August 2020

Mechanizing Webassembly Proposals

Jacob Richard Mischka
University of Wisconsin-Milwaukee

MECHANIZING WEBASSEMBLY PROPOSALS

by

Jacob Mischka

A Dissertation Submitted in

Partial Fulfillment of the

Requirements for the degree of

Master of Science

in Computer Science

at

The University of Wisconsin-Milwaukee

August 2020

ABSTRACT

MECHANIZING WEBASSEMBLY PROPOSALS

by

Jacob Mischka

The University of Wisconsin-Milwaukee, 2020
Under the Supervision of Professor John Boyland

WebAssembly is a modern low-level programming language designed to provide high performance and security. To enable these goals, the language specifies a relatively small number of low-level types, instructions, and language constructs. The language is proven to be sound with respect to its types and execution, and a separate mechanized formalization of the specification and type soundness proofs confirms this. As an emerging technology, the language is continuously being developed, with modifications being proposed and discussed in the open and on a frequent basis.

In order to ensure the soundness properties exhibited by the original core language are maintained as WebAssembly evolves, these proposals should too be mechanized and verified to be sound. This work extends the existing Isabelle mechanization to include three such proposals which add additional features to the language, and shows that the language maintains its soundness properties with their inclusion.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

Certainly, this work would not be possible without the foundational work by Conrad Watt. His work continues to be an inspiration, and his kind words of support about my project instills me with pride to this day.

I would also like to of course thank my adviser Dr. John Boyland, a phenomenal professor who inspired me to pursue proof mechanization and the study of programming language design in the first place, for his guidance, support, and needed nudging that enabled me to complete this project.

I would also like to thank my close friend Alison Saia for always supporting me and agreeing to proofread this work.

Finally, I must thank my family, my mother Deborah and sister Justine, for their everlasting and immense support of me and everything I do. I would be nowhere without them.

1 Introduction

The modern Web is based on three primary integrated components: document markup using HTML, styling using CSS, and interactivity using the JavaScript programming language. As the usage of the Web as a universal platform continued to expand, authors of software projects wished to enable their software to be run on the Web but various factors, some technical and others more superficial, prompted the demand of source languages besides JavaScript. Often, this was a lower-level language used for performance reasons, a stricter language for safety reasons, or another language with support for existing libraries or large projects that cannot easily be ported to JavaScript. To enable this, it became common for other programming languages to support JavaScript as a compile target, allowing developers to write software to be run on the Web without having to write JavaScript directly.

Some of these source languages were designed intentionally with this goal in mind, such as CoffeeScript[7], while others such as C or C++ had the capability added by external alternative compilers. The Emscripten[37] project compiles LLVM[26] intermediate representation software, which can be generated from C/C++, to JavaScript. The resulting compiled program utilizes a subset of JavaScript's available syntax and operations, called asm.js, which

is heavily optimized by JavaScript interpreters and just-in-time compilers, as well as the underlying hardware[31]. While this clever usage of the existing platform provided performance improvements and developer satisfaction, JavaScript was not designed as a low-level, performant compile target, and the resulting benefits of asm.js relied heavily on JavaScript runtime implementations. An attempt to define a formal specification and standardize the platform began in 2013, but was abandoned shortly thereafter in 2014[4]. The abandonment of asm.js's formalization coincided with discussions amongst its project leaders for defining a new "proper" bytecode solution for native code on the Web, followed shortly thereafter by initial discussions between browser developers and JavaScript runtime implementers about "WebAsm". As this project evolved and was fleshed out, the WebAssembly project began and was announced publicly on June 17, 2015[73].

2 WebAssembly

WebAssembly (Wasm) is a "safe, portable, low-level code format designed for efficient execution and compact representation"[64, p. 1]. The language is defined as a virtual instruction set, encapsulating the semantics of the program in a format that is independent of the underlying system on which the program runs. WebAssembly supports two official formats: a readable text format and a more efficient binary format[64]; both formats are based directly on the language's abstract syntax. The binary format is a space-efficient mapping of syntax expressions into a sequence of raw bytes, with each distinct sequence of bytes representing exactly one possible combination of syntax elements when decoded. The text format is largely similar to the grammar of the language's abstract syntax formatted into symbolic expressions (S-expressions).

WebAssembly provides safety guarantees by fully encapsulating logic and execution at a low level. Values consist of only 32- and 64-bit variants of integers and IEEE 754 floating point numbers[64, p. 2–3]. The operational semantics of the language are primarily that of a stack machine, wherein operations are performed by removing operands from a last-in-first-out stack of values and pushing the operation's result onto the stack afterward. Strictly

speaking this is not completely true, as the language's inclusion of local variables allows circumvention of completely stack-based execution[17]; this has led to challenges for implementers of Lightbeam, an optimizing streaming WebAssembly compiler[16]. Memory operations are fully sandboxed by the WebAssembly execution environment. WebAssembly memories are single lists of raw bytes[64, p. 16]. Each WebAssembly module can include a single memory vector, and any computations the module performs that require memory utilizes this list of bytes; no direct access of the underlying system's physical or virtual memory is permitted.

WebAssembly programs are executed by virtual machines, which provide an abstraction layer between the program and the underlying system. The connection between the host environment which runs the virtual machine (embedding environment) and the virtual machine executing the WebAssembly program is called an *embedder*. In order to interact with the host environment, WebAssembly embedders may expose functionality via application programming interfaces (APIs): defined functions that WebAssembly can use to interact with the environment. With the Web as the primary embedding environment, Web embeddings provide a defined JavaScript API that allows interaction with web pages.

Embedding environments are free to expose any level of functionality to

WebAssembly, enabled by the concept of *host functions* provided by the environment that can be invoked by WebAssembly. Host functions can receive and return values like regular functions, and they can also modify the *store*, the abstract model of the program's global state[64, p. 78]. WebAssembly itself provides no guarantees about the determinism of host functions—as far as WebAssembly is concerned, such functions can be assumed to succeed or fail arbitrarily at the whims of the embedding environment. If the function call succeeds, a compliant embedding environment is required to correctly remove the expected number of arguments from the stack, and return the declared number of values. If the store is modified, the resulting store must be an extension of the previous store, with only mutable contents modified and no contents removed, and its contents must be well-typed. Examples of host functions include functions for reading and writing system files or performing network operations in a non-Web environment.

WebAssembly instructions are grouped into several categories: numeric instructions which perform various mathematical operations on specific numeric value types, parametric instructions which can operate on any value type, variable instructions for manipulating local or global variables, memory instructions for manipulating the linear memory, and control instructions

for manipulating the program's control flow. Notable for a lower-level language, WebAssembly's control flow is fully structured: there is no rudimentary `goto`-like construct, and the flow of the program is manipulated only by higher-level instructions such as blocks and loops, if statements, and branch instructions. In addition to the *basic instructions* that are available to programmers, the specification also defines several *administrative instructions* which are used to express the reduction of various basic instructions. Administrative instructions cannot be called directly, but are internally used by the virtual machine to represent resulting intermediate states. As examples, the `trap` instruction represents the reduction of an erroneous state, and the `invoke` instruction represents the invocation of a function instance by its address in memory.

