Progressful Interpreters for Efficient WebAssembly Mechanisation

XIAOJIA RAO, Imperial College London, UK
STEFAN RADZIUK, Imperial College London, UK
CONRAD WATT, Nanyang Technological University, Singapore
PHILIPPA GARDNER, Imperial College London, UK

Mechanisations of programming language specifications are now increasingly common, providing machine-checked modelling of the specification and verification of desired properties such as type safety. However it is challenging to maintain these mechanisations, particularly in the face of an evolving specification. Existing mechanisations of the W3C WebAssembly (Wasm) standard have so far been able to keep pace as the standard evolves, helped enormously by the W3C Wasm standard's choice to state the language's semantics in terms of a fully formal specification. However a substantial incoming extension to Wasm, the 2.0 feature set, motivates the investigation of strategies for more efficient production of the core verification artefacts currently associated with the WasmCert-Coq mechanisation of Wasm.

In the classic formalisation of a typed operational semantics as followed by the W3C Wasm standard, both the type system and runtime operational semantics are defined as inductive relations, with associated type soundness properties (progress and preservation) and an independent sound interpreter. We investigate two more efficient strategies for producing these artefacts, which are currently all separately defined by WasmCert-Coq. First, the approach of Kokke, Siek, and Wadler for deriving a sound interpreter from a constructive progress proof — we show that this approach scales to the W3C Wasm 1.0 standard, but results in an inefficient interpreter in our setting. Second, inspired by results from intrinsically-typed languages, we define a *progressful* interpreter which uses Coq's dependent types to certify not only its own soundness, but also the progress property. We show that this interpreter can implement several performance optimisations while maintaining these certifications, which are fully erasable when the interpreter is extracted from Coq. Using this approach, we extend the WasmCert-Coq mechanisation to the significantly larger Wasm 2.0 feature set, discovering and correcting several errors in the expanded specification's type system.

CCS Concepts: • **Theory of computation → Logic and verification**; *Operational semantics*; • **Software and its engineering → Formal language definitions**; • **Security and privacy → Logic and verification**.

Additional Key Words and Phrases: Verified Computation, Proof Assistants, Type Soundness, Dependent Types, WebAssembly

ACM Reference Format:
Xiaojia Rao, Stefan Radziuk, Conrad Watt, and Philippa Gardner. 2025. Progressful Interpreters for Efficient WebAssembly Mechanisation. *Proc. ACM Program. Lang.* 9, POPL, Article 22 (January 2025), 29 pages. https://doi.org/10.1145/3704858

Authors' Contact Information: Xiaojia Rao, xiaojia.rao19@imperial.ac.uk, Imperial College London, UK; Stefan Radziuk, stefan@radzi.uk, Imperial College London, UK; Conrad Watt, Nanyang Technological University, Singapore, conrad.watt@ntu.edu.sg; Philippa Gardner, Imperial College London, UK, p.gardner@imperial.ac.uk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/1-ART22
https://doi.org/10.1145/3704858

1 Introduction

There is a growing body of work on the *mechanisation* of programming language specifications, such as C [7, 14, 26, 27, 32], JavaScript [8, 17, 35], Java [9, 19, 22], Standard ML [24, 25, 41], OCaml [33], Haskell [47], WebAssembly [44]. These mechanisations provide a machine checked model of the specification, and often verification of desired properties such as type safety. These efforts have been effective in providing assurance that the specifications are correctly defined, and satisfy their intended correctness properties, often revealing bugs and ambiguities.

However, mechanisation is an onerous process, to the point that many mechanised semantics cover only a fragment of their language's full feature set, especially if the language continues to evolve and expand with new versions. Mechanisations of the W3C WebAssembly (Wasm) standard [18] have so far been able to keep pace as the standard has evolved through various drafts to the 1.0 release [45], helped enormously by the standard's unusual decision to give the definition of WebAssembly entirely as a classic inductively defined formal semantics, complete with a statement of its type system's intended correctness properties. Several complete mechanisations of its 1.0 feature set have been developed, providing the artefacts listed above, while discovering and correcting errors in the language's type system [44, 45]. However the jump from Wasm 1.0 to the forthcoming Wasm 2.0 standard is particularly large, almost doubling the instruction set of the language. This work addresses the problem of producing Wasm's mechanisation artefacts in a sustainable and tractable way as the Wasm language expands, in order to ensure that Wasm's new features continue to enjoy the same verification guarantees as the 1.0 feature set.

In the classic formalisation of a typed operational semantics, as followed by the W3C Wasm standard, both the type system and runtime operational semantics are defined as inductive relations, and a statement is made of the desired *syntactic type soundness* properties (*progress* and *preservation*) which relate them [48]. This leads to a natural set of core artefacts which a mechanisation of such a semantics will aim to support:

- the mechanisation of the language definitions themselves;
- a proof of the type system's progress and preservation properties;
- a definition of a simple executable interpreter, since the inductive operational semantics is not directly executable;
- a proof that the interpreter is sound with respect to the operational semantics.

Each of these items must be separately defined and maintained — as they are in existing mechanisations of Wasm 1.0 [45].

For a small language related to System F, Kokke et al. [23] observed that a constructive proof of progress over a small-step semantics essentially embodies a sound one-step interpreter, since given a well-typed input configuration it represents a computational "recipe" for obtaining an output configuration that is allowed by the language's small-step reduction relation. By composing this step with a constructive proof of preservation and iterating, a simple verified interpreter is obtained "for free". Since Wasm mechanisations already commit to maintaining an analogous type soundness proof, the idea that a sound interpreter may be derived "for free" from this proof, without needing to maintain a separate definition (and verification of this definition), is enticing.

In this work, we push the techniques of Kokke et al. [23] to their limits by applying their approach to the WasmCert-Coq mechanisation of the W3C Wasm 1.0 standard, converting its type soundness proofs to be fully constructive, successfully deriving an interpreter from these proofs, and evaluating its performance. This approach allows us to eliminate WasmCert-Coq's separate interpreter definition and correctness proof. Going beyond Kokke et al. [23], we execute this interpreter end-to-end by composing it with WasmCert-Coq's verified type checker and parser,

identifying multiple ways in which innocuous choices in constructing the type soundness proofs can severely degrade its runtime performance.

We observe that one major limitation of this approach is that we lose direct control over the structure and internal representation of the interpreter, and therefore can only indirectly optimise its performance. Moreover, the proof tree of the input term's typing judgement needs to be explicitly represented in the interpreter's memory and manipulated at each step, since the interpreter is only allowed to take repeated steps by constructing a proof that each successor term is well-typed. Ideally we would like a solution which maintains our proof maintenance benefits, and does not require runtime manipulation of any typing proofs.

To accomplish this, we show in Coq that a formalised one-step interpreter over untyped Wasm terms can, with only modest effort, be augmented with a dependent type certifying the *contrapositive* of the progress property ("if the interpreter fails to step soundly according to the operational semantics, the input must be ill-typed"), which not only eliminates the need for a separate proof of progress, but also the need for a separate proof of the interpreter's soundness with respect to Wasm's operational semantics. We refer to such an interpreter as a *progressful* interpreter. The interwoven parts of the one-step interpreter which prove this contrapositive property can be erased during its extraction to executable code, since subsequent iterations of the interpreter do not require this property as input. This approach allows us more fine-grained control over the interpreter's internal state, which we exploit to implement several optimisations to Wasm's representation of evaluation contexts based on Watt et al. [46] while still significantly reducing the proof burden in comparison to the more traditional approach of separate type soundness and interpreter correctness proofs. We report on our experiences of extending WasmCert-Coq to Wasm 2.0, using this approach to save effort in updating what were previously separate definitions and correctness proofs.

Our approach is deeply resonant with previous work that focusses on the proof maintenance benefits that stem from defining the semantics of an intrinsically-typed language in terms of a dependently-typed definitional interpreter [5, 6, 38, 42]. We show that many of these benefits are still achievable in WasmCert-Coq, despite the fact that Wasm is not intrinsically typed, and our interpreter is not definitional — due to the design choices of the official formal specification.

All of our extensions to WasmCert-Coq are available publicly [43]. In summary, this work establishes the following takeaways.

- Kokke et al. [23]'s approach for deriving a sound interpreter from a constructive progress proof scales to the industrial W3C Wasm 1.0 standard, but optimising the performance of this interpreter is challenging. Moreover, the requirement that the derived interpreter traverses a proof of well-typedness of the input term at runtime represents an unavoidable performance penalty. (§2)
- Our *progressful* interpreter similarly allows multiple verification artefacts to be produced from a single definition, but gives us more control over performance. Our approach is related to formal properties most often discussed in the context of intrinsically-typed languages, but we realise them in Wasm, which is extrinsically typed. (§3, 4)
- We extend WasmCert-Coq [45] to the substantially larger Wasm 2.0 feature set, updating all related verification artefacts smoothly with the aid of our progressful interpreter. Through the process, several errors in the Wasm 2.0 specification have also been uncovered and reported to the specification editor, reinforcing the value of language mechanisations in improving industrial language standards. (§5)

2 Interpreter from Progress

We describe the method given by Kokke et al. [23] for constructing a sound interpreter from a proof of progress for a general language with a small-step, inductively-defined reduction semantics and type system. We apply this method to the W3C Wasm 1.0 standard, by adapting the mechanised progress proof in WasmCert-Coq [45], thus demonstrating that this method can scale to a full industrial language definition. While the automatic generation of this interpreter saves significant proof effort, we report on fundamental limitations on its performance that arise from the natural structure of the type soundness proofs, motivating our new approach in §3.

2.1 Type soundness

Consider a language with its execution specified by an inductively defined small-step operational semantics, given by the reduction relation $\mathrm{cfg} \hookrightarrow \mathrm{cfg}$ for configurations cfg and a typing relation $\vdash \mathrm{cfg} : t$ between configurations and configuration types.

The type soundness property is formulated using the traditional progress and preservation properties [48]:

Proposition 2.1 (Progress). *A typed configuration can either do a reduction or is a terminal configuration:*

$$\forall \mathrm{cfg}, t.\ \vdash \mathrm{cfg} : t \implies (\exists \mathrm{cfg}'.\mathrm{cfg} \hookrightarrow \mathrm{cfg}') \vee \mathsf{terminal}(\mathrm{cfg}) \tag{1}$$

Proposition 2.2 (Preservation). *The reduction relation preserves the typing relation:*

$$\forall \mathrm{cfg}, \mathrm{cfg}', t.\ (\vdash \mathrm{cfg} : t) \wedge (\mathrm{cfg} \hookrightarrow \mathrm{cfg}') \implies (\vdash \mathrm{cfg}' : t). \tag{2}$$

2.2 Interpreter from Progress

Kokke et al. [23] report that a constructive proof of the progress property (2.1) can be used as a one-step interpreter, also noting that this correspondance has appeared in scattered folklore. The progress proof takes a configuration cfg and a proof of the configuration's well-typedness as input, and concludes that either some cfg' must exist such that $\mathrm{cfg} \hookrightarrow \mathrm{cfg}'$ is allowed by the semantics, or cfg is terminal. In a constructive setting, the progress proof therefore computes some cfg' as a witness which, together with its conclusion that $\mathrm{cfg} \hookrightarrow \mathrm{cfg}'$ is allowed by the semantics, represents a sound one-step interpreter for the language.

One complication to using a constructively-proven progress theorem as a one-step interpreter is the requirement that a proof of well-typedness of the initial configuration must be additionally provided as input. Moreover to iterate the one-step interpreter, a constructive proof of preservation must be used to re-establish the well-typedness of intermediate configurations before each step.

To emphasize the computable nature of these constructive proofs, we present the shapes of the required statements in the WasmCert-Coq mechanisation in Figure 1, where we implemented the above method for Wasm. The progress function acts as described above, taking a configuration cfg, a configuration type t, and a proof term `HType` which associates cfg with the type t. It produces either a new configuration along with a proof term $\mathrm{cfg} \hookrightarrow \mathrm{cfg}'$ representing the soundness of the reduction step, or a proof that the configuration is terminal. Similarly, the preservation function takes the old and new configurations, a proof term representing that a sound reduction was carried out, and a proof term representing the well-typedness of the old configuration, producing a proof term representing the well-typedness of the new configuration. The verified type inference function, `infer_type`, takes a configuration as input and, if the type inference was successful, returns a configuration type along with a proof term representing the correctness of the returned type with respect to Wasm's type system.

```
Definition progress (cfg: config) (t: config_type) (HType: (|- cfg: t)):
    {cfg': config & (cfg --> cfg')} + (terminal cfg).

Definition preservation (cfg cfg': config) (t: config_type)
                        (HReduce: cfg --> cfg') (HType: (|- cfg: t)) :
    |- cfg' : t.

Definition infer_type (cfg: config) : option {t: config_type & (|- cfg : t)}.
```

Fig. 1. Definitions of progress, preservation, and type inference as computable functions

With the above definitions, we can describe the algorithm for creating a one-step and multi-step interpreter from the type soundness proofs, with the aid of the verified type inference function.

