A Mechanized Formalization of the WebAssembly Specification in Coq

Xuan Huang
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
xh5403@cs.rit.edu

Abstract—We developed a mechanized formalization of W3C WebAssembly Core Specification 1.0 [1] in the Coq [2] proof assistant, and (almost) completed a mechanized proof that this semantics is sound, implying both its type safety and memory safety. Wasm has been designed [3] and standardized [1] with formal semantics. In contrast to prior work [4], our mechanized formalization closely follows the standard, exposed issues and provided bug fixes. Our work serves as a foundation for certified software development in the Coq community. Existing projects can incorporate ours to build certified compilers (CompCert [5], Vellvm [6]) and certified interfacing layers (JSCert [7], CertiKOS [8]).

Index Terms—Programming languages; Mechanized semantics; Type soundness

I. INTRODUCTION

A. Theory of programming languages

The formal study of programming languages focuses on describing the semantics of programming languages with mathematical precision, to help with thoroughly understanding the meaning of computer programs and writing reliable software. Intellectual tools enable us to represent language structures as mathematical objects and language behaviors in terms of mathematical functions and relations, allowing the reasoning about and the proving of general properties: the dynamic semantics are often expressed as an operational semantics, a method of specifying the behavior of a programming language by writing an abstract interpreter on the abstract syntax; and the static semantics are often expressed as a type system, a method of specifying rules governing the correct use of a language. With these precise definitions, we can establish properties of all programs in a given language. One of the most important properties that we want to investigate and prove is safety, often referred to type soundness, establishing the coherence of the dynamic and static semantics and guaranteeing that well-typed programs are well-behaved and do not get stuck.

Traditionally [9][10], formalizations and proofs were written via pen-and-paper and the correctness could only be manually checked. As languages being studied become larger, it is increasingly impossible and error-prone to develop human-generated type safety proofs, as they often require checking thousands of cases. Some languages such as Standard ML, were formally defined [11] with early efforts [12] and later mechanized ones [13] on proving its soundness; however, most other languages such as Haskell are only believed to be sound. To solve these problems, much current research in programming languages is being done with the aid of automated proof assistants such as Coq, Isabelle/HOL, Twelf, Agda, NuPRL, where formalizations are written as code, and the proofs are constructed as rigorous logical arguments and their correctness are mechanically checked by the proof-checker. Formalization and proofs checked in this way are thus known as mechanized and formally-verified.

B. WebAssembly

WebAssembly is a safe, portable, low-level bytecode language and stack-based virtual machine designed for the increasingly sophisticated domain of Web applications. It offers compact representation, efficient validation and compilation, and low to no-overhead execution. It is the first new language on the Web since JavaScript’s long monopoly. Nowadays, a significant amount of JavaScript running on the Web is actually compiled from other languages. Although state-of-the-art JavaScript engines such as Google’s V8 have been optimized to be very fast, JavaScript is deficient as a compilation target due to its original design as a high-level scripting language and being memory-managed and dynamically-typed. One of the main goals of WebAssembly is to become a more appropriate and universal compilation target for the open Web platform.

WebAssembly is the first industrial-strength language or VM that has been designed from the start with a formal semantics, which is accomplished in lockstep and help with making the language semantics drastically simpler [3]. An initial WebAssembly design and specification was published in 2017 as a research paper appearing at PLDI, a premier venue for PL research. This paper provided a full-fledged formal specification of the WebAssembly language in a traditional programming language research manner, including the abstract syntax, typing rules, and operational semantics. The paper also claimed that the WebAssembly type system enjoys the standard soundness property, although no proof is pointed to in the paper. In 2018, Conrad Watt developed a mechanized specification and presented a machine-verified proof of the soundness in Isabelle, closely following the 2017 paper definition, together with a verified executable interpreter and type checker [4].
C. Motivations

The main motivations of our work are to mechanize the recently finalized W3C WebAssembly Core Specification 1.0 and to bring a WebAssembly formalization into the Coq community and support the existing projects there.

W3C WebAssembly Specification: WebAssembly has been standardized as an open W3C standard [1] and has recently (relative to the writing of this report) reached the final "Recommendation (REC)" phase on December 05, 2019 [14]. In addition to pseudo-code-like English prose, the standard specification also includes a complete set of formal semantics, using traditional textbooks approaches similar to the 2017 paper’s. However, it differs from the 2017 paper formalization in many detailed ways:

1) The step relation is defined with evaluation contexts that bring non-determinism.
2) Stack frames are made explicit and switched via evaluation contexts.
3) Allocations in the store and indirection through addresses are made explicit.
4) Some details are specified to correspond more closely with the actual byte encoding.
5) More structures are explicitly named and defined, reflecting more implementation details.
6) The store typing and extension (weakening) are defined differently.

We guess that most of the WebAssembly specification readers, e.g., engine developers, upstream compiler developers, and application developers would exclusively follow the published, standardized W3C specification. Thus our mechanization chooses to closely follow the official specification [1] rather than the 2017 paper definition [3]. We believe that doing so increases the verification strength and relevance of our work, and would be appreciated by potential certified software projects that might be built on top of our work.

