

Utilising and Scaling the WebAssembly Semantics

Edited by

Amal Ahmed¹, Andreas Rossberg², Deian Stefan³, and Conrad Watt⁴

¹ Northeastern University - Boston, US, amal@ccs.neu.edu

² München, DE, rossberg@mpi-sws.org

³ University of California - San Diego, US, deian@cs.ucsd.edu

⁴ Nanyang TU - Singapore, SG, conrad.watt@cl.cam.ac.uk

Abstract

WebAssembly (Wasm) is a safe and portable, low level bytecode format used in browsers, IoT applications, cloud, edge, embedded systems and blockchains. Its popularity as a technology for both practically building and theoretically investigating verified and secure systems has been growing rapidly. This Dagstuhl Seminar brought together leading academics and industry representatives involved in Wasm, both as designers, implementers or clients of the technology, to exchange ideas around topics such as tools for formal specification, verified compilation, software fault isolation and language interoperability.

Seminar June 9–13, 2025 – <http://www.dagstuhl.de/25241>

2012 ACM Subject Classification Theory of computation Program semantics

Keywords and phrases Compilers, Formal Methods, JavaScript, Proof Assistants, Runtimes, Software Verification, Webassembly

Digital Object Identifier 10.4230/DagRep.15.6.1

Edited in cooperation with Thalakottur, Michelle

0.1 Executive Summary

Andreas Rossberg

License © Creative Commons BY 3.0 Unported license

© Andreas Rossberg

Joint work of Amal Ahmed, Andreas Rossberg, Deian Stefan, and Conrad Watt

WebAssembly (Wasm) is a safe and portable code format used in a broad variety of computational environments, such as Web browsers, cloud, edge, IoT, embedded systems, and blockchains. As a low-level programming language its instruction set is close to that of physical hardware, yet its semantics enforces memory safety, isolation, and the absence of undefined behavior. A distinguishing feature of Wasm is that its official specification contains a complete formal semantics, based directly on techniques developed and established by the scientific community, and proved sound with machine-verified proofs. Its popularity as a technology for both practically building and theoretically investigating verified and secure systems has hence been growing rapidly.

This Dagstuhl Seminar, brought together all sides interested in Wasm, its formal semantics, and its application to verification and generation techniques. By including academics and industry representatives involved in Wasm, both as designers, implementers or clients of the technology (e.g., compiler writers), we hoped to initiate discussion and new research about evolving the specification, as well as utilizing it for implementing verified systems,



Except where otherwise noted, content of this report is licensed under a Creative Commons BY 3.0 Unported license

Utilising and Scaling the WebAssembly Semantics, *Dagstuhl Reports*, Vol. 15, Issue 06, pp. 1–17

Editors: Amal Ahmed, Andreas Rossberg, Deian Stefan, and Conrad Watt



Dagstuhl Reports

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

programming languages, and new forms of tooling. The main focus was around the following four topics:

Tools for formal specification. Maintaining the current level of rigor in Wasm’s specification is an ongoing challenge. While many new features have been proposed for Wasm, the risk is that either the formal specification gets in the way of evolving the language, or that the formalization falls behind.

Verified compilation. We believe that Wasm’s formal semantics is an excellent starting point for performing verification in depth for systems built around the language. This applies both to compilation from source languages to Wasm, and compilation from Wasm to native code as it occurs in Wasm engines and their just-in-time compilers.

Software fault isolation. Software fault isolation can be implemented at various granularities and using different techniques, ranging from inline software checks to hardware-based isolation. Such techniques can inform both Wasm’s design and its implementations. Wasm can also serve as a vehicle for lightweight isolation, from sandboxing libraries to more ambitious projects, such as a hypervisor or an embedded execution environment.

Language interoperability. In Wasm multi-language interoperation can occur in two ways: first, as the interaction between a language compiled to Wasm and the language in which the host environment operates, and second between multiple different languages compiled to Wasm.

