

Foundations of WebAssembly

Karthikeyan Bhargavan^{*1}, Jonathan Protzenko^{*2},
Andreas Rossberg^{*3}, and Deian Stefan^{*4}

- 1 INRIA – Paris, FR. karthikeyan.bhargavan@inria.fr
- 2 Microsoft – Redmond, US. jonathan.protzenko@gmail.com
- 3 München, DE. rossberg@mpi-sws.org
- 4 University of California – San Diego, US. deian@cs.ucsd.edu

Abstract

WebAssembly (Wasm) is a new portable code format with a formal semantics whose popularity has been growing fast, as a platform for new application domains, as a target for compilers and languages, and as a subject of research into its semantics, its performance, and its use in building verified and secure systems. This Dagstuhl Seminar brought together leading academics and industry representatives currently involved in the design, implementation and formal study of Wasm, to exchange ideas around topics such as formal methods for, verified compilation to, and verified implementation of Wasm.

Seminar March 5–10, 2023 – <https://www.dagstuhl.de/23101>

2012 ACM Subject Classification Software and its engineering → Formal language definitions; Software and its engineering → Virtual machines; Theory of computation → Semantics and reasoning

Keywords and phrases Compilation, Formal methods, Programming languages, Verification, Virtual machines, WebAssembly

Digital Object Identifier 10.4230/DagRep.13.3.1

1 Executive Summary

Andreas Rossberg

License  Creative Commons BY 4.0 International license
© Andreas Rossberg

WebAssembly – commonly known as Wasm – is a modern, portable code format and execution environment with a formal semantics that enforces safety and isolation. Though initially designed to run native, high-performance applications in Web browsers, Wasm is now used in many other applications domains – from CDNs to serverless, IoT, library sandboxing, and smart contracts. Wasm is one of the rare cases where practitioners are collaborating with the semantics and programming languages research community. This was exemplified by the initial design of Wasm itself, a collaboration with academia that culminated in a PLDI paper. The popularity of Wasm has since been growing exponentially as a platform for new application domains, as a target for compilers and languages, and as a subject of active scientific research – from its future semantics to its performance, and its use in building verified and secure systems.

This Dagstuhl Seminar brought together leading academics and industry representatives currently involved in the design, implementation and formal study of Wasm. It was a forum to exchange ideas that set new directions for WebAssembly research. The main focus was around three topics:

* Editor / Organizer



Except where otherwise noted, content of this report is licensed under a Creative Commons BY 4.0 International license

Foundations of WebAssembly, *Dagstuhl Reports*, Vol. 13, Issue 3, pp. 1–16

Editors: Karthikeyan Bhargavan, Jonathan Protzenko, Andreas Rossberg, and Deian Stefan



Dagstuhl Reports

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Formal methods for Wasm revolves around formalizing, reasoning and proving properties about Wasm itself. There are many WebAssembly extensions (e.g., bulk memory operations and vector instructions) which can benefit from formal semantics. Since Wasm is not a standalone language, there also is need to develop formal methods to reason about its interaction with the operating system, the execution of JITed Wasm code, etc. Finally, logics are needed that will allow us to formally capture interesting properties beyond what current work handles.

Verified Compilation to Wasm focusses on Wasm as a target of verified compilation toolchains. Wasm is positioned as a viable candidate for verified and secure compilation and we established that the clean design of Wasm offers greater simplicity when it comes to verifying a compilation toolchain – in particular, simpler and shorter proofs of compiler correctness and security.

Verified Compilation of Wasm studies the compilation of WebAssembly to native code, i.e., how to securely and correctly compile WebAssembly code to machine code. Wasm is growing rapidly, and is used on the Web and beyond (e.g., embedded systems, edge computing, IoT, and even OS kernels), and across different platforms and toolchains.

One particularly noteworthy result of the seminar was the birth of a new project that resulted in a collaboration between various participants of the seminar: to create a domain-specific language (DSL) for authoring the official Wasm specification. This project will enable creating a single source of truth for generating both the formalism and the alternative prose description in the standard, as well as transformations to representations in various theorem provers or executable reference interpreters that process the Wasm semantics for formal methods.