Maintaining a mechanization against a living specification could be challenging. We find that it is a perfect time for mechanizing since we know that we are targeting a fixed specification that is guaranteed to be relevant for a long time. A dream is that our mechanized specification can be part of the creation of future WebAssembly standards; many new proposals of WebAssembly have been planned and not yet fully formalized. Our work could help with exposing issues and proving that new proposals do not break soundness while adding expressiveness. In fact, we have already adopted the multi-value proposal [15] to be a little bit more future-proof, and our work has found 1 small mismatch [16] in the core specification and several larger ones (e.g. [17]) in the multi-value extension of the specification. We believe that starting with the most up-to-date specification makes it easier to integrate new proposals that also start with the same base specification.

The Coq Community: Coq has become one of the most successful platforms for modeling programming languages with formally verified assurances and developing large-scale, in-production certified software systems, notably CompCert [5], Vellvm [6], CertiKOS [8], JSCert [7] and many others such as those in the DeepSpec program [18]. We also discovered that a machine-checkable formal semantics of WebAssembly in Coq has been requested by the WebAssembly community [19], and there hasn’t been any work being accomplished and published yet.

A WebAssembly mechanized formalization will open many interesting possibilities. For instance, one can build a certified compiler that translates higher-level programming languages or intermediate languages into WebAssembly, and formally prove that such translation preserves the semantics of the original programs. Great candidates are a certified C-to-WebAssembly compiler with CompCert, and a certified WebAssembly backend for LLVM with Vellvm. Similarly, a certified interfacing layer between WebAssembly and its embedder can be proved to be safe and correct with respect to the defined specifications. WebAssembly JavaScript Interfaces [20] has been standardized, and WASI, or WebAssembly System Interface [21] is a recent ongoing effort to push WebAssembly beyond web browsers to native applications running atop standard operating systems through interfaces such as POSIX.

We can see opportunities to integrate our mechanization with JSCert and CertiKOS here.

D. Summary

The contributions of this paper are:

- A mechanized formalization of the W3C WebAssembly Core Specification 1.0 in Coq;
- A (almost) completed mechanized soundness proof of WebAssembly’s type system.

The source code of this project can be obtained here [22].

The rest of this paper is structured as follows. In Section 2 we give a brief overview of the project, describing the structures, results, and challenges in general. From Section 3 to 5 we anatomize the WebAssembly language into structure, validation, and execution, where we have organized our mechanization and paper in the same order as their correspondents in the official specification. We believe that structuring our mechanization and paper in this order greatly encourages the reader to look at the official specification side-by-side with our mechanization and observe our eyeball-closeness. Although some of the terminology used in the standard is different from that used in programming language research, we nonetheless adopt the specification’s terminology for the same reason of making it easier to align the formalization with the specification. We will walk through a selected subset of our Coq mechanization, describe specific challenges we faced in each part, and compare it with related work as appropriate. Section 6 introduces the soundness theorem and discusses the preservation and progress lemmas in detail. Section 7 discusses related work and Section 8 draw some final conclusion and discuss future work.
II. Overview

A. Methodology

WebAssembly, as most other typed languages, is formalized as inductive tree structures, i.e. abstract syntax, corresponding to the Structure section of the official specification, with judgments and inference rules for operational semantics and static typing rules, corresponding to the Execution and Validation sections of the official specification, respectively. We faithfully mechanized those mathematical definitions into Coq code with three design principles: eyeball-closeness, correctness, and good engineering.

To develop the type soundness proofs, we needed to extend our typing rules to cover runtime structures such as results, stores, configurations, and administrative instructions that are not present in the source but are necessary for reasoning about the validity (or type) of these structures during the execution of WebAssembly. The official specification includes a subsection in the appendix for Soundness which helpfully provided most of these typing rules. This subsection also provide the statements of the preservation lemma, the progress lemma, and the soundness theorem in a mostly formal approach which helped us embarking our journey. Finally, we independently developed machine-checked proofs of preservation and progress lemmas, by induction on syntax or derivations, and the soundness theorem, as a corollary.

B. Project Structures and Results

The structure and results are visualized in Figure 1. We started our mechanization as strict correspondents of these four sections (and their subsections), but soon found the dependency relation is circular. We ended up structuring our mechanization as the mapping relations visualized in Figure 1, where the Value.v, Types.v, Int.v, Float.v, Numerics.v are extracted for clearer dependency relations and separation of concerns. This structure was heavily influenced by the official WebAssembly reference interpreter written in OCaml. Coq and OCaml have similar module systems that allow us to mirror the abstraction and avoid similar constraints quite smoothly. Another reason is that one of our future plans would require us to interoperate with the reference interpreter in these parts and having a isomorphic structure would be very beneficial there.

To give a rough idea of the scale of the WebAssembly language, the PDF version of the official WebAssembly specification is 144 pages long, and the four sections that we are particularly interested in are 67 pages long in total. N.B. The specification contains both English prose and formal notations in roughly one-to-one ratio so the pure formal subset that we mechanized would be roughly 34 pages. Our mechanization currently are around 5000 lines of Coq code, with roughly 2000 of them being the mechanized formalization that corresponds to the 34 pages, developed in roughly 2.5 man-weeks. The line counts in Figure 1 are larger because many of them contain lemmas as well. The remaining 3000 lines of code are proofs of lemmas and theorems. Our current proof developed in roughly 4 man-weeks does not yet cover the full mechanized definition. However, our proof is able to show that the progress property holds for the numeric subset, as well as the type preservation property holds for both numeric subset and structured control flow subset. Importantly, during the development of these proofs, we spent substantial times investigating and improving our design principles. We believe we have passed major technical challenges and our proof structures and infrastructures
have been well-established according to our design principles, and are capable of extending to the full language. We estimate that the line counts of the full proof would double the numbers of current proof.