1 Table of Contents

Executive Summary	
<i>Andreas Rossberg</i>	1

Overview of Talks

Cross-language multi-core symbolic execution with Owi and more	
<i>Léo Andrès</i>	5
A Quick Tour on CompCert	
<i>Sandrine Blazy</i>	5
Motoko: Enhanced Orthogonal Persistence	
<i>Claudio Russo</i>	5
Automating Coq Mechanization for Webassembly via Spectec	
<i>Diego Cupello and Philippa Gardner</i>	6
RichWasm Realizability: Facilitating Formal Specification of ABIs for WebAssembly	
<i>Ryan Doenges</i>	6
WasmGC languages: where are we?	
<i>Sébastien Doeraene</i>	7
Verifying Wasm-to-Native Compilation with Authoritative ISA Specifications	
<i>Chris Fallin</i>	7
Binaryen IR and Type System	
<i>Thomas Lively</i>	8
A Logical Interface for Garbage Collectors Implemented in WebAssembly	
<i>Brianna Marshall</i>	8
Execution-Aware Program Reduction for WebAssembly via Record and Replay	
<i>Michael Pradel</i>	8
WasmCert-Rocq and the Designs for Future Mechanisations	
<i>Xiaojia Rao</i>	9
A Quick Tutorial on SpecTec	
<i>Andreas Rossberg</i>	9
Propagating Host Types for Multi-Language Static Analysis of WebAssembly	
<i>Michelle Thalakkottur</i>	10
Engine Interfaces for Wasm Instrumentation	
<i>Ben L. Titzer</i>	10
An Adjoint Separation Logic for the Wasm Call Stack	
<i>Andrew Wagner</i>	10
Future of concurrency in Wasm	
<i>Conrad Watt</i>	11
Towards Performant Static Analysis of WebAssembly via Staging and Continuations	
<i>Guannan Wei</i>	11
WEST: Generating Wasm Test Cases based on SpecTec	
<i>Dongjun Youn, Sukyoung Ryu, and Wonho Shin</i>	11

Working groups

Interoperability	
<i>Sébastien Doeraene</i>	12
Garbage Collection and Dynamic Languages	
<i>Chris Fallin</i>	12
Performance and Benchmarking	
<i>Daniel Lehmann</i>	13
SpecTec and Mechanisation	
<i>Andreas Rossberg</i>	14
Static Analysis	
<i>Michelle Thalakottur, Léo Andrès, Alexander Bai, Thomas Lively, Marco Patrignani,</i> <i>and Guannan Wei</i>	15
Instrumentation	
<i>Ben L. Titzer, Chris Fallin, Ralph Squillace, and Thomas Trenner</i>	15
Concurrency in Wasm	
<i>Conrad Watt</i>	16
Testing	
<i>Dongjun Youn</i>	17

2 Overview of Talks

2.1 Cross-language multi-core symbolic execution with Owi and more

Léo Andrès (OCamlPro - Paris, FR)

License © Creative Commons BY 3.0 Unported license
 © Léo Andrès
Joint work of Marques, Filipe; Chambart, Pierre; Carcano, Arthur
Main reference Andrès, L., Marques, F., Carcano, A., Chambart, P., Santos, J. F., Filliâtre, J. C. (2024). Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly. arXiv preprint arXiv:2412.06391.
URL <https://arxiv.org/abs/2412.06391>

I'll present new things happening in Owi, a Wasm toolkit featuring a symbolic execution engine. I'll show how we use it to perform automated bug-finding on C, C++, Rust and Zig programs and a bug we found in the Rust stdlib. I'll also present more recent work : how we run Wasm in a Unikernel using MirageOS, the work on owi iso to check Binaryen's optimizations, proof of programs by reusing ACSL (a specification language for C), Weasel (a draft Wasm specification language), support for advanced test coverage criteria (e.g. MCDC), and future work (handling Haskell, TinyGo, OCaml and Guile), build system integration and support for WASI/Component model/Common ABI.

2.2 A Quick Tour on CompCert

Sandrine Blazy (University of Rennes, FR)

License © Creative Commons BY 3.0 Unported license
 © Sandrine Blazy
URL <https://compcert.org/>

The CompCert verified C compiler is an open infrastructure for research. It accomplishes a series of 18 passes through 9 intermediate languages. I will explain how their semantics is mechanized and how this facilitates the reasoning required to prove compiler correctness.

2.3 Motoko: Enhanced Orthogonal Persistence


Claudio Russo

License © Creative Commons BY 3.0 Unported license
 © Claudio Russo
Joint work of Bläser, Luc; Russo, Claudio; Greif, Gabor; Vandersmith, Ryan; Ibrahim, Jason
Main reference Luc Bläser, Claudio Russo, Gabor Greif, Ryan Vandersmith, and Jason Ibrahim. 2024. Smarter Contract Upgrades with Orthogonal Persistence. In Proceedings of the 16th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '24). Association for Computing Machinery, New York, NY, USA, 32–42.
URL <https://doi.org/10.1145/3689490.3690401>

Motoko is a strongly-typed, actor based language with impure functional features that targets the Internet Computer blockchain. I'll give an overview on Motoko's unique support for data persistence across requests and code upgrades (the enhanced bit).