2.1 Non-Web Embeddings

As the name *WebAssembly* implies, its primary focus is Web-based software; though it is not solely tied to the Web. Other non-Web environments execute WebAssembly for a variety of reasons: its safe memory model, its high performance, and it being a universal, encapsulated, and independent compile target. The last is particularly interesting given its bidirectional implications: a programmer can truly write and compile a piece of software once and run

it nearly anywhere, and a programmer can depend on other software written in any language that supports WebAssembly.

WASI (WebAssembly System Interface) is a project that defines a family of APIs for WebAssembly to interact with a traditional system environment to perform common, core tasks such as interaction with the filesystem and networking[9]. These APIs are designed to be a common target for embedders to support, allowing optimal portability of systems-focused WebAssembly programs while maintaining the security and sandboxing of the WebAssembly platform[11]. WASI's `fd_readdir`, for example, is a host function that allows WebAssembly code to read the contents of a directory on the environment's filesystem.

2.2 Past Experiments

The concept of an alternative to JavaScript for running applications on the Web is not entirely new. Two previous notable examples, embedded Flash applications and Java Applets, were once-popular ways to write software that others could run in a web browser using other programming languages. While they are similar in concept to WebAssembly, there are several fundamental differences that allowed them to fall out of favor and eventually lead to the development of WebAssembly.

Firstly and primarily, both Flash and Java Applets were individual projects by single organizations, Adobe and Oracle respectively, that operated in web browsers through the use of plugins. They were not *part* of the standardized Web; they were instead isolated boxes that ran inside of web pages once a user installed a separate piece of software into their browsers that allowed them to work. Because they operated only using browser plugins, they are unsupported on popular mobile platforms such as Android and iOS. WebAssembly, by contrast, is developed in the open as a web standard—the direction of the project is not at the sole discretion of any one organization. WebAssembly virtual machines are integrated directly into modern browsers, requiring no additional action by the user to enable. If a given applet requires a newer version of the Java Runtime Environment (JRE) than is installed on the system, or if no JRE is installed at all, the user must download and install it. The Java 8 Runtime Environment installer currently is between 60 and 80 MB, depending on the platform. Performing this installation requires administrative privileges, preventing users on shared systems from running the applet if they are not able to perform the installation. While WebAssembly requires its own execution environment like Flash and Java Applets, on the Web this environment is intertwined heavily with existing JavaScript engines: the three most popular web browsers, Apple's Safari, Google's Chrome, and Mozilla's Firefox, each execute WebAssembly using their previously-existing

JavaScript engines[5, 8, 70].

Both Flash and Java Applets provided ways to interact with JavaScript and the web page, but the primary way for a user to interact with both were via the plugins' embedded objects within the page. This disconnect between the web page and the application resulted in a jarring disintegrated experience for users, and poor or nonexistent accessibility by default for screen readers. WebAssembly, by contrast, does *not* have its own way of interacting with the user—all interactions require utilizing the web embedding's host functions, defined by the WebAssembly JavaScript Interface[14] and Web-specific extensions provided by the Web API[15] and use JavaScript as an intermediary. While this may result in more work on the part of programmers, it results in a fully integrated experience by the user: something that is *part* of the Web, not something that is simply stuck inside the web page.

Finally, as previously mentioned, WebAssembly was designed with major goals of high performance and security. While Java applets and Flash applications can offer high performance in optimal circumstances, their reliance on software installed on the host system has security implications. Integrating with another piece of software introduces new potential vectors for attackers to exploit—the larger attack surface creates more opportunities for software bugs to result in security vulnerabilities[12, 1]. Security researchers[10] and

the United States Cybersecurity & Infrastructure Security Agency[38] suggest disabling such plugins in order to increase security.

Steve Jobs, CEO of Apple and the primary inventor of the iPhone, listed many of the above reasons for his refusal to provide support for Flash on iPhone devices[23]. This lack of support on popular mobile devices, along with the advancement of more powerful open and standardized Web platform features, resulted in Flash falling out of favor[18]. Flash was deprecated in 2017 by Adobe, with an end-of-life date of December 2020[2].

2.3 Evolving WebAssembly

The defined goals of the WebAssembly project detail the process for evolving the language: incrementally, after providing a minimum viable product (MVP) core language specification and implementation[60]. For the release of the 1.0 MVP, the main requirements of being a well-defined replacement for asm.js with distributable modules, efficient binary bytecode, and high performance were the focus[62]. Version 1.0 of the specification was tagged on July 20, 2019[53], and focus is now on adding features to increase the language's capabilities.

Development of the specification takes place in public GitHub repositories. While a large list of future features are at various levels of consideration

for post-MVP Wasm, the Community Group and Working Group utilize a phase-based proposal system for introducing, discussing, and implementing additional features for WebAssembly[59]. Proposals may introduce relatively minor behavior changes or describe major substantive modifications to the language; no proposal is too small to be considered, though larger proposals with greater surface area require more deliberation and agreement before being accepted.

For a feature to be adopted into the language, a proposal must be drafted which passes through a sequence of phases[65]:

0 **Pre-Proposal**: An individual contributor files an issue to present the idea, optionally adds the proposal to the proposal list, and a proposal champion or champions emerges who submit the proposal to the Community Group's biweekly agenda. The Community Group votes on general interest in further research and development of the proposal.

1 **Feature Proposal**: After being approved by the Community Group, the proposal is added to the proposal list if it has not been added already. A new repository is forked from the main WebAssembly specification, which is used to facilitate discussion, design, and specification of the feature.

2 **Proposed Specification Text Available**: Once the full proposed

English specification text is available in the proposal's repository, along with a reasonable community consensus, prototype implementations for the proposed features are created so that a test suite can be added.

3 **Implementation Phase**: After a satisfactory test suite is created and passes for the feature in some implementation, embedders implement the feature and integrate the changes into the reference WebAssembly interpreter.

4 **Standardize the Feature**: After Web VMs and toolchains implement the feature and the Community Group reaches consensus for the feature and its design choices, the feature is handed off to the Working Group for final stages of discussion and handling of edge cases. The Community Group is responsible for major feature changes and design decisions, so the proposal is essentially frozen once being passed to the Working Group.

5 **The Feature is Standardized**: Consensus is reached by the Working Group that the feature is complete.

In practice, in order to be deliberate about changes to the language, proposals can take a great deal of time before reaching phase 5 and being fully integrated into the WebAssembly specification; as of the time of this writing only five proposals have done so[45]. The first such proposal was

standardized relatively early in WebAssembly's life—prior even to the initial 1.0 release—with the Mutable Globals Proposal[48] on June 6, 2018[68]. The Multi-value Proposal, which was in phase 4 at the time of the original mechanization and already included in it, and the Non-Trapping Float-to-Int and Sign-Extension Operators proposals mechanized in this project were all integrated at once on March 11, 2020[69]. Finally, the JavaScript BigInt to WebAssembly i64 Integration proposal[58], a proposal which does not modify the core WebAssembly language itself but the JavaScript API, advanced on June 9, 2020[36]. There are 23 currently outstanding proposals: 6 in phase 3, 4 in phase 2, 9 in phase 1, and 4 in the pre-proposal phase 0[63].

