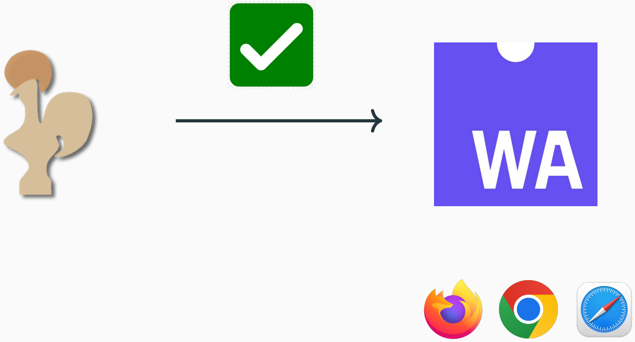
CertiCoq-Wasm: A Verified WebAssembly Backend for CertiCoq

Wolfgang Meier, Martin Jensen, Jean Pichon-Pharabod, Bas Spitters

January 21, 2025

Aarhus University

Project in one slide



Demo Application

Web application

Computing the SHA-256 sum on the web.

- SHA-256 implemented and verified in Coq, extracted with CertiCoq-Wasm
- Wasm binary integrated with JavaScript

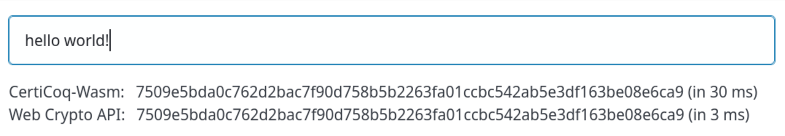


Figure 1: Try it at womeier.de/certicoqwasm-demo.html

Plan

1. WebAssembly
2. CertiCoq and $\lambda_{\texttt{ANF}}$
3. Wasm backend for CertiCoq
4. Evaluation
5. Application

Plan

1. **WebAssembly**
2. CertiCoq and $\lambda_{\texttt{ANF}}$
3. Wasm backend for CertiCoq
4. Evaluation
5. Application

WebAssembly

- Supported by every major browser...
  - ⇝ Brings Rust/C/... to the web
- and standalone WebAssembly runtimes
  - ⇝ Web3 & blockchains
  - ⇝ Edge computing
  - ⇝ Many other applications

- Secure sandbox with simple, clear semantics
- Almost native™ performance
- WebAssembly formalised in WasmCert-Isabelle and WasmCert-Coq

WebAssembly: Example

Example module

```
(module
  (global (mut i32) (i32.const 41))

  (func (export "add1")
    i32.const 1
    global.get 0
    i32.add
    global.set 0
  )
)
```

WebAssembly: Operational semantics

Host state:
- Stays the same, omitted

Frame:
- Locals
- ...

$$(hs, s, f, instr) \quad \hookrightarrow \quad (hs', s', f', instr')$$

Store:
- Globals
- Memory
- ...

List of instructions:
- Stack represented by leading `const` instr.

WebAssembly: Example execution

$(s, f, [\text{i32.const } 1; \underline{\text{global.get } 0}; \text{i32.add}; \text{global.set } 0])$

$\downarrow$ r_global_get, r_label (reduction in eval. context)

$(s, f, [\underline{\text{i32.const } 1; \text{i32.const } 41; \text{i32.add}}; \text{global.set } 0])$