2 Table of Contents

Executive Summary

| | |
|-----------------------------------|---|
| <i>Andreas Rossberg</i> | 1 |
|-----------------------------------|---|

Overview of Talks

| | |
|--|----|
| RichWasm: Bringing Shared Memory Interoperability to WebAssembly <i>Amal Ahmed</i> | 5 |
| Wasocaml: compiling OCaml to WebAssembly <i>Léo Andrès</i> | 5 |
| WebAssembly Diversification for Malware Evasion <i>Javier Cabrera Arteaga</i> | 6 |
| From Dynamic to Static Symbolic Execution for WebAssembly <i>José Fragoso Santos</i> | 6 |
| WasmCert-Coq: A Mechanised Specification of WebAssembly <i>Philippa Gardner</i> | 7 |
| Iris-Wasm, a mechanized separation logic for WebAssembly <i>Aïna Linn Georges</i> | 7 |
| Flexible and Secure Hardware-Assisted Wasm with HFI <i>Shravan Narayan, Evan Johnson, and Deian Stefan</i> | 8 |
| Let's Go Coroutine <i>Luna Phipps-Costin and Daniel Hillerström</i> | 8 |
| Wasm 2.0, 2.1 and beyond <i>Andreas Rossberg</i> | 9 |
| How to design, document, and implement programming languages <i>Sukyoung Ryu</i> | 9 |
| Wanilla: Sound Automated Horn-clause-based Noninterference Analysis for WebAssembly <i>Markus Scherer</i> | 10 |
| That's a Tough Call! Studying the Challenges of Call Graph Construction for WebAssembly <i>Michelle Thalakottur, Daniel Lehmann, Michael Pradel, and Frank Tip</i> | 10 |
| WebAssembly as the Basis of All Things? <i>Ben L. Titzer</i> | 10 |
| Verifying Instruction Selection in a Wasm-to-native Compiler <i>Alexa VanHattum</i> | 11 |
| MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code <i>Marco Vassena</i> | 11 |
| The Path to Components <i>Luke Wagner</i> | 12 |
| Usefully Mechanising All of WebAssembly <i>Conrad Watt</i> | 12 |


Working groups

| | |
|---|----|
| A DSL for writing the WebAssembly Specification <i>Andreas Rossberg, Joachim Breitner, Pierre Chambart, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Sukyoung Ryu, Luke Wagner, Conrad Watt, and Dongjun Youn</i> | 12 |
| Participants | 16 |

3 Overview of Talks

3.1 RichWasm: Bringing Shared Memory Interoperability to WebAssembly

Amal Ahmed (Northeastern University – Boston, US)

License  Creative Commons BY 4.0 International license
© Amal Ahmed

Joint work of Amal Ahmed, Michael Fitzgibbons, Zoe Paraskevopoulou, Michelle Thalakottur, Noble Mushtak, Jose Sulaiman Manzur

Though Wasm provides a safe, sandboxed environment for programs to run in, it lacks the facilities to enable safe, shared-memory interoperability between Wasm modules, a feature that we believe is essential for a low-level language in a multi-language world. I'll present RichWasm, a higher-level version of WebAssembly with an enriched capability-based type system to support fine-grained type-safe shared-memory interoperability. RichWasm is rich enough to serve as a typed compilation target for both typed garbage-collected languages and languages with an ownership-based type system and manually managed memory. RichWasm takes inspiration from earlier work on languages with linear capability types to support safe strong updates, and adds analogous unrestricted capability types for garbage-collected locations, allowing a module to provide fine-grained memory access to another module, regardless of memory-management strategy. RichWasm types are not intended to be made part of core Wasm; instead we compile RichWasm to core Wasm, allowing for use in existing environments. We have formalized RichWasm in Coq and are currently proving its safety via progress and preservation.

3.2 Wasocaml: compiling OCaml to WebAssembly

Léo Andrès (University Paris-Saclay – Orsay, FR)

License  Creative Commons BY 4.0 International license
© Léo Andrès

Joint work of Léo Andrès, Pierre Chambart

OCaml is a rich programming language. It is comprised of a lot of advanced functional and imperative features while allowing low-level manipulations. Our talk will begin with a description of the value representation technique as well as the memory layout used by the OCaml runtime. We will then examine the distinctions between various intermediate representations in the OCaml compiler, and then justify the selection of Flambda as a source language. Additionally, we will present our translation process from Flambda to wasm-gc, with a particular focus on the encoding of small scalars, heap-allocated blocks and functions closures. To top it off, we will provide a comparative analysis of our compiler against the alternatives, based on informative benchmarks.

