Weakening WebAssembly

CONRAD WATT, University of Cambridge, UK
ANDREAS ROSSBERG, Dfinity Stiftung, Germany
JEAN PICHON-PHARABOD, University of Cambridge, UK

WebAssembly (Wasm) is a safe, portable virtual instruction set that can be hosted in a wide range of environments, such as a Web browser. It is a low-level language whose instructions are intended to compile directly to bare hardware. While the initial version of Wasm focussed on single-threaded computation, a recent proposal extends it with low-level support for multiple threads and atomic instructions for synchronised access to shared memory. To support the correct compilation of concurrent programs, it is necessary to give a suitable specification of its memory model.

Wasm's language definition is based on a fully formalised specification that carefully avoids undefined behaviour. We present a substantial extension to this semantics, incorporating a relaxed memory model, along with a few proposed operational extensions. Wasm's memory model is unique in that its linear address space can be dynamically grown during execution, while all accesses are bounds-checked. This leads to the novel problem of specifying how observations about the size of the memory can propagate between threads. We argue that, considering desirable compilation schemes, we cannot give a sequentially consistent semantics to memory growth.

We show that our model guarantees Sequential Consistency of Data-Race-Free programs (SC-DRF). However, because Wasm is to run on the Web, we must also consider interoperability of its model with that of JavaScript. We show, by counter-example, that JavaScript's memory model is *not* SC-DRF, in contrast to what is claimed in its specification. We propose two axiomatic conditions that should be added to the JavaScript model to correct this difference.

We also describe a prototype SMT-based litmus tool which acts as an oracle for our axiomatic model, visualising its behaviours, including memory resizing.

CCS Concepts: • **Software and its engineering → Virtual machines**; **Assembly languages**; *Runtime environments*; *Just-in-time compilers.*

Additional Key Words and Phrases: Virtual machines, programming languages, assembly languages, just-in-time compilers, type systems

ACM Reference Format:
Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. 2019. Weakening WebAssembly. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 133 (October 2019), 28 pages. https://doi.org/10.1145/3360559

1 INTRODUCTION

WebAssembly [Haas et al. 2017] (abbreviated Wasm) is a safe virtual instruction set architecture that can be embedded into a range of *host environments*, such as Web browsers, content delivery networks, or cloud computing platforms. It is represented as a byte code designed to be just-in-time-compiled to native code on the target platform. Wasm is positioned to be an efficient compilation target for low-level languages like C++. Wasm is unusual, especially for a technology in the context

Authors' addresses: Conrad Watt, University of Cambridge, UK, conrad.watt@cl.cam.ac.uk; Andreas Rossberg, Dfinity Stiftung, Germany, rossberg@mpi-sws.org; Jean Pichon-Pharabod, University of Cambridge, UK, jean.pichon@cl.cam.ac.uk.

2475-1421/2019/10-ART133
https://doi.org/10.1145/3360559

of the Web, in that its normative specification is given as a fully formal semantics, informed by the state of the art in programming language semantics, and any new feature must be given a full formal specification before final adoption.

Wasm in its original form was purely single-threaded, however the ability to compile multi-threaded code to Wasm is considered to be of great importance by the Wasm Working Group. In order to fully support compilation of multi-threaded code to Wasm, it is necessary to extend it with threads as well as a memory consistency model [Boehm 2005]. A memory model describes the way in which architectural behaviour and compiler optimisations may combine to produce a *relaxed*, or *weak*, observed semantics of concurrent memory operations, which is not consistent with a naive sequential execution of the individual operations. Such *relaxed memory models* have become the subject of intense study in recent years, at the level of both architectural [Alglave et al. 2009; Flur et al. 2016; Higham et al. 2006; Mador-Haim et al. 2012; Owens et al. 2009] and source-level language [Batty et al. 2015; Dolan et al. 2018; Kang et al. 2017; Lochbihler 2018; Manson et al. 2005; Nienhuis et al. 2016] semantics.

It often proves difficult to balance the various concerns of developers and implementers. An intuitive, predictable source-level semantics often translates to an inefficient and error-prone implementation. The underlying hardware may exhibit weak behaviours which must be carefully mitigated by inserting memory barriers or other synchronisation primitives at compile time, and the compiler must be careful not to perform an optimisation which is valid in single-threaded code, but not when the effects of the code on memory may be observed by other threads [Boehm 2011; Ševčík and Aspinall 2008]. Conversely, a high-level language may, in order to achieve maximum performance, attempt to support all compiler optimisations and expose the union of all possible weak behaviours in the underlying hardware. As can be seen with C++11 relaxed atomics, this leads to a semantics which is almost impossible to reason about, or even circularly justified [Batty et al. 2013; Boehm and Demsky 2014; McKenney et al. 2005].

The memory model for Wasm faces a unique design pressure compared to existing work: all accesses are bounds-checked, and the bounds of Wasm's memory *address space* may be resized at runtime. The memory model must not only specify the values observed by concurrent accesses, but also the out-of-bounds behaviour of accesses in the presence of concurrent memory size alterations. Moreover, the need for a safe and portable semantics forbids any notion of undefined behaviour sneaking in.

Our contributions are as follows:

- We give the formal semantics of a concurrent extension of Wasm with threads, atomics, first-class references, and mutable tables.
- We present an axiomatic memory model for this extension which addresses the above challenges.
- We prove that our memory model is sequentially consistent for data-race-free programs (SC-DRF, see Section 6).
- We show by counter-example, verified by tool support, that JavaScript's memory model is *not* SC-DRF.
- We present an SMT-based litmus tool which visualises our semantics, including growth, and use it to experimentally validate a correspondence between our model and JavaScript's.
- We discuss compilation to and from Wasm, and the extent to which we can formally motivate correctness of compilation, given the current state of the art in relaxed memory research.

In developing the memory model, we extensively consulted with Wasm implementers and the Wasm Working Group. The model we present here is part of the wider Wasm threading specification [Smith 2019], which is still under active development and only allows concurrent

accesses to the linear memory. In this paper, we present a generalised design that already anticipates concurrent use of functions, global variables, and tables, allowing both atomic and non-atomic access to each, to illuminate the full extent of the design space.

2 BACKGROUND

Wasm is closely based on the instruction sets of real CPUs, but at the same time must be portable across hardware. Hence its memory model follows the lineage of models for low-level programming languages, rather than hardware models. To provide some necessary background, we survey the most directly relevant ancestors, the C++ and the JavaScript memory models, in the following.