The interpreter begins with a type inference on the input cfg and returns an error if it fails. If typing succeeds, the interpreter alternates between applying the progress and preservation functions as follows:

- Apply the progress property to the configuration cfg and the associated proof term $\vdash \mathrm{cfg} : t$, obtaining either a new configuration cfg' and a proof term representing $\mathrm{cfg} \hookrightarrow \mathrm{cfg}'$, or a termination result;
- If applying progress does not result in termination, apply the preservation property to the old and new configurations, the reduction proof term, and the old configuration typing proof term to produce a proof term $\vdash \mathrm{cfg}' : t$ for the new configuration, allowing iteration to continue.

To concretely define the above interpreter in Coq, we added an additional argument as the *fuel* for the interpreter, ensuring it terminates when it runs out of fuel. This is a standard practice to satisfy Coq's termination check.

This method demonstrates that, given a constructive proof of type soundness and a verified type inference function, a sound interpreter can be directly extracted with minimal effort. This is particularly valuable for maintaining mechanized artefacts, as it eliminates the need to maintain a separate interpreter and its soundness proof, leading to a smaller overall codebase to maintain.

In Chapman et al. [10], this approach is used to generate an interpreter for a small core language based on System F. However the interpreter cannot be run end-to-end due to the lack of a verified type checker. In this work, we apply the above approach to the full definition of Wasm 1.0, reusing the existing progress and preservation proofs of WasmCert-Coq [45] (an existing Coq mechanisation of Wasm 1.0) with only minor changes to make them fully computable. In addition, the verified type checker and parser of WasmCert-Coq allows our interpreter to be executed end-to-end, once extracted via standard Coq mechanisms [28, 29] to OCaml. WasmCert-Coq originally included its own separately-verified interpreter in addition to its type soundness proofs, which we are therefore able to make redundant.

However, there are several downsides to using this method for a real-world language like Wasm. The primary concerns are the inefficient performance of the interpreter and the difficulty of implementing optimizations.

Inefficient performance. Because the interpreter relies on the constructive progress proof as a one-step evaluation function, the proof term representing the well-typedness of the current configuration *must be included in the runtime representation of the interpreter's state.* This leads to a severe performance penalty as the progress and preservation functions may need to traverse substantial portions of this proof term to produce the next configuration. Additionally, certain proof

constructions, which may seem innocuous, have surprising knock-on effects on the performance of the derived interpreter. In particular all of the above is true in Coq's setting *even after the interpreter is extracted to OCaml* — the proof term remains concretely represented in memory, and is traversed by the code of the extracted interpreter.

Difficulty in optimizations. It is challenging to directly control the computational behaviour of the interpreter extracted from type soundness, as any structural changes to the interpreter can only be indirectly effected by changing the underlying proof structure.

To explain the above in more detail, we introduce a small fragment of the concrete semantics and type system of Wasm and describe wide-ranging performance issues that arise from re-using the existing proof of type soundness from WasmCert-Coq [45].

$$\text{(value type) } t ::= \mathbf{i32} \mid \mathbf{i64} \mid \mathbf{f32} \mid \mathbf{f64} \qquad \text{(function type) } ft ::= t^* \to t^*$$

$$\text{(value) } v ::= t.\mathbf{const}\ c \qquad \text{(immediate) } i, n, min, max ::= nat$$

$$\text{(instruction) } e ::= v \mid t.\mathbf{add} \mid \mathbf{local}.\{\mathbf{get/set}\}\ i \mid \mathbf{label}_n\{e_{\text{cont}}\}\ e^*\ \mathbf{end} \mid \ldots$$

$$\text{(frame) } F ::= \{\text{local} :: v^*, \text{inst} :: \ldots\}$$

$$\text{(evaluation context)} \mathcal{E}[_] ::= [_] \mid v^* +\!\!+ \mathcal{E}[_] +\!\!+ e^* \mid \mathbf{label}_n\{e^*\}\ (\mathcal{E}[_])\ \mathbf{end}$$

$$\text{(configuration tuple) cfg} ::= \{\text{store} :: S, \text{frame} :: F, \text{instructions} :: e^*\}$$

Fig. 2. Selected abstract syntax of Wasm 1.0

2.3 Wasm 1.0 Semantics

We begin with a brief introduction to the abstract syntax and runtime representation of Wasm.

Wasm is a stack-based language with execution specified by a small-step operational semantics on configuration tuples cfg of the form $(S; F; e^*)$[1]. Here, S is the *store*, which contains all the global states created during execution. F is the *frame*, which holds the local variables and a record *inst* that tracks the components of the store S accessible from the current function. The list e^* consists of instructions representing the combined instruction value stack, where the value stack is represented as a leading list of **const** instructions in the instruction stack.

We present a selected subset of the abstract syntax of Wasm 1.0 in Fig 2, eliding the details of some parts by leaving them in gray.

For the purpose of type soundness, the terminal configurations of Wasm are defined as configurations $(S; F; e^*)$ where e^* is a list consisting entirely of values, at which points the values are returned as the result of the Wasm program.

Execution takes place at the top of the instruction stack, consuming an appropriate number of values from the value stack as arguments and pushing some values back onto it. For example, the reduction rule for the numeric addition instruction is

$$(S; F; [t.\mathbf{const}\ c_1; t.\mathbf{const}\ c_2; t.\mathbf{add}]) \hookrightarrow (S; F; [t.\mathbf{const}\ (c_1 + c_2)]) \tag{3}$$

This rule describes that upon consuming two values of type t from the value stack, the instruction $t.\mathbf{add}$ executes by producing a value representing their sum and pushing it back onto the value stack.

[1]We follow the conventions used by the Wasm standard, where X^* represents a list of Xs, and X^n represents a list of Xs of length n.

Figure 3 includes a selected set of reduction rules from the Wasm 1.0 operational semantics[2].

$$\frac{}{[t.\mathbf{const}\ c_1; t.\mathbf{const}\ c_2; t.\mathbf{add}] \hookrightarrow [t.\mathbf{const}\ (c_1 + c_2)]}\ \text{add} \qquad \frac{F.\mathsf{local}[k] = v}{(S; F; [\mathbf{local.get}\ k]) \hookrightarrow (S; F; [v])}\ \text{local.get}$$

$$\frac{(S; F; e^*) \hookrightarrow (S'; F'; e'^*)}{(S; F; \mathcal{E}[e^*]) \hookrightarrow (S'; F'; \mathcal{E}[e'^*])}\ \text{context}$$

Fig. 3. Selected Reduction Rules of Wasm 1.0

Wasm's type system is based on the typing relation for code fragments in the shape of $C \vdash e^* : ft$, assigning the instruction list e^* with a function type ft that describes the effect of executing e^* on the value stack. For example, the typing rule for the numeric addition instruction is

$$\frac{}{C \vdash t.\mathbf{add} : [t; t] \rightarrow [t]}\ \text{add} \tag{4}$$

which specifies that the instruction $t.\mathbf{add}$ consumes two values of type t from the value stack and pushes one value of type t back to the value stack as a result.

The typing context C contains information regarding the types of global states in the scope of the current function execution, the types of local variables of the current function, and the types of the runtime control flow targets that are currently in scope. For example, the typing rule for **local.get** k instruction, which fetches the value of the kth local variable in the current function frame F and pushes it to the stack, requires access to the corresponding field in the context:

$$\frac{C.\mathsf{local}[k] = t}{C \vdash \mathbf{local.get}\ k : [] \rightarrow [t]}\ \text{local.get} \tag{5}$$

Fig 4 includes the full shape of the typing context (details of individual fields omitted) and a selected set of typing rules from the Wasm 1.0 type system.

$$\text{(typing context) } C \ ::= \begin{cases} \mathsf{type} :: ft^*, \mathsf{func} :: ft^*,\ \mathsf{table} :: tt^*,\ \mathsf{memory} :: mt^*,\ \mathsf{global} :: gt^*, \\ \mathsf{local} :: t^*,\ \mathsf{label} :: (t^*)^*,\ \mathsf{return} :: (t^*)^? \end{cases}$$

$$\frac{}{C \vdash [] : [] \rightarrow []}\ \text{empty} \qquad \frac{}{C \vdash [t.\mathbf{const}\ c] : [] \rightarrow [t]}\ \text{const}$$

$$\frac{}{C \vdash [t.\mathbf{add}] : [t; t] \rightarrow [t]}\ \text{add} \qquad \frac{C.\mathsf{local}[k] = t}{C \vdash [\mathbf{local.get}\ k] : [] \rightarrow [t]}\ \text{local.get}$$

$$\frac{C \vdash e^* : t_1^* \rightarrow t_3^* \quad C \vdash [e] : t_3^* \rightarrow t_2^*}{C \vdash e^* +\!\!+ [e] : t_1^* \rightarrow t_2^*}\ \text{composition}$$

$$\frac{C \vdash e^* : t_1^* \rightarrow t_2^*}{C \vdash e^* : t_3^* +\!\!+ t_1^* \rightarrow t_3^* +\!\!+ t_2^*}\ \text{subsumption}$$

Fig. 4. Selected Typing Rules of WebAssembly 1.0

[2]The Wasm standard uses a convention that omits the store S and frame F from the configuration tuples in the reduction rules where they are irrelevant; we adopt this convention in the figure.

Having defined the above typing rules for code fragments, WebAssembly defines its configuration typing relation by

$$\frac{\vdash_s S : \mathrm{ok} \quad S \vdash_f F : C \quad S; C \vdash e^* : [] \to t^*}{\vdash_{\mathrm{cfg}} (S; F; e^*) : t^*} \text{ config} \tag{6}$$

The above typing rule uses a generalised version of the code fragment typing relation that includes the store S for the purposes of typing certain intermediate representations[3]. In addition, it asserts a well-formedness condition $\vdash_s$ of the store S and a frame validity relation $\vdash_f$ that produces an associated typing context for each frame F and store S, the details of which are omitted here.

With the above definitions, we describe the concrete statements of progress and preservation for Wasm semantics as we defined abstractly in Propositions 2.1 and 2.2.

Proposition 2.3 (Progress for Wasm).

$$\forall S, F, e^*, t^*. \left((\vdash_{\mathrm{cfg}} (S; F; e^*) : t^*) \implies (\exists S', F', e'^*, (S; F; e^*) \hookrightarrow (S'; F'; e'^*)) \lor \mathsf{terminal}((S; F; e^*)) \right)$$

However, a more fine-grained progress statement on fragment typing relations is required to prove the above progress property for Wasm configurations:

Proposition 2.4 (Fragment Progress for Wasm).

$$\forall S, F, C, e^*, t_1^*, t_2^*. \left((S \vdash_f F : C) \land (\vdash_s S : \mathrm{ok}) \land (S; C \vdash e^* : t_1^* \to t_2^*) \right)$$
$$\implies \left[\forall v^*. (\mathsf{typeof}(v^*) = t_1^*) \implies (\exists S', F', e'^*, (S; F; v^* +\!\!+ e^*) \hookrightarrow (S'; F'; e'^*) \lor \mathsf{terminal}(e^*)) \right]$$

The fragment version of progress states that, for any Wasm program fragment with instructions e^*, if $\vdash e^* : t_1^* \to t_2^*$, then although e^* might not have a reduction on its own (when t_1^* is non-empty) due to missing values on the stack, it can always execute further with the correct types of values on the value stack. This stronger statement, which implies the "top-level" progress property, is necessary to construct a proper inductive hypothesis over Wasm's evaluation contexts when proving progress by way of induction on the definition of Wasm's typing judgement.