Proposals may depend on one another: for example, a notable proposal for Interface Types adds functionality to describe high-level non-primitive values such as strings or records so that values of these types can be passed between WebAssembly modules[40]. Because these types are not of the four primitive value types supported by WebAssembly, there must be a way to refer to these constructs indirectly. Thus, the Interface Types proposal depends on the Reference Types proposal, which adds typed reference values for functions and other external types[52].

The WebAssembly community consists of a collection of groups focused on particular parts of the language and ecosystem. The two primary groups

involved with the development of the language and ecosystem are the Working Group and Community Group. The Working Group is an official World Wide Web Consortium (W3C) group consisting of the primary group of stakeholders in charge of steering the development of the language. The Working Group currently is comprised of 48 participants representing 14 organizations including American organizations such as Google, Apple, Intel, Facebook, and Microsoft, and organizations from other countries such as Tencent, LG, and Huawei[39]. The Chair of the Working Group is Ben Smith, of Google. The Community Group represents the broad community as a whole, and all community members involved with or interested in the development of the language are welcome; the only requirement is registering for a W3C account and agreeing to the terms of the group. The Community Group's meetings and discussions are facilitated primarily using a dedicated GitHub repository[61], and it hosts biweekly meetings via video conference. Meeting minutes and supplemental materials are posted afterward in the repository, and off-cycle discussion takes place in the repository's issue tracker.

In addition to the Community Group and Working Group, smaller subgroups target more specific facets of the language or associated platforms. As examples, there is the WASI subgroup for development of the WebAssembly System Interface[67], a Debugging subgroup for development of debugging tools[66], and subgroups for particularly large and critical proposals such as

the Interface Types, Garbage Collection, and SIMD proposals. Each subgroup determines its own meeting schedule and processes, though there are discussions of unifying them[51].

3 Proof Mechanization

Formal handwritten proofs continue to be the standard for showing that a given system holds certain properties, but they suffer from several drawbacks. They can be difficult to both read and write, because often they require maintaining a great deal of state of the problem space within one's head at a given time. Additionally, making adjustments to logical assertions or intermediate lemmas is challenging, as even a minor adjustment in one section of the proof may cause seemingly unrelated statements elsewhere to no longer hold. Finally, they can leave small hidden gaps of missing logic that are easily glossed over by reviewers, such as when something seems so *obviously* the case that no one bothers to question it, or when ambiguously phrased assertions result in potential logical errors.

Proof assistants aim to address these goals by providing a framework in which logical proofs can be encoded in a structured form and verified automatically by the proof assistant software. This encoding of a natural language proof to such a verifiable format is called *mechanization*. Proof assistants can be used to model a variety of logical or mathematical systems, though have received much attention for proving properties about programming languages in particular[6, 24, 41]; the (hopefully strict and well-defined) specifications

of such languages lend themselves well to translation into mechanized forms. Ideally, these mechanizations are also human-readable, and provide a high level of "eyeball closeness" to any natural language proofs and other source materials such as language specifications[6].

Many proof assistants exist, including Coq[72], Isabelle[30], Lean[29], and SASyLF[3]. In addition to the fundamental purpose of proof verification, many such tools assist proof writers by providing automated proof *tactics* or *methods* which attempt to automate the proof for a given assertion. While this can greatly reduce the tedium of both writing and reading proofs that apply logical steps individually, it also can result in confusion when such proof methods do not behave as expected. In writing, debugging such cases can be challenging due to nonexistent failure messages, and even cases of simply forgetting a rule when providing the proof method with the list of rules to apply are often unclear. When reading, automatically-proved assertions provide little context, requiring the reader to simply trust the method's claim about the assertion. This is particularly the case when certain tactics implicitly include lemmas or rules, which is possible in certain circumstances. In order to understand the details of the claim, one must deconstruct it either mentally when reading or by reducing it to smaller logical steps when debugging. As a result, such automation is useful, but its overuse can result in terse mechanizations that are difficult to understand when reading or

performing modifications.

4 Soundness

A programming language with a *sound* or *safe* type system provides certain guarantees that operations are only performed on values of the appropriate type. For example, it prevents one from mistakenly adding a number to a string of textual characters, or from attempting to access a nonexistent property of a record or object. Typically, a programming language's type system is considered sound when it exhibits the following two primary properties[19]:

1. *Preservation*: If an expression is well-typed and performs a step of evaluation, then the resulting expression maintains the original type.

2. *Progress*: If an expression is well-typed, it either has been reduced to a value, or it can be further evaluated.

It is important to note that just because a language is considered sound does not mean that every program written for it is entirely safe in all contexts. Firstly, the soundness of a language's specification provides no guarantee of the soundness of a particular *implementation* of the language; a mistake in a particular WebAssembly virtual machine may still result in undefined behavior. Secondly, while a type-safe programming language prevents programmers from making mistakes with respect to unexpected or unconsidered

possibilities regarding values and their types, it by no means ensures that all code is bug-free. As an example, consider an `add` function that accepts two numbers and returns one number. While strong typing and type safety ensures that it will only be operating on numbers as intended, it does not ensure that the function will properly perform the addition—the type system can only prevent errors involving types. Any number of other logical or operational errors can still arise: the addition operation in `add` may overflow the maximum value that can be expressed in the number type resulting in the incorrect value being returned, or complex conditional logic or even a character input mistake by the programmer may result in a subtraction being performed instead, for example. In all of such cases the types of the values are correct and sound, but the values themselves are not.

Finally, the safety of a language's type system makes no guarantees about the safety and security of the language itself. As an illustration of this, despite WebAssembly's type-safety and its design goal of security, vulnerabilities have been found primarily due to its simple linear memory model and the ability to escape the virtual machine sandbox using embedder host functions[25]. However, one such issue could have been prevented by a differently designed sound type system: that memory addresses are simply primitive 32-bit integers. If instead memory addresses had a dedicated type to themselves, functions designed to operate on integers and functions designed to

accept memory address pointers would be incompatible, preventing certain remote code execution attacks[25, p. 10].

Core WebAssembly's type system is proved to be sound via a natural language proof included in the specification[64, p. 140]. No additions will be allowed to the specification that knowingly violate WebAssembly's soundness guarantees. In order to confirm the soundness claims made by the rather terse proof, a mechanized proof of the full language specification was developed independently, using the Isabelle/HOL proof assistant[41]. Watt's mechanization of core WebAssembly took place separately from the development of the language, though it proved crucial in its foundation by identifying several major errors in the official WebAssembly specification. These issues were brought to the attention of the Working Group and fixed, resulting in foundational language features such as exception propagation and the `Return` operation to be rewritten[41, p. 60-61]. Additionally, the declared requirements of host functions, functions provided by the embedding environment that can be called by WebAssembly code, were discovered to be too weak to maintain type safety and rewritten as a result[41, p. 61].