2.1 The C++ Memory Model

The C++ axiomatic memory model [Batty et al. 2011; Boehm and Adve 2008] is in many ways a seminal work in the area of weak memory semantics. Objects in C++ may be declared as *atomic*, and *atomic operations* on these objects may be parameterised with one of several *consistency modes*, which form a hierarchy from relaxed to seqcst (sequentially consistent). Stronger consistency modes provide more semantic guarantees, but require additional synchronisation in the compilation scheme [Sewell and Sevcik 2016], allowing expert programmers to unlock the full performance of the underlying hardware by carefully designing their program to use weaker consistency modes, with relaxed atomics being designed to compile to bare loads and stores [Sewell and Sevcik 2016].

However, relaxed atomics are so weak that various deficiencies have been found in their semantics [Batty et al. 2013; Boehm and Demsky 2014; McKenney et al. 2005]. In particular, we have the issue of *out-of-thin-air* reads, whose value may be circularly justified, as in the notorious example where the value 42 may appear in a program that contains no constant numbers or arithmetic [Batty and Sewell 2014]. It is not expected that real hardware or compiler optimisations could ever give rise to such an astonishing execution, so clearly there is space for relaxed atomics to be given a stronger semantics while still compiling to bare loads and stores. However, properly specifying such a strengthening while still admitting all current compiler optimisations is an open problem [Batty et al. 2015].

A location which is not declared as atomic may still be accessed concurrently by multiple threads. However any *data race* involving a *non-atomic* operation triggers C++ *undefined behaviour*. Undefined behaviour in C++ is specified rather brutally. If it is potentially triggered as part of an execution, every execution of the program is allowed to have arbitrary behaviour, even for operations that took place in the past, before the behaviour is triggered, or in executions where it is not triggered at all. This is an ultimate safety valve in the specification where it would be otherwise impossible to give a sensible semantics. The initializing write to the atomic location is modelled as a non-atomic write. Aside from this, atomic locations cannot experience non-atomic accesses.

Because the C++ model contains many consistency modes and concurrency features, its full model is rather large. In order to explain its relationship with the Wasm model, it suffices for us to consider only the “C++ Model Supporting Low-Level Atomics" fragment initially described in [Boehm and Adve 2008], which supports only seqcst and non-atomic consistency modes.

To summarise the core of the model briefly, memory accesses over the course of program execution are collected as a set of abstract records, recording which location was accessed, which value was read/written, and the consistency mode. Accesses that execute sequentially in the same thread are related by *sequenced-before*. The specification guarantees that all observable executions are *valid*; that is, it must be possible to give definitions for the relations over accesses *reads-from*, *synchronizes-with*, *happens-before*, and *sc* such that the axiomatic conditions of the model hold. We reproduce these conditions below, grouping some sub-conditions as “value-consistent”,

"hb-consistent", and "sc-last-visible" to facilitate comparisons to other models presented in this paper.

- *happens-before* is a strict partial order, and *sc* is a strict total order on sequentially consistent accesses.
- *happens-before* is the transitive closure of *sequenced-before* ∪ *synchronizes-with*, and *sc* is compatible with *happens-before*.
- For all read accesses R, there must exist a write access W such that R *reads-from* W.
- For all accesses R and W, such that R *reads-from* W, the following must hold:
  - ⋆ value-consistent:
    (1) W must access the same location as R, and the observed values are consistent.
  - ⋆ hb-consistent:
    (1) It is not the case that R *happens-before* W.
    (2) W *synchronizes-with* R iff both R and W are seqcst.
    (3) There exists no W′ such that W *happens-before* W', W′ *happens-before* R, and W′ writes to the same location that R and W access,
  - ⋆ sc-last-visible:
    (1) If both R and W are seqcst, then W must be the last write to the location of R that is *sc* before R.
    (2) If R is seqcst and W is non-atomic, then there exists no W′ such that W *happens-before* W′, W′ is *sc* before R, and W′ writes to the same location that R and W access,. (†)

The condition highlighted and marked (†) was added to the model [Batty 2014; Batty et al. 2011] after the original draft was found not to guarantee Sequential Consistency of Data-Race-Free programs (SC-DRF), a crucial correctness condition (see Section 6).

The memory model of C++ is specified in a mostly formal manner. However, the wider specification is not a formal semantics. This means that there are inevitable imprecisions in how behaviour in other areas of the specification can be related to the sets of accesses manipulated by the memory model, for example in programs exhibiting undefined behaviour or non-terminating executions.

The C++ memory model implicitly relies on several language-level invariants of the C++ semantics.

- As previously mentioned, it can be assumed that there are no racing non-atomics, since otherwise the program has undefined behaviour.
- Accesses are guaranteed to be to discrete locations which never overlap each other, as a consequence of the C++ "effective type" rules.
- By the same rule, no location can experience a mixture of atomic and non-atomic accesses, except for initializing writes to atomic locations. This exception was the cause of the deficiency corrected by (†).

None of these assumptions hold for Wasm's more low-level instruction set.

2.2 The JavaScript Memory Model

JavaScript's shared memory operations are defined over *shared array buffers*, linear buffers of raw bytes that can be accessed by multiple threads in an array-like fashion through (potentially different) *data views*. Unlike C++, a single access is therefore defined as affecting the values of a range of bytes, rather than the value of a single abstract location. Moreover, since a JavaScript program may have multiple shared array buffers, events must also track *which* buffer they are accessing.

The JavaScript memory model [ECMA International 2018b] defines two consistency modes that can be used programmatically: unordered, and seqcst. While C++ models initial values as

non-atomic writes, JavaScript models them using a third consistency mode, init. The init mode functions mostly as unordered, except that it is guaranteed to occur before other events. This is strictly stronger than the C/C++ notion of initialisation, which can be delayed, while JavaScript buffers are guaranteed to be zero-initialised at the moment of their creation.

The model's core can be briefly summarised in a similar manner to that of the C++ model. Again, *sequenced-before* has the same meaning, and every execution must respect the following constraints on *reads-from*, *synchronizes-with*, and *happens-before*. However the JavaScript model uses a slightly different formulation of *sc*. Instead of a total order over only seqcst events, the model requires the existence of a total order across all events, which we will refer to as *tot* throughout the paper. This distinction is trivial, as the JavaScript model never uses *tot* to restrict the behaviour of non-atomic events, meaning that the model could equally well be formulated using *sc* (which would be *tot* restricted to seqcst events). Moreover, read and write events are now characterised by a *list* of byte values rather than a single value as in C++, and *reads-from* now relates a read event R to a list of write events, with each list element describing the source of one byte in R's range[1]. We adopt the convention that R *reads-from*(i) W describes W as the i-th event in the list.