$\downarrow$ rs_binop, r_label

$(s, f, [\underline{\text{i32.const } 42; \text{global.set } 0}])$

$\downarrow$ r_global_set

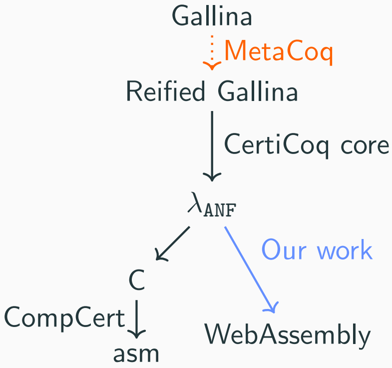
$(s', f, [])$

$s.\text{globals}_0$: 41 | $s'.\text{globals}_0$: 42

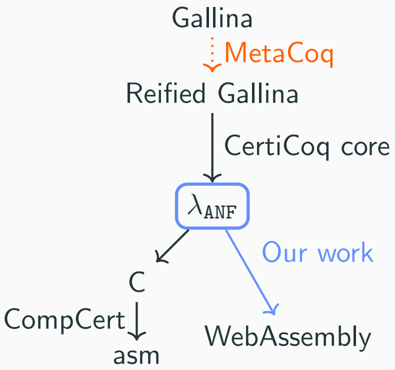
Plan

1. WebAssembly
2. **CertiCoq and** $\lambda_{\texttt{ANF}}$
3. Wasm backend for CertiCoq
4. Evaluation
5. Application

CertiCoq: A verified compiler for Coq's Gallina language



CertiCoq: A verified compiler for Coq's Gallina language



The intermediate language $\lambda_{\texttt{ANF}}$

(Variable)	x, y, f	$\in \texttt{Var}$		
(Constructor)	$\texttt{C}$	$\in \texttt{Constr}$		
(Function def)	fd	$::=$	$(f(\overline{y}) = e)$	
(Primitive val)	p	$\in \texttt{PrimVal}$		
(Primitive op)	op	$\in \texttt{PrimOp}$		
(Expression)	e	$::=$	$\text{let } x = \texttt{C}(\bar{y}) \text{ in } e$	Constructor
		$\mid$	$\text{let } x = y.i \text{ in } e$	Projection
		$\mid$	$\text{case } y \text{ of } [C_i \to e_i]_{i \in I}$	Case
		$\mid$	$\text{let } \overline{fd} \text{ in } e$	Function defs
		$\mid$	$\text{let } x = f\ \overline{y} \text{ in } e$	Function call
		$\mid$	$f\ \overline{y}$	Tail call
		$\mid$	$\text{let } x = p \text{ in } e$	Primitive value
		$\mid$	$\text{let } x = op\ \overline{y} \text{ in } e$	Primitive operation
		$\mid$	$\text{halt } y$	Function return
(Value)	v	$::=$	$(C, \overline{v}) \mid \left(\rho, \overline{fd}, f\right) \mid p$	
(Environment)	ρ	$::=$	$\cdot \mid \rho, x \mapsto v$	

The intermediate language $\lambda_{\texttt{ANF}}$

$\lambda_{\texttt{ANF}}$ program for "length [0]"

```
let [
  fun length_known_101(l_103) =
    case l_103 of {
    | nil =>
      let y_104 = O() in
      halt y_104
    | cons =>
      let l'_105 = l_103.1 in
      let y_106 = app length_known_101(l'_105) in
      let y_107 = S(y_106) in
      halt y_107
    }
] in

let y_108 = O() in
let y_109 = nil() in
let y_110 = cons(y_108,y_109) in
let foo.foo_112 = app length_known_101(y_110) in
halt foo.foo_112
```

The intermediate language $\lambda_{\texttt{ANF}}$

$\lambda_{\texttt{ANF}}$ program for "length [0]"

```
let [
  fun length_known_101(l_103) =
    case l_103 of {
    | nil =>
      let y_104 = O() in
      halt y_104
    | cons =>
      let l'_105 = l_103.1 in
      let y_106 = app length_known_101(l'_105) in
      let y_107 = S(y_106) in
      halt y_107
    }
] in

let y_108 = O() in
let y_109 = nil() in
let y_110 = cons(y_108,y_109) in
let foo.foo_112 = app length_known_101(y_110) in
halt foo.foo_112
```

function definitions

main expression

Plan

1. WebAssembly
2. CertiCoq and $\lambda_{\texttt{ANF}}$
3. **Wasm backend for CertiCoq**
4. Evaluation
5. Application

Wasm backend: Code generation

- Function *compile* takes a $\lambda_{\texttt{ANF}}$ expression[1] and produces a WebAssembly module with:
  - Wasm function for every $\lambda_{\texttt{ANF}}$ function
  - Main function containing the translation of the main expression
  - Linear memory
  - Globals: `result`, `out_of_mem`, ...
- Function *codegen* translates a $\lambda_{\texttt{ANF}}$ expression[2] into a list of Wasm instructions $\rightsquigarrow$ function body
- Simple memory model without garbage collection
- Representation of $\lambda_{\texttt{ANF}}$ values similar to CertiCoq's C backend

[1] with top-level function definitions

[2] without function definitions

Wasm backend: Code generation

Value relation $v \simeq^{\mathrm{val}}_{sr} v'$:

- v' is a i32
- Function value: index of corresponding function
- 63-bit integer value: pointer to linear memory holding i64
- Non-nullary constructor: pointer to linear memory holding ordinal followed by arguments
- Nullary constructor C: the value $2 \cdot C + 1$