The preservation statement is straightforward :

Proposition 2.5 (Preservation for Wasm).

$$\forall S, F, e^*, S', F', e'^*, t^*. \left(((\vdash (S; F; e^*) : t^*) \land ((S; F; e^*) \hookrightarrow (S'; F'; e'^*))) \implies (\vdash (S'; F'; e'^*) : t^*) \right).$$

The proof of the preservation property is established by induction over the reduction relation.

As discussed above, we make minor adaptations to the existing Wasm type soundness proof of Watt et al. [45] in order to make the proofs fully constructive and executable. The proofs were already almost in the right form — the main change involved replacing occurrences of Coq's regular existential quantification (in `Prop`) with Sigma-type existential quantification (in `Type`) throughout the proof. This adjustment was minimally-invasive and completed within a handful of hours.

With the above foundation, we can now discuss in detail the causes of inefficiency of the interpreter extracted from type soundness, and the extent to which they can be addressed through changes in the structure of the proofs. Ultimately, some performance issues are fundamental to this approach, motivating the alternative approach we describe in Section 3.

[3] For example, the label typing rule in Fig 4 is, in fact, defined using this generalised version in the WebAssembly standard, thereby including the store S; however, it is not accessed in the rule at all. We therefore omitted it from the rule in Fig 4 to avoid confusion.

2.4 Inefficient proof tree traversals

As mentioned, the extracted interpreter relies on the progress proof to provide one-step execution results and the preservation proof to provide an associated typing term. At each iteration step, the progress and preservation proofs may need to traverse large portions of the typing or reduction proof trees in order to produce their results. As a result, the performance of the extracted interpreter is significantly impacted by the large sizes of these proof trees.

This issue is particularly pronounced for industrial languages like Wasm, which have complex structural typing rules. For instance, the reduction and typing rules for **label**, a structural instruction that models blocks of code on the stack, illustrate this complexity. Figure 5 displays some of the reduction and typing rules for **label**.

$$\frac{(S; F; e^*) \hookrightarrow (S'; F'; e'^*)}{(S; F; \mathbf{label}_n\{e^*_{\text{cont}}\}\ e^*\ \mathbf{end}) \hookrightarrow (S'; F'; \mathbf{label}_n\{e^*_{\text{cont}}\}\ e'^*\ \mathbf{end})}\ \text{label_reduction}$$

$$\frac{}{[\mathbf{label}_n\{e^*_{\text{cont}}\}\ v^*\ \mathbf{end}] \hookrightarrow v^*}\ \text{label_exit}$$

$$\frac{C \vdash e^*_{\text{cont}} : t_1^n \to t_2^* \quad C' = C\left[\text{label} := [t_1^n] +\!\!+ C.\text{label}\right] \quad C' \vdash e^* : [] \to t_2^*}{C \vdash \mathbf{label}_n\{e^*_{\text{cont}}\}\ e^*\ \mathbf{end} : [] \to t_2^*}\ \text{label_typing}$$

Fig. 5. Selected reduction and typing rules for **label**

The label_reduction rule states that a label instruction can be reduced if its body can be reduced, while the label_exit rule states that if the body is already a list of values, the label can be reduced to that value list. The typing rule specifies that for a **label** instruction to be well-typed, its body must be typeable with the same type — the slightly different context is used to type control flow instructions which we do not describe in detail here.

Recall that the progress proof proceeds by induction over the typing of the configuration. In the **label** case, we must apply the induction hypothesis that, since the body of the **label** is also well-typed, it must either take a step or be terminal. if e^* takes a step, then the original **label** takes a step according to the label_reduction rule; if e^* is terminal, then the **label** takes a step by exiting the label according to the label_exit rule [4]. To see the way that this proof structure gives rise to inefficiency of the interpreter, consider the typing term of the following small program in Figure 6 involving multiple nested labels.

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\vdash [\mathbf{i32.const}\ c_1; \mathbf{i32.const}\ c_2] : [] \to [\mathbf{i32}; \mathbf{i32}]}\ \text{composition} \qquad \dfrac{}{\vdash [\mathbf{i32.add}] : [\mathbf{i32}; \mathbf{i32}] \to [\mathbf{i32}]}\ \text{add}}{\vdash [\mathbf{i32.const}\ c_1; \mathbf{i32.const}\ c_2; \mathbf{i32.add}] : [] \to [\mathbf{i32}]}\ \text{composition}}{\vdash [\mathbf{label}_0\{\}\ [\mathbf{i32.const}\ c_1; \mathbf{i32.const}\ c_2; \mathbf{i32.add}]\ \mathbf{end}] : [] \to [\mathbf{i32}]}\ \text{label_typing}}{\vdash [\mathbf{label}_0\{\}\ [\mathbf{label}_0\{\}\ [\mathbf{i32.const}\ c_1; \mathbf{i32.const}\ c_2; \mathbf{i32.add}]\ \mathbf{end}]\ \mathbf{end}] : [] \to [\mathbf{i32}]}\ \text{label_typing}$$

Fig. 6. Typing term of a program with deeply nested **labels**

As part of "executing" the progress definition, the application of the induction hypothesis in the label typing case translates to a recursive invocation on the label's body. The progress definition must

[4] Special treatment is required when the body e^* contains a control flow instruction **br** at its hole, in which case the continuation is targeted. The corresponding case in the progress proof requires a decision process on decomposing the label body e^* at every step, which is also inefficient. We omit these details in the paper due to space constraints.

be recursively invoked over each nested label body to produce one step of execution, backtracking through all nested levels to construct the execution result and the corresponding reduction proof tree. Similarly, the preservation function must traverse the reduction tree composed of nested label_reduction rules to form the typing tree of the result configuration. Each time the interpreter steps, this recursion and reconstruction process is repeated. This process is highly inefficient and does not scale well to larger Wasm programs.

Improving the performance of this case is challenging — it arises directly due to the inductive structure of the progress proof over **label**. As we will discuss in Section 3.3, we instead want direct control over the interpreter's representation of the evaluation context, rather than relying on the structure inherent in the naturally-arranged proof of progress.

2.5 Proof tree explosion

Another cause of inefficiency in the type soundness interpreter is the *explosion* in the size of proof trees, resulting from unexpected interactions between the proofs and some of Wasm's typing rules. Recall that our automatically-derived interpreter, even when extracted to OCaml, must explicitly represent the proof term of well-typedness of the configuration in memory as a full proof tree, and traverse large portions of it at each step. This means that a larger proof tree directly translates to lower performance of the interpreter. We encountered this issue while benchmarking the extracted interpreter and observed a super-linear complexity for a program that computes the nth Fibonacci number using a loop, which should theoretically have $O(n)$ time complexity. Importantly, this issue is orthogonal to the above issue with **label** contexts, as the iterative Fibonacci algorithm does not introduce deeply-nested labels.

Figure 7a shows the execution of an $O(n)$ iterative Fibonacci Wasm program[5] that computes the nth Fibonacci number. By graphing the maximum and average size of the proof tree[6] during execution against different values of the input size n, we can observe how the proof tree size grows linearly with the number of loop iterations. Since each execution step of the type-safety interpreter involves traversing a large part of the proof tree, we expect the time complexity of the execution per step to be $O(n)$. On the other hand, $O(n)$ steps are required to compute the nth Fibonacci number. This results in an overall time complexity of $O(n^2)$, which agrees with our observation in Figure 7b[7].

The proof of Watt et al. [45], from which we automatically derive the interpreter, was not structured so as to minimise the size of the generated proof term. In particular, we identify that subtle choices in the use of the composition and subsumption typing rules can have massive impact on the size of the proof term.

To illustrate this problem, we revisit the composition typing rule, which describes how a sequence of instructions is typed. We currently formulate Wasm's composition rule by allowing the splitting of one element at the end of the sequence each time as follows:

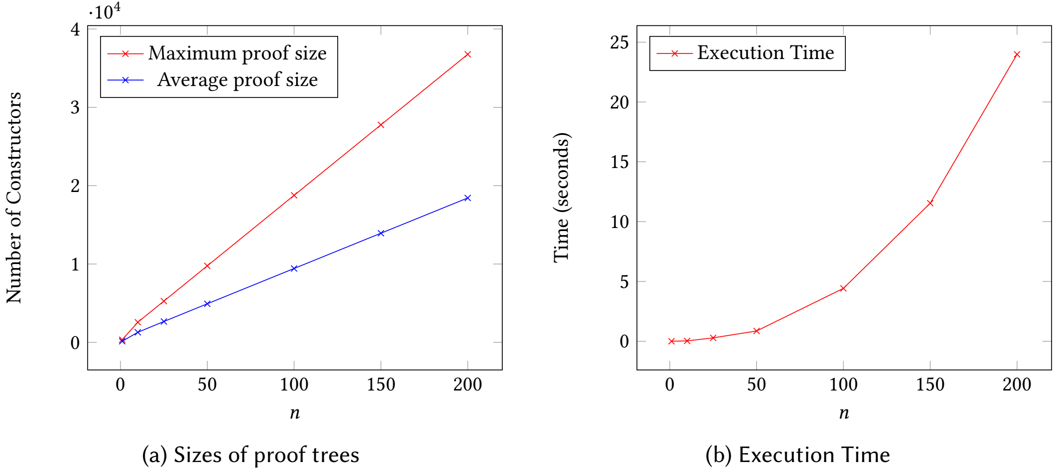
$$\frac{C \vdash e^* : t_1^* \to t_3^* \quad C \vdash [e] : t_3^* \to t_2^*}{C \vdash e^* +\!\!+ [e] : t_1^* \to t_2^*}\ \text{composition}$$

There are various other formulations that are equivalent to this choice from the perspective of the type system. For example, the composition rule could also be stated to split one element from the head of the list each time, or to simply allow general list concatenations. The formulation given above most closely follows the structure of industrial type checking algorithms for Wasm, which

[5]That is, a for-loop style program that calculates all the values in one pass.

[6]We define the *size* of the proof tree to be the number of basic and administrative typing rule constructors that appear in the typing derivation tree. The *average* and *maximum* sizes are calculated from the sizes at every step of execution.

[7]Further benchmarks and information regarding the benchmarks are later described in § 4.



(a) Sizes of proof trees (b) Execution Time

Fig. 7. Benchmark: $O(n)$ iterative Fibonacci with input n

operate in a single linear and incremental pass of the program. Note though our discussion in §5.1, where we report our discovery that the latest official Wasm specification includes a flatly incorrect version of this rule.

However, this seemingly innocuous typing rule is a major contributor to the proof tree explosion problem. Consider the following *typing inversion* lemma:

Lemma 2.6.

$$\forall e_1^*, e_2^*, t_1^*, t_2^*.(\vdash e_1^* +\!\!+ e_2^* : t_1^* \to t_2^*) \implies (\exists t_3^*. \vdash e_1^* : t_1^* \to t_3^* \wedge \vdash e_2^* : t_3^* \to t_2^*).$$

The above lemma states that if a concatenation of two program fragments e_1^* and e_2^* has a function type $t_1^* \to t_2^*$, then it must be the case where e_1^* and e_2^* are separately well-typed with some appropriate function types that match.

The proof is naturally conducted by (snoc) induction on e_2^* from the end. If e_2^* is empty, the desired typing term for e_2^* is obtained by the empty typing rule, and the desired typing term for e_1^* is simply the original typing term given in the premise. Otherwise, let $e_2^* = e'^* +\!\!+ [e]$. The original premise can be rearranged to

$$\vdash (e_1^* +\!\!+ e'^*) +\!\!+ [e] : v_1^* \to v_2^*$$

We apply an auxiliary lemma, which is a special case of the original lemma to be proved when the second instruction list e_2^* is a singleton list (proof omitted) to the above and obtain

$$\exists t_3^*. \vdash e_1^* +\!\!+ e'^* : t_1^* \to t_3^* \wedge \vdash [e] : t_3^* \to t_2^*$$

Now, by the induction hypothesis on e'^* we have

$$\exists t_4^*. \vdash e_1^* : t_1^* \to t_4^* \wedge \vdash e'^* : t_4^* \to t_3^*$$

The first part of the conjunction provides the typing term for e_1^*, and the typing term for $e_2^* = e'^* +\!\!+ [e]$ can be applied by applying the composition typing rule with the typing terms for e'^* and $[e]$ we've obtained.