- *happens-before* is a strict partial order, and *tot* is a strict total order on accesses.
- *happens-before* is the transitive closure of *sequenced-before* ∪ *synchronizes-with*, and *happens-before* is a subset of *tot*.
- For all read accesses R, for all $i <$ size R, there must exist a write access W such that R *reads-from*(i) W, and R and W access the same shared array buffer.
- init events happen before all other accesses with overlapping ranges to the same buffer.
- For all accesses R and W, for all $i <$ size R such that R *reads-from*(i) W, the following must hold:
  - ⋆ value-consistent:
    (1) W must access (among others) the i-th byte index of R, and the value read by R must be consistent with the value written by W at that index.
  - ⋆ hb-consistent:
    (1) It is not the case that R *happens-before* W.
    (2) W *synchronizes-with* R iff both R and W are seqcst, and affect equal byte ranges.
    (3) There exists no W′ such that W *happens-before* W′, W′ *happens-before* R, W′ writes to the i-th byte index of R, and W′ accesses the same buffer as R and W.
  - ⋆ sc-last-visible:
    (1) If both R and W are seqcst and have equal byte ranges, then W is the last seqcst write with equal byte range to R that is *tot* before R.
    (2) If R is seqcst, W is unordered, then there exists no seqcst write W′ such that W *happens-before* W′, W *happens-before* R, W′ is *tot* before R, W′ has equal byte range to R, and W′ accesses the same buffer as R and W. (†)
    (3) If R is unordered and W is seqcst, then there exists no seqcst write W′ such that W′ *happens-before* R, W *happens-before* R, W is *tot* before W′, W′ has equal byte range to W, and W′ accesses the same buffer as R and W. (‡)
  - ⋆ no-tear:
    (1) If both R and W have equal ranges, and R and W are *tear-free*, then no other access W′ and index i' can exist such that R *reads-from*(i') W′, R and W and W′ have equal ranges, and W′ is *tear-free*.

[1]This sketch glosses over the exact formal details of how indices and ranges are compared and related. The JavaScript language specification gives this aspect of the model an exceptionally complicated and prosaic definition, which is not necessary to discuss the model intuitively.

As part of this work, we identified that the JavaScript model replicated the SC-DRF violation of the uncorrected C++ model. As in C++, this violation is corrected by extending sc-last-visible with the (†) rule, slightly modified to explicitly not apply in the case of a data race between W and R, which is implicit in the C++ model. In addition, we discovered a dual violation caused by an unord read of a seqcst write. This has no direct analogy in C++, as such accesses are not permitted by the language. This violation is corrected by the (‡) rule. We have proposed the addition of both these rules to the JavaScript model. We discuss this in more detail in Section 6.

The final condition, "no-tear", describes the circumstances in which a write to multiple bytes may be decomposed into independantly observable writes to individual bytes. This is possible when dealing with non-aligned, racing writes, or writes larger than the word size of the architecture. The *tear-free* predicate describes when a write is guaranteed to be visible indivisibly to reads of identical alignment and range, even when racing. All seqcst accesses are guaranteed to be *tear-free*. Our memory model for Wasm adopts the basic approach of the JavaScript model.

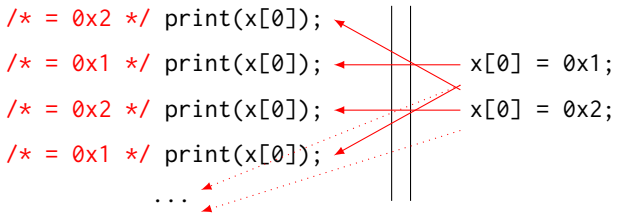

Fig. 1. A surprising JavaScript execution, permitted by its memory model in lieu of a totally undefined semantics; all arrows represent *reads-from*(0)

2.3 Contrasting C++ and JavaScript

JavaScript has found itself co-opted as an ad-hoc compilation target for C++ [Herman et al. 2014]. It is therefore not surprising that their memory models have many similarities. In a JavaScript program which respects the following conditions, it can be seen that unordered JavaScript accesses are equivalent to non-atomic C++ accesses, and JavaScript and C++ seqcst accesses are equivalent to each other, in the sense of the memory consistency behaviours that are allowed:

- There are no data races involving unordered access.
- All accesses are *naturally aligned*.
- No two accesses have overlapping but non-equal ranges.
- No access ranges beyond the bounds of the buffer.

We can observe that these restrictions effectively re-establish the language-level invariants of C++, and ensure that the byte ranges of JavaScript accesses can each be treated as a discrete location. Compilation from C++ to JavaScript can then be accomplished by allocating each shared object on a disjoint, aligned area of a shared array buffer, and promoting all C++ atomic accesses (of any consistency) to seqcst. C++ pointers to memory then become indices into a data view over the shared array buffer in the translated code.

The JavaScript and C++ models differ in the consistency behaviour that is allowed when the aforementioned conditions are not met. In C++, dereferencing a null pointer or a pointer to unallocated memory results in undefined behaviour. In JavaScript, accessing a shared array buffer at an out-of-bounds or "nonsense" index results in a regular JavaScript value that is, confusingly enough, named undefined. Consequently, executions with out-of-bounds accesses have defined behaviour. Moreover, in C++, data races and overlapping mixed-size accesses all instantly trigger undefined behaviour. The JavaScript specification instead chooses to maintain a defined behaviour, but one that is far weaker than the behaviour of real hardware. In particular, unordered accesses

which race with other accesses may be freely read from, without creating any *coherence* guarantees. This means that executions such as the one shown in Fig. 1 are possible, and well-defined behaviour according to the JavaScript specification.

We have presented the core of both the C++ and JavaScript models, focussing on the semantics of data accesses. Beyond these, the full languages contain additional features which interact with the memory model such as locks, thread creation and suspension, and so on. They are not included here because there are few similarities between the feature sets of the two languages. Generally these features imply additional *happens-before* edges, for example, if one thread spawns another, all previous actions in the spawning thread will be *happens-before* all actions in the spawned thread.