Wasm backend: Correctness

Theorem 1. Correctness of lowering

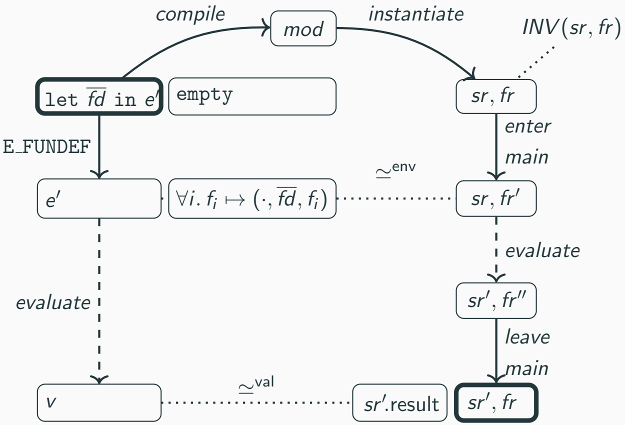
$$\begin{pmatrix} e = \texttt{let}\ \overline{fd}\ \texttt{in}\ e' \ \wedge \\ \cdot \vdash e \Downarrow v \ \wedge \\ compile\ e = (mod, \ldots) \end{pmatrix} \implies$$

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

Proof.

Forward simulation over the evaluation derivation. □

Wasm backend: Correctness



Plan

1. WebAssembly
2. CertiCoq and $\lambda_{\texttt{ANF}}$
3. Wasm backend for CertiCoq
4. **Evaluation**
5. Application

Evaluation

Benchmarks

- CertiCoq benchmark suite
  - demo1: appending two lists of bools
  - vs_easy, vs_hard: Veristar, decision procedure for decidable subset of separation logic
  - binom: merge two binomial queues, find maximum
  - color: graph coloring of a sizable graph
  - sha_fast: sha256 sum of a string of length 620
- ack_3_9: Ackermann function
- coqprime: Validating primality certificate of 100-digit prime

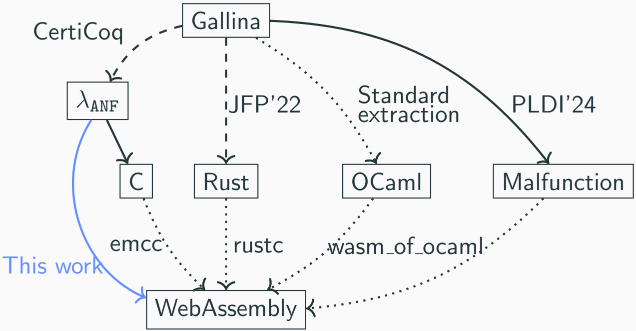


Figure 2: Extraction from Coq to WebAssembly.
Solid means verified, dotted means unverified.
Dashed means partly verified, or work in progress.

Evaluation: Other extractions

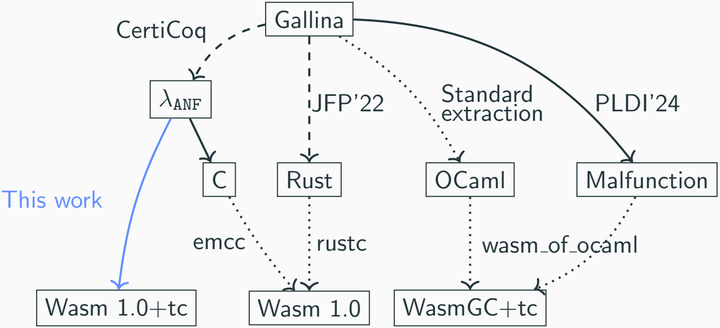


Figure 3: Extraction from Coq to WebAssembly.
Solid means verified, dotted means unverified.
Dashed means partly verified, or work in progress.

