CertiCoq-Wasm: Verified compilation from Coq to WebAssembly

Wolfgang Meier, Jean Pichon-Pharabod, Bas Spitters

January 20, 2024

Aarhus University

Project in one slide

~~Demo~~

Coq file

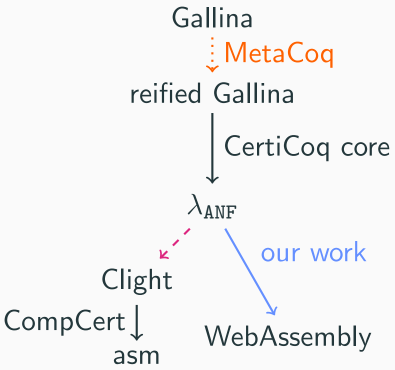
```
From CertiCoq.Plugin Require Import CertiCoq.
(...)
Definition foo := map odd [1; 2; 3].
CertiCoq Generate WASM -cps -file "foo" foo.
```

Compile & Run Wasm file in Node.js

```
$ coqc test.v
$ wasm2wat foo.wasm > foo.wat
$ ./insert_tailcalls.py --path_in foo.wat
                        --path_out foo-tail.wat
$ wat2wasm --enable-tail-call foo-tail.wat -o foo.wasm
$ node --experimental-wasm-return_call foo.js

==> (cons true (cons false (cons true nil)))
```

The CertiCoq pipeline



Plan

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

Plan

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

WebAssembly

- Almost native performance
- Secure sandbox *with simple, clear semantics*
- Supported by every major browser...
  - ⇝ Brings Rust/C/... to the web
  - ⇝ Web3 & blockchains
- and standalone runtimes

- WebAssembly 1.0 completely formalised
  in WasmCert-Isabelle and WasmCert-Coq

WebAssembly: Example

```
(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:
- may change when calling functions provided by the host

Frame:
- local vars
- runtime repr. of module

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

Store:
- global vars
- memory

...

List of instructions
- stack represented by leading `const` instr.

WebAssembly: Example execution

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

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

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

$\big\downarrow$ rs_binop, r_label

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

$\big\downarrow$ r_set_global

$(hs, s', f, [])$

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

Plan

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

CertiCoq

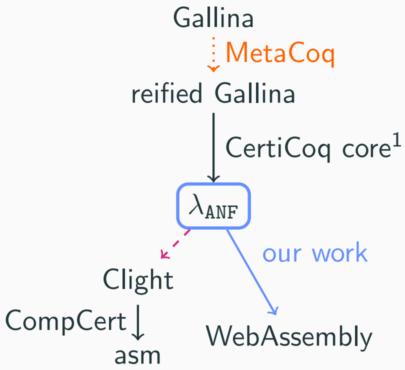


Figure 1: The CertiCoq pipeline. Proofs in progress are in dashed magenta, and MetaCoq (which has to be trusted) is in dotted orange. Our verified contribution is in blue.

[1] includes optimisations on $\lambda_{\texttt{ANF}}$

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

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

Figure 2: Syntax of CertiCoq's $\lambda_{\texttt{ANF}}$ intermediate language, primitive operations omitted

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

Restrictions on $\lambda_{\texttt{ANF}}$

- Variables globally unique
- No primitives
- Expression in CPS
- Function definitions at the top (lambda lifting, hoisting)
- Size (everything fits in `i32` vars)

Restrictions are either...

- enforced by check: refuse to compile otherwise
- or ensured by previous pipeline stages

Plan

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

Wasm backend: General

- Inspired by CertiCoq's C backend
- Function *codegen* takes a $\lambda_{\texttt{ANF}}$ expr (without function defs.) and produces a list of Wasm instructions
  - let-bound vars are mapped to (indices of) local vars
- Function *compile* takes a $\lambda_{\texttt{ANF}}$ expr and produces a Wasm module with:
  - a main function containing the translation of e
  - a Wasm function for every $\lambda_{\texttt{ANF}}$ function
  - a function that given an `i32`, prints the corresponding S-expr.
  - global vars: `result`, `out_of_mem`, `memory_pointer` (points to next free memory, no GC)

    ...
- Simple representation of $\lambda_{\texttt{ANF}}$ constructor values in Wasm linear memory: tag followed by arguments

Wasm backend: Codegen function

```
Fixpoint translate_exp (...) (e : exp)
  : error (list basic_instruction) :=
  match e with
  | Eproj x tg n y e' =>
      following_instr <- translate_exp (...) e';;
      y_var <- trans_var nenv lenv y;;
      x_var <- trans_var nenv lenv x;;