(value types)	$t ::= nt \mid rt$	(sharing)	$sh ::= \text{local} \mid \text{shared}$
(numeric types)	$nt ::= \text{i32} \mid \text{i64} \mid \text{f32} \mid \text{f64}$	(reference types)	$rt ::= sh\ \text{anyref} \mid sh\ \text{funcref}$
(sign extension)	$sx ::= \text{s} \mid \text{u}$	(function types)	$ft ::= sh\ t^* \to t^*$
(storage size)	$sz ::= 8 \mid 16 \mid 32 \mid 64$	(global types)	$gt ::= sh\ mut\ t$
(packed type)	$pt ::= sz\_sx^?$	(table types)	$tt ::= sh\ rt[n]$
(order)	$ord ::= \text{unord} \mid \text{seqcst}$	(memory types)	$mt ::= sh\ [n]$

(instructions) $e ::= \ldots \mid$ **call** $i \mid$ **call_indirect**.$ord\ ft \mid$ **global.get**.$ord\ i \mid$ **global.set**.$ord\ i \mid$
table.get.$ord \mid$ **table.set**.$ord \mid$ **table.size** $\mid$ **table.grow** $\mid$ **ref.null** $\mid$ **ref.func** $i \mid$
nt.**load**.$ord\ pt\ a\ o \mid nt$.**store**.$ord\ sz\ a\ o \mid nt$.**rmw**.$binop\ pt\ a\ o \mid$
nt.**wait** $\mid$ **notify** $\mid$ **memory.size** $\mid$ **memory.grow** $\mid$
fork $i \mid$ **instantiate** mod

(functions) $func ::= ex^*$ **func** $ft\ im \mid ex^*$ **func** ft **local** $t^*\ e^*$
(globals) $glob ::= ex^*$ **global** $gt\ im \mid ex^*$ **global** $gt\ e^*$
(tables) $tab ::= ex^*$ **table** $tt\ im \mid ex^*$ **table** $tt\ (e^*)^*$
(memories) $mem ::= ex^*$ **memory** $mt\ im \mid ex^*$ **memory** mt
(export) $ex ::=$ **export** "$name$"
(import) $im ::=$ **import** "$name$" "$name$"
(modules) $mod ::=$ **module** $func^*\ glob^*\ tab^?\ mem^?$

Fig. 2. Abstract syntax of concurrent Wasm (excerpt)

3 CONCURRENT WASM

The concurrent version of Wasm that we describe in this paper is an extension of the base language defined by the official specification [WebAssembly Working Group 2019], which itself is heavily based on [Haas et al. 2017]. Fig. 2 shows an extract of the abstract syntax of concurrent Wasm. For space reasons, we omit instructions that are not relevant to the memory model and carry over unchanged from basic Wasm.

Wasm code is organised into individual *functions* that are in turn bundled into a *module*, forming a Wasm binary. Wasm code is executed in a *host environment* that it can only interact with through a module's *imports* and *exports*. In particular, the host may invoke exported Wasm functions, and Wasm code may call imported *host functions*.

A module can also define, import, or export stateful definitions. Three forms of global state exist in Wasm: the linear *memory* providing a bounds-checked address space of raw bytes, *tables* storing and indexing opaque references to functions, and plain *global* variables. Through import and export, access to stateful definitions can be shared with other modules or the host, which can potentially

mutate them. Separate modules can define linear memories or tables separately, such that Wasm effectively supports multiple disjoint address spaces as well as the dynamic creation of new ones.

A recent proposal for Wasm [Rossberg 2018], which will soon be adopted by the standard, turns references to functions or stateful objects into first-class values and generalises the notion of table to a general store for opaque reference values. Because this extension is deeply affected by threading as well, we include and describe it here; respective constructs are highlighted in blue in Fig. 2.

References. The set of values that a program can store or compute over is codified by Wasm's notion of *value type*. As a low-level language, Wasm so far only allowed *numeric* value types (integers and floats), which also encode pointers into linear memory. A recent extension proposed to complement them with *reference types* [Rossberg 2018], which abstract physical pointers into the host system's memory. This extension enables Wasm code to safely round-trip pointers to *host objects* (such as DOM objects on the Web), which previously required bijective mappings to numbers at the language boundary and brittle manual lifetime management. The extension also enables a first-class representation of *function references*, and hence (in another extension not considered here) type-safe indirect calls. We consider only minimal support for references here, where the only types available are anyref and funcref – the former is the top type of all references, the latter includes all function references.

Wasm code can either form references from a local function index (**ref.func**) or as the null reference (**ref.null**) – both anyref and funcref are inhabited by null (future refinements to the type system will exclude it from certain types). In addition, we assume that the host environment can create unspecified forms of references and pass them to Wasm.

Unlike numeric types, whose representation is *transparent* and whose values can hence be stored into memory, reference types must be *opaque* for safety and security reasons; that is, their bit pattern must not be observable and they cannot be allowed into raw memory.

Tables. To make up for this, Wasm's existing notion of *table* is generalised. Originally, it only allowed holding function references, which was useful for emulating raw function pointers and indirect calls, especially when compiling C-like languages [Haas et al. 2017]. The reference proposal repurposes tables as a general storage for references, by allowing any reference type as element.

Accordingly, the instruction set is extended with instructions for manipulating table slots (**table.get** and **table.set**) and table size (**table.size** and **table.grow**), the latter being analogous to the existing instructions for memories.[2]

Threads. Wasm support for threading with shared-state concurrency and atomics is added by another language proposal [Smith 2019]. The current proposal only supports shared linear memory, but here we generalise it to globals, tables, and references, since such further extensions are on Wasm's long-term roadmap, and we aim to have a formalism that can handle the full enchilada seamlessly. In Fig. 2, all respective extensions are highlighted in red.

Sharing. Unlike most other languages, concurrent Wasm is explicit about which objects can be *shared* between threads, and which ones are only accessed in a thread-local manner. Accordingly, all definitions and references are complemented with a sharing annotation *sh*. These annotations make it easy for engines to pick the most efficient compilation scheme for each access to mutable state, for example, by avoiding unnecessary barriers. Validation (see supplemental material) ensures that annotations are consistent and transitive, e.g., a shared reference can only refer to shared definitions.

[2]The ability to mutate tables and their size always existed in Wasm, but was previously only accessible through the host-side API.

Atomics. In order to enable synchronisation between multiple threads, concurrent Wasm incorporates the ability to specify an *ordering* constraint (or *consistency* mode) for instructions that access a program's state. In the current proposal, which follows JavaScript in that regard, only two modes are supported: non-atomic *unordered* access (unord) and *sequentially consistent* atomic access (seqcst). Additional atomic consistency modes like acquire/release may be added as future extensions.

An ordering annotation is included in instructions for accessing the Wasm memory (*t*.**load**, *t*.**store**), as well as instructions accessing global variables (**global.get**, **global.set**) and table slots, including indirect calls (**table.get**, **table.set**, **call_indirect**). In addition, the language offers an atomic read-modify-write instruction (*t*.**rmw**) for memory access, where the modification can be drawn from a large set of binary numeric operators *binop*, whose definition we omit here. It also provides a pair of low-level **wait** and **notify** instructions that block a thread and resume blocked threads, respectively, indexed on a memory location.