While the above proof is valid from the perspective of proving the lemma, and is in some sense the most "generic" formulation of the proof, the key observation is that it generates a typing tree with n composition rules for e_2^* when $|e_2^*| = n$, instead of the $n - 1$ required. This is because the

induction's base case is $e_2^* = []$, leading to a redundant application of the composition typing rule when the induction hypothesis is applied when $e' = []$ in the inductive case to construct a typing term for $\vdash [] +\!\!+ [e]$ from that of $\vdash []$ and $\vdash [e]$.

While a redundancy of 1 can seem negligible, the above lemma is important in proving the preservation property, specifically in the case of the proof (done by induction on the reduction relation) where the top of the reduction term is a context reduction rule, and specifically the sequential context which is described by the following reduction rule

$$\frac{(S; F; e^*) \hookrightarrow (S'; F'; e'^*)}{(S; F; v_c^* +\!\!+ e^* +\!\!+ e_c^*) \hookrightarrow (S'; F'; v_c^* +\!\!+ e'^* +\!\!+ e_c^*)} \text{ context_sequence}$$

The preservation proof for this assumes a reduction term $(S; F; e^*) \hookrightarrow (S'; F'; e'^*)$, along with a typing term for the previous configuration $(S; F; v_c^* +\!\!+ e^* +\!\!+ e_c^*)$, and must construct a typing term for the result configuration $(S'; F'; v_c^* +\!\!+ e'^* +\!\!+ e_c^*)$. The proof starts by first applying the typing inversion lemma 2.6 twice to obtain the typing terms for v_c^*, e^*, and e_c^* individually from the original typing term, introducing a redundancy of 1 to the size of the typing terms of e'^* and e_c^*. The proof then applies its inductive hypothesis to obtain a typing term for e'^* from the typing term of e^* and the reduction term; this means that the redundancy of 1 is inherited in this step for each application of the context_sequence rule in the reduction term. As a result, the preservation proof in fact introduces a redundancy equal to at least the number of times that context_sequence appears in the reduction term.

Exacerbating the problem, the reduction term produced by the progress proof contains the same number of context_sequence rule as the length of the program, as the natural induction over the typing term applies the context_sequence rule once per instruction (details omitted). Therefore, for larger programs, the typing term is quickly inundated with an insurmountable number of redundant composition rules for the interpreter to deal with.

In fact, when we modify the inductive case of Lemma (2.6) to specifically test for and avoid applying the induction hypothesis in the case where $e' = []$, an additional case split which is irrelevant from the point of view of completing the proof, the observed size of the proof tree in memory shrinks by two orders of magnitude. We report in §4 the performance of an "optimised" automatically-derived interpreter with as many of these lemma fixups as possible. The sensitivity of the size of the proof tree to these small changes, which are at odds with the most uniform approach to stating the relevant inductive proofs, creates an unfortunate tension between the performance of the interpreter and the natural structure of the proof. This observation, combined with our observation about the inefficiency of **label** execution above, motivates a converse approach where the interpreter is directly defined, and the proof follows the structure of the interpreter, which we will now detail.

3 Progress from Progressful Interpreter

In this section, we present an alternative approach that augments a one-step interpreter with dependently-typed certifications, enabling direct control over the interpreter's structure. We are motivated by the following high-level observation: in the approach of §2, the constructive progress proof gives rise to an interpreter and its soundness proof because of obvious structural similarities between their natural definitions. In fact, as we discuss in §3.1, in certain intrinsically-typed settings there is no distinction between the three. Our approach in this section seeks to invert this relationship between the progress proof and the interpreter — showing that, even in an extrinsically-typed setting, a dependently-typed interpreter can give rise to a proof of the classic progress property. That is, §2 focuses on getting a sound but inefficient interpreter "for free" from

a proof of the progress property. Now, our *progressful* interpreter approach provides a proof of progress and more "for cheap" from the dependently-typed definition of an efficient interpreter.

By selecting the appropriate certifying proposition as part of the interpreter's type, this method allows us to combine the definition of the interpreter, the proof of the interpreter's soundness, and the proof of progress into a single definition — maintaining many of the proof maintenance benefits of the previous section's approach with two distinct advantages. First, we can arrange the interpreter so that these proof terms are erased upon extraction from Coq, and second, we can directly optimise the interpreter's internal state in order to improve its performance. As we discuss in §3.1, we make the decision to keep preservation as a separate lemma, for reasons related to the non-determinism of Wasm's inductive operational semantics.

Consider the standard version of the one-step interpreter, which takes an input configuration cfg and either returns a new configuration cfg′ as a successful one-step execution result or produces an error (for example, in the case that the input configuration is ill-typed). Now consider extending the one-step interpreter to a dependently-typed function such that, in the successful case where a result configuration cfg′ is produced, a proof term representing $\mathrm{cfg} \hookrightarrow \mathrm{cfg}'$ is also produced. Such an approach would mean that the implementation and successful typing of the function itself proves that the interpreter is sound. This approach of using dependently-typed functions to simultaneously build certification along with the definition is well-understood in previous work as shown by Chlipala [11].

Now, we additionally augment the error case with a proof term explaining why the interpreter failed to take a step given its input configuration cfg. For Wasm, this would appear as follows:

$$\mathsf{terminal}(\mathsf{cfg}) \lor (\forall t.\ \vdash \mathsf{cfg} : t \implies \mathsf{False})$$

Our key observation is that this proof term (together with the already established proof term in the mutually-exclusive successful case) represents the *contrapositive* of the progress property - "if the interpreter fails to step soundly according to the operational semantics, the input must be ill-typed". This certification establishes that the interpreter is a sufficient witness for a constructive proof of the progress property, and therefore by extending our interpreter with these proof terms, we have established the progress property of Wasm's original inductive semantics while certifying not just the soundness of the interpreter, but a stronger property that it will always successfully take a step if its input is well-typed. Throughout this paper, we will describe an interpreter carrying such a certification of the progress property as *progressful*. In contrast to the more traditional approach of Watt et al. [45], which requires a separate interpreter definition, soundness proof of the interpreter, and proof of progress, in our setting all of these definitions and proofs are combined into a single dependently-typed interpreter definition. Since the proof term components of the interpreter are not required as input, they can be *fully erased* when extracting the interpreter to OCaml, in contrast to Kokke et al. [23] which requires a runtime representation of the typing term. Moreover, in contrast to §2, we demonstrate that we can directly optimise the internal representation of the interpreter while maintaining the benefits of having a unified definition. In this section we first present a fairly naïve interpreter design, before describing our optimisations in §3.3.

3.1 Connection to intrinsically typed languages

Previous work by Bach Poulsen et al. [6] has described a deeply-related approach, centred around an intrinsically typed language definition combined with a dependently-typed *definitional* interpreter. This approach avoids the need for a separate type soundness proof, as the successful definition and typing of the interpreter itself in the host metatheory (e.g. Agda) embodies guarantees equivalent to progress and preservation. If the interpreter's (host) type signature guarantees that an intrinsically-typed input will result in a well-formed output (as opposed to some distinguished error value), this

embodies progress. If the input and output configurations of the interpreter are specified in the interpreter's (host) type signature as having the same (language-level) intrinsic type, this embodies preservation. Similarly, since there is no separate inductive operational semantics, it is meaningless to ask whether the interpreter is sound or complete — it is simply the normative definition of the language's semantics. Their work argues that for a deterministic language there seems to be no inherent drawback in presenting a small-step semantics as a definitional interpreter rather than as an inductive relation.

We are able to capture much of the spirit of this intrinsically-typed setting in our work. In both settings, the computational structure of the interpreter itself is used to "share work" with proofs of a similar structure — the term manipulation performed by an interpreter in the intrinsic setting in order to successfully establish its host type certifying that its (intrinsically well-typed) input results in a non-erroneous output closely parallels the reasoning necessary for our progressful interpreter to establish the contrapositive certification in our error case. Moreover, since the structure of this computation and reasoning closely parallels the structure of both an interpreter soundness proof and a progress proof (as exploited in the other direction by §2), we can cheaply establish the necessary certification in the non-erroneous step case such that our interpreter's certification as a whole not only implies its own soundness with respect to Wasm's operational semantics, but also the progress property. As one distinction, all of our type-level reasoning can be erased during Coq extraction, which helps us make our interpreter as efficient as possible.

If Wasm was deterministic, we could also incorporate preservation into this certification with little effort, again paralleling Bach Poulsen et al. [6]. Non-deterministic language semantics present somewhat of a challenge to approaches based on a definitional interpreter, which must directly represent every outcome that is intended to be allowed, potentially requiring fiddly or computationally-inefficient manipulation of constructs such as choice monads. In particular Wasm, despite its design aiming for determinism wherever possible, is non-deterministic and non-confluent in several edge-cases, which perhaps explains the official specification's decision to define its operational semantics in terms of an inductive relation. This motivates our decision to define a sound dependently-typed interpreter which embodies the progress property, while leaving preservation as a separate proof — in order to also incorporate preservation we would require not only the definition of an interpreter with non-deterministic choice but *also* a proof that this choice mechanism is complete with respect to Wasm's inductive operational semantics. We note that this approach would be theoretically feasible, but feel that it conflicts with our goals of minimizing the maintenance burden of WasmCert-Coq's mechanisation.

3.2 Progressful WebAssembly 1.0 Interpreter

We now show how the above approach for a dependently typed progressful interpreter can be realised in the WasmCert-Coq mechanisation of Wasm. We describe our implementation of the progressful Wasm one-step interpreter, by starting with the original interpreter of WasmCert-Coq, and showing how its result type and body can be extended with a certification of progressfulness.

Recall from Figure 3 that WebAssembly's operational semantics is defined as an inductive relation between Wasm's configuration tuples $(S; F; e^*)$. The ordinary one-step interpreter eval_1 is therefore a function that takes a configuration tuple $(S; F; e^*)$ as its input argument, and returns one of the following results:

- $\mathbf{R}_{\text{normal}}\ (S'; F'; e'^*)$ – a normal step of computation returning a new configuration tuple;
- $\mathbf{R}_{\text{value}}\ v^*$ – a termination result that the instructions e^* to be executed in the input configuration is already a list of values v^*;
- $\mathbf{R}_{\text{error}}$ – an assertion failure that should only occur if the input configuration is ill-typed;

- $\mathbf{R}_{br}\ k\ v^*$ and $\mathbf{R}_{return}\ v^*$ – exceptional results used to represent WebAssembly's special structured control flow instructions **br** and **return** respectively. A detailed description of these cases is not required to understand the approach of this work, but WasmCert-Coq handles them in full.

The progressful interpreter takes a configuration $(S; F; e^*)$ as input and returns a dependently-typed result that includes the necessary progressful certifications for each case:

- $\mathbf{R}_{\mathsf{normal}}\ (S'; F'; e'^*)\ (H_{\mathsf{reduce}} : (S; F; e^*) \hookrightarrow (S'; F;' e'^*))$, a normal step of computation with a proof of the corresponding reduction;
- $\mathbf{R}_{\mathsf{value}}\ v^*\ (H_{\mathsf{val}} : \mathsf{to_value}(e^*) = v^*)$, a termination result with a proof that the input configuration is already a list of values v^*;
- $\mathbf{R}_{\mathsf{error}}\ (H_{\mathsf{error}} : \mathsf{fragment_illtyped}\ S\ F\ e^*)$, an error result with a proof that the input configuration is ill-typed. Note that in the error case, our progressful interpreter needs to certify the following:
$$\forall t^*.(\vdash (S; F; e^*) : t^* \longrightarrow \mathsf{False})$$
For technical reasons related to Wasm's exceptional control flow cases, we first establish a stronger version of this property, fragment_illtyped, which implies this top-level ill-typedness statement. We elide its full definition for space reasons.
- $\mathbf{R}_{br}\ k\ v^*$ and $\mathbf{R}_{return}\ v^*$ $(H_{\mathsf{error}} : \mathsf{wf_\{br}\ k\ v^*, \mathsf{return}\ v^*\}\ S\ F\ e^*)$ are also augmented with corresponding certifications which we elide here but handle in full in the mechanisation.