      Ret ([ BI_get_local y_var
             (* skip ctor_id and previous constr arguments *)
           ; BI_const (nat_to_value (((N.to_nat n) + 1) * 4))
           ; BI_binop T_i32 (Binop_i BOI_add)
           ; BI_load T_i32 None 2%N 0%N
           ; BI_set_local x_var
           ] ++ following_instr)

  | Ehalt x =>
     x_var <- translate_var nenv lenv x;;
     Ret [ BI_get_local x_var; BI_set_global result_var ]

  | ...
  end.
```

Correctness of Wasm backend

Theorem 1. Correctness of lowering

$$\left(\begin{array}{l} \cdot \vdash e \Downarrow v \wedge \mathit{compile}\ e = (\mathit{mod}, \ldots) \wedge \\ \mathit{instantiate}\ \mathit{mod} = (\mathit{sr}, \ldots) \end{array}\right) \Longrightarrow$$

$$\exists \mathit{sr}'.\ \begin{array}{l} (\mathit{hs}, \mathit{sr}, \mathit{fr}, [\text{call}\ \mathit{idx}_{\mathrm{main}}]) \to^{*} (\mathit{hs}, \mathit{sr}', \mathit{fr}, []) \wedge \\ \left(v \simeq^{\mathrm{val}}_{\mathit{sr}'} \mathit{sr}'.\text{globals}_{\mathrm{res}} \vee \mathit{sr}'.\text{globals}_{\mathrm{out\_of\_mem}} = 1\right) \end{array}$$

Proof. By Theorem 2, helper lemmas.

- Variables in e unique
- Size of e restricted, everything has to fit in `i32` vars
- Function definitions only at top-level
- Some technical details omitted

Correctness of Wasm backend

Value relation

(VR_FUN)

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

(VR_CONSTR)

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

Figure 3: Value relation, relating a $\lambda_{\mathtt{ANF}}$ value to a Wasm i32

Correctness of Wasm backend

Theorem 2. Generalised correctness of lowering

$$\left(\begin{array}{l}\rho \vdash e \Downarrow v \wedge \mathit{INV}(sr, fr) \wedge \\ \mathit{codegen}\ e = e' \wedge \rho \simeq^{\mathrm{env}}_{e} (sr, fr)\end{array}\right) \Longrightarrow$$

$$\exists sr'\, fr'. \quad \begin{array}{l}(hs, sr, fr, e') \to^{*} \big(hs, sr', fr', []\big) \wedge \\ \Big(v \simeq^{\mathrm{val}}_{sr'} sr'.\mathsf{globals}_{\mathrm{res}} \vee sr'.\mathsf{globals}_{\mathrm{out\_of\_mem}} = 1\Big)\end{array}$$

Proof. By induction on the evaluation derivation.

Environment relation $\rho \simeq^{\mathrm{env}}_{e} (sr, fr)$:

- for function values: provides related indices into sr.funcs
- for variables x free in e (the results of previous computations): provides a local var containing an `i32` related to the value v, which is provided by ρ for x.

Plan

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

Evaluation

Lines of code

Backend	Correctness proof
ca. 450	ca. 6200

Benchmarks

- Benchmarks from CertiCoq[2]:
  - binom: merge two binomial queues, find maximum
  - sha_fast: sha256 sum of a string of length 620
  - vs_easy, vs_hard: Veristar, decision procedure for decidable subset of separation logic

[2] Graph coloring benchmark didn't work, `wat2wasm` crashed due to file size

Evaluation

		vs_easy	vs_hard	sha_fast	binom
C	(gcc -O2)	7 ms	44 ms	35 ms	6 ms
C	(CompCert -O2)	10 ms	59 ms	43 ms	16 ms
Wasm	(Node.js)	81 ms	700 ms	190 ms	13 ms

Table 1: Runtime performance[3], all in CPS, average of 10 runs

	vs_easy	vs_hard	sha_fast	binom
Wasm backend	20.8 MB	142.7 MB	74.8 MB	504 KB

Table 2: Usage of linear memory[4], not garbage collected

[3] on Intel i5-8250U, with Node.js 18.19.0, gcc 12.2.0, CompCert 3.13
[4] Wasm linear memory is limited to 4GB

Plan

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

Limitations

Ideas for improvement

- Proof of instantiation of generated Wasm module
- Add garbage collection
  - Link with verified GC (like CertiCoq's C-backend)
  - Or use WebAssembly's native GC[5]
- Support for primitives
- Add tail calls to WasmCert[6]
- Performance improvements
- Support compiling $\lambda_{\texttt{ANF}}$ expressions with normal calls

[5] Just shipped in Nov. 2023 in Chrome and Firefox, not yet in WasmCert

[6] Currently, $\lambda_{\texttt{ANF}}$ tail calls are translated to normal Wasm calls, replaced with tail call instructions in binary by script

Conclusion

- Verified Wasm backend for CertiCoq
- Code at: `github.com/womeier/certicoqwasm`
- Future plans: GC, improve performance