Host Instructions. Although not part of the current threading proposal, which assumes that threads are only created by the host environment, we also include an instruction for spawning a thread from a function (**fork**). In addition, we provide a pseudo instruction (**instantiate**) for *instantiating* and linking a module [Haas et al. 2017].

Including these two instructions allows us to express all interesting effects that can be performed by the host environment – in particular, the dynamic creation of threads and the dynamic allocation of new pieces of shared state (including new memories, i.e., address spaces) – as Wasm code. That in turn enables us to model all relevant host computation as Wasm computation, and all interesting interactions with host threads can be expressed as interaction with other Wasm threads.

(global configuration)	$conf$	::=	$s; p^*$
(local configuration)	$lconf$	::=	$s; f; e^*$
(address)	a		
(time stamp)	h		
(threads)	p	::=	$(e^*)_h$
(frames)	f	::=	$\{\text{module } m,\ \text{local } v^*\}$
(module instances)	m	::=	$\{\text{func } a^*,\ \text{global } a^*,\ \text{table } a^?,\ \text{mem } a^?\}$
(store)	s	::=	$\{(a\ obj)^*_h\}$
(instance objects)	obj	::=	func m $func$ | global gt | table tt | mem mt (where $func$ is restricted to the form **func** ft **local** t^* e^*)
(administrative instr's)	e	::=	$\ldots$ | **ref** a | **trap** | **call'** a | **fork'** e^* | **wait'** l n | **notify'** l n m
(values)	v	::=	nt.**const** c | **ref.null** | **ref** a
(store values)	w	::=	$b \mid v$
(events)	ev	::=	$(act^*)_h \mid \epsilon$
(actions)	act	::=	$\text{rd}_{ord}\ l\ w^* \mid \text{wr}_{ord}\ l\ w^* \mid \text{rmw}\ l\ w^m\ w^m$
(field)	fld	::=	val | data | elem | length
(region)	r	::=	$a.fld$
(location)	l	::=	$r \mid r[i]$
(thread context)	P	::=	$p^*\ [\_]\ p^*$

Fig. 3. Runtime structure

4 OPERATIONAL SEMANTICS

The execution semantics of concurrent Wasm is defined in two layers: an *operational* semantics, specifying execution of individual instructions (described in this Section), and an *axiomatic* semantics, specifying the interaction with the memory (described in Section 5). Both interact through *events* that are generated by the operational semantics and "wired up" by the axiomatic semantics.

Configurations. The operational semantics is defined via two small-step reduction relations: (1) reduction of *local configurations*, i.e., individual threads, and (2) reduction of *global configurations*, i.e., the entire program state. Their definitions are given in Fig. 3.

Global configurations consist of a set of *threads* p^*, each represented by its remaining instruction sequence, annotated by a *time stamp* h whose explanation we defer to Section 5, and the global program *store* s, which records all abstract *instance objects* that have been created (i.e., allocated) by the program. These objects are the runtime instances of all entities that can be defined and ex/imported by modules, i.e., functions, globals, tables, and memories. Every object is identified by a unique abstract *address* a in the store. Every entry is also indexed with a time stamp h that indicates when it was created. We write $s(a)$ for the object associated with a in the store s, and $s_{\text{time}}(a)$ for its respective creation time h.

Local configurations consist of the store, the *frame* f of the current function, and the instruction sequence e^* left to execute. The frame records the module instance a function resides in and the state of its local variables.

Actions and Events. We make a key generalisation of existing work by defining *actions*, which record an individual access to shared state, and *events*, which are the units ordered by the consistency predicate of the axiomatic semantics, as distinct formal objects. Existing formal memory models implicitly unify these, but in our model a single event may contain multiple actions, which are all considered to be performed atomically. Both are also defined in Fig. 3. Like threads, events are annotated with a time stamp, a matter we will explain in Section 5. Per convention, we implicitly identify all events $()_h$ with no action with the empty event ϵ.

Intuitively, actions express those store operations that can be *reordered*, subject to certain conditions that the axiomatic semantics defines. An action can be one of three flavours of access to a mutable *location* l: read (rd), write (wr), or atomic read-modify-write (rmw). In each case the action records the sequence of *store values* w^* read or written – or both in the case of read-modify-write. Reads and writes may be annotated with different consistency modes; as a convention, we abbreviate $\mathsf{rd}_{\mathsf{unord}}$ and $\mathsf{wr}_{\mathsf{unord}}$ to just rd and wr, respectively.

A location describes the component of an instance object that is being accessed. It is given as an abstract address a paired with a virtual *field* name *fld*. Globals only have a val field that is their current value. Tables and memories both have a len field storing their current size, and an elem field (for tables) or data field (for memories) that denotes the respective content. The latter are *vectors* of store values (references for elem, bytes for data). Hence a location in these fields additionally involves an *offset* i.

Local Reduction. The local reduction relation is labelled by sets of actions. Figs. 4, 5, 6, 7, 8, 9, 10 show a selection of all local reduction rules that touch the store. Rules for constructs missing from the figure do not access the store and hence carry over unchanged from [Haas et al. 2017].

$$s; f; v\ (\textbf{global.set}.ord\ i) \quad \hookrightarrow^{(\mathsf{wr}_{ord}\ a.\mathsf{val}\ v)} \quad \epsilon \qquad (\text{if } a = f_{\mathsf{global}}(i))$$

$$s; f; (\textbf{global.get}.ord\ i) \quad \hookrightarrow^{(\mathsf{rd}_{ord}\ a.\mathsf{val}\ v)} \quad v \qquad (\text{if } s \vdash v : t \wedge \mathsf{type}(s(a)) = sh\ mut\ t \wedge a = f_{\mathsf{global}}(i))$$