In addition to proving soundness of the core language via progress and preservation, the original mechanization included several notable additional features. Firstly, it included one proposal not existing in the initial version

of the core language: the Multi-value Proposal[47]. In the original specification, functions and instructions can return at most one result. This restriction is removed by the Multi-value Proposal, allowing multiple return values for each[46]. Additionally, as well as the soundness proof of the core language, the initial mechanization included two separate modules which were also proven sound with respect to the mechanized specification: an executable type checker[41, p. 61] and an executable interpreter[41, p.62]. These additional modules are important because they provide a link between the mechanized, verified specification and actual existing code. By comparing the results of these modules to their official implementations by utilizing the WebAssembly test suite or existing real-world software, trust of correctness is established in the mechanization[6]. Trust that the mechanization is correct with respect to the specification strengthens the claims that the mechanization makes about the soundness of the specification itself.

Unfortunately, the mechanization of the language plays no role currently in steering WebAssembly development. To my knowledge, no work is being done to extend it, and new language features are not required to be mechanized before they are adopted into the standard. Aside from human checks by the Working Group, which are fallible for reasons mentioned in the previous chapter, no process currently exists to prevent proposals from causing errors making the type system no longer sound. While no requirement exists

currently, it would be useful to require proof that the type system is not adversely affected before a proposal can be finalized and adopted. The Isabelle mechanization, particularly when augmented with the language additions included with this work, is the best starting point for such a requirement and the adoption of a more rigid process for ensuring the type system stays intact.

5 Incremental Soundness

My project builds on the existing Isabelle proof, extending the initial mechanization to include extensions provided by three late-stage proposals: Non-trapping Float-to-Int Conversions[49], Sign Extension Operators[55], and Tail Call[56]. Definitions and proofs for the type checker and interpreter are also updated accordingly. The source code for my mechanization can be found on GitHub[28].

A great deal of the challenge in this task involved learning how to effectively use Isabelle. Prior to this project I was only vaguely familiar with the advanced features utilized in the original WebAssembly mechanization, despite being familiar with the core concepts of programming language proof mechanization as a whole with another proof assistant, SASyLF, and reading the official Isabelle tutorials and reference manuals. The SASyLF proof assistant is designed to be explicit: one must specify exactly which logical rules and assumptions are being applied each step in the process. Additionally, it is designed to be simple to use, with relatively few advanced features and logical constructs. By contrast, Isabelle both embraces implicitness and equips users with many advanced features and slightly nuanced ways to perform similar tasks.

An Isabelle proof can consist of several distinct but related types of syntax. The most common and recommended framework used in modern Isabelle proofs is Isabelle/Isar, which is modeled after human-readable logical proofs[71]. In addition to the structured Isar syntax, logical steps can be applied individually using the older *apply script* syntax. The Isar and apply script syntaxes can be interwoven, resulting in a jarring shift when encountered by a less experienced user. In addition to the two primary *inner* proof syntaxes which perform the specifications and proofs, Isabelle proofs must make use of another layer of *outer* syntax of Isabelle types and logical terms: an object-logic, which is most commonly Isabelle/HOL (higher-order logic). Additionally, logical rules can be defined using the Isabelle/Pure syntax. As a result, in order to become productive with Isabelle one must learn many different sublanguages; particularly so when diving into a large proof that makes extensive use of more advanced features of each.

Isabelle provides implicit definitions for a great deal of constructs, such as application rules for defined functions and possible outcomes of an analysis of possible cases. These definitions are named according to documented rules, though they often include numeric indices when dealing with collections of premises or a list of possible outcomes. Occasionally, these indices are the only or most convenient way to refer to a premise or rule in order to make use of it, leading to a given assertion being shown simply by `3`, for example.

While the meaning of such names can be shown using an Isabelle command, reading such a proof can quickly become confusing even to advanced users. Finally, these numeric indices can also cause issues when refactoring or adding to proofs, as in the case of my mechanization, where the definition of a new instruction type causes all subsequent instruction types to have their corresponding indices shifted accordingly. Thus, it was common for me to have to adjust a reference from `14` to `15` because of the newly-added rule `12`, for example.

At the outset of my mechanization, two of the newly-mechanized proposals, Non-Trapping Float-to-Int Conversions and Sign Extension Operators, were at phase 4. After the mechanized proofs were completed as part of this project, they (along with the Multi-value Proposal already present in the initial mechanization) have advanced to phase 5 and were fully integrated into version 1.1 of the WebAssembly specification as of April 9, 2020[35]. Tail Call remains a phase 3 proposal currently in the implementation phase[63].

5.1 Non-trapping Float-to-Int Conversions

The first of the three mechanized, this proposal introduces 8 new floating point truncation instructions of the form `i{32,64}.trunc_sat_f{32,64}_{u,s}`, the bracketed groups representing the selection of destination type, source

type, and signedness, respectively[50].

These new instructions are *saturating*, meaning that their results are limited to the maximum or minimum possible value for the given destination type. Non-saturating versions of these instructions were present in the initial release of WebAssembly, and were defined to trap in the event that the floating point number could not be represented in the target integer type's possible range of values, or in the event that the floating point number was the special "not a number" value as defined by the IEEE floating point standard (`NaN`). Instead of trapping, these new instructions return the minimum or maximum target integer value in case of underflow or overflow, and 0 in the event that the source floating point number is `NaN`, resulting in operations that never fail.

In the mechanization, numeric conversion is performed using a `cvt` conversion function which determines the correct operations for the types of the values passed to it, deferring to the correct `cvt_`*(value type)* function which inserts the appropriate instruction for the given conversion. Because some conversions between types are impossible, and some conversion operations can fail resulting in a trap, `cvt` returns an `Option` value indicating `Some` value if the conversion succeeded or `None` otherwise, indicating an impossible conversion, or a conversion operation that failed and trapped. At a high level, the new `cvt_sat` saturating conversion function behaves identically to

the existing `cvt` but simply fails in fewer cases; as a result the required additions were straightforward to model based on the existing definition. Because the new instructions do not trap, I initially set out to model the `cvt_sat` collection of operations so that they did not return an `Option` and simply returned a value in all cases. This was a mistake; in order for the new saturating conversion function to fit nicely into the existing mechanization, it must to be able to return `None` when the operation was impossible, such as converting to the same type as the original input type. After struggling to modify the surrounding proofs to handle impossible conversions elsewhere, reverting `cvt_sat` to return an `Option` like the non-saturating version, but which always returns `Some` value when performing a truncation, resulted in a straightforward addition of less resistance, maintaining the language's proof of soundness without requiring drastic changes to the existing theorems and lemmas. Overall, 172 lines were added and 176 lines were modified during the mechanization of this proposal, many of the modifications being minor numeric index adjustments accounting for the new rules.

