANF} expression, the generated Wasm instructions simulate the original behaviour. After the execution of the main function, the `result` variable is set correctly. The proof of Theorem 2.1 is visualised in Section 2.2.1 and explained in the following section.

Theorem 2.1 (Correctness of lowering). *For any closed, well-formed λ_{ANF} expression e with globally unique bound variables, well-formed constructor environment, and well-formed prim_funs environment (see §2.3), if compilation is successful, the resulting Wasm module instantiates and its 'main' function evaluates to the same value as e, or runs out of memory:*

$$\begin{pmatrix} e = \texttt{let}\ \overline{fd}\ \texttt{in}\ e' \wedge \\ \cdot \vdash e \Downarrow v \wedge \\ compile\ e = (mod, \ldots) \end{pmatrix} \Longrightarrow$$

$$\exists sr, sr', fr. \begin{pmatrix} instantiate\ mod\ (sr, fr) \wedge \\ (sr, fr, [\text{call}\ idx_{\text{main}}]) \hookrightarrow^* (sr', fr, []) \ \wedge \\ \left(v \simeq^{\text{val}}_{sr'} sr'.\text{globals}_{\text{result}} \vee sr'.\text{globals}_{\text{out\_of\_mem}} = 1\right) \end{pmatrix}$$

where 'compile' is our extraction to Wasm, which generates a Wasm module mod. The module is then instantiated, inducing a 'store' sr and a 'frame' fr. Calling the main function reduces to the final 'store' sr' holding the result.

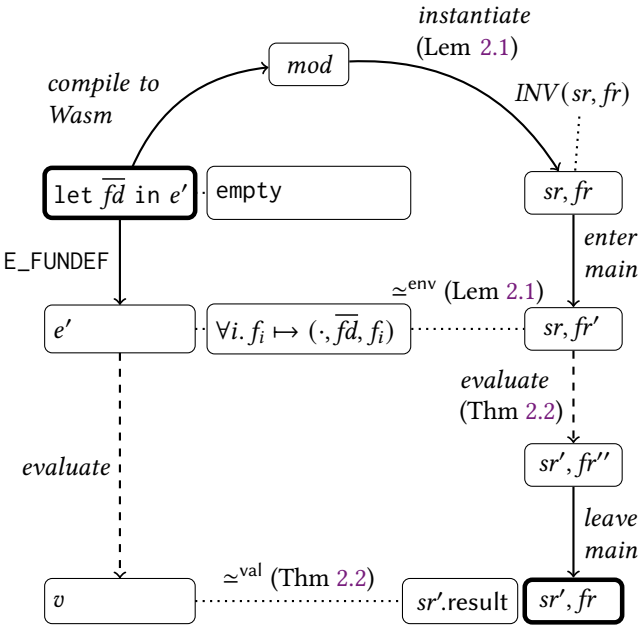

Figure 6. Visualisation of the proof of Theorem 2.1. Dashed arrows indicate multiple steps.

Proof. The proof is visualised in Figure 6 and explained in the following sections. It follows from Theorem 2.2 (Generalised correctness of lowering) and Lemma 2.1 (Module instantiation). □

This statement is complemented by the determinacy of the fragment of WebAssembly that CertiCoq-Wasm targets, which is also crucial for the use of WebAssembly on the blockchain. We make use of the fact that non-determinism may only occur locally[5] in WebAssembly 1.0. For example, CertiCoq-Wasm generates binaries that abort the execution when a memory allocation fails. Another common source of non-determinism are floating point operations, which we don't use. (However, we may do so in the future to support translating Coq's primitive float operations.)

2.2.1 Correctness Proof Visualised.

Figure 6 illustrates the structure of the proof of Theorem 2.1.

λ_{ANF} Evaluation (Left, Downwards). The top-level λ_{ANF} expression (thick border, top left) defines a set of functions $\overline{fd}$. It is evaluated on an empty environment, resulting in a final value v. The first evaluation step (`E_FUNDEF`) extends the environment with the function names f_i from the $\overline{fd}$. Then, the continuation e evaluates to the value v under this new environment.

[5] https://github.com/WebAssembly/design/blob/main/Nondeterminism.md

Compilation and Instantiation (Horizontal). The λ_{ANF} expression is compiled to a module *mod* using CertiCoq-Wasm. This module is then instantiated, inducing the store *sr* and the frame *fr*. After instantiation, the set of invariants *INV* holds, and *sr*.funcs contains the translated functions. The invariants *INV* are preserved until the final store sr'.

WebAssembly Evaluation (Right, Downwards). The main function is entered; fr' is its frame right after function entry. The body is executed, inducing a store sr', which has the `result` variable set correctly. The reduction of the body follows from the generalised correctness Theorem 2.2. To apply it, one has to show its various assumptions, in particular (1) that the set of invariants *INV* holds, and (2) that the environment relation holds. At this point, the environment relation only says that the functions $\overline{fd}$ are present in store *sr*.

2.2.2 Environment Relation.

Before stating the generalised correctness theorem, we introduce what we call the *environment relation* (the correctness statement of the C backend uses a similar relation [41, §4.3.1], which the author calls the *memory relation*). It captures what it means for a λ_{ANF} expression to correspond to a Wasm program by relating a λ_{ANF} environment to the Wasm environment representing the same state, see Figure 7.

Functions. For function values obtained from ρ, the environment relation provides the index of the corresponding Wasm function.

The first conjunct in Figure 7 states that all function values in ρ refer to the top-level set of functions. This includes e.g. when a function value is an argument of a constructor value, which is captured by the reflexive-transitive closure of the (structural) subvalue relation, denoted by $\sqsubseteq$. The ρ' is empty because function definitions are initially added (see rule `E_FUNDEF`) to the empty environment, and there is no way to obtain other function values.