3.3 WebAssembly Diversification for Malware Evasion

Javier Cabrera Arteaga (KTH Royal Institute of Technology – Stockholm, SE)

License © Creative Commons BY 4.0 International license
© Javier Cabrera Arteaga

Joint work of Javier Cabrera Arteaga, Martin Monperrus, Benoit Baudry

Main reference Javier Cabrera-Arteaga, Martin Monperrus, Tim Toady, Benoit Baudry: “WebAssembly Diversification for Malware Evasion”, CoRR, Vol. abs/2212.08427, 2022.

URL <https://doi.org/10.48550/arXiv.2212.08427>

WebAssembly is an important binary format that has become an integral part of the modern web. This technology offers a faster alternative to JavaScript in web browsers, but it has also been utilized for cryptojacking since its inception. To counter this threat, considerable efforts have been made to develop defenses that can detect WebAssembly malware. However, these defenses have not taken into account the possibility that attackers may use complex evasion techniques. We explore how to evade detection by WebAssembly cryptojacking detectors. We propose a technique that uses `wasm-mutate`, a fuzzing tailored tool of `wasmtime`, to create variants of the original code that can evade the detectors and demystify the previous assumption. To evaluate our technique, we used VirusTotal. Our results demonstrate that our approach swiftly generates WebAssembly cryptojacking variants that evade detection, while the generated WebAssembly binaries show only minimal performance overhead. Our experiments also provide valuable insights into which WebAssembly code transformations are best suited for evading malware detection. This knowledge can be used to improve the state of the art in WebAssembly malware detection, which will benefit the wider community. Although our technique exposes weaknesses in detection mechanisms, it also serves as a valuable tool for testing other systems using WebAssembly as an input, e.g. compilers, validators and verification tools.

3.4 From Dynamic to Static Symbolic Execution for WebAssembly

José Fragoso Santos (INESC-ID – Lisbon, PT)

License © Creative Commons BY 4.0 International license
© José Fragoso Santos

Joint work of José Fragoso Santos, Filipe Marques, Nuno Santos, Pedro Adão

We present WASP, a configurable symbolic execution engine for analysing Wasm modules. WASP works directly on Wasm code and is built on top of the official Wasm reference interpreter. One key advantage of WASP compared to other symbolic execution engines is that it is highly configurable, supporting various flavours of symbolic execution and exploration strategies of the program’s state space.

Using WASP, we created WASP-C, a new symbolic execution framework for testing C programs. WASP-C was used to symbolically test a generic data-structure library for C and the Amazon Encryption SDK, demonstrating that it can find new bugs and generate high-coverage testing inputs for real-world C code. WASP-C was further tested against the Test-Comp 2022/2023 benchmarks, obtaining results comparable to well-established symbolic execution and testing tools for C.

3.5 WasmCert-Coq: A Mechanised Specification of WebAssembly

Philippa Gardner (Imperial College London, GB)

License  Creative Commons BY 4.0 International license
© Philippa Gardner

Milner pioneered formal language specification, proving hand-written correctness results about type safety and module instantiation. His work led to many formal then mechanised specifications including the large Coq-mechanisation of JavaScript, JSCert [1], developed at Imperial and Inria. Most of these large mechanised specifications were developed long after the language standards had been essentially settled. The challenge now is to establish mechanised language specification within the language standardisation process.

The W3C WebAssembly (Wasm) language is the first programming language to have a formal standard as envisaged by Milner. Inspired by JSCert, Gardner and Watt developed the mechanised specification of the Wasm 1.0 standard, WasmCert-Coq and WasmCert-Isabelle [2]: crucially, Watt fixed errors in the specification and type-safety result before the Wasm draft publication [3], adapting ideas from JSCert; correctness of module instantiation was proved in WasmCert.

In Conrad’s Dagstuhl talk, he will present WasmRef-Isabelle [4], an efficient certified reference interpreter for Wasm 1.0 supported by the ByteCode Alliance. In this talk, I will present WasmCert-Coq and explore how to define a certified reference interpreter, WasmRef-Coq, in such a way that the definitions and correctness proofs might have a better chance to keep up with the evolving standard in future.