We can compare the scale of WebAssembly with $\lambda^\rightarrow$ (the simply typed $\lambda$ calculus) that is frequently used in the textbooks. As we visualized in Figure 2, $\lambda^\rightarrow$ takes roughly 7 pages of definitions and 8 pages of proofs to establish type safety in *Types and Programming Languages* [9]. N.B. As a textbook, the ratio of English prose to formal notation is likely higher than that of the WebAssembly specification. *Software Foundation - Programming Language Foundation* [23] presents a mechanized definition of $\lambda^\rightarrow$ in less than 200 lines of Coq code and proves the soundness properties in under 300 lines of Coq. In conclusion, WebAssembly is roughly 10 times larger than the $\lambda^\rightarrow$. As for the informal/formal ratio, our mechanization demonstrates a comparable conciseness with the textbooks in the definition part of mechanization. The proof, however, doesn’t seem like it will grow linearly as the definitions.

**C. Design Principles and Challenges**

Challenges arise for each of the design principles we wanted to pursue.

*Eyeball-closeness:* Mechanization can be questioned on how faithful they can represent the official specifications. JSCert [7] advocated the approach of establishing the textual correspondence between the official specification and the mechanization for obtaining trust. We heavily adopted this approach, especially because the official WebAssembly specification has already invested on maintaining a set of formal semantics, which would drastically reduce the difficulties of eyeball assessment.

Our mechanization use the same naming and order as the specification whenever possible. We heavily leveraged the Coq’s syntax extensions and interpretation scopes [24] to be as close as possible to the formal notations used in official specifications. The challenges often come with the presence of syntax overloading in the official specification that is convenient for the specification authors, as a “slightly-informal” formal notations. While in Coq, we have to inconveniently make those cases explicit and still strive for a good-enough eyeball closeness.

*Correctness:* Correctness is arguably the most critical principles that a mechanization should achieve. Here we defined the correctness as "capturing all the important details precisely". We strive to be as precise as possible and there are two significant challenges.

The first is that the official specification, although notated formally, is still designed as less than a fully formal specification and is more likely to obscure facts and details, as compared to its original paper alternatives. This has been a major reason that related work [4] did not choose to follow it in a mechanization. Fortunately, the official specification authors have been very supportive to our work.

The second is that the official specification contained details that might not be relevant to its meta-theory thus could be safely omitted or generalized in the mechanization. For example, the underlying representation of integers, floating point numbers and memories are abstracted out as parameters. Another examples are the vectors, indexes, and addresses, which are bounded in the official specification, but generalized to be unbounded in our mechanization.

We also have future plans to further validate our correctness, as discussed in Section 8.

*Qualities of Engineering:* Qualities of engineering are sometimes overlooked when discussing mechanized proof because people are usually more interested in the result and (explicitly) not interested in looking into the actual code. We care about proof engineering because we expect our mechanization will grow as the official specification of WebAssembly grows, and we don’t want that growing to be so painful.

We focus on improving of our proof engineering from the aspects of scalability and maintainability: Lemmas that are reusable are extracted into the *ProofAux.v* module; proofs are decoupled into cases that would not interfere with siblings; duplicate proof routines are optimized as Coq’s *Ltac* tactics; code and proof are documented, *etc*. We also divide large modules so that the build process is more incremental and cache friendly. The challenge is that improving qualities of engineering takes time: additional time needed to be spent on refactoring working, but unsatisfactory (from a proof engineering standpoint), code.

**III. Structure**

WebAssembly has two concrete representations: a binary format and a text format. Both map to the same structure and are described by traditional *abstract syntax* in the standard. Our mechanization is concise enough to be squeezed into one page and is presented in Listing 1.

![Fig. 2. $\lambda^\rightarrow$ in textbooks](image)
A. Values

As a low-level language, WebAssembly operates only on primitive numeric values. However, according to their interpretation, the standard defines them into four classes:

- **Bytes** that are raw and uninterpreted;
- **Integers** with bit width \( N \) and whether they are signed or unsigned;
- **Floating-point** numbers in either 32 or 64 bits, mostly follow the IEEE754-2019 standard;
- **Names** and `chars`, defined by Unicode.

Since our focus is not the actual encoding of values, we abstracted out their underlying representations as `Parameter` objects in Coq. The `Int` and `Float` modules establish the interface of the operations that we expect the concrete implementations of integers and floating-point numbers to provide and export `I32.t`, `I64.t`, `F32.t`, and `F64.t` as abstract types and grouped as `val`.

```coq
Inductive valtype := T_i32 | T_i64 | T_f32 | T_f64.
Notation resulttype := (list valtype).
Inductive functype := FT (_ _: resulttype).
Notation "ts1 −→ ts2" := (FT ts1 ts2).
```

This grouping is inspired from the official WebAssembly reference interpreter written in OCaml. Because most numeric operations are **bounded-polymorphic** over these four values, such groupings can be extremely convenient for code reuses and concise mechanizations.
Instructions manipulate values on an implicit operand stack that WebAssembly is conceptually a stack machine where C. Instructions as well.