Fig. 4. Local reduction (globals)

$$s; f; (\textbf{i32.const}\ i)\ v\ \textbf{table.set}.ord \quad \hookrightarrow^{(\text{rd}\ a.\text{len}\ n)} \quad \textbf{trap} \qquad (\text{if } a = f_{\text{table}} \wedge i \geq n)$$

$$s; f; (\textbf{i32.const}\ i)\ v\ \textbf{table.set}.ord \quad \hookrightarrow^{(\text{rd}\ a.\text{len}\ n)\ (\text{wr}_{ord}\ a.\text{elem}[i]\ v)} \quad \epsilon (\text{if } a = f_{\text{table}} \wedge i < n)$$

$$s; f; (\textbf{i32.const}\ i)\ \textbf{table.get}.ord \quad \hookrightarrow^{(\text{rd}\ a.\text{len}\ n)} \quad \textbf{trap} \qquad (\text{if } a = f_{\text{table}} \wedge i \geq n)$$

$$s; f; (\textbf{i32.const}\ i)\ \textbf{table.get}.ord \quad \hookrightarrow^{(\text{rd}\ a.\text{len}\ n)\ (\text{rd}_{ord}\ a.\text{elem}[i]\ v)} \quad v \quad (\text{if } s \vdash v : t \wedge \text{type}(s(a)) = sh\ t[n'] \wedge a = f_{\text{table}} \wedge i < n)$$

Fig. 5. Local reduction (table access)

$$f; \textbf{table.size} \quad \hookrightarrow^{(\text{rd}_{\text{seqcst}}\ a.\text{len}\ n)} \quad \textbf{i32.const}\ n \qquad (\text{if } a = f_{\text{table}})$$

$$f; (\textbf{i32.const}\ k)\ v\ \textbf{table.grow} \quad \hookrightarrow^{(\text{rd}_{\text{seqcst}}\ a.\text{len}\ n)} \quad \textbf{i32.const}\ (-1) \qquad (\text{if } a = f_{\text{table}})$$

$$f; (\textbf{i32.const}\ k)\ v\ \textbf{table.grow} \quad \hookrightarrow^{(\text{rmw}\ a.\text{len}\ n\ (n+k))\ (\text{wr}\ a.\text{elem}[n]\ v^k)} \quad \textbf{i32.const}\ n \quad (\text{if } a = f_{\text{table}})$$

Fig. 6. Local reduction (table growth)

$$s; f; (\textbf{i32.const}\ i)\ (\textbf{call\_indirect}.ord\ ft) \quad \hookrightarrow^{(\text{rd}\ a.\text{len}\ n)} \quad \textbf{trap} \qquad (\text{if } a = f_{\text{table}} \wedge i \geq n)$$

$$s; f; (\textbf{i32.const}\ i)\ (\textbf{call\_indirect}.ord\ ft) \quad \hookrightarrow^{(\text{rd}\ a.\text{len}\ n)\ (\text{rd}_{ord}\ a.\text{elem}[i]\ (\textbf{ref}\ a'))} \quad \textbf{trap} \quad (\text{if } a = f_{\text{table}} \wedge i < n \wedge \text{type}(s(a')) \neq ft)$$

$$s; f; (\textbf{i32.const}\ i)\ (\textbf{call\_indirect}.ord\ ft) \quad \hookrightarrow^{(\text{rd}\ a.\text{len}\ n)\ (\text{rd}_{ord}\ a.\text{elem}[i]\ (\textbf{ref}\ a'))} \quad \textbf{call'}\ a' \quad (\text{if } a = f_{\text{table}} \wedge i < n \wedge \text{type}(s(a')) = ft)$$

$$f; (\textbf{ref.func}\ i) \quad \hookrightarrow \quad \textbf{ref}\ f_{\text{func}}(i)$$

$$f; (\textbf{call}\ i) \quad \hookrightarrow \quad \textbf{call'}\ f_{\text{func}}(i)$$

$$s; v^n\ (\textbf{call'}\ a) \quad \hookrightarrow \quad \textbf{frame}_m\{\text{module}\ m, \text{local}\ v^n\ (t.\textbf{const}\ 0)^k\}\ e^*\ \textbf{end} \quad (\text{if } s(a) = \text{func}\ m\ (\textbf{func}\ (sh\ t_1^n \to t_2^m)\ \textbf{local}\ t^k\ e^*))$$

Fig. 7. Local reduction (functions)

$$f; (\textbf{i32.const}\ k)\ (t.\textbf{load}.ord\ pt\ a\ o) \quad \hookrightarrow^{(\text{rd}\ a.\text{len}\ n)} \quad \textbf{trap} \qquad (\text{if } a = f_{\text{mem}} \wedge k + o + |pt| > n)$$

$$f; (\textbf{i32.const}\ k)\ (t.\textbf{load}.ord\ pt\ a\ o) \quad \hookrightarrow^{(\text{rd}\ a.\text{len}\ n)\ (\text{rd}_{ord}\ a.\text{data}[k+o]\ \text{bits}_t^{pt}(c))} \quad t.\textbf{const}\ c \quad (\text{if } a = f_{\text{mem}} \wedge k + o + |pt| \leq n)$$

$$f; (\textbf{i32.const}\ k)\ (t.\textbf{const}\ c)\ (t.\textbf{store}.ord\ sz\ a\ o) \quad \hookrightarrow^{(\text{rd}\ a.\text{len}\ n)} \quad \textbf{trap} \qquad (\text{if } a = f_{\text{mem}} \wedge k + o + sz > n)$$

$$f; (\textbf{i32.const}\ k)\ (t.\textbf{const}\ c)\ (t.\textbf{store}.ord\ sz\ a\ o) \quad \hookrightarrow^{(\text{rd}\ a.\text{len}\ n)\ (\text{wr}_{ord}\ a.\text{data}[k+o]\ \text{bits}_t^{sz}(c))} \quad \epsilon \quad (\text{if } a = f_{\text{mem}} \wedge k + o + sz \leq n)$$

$$f; (\textbf{i32.const}\ k)\ (t.\textbf{const}\ c_2)\ (t.\textbf{rmw}.op\ pt\ a\ o) \quad \hookrightarrow^{(\text{rd}\ a.\text{len}\ n)} \quad \textbf{trap} \qquad (\text{if } a = f_{\text{mem}} \wedge k + o + |pt| > n)$$

$$f; (\textbf{i32.const}\ k)\ (t.\textbf{const}\ c_2)\ (t.\textbf{rmw}.op\ pt\ a\ o) \quad \hookrightarrow^{(\text{rd}\ a.\text{len}\ n)\ (\text{rmw}\ a.\text{data}[k+o]\ \text{bits}_t^{pt}(c_1)\ \text{bits}_t^{pt}(op_t(c_1,c_2)))} \quad c_1 \quad (\text{if } a = f_{\text{mem}} \wedge k + o + |pt| \leq n)$$