We present a pseudocode of our implementation in Figure 8. In the pseudocode, `split_vals` is a function that splits up the value stack v^* and instruction stack from the input instruction list e^*. Execution then follows by looking up the top instruction e if there is one to execute and performs a case split and returns a terminal result otherwise. For each case of the instruction e, if the associated constraints are satisfied by the input, the interpreter returns a successful result $\mathbf{R}_{\mathsf{normal}}$ and constructs a successful certification H_{reduce} *in place*. Otherwise, the interpreter returns an error result $\mathbf{R}_{\mathsf{error}}$ with a certification H_{error} proving the ill-typedness of the input configuration.

```
1:  Input : (S; F; e*)
2:  match split_vals e* with
3:  | (v*, []) ⟹ return R_value rev(v*) (H_value : ...)
4:  | (v*, e :: e0*) ⟹
5:     match e with
6:     | t.add :
7:        if v* = [t.const c2; t.const c1] ++ v'*:
8:           return R_normal (S; F; rev(v'*) ++ [t.const (c1 + c2)] ++ e0*) (H_reduce : ...)
9:        else
10:          return R_error (H_error : ...)
11:    | local.get j :
12:       if F.local[j] = Some v:
13:          return R_normal (S; F; rev(v'*) ++ [v] ++ e0*) (H_reduce : ...)
14:       else
15:          return R_error (H_error : ...)
16:    ...
17:    end match
18: end match
```

Fig. 8. Pseudocode of the Wasm 1.0 progressful one-step interpreter

We omit the concrete syntax of constructing the successful and error certifications in each case from the above figure. However, we will now explain the structure of our method in detail to demonstrate how ill-typedness proofs like the above can be constructed in a modular and scalable way.

In principle, each ill-typedness statement is an implication from a typing term to False, therefore the proof is performed naturally by inverting the structure of the typing term. However, Wasm's subsumption typing rule makes this proof slightly more difficult, as each code fragment e^* can be associated with different function types up to the subsumption rule. To design a scalable infrastructure for these proofs, we utilise a set of *typing inversion* lemmas, which were originally established for proving the preservation property. Given a fragment typing relation, the corresponding typing inversion lemma provides a set of constraints that the associated function type needs to satisfy.

We display several typing inversion lemmas in Figure 9.

$$
\begin{array}{rrl}
\text{(empty)}\ \forall t_1^*, t_2^*. & (\vdash [\,] : t_1^* \to t_2^*) & \Longrightarrow (t_1^* = t_2^*) \\
\text{(local.get)}\ \forall C, t_1^*, t_2^*. & (C \vdash [\textbf{local.get}\ j] : t_1^* \to t_2^*) & \Longrightarrow (\exists t.(C.\mathsf{local}[j] = \mathsf{Some}\ t) \land (t_2^* = t_1^* +\!\!+ [t])) \\
\text{(composition)}\ \forall e_1^*, e_2^*, t_1^*, t_2^*. & (\vdash e_1^* +\!\!+ e_2^* : t_1^* \to t_2^*) & \Longrightarrow (\exists t_3^*.(\vdash e_1^* : t_1^* \to t_3^*) \land (\vdash e_2^* : t_3^* \to t_2^*)) \\
\text{(values)}\ \forall v^*, t_1^*, t_2^*. & (\vdash v^* : t_1^* \to t_2^*) & \Longrightarrow (t_2^* = t_1^* +\!\!+ (\mathsf{typeof}\ v^*))
\end{array}
$$

Fig. 9. Selected Typing Inversion Lemmas

The proofs of ill-typedness proceed by proving a contradiction based on the information of the input configuration that goes into the error execution case.

As a specific example[8], we prove the error certification H_{error} for the case of **local.get** j.

PROPOSITION 3.1. *If* $F.\mathsf{local}[j] = \mathsf{None}$, *then*

$$\forall t^*, (\vdash (S; F; v^* +\!\!+ [\textbf{local.get}\ j] +\!\!+ e'^*) : [\,] \to t^*) \implies \mathsf{False}).$$

PROOF. We first expand the configuration typing relation to obtain the following typing relation for program fragment:

$$\ldots \land (S \vdash_{\mathsf{f}} F : C) \land (S; C \vdash (v^* +\!\!+ [\textbf{local.get}\ j] +\!\!+ e'^*) : [\,] \to t^*)$$

The frame validity relation, whose detailed definition we omitted in this paper, would provide that

$$C.\mathsf{local}[j] = \mathsf{None}.$$

Thus

$$
\begin{aligned}
& S; C \vdash (v^* +\!\!+ [\textbf{local.get}\ j] +\!\!+ e'^*) : [\,] \to t^* \\
\Longrightarrow\ & \exists t_3^*.(S; C \vdash v^* : [\,] \to t_3^*) \land (S; C \vdash ([\textbf{local.get}\ j] +\!\!+ e'^*) : t_3^* \to t^*) \\
\Longrightarrow\ & (S; C \vdash ([\textbf{local.get}\ j] +\!\!+ e'^*) : (\mathsf{typeof}\ v^*) \to t^*) \\
\Longrightarrow\ & \exists t_3^*.(S; C \vdash [\textbf{local.get}\ j] : (\mathsf{typeof}\ v^*) \to t_3^*) \\
\Longrightarrow\ & \exists t.(C.\mathsf{local}[j] = \mathsf{Some}\ t)
\end{aligned}
$$

But the last line contradicts with the premise that $C.\mathsf{local}[j] = \mathsf{None}$. □

[8]For demonstration in the paper, we only present the proof for the config ill-typedness. However, the full fragment ill-typedness is essentially done in the same way in the mechanisation.

The execution for other cases, in particular for control-flow instructions **br** and **return**, and the related block-like instructions **label** and **frame**, require special treatment in the interpreter, where error certifications are propagated through exiting of the blocks, which are more difficult to handle. However, we elide the details of these cases due to space constraints of the paper.

With the above interpreter defined, we have constructed, in one go, a dependently-typed interpreter $\mathcal{E}_1$ augmented with:

- A proof that its successful executions are sound;
- A proof that its erroneous executions arise from ill-typed inputs.

From these together, we can derive the progress property by contrapositive reasoning.

3.3 Optimising the Augmented Interpreter: Efficient Runtime Representation

One major advantage of our progressful interpreter in §3.2 over the interpreter derived from type soundness proofs is our ability to directly control the structure of the interpreter, making optimisations more feasible. In this section, we demonstrate an optimisation to the augmented Wasm interpreter by using a significantly more efficient runtime representation for evaluation contexts. This optimisation was first discussed in WasmRef-Isabelle by Watt et al. [46], though we employ a slightly different formulation in our work.

Recall the one-step interpreter in Figure 8. At every step of execution, the interpreter needs to decompose the input instruction list e^* into an evaluation context with a hole containing at most one instruction[9] $\mathcal{E}[e^?]$. This procedure requires traversing not only each nested **label** and **frame** context in the input instruction e^*, but also the linear instruction list (line 2, Figure 8) every time to locate the top instruction to be executed. As we discuss in Section 2, our automatically-derived interpreter suffers from a similar inefficiency.

The solution proposed by Watt et al. [46] addresses this inefficiency by switching to a more efficient representation of the evaluation context. Instead of naïvely defining the interpreter on Wasm's configuration tuple, this approach moves entered labels into a side data structure, avoiding re-recursing into them at each execution step.

The optimised representation defines three kinds of nested single-hole contexts, $\mathcal{E}_{\mathsf{stack}}$, $\mathcal{E}_{\mathsf{label}}$, and $\mathcal{E}_{\mathsf{frame}}$, as shown in Figure 10. Each of these contexts can be mapped to a regular Wasm term using the family of $\mathcal{E}[\![e^*]\!]$ functions described in Figure 10, which fill the context hole with the argument e^*.

$$
\begin{aligned}
&\text{(interpreter runtime tuple) } (S; \mathcal{E}^*_{\mathsf{frame}}; \mathcal{E}_{\mathsf{stack}}; e^?)\\
&\text{(stack context) } \mathcal{E}_{\mathsf{stack}} := \textbf{stack_ctx}\ v^*_{\mathsf{stack}}\ e^*_{\mathsf{stack}}\\
&\text{(label context) } \mathcal{E}_{\mathsf{label}} := \textbf{label_ctx}\ n\ e^*_{\mathsf{cont}}\ \mathcal{E}_{\mathsf{stack}}\\
&\text{(frame context) } \mathcal{E}_{\mathsf{frame}} := \textbf{frame_ctx}\ n\ F\ \mathcal{E}^*_{\mathsf{label}}\ \mathcal{E}_{\mathsf{stack}}
\end{aligned}
$$

$$
\begin{aligned}
&(\textbf{stack_ctx}\ v^*_{\mathsf{stack}}\ e^*_{\mathsf{stack}})[\![e^*]\!] = \mathsf{rev}(v^*_{\mathsf{stack}}) +\!\!+\ e^* +\!\!+\ e^*_{\mathsf{cont}}\\
&(\textbf{label_ctx}\ n\ e^*_{\mathsf{cont}}\ \mathcal{E}_{\mathsf{stack}})[\![e^*]\!] = \mathcal{E}_{\mathsf{stack}}[\![\textbf{label}_n\{e^*_{\mathsf{cont}}\}\ e^*\ \textbf{end}]\!]\\
&(\mathcal{E}_{\mathsf{label}} :: \mathcal{E}^*_{\mathsf{label}})[\![e^*]\!] = \mathcal{E}^*_{\mathsf{label}}[\![\mathcal{E}_{\mathsf{label}}[\![e^*]\!]]\!]\\
&(\textbf{frame_ctx}\ n\ F\ \mathcal{E}^*_{\mathsf{label}}\ \mathcal{E}_{\mathsf{stack}})[\![e^*]\!] = \mathcal{E}_{\mathsf{stack}}[\![\textbf{frame}_n\ F\ (\mathcal{E}^*_{\mathsf{label}}[\![e^*]\!])\ \textbf{end}]\!]\\
&(\mathcal{E}_{\mathsf{frame}} :: \mathcal{E}^*_{\mathsf{frame}})[\![e^*]\!] = \mathcal{E}^*_{\mathsf{frame}}[\![\mathcal{E}_{\mathsf{frame}}[\![e^*]\!]]\!]\\
&(S; \mathcal{E}^*_{\mathsf{frame}} +\!\!+\ [\textbf{frame_ctx}\ n\ F\ \mathcal{E}^*_{\mathsf{label}}\ \mathcal{E}_{\mathsf{stack}}])[\![e^*]\!] = (S; F; \mathcal{E}^*_{\mathsf{label}}[\![\mathcal{E}^*_{\mathsf{frame}}[\![e^*]\!]]\!])
\end{aligned}
$$

Fig. 10. Optimised Runtime Representation for Wasm Interpreter and Composition between Interpreter Contexts and Instructions

[9]The top instruction may not exist when the current configuration represents, for example, a **label** or **frame** with an empty body. In such cases, the innermost context is exited via the label_exit or frame_exit rule at the next execution step.

The optimised one-step interpreter now takes an interpreter runtime tuple $(S; \mathcal{E}^*_{\text{frame}}; \mathcal{E}_{\text{stack}}; e^?)$ as its input and returns a tuple of the same shape as the output when successful. This representation is efficient as it avoids traversing through all the evaluation contexts on the instruction stack at every step of execution. Instead, after every step of execution, the one-step interpreter can simply retrieve the next instruction to be executed from the context.

Incidentally, the return type of the augmented interpreter is in fact *simplified* by implementing this optimisation. This is because the new interpreter runtime tuple directly tracks all the existing evaluation contexts on its representation; therefore, the error certification no longer needs to work with *fragment* ill-typedness, but can instead be formulated directly in terms of ill-typedness of the whole Wasm configuration. In addition, the special return types for **br** and **return** are no longer required, as executing **br** and **return** can now simply be done by exiting from the corresponding label or frame contexts directly in the runtime representation.

Concretely, the augmented, progress-deriving interpreter with the above optimisation now returns the following types of results:

- $\mathbf{R}_{\text{normal}}\ (S'; \mathcal{E}'^*_{\text{frame}}; \mathcal{E}'_{\text{stack}}; e'^?)$, with certification

  $$H_{\text{reduce}} : (S; \mathcal{E}^*_{\text{frame}})[\![\mathcal{E}_{\text{stack}}[\![e^?]\!]]\!] \hookrightarrow (S'; \mathcal{E}'^*_{\text{frame}})[\![\mathcal{E}'_{\text{stack}}[\![e'^?]\!]]\!]$$

  Along with a proof that the context filling defined in Fig 10 and the decomposition from the Wasm configuration to the interpreter runtime tuple are inverse to each other, this implies the soundness of the interpreter with respect to the operational semantics.[10]
- $\mathbf{R}_{\text{value}}\ v^*\ (H_{\text{val}} : ...)$, a terminating result indicating that the original input configuration is already a value. We omit the details of the certifying proposition as it also needs to account for the original decomposition procedure;
- $\mathbf{R}_{\text{error}}\ (H_{\text{error}} : \forall t^*, (\vdash (S; \mathcal{E}^*_{\text{frame}})[\![\mathcal{E}_{\text{stack}}[\![e^?]\!]]\!] : t^*) \implies \text{False})$, an error result with a certification that the corresponding Wasm configuration of the input representation is ill-typed.