5.2 Sign Extension Operators

Similarly to the previous proposal, the Sign Extension Operators proposal introduces several new numeric conversion operations which are similar to

existing instructions[54]:

- `i32.extend8_s`: sign extend the lower 8 bits of a 32-bit integer to the full integer width
- `i32.extend16_s`: likewise, but the lower 16 bits
- `i64.extend8_s`: sign extend the lower 8 bits of a 64-bit integer to the full integer width
- `i64.extend16_s`: likewise, but the lower 16 bits
- `i64.extend32_s`: likewise, but the lower 32 bits

Already present in the MVP release of WebAssembly are `i64.extend_s` and `i64.extend_u`[64, p. 11], for extending a 32-bit integer value (signed or unsigned, respectively) to a 64-bit integer value. At first inspection, these seem like the only extension operations required, as WebAssembly only supports 32- and 64-bit integers. These new instructions, by contrast, do not actually modify the type of the value, instead only its contents: `i32.extend8_s` receives an `i32` type as input and returns an `i32` type as output. These new instructions reinterpret the contents of a value, operating on smaller signed integer values packed inside the bits of one of WebAssembly's integer types. While such cases are unlikely when using pure WebAssembly, interaction with other environments or languages may result in such values existing.

The distinction is confusing. In fact, the distinction was even missed in initial discussions for this proposal, resulting in `i64.extend32_s` being dismissed as equivalent to the existing `i64.extend_s` instruction. However, `i64.extend_s` receives an `i32` type as input, returning an `i64` type. This contrasts with the behavior of the added instructions, which do not change the type of the input but instead reinterpret its contents. As such, `i64.extend32_s` receives an `i64` type containing the bits of a signed 32-bit integer and reinterprets it, without changing the container's type, to an `i64` type containing a signed 64-bit integer. This is indeed a distinct operation, and was hence added to the proposal afterward[54].

Of the three proposals added, Sign Extension Operators was the most straightforward to mechanize. Each of the new instructions were added, along with a new `extendsop` typing rule and `extendsop_i32` and `extendsop_i64` reduction rules. Each of the new rules receives an `ExtendS` argument, which is a new basic instruction variant representing either 8, 16, or 32-bit sign extension operation. Because `extend32_s` is only valid for 64-bit integers, the `i64` type had to be extended from its original `wasm_int` defined type to a newly-created `wasm_int64` type which contains the `extend32_s` instruction. After adding the new instructions and rules, a new `extendsop` application definition was created for each integer type. Finally, with the additions in place, all that remained was to account for the new possible instruction

types in several existing lemmas and theorems, and to update the interpreter and checker modules. The interpreter additionally required one new lemma stating that a sign extension operation always either succeeds or results in an error if the types are invalid, which Isabelle was able to prove trivially by case analysis. During the mechanization of this proposal 104 lines were added and 223 modified, many modifications being minor numeric index adjustments.

5.3 Tail Call

Like the previous two proposals mechanized in this project, the Tail Call proposal also does not introduce any completely new behavior at a high level. Core WebAssembly explicitly disallows tail-call optimizations for functions that return the result of a function call. To alleviate this and allow developers to opt into such optimizations, this proposal introduces a "return" version of each existing call instruction[34]:

- `return_call`, based on the behavior of the existing `call` instruction which calls a function directly
- `return_call_indirect`, based on `call_indirect` which calls a function via a reference to memory

These new instructions are defined to "unwind the stack" in the same way a call to the normal `return` instruction does: by clearing the current

execution frame and removing it, along with its local values, from the stack. Because the arguments to the called function are also a part of that frame, the new instructions first must remove those argument values, and replace them after popping the stack frame and before invoking the new function—behavior not possible using the existing instructions alone.

Semantically, the new instructions actually do very little themselves: `return_call` is defined almost identically to `call`, and `return_call_indirect` to `call_indirect`. The substantial addition is the `return_invoke` tail-invocation administrative instruction, which defines the execution semantics of the additional behavior. First, it validates that the function to be called exists, and that there are enough values on top of the stack to be used as its defined number of arguments. It then removes these arguments, ensures that there is a frame on the stack, and removes elements from the stack until the frame is cleared and popped itself. Finally, the arguments are returned to the stack, and the existing `invoke` administrative instruction is called to perform the execution of the new function.

While the behavior provided by Tail Call is not entirely new on a high level, the new operations interact with the environment in a nontrivial way by unwinding the stack. Of the three proposals, Tail Call was by far the most complicated to mechanize. Modeling it is relatively straightforward; the typing and execution semantics of the instructions are well-defined by

the specification. However, because of their control-flow nature, many properties had to be shown to still hold after the addition of the tail-invoke administrative instruction. Thankfully, `return_invoke`'s stack manipulation is similar enough to `return` and `break` that the latter two provided great starting points for the additions.

Unlike the two previous proposals, updating the interpreter after modeling the new instructions was not trivial. Because tail-invocation introduces new behavior at the completion of a step of evaluation, a new result step type, `RSTailInvoke`, had to be introduced. The existing returning result step type, `RSReturn`, was not sufficient as it only contains the list of returned values to the next step of evaluation, while `RSTailInvoke` contains the function to be invoked as well. This modeling, while determined independently, mirrors that used in the official proposal interpreter[32]. With the addition of this new result step type, its occurrence had to be handled at the end of a step of evaluation requiring new logic (though similar to that of the existing `RSReturn`) instead of mere updates, and its possibility had to be accounted for in various places throughout the interpreter and its soundness and completeness assertions.

Owing both to the amount of new behavior introduced, as well as the requirement for several helper lemmas to account for new language constructs, the amount of added code compared to modified code is significantly higher

than with the previous two proposals: 1117 lines were added and 132 modified.

While performing the modeling for the proposal's mechanization, I encountered a mistake in the proposed specification. Initially, after checking that the function exists and determining its type and number of arguments and return values, the specification mistakenly required that the stack contain at least as many values as the function is declared to return, instead of ensuring that there were enough values to use as arguments to the function. The subsequent behavior was defined correctly, as the number of arguments were removed, but the check would not ensure they existed beforehand. Thus, as initially written, the behavior was both overly restrictive (valid `return_invoke` calls could fail if the function returned more values than existed on the stack) and unsafe (functions with more arguments than return values could result in non-values being passed as arguments). This mistake was brought to the attention of the maintainers and was fixed shortly thereafter[27].

Despite the oversight's relatively minor size (only a single letter's difference), finding otherwise easily-overlooked mistakes is one of the primary strengths of mechanization in proofs as mentioned earlier in this document; while the human eye can easily disregard a small typo or gloss over something that seems obvious, proof assistants insist that everything be proved

without a doubt. This typo may have been caught by a reviewer before being fully adopted into the official language specification, though the statement existed in its incorrect state for over two years, since the proposal's initial authoring[33].