The second conjunct in Figure 7 states that the top-level functions $\overline{fd}$ are translated correctly, and are present in the store.

Free Variables. For every free variable in *e*, the environment relation provides a local variable holding an i32 value, that is related to the corresponding λ_{ANF} value via the value relation.

Recall that let-bound λ_{ANF} variables are mapped to locals in Wasm. Whenever a variable *x* is let-bound, the continuation is evaluated in the original environment extended with *x* (see e.g. rule `E_PROJ` in Figure 3). When executing the Wasm instructions generated for the continuation, the values of the free variables, including *x*, must be available in the frame's locals. This is captured by the third conjunct in Figure 7.

2.2.3 Generalised Correctness.

We can now state the generalised correctness of lowering in Theorem 2.2.

$$
\begin{aligned}
&\rho \simeq^{\mathsf{env}}_{fd,e} (sr, fr) \;:=\\
&\left(\begin{array}{l}\forall x, v, \rho', fd', f.\\ \rho(x) = v \;\wedge\; (\rho', \overline{fd'}, f) \sqsubseteq v \implies\\ f \in \mathit{names}(\overline{fd}) \;\wedge\; \rho' = \cdot \;\wedge\; \overline{fd} = \overline{fd'}\end{array}\right) \wedge\\
&\left(\begin{array}{l}\forall f.\\ f \in \mathit{names}(\overline{fd}) \implies\\ \exists fi.\ \mathit{repr\_funvar}\ f\ fi \;\wedge\; (\cdot, \overline{fd}, f) \simeq^{\mathsf{val}}_{sr} fi\end{array}\right) \wedge\\
&\left(\begin{array}{l}\forall x.\\ x \in \mathit{free\_vars}(e) \implies\\ \exists v, v', l.\ \rho(x) = v \;\wedge\; v \simeq^{\mathsf{val}}_{sr} v' \;\wedge\; fr.\mathsf{locals}_l = v'\end{array}\right)
\end{aligned}
$$

Figure 7. Environment relation. Relates a λ_{ANF} environment ρ to Wasm store *sr* and frame *fr*.

The theorem states that, given a λ_{ANF} expression *e* that evaluates to a value *v* in a context ρ, the Wasm instructions e' which are generated for *e* simulate the behaviour of *e*. Executing e' ultimately results in a store sr' which has the `result` variable set to an i32 related to *v*.

The statement requires the presence of a function frame for the tail-call case. During a tail invocation, the frame is replaced with the frame of the called function. Moreover, the fact that all variables *x* let-bound in *e* are not yet set in ρ stems from the fact that all λ_{ANF} variables are globally unique. Thus, values in ρ are not overwritten or 'shadowed'. This transfers to the generated module: locals are only set once.

Theorem 2.2 (Generalised correctness of lowering)**.** *For any well-formed, size-restricted λ_{ANF} expression e with globally unique bound variables, a global set of function definitions $\overline{fd}$, a block context B^k, and well-formed prim_funs environment, constructor environment, local environment, and function environment, if*

$$
\left(\begin{array}{l}\rho \vdash e \Downarrow v \;\wedge\; \mathit{codegen}\ e = e' \;\wedge\\ \mathit{INV}(sr, fr) \;\wedge\; \rho \simeq^{\mathsf{env}}_{fd,e} (sr, fr) \;\wedge\\ \text{translated } \overline{fd} \text{ in } sr.\mathsf{funcs}\\ (\forall x.\, x \text{ let-bound in } e \implies x \notin \rho)\end{array}\right)
$$

then we have

$$
\begin{aligned}
&\exists sr', fr', \mathsf{C}^{k'}.\\
&\left(\begin{array}{l}(sr, \ldots, \mathtt{frame}_0\ \{fr\}\ \mathsf{B}^k[e']\ \mathtt{end}) \hookrightarrow^*\\ (sr', \ldots, \mathtt{frame}_0\ \{fr'\}\ \mathsf{C}^{k'}[\mathtt{return}]\ \mathtt{end}) \wedge\\ \\ \left(\begin{array}{l}(v \simeq^{\mathsf{val}}_{sr'} sr'.\mathsf{globals}_{\mathrm{result}} \wedge \mathit{INV}(sr', fr'')) \;\vee\\ sr'.\mathsf{globals}_{\mathrm{out\_of\_mem}} = 1\end{array}\right) \wedge\\ (\forall v, v'.\, v \simeq^{\mathsf{val}}_{sr} v' \implies v \simeq^{\mathsf{val}}_{sr'} v')\end{array}\right)
\end{aligned}
$$

where 'INV' is the set of invariants described in Section 2.2.4, and the various environments are omitted for readability. $\mathsf{C}^{k'}$ is a new (possibly deeper) block context.

Proof. The proof goes by induction over the big-step evaluation derivation $\rho \vdash e \Downarrow v$, as per the rules in Figure 3.

It differs from the proof of the C backend [41, §4.3] due to Wasm specifics and the different memory management.

Stepping through e' may reach a continuation that is to be executed from a store and a frame. To use the induction hypothesis to step through the continuation, we need to show the two main assumptions that (1) the invariants hold and that (2) the environment relation holds:

(1) The generated Wasm instructions preserve the invariants about mutability, bounds, etc. as required.