Due to the simplified process of dealing with control flow instructions as well as no longer needing to deal with fragment ill-typedness, the optimised progressful interpreter is not only more efficient in terms of execution time, but can also be implemented in a *smaller* code size. We will compare these characteristics of our different versions of interpreters in Section 4.

4 Evaluation

In this section, we compare several key metrics across the different interpreters we have discussed and implemented in this paper, including their runtime performances and the engineering efforts. As part of the discussion of the proof engineering, we discuss our experience of implementing our dependently-typed interpreter in Coq's *proof mode*, instead of directly constructing it functionally using the *convoy pattern* [11].

4.1 Runtime performance

In addition to the various versions of interpreters we implemented in this work, we have also fetched some external interpreters to provide more baselines for comparison. The interpreters we tested are listed in Figure 11. For external existing interpreters, we include a source of the interpreter in the figure. For interpreters implemented in this work, we note the sections where they are first discussed in the paper. All interpreters from Coq mechanisations are extracted to

[10] Note that $(S; \mathcal{E}^*)[\![-]\!]$ is only well-defined when $\mathcal{E}^*$ is non-empty, as it needs to contain an outermost special frame that provides the overall frame in the restored Wasm configuration. This is always the case for any decomposition of a Wasm config tuple (whose detail we omitted here) as the original Wasm configuration always contains a frame. Our interpreter additionally proves (with trivial effort) that any successful result it returns preserves this as an invariant.

OCaml as the target language. All benchmarks were run on a MacBook Pro (2019) with 2.3GHz Intel Core i9 Processor and 16GB RAM.[11]

Interpreter	Description	Source
Type Soundness Interpreter (Original)	Interpreter from WasmCert-Coq [45] type soundness proofs	Section 2
Type Soundness Interpreter (Optimised)	Type Soundness Interpreter with optimised proof trees	Section 2
Progressful Interpreter	Dependently-typed Interpreter in Section 3	Section 3
Progressful Interpreter (Optimised)	Above optimised by methods from WasmRef-Isabelle [46]	Section 3.3
Reference Interpreter	Official Reference Interpreter from the Wasm Standard	[16]
WasmCert-Coq	Original WasmCert-Coq [45] Interpreter	[45]
WasmRef-Isabelle	Monadic Interpreter from an Isabelle mechanisation of Wasm	[46]
Wasmtime	An industrial interpreter from Bytecode Alliance	[4]

Fig. 11. List of interpreters compared

The first benchmark is a simple $O(n)$ iterative Fibonacci function, computing $\mathrm{Fib}(n)$ using a loop. The result is displayed in Figure 12.

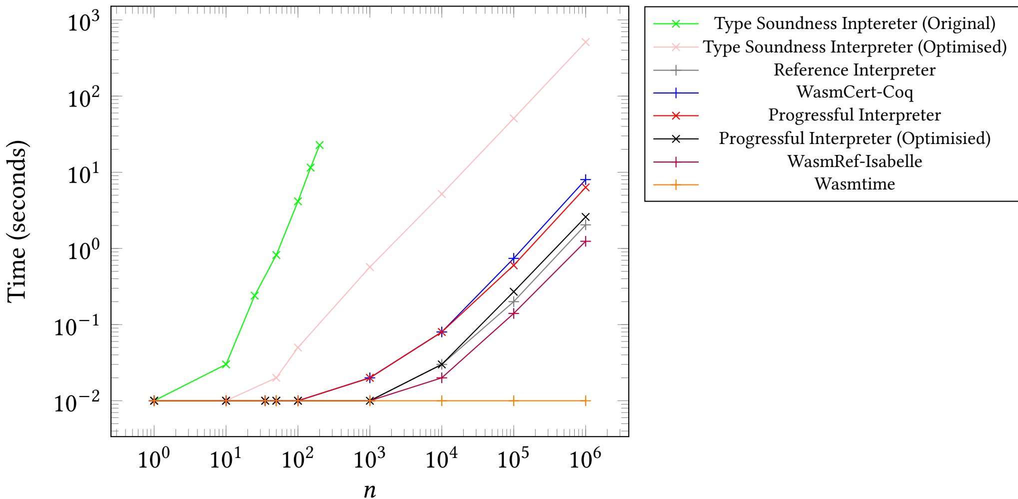


Fig. 12. Benchmark: Iterative Fibonacci with input n, log-scale

We observe that the interpreter directly extracted from the WasmCert-Coq type soundness proofs quickly falls behind due to its super-linear performance[12]. In fact, running the interpreter on an input of $n = 10^3$ resulted in a stack overflow after approximately 20 minutes. An "optimised" version of the interpreter, which changes the structure of the type soundness proofs to minimize the size of the proof tree in memory as discussed in §2.5, achieves a linear runtime, although it takes ~70 times longer than the original WasmCert-Coq interpreter and the unoptimised progressful interpreter. The optimised progressful interpreter has a runtime similar to the reference interpreter from the Wasm standard, while the WasmRef-Isabelle interpreter is approximately twice as fast. Finally, the industrial interpreter Wasmtime from Bytecode Alliance executed the largest test ($n = 10^6$) within 0.01 seconds.

[11]The benchmark results were obtained by averaging from 3 executions and rounded to the nearest 0.01 seconds (with a floor of 0.01 seconds due to the logarithmic plot).

[12]Figure 12 is plotted on a logarithmic scale, so the super-linear performance is reflected by the gradient being larger than 1.

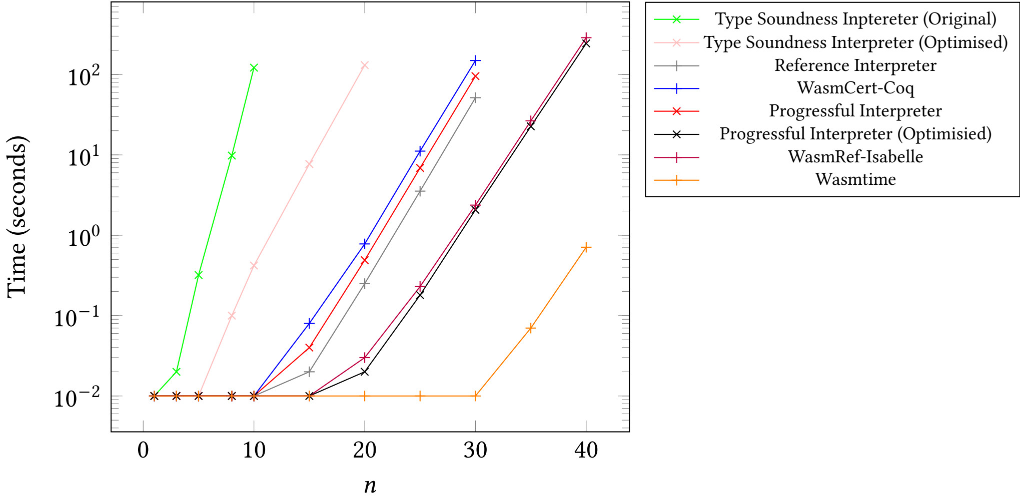


Fig. 13. Benchmark: Recursive Fibonacci with input n, log-scaled time axis

The other benchmark we performed involved a recursive Fibonacci program that calculates the nth Fibonacci number using a recursive function, which has a exponential time complexity. This benchmark was similarly used in Watt et al. [46] to demonstrate the impact of the optimised runtime representation for evaluation contexts. Without this optimisation, the runtime configuration quickly grows in size and contains increasingly deep nested function call frames and labels. Naïve interpreters that use the native Wasm configuration tuples as runtime representation will suffer from performance issues due to the need to traverse all the nested evaluation contexts at every step of execution, as explained in Section 3.3.

Figure 13 displays the benchmark result. All the interpreters without evaluation context optimizations (i.e., all except the optimised progressful interpreter, WasmRef-Isabelle, and WasmTime) can only be tested on inputs up to $n = 30$ or less, either due to excessive runtime or crashing from stack overflow. Our optimised progressful interpreter shows some advantage over WasmRef-Isabelle on all inputs, being approximately 20% faster. The industrial interpreter Wasmtime executes roughly 300 times faster than both of the verified and optimised interpreters.

4.2 Proof engineering effort

Lines of code comparison. We present a comparison of the lines of code used across different implementations of the interpreters in Figure 14, including a detailed breakdown on different components.[13] Auxiliary proofs and definitions, such as the typing inversion lemmas, automation tactics, and proofs to the preservation properties are omitted from the comparison table as they are required for all the approaches.

We note that the interpreter from type soundness comes with the shortest code: this is because the original type soundness proof of WasmCert-Coq is already almost constructive, so we only need to transform the Coq code from the **`Prop`** sort to **`Type`** to extract the proofs from Coq to OCaml.

[13] The lines of code metric (LOC) is an unreliable basis for comparing different implementations because it is influenced by various factors, such as individual engineering practices and coding styles. In this case, the versions being compared were developed by different authors, so the LOC values shown in the table should be viewed as approximate indicators rather than precise measurements. They are intended only to offer a rough comparison of the various methods.
[14] This includes the estimated lines of code in the original proofs that had to be modified.

Components	WasmCert-Coq	Type Soundness Interpreter	Progressful Interpreter	Progressful Interpreter (Optimised)	Progressful Interpreter (Optimised, Wasm 2.0)
Interpreter	456	~300[14]	2396	1940	2463
Interpreter Soundness	1084	Embedded	Embedded	Embedded	Embedded
Interpreter Progressfulness	Not Proved	Embedded	Embedded	Embedded	Embedded
Progress	1039	1039	42 (trivial)	37 (trivial)	37 (trivial)
Total	2579	~1339	2438	1977	2500

Fig. 14. List of interpreters compared

A wrapper function is then implemented to chain the progress, preservation, and type inference functions together accordingly.

We also observe that implementing the evaluation context optimisation from Watt et al. [46] does not result in a larger interpreter definition. A contributing factor may be that the optimised interpreter no longer needs to deal with the control flow instructions **br** and **return** using special auxiliary return results. Not all optimisations can result in simplifications as the above, but this nevertheless demonstrates the feasibility of performing structural optimisations to the progressful interpreter conveniently.

We report on our experience updating the WasmCert-Coq mechanisation to the Wasm 2.0 feature set in §5. For now, we note that the update to Wasm 2.0 results in a $\sim 30\%$ larger code size for the optimised interpreter, while the underlying number of instructions almost doubled.

Implementing Dependently-Typed Functions in Coq's Proof Mode. As part of our experience of implementing the various interpreters in this paper, we find Coq's *proof mode* especially convenient for implementing dependently-typed functions in Coq comparing to using the traditional functional syntax and using convoy patterns for constructing the proof terms.

For a concrete example, we display an outline of the original WasmCert interpreter implementation for **local.get** j in Figure 15a. Transforming it to a progessful interpreter with pseudocode in Figure 8 requires adding a successful certification H_{reduce} to the successful R_{normal} result and an error certification H_{error} to the R_{error} result. We focus on H_{reduce}, which can be constructed as follows:

$$\dfrac{\dfrac{F.\text{local}[j] = v_j}{(S; F; [\textbf{local.get } j]) \hookrightarrow (S; F; [v_j])}\ \text{local.get}}{(S; F; vs \mathbin{+\!\!+} [\textbf{local.get } j] \mathbin{+\!\!+} es') \hookrightarrow (S; F; vs \mathbin{+\!\!+} [v_j] \mathbin{+\!\!+} es')}\ \text{context}$$

The required proof term H_{reduce} can therefore be constructed by applying the two constructors for the context and local.get constructors, provided the knowledge that $F.\text{local}[j] = v_j$. However, while this equality is valid within the specific case of the `match` statement, there is no way to extract this information directly from the original `match` syntax. Instead, we need to use a trick called the *convoy pattern* [11], which works by expanding the return type of the match to a function from the match equality to the original desired result. In this way, the match equality would then be available as an argument in constructing the match result. In the end, the function returned by the match is applied to a trivial reflexive equality of the match argument, thereby constructing the desired result. The shape of the resulting definition of the function is displayed in Figure 15b.