References

- 1 Martin Bodin, Arthur Charguèraud, Daniele Filaretti, Philippa Gardner, Sergio Maffei, Daiva Naudžiūnienė, Alan Schmitt and Gareth Smith. *A Trusted Mechanised JavaScript Specification*. Principles of Programming Languages (POPL), 2014
- 2 Watt, Rao, Pichon-Pharabod, Bodin and Gardner. *Two Mechanisations of WebAssembly 1.0*. Formal Methods (FM), 2021
- 3 Conrad Watt. *Mechanising and Verifying the WebAssembly Specification*. Certified Programs and Proofs (CPP), 2018
- 4 Watt, Trela, Lammich, Märki. *WasmRef-Isabelle: a Verified Monadic Interpreter and Industrial Fuzzing Oracle for WebAssembly*. Programming Language Design and Implementation (PLDI), 2023

3.6 Iris-Wasm, a mechanized separation logic for WebAssembly

Aina Linn Georges (Aarhus University, DK)

License  Creative Commons BY 4.0 International license
© Aina Linn Georges

Joint work of Xiaojia Rao, Aina Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, Lars Birkedal

Main reference Xiaojia Rao, Aina Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, Lars Birkedal: “Iris-Wasm: Robust and Modular Verification of WebAssembly Programs”, Proc. ACM Program. Lang., Vol. 7(PLDI), Association for Computing Machinery, 2023.

URL <https://doi.org/10.1145/3591265>

Iris-Wasm is a mechanized separation logic that we have developed and used to practically verify both individual Wasm programs and properties of Wasm itself. In this talk, we will explore how Iris enables the specification and verification of individual modules separately,

which can then be combined modularly to reason about complex programs. Additionally, we will demonstrate how Iris enables the verification of functional correctness in WebAssembly programs, even when they interact with unknown or adversarial code, demonstrating the promise of WebAssembly’s module isolation. Iris is a rich and expressive higher-order separation logic that provides a powerful toolset for program verification. By successfully instantiating the full language standard into Iris, we can, going forward, leverage its numerous applications, such as exploring weak memory, robust safety, capabilities, effects, secure compilation, garbage collection, and more. Thus, we open up many exciting prospects for the verification of WebAssembly programs, and create an expressive foundational support for the WebAssembly ecosystem. In this talk, I will present a high level overview of Iris-Wasm, demonstrating the kind of verification it enables and providing a practical demonstration of what a proof in Iris-Wasm looks like.

3.7 Flexible and Secure Hardware-Assisted Wasm with HFI

Shravan Narayan (University of California – San Diego, US), Evan Johnson (University of California – San Diego, US), and Deian Stefan (University of California – San Diego, US)

License © Creative Commons BY 4.0 International license
© Shravan Narayan, Evan Johnson, and Deian Stefan

Joint work of Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, Dean Tullsen, Deian Stefan

Main reference Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, Dean M. Tullsen, Deian Stefan: “Going beyond the Limits of SFI: Flexible and Secure Hardware-Assisted In-Process Isolation with HFI”, in Proc. of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023, pp. 266–281, ACM, 2023.

URL <https://doi.org/10.1145/3582016.3582023>

In this talk, I will introduce Hardware-assisted Fault Isolation (HFI), a simple extension to existing processors to support secure, flexible, and efficient in-process isolation. HFI addresses the limitations of existing software-based isolation (SFI) systems including: runtime overheads, limited scalability, vulnerability to Spectre attacks, and limited compatibility with existing code. HFI can seamlessly integrate with current SFI systems (e.g., WebAssembly), or directly sandbox unmodified native binaries. To ease adoption, HFI relies only on incremental changes to the data and control path of existing high-performance processors. I will also cover our evaluation of HFI for x86-64 using the gem5 simulator and compiler-based emulation on a mix of real and synthetic workloads

3.8 Let’s Go Coroutine

Luna Phipps-Costin (Northeastern University – Boston, US) and Daniel Hillerström (Huawei Technologies – Zürich, CH)

License © Creative Commons BY 4.0 International license
© Luna Phipps-Costin and Daniel Hillerström

Joint work of Arjun Guha, Daan Leijen, Sam Lindley, Matija Pretnar, Andreas Rossberg, KC Sivaramakrishnan

Non-local control flow features provide the ability to suspend the current execution context and later resume it. Many industrial-strength programming languages feature a wealth of non-local control flow features such as `async/await`, coroutines, generators/iterators, effect

handlers, call/cc, and so forth. For some programming languages non-local control flow is central to their identity, meaning that they rely on non-local control flow for efficiency, e.g. to support massively scalable concurrency. Currently, WebAssembly lacks support for implementing such features directly and efficiently without a circuitous global transformation of source programs on the producer side. During this talk we will introduce WasmFX, an extension of Wasm with effect handlers for handling non-local control-flow in a structured manner. We will demonstrate WasmFX by example by compiling and running some coroutine programs live (uh-oh). We will also discuss the implementation of WasmFX in wasmtime, and the future directions that we are looking to explore.