(2) If the expression let-binds a variable x, then one has to show that the environment relation holds on the original environment ρ extended with x set to a value v, assuming that it holds on the original ρ, fr and sr. This is done by showing its three conjuncts.
(2.1) For the first one, one has to show that the value v only contains function values referring to the top-level definitions. This is the case, since the environment relation is known to hold on the original ρ, and one can only obtain a function value present in ρ, since it is not possible to construct a function value in any other way.
(2.2) For the second one, one has to show that the store's funcs contain the translated top-level function definitions $\overline{fd}$. This is the case because it holds for the previous store, and the funcs of a store are not modified during execution.
(2.3) The third one is central to the proof. It enforces that, after executing the instructions generated for the particular expression, the Wasm local x' holds a Wasm i32 related to v via the value relation (where the local environment maps x to x'). Further, the values provided by ρ are still related to the fr and sr. This is the case because the generated code is in SSA form: locals previously assigned are not overwritten. Similarly, values in the linear memory are not overwritten, as per our memory management, see Section 2.1.1.
The intermediate i32 values are kept in the locals, and every function body ends with a `halt`. The Wasm instructions generated for this expression set the result variable correctly. □

2.2.4 Invariants. A set of invariants *INV* is required to hold on the Wasm store and frame throughout execution. They capture the pre-conditions of Wasm's reduction rules: For example, that certain globals are mutable, or that pointers to the linear memory are in-bound (and that loads and stores therefore succeed).

2.2.5 Instantiation. In WebAssembly, a module is a static object, which for example declares a memory, but does not define its contents. A module is turned into its runtime representation, a *module instance* during *instantiation*, which results in a store (which contains the module instance) and a frame. After instantiation, the invariants *INV* hold on the store and frame, and the translated functions are present in the store.

Lemma 2.1 (Module instantiates). *For any λ_{ANF} expression e with globally unique bound variables, and a well-formed constructor environment,*

$$compile\ e\ = (mod, \ldots) \implies$$

$$\exists sr, fr. \begin{pmatrix} instantiate\ mod\ (sr, fr) \wedge \\ INV\ sr\ fr \wedge \\ \text{translated } \overline{fd} \text{ in } sr.\text{funcs} \wedge \\ func_{\text{main}} \text{ in } sr.\text{funcs} \wedge \ldots \end{pmatrix}$$

Proof. One has to show that the module will successfully instantiate, producing a store sr and a frame fr of the right form:

(1) Going through instantiation requires in particular that the generated Wasm module is well-typed.

(2) The set of invariants *INV* has to hold after instantiation. This is true by construction of the module. For example, the global `mem_ptr` is initialised with 0, and is thus in the bounds of the linear memory.

(3) Further, '*compile*' translates the $\overline{fd}$ correctly, so they are present in the store after instantiation. □

2.3 Assumptions

CertiCoq does not yet have an end-to-end correctness statement linking its separately verified stages together. Once that is stated, the following assumptions are required to connect Theorem 2.1 to the rest of CertiCoq, obtaining a Gallina to Wasm result:

Constructor Environment. The constructor environment maps every constructor tag to a record that contains its arity, its constructor ordinal (an integer uniquely identifying this constructor), and the inductive type it belongs to. The Wasm backend expects this environment to be well-formed. In particular, the arity has to agree with the number of arguments that constructor expressions in e are actually applied to. Furthermore, CertiCoq-Wasm expects that certain constructors are given a specific constructor ordinal. For example, the translation of the primitive operation `eqb` depends on the fact that `true` and `false` are assigned 1 and 0, respectively. All constructors (except booleans) are assigned ordinals according to the order Coq's type definition lists them in. The `bool` constructors are swapped by an earlier stage of CertiCoq to achieve the same ordering as OCaml's booleans. This ordering is not checked by the Wasm backend, as there is no clear way to check it.

Prim_funs Environment. The prim_funs environment maps λ_{ANF} primitive operators to the corresponding Coq primitive. We assume that this environment is well-formed in the sense that it maps all (supported) primitive operators to WebAssembly code implementing that operator. Currently, this includes all operations on unsigned 63-bit integers, but

not floats or arrays. Programs using non-supported primitives are rejected, and so not supporting a primitive does not compromise soundness, merely completeness of the compiler.

Size Restrictions. CertiCoq-Wasm requires the λ_{ANF} expression to be constrained in size, so that all relevant values fit in the range of Wasm's i32s. These restrictions are enforced: CertiCoq-Wasm refuses to compile expressions containing an inductive data type with more than $2^{31} - 1$ nullary or non-nullary constructors. The original C backend of CertiCoq makes similar restrictions [10, §5.2]. Given that this limit stems from the constraints of the backend, it does make sense to test and flag it at this stage.

Closed Expression. The λ_{ANF} expression is assumed *not* to have free variables. This is guaranteed by CertiCoq's previous pipeline, and not checked by the Wasm backend.

Unique Bound Variables. The variables of the λ_{ANF} expression are assumed to be globally unique. Again, this is guaranteed by CertiCoq's previous pipeline, and not checked by the Wasm backend.

3 Evaluation

3.1 Benchmarks

We evaluate the performance of WebAssembly code produced by CertiCoq-Wasm for a set of Gallina programs. Extraction mechanisms for Coq are commonly evaluated on CertiCoq's test suite [34, §8.2], from which we select the benchmarks `demo1`, `vs_easy`, `vs_hard`, `binom`, `color` and `sha_fast`. Additionally, we include the Ackermann function due to its many recursive calls as `ack_3_9`, and the `coqprime` benchmark due to its use of Coq's primitive 63-bit integer operations:

- `demo1`: appends two lists of booleans of length 500 and 300.
- `vs_easy`: compiles VeriStar [44] and checks the entailment of a decidable fragment of separation logic
- `vs_hard`: same as above, but for a harder entailment.
- `binom`: constructs two binomials queues, merges them, and finds the maximum [5].
- `color`: graph colouring based on a verified implementation [5] of the Kempe-Chaitin algorithm [11].
- `sha_fast`: computes the SHA-256 sum of a string of length 484.
- `ack_3_9`: computes Ackermann's function on the numbers 3 and 9.
- `coqprime`: verifies the primality certificate of a prime with 100 digits.