The above method is valid in itself. However, despite several shortcuts that further simplify the syntax slightly, we still find it tedious to completely adopt this pattern for every match case in our implementation. The breaking point that led us to abandon the functional syntax was that a direct definition of the interpreter does not satisfy Coq's strict syntactic termination check for `Fixpoint`,

```
Fixpoint run_one_step S F es :
    run_result S F es :=
...
match e with
| ... => ...
| local_get j =>
    match nth_error F.local j with


    | Some v_at_j =>
        R_normal
            (S; F; rev vs ++ [v_at_j] ++ es')

    | None =>
        R_error
    end
| ... => ...
end
...
```

(a) Original interpreter of WasmCert-Coq

```
Fixpoint run_one_step S F es :
    run_result S F es :=
...
match e with
| ... => ...
| local_get j =>
    match nth_error F.local j as nth_res
    return (nth_error F.local j = nth_res
            -> run_result S F es) with
    | Some v_at_j => (fun Hnth =>
        (R_normal ...
            (S; F; rev vs ++ [v_at_j] ++ es')
            (r_local_get ... Hnth)))
    | None => (fun Hnth =>
        (R_error ... (?HError)))
    end (eq_refl (nth_error F.local j))
| ... => ...
end
...
```

(b) Progressful interpreter using convoy patterns

```
Definition run_one_step S F es : run_result S F es.
Proof.
    ...
    destruct e as
    [ (* Other cases *) ... |
      (* local.get *) j |
      (* Other cases *) ... ].
    ...
    (* local.get *)
    {
        destruct (nth_error F.local j) as [v_at_j |] eqn:Hnth.
        - (* Success *)
            apply (R_normal ... (S; F; vs ++ [v_at_j] ++ es')).
          (* Proof obligation: prove the corresponding Wasm reduction *)
            ...
        - (* Error *)
            apply (R_error ...).
          (* Proof obligation: prove the ill-typedness of the input *)
            ...
    }
Defined.
```

(c) Progressful interpreter in Coq's proof mode

Fig. 15. Different attempts of implementing interpreter execution for **local**.**get**

due to Wasm's structured control flow instructions. Therefore, the interpreter needs to incorporate a decreasing measure calculated from the structural complexity of the input configuration. This proved to be overly complicated to implement with the convoy pattern in the end.

As a result, we opted for an unusual method of defining the entire interpreter in the proof mode of Coq, essentially treating the definition of a function as a proof obligation depending on the input. This approach avoids the convoy pattern altogether and allows Coq's **Ltac** tactics to be directly used in the interpreter construction. Moreover, as displayed in Figure 15c, the certification for each case is constructed as a separate proof obligation after specifying the computation result. This is because constructors of inductive types in Coq can be equivalently used as lemmas that can be applied with the corresponding arguments. A partial application of a constructor means

the remaining arguments (in our case, the certification) become proof obligations, which are then directly proven in the proof mode instead of being constructed functionally from the constructors.

To facilitate recursive calls, the main structure of the interpreter becomes an induction on the size of the input configuration, and recursive calls of the interpreter become applications of the inductive hypothesis, with the necessary premises conveniently established in the proof mode.

Overall, we found this method of engineering eased the tediousness of implementing large dependently-typed functions as complex as our progressful interpreter in Coq.

5 Updating the Wasm 1.0 Mechanisation

As we discuss throughout this work, our key motivation is to reduce the maintenance burden associated with WasmCert-Coq's verified interpreter and type soundness proofs. In particular, we are able to completely remove the old interpreter, its soundness proof, and the type system's proof of progress by switching to our progressful interpreter (§3). We report on our extension of the existing mechanisation of WasmCert-Coq [45] from Wasm 1.0 to the Wasm 2.0 feature set, facilitated by these efficiency savings. Once the Wasm model itself is updated, we now only need to update the progressful interpreter and our separate preservation proof, as opposed to the four separate definitions (progress, preservation, interpreter, interpreter soundness proof) that were previously present. As we note in §4.2 our progressful interpreter takes notably fewer lines of code overall to establish the same results as the definitions and proofs it is replacing. In our subjective experience, we found the progressful interpreter intuitive to work with, and we believe we saved significant effort in only have to update its single definition to Wasm 2.0, as opposed to the three definitions it replaces.

Wasm 2.0 is a major update over version 1.0, integrating multiple extension proposals that vastly extend the abstract syntax and semantics of the language. The following parts of the Wasm 2.0 update interact with our progressful interpreter:

Expanded instruction set. Wasm 2.0 majorly expands Wasm's instruction set, introducing several new types of operations such as function reference operations, vector operations, bulk memory (`memcpy`-like) operations, along with extensions to the capabilities of existing control flow instructions. This is the change that causes the greatest expansion to the mechanisation: the number of instructions in the mechanisation greatly increased from 32 to 57, and the number of reduction rules in the mechanisation increased from 55 to 95, almost doubling the size for both. This essentially means between 25 and 40 new inductive cases must be handled in every key proof or definition that is related to type soundness or the interpreter.

New primitive types. Wasm 2.0 introduced two new basic families of value types in addition to the existing four scalar numeric types **i32**, **i64**, **f32** and **f64**. These include the *vector types* for vector instructions, and the *reference types* for (function) references. These generalisations impacts the existing cases of the interpreter, because these cases now need expanded reasoning to describe new situations in which their input values may be ill-typed (e.g. if a vector value is provided to a scalar numeric operation).

Enriched type system and subtyping. Another key extension is the introduction of a type lattice and subtyping system into Wasm, including a new subsumption rule. This change had a great impact on the infrastructural lemmas in the existing mechanisations, as many assumptions that the existing proofs based on no longer hold – for example, instead of every value being associated to a unique type, a value can now be associated with a set of types, among which one of them is the *principal* type. For the progressful interpreter, the error certification is similarly affected through the updates to the typing inversion lemmas.

Our update to the Wasm 2.0 specification shares a similar trusted computing base with the original WasmCert-Coq [45] mechanisation for Wasm 1.0 and the related verified interpreter from WasmRef-Isabelle [46]. In particular, the extraction process from Coq to OCaml and the OCaml tools themselves need to be trusted, and the vector instructions are similarly implemented as opaque instructions whose behavior agrees with the function type defined in the specification, with the concrete implementation left to be generated at the OCaml level, as we lack a mature formalisation of the relevant machine-level vector operations. This mirrors the approach used by Watt et al. [46] for floating point operations in their Isabelle/HOL mechanisation of Wasm 1.0. Besides the above, we continue to use the verified CompCert [26] numerics for integer and floating point arithmetic and the Parseque parser combinator library [2, 3] to generate the binary format parser.

Overall, the experience of updating the progressful interpreter was smooth. In fact, the most tedious part of the update is to correctly formulate the new definitions introduced by the new version of Wasm and making sure they are organised in a sensible structure.

5.1 Mistakes found

Just as Watt [44] discovered errors in WebAssembly's draft type system, our extension of the existing WasmCert-Coq to the WebAssembly 2.0 feature set has uncovered several errors in the official specification.

5.1.1 Composition and subsumption. The Wasm 2.0 specification introduces a new concept of subtyping between value types given by $\leq$, and attempts to refactor subsumption (with subtyping) and composition into a single composite typing rule, as follows:

$$\frac{}{C \vdash [] : t^* \to t^*} \ \mathsf{emp-poly}$$

$$\frac{C \vdash e^* : t_1^* \to t_0^* +\!\!+ t'^* \qquad t'^* \leq t^* \qquad C \vdash e_N : t^* \to t_3^*}{C \vdash e^* +\!\!+ e_N : t_1^* \to t_0^* +\!\!+ t_3^*} \ \mathsf{seq-poly}$$

However this typing rule is erroneous. As a counter-example, consider the instruction ($\mathbf{block}\ \{t'^* \to t^*\}\ []$) — a block instruction with an empty body. If $t'^* \leq t^*$, then this instruction should successfully type-check, but the typing rules above fail to support this. This error in the official standard has been acknowledged by Wasm's specification editor. While an official fix is in progress, WasmCert-Coq instead uses more traditional subsumption and composition typing rules, drawn from earlier drafts of the type system.

5.1.2 Module typing. Wasm allows the pre-declaration of *active element segments*, which populate a module's function table with a list of pre-declared functions at startup type. With Wasm 2.0, the typing rule for these segments was re-written in anticipation of future features which would generalise their structure. However this revised typing rule omitted a key step in constructing a context representing the types of global module declarations, resulting in a rule which implied that every active element segment was ill-typed. This error was identified concurrently by both ourselves and an independent standards contributor, and has now been fixed.

5.1.3 Missing component of typing context. Wasm 2.0 introduces a new component of the typing context, C.refs, representing a declared list of functions which are permitted to escape the boundaries of the module as dynamic references. However, due to an editorial oversight, C.refs was inconsistently propagated through the typing rules, making certain typing rules erroneously strict. These corrections have been adopted into the specification.

5.1.4 Memory soundness. The appendix of the Wasm specification includes auxiliary definitions necessary for the official statement of the intended soundness properties of the Wasm type system. However one of these auxiliary definitions declared that a Wasm memory has a runtime type based on its current size which remains unchanged during program execution, without correctly accounting for the possibility of memory size increasing due to the **memory.grow** instruction. After we discovered this error, a fix was adopted into the specification.

5.1.5 Missing subsumption rule for values. Another issue with the soundness appendix, the following subsumption rule on the runtime type of values was inadvertently elided but assumed to exist in other definitions.

$$\frac{S \vdash v : t \qquad \vdash t : \mathrm{ok} \qquad t \leq t'}{S \vdash v : t'} \ \mathsf{val-sub}$$

6 Related Works

As we discuss in §3.1, Bach Poulsen et al. [6] present an alternative style for defining a deterministic programming language's semantics, using an intrinsically-typed representation combined with a dependently-typed definitional interpreter. The deep links between dependent types, definitional interpreters, and intrinsically-typed language definitions have also been widely discussed and explored by related work [5, 12, 34, 37, 42]. We believe our work is novel in mapping out the extent to which accepted benefits of the above setting can be transferred to "classical" inductively-defined semantics, without requiring all three of the above approaches to be simultaneously adopted. In particular, Wasm is non-deterministic, its configurations are not intrinsically typed, and our interpreter cannot be definitional. All of these constraints flow directly from the formalisation of Wasm's official standard, which we aim to stay close to wherever possible — any effort to (for example) define an intrinsically-typed Wasm or non-deterministic "definitional" interpreter would *increase* the maintenance burden of WasmCert-Coq, as a correspondence would need to be proved between this new definition and the faithful base mechanisation of the specification. We feel that our approach strikes the right balance in capturing many of the benefits of the intrinsically typed approach, without getting bogged down in these additional complications.

Chapman et al. [10] (drawing from Kokke et al. [23]) is an interesting mid-point between our setting and the intrinsically typed + definitional interpreter setting, as it works with an intrinsically typed language but an inductive relational definition of the language's operational semantics. Such an approach causes the definition of the operational semantics to inherently embody *preservation* in its (host) type signature, but not *progress*, which is constructively proven separately. From this progress property, a sound interpreter can be automatically derived, as we discuss in Section 2. The authors of the above work are not able to execute their interpreter end-to-end as, in contrast to our work, they lack a verified type checker to initiate the interpreter loop. Therefore they do not investigate the relationship between the structure of their soundness proof and the runtime performance of the derived interpreter. Related to the need of a verified type checker, Adjedj et al. [1] describes a mechanised metatheory of the Martin-Löf Type Theory in Coq that could produce a certified and executable type checker from a decidability proof of type checking for their theory.

Youn et al. [49] describes a Domain-Specific Language (DSL) for Wasm specification, SpecTec, which aims to automatically generate Wasm specification artefacts and mechanised semantics, thereby providing a maintainable way to produce mechanised definitions for Wasm. This effort is neatly complementary to our work, as our approach tackles a different pain point in the mechanisation process of maintaining the proofs *on top of an already-produced model*. There is a minor overlap in the *meta-level interpreter* reported by SpecTec, as it could be seen as eliminating the need for any

verified interpreter at all. However, SpecTec's approach is constrained in the optimisations that can be performed, since its intermediate representations must be derived automatically from the *original* Wasm definitions, while we can implement more ambitious optimisations while maintaining full trustworthiness through interactive proofs, as we reported earlier. Overall, our method would lessen the burden of maintaining relevant proofs when a SpecTec-derived mechanisation is extended.