Subcategories:
- Constants
- Opcodes

Out of 171 would have to explicitly exclude those cases through extra language level to define only the valid combinations. We parameterizing bit width and signed/unsigned at the metasubcategories; the official specification hacked these by grouping are not fully regular over different value types. N.B. code reuses and eyeball-closeness; see Listing 3.

During the mechanization of this category we discovered a difference between the current specification and original paper definition where the current specification constrains control instructions to consume zero or more values from the stack and to produce at most one result. During the mechanization of this section. Control Instructions: During the mechanization of this category we discovered a difference between the current specification and original paper definition where the current specification constrains control instructions to consume zero or more values from the stack and to produce at most one result. To be a somewhat future-proof and to also have the convenience of greater generality, we adopted the multi-value proposal [15], which introduces block types that are expressed either by directly unfolding an optional value type (which we represent as Coq option) into a function type or taking an indirection through a type index. Another caveat is that the specification includes an explicit end opcode among the control instructions, which we ignored for now and went with the original paper definition. This will likely to change in the future.

B. Types

There are only four basic machine types, or value types in WebAssembly: integers and floating-point numbers, in either 32 or 64 bit width. Similar to other stack-based virtual machines, executing instructions in the WebAssembly VM consumes a sequence of such values from the stack as arguments and leaves a sequence of such values on the stack as the result. Sequence of values are classified as a result type, and a function type is defined as a pair of result types (argument and result). We take advantages of Coq’s syntax extension mechanisms to make the notation for function types close to the standard. In addition to the stack, other structures such as linear memory, tables, and globals in WebAssembly are typed as well.

C. Instructions

Instructions are the core of WebAssembly programs. Recall that WebAssembly is conceptually a stack machine where instructions manipulate values on an implicit operand stack, consuming (popping) argument values and producing (pushing) result values. The specification defines 171 individual opcodes [25], each corresponding to a dedicated instructions. We highlight two challenges we faced in this section.

Numeric Instructions: This category corresponds to 126 out of 171 opcodes, but they can be divided into only 6 subcategories: Constants, Unary, Binary, Tests, Comparisons, and Conversions. The standard uses conventions to occasionally write them in the grouped manner for convenience. Our mechanization mimics such conventions via module-level and term-level parameterization and grouping, again, inspired by the official WebAssembly reference interpreter, for both code reuses and eyeball-closeness; see Listing 3. N.B. These groupings are not fully regular over different value types and subcategories; the official specification hacked these by parameterizing bit width and signed/unsigned at the metalanguage level to define only the valid combinations. We would have to explicitly exclude those cases through extra definitions.

Control Instructions: During the mechanization of this category we discovered a difference between the current specification and original paper definition where the current specification constrains control instructions to consume zero or more values from the stack and to produce at most one value. To be a somewhat future-proof and to also have the

B. Types

There are only four basic machine types, or value types in WebAssembly: integers and floating-point numbers, in either 32 or 64 bit width. Similar to other stack-based virtual machines, executing instructions in the WebAssembly VM consumes a sequence of such values from the stack as arguments and leaves a sequence of such values on the stack as the result. Sequence of values are classified as a result type, and a function type is defined as a pair of result types (argument and result). We take advantages of Coq’s syntax extension mechanisms to make the notation for function types close to the standard. In addition to the stack, other structures such as linear memory, tables, and globals in WebAssembly are typed as well.

C. Instructions

Instructions are the core of WebAssembly programs. Recall that WebAssembly is conceptually a stack machine where instructions manipulate values on an implicit operand stack, consuming (popping) argument values and producing (pushing) result values. The specification defines 171 individual opcodes [25], each corresponding to a dedicated instructions. We highlight two challenges we faced in this section.

Numeric Instructions: This category corresponds to 126 out of 171 opcodes, but they can be divided into only 6 subcategories: Constants, Unary, Binary, Tests, Comparisons, and Conversions. The standard uses conventions to occasionally write them in the grouped manner for convenience. Our mechanization mimics such conventions via module-level and term-level parameterization and grouping, again, inspired by the official WebAssembly reference interpreter, for both code reuses and eyeball-closeness; see Listing 3. N.B. These groupings are not fully regular over different value types and subcategories; the official specification hacked these by parameterizing bit width and signed/unsigned at the metalanguage level to define only the valid combinations. We would have to explicitly exclude those cases through extra definitions.

Control Instructions: During the mechanization of this category we discovered a difference between the current specification and original paper definition where the current specification constrains control instructions to consume zero or more values from the stack and to produce at most one value. To be a somewhat future-proof and to also have the
A. Contexts

A necessary component for type checking is a typing context, defined in the Conventions subsection of the Validation section of the standard. Coq's Record feature provides optimized notation for accessing fields, but that is not sufficient to us. The official specification defined conventions for accessing and updating list items and record fields. We did our best to establish good-enough eyeball-closeness for all of these cases, as demonstrated in Listing 5.

and typing rules are presented in Listings 4 and 6. The Reserved Notation we introduced are included in the code samples below to introduce the notations for readers, but we omit the where clause that connect them with definitions for conciseness.