6 Related Work

Huang developed an independent mechanization of WebAssembly using the Coq proof assistant, and came close to proving its soundness[20]. Challenges involving control instructions, polymorphism, stack-unwinding, and infrastructural difficulties involving unintuitive stack ordering in certain contexts prevented the author from fully completing the correctness proof. Fortunately for myself, all of these challenges were already addressed in the Isabelle mechanization on which my work was based, and I was able to take advantage of these existing definitions when implementing the proposals. The author mentions plans to finish the correctness proof, implement an interpreter and type checker, and integrate proposal changes into the mechanization in order to prove their soundness as well, much like the Isabelle mechanization and my work does and plans to do.

Bodin et al. mechanized the JavaScript ECMAScript 5 standard in Coq, and created a reference interpreter proved correct with respect to the mechanization. Unlike WebAssembly with its small number of values and instruction set, mechanization of large-scale languages that are intended to be written directly are massive undertakings and significantly more challenging. Because of this, the JSCert mechanization focuses on the core aspects

of the language, omitting certain superficial niceties such as for-in loops as well as many standard library functions and libraries that can be considered derivable from core language features.

Jung et al. mechanized a core calculus of the Rust programming language using the Iris logic framework and an accompanying *extensible soundness proof* [24]. Rust is a modern systems language and one of the leading languages with support for compiling to WebAssembly. Notable features of the language include its novel ownership and borrowing lifetime model for references and their valid lifetimes, and its composable trait-based inheritance scheme. Like with JSCert, the full language is too massive of an undertaking to be implemented at once—in fact, no full language specification even exists to use as a starting point—so the authors reduced the language to a continuation-passing style language that includes the core lifetime features mentioned above named λ_{Rust}. Iris[21], a "Higher-Order Concurrent Separation Logic Framework, implemented and verified in the Coq proof assistant," provides built-in support for ownership reasoning, making it a fitting choice for a proof assistant for this task. Dang et al. extended this work, accounting for relaxed-memory operations in use by concurrent Rust programs[13].

Watt et al., the author of the original Isabelle WebAssembly mechanization on which my work is based, used the Alloy model checker[22] to find errors in JavaScript's specification causing concurrency issues and compilation

problems[44]. In particular, the authors show that JavaScript's concurrency model does not in fact support compilation to the ARMv8 scheme which is used in real-world applications without violating its specified guarantees, and also show that JavaScript's model does not guarantee the essential correctness condition of Sequential Consistency for Data-Race-Free (SC-DRF) programs. An amended version of the specification is proposed that fixes these errors, and mechanized in Coq with proofs of compilation and SC-DRF correctness.

7 Future Work

Certainly, much remains to be done in the area of WebAssembly soundness validation. With the three proposals included in this work, as of the time of this writing no other proposals which affect the language specification and its soundness have been fully adopted into the language. However, many proposals remain outstanding. In order to maximize the effectiveness of proof mechanization, such attempts should be performed before they are adopted in order to confirm that they do not adversely affect the soundness of the language's type system; though attempts to mechanize a proposal which is changing rapidly poses challenges in itself. Five proposals in phase 3 or greater are currently outstanding without a mechanized proof. Of particular importance are proposals with larger scope that affect core behavior of the system and its typing rules, such as the phase 4 Reference Types Proposal[52, 43] and phase 2 Threading Proposal[57, 43].

The mechanization's extensions have been approved by its original author, Watt, and plans are in place to integrate the changes into the official source and the WebAssembly entry in Isabelle's Archive of Formal Proofs[42].

8 Conclusion

This paper investigates the WebAssembly language and the soundness of its type system. Beginning with an existing mechanized proof of the core language using the Isabelle proof assistant, my project extends the mechanization to include three additions to the language, two of which have since been included into a new release of the official WebAssembly language specification, with the third remaining a proposed addition still under review and finalization.

Over the course of this project I have had the opportunity to join the WebAssembly community, learn about the language and its origins, and take part in Community Group and subgroup video conference meetings. Gaining understanding of a core standardized language for the Web during its infancy and becoming familiar with its evolution process will be of immense benefit to me as a web developer as the language and ecosystem matures. On a higher level, my efforts in understanding and modeling a language specification and its type system and proving its soundness have given me a deeper understanding of programming language design and type theory as a whole. Small details in an instruction's execution or typing rules often have large implications when the program is validated or executed.

Even for a rather simple language like WebAssembly with its four primitive types, small instruction set, and straightforward memory and execution models, issues can quickly arise and soundness can be lost if a language feature is modeled even slightly incorrectly—for example when asserting that an incorrect number of values are present. Ensuring soundness and proper execution is critical when designing a language so that it holds the properties that it claims to: the type system will prevent bugs and the language will do what the programmer tells it to. Despite the importance of this assurance, it is challenging to do so, and particularly difficult to ensure that it stays so when making changes to the language. Proof mechanization is one way to ease this burden, providing language authors a more foolproof way to keep their promises to their users.

Bibliography

[1] Adobe. *Security updates for Adobe Flash Player*. June 9, 2020. URL: https://helpx.adobe.com/security/products/flash-player.html (visited on 07/14/2020).

[2] Adobe Corporate Communications. *Flash & The Future of Interactive Content*. July 25, 2017. URL: https://theblog.adobe.com/adobe-flash-update/ (visited on 06/19/2020).

[3] Jonathan Aldrich, Robert J. Simmons, and Key Shin. "SASyLF: An Educational Proof Assistant for Language Theory". In: *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education*. FDPE '08. Victoria, BC, Canada: Association for Computing Machinery, 2008, 31–40. ISBN: 9781605580685. DOI: 10.1145/1411260.1411266. URL: https://doi.org/10.1145/1411260.1411266.

[4] *asm.js. Working Draft*. Aug. 18, 2014. URL: http://asmjs.org/spec/latest/ (visited on 04/12/2020).

[5] JF Bastien, Keith Miller, and Saam Barati. *Assembling WebAssembly*. June 6, 2017. URL: https://webkit.org/blog/7691/webassembly/ (visited on 06/28/2020).

[6] Martin Bodin et al. "A Trusted Mechanised JavaScript Specification". In: *SIGPLAN Not.* 49.1 (Jan. 2014), 87–100. ISSN: 0362-1340. DOI: 10.1145/2578855.2535876. URL: https://doi.org/10.1145/2578855.2535876.

[7] Geoffrey Booth. *CoffeeScript*. May 25, 2020. URL: https://coffeescript.org/ (visited on 07/15/2020).

[8] Mathias Bynens. *V8 JavaScript engine*. Oct. 25, 2019. URL: https://v8.dev/ (visited on 06/28/2020).

[9] Bytecode Alliance. *WASI: WebAssembly System Interface*. Nov. 12, 2019. URL: https://github.com/WebAssembly/meetings/blob/master/process/phases.md (visited on 04/19/2020).