3.9 Wasm 2.0, 2.1 and beyond


Andreas Rossberg (München, DE)

License  Creative Commons BY 4.0 International license
© Andreas Rossberg

Since the release of Wasm 1.0, many formal proposals for language extensions have been and are still being developed. A first batch has been included as part of Wasm 2.0, another one is soon expected with Wasm 2.1. I gave an overview of the extensions already adopted, those nearing completion, and those still under active development, and briefly touched on the wider implications they might have on Wasm's semantics. I also explained the proposal process itself and its requirements, for those who are not following the Wasm CG closely.

3.10 How to design, document, and implement programming languages

Sukyong Ryu (KAIST – Daejeon, KR)


License  Creative Commons BY 4.0 International license
© Sukyong Ryu

Since 2015, the JavaScript language has rapidly evolved with a yearly release cadence and open development process. However, it results in the gap between the language specification written in English and tools, such as parsers, interpreters, and static analyzers, which makes language designers and tool developers suffer from manually filling the gap. JISET and its extensions lessen the burden by automatically extracting a mechanized specification from the language specification in prose.

We introduce several tools in the JISET family and show how they fill the gap between the language specification and tools. We then discuss how to apply this technique to WebAssembly.

3.11 Wanilla: Sound Automated Horn-clause-based Noninterference Analysis for WebAssembly

Markus Scherer (TU Wien, AT)

License  Creative Commons BY 4.0 International license
© Markus Scherer

Joint work of Jeppe Fredsgaard Blaabjerg, Markus Scherer, Magdalena Solitro, Alexander Sjösten, Matteo Maffei

Noninterference is an important information flow property that can be formalized as 2-safety-property. In this talk we will explore how to leverage horn-clause-based abstractions to overapproximate it as a reachability property. Having done so, we can use solvers for SMT's Constrained-Horn-Clause fragment to assess noninterference in WebAssembly, a compilation target widely used in the real world. Our approach aims to be sound and automated at the same time which forces us to carefully balance runtime performance and precision.

3.12 That's a Tough Call! Studying the Challenges of Call Graph Construction for WebAssembly


Michelle Thalakottur (Northeastern University – Boston, US), Daniel Lehmann (Universität Stuttgart, DE & Google – München, DE), Michael Pradel (Universität Stuttgart, DE), and Frank Tip

License  Creative Commons BY 4.0 International license
© Michelle Thalakottur, Daniel Lehmann, Michael Pradel, and Frank Tip

Call graphs are at the core of many inter-procedural static analysis and optimization techniques. However, WebAssembly poses some unique challenges for static call graph construction. Currently, these challenges are neither well understood, nor is it clear to what extent existing techniques address them. We systematically study WebAssembly specific challenges for static call graph construction and identify and classify 12 challenges. We then measure their prevalence in real-world binaries. We also study the soundness and precision of four existing static analyses. Our findings include that, surprisingly, all of the existing techniques are unsound, without this being documented anywhere. We envision our work to provide guidance for improving static call graph construction for WebAssembly.

3.13 WebAssembly as the Basis of All Things?

Ben L. Titzer (Carnegie Mellon University – Pittsburgh, US)

License  Creative Commons BY 4.0 International license
© Ben L. Titzer

WebAssembly is a low-level, portable bytecode offering a compilation target with near-native performance. Now a standard feature of all web browsers, Wasm has started to expand to many other applications such as edge computing, distributed cryptographic digital contracts, networking stacks, and more. As Wasm gains features through the standardization process, it becomes a more attractive target for new kinds of languages. In this talk I will fast-forward to look at a whole new set of language runtime system designs that are made possible with some new features that can be added to Wasm. In particular, can we build a *really* fast

language implementation without having to write a new JIT compiler? A new code format (with validator)? A new garbage collector? Can we do this without ever having to look at assembly language? I hope so! Let's look at what I've discovered and what I think that means.