B. Types

Type declarations require little validation. There was restriction on function types where the arity of the returned type can not be larger than 1, but this is lifted by the multi-value proposal we adopted. So the only interesting checks are in the occurrence of limits and block types that we discussed.
in Section 3. Limits are validated by ensuring the requested amount is between the lower bound and the optional upper bounds. We give an example of instantiating this rule to validate table types in Listing 4; validating memory types is similar.

C. Instructions

Instructions are given function types that describe how they manipulate the operand stack, and we can validate (type check) programs that consist of sequences of instructions. We select some of them to present in Listing 4. Those instructions are easier due to being monomorphic:

- Binary operations consume two values and produce one; all values must have the same type, which is specified by the operator.
- Getting a local variable requires it to be defined in the context.
- Conditionals take an i32 value (interpreted as a boolean) and require the two branches to have the same block type.
- Function calls are typed according to the corresponding function definition in the context.

The more interesting ones are those that are polymorphic and there are two degrees of them:

- value-polymorphic: the value types of one or several individual operands is unconstrained. This is the case for all parametric instructions, including drop and select.
- stack-polymorphic: all or most of the function type is unconstrained. This is the case for all control instructions that performs a unconditional control transfer, e.g. br, br_table, unreachable, and return.

The rules for validating a sequence of instructions are defined (via mutual recursion) with the rules for validating a single instruction.

D. Modules

Modules need to be validated as well, by validating all of the components they contain. Most of these are a straightforward translation of the standard, so we only include here the typing rule for functions. We explicitly introduced rules for typing a sequence of components with same kind, e.g. the valid_funcs was introduced to more closely match the convenient notation used by the standard. The module typing rule is huge with 19 premises. We show here in Listing 6 only a specialized rule for typing a module comprised of only types, funcs and tables.

V. Execution

Execution is the most complicated part of the official specification. It has exposed most of the challenges during the mechanization phase of our work.

The execution behavior of WebAssembly is defined in terms of an abstract machine that steps between different program states, which includes a stack that records all the instructions and operand values and that we represented as a Coq list and an abstract store that consists of all instances of functions, tables, memories, and globals that have been allocated and that we represented as a Coq record. Those instances in the store are referenced through abstract addresses. Although a concrete implementation would likely to assign virtual memory addresses to those instances dynamically, the specification abstracted out these implementation details by defining addresses as simply indices of the respective store components, which we represented as unbounded nat.

The WebAssembly source program are represented as modules which need to be first instantiated before running. The result of instantiation, termed module instances, collects the instances of exports and addresses for other entities, corresponding to their respective declarations from the original module, in the same order as the static indices of module.

In conclusion, a module and its respective module instance are connected through static indices, while a module instance and its respective store are connected through indirectness with addresses. We have presented our mechanization of store and
module instances in Listing 7. N.B. The official specification includes the instantiation and allocation process but they are not necessary for the proof of type soundness. So they are safely omitted.

A. Runtime Structures

We selectively discuss other runtime structures that have raised interesting challenges.

Values: The specification reuses the same notation for values as for the const instructions. Such coincidence is convenient for the official specification but raises challenges for us — how should we model this? We explored two options:

1) A dependent pair of an instruction and an evidence that this instruction is a const instruction.

2) A dedicated definition for values, which could however be lifted into instruction as appropriate.

Both should work but we proceeded with the second option – the val definition we have presented in Section 3; the const instructions are simply defined as a thin wrapper of val. The benefits are that it works with other components of our mechanization coherently and it’s simpler. We tried to bridge gap of eyeball-closeness via Coq’s coercion mechanisms but it was more limited than we expected and was not able to satisfy us in most cases [26].

Results: Results could either be a sequence of values or a trap. This is one of the places that required us to be able to distinguish values from other instructions statically and our val definition has been capable for this purpose.

Administrative Instructions: In order to express the reductions of traps, function calls, and control flows, the syntax of instructions is extended in the specification. The two most interesting ones are the Label and Frame administrative instructions that model labels and frames "on the stack".

The specification syntactically extends the original set of instructions with the administrative instructions and implicit transforms the meaning of the instr syntactic class to be the extended instruction set from that point onward. Although this arrangement looks convenient and we could do the same in Coq as well, it would not able to exploit the fact that these are two distinct sets and that there is a clear separation in their use sites in the mechanization. Our mechanization thus chooses to keep instr closed and scoped in the use of structure and validation, and introduces another new admin_instr type with a constructor Plain that can lift plain instruction (instr) as a admin_instr to accomplish the set inclusion. The challenge that arises is how to keep a good-enough eyeball closeness. Since Coq’s coercion mechanisms could not satisfy our use cases [26], we introduced dedicated short prefix notation to visually mediate these problems.