3.2 Other Extraction Mechanisms to Wasm

We evaluate CertiCoq-Wasm compared to other extraction mechanisms from Coq to Wasm. Since we are not aware of other tools targeting WebAssembly directly, we combine Coq's various extraction mechanisms with other tools, as follows (Figure 8):

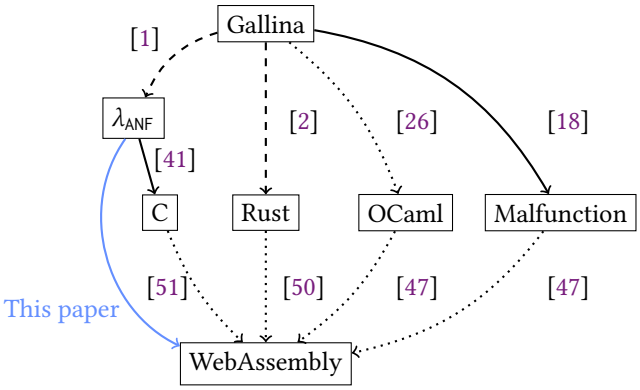

Figure 8. Overview of various tools for extracting Coq programs to WebAssembly. Solid arrows indicate verified, dotted arrows unverified components. The dashed arrow for [2] indicates that the pipeline is partly verified; the other dashed arrow for [1] indicates that the verification is currently a work in progress.

Via C. The extraction from Coq to C via CertiCoq [1], combined with Emscripten [51]. Emscripten is based on LLVM, and is *the* standard tool for compiling C-like languages to Wasm; it compiles the C code generated by CertiCoq linked with a verified garbage collector [48] to WebAssembly. The current version (3.1.58) of Emscripten targets WebAssembly 1.0 by default. It can generate tail call instructions, but this requires compiler directives in the C code, which CertiCoq's C backend doesn't generate.

Via Rust. The extraction from Coq to Rust provided by ConCert's Coq-rust-extraction [2], combined with Wasm-pack [50]. Coq-rust-extraction generates Rust code with a memory model similar to the one of CertiCoq-Wasm. Wasm-pack uses the Rust compiler to generate a Wasm binary, and generates bindings around it for JavaScript applications using e.g. Node.js. The generated binary is compatible with WebAssembly 1.0 (and so does not support tail calls; we are aware of work-in-progress to support them).

Via OCaml. The extraction from Coq to OCaml [26], combined with Wasm_of_ocaml [47]. The generated binary makes use of both the WebAssembly *WasmGC* and the *Tail call* proposal.

Via Malfunction. The verified extraction from Coq to Malfunction [18], combined with Wasm_of_ocaml [47]. Malfunction [16] is a wrapper around the Lambda intermediate representation of the OCaml compiler. The generated binary makes use of both the WebAssembly *WasmGC* and the *Tail call* proposal.

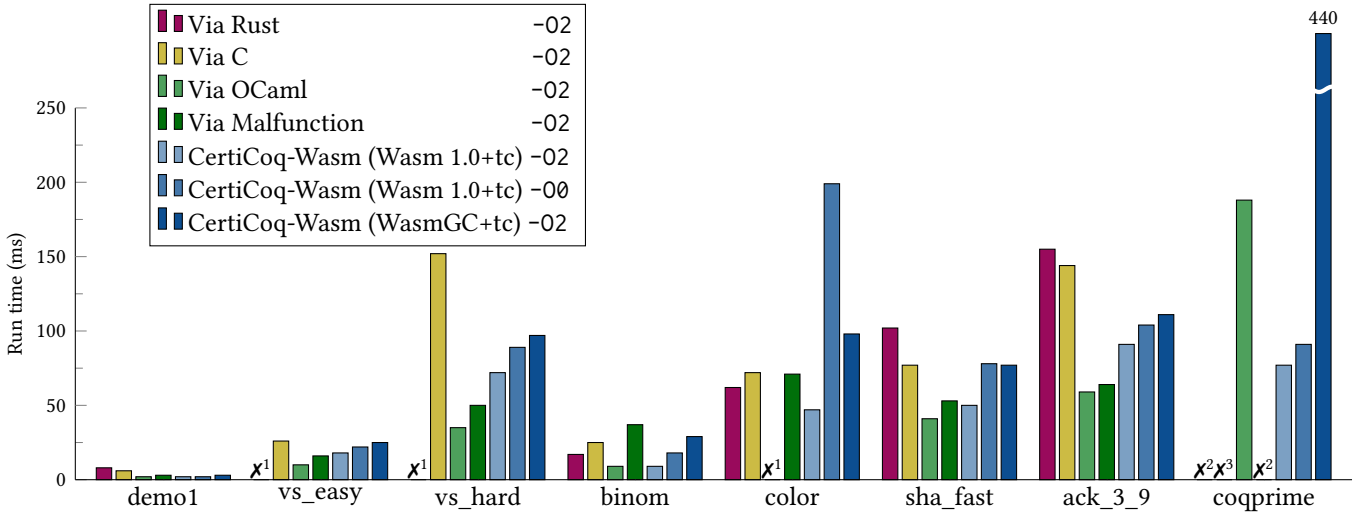

Figure 9. Run time (startup + main) of Wasm binaries from different extraction mechanisms. All binaries are run with Node.js. CertiCoq-Wasm outperforms the ones via Rust and C for all benchmarks but one, and all benchmarks when optimised (`-O2`). The one via OCaml is better than CertiCoq-Wasm except for `coqprime`; CertiCoq-Wasm is on par with the one via Malfunction.