2.4 Automating Coq Mechanization for Webassembly via Spectec

Diego Cupello (Imperial College London, GB) and Philippa Gardner (Imperial College London, GB)

License  Creative Commons BY 3.0 Unported license
© Diego Cupello and Philippa Gardner

Spectec is a new toolchain for inputting specifications, namely for Wasm. It delivers specification requirements that all come from a "single source of truth", being a new DSL that is designed with human readability and expressivity in mind. I will give a small overview of the prototype pass made to generate Rocq definitions, and give some advances in designing the future structure that allows for generation of Inductive definitions in many interactive theorem provers, including Rcoq, Isabelle, Lean and Agda. Furthermore, I will also give a small proof of concept extension that allows lemmas and theorems to be in part of the DSL, with a mechanism that allows the generation of boilerplate lemmas.

2.5 RichWasm Realizability: Facilitating Formal Specification of ABIs for WebAssembly

Ryan Doenges (Northeastern University - Boston, US)

License  Creative Commons BY 3.0 Unported license
© Ryan Doenges

Joint work of Brianna Marshall, Ryan Doenges, Maxime Legoupil, Lars Birkedal, Amal Ahmed

For programs in different high-level languages to interoperate after translation to WebAssembly, they need a shared Application Binary Interface (ABI) that standardizes data layout, memory management policies, and calling conventions. This means each language has to have an independent definition of how its types relate to the ABI, which leads to incompatibilities and duplication of effort.

To unify these ABI specifications, we describe RichWasm, a low-level IR with high-level types for interoperation between GC'd and manually managed languages, and equip it with a realizability model in Iris. The model interprets RichWasm types as separation logic specifications for WebAssembly terms, while the type system of RichWasm itself is expressive enough to support type-preserving compilation from source languages like ML or L3, a calculus of linear references. Our model is implemented in the Iris-Wasm program logic. To show the model is useful, we prove it is adequate for a memory isolation property. To show the model is not vacuous, we implement a compiler from RichWasm to WebAssembly and prove that it sends terms of type T to inhabitants of the realizability model at T.

This talk describes work in progress.

2.6 WasmGC languages: where are we?

Sébastien Doeraene (EPFL Lausanne, CH)

License © Creative Commons BY 3.0 Unported license
© Sébastien Doeraene

Joint work of Taniguchi, Rikito; Schlatter, Tobias

A look at the state of WasmGC, in terms of the capabilities it gives to languages ... and what it doesn't give. We will particularly look at questions of interoperability (with JS and with the Component Model) and performance ("unnecessary" casts, late binding).

Interoperability with JavaScript has made leaps with the introduction of WasmGC, as we can now have cycles between GCed structures across JavaScript and Wasm. There are still some gaps, but proposals are on their way to address them. Using the Component Model from a GC language remains a big question mark, especially with the requirements for ubiquitous copies, and unknowns regarding drop semantics.

Performance is fine, provided a decent ahead-of-time optimizer. In some cases, our best efforts are still slower than equivalent JavaScript code. The main performance problems derive from two sources: calls to small JavaScript bridge functions, and virtual/interface method dispatch. We also discuss some smaller, possibly low-hanging fruit.

2.7 Verifying Wasm-to-Native Compilation with Authoritative ISA Specifications

Chris Fallin (F5 - San Jose, US)

License © Creative Commons BY 3.0 Unported license
© Chris Fallin

Joint work of Alexa VanHattum, Michael McLoughlin, Adrian Sampson, Brian Parno, Fraser Brown, Monica Parneshi

Main reference Alexa VanHattum, Monica Parneshi, Chris Fallin, Adrian Sampson, Fraser Brown. Lightweight, Modular Verification for WebAssembly-to-Native Instruction Selection. In ASPLOS 2024.

Main reference Michael McLoughlin, Ashley Sheng, Chris Fallin, Bryan Parno, Fraser Brown, Alexa VanHattum. Scaling Instruction-Selection Verification with Authoritative ISA Semantics in an Industrial WebAssembly Compiler. To appear in OOPSLA 2025.

When compiled to native execution, WebAssembly's guarantees are only as strong as the compiler's. Subtle wrong-code bugs, including in instruction selection, have introduced serious security vulnerabilities. In this talk, we'll describe our system for lightweight, modular verification of instruction-lowering rules in Wasmtime's Cranelift compiler backend. We now automatically derive machine-specific specifications from vendor-provided ISA semantics (via partial evaluation of ASL) for ARM aarch64. We'll discuss the particular challenges of handling partially-symbolic instructions, e.g., for immediate operands. Our design also benefits from lightweight state modeling and compositional reasoning over chains of rewrite rules. We reproduce known bugs, including one of the most severe security bugs in Wasmtime's history (from 2023, concurrent with the last Wasm Dagstuhl).