Fig. 8. Local reduction (memory access)

$$f; \textbf{memory.size} \hookrightarrow^{(\mathrm{rd}_{\mathrm{seqcst}}\ a.\mathrm{len}\ n)} \textbf{i32.const}\ n/64\,\mathrm{Ki} \quad (\text{if } a = f_{\mathrm{mem}})$$

$$f; (\textbf{i32.const}\ k/64\,\mathrm{Ki})\ \textbf{memory.grow} \hookrightarrow^{(\mathrm{rd}_{\mathrm{seqcst}}\ a.\mathrm{len}\ n)} \textbf{i32.const}\ (-1) \quad (\text{if } a = f_{\mathrm{mem}})$$

$$f; (\textbf{i32.const}\ k/64\,\mathrm{Ki})\ \textbf{memory.grow} \hookrightarrow^{(\mathrm{rmw}\ a.\mathrm{len}\ n\ (n+k))\ (\mathrm{wr}\ a.\mathrm{data}[n]\ (0)^k)} \textbf{i32.const}\ n/64\,\mathrm{Ki} \quad (\text{if } a = f_{\mathrm{mem}})$$

Fig. 9. Local reduction (memory growth)

$$f; (\textbf{i64.const}\ q)\ (t.\textbf{const}\ c)\ (\textbf{i32.const}\ k)\ t.\textbf{wait} \hookrightarrow \textbf{trap} \quad (\text{if } k \bmod |t| \neq 0)$$

$$f; (\textbf{i64.const}\ q)\ (t.\textbf{const}\ c)\ (\textbf{i32.const}\ k)\ t.\textbf{wait} \hookrightarrow^{(\mathrm{rd}\ a.\mathrm{len}\ n)} \textbf{trap} \quad (\text{if } a = f_{\mathrm{mem}} \wedge k + |t| > n)$$

$$f; (\textbf{i64.const}\ q)\ (t.\textbf{const}\ c)\ (\textbf{i32.const}\ k)\ t.\textbf{wait} \hookrightarrow^{(\mathrm{rd}\ a.\mathrm{len}\ n)\ (\mathrm{rd}_{\mathrm{seqcst}}\ a.\mathrm{data}[k]\ b^*)} \textbf{wait'}\ a.\mathrm{data}[k]\ q \quad (\text{if } a = f_{\mathrm{mem}} \wedge k + |t| \leq n \wedge b^* = \mathrm{bits}_t(c))$$

$$f; (\textbf{i64.const}\ q)\ (t.\textbf{const}\ c)\ (\textbf{i32.const}\ k)\ t.\textbf{wait} \hookrightarrow^{(\mathrm{rd}\ a.\mathrm{len}\ n)\ (\mathrm{rd}_{\mathrm{seqcst}}\ a.\mathrm{data}[k]\ b^*)} \textbf{i32.const}\ 1 \quad (\text{if } a = f_{\mathrm{mem}} \wedge k + |t| \leq n \wedge b^* \neq \mathrm{bits}_t(c))$$

$$f; (\textbf{i32.const}\ m)\ (\textbf{i32.const}\ k)\ \textbf{notify} \hookrightarrow \textbf{notify'}\ a.\mathrm{data}[k]\ 0\ m \quad (\text{if } a = f_{\mathrm{mem}} \wedge n \leq m)$$

$$s; f; v^n\ (\textbf{fork}\ i) \hookrightarrow \textbf{fork'}\ (v^n\ (\textbf{call'}\ a)) \quad (\text{if } a = f_{\mathrm{func}}(i) \wedge \mathrm{type}(s(a)) = \mathrm{shared}\ t^n \to \epsilon)$$

$$s; (\textbf{ref}\ a_{\mathrm{i}})^*\ (\textbf{instantiate}\ mod) \hookrightarrow^{(\mathrm{wr}\ l\ w^*)^*} s\ (a\ obj)^*_{l}; (\textbf{ref}\ a_{\mathrm{e}})^* \quad (\text{side conditions omitted})$$

Fig. 10. Local reduction (thread synchronization and creation)

Let us start with the simplest rules, those for accessing globals (**global.get**, **global.set**, Fig. 4). They look up the address a of the global under its static index i in the current frame's module instance and then perform a single action to read or write the val field of that global, with the appropriate memory ordering. The respective value v that is observed by these actions is picked non-deterministically in these rules, but the axiomatic semantics will constrain this choice such that each read matches up with some write to the same location (Section 5).

Accessing tables (**table.get**, **table.set**, Fig. 5) is more interesting. It involves an index that could be out of bounds, but the bounds may be dynamically altered by the execution of a **table.grow** instruction. It is here where multiple actions per event come into play, because such an access involves an *implicit*, observable access to the table's len field as well, to check its size.[3] The size is a value of the form (**i32.const** n), which we abbreviate to n here. If $n \leq i$ then the access is out of bounds and execution traps. Otherwise, the actual read or write action for the table slot at the indexed location is also performed. Both these actions are to be performed as one atomic event. The size may also be *explicitly* queried (**table.size**, Fig. 6), and updated (**table.grow**, Fig. 6). The chosen consistency behaviours of different implicit and explicit methods that observe the table size must support desired compilation schemes. Implicit bounds checks are permitted to observe changes to the table size with a semantics that is not sequentially consistent, for reasons which will be justified in Section 4.1.

[3] An implementation might use more efficient ways to implement this check, e.g., via hardware protection, but the semantics must be the same.

Some other instructions deal with manipulating function references (Fig. 7) , which may be stored in tables. The (type-annotated) **call_indirect** instruction dynamically retrieves a function from a table of function references and calls it. The **ref.func** instruction returns a first-class function reference for the function at the static index i. When a function is called, a new local frame is created for the function's body, with the function arguments becoming new local variables.

Loads and stores to memory (t**.load**, t**.store**, Fig. 8) work analogously to table accesses, except that they (1) add an additional static offset o, and (2) operate on a *sequence* of multiple bytes at once and interconvert to a numeric value, by interpretation through the meta function bits [Haas et al. 2017]. The latter has the additional implication that the axiomatic semantics allows non-atomic accesses to *tear*, i.e., reads can observe individual bytes from multiple different writes. Instructions manipulating memory size (**memory.size**, **memory.grow**, Fig. 9) are similar to the table analogues, the only complication being that size values are given in units of *page size*, which is 64 KiB.