Function Instances: Function instances have also exposed special challenges on our mechanization. The specification conveniently overload the funcinst syntactic class which is actually a sum of WebAssembly-native function instances and host function instances. Further complicating things, the use of the type field is overloaded as well. Our solution is presented in the Listing 7, where we carefully leverage both a coercion and a notation to create an illusion that the FI_type is overloaded across the two instances. An alternative solution would be using a dependent record but we found this simple solution is sufficient for such small, one-off use case.
Evaluation/Block Contexts: The specification heavily uses two contexts: *Evaluation contexts* and *Block contexts*. Mechanizing them required not just Inductive definitions but also Fixpoint for plugging those contexts. We haven’t introduced a complicated notation but have instead simply used short-handed names for those plugging functions. The structure of *Block contexts* by itself might be the single most interesting one from a type-theory perspective — it’s indexed by how many labels surrounding the hole and is dependently-typed. Fortunately, Coq is also a dependently-typed programming language and we can define such structure in the most precise manner. Our mechanization is given in Listing 8.

Configurations: The specification defines a configuration as a product of store and thread and defines a thread as another product of a frame and an instruction sequence. However, the actual stepping relation is defined on a unfolded form of configuration. In Coq, we have to perform such unfolding explicitly, so we introduced a special $ notation to unfold our config type as a 3-tuple, captured as S_F_instrs. The reason of having an explicit definition of a thread is both for the eyeball-closeness, and as a preparation to welcome the future threads proposal of WebAssembly [27].

B. Instructions

The execution of WebAssembly is performed by executing each individual instructions. The specification conveniently omits $ (store) and/or $ (frame) when they are not changed by the instruction. We discovered the operational semantics of
instructions can be roughly grouped as the *simple* ones, that do not change the store and frame at all, and the *full* ones, that either change store or frame. We showcase a selective subset of both cases in our mechanization in Listing 9.

**VI. Soundness**

The soundness property of WebAssembly implies its type safety and memory safety. Such properties are usually a requirement for any toy language in academic programming language research, but it is nontrivial for larger and industry-originated programming languages such as WebAssembly to have such property. Both the original paper and the official specification have stated these theorems (without proof); we discuss the development of our mechanized soundness proof in this section.

A. Extended Typing

The typing rules defined in the Validation section only cover the source portion of the WebAssembly language. With the introduction of runtime structures such as stores, instance, and administrative instructions in the Execution section, we need to add typing rules to describe them in order to reason about the full language. Another group of relations introduced are extensions, written as \( \leq \), which capture the properties that entries in stores and instances in stores must not be modified or removed during the lifetime of the program.

The Soundness subsection in the official specification helped to fully described all those necessary pieces that we could directly mechanize. Most of those rules are quite verbose and require us to be very careful to correctly capture all the details, but they did not introduce any significant new challenges.

B. Theorem Statements

The official specification includes a set of theorem statements for preservation, progress, and soundness in the end of the Soundness subsection. Those theorems are written in English with formal notation appearing within the parenthesis. Both the preservation and progress statements are fairly standard, so our mechanization is literally the same with only one exception. The progress theorems in the specification states what are *terminal* threads and configurations in English prose, while we have explicitly defined a relation \( \text{result-thread} \) that unpacks a thread and requires that its instruction sequence component is a result, so that we can define progress in Coq both concisely and with eyeball-closeness, without losing any degree of formalism, as presented in Listing 10.

The soundness theorem in the specification, however, is defined in an equivalent, but slightly unfolded or interpreted way. Fortunately, it’s well known that soundness is a corollary of preservation and progress lemma so we shall simply focus on showing these two properties in our mechanized proofs.

C. Theorem Proofs

There have been several challenges that were specific to the proving of the theorems. **Generality:** We started our proofs by directly proving the mechanized theorems that correspond directly to the specification theorems; we were able to show that both preservation and progress hold for the numeric subset of the WebAssembly language without too many problems. However, we got stuck after enriching the active subset of our mechanization to include control instructions. The major problem that we had was that the theorem statements are too specific, resulting in an induction hypothesis that was too weak to be useful. Our later investigations determined that the specification theorems are actually specialized to the typing rules for configurations, caring only about the cases of *top-level frame*, where there should be no return values and instructions of current thread are started in an empty stack. Such statements are too specific to yield an induction hypothesis that covers all of the intermediate program states during the execution.

As a result, we fully-generalized the theorem statements to strengthen the induction hypothesis and use it as the central proof, making the *top-level frame* case as a corollary. We give an example of the generalized preservation in Listing 10. Using these techniques, we are able to prove the structured control flow subset enjoys the preservation property as well. We estimated this approach would eliminate many issues once for all.

**Non-determinism and Polymorphism:** Many components of the WebAssembly languages are highly non-deterministic and polymorphic and make the proof trickier. We briefly discuss three sources of them:

1) **Evaluation contexts** are recursively defined and their decompositions are extremely flexible. There are always infinite numbers of ways to decompose an instruction sequence. There is usually only one possible choice of decomposition for each specific reduction rule, but the specific choice often requires manual effort to find. To make things worse, proving cases such as SC_E (Listing 9) that operate on an arbitrary evaluation context can be surprisingly challenging.

2) The **validity of instruction sequences** in the original paper definition is defined by three rules: one for \( \epsilon \) (empty sequence), one for composition, and one for weakening. The weakening rule is not syntax directed and make use of an arbitrary (unconstrained) sequence of types. The official specification merged this weakening rule into the previous two rules, which not only does not make this non-determinism better, but also makes the rules harder to use and obscures the truth. We ended up deriving the weakening rule from the merged rules.