2.8 Binaryen IR and Type System

Thomas Lively (Google - Mountain View, US)

License  Creative Commons BY 3.0 Unported license
© Thomas Lively

Joint work of Lively, Thomas; Zakai, Alon; The WebAssembly Community Group
URL <https://github.com/webassembly/binaryen>

Binaryen is a widely used WebAssembly optimizer, but its IR design predates the decision to make WebAssembly a stack machine. This, among other considerations, leads to some interesting differences between the standard WebAssembly type system and the one used in Binaryen. This talk will be a brief tour of those differences and their consequences.

2.9 A Logical Interface for Garbage Collectors Implemented in WebAssembly

Brianna Marshall (Northeastern University - Boston, US)

License  Creative Commons BY 3.0 Unported license
© Brianna Marshall

Joint work of Marshall, Brianna; Doenges, Ryan; Legoupil, Maxime; Duckham, Owen; Prakash, Ari; Birkedal, Lars; Ahmed, Amal

As part of defining an ABI for RichWasm, we must specify how garbage collected references are encoded in WebAssembly. Because RichWasm requires that the garbage collector can free linear references owned by a collected object, the target GC needs to support finalizers, which rules out the Wasm GC extension. We describe an approach for "rolling our own GC" in Iris-Wasm that encapsulates the physical memory owned by the GC behind a separation logic interface. The mutator instead owns blocks of abstract memory, which can be converted temporarily to the underlying physical resources so that the memory can be used directly between invocations of the GC.

2.10 Execution-Aware Program Reduction for WebAssembly via Record and Replay

Michael Pradel (Universität Stuttgart, DE)

License  Creative Commons BY 3.0 Unported license
© Michael Pradel

Joint work of Doehyun Baek, Daniel Lehmann, Ben L. Titzer, Sukyoung Ryu
URL <https://arxiv.org/abs/2506.07834>

WebAssembly (Wasm) programs may trigger bugs in their engine implementations. To aid debugging, program reduction techniques try to produce a smaller variant of the input program that still triggers the bug. However, existing execution-unaware program reduction techniques struggle with large and complex Wasm programs, because they rely on static information and apply syntactic transformations, while ignoring the valuable information offered by the input program's execution behavior. We present RR-Reduce and Hybrid-Reduce, novel execution-aware program reduction techniques that leverage execution behaviors via record and replay. RR-Reduce identifies a bug-triggering function as the target function, isolates that function from the rest of the program, and generates a reduced program that replays

only the interactions between the target function and the rest of the program. Hybrid-Reduce combines a complementary execution-unaware reduction technique with RR-Reduce to further reduce program size. We evaluate RR-Reduce and Hybrid-Reduce on 28 Wasm programs that trigger a diverse set of bugs in three engines. On average, RR-Reduce reduces the programs to 1.20

2.11 WasmCert-Rocq and the Designs for Future Mechanisations

Xiaoja Rao (Imperial College London, GB)

License © Creative Commons BY 3.0 Unported license
 © Xiaoja Rao
URL <https://github.com/WasmCert/WasmCert-Coq>

WasmCert-Rocq has evolved much further since its initial formulation of Wasm 1.0. The mechanisation has shifted its target version of Wasm from 1.0 to 2.0 and beyond, and the scope of work included in the mechanisation has also vastly expanded from the initial soundness results. In this talk, I will give an overview of the current structure of the Rocq mechanisation, which formalises Wasm 2.0 + tail call, implementing the subtyping system from future proposals. I will then highlight some of the most interesting designs devised since the inception of the mechanisation, including the progressful interpreter for proof consolidation, an efficient persistent data structure for extracting memory representations without resolving fully to monadic state operations, and some consideration in extraction for convenient host interoperations. These designs can stay relevant in producing an efficient verified reference interpreter in the future, when a SpecTec-generated mechanisation comes to fruition. The extracted runtime now passes the Wasm 2.0 core test suite fully.

2.12 A Quick Tutorial on SpecTec


Andreas Rossberg

License © Creative Commons BY 3.0 Unported license
 © Andreas Rossberg
Main reference Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaoja Rao, Conrad Watt, Andreas Rossberg: Bringing the WebAssembly Standard up to Speed with SpecTec. *Proc. ACM Program. Lang.* 8(PLDI): 1559-1584 (2024)

SpecTec is the new DSL and tool chain for authoring the Wasm spec. It can be used to generate various output formats from a formal definition, like Latex rules, English prose, Coq definitions, etc. This presentation gave a brief overview of the SpecTec language itself, using a small fragment of Wasm as a running example and showing how a specification can be created with it.