Evaluation: Other extractions

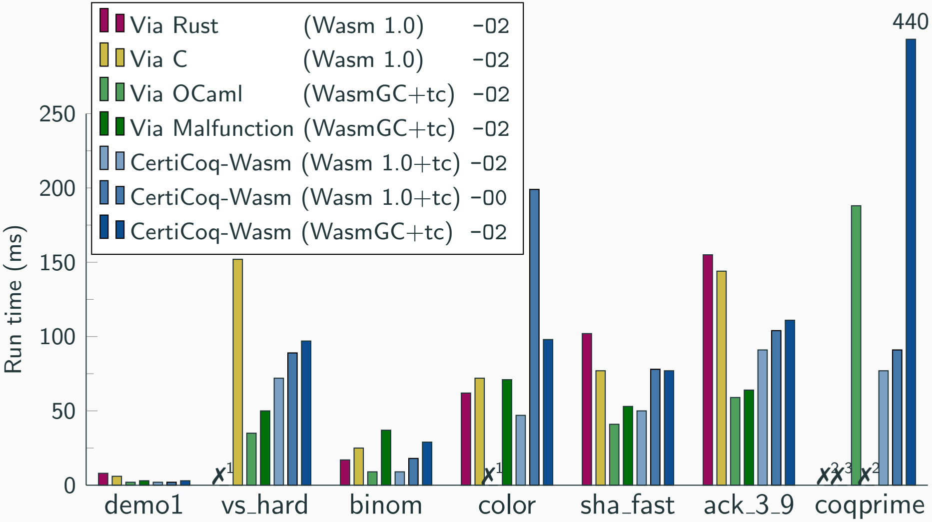


Figure 4: Run time using Node.js 22.1.0, different extractions to Wasm.

Evaluation: Binary size

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

Table 1: Binary size for Wasm binaries generated by different extraction mechanisms, in KB. All binaries are optimised with `wasm-opt -O2`.

Plan

1. WebAssembly
2. CertiCoq and $\lambda_{\texttt{ANF}}$
3. Wasm backend for CertiCoq
4. Evaluation
5. **Application**

Application I

Web application

Computing the SHA-256 sum on the web.

- SHA-256 implemented and verified in Coq, extracted with CertiCoq-Wasm
- Integrate Wasm binaries with JavaScript

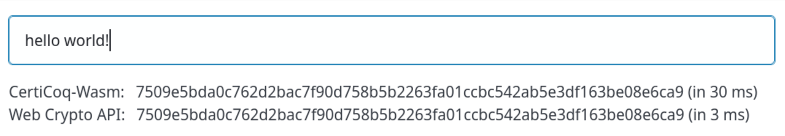


Figure 5: Try it at womeier.de/certicoqwasm-demo.html

Application II

Blockchain application

Extracting ConCert smart contract to the Concordium blockchain using CertiCoq-Wasm.

Concordium blockchain

- Uses WebAssembly 1.0 for smart contracts

ConCert

- Framework for developing certified smart contracts
- Counter contract from test suite
- Extraction to real blockchains, including Concordium via Rust

Blockchain Demo: Extracting a smart contract to Concordium

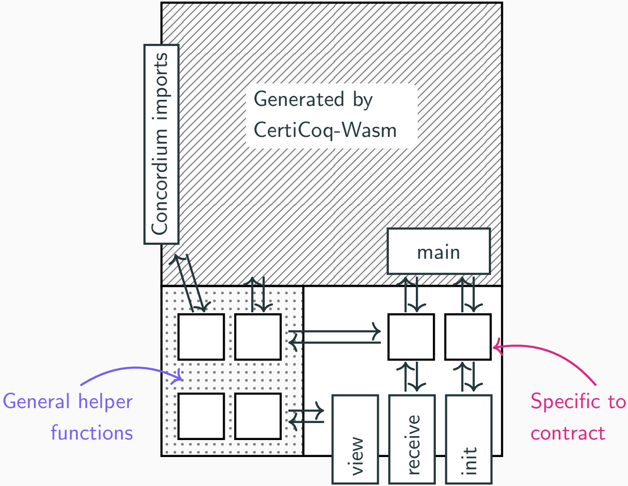


Figure 6: Wasm module for Concordium blockchain, counter contract. Hatched background means verified.

Conclusion

- Verified Wasm backend for CertiCoq
- TCB: Coq, CertiCoq-Wasm, WasmCert-Coq, Wasm runtime
- Practical: Performance & binary size
- Verified 63-bit primitive integer operations
  - $\rightsquigarrow$ Proof of False via bug in Coq's vm_compute
- Future work: Optimisations on Wasm (verified wasm-opt)

Questions?