CertiCoq-Wasm. CertiCoq-Wasm generates WebAssembly directly in a verified way, assuming that the earlier stages of CertiCoq are correct. By default, it generates binaries compatible with WebAssembly 1.0 with the tail call extension, but can also produce strict WebAssembly 1.0 code if needed — for example for blockchains such as Concordium. However, we do not make this the default, as these binaries are slower and the call-stack might overflow for larger programs.

3.3 Performance of Different Extraction Mechanisms

In Figure 9, we compare the various extraction mechanisms using the V8 Wasm runtime in Node.js 22.1.0.

We include CertiCoq-Wasm as is (`-O0`) and with optimisations (`-O2`), and the unverified CertiCoq-Wasm prototype targeting *WasmGC*. Optimisations are performed on the final Wasm binaries by binaryen's `wasm-opt` tool. All three versions make use of the *Tail call* proposal. The other extraction tools are run with their respective `-O2` optimisations, and the generated Wasm binaries optimised with `wasm-opt -O2`.

We observe that CertiCoq-Wasm with optimisations outperforms the extractions via Rust and C on all benchmarks, on average by 42.2% (for the benchmarks which extract to Rust) and 42.4%, respectively. Unoptimised CertiCoq-Wasm outperforms the two extractions for all but the `color` benchmark. Both extractions are based on the LLVM infrastructure; we estimate that if they were made to generate tail calls, the gap in performance would be smaller.

The extraction via Malfunction is roughly on par with optimised CertiCoq-Wasm. CertiCoq-Wasm with optimisations is 8% faster than the extraction via OCaml on average over all benchmarks. However, when excluding the `coqprime` benchmark with the primitive integer operations, OCaml is 36% faster than CertiCoq-Wasm.

The binaries obtained via OCaml and Malfunction both make use of *WasmGC*, and both outperform our CertiCoq-Wasm prototype mode targeting *WasmGC*. However, the performance of our prototype could most likely be improved.

Coq's primitive integers are not yet supported by the extractions via Malfunction and Rust. For the `coqprime` benchmark, unoptimised CertiCoq-Wasm is 51% faster than the unverified extraction via OCaml. We suspect that part of this difference is due to OCaml using boxed 64-bit integers.

3.4 Optimisations with Binaryen's `Wasm-opt`

Optimising CertiCoq-Wasm's binaries with `wasm-opt -O2` improves performance by 39% on average across the benchmarks (21% when excluding `color`).

The `color` benchmark benefits the most, which is due to `wasm-opt`'s coalescing of locals (Wasm's local variables). Since CertiCoq puts code in SSA form, many constants in the original Gallina program lead to many locals in the generated Wasm functions. For the `color` binary, which includes the encoding of a sizeable graph, the generated main function has over 20,000 locals. They are coalesced to around 150, making up most of the 3x improvement achieved with `wasm-opt -O2` (Figure 9).

We currently only rely on `wasm-opt` to achieve reasonable performance for such programs. A verified CSE optimisation on CertiCoq's middle-end and verified Wasm optimisations such as local coalescing are left as future work.

Table 1. Linear memory usage of Wasm binaries generated by CertiCoq-Wasm, in KB. All binaries are optimised with `wasm-opt -O2`.

demo1	vs_easy	vs_hard	binom	color	sha_fast	ack_3_9	coqprime
22	6,128	38,141	248	16,515	25,642	44,673	35,546

Table 2. Binary size for Wasm binaries generated by different extraction mechanisms, in KB. All binaries are optimised with `wasm-opt -O2`.

	demo1	vs_easy	vs_hard	binom	color	sha_fast	ack_3_9	coqprime
Rust	73	✗	✗	300	173	364	21	✗
C	67	439	441	260	1296	523	22	✗
OCaml	13	36	37	40	✗	77	7	89
Malfunction	13	43	43	40	174	68	7	✗
CC-Wasm	32	162	162	159	799	281	1	314
CC-WasmGC	7	88	87	36	176	78	1	216

3.5 Memory Usage

Wasm binaries generated by CertiCoq-Wasm store constructor values and primitives in the linear memory. Allocated memory is not freed due to CertiCoq-Wasm's simplistic memory management. We thus measure and report in Table 1, how much linear memory each benchmark uses when calling the main function. We observe that typical programs, including all of CertiCoq's test suite, fit well within the limit of around 2 GB. However, because this limit of 2 GB does not include garbage collection, it is still a limitation for our backend, as we discuss in Section 2.1.1. For example, the `coqprime` benchmark uses a 100-digit prime, it would run out of memory for a 150-digit prime.

3.6 Binary Size

Table 2 shows the size of the binaries generated by CertiCoq-Wasm. CertiCoq-Wasm consistently generates smaller binaries than the extraction via C or Rust (except for the `color` benchmark).

The smallest binaries are generated by extraction mechanisms targeting *WasmGC*. Binary size is particularly important for blockchain applications, as one needs to pay for their permanent storage. As *WasmGC* is not generally available on blockchains, CertiCoq-Wasm remains the best option for blockchain applications.