2.13 Propagating Host Types for Multi-Language Static Analysis of WebAssembly

Michelle Thalakottur (Northeastern University - Boston, US)

License  Creative Commons BY 3.0 Unported license


© Michelle Thalakottur

Joint work of Thalakottur, Michelle; Garg, Harshit; Mane Siddhant; Fallin Chris; Ahmed Amal; Tip, Frank

Current state of the art static analysis tools analyze WebAssembly binaries in isolation, that is, they perform a closed-world analysis where they make no assumptions about the Javascript client interacting with the WebAssembly binary, resulting in significant loss of precision. Inspired by real-world invocation patterns of WebAssembly by compiler-generated Javascript wrappers, we design a refinement type system over WebAssembly. We infer refined WebAssembly types by generating and solving constraints and use them to generate call graphs that are more precise than those generated from a close-world analysis.

2.14 Engine Interfaces for Wasm Instrumentation

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


License  Creative Commons BY 3.0 Unported license

© Ben L. Titzer

While Wasm enjoys excellent performance on many platforms due to investment in engine optimization, challenges remain to improve Wasm as a development target. In particular, the lack of standardized debugging and profiling tools across engines has led to fragmentation and less than ideal developer experience. Wasm lags behind other environments like the JVM by a wide margin. In this talk I will outline steps that we've taken to define an engine interface for programmable instrumentation with low overhead and (hopefully) low engine implementation effort.

2.15 An Adjoint Separation Logic for the Wasm Call Stack

Andrew Wagner (Northeastern University - Boston, US)

License  Creative Commons BY 3.0 Unported license

© Andrew Wagner

Joint work of Wagner, Andrew; Eisbach, Zachary; Ahmed, Amal

We propose a pair of adjoint modalities for separation logics over stacks of resources. This abstraction guarantees the encapsulation expected of a stack discipline in a completely local, small footprint style. In the context of Wasm, these modalities can alleviate the burden of explicit threading around a monolithic resource for each activation frame.

2.16 Future of concurrency in Wasm

Conrad Watt (Nanyang TU - Singapore, SG)

License © Creative Commons BY 3.0 Unported license
© Conrad Watt

Main reference Conrad Watt, Concurrency in WebAssembly, ACM Queue Volume 23 Issue 3
URL <https://dl.acm.org/doi/abs/10.1145/3747201.3746173>

WebAssembly has basic support for concurrency through the use of shared memories and the corresponding `SharedArrayBuffer` in JavaScript. However, because these concurrency capabilities only support the sharing of data in a bitwise representation, some language with richer concurrency features run into expressivity limitations when compiling to WebAssembly. This talk discusses these limitations, and sketches ongoing work within the WebAssembly standards community to extend the language with more powerful concurrency capabilities. Of particular interest are "shared functions", which would give WebAssembly the capability to share executable code cross-thread.

2.17 Towards Performant Static Analysis of WebAssembly via Staging and Continuations

Guannan Wei (INRIA - Paris, FR & Tufts University - Medford, US)

License © Creative Commons BY 3.0 Unported license
© Guannan Wei

Joint work of Guannan Wei, Dinghong Zhong, Alex Bai

The official WebAssembly specification provides a small-step reduction semantics. However, this semantics is complicated by additional "administrative instructions" to account for WebAssembly's structured control-flow constructs. As a result, the semantics is not compositional and is not amenable for partial evaluation or staging.

In this talk, I presented an alternative natural semantics for WebAssembly in continuation-passing style (CPS), which can be implemented as a concise, compositional, and tail-recursive definitional interpreter. By using continuations from the meta-language, this approach streamlines the semantics and eliminates the need for administrative instructions.

I also discussed our ongoing work using this CPS semantics as a foundation for efficient concolic execution engines of WebAssembly, building on the first Futamura projection. Finally, I outlined future directions, including the use of SpecTec to specify interderivable semantics by program transformation.

2.18 WEST: Generating Wasm Test Cases based on SpecTec

Dongjun Youn (KAIST - Daejeon, KR), Sukyoung Ryu, and Wonho Shin

License © Creative Commons BY 3.0 Unported license
© Dongjun Youn, Sukyoung Ryu, and Wonho Shin


We present WEST (WebAssembly Specification-based Testing), a framework that automatically produces Wasm test cases from mechanized specifications written in SpecTec. Given any full or subset variant of the Wasm specification as input, WEST aims to systematically

generate test programs that conform to the input grammar and validation rules, and capture the runtime behavior defined by its execution semantics.

3 Working groups

3.1 Interoperability

Sébastien Doeraene (EPFL Lausanne, CH)

License  Creative Commons BY 3.0 Unported license
© Sébastien Doeraene

This working group focused on two broad areas of interest related to interoperability between languages on the Wasm platform. First, the run-time cost of cross-language boundaries. Second, the interaction of various type systems.