`https://github.com/womeier/certicoqwasm`

Backup slides

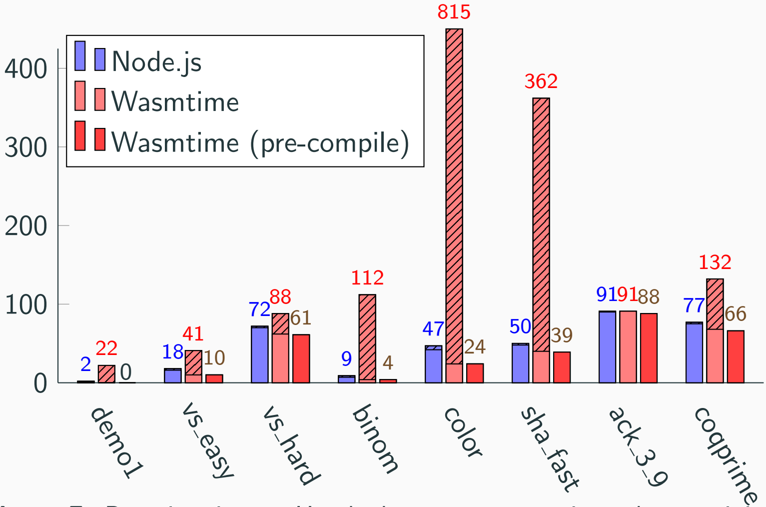


Figure 7: Run time in ms. Hatched means startup time, the remaining the run time of main. All optimised with `wasm-opt -O2`. Pre-compiled Wasmtime 38% better than Node.js.

Evaluation: Linear memory usage

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: Linear memory usage[3] of Wasm binaries generated by CertiCoq-Wasm (-O2), in KB.

[3]Wasm's linear memory can grow to at most 2GB.

Evaluation: Miscellaneous

File	LoC	Comment
LambdaANF_to_Wasm.v	470	Code generation
LambdaANF_to_Wasm_correct.v	5509	Generalised correctness theorem
LambdaANF_to_Wasm_primitives.v	435	Code generation for primitives
LambdaANF_to_Wasm_primitives_correct.v	4935	Correctness of primitives
LambdaANF_to_Wasm_instantiation.v	2261	Instantiation
LambdaANF_to_Wasm_restrictions.v	293	Size restrictions on input expression
LambdaANF_to_Wasm_utils.v	3232	General helper lemmas
toplevel_theorems.v	236	Correctness statement
total = 17371		

Table 3: LoC obtained by *cloc* excluding comments and blank lines.

Wasm backend: Correctness

Theorem 2. Generalised correctness of lowering

For any well-formed, size-restricted $\lambda_{\texttt{ANF}}$ expression e with globally unique bound variables, a global set of function definitions $\overline{fd}$, a block context $\texttt{B}^k$, and well-formed environments,

$$\begin{pmatrix} \rho \vdash e \Downarrow v \ \land \ \mathit{codegen}\ e = e' \ \land \\ \mathit{INV}(sr, fr) \ \land \ \rho \simeq^{\mathsf{env}}_{fd,e} (sr, fr) \land \\ \text{translated } \overline{fd} \text{ in } sr.\text{funcs} \\ (\forall x.\, x \text{ let-bound in } e \Rightarrow x \notin \rho) \end{pmatrix} \Longrightarrow$$

$$\exists sr', fr', \texttt{C}^{k'}.\ \begin{pmatrix} (sr, \ldots, \texttt{frame}_0\ \{fr\}\ \texttt{B}^k[e']\ \texttt{end}) \hookrightarrow^* \\ (sr', \ldots, \texttt{frame}_0\ \{fr'\}\ \texttt{C}^{k'}[\texttt{return}]\ \texttt{end}) \ \land \\ \\ \begin{pmatrix} (v \simeq^{\text{val}}_{sr'} sr'.\text{globals}_{\text{result}} \land \mathit{INV}(sr', fr')) \ \lor \\ sr'.\text{globals}_{\text{out\_of\_mem}} = 1 \end{pmatrix} \land \\ (\forall v, v'.\, v \simeq^{\text{val}}_{sr} v' \Rightarrow v \simeq^{\text{val}}_{sr'} v') \end{pmatrix}$$

Proof. By induction on the evaluation derivation. □

Wasm backend: Correctness

Theorem 3: Instantiation

For any $\lambda_{\texttt{ANF}}$ expression e with globally unique bound variables, and a well-formed constructor environment,

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

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

Proof. The module is well-formed by construction. The invariants hold initially and functions are present in the store. □

How to use

Coq file test.v

```
From CertiCoq.Plugin Require Import CertiCoq.
(...)
Definition foo := map odd [1; 2; 3].

(* CertiCoq Compile -file "foo" foo. generates C *)
CertiCoq Compile Wasm -file "foo" foo.
```

Compile & Run Wasm module in Node.js

```
$ coqc test.v
$ node run-node.js . foo
==> (cons true (cons false (cons true nil)))
```