3.7 Performance of Different Wasm Runtimes

We evaluate CertiCoq-Wasm with two popular Wasm runtimes, Node.js 22.1.0 and Wasmtime 21.0.1 in Figure 10. The

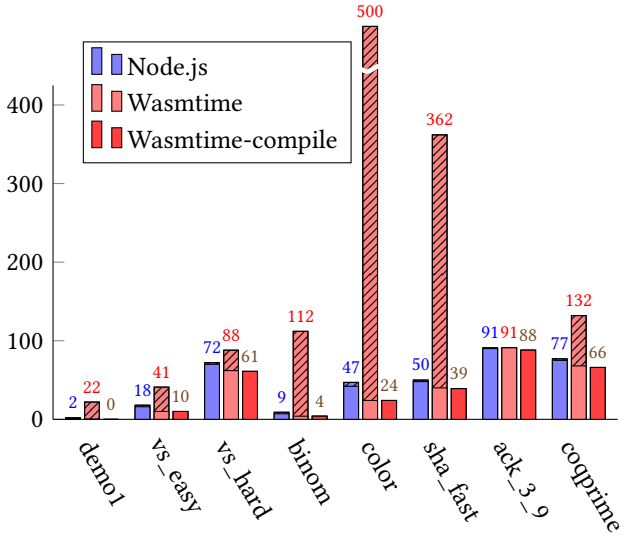

Figure 10. Run time of CertiCoq-Wasm-generated Wasm binaries with different runtimes, in ms. The hatched part is the startup time, the remaining the run time of the main function. All binaries are optimised with `wasm-opt -O2`.

former uses Google's V8 engine that also powers Chrome; the latter is popular for non-browser Wasm applications[6].

We show the startup time, which is the time to load the Wasm binary, transfer it into the engine's runtime representation, and instantiate it, in hatched. Node.js outperforms Wasmtime significantly on all benchmarks except `ack_3_9`. For the `color` benchmark, Node.js is 20x faster than Wasmtime.

Startup time makes up the lion's share of Wasmtime's run time, while it is negligible for Node.js. This is because Wasmtime performs Ahead-Of-Time compilation: a Wasm binary is first compiled to x86 that is then run. Node.js performs JIT compilation consisting of two compilers: the fast and simple *Liftoff* compiler produces runnable code in a single pass, and hot functions are recompiled with *Turbofan*, a sophisticated multi-pass compiler.

However, the recommended way of using Wasmtime includes pre-compiling to x86 (using Wasmtime), whenever the Wasm binary is available statically. This is measured as Wasmtime-compile. With pre-compilation, Wasmtime is actually faster than Node.js.

3.8 Trusted Computing Base

Because CertiCoq-Wasm builds on the CertiCoq front-end and middle-end, it inherits all of CertiCoq's "upper" assumptions. This includes, in particular, MetaCoq's verified erasure [43], CertiCoq's ANF transformation (whose proof is

[6]To allow evaluating other Wasm runtimes, the Wasm binaries for our benchmarks and our testing setup are available at https://github.com/womeier/certicoqwasm-testing.

currently work in progress), and CertiCoq's various verified optimisations on λ_{ANF} [35].

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.

CertiCoq-Wasm is almost fully implemented in Gallina; the only exception is minimal OCaml code used to register our backend in the CertiCoq plugin.

4 Applications

We demonstrate the practical usability of CertiCoq-Wasm with two case studies, in two key use cases of WebAssembly: on the web, and on the blockchain.

4.1 JavaScript Application

We show how one can integrate Wasm binaries generated by CertiCoq-Wasm in a JavaScript application. We use CertiCoq-Wasm to extract a Gallina implementation of the SHA-256 algorithm verified in Coq [4]. The generated Wasm binary is integrated into a simple web interface, so one can compute the SHA-256 sum of a custom string. The site is available at https://womeier.de/certicoqwasm-demo.html.

4.2 Blockchain Application

In a second case study, we use CertiCoq-Wasm to extract a ConCert [3] smart contract to a Wasm module, and deploy it on the Concordium blockchain [12].

We select the `counter` contract from ConCert's test suite, and extract it using CertiCoq-Wasm using the strict 1.0 mode, modified to insert all the function imports from the Concordium API. The binary is combined with (unverified) Wasm glue code that bridges the gap between the extracted contract and the Concordium API. Thanks to CertiCoq-Wasm's correctness, the generated smart contract can only go wrong due to a mistake in the glue code. We deployed it on Concordium's testnet, and observed that it behaves as expected: one can initialise the counter and increase its value.

The extraction is currently tailored towards the `counter` contract and Concordium. It could be generalised to extract other contracts to Concordium, or target other blockchains based on WebAssembly. Currently, the Wasm glue code has to be adapted to every contract; this could be made automatic.

5 Related Work

To the best of our knowledge, CertiCoq-Wasm is the only verified compiler from Coq to WebAssembly (up to the caveats of CertiCoq of §1.1) at the time of writing. However, there are several extraction mechanisms that target more or less mainstream languages:

Coq comes with a built-in extraction-mechanism to OCaml, Haskell, and Scheme [26]. Several projects, such as CertiCoq [1] and CompCert [25], depend on the OCaml extraction. Even though it works well in practice, it is not verified. Multiple bugs have been found, e.g. CertiCoq's `color` benchmark is extracted to ill-formed OCaml code.

The verified Malfunction extraction [18] improves upon the current extraction to OCaml, and there are plans for the former to replace the latter in the near future. It targets Malfunction [16], a functional language that is very close to one of the OCaml compiler's intermediate representations. The Malfunction code can be compiled to OCaml byte code that is compatible with the one obtained using the unverified, built-in extraction.

The CertiCoq [1] compiler allows extracting Coq to Clight, a subset of the C programming language. Most stages of its pipeline are proven correct, including its various optimisation passes. Combining it with CompCert allows extracting Assembly in a verified way, but not WebAssembly, since CompCert does not target Wasm at this time. By extending CompCert with such a backend one could also obtain a Gallina-to-Wasm pipeline. However, the generated binaries may suffer similar performance and size limitations to the ones obtained via Emscripten. Further, when targeting WasmGC, the generated Wasm binaries would be larger (and likely less efficient) as they include CertiCoq's garbage collector.