Our challenge of proof maintenance is related to the *expression problem* (EP) [15], which describes the challenge of extending existing inductive definitions with new constructors while reusing the old proofs. The approach of *modular semantics* [30, 31] has been suggested to address this problem. With this approach, a language's syntax and semantic rules are given incrementally from reusable blocks, thereby avoiding the need of reformulation when further constructs are added to the language, and potentially allowing highly modularised proofs. This may be helpful in mechanising Wasm, as the feature proposals of Wasm can often be viewed as optional extensions of the semantics. Previous works have studied concrete implementations of extensible semantics in proof assistants such as Coq and Agda, where Delaware et al. [13], Keuchel and Schrijvers [21], Schwaab and Siek [39] followed the method of Swierstra [40] with workarounds for the respective proof assistants, and Jin et al. [20] opted for a slightly different approach by directly extending the linguistic facilities of Coq. However, industrial language specifications such as Wasm are not always defined in a way that is amenable to modularisation. In particular, new features are simply added as inline patches to the specification text, and some features involve cross-cutting changes to Wasm's abstract representation and control flow. WasmCert-Coq places a high priority on its faithfulness to the original Wasm specification, and so careful consideration would be needed regarding the extent to which attempts at modularising the mechanisation would compromise this correspondence, and the extent to which such a modularisation would be robust against more cross-cutting changes.

7 Conclusion and Future Work

In this paper, we have successfully applied the outline method described in Kokke et al. [23], extracting an end-to-end interpreter capable of running Wasm binaries from the type soundness proofs of the existing WasmCert-Coq mechanization. Additionally, we have for the first time investigated the concrete sources of inefficiency in this approach related to the structure of the type soundness proofs, devising mitigations that improved the super-linear complexity of the directly extracted interpreter, achieving a linear runtime performance.

More importantly, recognizing the inadequacy of the method in Kokke et al. [23] for producing efficient interpreters, we proposed an alternative approach that uses a dependently-typed *progressful* interpreter with certifications for both successful and error results, directly implying the progress property. We fully implemented this alternative method on Wasm 1.0, created an executable artifact, incorporated optimisations from Watt et al. [46], and successfully updated the underlying semantics from Wasm 1.0 to Wasm 2.0 and beyond. This showcased the robustness and scalability of our approach and highlighted the maintenance benefits it offers. Our optimised interpreter achieved performance similar to the non-dependently-typed interpreter in Watt et al. [46] on benchmarks testing the subset of optimisations we implemented. This further demonstrates the feasibility of our approach for maintaining language mechanisations while producing executable interpreters with competitive performance.

However, there are many possible extensions to this work. We chose to only focus on one specific optimisation for the runtime representation discussed in Watt et al. [46]. A direct line of future work is to similarly implement the other optimisation regarding switching to a monadic heap for more efficient state-manipulating operations, which should allow the performance of our interpreter to be competitive in all scenarios with that of Watt et al. [46]. It may also be fruitful to revisit whether integrating the proof of the preservation property into the progressful interpreter

is worthwhile. For a fully deterministic language, it may be a particularly sensible choice. Even for a non-deterministic language like Wasm, careful engineering and automation may reveal further benefits to this approach — it is tempting to consider a truly all-in-one integrated interpreter that derives the entire type-safety property while preserving the benefits of proof maintenance.

Finally, Youn et al. [49] raise the possibility of one day automatically generating a mechanisation of the Wasm semantics directly from a normative DSL. Our current work optimises the process of verifying key artefacts once a Wasm mechanised model has been defined. In combination with an approach such as Youn et al. [49] for automatically defining the model, we can envisage a world where all new Wasm features are rapidly verified as they appear, with minimal effort.

Acknowledgments

Conrad Watt is supported by an NTU Nanyang Assistant Professorship Start-Up Grant. Rao is supported by a Doctoral Scholarship Award from Department of Computing, Imperial College London. Philippa Gardner was supported by the EPSRC fellowship VeTSpec: Verified Trustworthy Software Specification (EP/R034567/1) for much of this work.

Artifact Availability

The artifact of the paper is available as a permanent Zenodo entry [36] and on GitHub [43].

References

[1] Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. 2024. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs* (London, UK) *(CPP 2024)*. Association for Computing Machinery, New York, NY, USA, 230–245. https://doi.org/10.1145/3636501.3636951

[2] Guillaume Allais. 2017. GitHub - Total Parser Combinators in Coq. https://github.com/gallais/parseque.

[3] Guillaume Allais. 2018. Agdarsec - total parser combinators. 45–59. Publisher Copyright: © JFLA 2018 - Journees Francophones des Langages Applicatifs. All rights reserved. Sylvie Boldo, Nicolas Magaud. Journées Francophones des Langages Applicatifs 2018. Sylvie Boldo; Nicolas Magaud. Journées Francophones des Langages Applicatifs 2018, Jan 2018, Banyuls-sur-Mer, France. publié par les auteurs, 2018. ⟨hal-01707376⟩; Vingt-neuviemes Journees Francophones des Langages Applicatifs, JFLA 2018 - 29th French-Speaking Conference on Applicative Languages, JFLA 2018 ; Conference date: 24-01-2018 Through 27-01-2018.

[4] Bytecode Alliance. [n. d.]. GitHub - bytecodealliance/wasmtime: A fast and secure runtime for WebAssembly. https://github.com/bytecodealliance/wasmtime. [Accessed 01-07-2024].

[5] Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 666–679. https://doi.org/10.1145/3009837.3009866

[6] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2017. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.* 2, POPL, Article 16 (dec 2017), 34 pages. https://doi.org/10.1145/3158104

[7] Sandrine Blazy and Xavier Leroy. 2009. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning* 43, 3 (2009), 263–288.

[8] Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A trusted mechanised JavaScript specification. *SIGPLAN Not.* 49, 1 (jan 2014), 87–100. https://doi.org/10.1145/2578855.2535876

[9] Denis Bogdanas and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. *SIGPLAN Not.* 50, 1 (jan 2015), 445–456. https://doi.org/10.1145/2775051.2676982

[10] James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. 2019. System F in Agda, for Fun and Profit. In *Mathematics of Program Construction: 13th International Conference, MPC 2019, Porto, Portugal, October 7–9, 2019, Proceedings* (Porto, Portugal). Springer-Verlag, Berlin, Heidelberg, 255–297. https://doi.org/10.1007/978-3-030-33636-3_10

[11] Adam Chlipala. 2022. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant.* MIT Press.

[12] Nils Anders Danielsson. 2012. Operational semantics using the partiality monad. *SIGPLAN Not.* 47, 9 (sep 2012), 127–138. https://doi.org/10.1145/2398856.2364546

[13] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013. Meta-theory à la carte. *SIGPLAN Not.* 48, 1 (jan 2013), 207–218. https://doi.org/10.1145/2480359.2429094

[14] Chucky Ellison and Grigore Rosu. 2012. An executable formal semantics of C with applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 533–544. https://doi.org/10.1145/2103656.2103719

[15] Philip Wadler et al. 1998. The expression problem. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

[16] W3C WebAssembly Community Group. [n. d.]. GitHub - WebAssembly/spec: WebAssembly specification, reference interpreter, and test suite. https://github.com/WebAssembly/spec/tree/main/interpreter. [Accessed 09-07-2024].

[17] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. *The Essence of JavaScript*. Springer Berlin Heidelberg, 126–150. https://doi.org/10.1007/978-3-642-14107-2_7

[18] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. *SIGPLAN Not.* 52, 6 (jun 2017), 185–200. https://doi.org/10.1145/3140587.3062363

[19] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (may 2001), 396–450. https://doi.org/10.1145/503502.503505

[20] Ende Jin, Nada Amin, and Yizhou Zhang. 2023. Extensible Metatheory Mechanization via Family Polymorphism. *Proc. ACM Program. Lang.* 7, PLDI, Article 172 (jun 2023), 25 pages. https://doi.org/10.1145/3591286

[21] Steven Keuchel and Tom Schrijvers. 2013. Generic datatypes à la carte. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming* (Boston, Massachusetts, USA) *(WGP '13)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/2502488.2502491

[22] Gerwin Klein and Tobias Nipkow. 2006. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 4 (2006), 619–695.

[23] Wen Kokke, Jeremy G. Siek, and Philip Wadler. 2020. Programming language foundations in Agda. *Science of Computer Programming* 194 (2020), 102440. https://doi.org/10.1016/j.scico.2020.102440

[24] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. *ACM SIGPLAN Notices* 49, 1 (2014), 179–191.

[25] Daniel K Lee, Karl Crary, and Robert Harper. 2007. Towards a mechanized metatheory of Standard ML. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 173–184.

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

[27] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

[28] Pierre Letouzey. 2002. A new extraction for Coq. In *International Workshop on Types for Proofs and Programs*. Springer, 200–219.

[29] Pierre Letouzey. 2008. Extraction in coq: An overview. In *Logic and Theory of Algorithms: 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008 Proceedings 4*. Springer, 359–369.

[30] Sheng Liang and Paul Hudak. 1996. Modular denotational semantics for compiler construction. In *European Symposium on Programming*. Springer, 219–234.

[31] Peter D Mosses. 2004. Modular structural operational semantics. *The Journal of Logic and Algebraic Programming* 60 (2004), 195–228.

[32] Michael Norrish. 1998. *C formalised in HOL*. Technical Report UCAM-CL-TR-453. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-453

[33] Scott Owens. 2008. A sound semantics for OCaml light. In *European Symposium on Programming*. Springer, 1–15.

[34] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*. Springer-Verlag, Berlin, Heidelberg, 589–615. https://doi.org/10.1007/978-3-662-49498-1_23

[35] Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 346–356.

[36] Xiaojia Rao, Stefan Radziuk, Conrad Watt, and Philippa Gardner. 2024. *Artifact: Progressful Interpreters for Efficient WebAssembly Mechanisation*. https://doi.org/10.5281/zenodo.14052598

[37] John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, Massachusetts, USA) *(ACM '72)*. Association for Computing Machinery, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

[38] Tiark Rompf and Nada Amin. 2016. From F to DOT: Type Soundness Proofs with Definitional Interpreters. arXiv:1510.05216 [cs.PL] https://arxiv.org/abs/1510.05216

[39] Christopher Schwaab and Jeremy G. Siek. 2013. Modular type-safety proofs in Agda. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification* (Rome, Italy) *(PLPV '13)*. Association for Computing Machinery, New York, NY, USA, 3–12. https://doi.org/10.1145/2428116.2428120

[40] Wouter Swierstra. 2008. Data types à la carte. *Journal of Functional Programming* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

[41] Yong Kiam Tan, Magnus O Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *Journal of Functional Programming* 29 (2019), e2.

[42] Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser, and Peter Mosses. 2022. Intrinsically-typed definitional interpreters à la carte. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 192 (oct 2022), 30 pages. https://doi.org/10.1145/3563355

[43] WasmCert. 2024. WasmCert-Coq: A mechanisation of Wasm in Coq. https://github.com/WasmCert/WasmCert-Coq/

[44] Conrad Watt. 2018. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 53–65. https://doi.org/10.1145/3167082

[45] Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. 2021. Two Mechanisations of WebAssembly 1.0. In *Proceedings of the 24th international symposium of Formal Methods (FM21), Beijing, China; November 20-25, 2021 (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

[46] Conrad Watt, Maja Trela, Peter Lammich, and Florian Märkl. 2023. WasmRef-Isabelle: A Verified Monadic Interpreter and Industrial Fuzzing Oracle for WebAssembly. *Proc. ACM Program. Lang.* 7, PLDI, Article 110 (jun 2023), 24 pages. https://doi.org/10.1145/3591224

[47] Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A Eisenberg. 2017. A specification for dependent types in Haskell. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–29.

[48] A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

[49] Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. 2024. Bringing the WebAssembly Standard up to Speed with SpecTec. *Proc. ACM Program. Lang.* 8, PLDI, Article 210 (jun 2024), 26 pages. https://doi.org/10.1145/3656440

Received 2024-07-11; accepted 2024-11-07