3.14 Verifying Instruction Selection in a Wasm-to-native Compiler

Alexa VanHattum (Cornell University – Ithaca, US)

License © Creative Commons BY 4.0 International license
© Alexa VanHattum

Joint work of Alexa VanHattum, Monica Pardeshi, Chris Fallin, Adrian Sampson, Fraser Brown

For ahead-of-time or just-in-time compilation, Wasm's sandboxing guarantees rely on the correctness of the generated native assembly. Subtle wrong-code bugs in native instruction selection can introduce security flaws. In this talk, I'll present our efforts toward automated verification for instruction lowering rules within Cranelift, a production code generator for Wasmtime. I'll discuss our approach to modeling the Cranelift intermediate representation and ARM aarch64 backend, challenges with generalizing over types, and several case studies of faults analyzed by our tool.

3.15 MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code

Marco Vassena (Utrecht University, NL)

License © Creative Commons BY 4.0 International license
© Marco Vassena

Joint work of Alexandra E. Michael, Anitha Gollamudi, Jay Bosamiya, Evan Johnson, Aidan Denlinger, Craig Disselkoe, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, Deian Stefan

Main reference Alexandra E. Michael, Anitha Gollamudi, Jay Bosamiya, Evan Johnson, Aidan Denlinger, Craig Disselkoe, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, Deian Stefan: "MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code", Proc. ACM Program. Lang., Vol. 7(POPL), pp. 425–454, 2023.

URL <https://doi.org/10.1145/3571208>

Most programs compiled to WebAssembly (Wasm) today are written in unsafe languages like C and C++. Unfortunately, memory-unsafe C code remains unsafe when compiled to Wasm—and attackers can exploit buffer overflows and use-after-frees in Wasm almost as easily as they can on native platforms. This talk presents Memory-Safe WebAssembly (MSWasm), an extension of Wasm with language-level memory-safety abstractions to precisely address this problem. In the talk, we will discuss the design of MSWasm and show how compilers can leverage these abstractions to automatically eliminate memory vulnerabilities from unsafe code. We have developed a C-to-MSWasm compiler on top of Clang and two compilers of MSWasm to native code, which support different enforcement mechanisms, and thus allow developers to make security-performance trade-offs according to their needs. More importantly, MSWasm's design makes it easy to swap between enforcement mechanisms; as fast (especially hardware-based) enforcement techniques become available, MSWasm will be able to take advantage of these advances almost for free.

3.16 The Path to Components

Luke Wagner (Fastly – San Francisco, US)

License  Creative Commons BY 4.0 International license
© Luke Wagner

This talk described the motivation for starting work on a new set of standards focused on portably and securely executing WebAssembly outside the browser, viz., WASI and the Component Model. Next, the talk covered the motivation for not simply adopting the well-established design of POSIX by identifying 4 areas where POSIX has performance and composability problems, viz., around linking, the passing of high-level values between processes, the handling of external resources and the basis for concurrency. Lastly, the talk gave a quick preview of how the tooling could work in practice and be shared across a heterogeneous ecosystem of platforms executing WebAssembly.

3.17 Usefully Mechanising All of WebAssembly

Conrad Watt (University of Cambridge, GB)


License  Creative Commons BY 4.0 International license
© Conrad Watt

This talk will describe WasmCert-Isabelle, an Isabelle/HOL mechanisation of the WebAssembly specification; and recent work on developing WasmRef-Isabelle, a practically-useful reference implementation of WebAssembly that is verified with respect to this mechanisation. We describe our successes in driving WasmRef-Isabelle’s adoption as a fuzzing oracle for the widely-used Wasmtime implementation of WebAssembly, as well as the challenges we will face in keeping our work up to date with the ever-evolving WebAssembly standard.

4 Working groups

4.1 A DSL for writing the WebAssembly Specification

Andreas Rossberg (München, DE), Joachim Breitner (Freiburg, DE), Pierre Chambart (Société OCamlPro SAS – Paris, FR), Philippa Gardner (Imperial College London, GB), Sam Lindley (University of Edinburgh, GB), Matija Pretnar (University of Ljubljana, SI), Xlaojia Rao, Sukyoung Ryu (KAIST – Daejeon, KR), Luke Wagner (Fastly – San Francisco, US), Conrad Watt (University of Cambridge, GB), and Dongjun Youn (KAIST – Daejeon, KR)

License  Creative Commons BY 4.0 International license
© Andreas Rossberg, Joachim Breitner, Pierre Chambart, Philippa Gardner, Sam Lindley, Matija Pretnar, Xlaojia Rao, Sukyoung Ryu, Luke Wagner, Conrad Watt, and Dongjun Youn

Motivation

To standardise a WebAssembly feature, the following specification artefacts must be produced:

- a formal specification of the feature in LaTeX
- a prose description of the feature
- a reference implementation in OCaml

This process is onerous and several important upcoming WebAssembly features such as Threads and Exception Handling have not yet been standardised purely because they do not meet these requirements, despite widespread industrial support, implementation, and use. In addition, inconsistencies between these definitions can lead to divergences in implementations¹.