Similar to CertiCoq, Œuf [31] compiles Gallina to C in a verified way. It is aimed at compiling specific Coq applications and thus fairly limited in practice. As opposed to CertiCoq (and CertiCoq-Wasm), it cannot translate user-defined inductive data types, pattern matching or recursion.

ConCert [3] is a framework for developing and verifying smart contracts in Coq. To deploy such a smart contract on a blockchain, it has to be extracted to the respective smart contract language. ConCert includes extraction mechanisms [2] to Rust and Elm [17] and also supports extracting to OCaml-like languages, like Liquidity [9]. CertiCoq-Wasm can be used to extract ConCert smart contracts directly to Wasm, e.g. for the Concordium blockchain. Our extraction has a smaller TCB compared to ConCert's Concordium extraction, which depends on the Rust compiler. However, as opposed to ConCert's Concordium extraction via Rust, one currently has to adapt the Wasm glue code for each contract; this could be made automatic.

Pilsner [32] is a verified compiler from a subset of ML to an assembly-like target language. Their main contribution is in composability: multiple modules can be compiled separately, and their results linked in a verified way. The main parts of Pilsner's correctness proof are parametrised over the source and target languages, which allows linking modules obtained even from different compilers. CertiCoq-Wasm is concerned with *whole* program compilation. It would be possible to extend CertiCoq-Wasm with a foreign function interface on the Wasm level, making use of CertiCoq's VeriFFI [24].

Further afield, Isabelle's code generation, like the classic extraction of Coq, relies on the closeness of the source

and target languages: SML, OCaml, Haskell, and Scala. Code generation for SML was verified [22, 23] with respect to CakeML's subset of SML [45], with which it can then be combined to generate assembly (we are aware of a project to make CakeML generate WebAssembly [28]). In addition, Isabelle has been extended with a Go backend, which, like CertiCoq, requires more extensive compilation [45]. To our knowledge, Lean, Minlog, and Nuprl's extraction mechanisms are to date unverified.

Finally, jsCoq [6] is a port of Coq itself (of its OCaml code, not of Gallina code) to JavaScript using is_of_ocaml. As a side effect of our project, one could implement `native_compute` in jsCoq to generate WebAssembly and run directly in the browser, bypassing the layers of indirection that `vm_compute` has to go through.

6 Proof Effort and Experience

Our main contribution is the artefact that allows compiling Gallina to WebAssembly in a verified way. The earlier CertiCoq-Wasm prototype [29] was the result of adapting CertiCoq's C backend in the canonical way to target WebAssembly. We subsequently addressed the limitations reported for this prototype and substantially improved overall performance. This required engineering work, but no new proof-techniques.

The proof took about 21 person-months of work, and totals to about 4 kLoC of specification, and 13 kLoC of proof (using `coqwc`, without comments).

Acknowledgments

We thank Zoe Paraskevopoulou for suggesting the project and providing advice. We are grateful to Eske Hoy Nielsen and Andreas Stenbæk Larsen for advice regarding the smart contract demo, and Xiaojia Rao regarding the WasmCert mechanisation. Finally, we thank the anonymous reviewers for their comments. 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 *The third international workshop on Coq for programming languages (CoqPL)*.

[2] Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. 2022. Extracting functional programs from Coq, in Coq. *Journal of Functional Programming* 32 (2022), e11. https://doi.org/10.1017/S0956796822000077

[3] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. 2020. ConCert: a smart contract certification framework in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 215–228. https://doi.org/10.1145/3372885.3373829

[4] Andrew W Appel. 2015. Verification of a cryptographic primitive: SHA-256. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37, 2 (2015), 1–31. https://doi.org/10.1145/2737924.2774972

[5] Andrew W Appel. 2022. Verified Functional Algorithms (Software Foundations, Vol. 3). *Electronic textbook* (2022).

[6] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. 2016. jsCoq: Towards Hybrid Theorem Proving Interfaces. In *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers, UITP 2016, Coimbra, Portugal, 2nd July 2016 (EPTCS, Vol. 239)*. 15–27. https://doi.org/10.4204/EPTCS.239.2

[7] Benl, Berger, Schwichtenberg, Seisenberger, and Zuber. 1998. *Proof Theory at Work: Program Development in the Minlog System*. Springer Netherlands, Dordrecht, 41–71. https://doi.org/10.1007/978-94-017-0435-9_2

[8] 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

[9] Çagdas Bozman, Mohamed Iguernlala, Michael Laporte, FL Fessant, and Alain Mebsout. 2018. Liquidity: Ocaml pour la blockchain. *Journées Francophones des Langages Applicatifs* 2018 (2018).

[10] Olivier Savary Bélanger, Matthew Z. Weaver, , and Andrew W. Appel. 2019. *Certified Code Generation from CPS to C*. Technical Report. Princeton University.

[11] Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. 1981. Register allocation via coloring. *Computer languages* 6, 1 (1981), 47–57. https://doi.org/10.1016/0096-0551(81)90048-5

[12] Concordium Development Team. 2021. Concordium whitepaper, version 1.8.1. https://go.concordium.com/hubfs/White%20paper%20-%20RWR/Concordium%20White%20Paper%20v1.8.pdf

[13] Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, Robert Harper, Douglas J. Howe, Todd B. Knoblock, Nax Paul Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. 1986. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall. https://nuprl-web.cs.cornell.edu/book/

[14] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[15] Coq development team. 2023. The Gallina specification language. https://coq.inria.fr/doc/V8.18.0/refman/language/gallina-specification-language.html

[16] Stephen Dolan. 2016. Malfunctional programming. In *ML Workshop*.

[17] Richard Feldman. 2020. *Elm in action*. Manning Publications.