[10] Matteo Campofiorito. *Pwn2Own 2010: interview with Charlie Miller*. Mar. 1, 2010. URL: https://web.archive.org/web/20110424022058/http://www.oneitsecurity.it/01/03/2010/interview-with-charlie-miller-pwn2own/ (visited on 07/13/2020).

[11] Lin Clark. *Standardizing WASI: A system interface to run WebAssembly outside the web*. Mar. 27, 2019. URL: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/ (visited on 06/03/2020).

[12] CVE Details. *Oracle JRE: List of security vulnerabilities*. July 30, 2019. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-93/product_id-19117/Oracle-JRE.html (visited on 07/14/2020).

[13] Hoang-Hai Dang et al. "RustBelt Meets Relaxed Memory". In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371102. URL: https://doi-org.ezproxy.lib.uwm.edu/10.1145/3371102.

[14] Daniel Ehrenberg, ed. *WebAssembly JavaScript Interface*. Version 1. W3C, Dec. 5, 2019. URL: https://www.w3.org/TR/wasm-js-api-1/ (visited on 07/01/2020).

[15] Daniel Ehrenberg, ed. *WebAssembly Web API*. Version 1. W3C, Dec. 5, 2019. URL: https://www.w3.org/TR/wasm-web-api-1/ (visited on 07/01/2020).

[16] Jack Fransham. *Introducing Lightbeam: An Optimising Streaming WebAssembly Compiler*. May 4, 2019. URL: http://troubles.md/lightbeam/ (visited on 04/19/2020).

[17] Jack Fransham. *WebAssembly Troubles part 1: WebAssembly Is Not a Stack Machine*. Jan. 30, 2019. URL: http://troubles.md/posts/wasm-is-not-a-stack-machine/ (visited on 04/19/2020).

[18] Andrew Griffin. *Adobe Flash to be killed off by 2020, killed off by the iPhone and new web technologies*. July 26, 2017. URL: https://www.independent.co.uk/life-style/gadgets-and-tech/news/adobe-flash-dead-2020-killed-off-iphone-html-5-steve-jobs-what-to-use-how-to-fix-not-working-a7860346.html (visited on 06/03/2020).

[19] Robert Harper. *Practical Foundations for Programming Languages, 2nd edition*. Cambridge University Press, 2016. ISBN: 9781107150300.

[20] Xuan Huang. *A Mechanized Formalization of the WebAssembly Specification in Coq*. Rochester, NY 14586, 2019. URL: https://www.cs.rit.edu/~mtf/student-resources/20191_huang_mscourse.pdf (visited on 07/14/2020).

[21] *Iris Project*. 2020. URL: https://iris-project.org/ (visited on 07/15/2020).

[22] Daniel Jackson. "Alloy: A Lightweight Object Modelling Notation". In: *ACM Trans. Softw. Eng. Methodol.* 11.2 (Apr. 2002), 256–290. ISSN:

1049-331X. DOI: 10.1145/505145.505149. URL: https://doi.org/10.1145/505145.505149.

[23] Steve Jobs. *Thoughts on Flash*. Apr. 2010. URL: https://www.apple.com/hotnews/thoughts-on-flash/ (visited on 06/03/2020).

[24] Ralf Jung et al. "RustBelt: Securing the Foundations of the Rust Programming Language". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158154. URL: https://doi-org.ezproxy.lib.uwm.edu/10.1145/3158154.

[25] Daniel Lehmann, Johannes Kinder, and Michael Pradel. "Everything Old is New Again: Binary Security of WebAssembly". In: (2020). URL: http://www.software-lab.org/publications/usenixSec2020-WebAssembly.pdf (visited on 07/03/2020).

[26] llvm-admin team. *The LLVM Compiler Infrastructure*. URL: https://llvm.org/ (visited on 07/27/2020).

[27] Jacob Mischka. *Question about tail-invocation stack value length assertion (item 3)*. Apr. 25, 2020. URL: https://github.com/WebAssembly/tail-call/issues/10 (visited on 05/11/2020).

[28] Jacob Mischka and Conrad Watt. May 17, 2020. URL: https://github.com/jacobmischka/wasm-isabelle (visited on 07/14/2020).

[29] Leonardo de Moura et al. “The Lean Theorem Prover”. In: (2015). URL: https://leanprover.github.io/papers/system.pdf (visited on 07/10/2020).

[30] Nipkow, Tobias and Paulson, Lawrence and Wenzel, Makarius and Klein, Gerwin and Haftmann, Florian and Weber, Tjark and Hölzl, Johannes. *Overview*. URL: https://isabelle.in.tum.de/overview.html (visited on 06/19/2020).

[31] John Resig. *Asm.js: The JavaScript Compile Target*. Apr. 3, 2013. URL: https://johnresig.com/blog/asmjs-javascript-compile-target/ (visited on 04/12/2020).

[32] Andreas Rossberg. *eval.ml*. Apr. 4, 2018. URL: https://github.com/WebAssembly/tail-call/b‘lob/master/interpreter/exec/eval.ml#L56 (visited on 07/11/2020).

[33] Andreas Rossberg. *Implement basic tail-call proposal*. Apr. 4, 2018. URL: https://github.com/WebAssembly/tail-call/commit/8e82d8c9f534b0d262c756677c0576e527eb840d (visited on 07/07/2020).

[34] Andreas Rossberg. *Tail Call Extension*. Dec. 13, 2019. URL: https://github.com/WebAssembly/tail-call/blob/master/proposals/tail-call/Overview.md (visited on 05/11/2020).

[35] Ben Smith. *Move 3 proposals from phase 4 to phase 5 (finished)*. Apr. 9, 2020. URL: https://github.com/WebAssembly/proposals/commit/f0ff21e00218bd050f479ef057085d9867f724da (visited on 05/11/2020).

[36] Ben Smith. *WebAssembly Working Group Mailing List Archive. Re: [WebAssembly Working Group]*. June 9, 2020. URL: https://lists.w3.org/Archives/Public/public-webassembly/2020Jun/0000.html (visited on 06/11/2020).

[37] The Emscripten Project. *About Emscripten*. 2020. URL: https://emscripten.org/docs/introducing_emscripten/about_emscripten.html (visited on 04/12/2020).

[38] United States Cybersecurity & Infrastructure Security Agency. *Securing Your Web Browser*. URL: https://us-cert.cisa.gov/publications/securing-your-web-browser (visited on 07/13/2020).

[39] W3C Team. *WebAssembly Working Group - Participants*. 2020. URL: https://www.w3.org/groups/wg/wasm/participants (visited on 06/12/2020).

[40] Luke Wagner. *Interface Types Proposal*. Apr. 15, 2020. URL: https://github.com/WebAssembly/interface-types/blob/master/proposals/interface-types/Explainer.md (visited on 06/11/2020).

[41] Conrad Watt. "Mechanising and Verifying the WebAssembly Specification". In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. Los Angeles, CA, USA: Association for Computing Machinery, 2018, 53–65. ISBN: 9781450355865. DOI: 10.1145/3167082. URL: https://doi.org/10.1145/3167082.