We discussed what seems to be the biggest factor of the run-time cost of interoperability: extra copies of data. Those copies are motivated by a combination of a) differences in data layout (including GC versus linear models) and b) protections of each language's abstractions. We discussed the need for a unified semantic framework. It could provide the necessary ground for enforcing abstractions across languages, or at least a common denominator of the most important ones. If we are willing to recompile modules when combining them, a unified framework may also be used to specialize modules to various data layouts.

The interactions between type systems are perhaps even trickier, with no clear path forward. Here as well, some notion of common denominator could be used. RichWasm was discussed at length as a possible foundation for type systems to compile down to. Generics were brought up as a particularly hard topic. IDLs offer a different look at the problem space: rather than compiling language type systems down to a common ground, we can "lift up" IDL definitions into source languages.

As takeaway from the working group, it is clear that interoperability is a hard topic. Rather than finding an all-encompassing solution, the most promising path forward seems to take concrete pairs of languages, and find dedicated solutions for those. With time, we may be able to generalize the learning outcomes of several approaches. The "common denominator" was a recurring concept. There is some inherent tension between the various levels at which it could be placed. WasmGC might provide a better foundation for an interoperability layer than the traditional linear-memory-based approaches. Regardless of the approach taken, over time we will need a set of building blocks that languages can leverage.

3.2 Garbage Collection and Dynamic Languages

Chris Fallin (F5 - San Jose, US)

License  Creative Commons BY 3.0 Unported license
© Chris Fallin

A working group discussed two topics related to high-level language compilation to WebAssembly: the Wasm GC proposal (modes of use, potential extensions) and the handling of dynamism (dynamic types and other kinds of highly-dynamic language semantics).

We first discussed garbage collection adoption in several Wasm guest language compilers, and received an experience report from an author of the ScalaWasm project, including a

discussion of the compilation strategy for interfaces, vtables, and approaches that could avoid a sparse layout (slots for every possible interface). We discussed several possible extensions, including those named during the GC proposal development process as post-MVP features: sum types, trailing arrays, intersection types, support for inner pointers and inlined objects (as supported by languages such as Go), finalizers and weak-references, and others. Consensus was that there are use-cases for all of these features, but some of them may be more difficult than others to support in existing Wasm engines. In particular, we discussed the impact that Web engines' existing JS garbage collector semantics may have: for example, Wasm GC finalizers may be constrained to JS semantics (postmortem finalizers, rather than premortem finalizers that have an opportunity to re-create an edge to an object to retain it), and interior pointers may be difficult to support. We discussed whether formal verification techniques, such as MS-Wasm, may be useful to achieve safer garbage collection implementation; and several sandboxing techniques were also discussed (including engines such as Wasmtime that build GC storage on top of an untrusted Wasm linear memory internally). We discussed test coverage in the official spec test suite and agreed that it should be improved.

Next, we transitioned to a discussion of dynamic language implementation, focusing on inline caches (ICs) as a means to optimize behavior that cannot be known until runtime. We held an extensive discussion of the approach that an existing JavaScript-to-Wasm compilation (using SpiderMonkey and partial evaluation in the StarlingMonkey engine) takes to implement ICs, using an ahead-of-time-collected corpus of fastpath ICs and function pointers to these ICs that are updated. We discussed whether it might make sense to include a feature in core Wasm to allow for small functions with fast hot-patching, and perhaps first-class visibility of IC chains. We concluded that work should first focus on making use of newer Wasm features to reduce the function-call overhead seen in existing ICs (e.g., typed non-nullable function references), combined with targeted engine optimizations (e.g., minimizing the fixed overhead of function prologues and/or pushing function frame creation to cold-paths, similar to how native ICs work) and this should in theory be sufficient to match native code generated by typical dynamic language JITs.

3.3 Performance and Benchmarking

Daniel Lehmann (Google - München, DE)

License © Creative Commons BY 3.0 Unported license
© Daniel Lehmann

A small working group discussed topics related to WebAssembly benchmarking and performance, touching on measurement methodology, relevant metrics, tools employed by developers, and different audiences for performance work.

The first observation is that benchmarking and performance measurement methodology varies widely. One group, e.g., the Scala.js developer team and likely other Wasm toolchains focus for now on exercising different language features via microbenchmarks or synthetic workloads. One point of interest for that group is JavaScript interop performance (e.g., passing strings or the language's number type across the Wasm-JS boundary). They don't have large workloads yet that would mimic a full, realistic application. The Ocaml toolchain by Tarides however is in fact evaluated on micro and macro benchmarks, e.g., provided by Jane Street. Since those are closed source or proprietary, it's hard to share them across the Wasm community. Finally, other teams such as the Google V8 team do systematic benchmarking

on their CI/testing infrastructure, with automated performance alerts on regressions, larger workloads, and comparisons across different Wasm implementations. Toolchain and engine developers alike are searching for realistic, larger workloads to evaluate performance on, but they are hard to come by, especially during early stages of a toolchain or feature.