[18] Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. 2024. Verified Extraction from Coq to OCaml. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 52–75. https://doi.org/10.1145/3656379

[19] Stéphane Glondu. 2009. Extraction certifiée dans Coq-en-Coq. In *JFLA 2009, Vingtièmes Journées Francophones des Langages Applicatifs, Saint Quentin sur Isère, France, January 31 - February 3, 2009. Proceedings (Studia Informatica Universalis, Vol. 7.2)*, Alan Schmitt (Ed.). 383–410.

[20] Stéphane Glondu. 2012. *Vers une certification de l'extraction de Coq. (Towards certification of the extraction of Coq)*. Ph. D. Dissertation. Paris Diderot University, France.

[21] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 185–200. https://doi.org/10.1145/3062341.3062363

[22] Lars Hupel. 2019. *Verified Code Generation from Isabelle/HOL*. Ph. D. Dissertation. Technical University of Munich, Germany. https://nbn-resolving.org/urn:nbn:de:bvb:91-diss-20190711-1473785-1-3

[23] Lars Hupel and Tobias Nipkow. 2018. A Verified Compiler from Isabelle/HOL to CakeML. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the*

European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801), Amal Ahmed (Ed.). Springer, 999–1026. https://doi.org/10.1007/978-3-319-89884-1_35

[24] Joomy Korkut, Kathrin Stark, and Andrew W. Appel. 2025. A Verified Foreign Function Interface Between Coq and C. *Proc. ACM Program. Lang.* 9, POPL (2025).

[25] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[26] Pierre Letouzey. 2002. A New Extraction for Coq.. In *TYPES (Lecture Notes in Computer Science, Vol. 2646)*. Springer, 200–219. https://doi.org/10.1007/3-540-39185-1_12

[27] Pierre Letouzey. 2004. *Programmation fonctionnelle certifiée : L'extraction de programmes dans l'assistant Coq. (Certified functional programming : Program extraction within Coq proof assistant).* Ph. D. Dissertation. University of Paris-Sud, Orsay, France.

[28] Lorenz Leutgeb. 2018. Towards Verified Compilation of CakeML to WebAssembly. presented at ViennaJS.

[29] Wolfgang Meier, Jean Pichon-Pharabod, and Bas Spitters. 2024. CertiCoq-Wasm: Verified compilation from Coq to WebAssembly. presented at CoqPL'24. https://popl24.sigplan.org/details/CoqPL-2024-papers/3/CertiCoq-Wasm-Verified-compilation-from-Coq-to-WebAssembly

[30] Mozilla. 2024. Mozilla Developper Network: WebAssembly. https://developer.mozilla.org/en-US/docs/WebAssembly

[31] Eric Mullen, Stuart Pernsteiner, James R Wilcox, Zachary Tatlock, and Dan Grossman. 2018. Œuf: minimizing the Coq extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 172–185. https://doi.org/10.1145/3167089

[32] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 166–178. https://doi.org/10.1145/2784731.2784764

[33] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media. https://doi.org/10.1007/3-540-45949-9

[34] Zoe Paraskevopoulou. 2020. *Verified Optimizations for Functional Languages*. Ph. D. Dissertation. Princeton University.

[35] 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

[36] Christine Paulin-Mohring. 1989. Extracting Fω's Programs from Proofs in the Calculus of Constructions. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 89–104. https://doi.org/10.1145/75277.75285

[37] Christine Paulin-Mohring. 1989. *Extraction de programmes dans le Calcul des Constructions. (Program Extraction in the Calculus of Constructions).* Ph. D. Dissertation. Paris Diderot University, France. https://tel.archives-ouvertes.fr/tel-00431825

[38] Christine Paulin-Mohring and Benjamin Werner. 1993. Synthesis of ML Programs in the System Coq. *J. Symb. Comput.* 15, 5/6 (1993), 607–640. https://doi.org/10.1016/S0747-7171(06)80007-6

[39] Andreas Rossberg. 2019. *WebAssembly Core Specification W3C Recommendation*. Technical Report. W3C. https://www.w3.org/TR/wasm-core-1/

[40] Andreas Rossberg. 2023. *WebAssembly Core Specification*. Technical Report. W3C. https://webassembly.github.io/tail-call/core/

[41] Olivier Savary Bélanger. 2019. *Verified Extraction for Coq*. Ph. D. Dissertation. Princeton University.

[42] 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

[43] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2020. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 8:1–8:28. https://doi.org/10.1145/3371076

[44] Gordon Stewart, Lennart Beringer, and Andrew W. Appel. 2012. Verified heap theorem prover by paramodulation. (2012), 3–14. https://doi.org/10.1145/2364527.2364531

[45] Terru Stübinger and Lars Hupel. 2023. Extending Isabelle/HOL's Code Generator with support for the Go programming language. *CoRR* abs/2310.02704 (2023). https://doi.org/10.48550/ARXIV.2310.02704 arXiv:2310.02704

[46] The Coq Development Team. 2023. *The Coq Proof Assistant*. https://doi.org/10.5281/zenodo.8161141

[47] Jerome Vouillon and contributors. 2010. Wasm_of_ocaml: a fork of Js_of_ocaml which compiles OCaml bytecode to WebAssembly. https://github.com/ocaml-wasm/wasm_of_ocaml

[48] Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. 2019. Certifying graph-manipulating C programs via localizations within data structures. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30. https://doi.org/10.1145/3360597

[49] 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

[50] Ashley Williams and contributors. 2018. Wasm-pack: your favourite rust to wasm workflow tool. https://github.com/rustwasm/wasm-pack

[51] Alon Zakai. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 301–312. https://doi.org/10.1145/2048147.2048224

Received 2024-09-16; accepted 2024-11-19