3) **Stack unwinding and stack-polymorphism** are used for execution and validation of structured control flow. Consider an unconditional branch. During execution, as the SS_br rule presented in Listing 9, an unconditional branch will (temporarily) pop a number of values from the operand stack, then discard values and instructions (presented as a block context) as it unwinds the stack to the target label, and then push the previously popped values together with the target label’s continuation.
During validation, as the VI_br rule presented in Listing 4, an unconditional branch is stack-polymorphic, due to the values and instructions that will be discarded during stack unwinding. Such parameters need to be chosen carefully from the proof context to make sure the rules are invoked as appropriate.

Infrastructures: Some structures that are frequently used in the specification have required us to develop a substantial amount of lemmas and tactics for them. List "snoc" is the most representative example of that. Both the typing rules and operational semantics of WebAssembly are defined in a surprising "snoc" (the opposite of "cons") order. This is extremely anti-ergonomic as we are still using Coq's list type and most of the standard libraries are written for "cons" (standard) order. We had to develop our own rich set of lemmas and tactics for "snoc" order. Other examples include a Forall2 predicate used to establish correspondence between two (equal lengthed) lists; length-related reasoning on lists that is used whenever a static index is relevant; equality and distributive properties over lifting and injection, etc. There are yet others that are more specialized to the WebAssembly proof.

VII. RELATED WORK

A. Mechanized Programming Languages and Coq

Pen-and-paper specifications and proofs written in mathematical languages (also known as formal languages) are considered formal but their reliability has been questioned nowadays. Mechanization (that is, formal, machine-checked verification techniques), have elevated the standard of being formal in programming languages study to a whole new level — "formal proofs are code." More and more current research of or related to programming languages has adopted this new standard and applied it in a vast variety of projects. The number of them has developed beyond we can reasonably cover, so we only highlight those that are most closely related to our motivations.

Theorem preservation :

\[ \forall S \ S' \ F \ ainstrs \ ainstrs' \ ts1 \ ts2, \]
\[ \vdash S \ ok \rightarrow (S, F) \in C \rightarrow (S', F', ainstrs') \rightarrow (S', C) \vdash \text{a}* \ ainstrs' \in ts1 \rightarrow ts2 \rightarrow (S, F, ainstrs) \rightarrow (S', F', ainstrs') \rightarrow \]
\[ (S' \subseteq S'') \]

Proof. ... Qed.

Corollary preservation_toplevel :

\[ \forall S \ T \ S' \ T' \ rt, \]
\[ \vdash (S, T) \in rt \rightarrow (S', T') \in rt \rightarrow (S', T') \in rt \rightarrow (S, T) \vdash (S, T). \]

Proof. ... Qed.

In the CompCert project [28], Leroy [5] built a formally-verified optimizing compiler for a significant subset of C (later extending to almost all of the ISO C99 standard), pioneering the approach of using Coq proof assistant both for programming the software and for proving its correctness. To accomplish this job, each phase of the compiler had to be mechanized, including their C subset Clight [29] all the way to PowerPC assembly. Zhao et al. introduced Vellvm (verified LLVM) [6], which provides a mechanized formal semantics of the LLVM IR (intermediate representation), type system and properties for reasoning about programs expressed in LLVM IR and transformations that operate on it. Gu et al. developed JSCert [7], a mechanized formalization of the current ECMAScript 5 standard in Coq, where they advocate the eyeball-closeness of the textual correspondence between the official specification and the mechanization as one of the main methodologies for obtaining trust in a mechanized formalization; this work directly influenced our design principle.

We also shall mention that Coq is not the only tool for mechanizing and verifying programming languages. For instance, CakeML [31], a verified implementation of ML is a recent notable project from the Isabelle community. The central, common ideas of all these projects are the benefits of mechanized metatheory [32], [33] and deep specification [18]. Our project is an instance of following such trends and help to make these practices mainstream.

B. Mechanized WebAssembly

As we have discussed in Section 1, there have been three substantial efforts on formalizing WebAssembly: a preliminary specification [3] emphasizing formal semantics and aimed at the programming languages research community, a mechanized specification and machine-verified proof of soundness developed by Conrad Watt [4], and the W3C WebAssembly Core Specification 1.0 [1] aimed at a wide audience. Here, we
discuss the differences between Conrad Watt's mechanization and our mechanization in detail.

**On Proof Assistants:** The first significant difference is in the choice of proof assistant. Watt used Isabelle/HOL and we used Coq; both are among the most mature proof assistants available. In contrast with many relatively new proof assistants that have only been used in academia, they have both been used for many successful large-scale and production-quality formal verification projects that are relevant to both academia and industry. We assume that both proof assistants are correct and provide strong verification strength with no significant difference.

There are some technical differences in their underlying logics and development experiences. Isabelle is based on higher-order (classical) logic and Coq is based on the calculus of (co)inductive constructions, a type theory which can serve as both dependently-typed programming language and as a constructive logic (a.k.a., intuitionistic logic) and does not include the law of the excluded middle as an axiom. We found one structure in the WebAssembly specification that is dependently-typed — the block context that defined as labels indexed by depth — where Coq admits a slightly more convenient definition. Both logical systems are sufficiently expressive for our mechanization purpose without any problems. Isabelle and its precursor LCF are often classified as automated in which the main proof method is resolution and unification, while the main proof method in Coq is constructive proof. However, Coq includes automatic theorem proving tactics as an extension to bridge this gap.