A second theme were differences as to what constitutes good performance. For some web applications, it might be peak performance or startup latency or memory usage. In particular the embedded use case of Wasm at Siemens is different in that it focuses on tight instruction budgets, because Wasm is used in control loops with a fixed time window. For those applications inconsistent performance is worse than simply slow execution speeds.


A third question is who is the audience for performance work and benchmarking, or put differently: where to optimize if performance goals are not met. That can be application developers working on the source language code, e.g., C++ or Scala compiled to Wasm; it can be toolchain authors, such as the compilers from Scala to Wasm or Emscripten or Binaryen; and finally it can be engine developers with optimizing JIT compilers.

An open idea during the working group was domain specific acceleration. E.g., similar to the JS-string-builtins proposal, should there be other intrinsics or fast imported functions for AES-NI/crypto primitives? One open question for the Wasm community could be to come up with a generic "builtin" mechanism, which may be polyfilled by a slow, software/scalar implementation as a Wasm module (similar to how JS-string-builtins are polyfillable in JavaScript host code).

Finally, there was a concrete call to action for the Wasm community, namely to engage more among/with the community (consisting of application, toolchain, and engine developers) to collect best practices and interesting workloads for benchmarking, since there is no centralized place for such information as of today. A WebAssembly benchmarking community subgroup already exists but is dormant, it could be revived for that purpose. One toolchain developer also wished for better documentation on what Wasm code is well optimizable by engines and which performance properties one could expect, similar to documents of native micro-architectures.

3.4 SpecTec and Mechanisation

Andreas Rossberg

License  Creative Commons BY 3.0 Unported license

© Andreas Rossberg

Main reference Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, Andreas Rossberg: Bringing the WebAssembly Standard up to Speed with SpecTec. Proc. ACM Program. Lang. 8(PLDI): 1559-1584 (2024)

SpecTec is the new specification language for the official WebAssembly definition that grew out of the previous Dagstuhl on WebAssembly. This session was concerned with possible new use cases, open problems, as well as possible improvement. We discussed of various ideas, from near-term to very ambitious.

Various suggestions were made for other artefacts that could potentially be generated from SpecTec, such as different flavours of interpreters (executing the reduction semantics, parsers, symbolic interpreters, CPS transforms), more tests, or other IRs (e.g., for tools like Binaryen). A prover backend might also be able to generate tests for the semantic specification itself, as an aid to language and proposal designers before all proofs are pushed through by mechanisation experts. Such a generator could take advantage of the stated meta-theoretical properties.

A problem for interpreters is the handling of non-determinism in the specification. While it may be desirable to explore the entire space of possible results (e.g., with abstract interpretation), such an attempt is likely to be computationally infeasible in the presence of features like the relaxed memory model. A simple fallback would be random execution (with a reproducible seed), although that cannot capture relaxed executions that involve reorderings of memory accesses.

Other topics of interest were the integration with spec update process, especially when certain backends lag behind, customising and subsetting specifications, and extending SpecTec to include theorem statements, in a way that would be abstract enough for different mechanisation backends. SpecTec’s general expressiveness was briefly discussed (e.g., the availability of evaluation contexts, binders, quantifiers, or judgements as expressions), as well as its IL semantics and possible meta-theory, a more robust and configurable IL-to-AL translation, and paths towards the ideal backend modularisation.

3.5 Static Analysis

Michelle Thalakottur (Northeastern University - Boston, US), Léo Andrès (OCamlPro - Paris, FR), Alexander Bai (MPI für Software Systems - Saarbrücken, DE), Thomas Lively (Google - Mountain View, US), Marco Patrignani (University of Trento, IT), and Guannan Wei (INRIA - Paris, FR & Tufts University - Medford, US)

License © Creative Commons BY 3.0 Unported license
© Michelle Thalakottur, Léo Andrès, Alexander Bai, Thomas Lively, Marco Patrignani, and Guannan Wei

We discussed what each of us were interested in, which was, performant static analyses, composable static analyses and multi-language analyses. We talked about handling various proposals and how non-determinism is a hard open problem. We also talked about how modeling system calls and calls to JavaScript is a problem faced by every static analysis tool. We all agreed that compiler-produced annotations or hints that are preserved from the source and embedded in custom sections in the Wasm binary, would be useful for a more precise analysis. Thomas Lively gave us an overview of how binaryen, a WebAssembly toolchain from Google, handles the Javascript environment in its static analysis, the different analyses being done in binaryen and what they would like to see implemented or improved. We talked about the difficulties we face while productionizing research static analysis tools. We ended by talking about the possibility of using an analysis platform like CodeQL for multi-language analysis.