Moreover, academic mechanisations of WebAssembly such as WasmCert-Isabelle² and WasmCert-Coq³ must be updated with each new feature if they wish to remain correspondant to the current version of WebAssembly. Even when this effort is undertaken, academics working with other popular theorem provers such as Lean and Agda are not be able to make use of these models and would need to write their own from scratch in order to carry out mechanisation-related research on WebAssembly.

We propose to develop a machine readable domain-specific language (DSL) that will function as a unified source of truth for the WebAssembly specification, and for which we can define a number of separate *backends* to generate not only the above specification artefacts, but also mechanisations in all major theorem provers. This will significantly improve the productivity of WebAssembly’s industrial standards body, and allow academics to access a feature-complete mechanisation of WebAssembly no matter their preferred theorem prover.

Why not use an existing DSL?

There are a number of existing tools for writing language definitions [1, 2, 3], with backends that could generate \LaTeX , parts of a reference interpreter, and stubs for proof assistants. We have considered using them for the Wasm specification, but ultimately decided against them – we believe that we will be able to generate higher-quality artefacts by building our DSL to intrinsically make use of domain-specific knowledge about the WebAssembly specification.

As an immediate example, it would likely be impossible to generate a prose description acceptable to WebAssembly’s standards body from a generic IR. In addition, our initial work suggests that our generated interpreter will be significantly more efficient if we are able to make use of certain knowledge regarding the restricted structure of WebAssembly’s evaluation contexts.

Moreover, the primary audience of the DSL is the existing specification authors, so we intend for the DSL to closely reflect the current style of the hand-written specification – for example the pervasive use of sequences and corresponding iterators. For the same reason, we have also quickly abandoned a prototype of an embedded DSL in OCaml⁴.

Proposed solution

The solution proposed by Andreas Rossberg and quickly adopted by the whole group is SpecTec⁵. In it, the specification is written in a text format similar to the existing specification. The language consists of few generic concepts:

¹ <https://github.com/WebAssembly/threads/issues/195>

² <https://github.com/WasmCert/WasmCert-Isabelle>

³ <https://github.com/WasmCert/WasmCert-Coq>

⁴ <https://github.com/matijapretnar/wasm-spec-dsl>

⁵ <https://github.com/Wasm-DSL/spectec>

- *Syntax definitions*, describing the grammar of the input language or auxiliary constructs. These are essentially type definitions for the object language. For example:

```

syntax valtype = | I32 | I64 | F32 | F64
syntax functype = valtype* -> valtype*
syntax instr = | NOP | BLOCK instr* | IF instr* ELSE instr*
syntax context = { FUNC functype*, LABEL (valtype)* }
syntax config = state; instr*

```

- *Variable declarations*, ascribing the syntactic class (i.e., type) that meta variables used in rules range over. For example:

```

var t : valtype
var ft : functype
var 'C : context

```

(Also, every type name is implicitly usable as a variable of the respective type.)

- *Relation declarations*, defining the shape of judgement forms, such as typing or reduction relations. These are essentially type declarations for the meta language. For example:

```

relation Instr_ok: context |- instr : functype
relation Step: config ~> config

```

- *Rule definitions*, expressing the individual rules defining relations. For example:

```

rule Instr_ok/nop:
  'C |- NOP : epsilon -> epsilon

rule Instr_ok/if:
  'C |- IF instr_1* ELSE instr_2* : t_1* -> t_2
  -- InstrSeq_ok: 'C, LABEL t_2* |- instr_1* : t_1* -> t_2*
  -- InstrSeq_ok: 'C, LABEL t_2* |- instr_2* : t_1* -> t_2*

rule Step/nop:
  z; NOP ~> z; epsilon

rule Step/if-true:
  z; (I32.CONST c) (IF instr_1* ELSE instr_2*) ~> z; (BLOCK instr_1*)
  -- if c != 0
rule Step/if-false:
  z; (I32.CONST c) (IF instr_1* ELSE instr_2*) ~> z; (BLOCK instr_2*)
  -- if c = 0

```

Every rule is named, so that it can be referenced. Each premise is introduced by a dash and includes the name of the relation it is referencing, easing checking and processing.