**On Following the Specification:** The second difference is in the choice of specification that the mechanization mostly follows. Both Watt and we observe that the original WebAssembly paper differs from the W3C specification in their representation of certain specification artifacts; Watt was observing this on the 2017 draft version of the specification and we are observing this on the version 1.0 that was recently finalized in December 2019. In the future, we expect there will be even greater divergence from the original WebAssembly paper and the W3C WebAssembly specification.

As Watt mentioned in his paper, he tried to primarily follow the paper’s representations except for a few cases, since it was designed as a formal specification. We, in contrast, choose to follow the W3C WebAssembly specification, for the reasons of being more real-world and being able to potentially reach a wider audience. This decision does make our work harder in many significant ways. Essentially, more verbose and structured definitions and more explicit and detailed arrangements were required for the specification to be sufficiently concrete to be a useful guide for implementations and, simultaneously, to be sufficiently abstract so as to not constrain a implementation too much; this enlarged the scale of the definitions and obscured facts during the proofs of many theorems, and contributed to the relative unfriendliness of the W3C WebAssembly specification for mechanization.

**On Interoperability:** The third difference is about the interoperability of the mechanization. Unsurprisingly, this is closely related to the proof assistant being used. Proof assistants often resemble and/or include a programming language. In fact, type-theory-based proof assistants such as Coq, Agda, and Lean are often seen as a dependently-typed programming language whose type system happens to be expressive enough to be capable of expressing logical statements (and, therefore, whose well-typed terms are proofs of logical statements). When discussing interoperability, programming languages are often considered to be a soft boundary. Although not always, there are quite a lot of cases where multiple programming languages can communicate as part of the same system either through translations, or through a common runtime system, or through foreign function interfaces. At a minimum, software written in different programming languages can communicate through either internet protocols (e.g. HTTP) or through an operating system (e.g. inter-process communications or files).

Proof assistants, or proof-capable programming languages, however, are much weaker from the perspective of interoperability. To our knowledge, there have been no such foreign function interfaces yet developed that can communicate across proof assistants and still preserve the strength of verification. (There do exist approaches such as translation validation with verified validators [5] and proof-carrying code that allow the rebuilding of trust in some or most degree either by the reconstruction of proofs or via a certificate.) As a result, both Watt’s and our mechanizations are effectively scoped within their respective proof assistant’s community. Our mechanization could not be used by notable Isabelle projects such as CakeML or seL4, and neither could the the Isabelle mechanization be used by Coq projects.

**VIII. CONCLUSIONS AND FUTURE WORK**

This paper has described how we developed a trusted, mechanized formalization of the W3C WebAssembly Core Specification 1.0 in Coq. We have mechanized the formal semantics included in the specification including the structures (abstract syntax), validation (type system), and execution (operational semantics). Our mechanization aims to establish a eyeball-closeness with the official specification. We have also mechanized all extended typing rules for runtime structures required for our mechanized soundness proof. Our proof is able to show that the progress property and preservation property holds for a substantial subset of WebAssembly with respect to our mechanization. We believed we have passed major technical challenges and our proof structures and infrastructures are capable of extending to the full language.

It remains to be seen what impact could our work will make in the larger WebAssembly community. But we are optimistic about the relevance of and opportunities for our mechanization. It is unprecedented that four major Web browser vendors [3] can collaborate to bringing a new, low-level language and virtual machine that enables high performance applications on the Web without compromising the safety of the users. We have also seen WebAssembly being attractive to domains beyond just the Web [21]. The uses of and interests in WebAssembly has been growing fast and we expect this trend will
continue. As with other Web technologies, WebAssembly is standardized by W3C working groups. The version 1.0 of three W3C standards of WebAssembly have been recently finalized [14] and are still evolving. What is also unprecedented is the inclusion of a complete set of formal semantics in a W3C standard. Substantial efforts have been made to develop and maintain this formal semantics and the WebAssembly community group is further investing in this approach in all of the new proposals [27] including multi-value, reference types, tail call, threads, SIMD, etc.; supporting formal verification techniques and certified software developments are one the main goals of having these formal semantics.

We have many future plans. The most direct and obvious one is to complete the soundness proof. Then, we would be interested in enriching our mechanization to include active proposals [27], either by following the order of their status in standardization process, or by selecting those that are most interesting in terms of having impact on type soundness, e.g. reference types. Third, we would like to develop a reference interpreter and type checker, ones that are proved to be correct with respect to our mechanized specification, and that extracted from Coq to OCaml. This approach has been pioneered by the JSRef reference interpreter presented in the JSCert project [7] and it has also been adopted in Conrad Watt’s Isabelle mechanization, which cleverly reuse parts from the official WebAssembly reference interpreter written in OCaml [4]. Our mechanization has been developed with this possibility in mind and such executable artifacts can increase the practically of the project and allow the use of official testing suites to increase trust in our mechanization. Finally, we are, of course, interested in interoperating with all the existing projects within the Coq community to develop certified software, as discussed in Section 1.

ACKNOWLEDGMENT

This work was supervised by Prof. Matthew Fluet. We thank Andreas Rossberg for his support during this work.

REFERENCES