3.6 Instrumentation

Ben L. Titzer (Carnegie Mellon University - Pittsburgh, US), Chris Fallin (F5 - San Jose, US), Ralph Squillace (Microsoft - Redmond, US), and Thomas Trenner (Siemens AG - Nürnberg, DE)

License © Creative Commons BY 3.0 Unported license
© Ben L. Titzer, Chris Fallin, Ralph Squillace, and Thomas Trenner

This working group focused on three industrial-relevant topics: debugging, introspection, and replay.

Key issues raised were the developer experience, such as IDE integration, and how difficult it can be to debug containers. Since many scenarios deploy Wasm code inside a container, this greatly improves the challenge. For dynamic languages whose runtimes already support remote debugging over a socket, the experience seems to somewhat work. But there is diversity here; every language has its own tooling, workflow, APIs, etc for doing this. Microsoft has had some success getting debugging to work for StarlingMonkey (JavaScript) and Python, but other languages don't have this yet.


The working group discussed debugging different semantic levels: the machine (ISA) level, the Wasm bytecode level, and the source level. Different engines have different support. E.g. wasmtime can allow debugging at the machine level in some situations, bytecode at others, and limited support for WasmDWARF.

Other issues that were raised were the inability to debug live cyberphysical systems (e.g. cannot step a \$100k machine). There are IP issues about debugging code on some platforms.

The group discussed remote debugging protocols and replay. More standardization and standard interfaces within the Wasm community would help. Members considered syncing again on how to design remote interfaces, e.g. to dig into how the debugging protocols work in VSCode.

3.7 Concurrency in Wasm

Conrad Watt (Nanyang TU - Singapore, SG)

License  Creative Commons BY 3.0 Unported license
© Conrad Watt

This working group was convened to discuss ways to support the compilation of concurrent code to Wasm, and the current standards body efforts under the "shared-everything threads" proposal to extend the concurrency capabilities of Wasm. The group discussed several topics in detail.


(1) some source languages have the capability to share dynamically-loaded executable code between threads (e.g. C through the `dlopen` system call). Wasm's support for this capability is currently poor and an extension to support Wasm-level shared functions would alleviate this issue.

(2) some embedded system implementations of Wasm may stretch the limitations of the standard and in particular the standard's specification of host environment interaction in order to provide more powerful concurrency capabilities. After some discussion, it was tentatively determined that many of these behaviours could actually be interpreted as standards-compliant, providing reassurance that the ecosystem can be kept somewhat unified.

(3) as Wasm transitions to a new specification infrastructure based on the SpecTec tool and DSL, new threads features being added to Wasm will require SpecTec to be extended. This represents an opportunity to ensure threads are specified precisely, as well as a hazard that SpecTec extension for this feature may prove to be onerous.

3.8 Testing

Dongjun Youn (KAIST - Daejeon, KR)

License  Creative Commons BY 3.0 Unported license
© Dongjun Youn

In the Wasm testing breakout session, various aspects of the topic were discussed. Both academia and industry are actively working on improving Wasm testing through multiple approaches. Current testing methods include random testing, where random inputs are generated for testing; mutation testing, which involves intentionally altering Wasm code to test the system's response; and using SpecTec as oracle for test generation, or using differential testing. There is also an ongoing effort to reduce redundancy and improve test performance through dynamic checks. New ideas and approaches presented during the meeting included generating negative tests, which focus on creating invalid or incorrect syntax, and testing for invariants to ensure certain conditions hold throughout execution. Property-based testing leveraging SpecTec for verification, was also discussed as a promising method. The potential use of large language models (LLMs) for generating testing was also explored. Evaluation of testing approaches emphasized the importance of test coverage. A suggestion was made to require a certain level of coverage of specification or reference interpreter before tests are officially adopted. Additionally, mutation testing was proposed, where faults are deliberately injected into the specification to test if existing tests can identify these errors. Several open problems and limitations were identified, including testing interoperation with other components like JavaScript, WASI, and polyfills. The issue of test format incompatibility was highlighted, as well as the performance tradeoffs in dynamic checks and challenges posed by nondeterminism in the tests. In conclusion, the discussion emphasized the importance of continuous cooperation between academia and industry to address these challenges and advance Wasm testing.