- *Auxiliary Functions*, allowing to abstract complex conditions into separate definitions. For example:

```

def $size(numtype) : nat
def $size(I32) = 32
def $size(I64) = 64
def $size(F32) = 32
def $size(F64) = 64

```

- *Hint annotations* that are uninterpreted by default, but may offer occasional extra guidance for different backends (eg. \LaTeX macros to be used).

The implementation defines two AST representations: an *external language* (EL), which is close to the written specification and suitable for backends generating \LaTeX , and an *internal language* (IL), suitable for backends generating programs. Elaboration from EL into IL infers additional information and makes it explicit in the representation:

- resolve notational overloading and mixfix applications,
- resolve overloading of variant constructors and annotate them with their type,
- insert injections from variant subtypes into supertypes,
- insert injections from singletons into options/lists,
- insert binders and types for local variables in rules and functions,
- mark recursion groups and group definitions with rules, ordering everything by dependency.

Progress since the seminar

Since the seminar, there has been significant progress on the implementation. In addition to the basic infrastructure and a \LaTeX backend, there is a prototype backend generating prose English, as well as ones generating Agda, Coq and Lean code, and a number of passes that further elaborate the IL. Particular effort is being put into *animating* the operational semantics, i.e., transforming the rules into algorithmic steps that then can be used for generating both the prose part of the specification and a reference interpreter.

References

- 1 Martin Bodin, Philippa Gardner, Thomas P. Jensen, and Alan Schmitt. Skeletal semantics and their interpretations. *Proc. ACM Program. Lang.*, 3(POPL):44:1–44:31, 2019.
- 2 Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In *ICFP*, pages 175–188. ACM, 2014.
- 3 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.

Participants

- Amal Ahmed
Northeastern University – Boston, US
- Léo Andrés
University Paris-Saclay – Orsay, FR
- Javier Cabrera Arteaga
KTH Royal Institute of Technology – Stockholm, SE
- Karthikeyan Bhargavan
INRIA – Paris, FR
- Joachim Breitner
Freiburg, DE
- Pierre Chambart
Société OCamlPro SAS – Paris, FR
- Martin Fink
TU München – Garching, DE
- Philippa Gardner
Imperial College – London, UK
- Aina Linn Georges
Aarhus University, DK
- Arjun Guha
Northeastern University – Boston, US
- Reiner Hähnle
TU Darmstadt, DE
- Daniel Hillerström
Huawei Technologies – Zürich, CH
- Evan Johnson
University of California – San Diego, US
- Daniel Lehmann
Google – München, DE
- Sam Lindley
University of Edinburgh, UK
- Tyler McMullen
Fastly – San Francisco, US
- Lucy Menon
Northeastern University – Boston, US
- Shravan Narayan
University of California – San Diego, US
- Luna Phipps-Costin
Northeastern University – Boston, US
- Jean Pichon-Pharabod
Aarhus University, DK
- Michael Pradel
Universität Stuttgart, DE
- Matija Pretnar
University of Ljubljana, SI
- Jonathan Protzenko
Microsoft – Redmond, US
- Andreas Rossberg
München, DE
- José Fragoso Santos
INESC-ID – Lisbon, PT
- Claudio Russo
Dfinity – Cambridge, UK
- Sukyoung Ryu
KAIST – Daejeon, KR
- Markus Scherer
TU Wien, AT
- Sabine Schmaltz
Tarides – Saarbrücken, DE
- Till Schneidereit
Fermion – Heidelberg, DE
- KC Sivaramakrishnan
Indian Institute of Technology – Madras, IN
- Deian Stefan
University of California – San Diego, US
- Michelle Thalakkottur
Northeastern University – Boston, US
- David Thien
University of California – San Diego, US
- Ben Titzer
Carnegie Mellon University – Pittsburgh, US
- Alexa VanHattum
Cornell University – Ithaca, US
- Marco Vassena
Utrecht University, NL
- Luke Wagner
Fastly – San Francisco, US
- Conrad Watt
University of Cambridge, UK
- Dongjun Youn
KAIST – Daejeon, KR