[42] Conrad Watt. "WebAssembly". In: *Archive of Formal Proofs* (Apr. 2018). http://isa-afp.org/entries/WebAssembly.html, Formal proof development. ISSN: 2150-914x.

[43] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. "Weakening WebAssembly". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360559. URL: https://doi.org/10.1145/3360559.

[44] Conrad Watt et al. "Repairing and mechanising the JavaScript relaxed memory model". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (2020). DOI: 10.1145/3385412.3385973. URL: http://dx.doi.org/10.1145/3385412.3385973.

[45] WebAssembly Community Group. *Finished Proposals*. June 10, 2020. URL: https://github.com/WebAssembly/proposals/blob/master/finished-proposals.md (visited on 06/10/2020).

[46] WebAssembly Community Group. *Multi-value Extension*. Feb. 17, 2020. URL: https://github.com/WebAssembly/multi-value/blob/master/proposals/multi-value/Overview.md (visited on 05/22/2020).

[47] WebAssembly Community Group. *Multi-value Proposal for WebAssembly*. May 4, 2020. URL: https://github.com/WebAssembly/multi-value (visited on 05/22/2020).

[48] WebAssembly Community Group. *Mutable Global Proposal for WebAssembly*. June 24, 2018. URL: https://github.com/WebAssembly/mutable-global (visited on 06/10/2020).

[49] WebAssembly Community Group. *Non-trapping float-to-int Conversion Proposal for WebAssembly*. Aug. 27, 2019. URL: https://github.com/WebAssembly/nontrapping-float-to-int-conversions (visited on 05/11/2020).

[50] WebAssembly Community Group. *Non-trapping float-to-int Conversion Proposal for WebAssembly*. Aug. 27, 2019. URL: https://github.com/WebAssembly/nontrapping-float-to-int-conversions/

blob/master/proposals/nontrapping-float-to-int-conversion/Overview.md (visited on 05/11/2020).

[51] WebAssembly Community Group. *Process, and meetings for subgroups*. Ed. by Deepti Gandluri. Apr. 27, 2020. URL: https://github.com/WebAssembly/meetings/issues/549 (visited on 06/12/2020).

[52] WebAssembly Community Group. *Reference Types for WebAssembly*. Ed. by Andreas Rossberg. June 5, 2020. URL: https://github.com/WebAssembly/reference-types/blob/master/proposals/reference-types/Overview.md (visited on 06/11/2020).

[53] WebAssembly Community Group. *Release wg-1.0*. Ed. by Andreas Rossberg. July 20, 2019. URL: https://github.com/WebAssembly/spec/releases/tag/wg-1.0 (visited on 05/29/2020).

[54] WebAssembly Community Group. *Sign-extension operators proposal for WebAssembly*. Apr. 28, 2019. URL: https://github.com/WebAssembly/sign-extension-ops/blob/master/proposals/sign-extension-ops/Overview.md (visited on 05/15/2020).

[55] WebAssembly Community Group. *Sign Extension Operators Proposal for WebAssembly*. Jan. 17, 2020. URL: https://github.com/WebAssembly/sign-extension-ops (visited on 05/11/2020).

[56] WebAssembly Community Group. *Tail Call Proposal for WebAssembly*. Apr. 27, 2020. URL: https://github.com/WebAssembly/tail-call (visited on 05/11/2020).

[57] WebAssembly Community Group. *Threading proposal for WebAssembly*. Ed. by Ben Smith. Apr. 17, 2020. URL: https://github.com/WebAssembly/threads/blob/master/proposals/threads/Overview.md (visited on 07/10/2020).

[58] WebAssembly Community Group. *WebAssembly BigInt<->i64 conversion in JS API*. June 1, 2020. URL: https://github.com/WebAssembly/JS-BigInt-integration (visited on 06/10/2020).

[59] WebAssembly Community Group. *WebAssembly Future Features*. Apr. 2020. URL: https://webassembly.org/docs/high-level-goals/ (visited on 04/12/2020).

[60] WebAssembly Community Group. *WebAssembly High-Level Goals*. Apr. 2020. URL: https://webassembly.org/docs/high-level-goals/ (visited on 04/12/2020).

[61] WebAssembly Community Group. *WebAssembly meetings. WebAssembly meetings (VC or in-person), agendas, and notes*. June 5, 2020. URL: https://github.com/WebAssembly/meetings (visited on 06/12/2020).

[62] WebAssembly Community Group. *WebAssembly Minimum Viable Product*. Apr. 2020. URL: https://webassembly.org/docs/mvp/ (visited on 04/12/2020).

[63] WebAssembly Community Group. *WebAssembly Proposals*. 2020. URL: https://github.com/webassembly/proposals (visited on 04/12/2020).

[64] WebAssembly Community Group. *WebAssembly Specification*. Ed. by Andreas Rossberg. Version 1. Apr. 4, 2020. URL: https://webassembly.github.io/spec/core/_download/WebAssembly.pdf (visited on 04/05/2020).

[65] WebAssembly Community Group. *WebAssembly W3C Process*. Apr. 2020. URL: https://github.com/WebAssembly/meetings/blob/master/process/phases.md (visited on 04/12/2020).

[66] WebAssembly Debugging Subgroup. *WebAssembly Debugging Subgroup*. Ed. by Derek Schuff. May 26, 2020. URL: https://github.com/WebAssembly/debugging (visited on 06/12/2020).

[67] WebAssembly Debugging Subgroup. *WebAssembly System Interface*. Ed. by Derek Schuff. June 8, 2020. URL: https://github.com/WebAssembly/WASI (visited on 06/12/2020).

[68] WebAssembly Working Group. *Meeting Notes for June 6 video call of WebAssembly's Working Group. Discussion on the status of the working draft*. Ed. by Ben Smith. June 6, 2018. URL: https://github.com/WebAssembly/meetings/blob/master/main/2018/WG-06-06.md#discussion-on-status-of-the-working-draft (visited on 06/11/2020).

[69] WebAssembly Working Group. *Meeting Notes for March 11 video call of WebAssembly's Working Group*. Ed. by Ben Smith. Mar. 11, 2020. URL: https://github.com/WebAssembly/meetings/blob/master/main/2020/WG-03-11.md#meeting-notes (visited on 06/11/2020).

[70] *Welcome to SpiderMonkey*. URL: https://mozilla-spidermonkey.github.io/ (visited on 06/28/2020).

[71] Makarius Wenzel. *The Isabelle/Isar Reference Manual*. Apr. 15, 2020.

[72] *What is Coq? The Coq Proof Assistant*. URL: https://coq.inria.fr/about-coq (visited on 09/07/2019).

[73] Alon Zakai. *History of WebAssembly. Chrome University 2019*. Feb. 7, 2020. URL: https://www.youtube.com/watch?v=6r0NKEQqkz0 (visited on 04/12/2020).