The next bunch of instructions (**wait**, **notify**, **fork**, Fig. 10) are related to threads. Their semantics is mostly defined by the global reduction relation, while the local relation only handles their operands and respective side conditions. To that end, these instructions are reduced to auxiliary *administrative* variants that carry the final operands, to be picked up by global reduction. The t**.wait** instruction performs a bounds-checked read and suspends if the read value equals the operand, or immediately returns 1 otherwise. It traps if the access is out of bounds or unaligned (a behaviour chosen to align with common hardware). Suspending is represented by the administrative variant **wait'** that records the location and the time-out value q (in nanoseconds). The symmetric operation is **notify**, which, given a memory index k and a number m, will notify at most m threads waiting for the same location to wake them up. This is analogously modelled by an administrative variant **notify'** recording the location, the number n of woken threads (0 initially) and the maximum m. The **fork** instruction performs a function call in a new thread by looking up the function in the local frame and then forking the actual call via the auxiliary **fork'**.

Lastly, the **instantiate** instruction creates a new module instance. It takes a reference for each import, allocates and initialises the instance objects attached to the module, and returns a reference for each export. It is this instruction that extends the store with new objects. Details can be found in the supplemental material.

$$\frac{s; f_\epsilon; e^* \hookrightarrow^{act^*} s \uplus s'; f_\epsilon; e'^* \qquad h \prec_{\mathbf{hb}} h' \qquad (s_{\text{time}}(\text{addr}(act)) \prec_{\mathbf{hb}} h')^* \qquad s' = \{(a\ obj)^*_{h'}\}}{s; (e^*)_h\ p^* \hookrightarrow^{(act^*)_{h'}} s \uplus s'; (e'^*)_{h'}\ p^*}$$

$$\frac{h \prec_{\mathbf{hb}} h'}{s; (E[\mathbf{fork'}\ e^*])_h\ p^* \hookrightarrow s'; (E[\epsilon])_{h'}\ (e^*)_{h'}\ p^*}$$

$$\frac{n < m \qquad h \prec_{\mathbf{hb}} h'}{s; (E[\mathbf{wait'}\ l\ q])_h\ (E'[\mathbf{notify'}\ l\ n\ m])_{h'}\ p^* \hookrightarrow s; (E[\mathbf{i32.const}\ 0])_{h'}\ (E'[\mathbf{notify'}\ l\ (n+1)\ m])_{h'}\ p^*}$$

$$\frac{n = m \vee \neg\exists E', h', q,\ h' \prec_{\mathbf{hb}} h \wedge (E'[\mathbf{wait'}\ l\ q])_{h'} \in p^*}{s; (E[\mathbf{notify'}\ l\ n\ m])_h\ p^* \hookrightarrow s; (E[\mathbf{i32.const}\ n])_h\ p^*}$$

$$\frac{\text{signed}(q) \geq 0 \qquad h \prec_{\mathbf{hb}} h'}{s; (E[\mathbf{wait'}\ l\ q])_h\ p^* \hookrightarrow s; (E[\mathbf{i32.const}\ 2])_{h'}\ p^*}$$

Fig. 11. Global reduction

Global reduction. Fig. 11 defines global reduction. In all rules we assume that the sequence p^* is in fact a set that can be implicitly reordered.

The main rule non-deterministically selects a thread and advances it one step by invoking the local reduction relation with an empty frame f_ϵ. The previous time stamp h of the thread is replaced with a newer h', which also is taken as the time stamp of the atomic event formed by the performed actions and of any newly allocated objects in the store's extension s'. As we will see soon, it is this condition that imposes *program order* on all events within the same thread. Another side condition checks that the actions only refer to object addresses that have previously been allocated in the store, at an earlier time, ensuring respective causality and avoiding use-before-definition. This obviates the need for the ad-hoc init consistency mode of the JavaScript model.

The effect of the **fork'** instruction is to simply add a new thread to the configuration. Both the new and the originating thread will be assigned the same new time stamp h'.

The interaction between **wait'** and **notify'** on a common location l is modelled by a kind of reaction rule. It reduces **wait'** to the result 0, which indicates successful notification to the program, thereby waking up the thread for future local reductions. This time, the side condition $h \prec_{\mathbf{hb}} h'$ enforces an ordering relation between the thread performing the notification (at time h') and the thread woken up (which was suspended earlier at time h). The **notify'** instruction keeps iterating with an increased wake count n. Once n reaches m, or no more matching waits can be found, the next rule finalises the operation and returns n.

A **wait'** instruction may also time out, if (the signed interpretation of) its time-out value is not negative (indicating infinite timeout). Since there is no consistent notion of execution speed across platforms, time-out is simply modelled as a non-deterministic reduction in the semantics; a result of 2 indicates this outcome to the program.

4.1 Bounds Checks

As previously discussed, all accesses to memories or tables are bounds-checked, and out-of-bounds access will immediately **trap**. There have been ongoing discussions with implementers regarding how they expect to be able to compile Wasm accesses to memories and tables, considering their bounds-checking behaviour [Hansen 2017]. A key motivating example is given in Fig 12. Given desirable implementation schemes, one must ask "if the first access of thread 1 (c) succeeds, requiring the memory growth (b) to be "visible", is it guaranteed that the store of (a) is visible to (d)?" This is a standard *message passing* (MP) shape, but with some data accesses replaced by length accesses.

More efficient implementation techniques on modern hardware notwithstanding, it is inevitable that some platforms will have to compile explicit bounds checks, as noted in the threads proposal [Smith 2019]. In that case, a large section of memory is pre-allocated, and a real memory location is used to store the current maximum bound. This way, a **memory.grow** or **table.grow** instruction can be implemented as a simple atomic increment of this location, without the need for further allocation. All compiled Wasm accesses are then guarded by a conditional jump to a **trap** procedure based on the current value of this location.

Thread 0:	Thread 1:
a: (**i32.store** (**i32.const** k) (**i32.const** 54))	c: (**i32.load** (**i32.const** j))
b: (**memory.grow** ...)	d: (**i32.load** (**i32.const** k))

Fig. 12. A message-passing (MP) shape involving memory growth (assuming index k is in-bounds, and index j is out-of-bounds before the growth (b), and in-bounds after). Even if (c) successfully executes, it is not guaranteed that (d) will read 54.

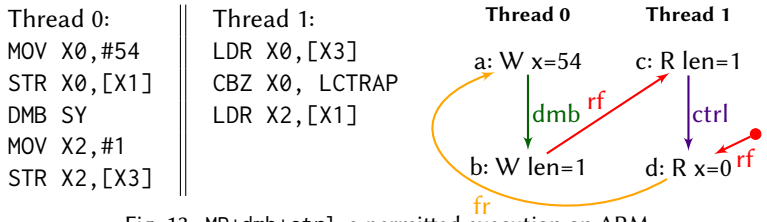

Fig. 13. MP+dmb+ctrl, a permitted execution on ARM

This leads to a natural view of a WebAssembly memory and table instruction as abstractly carrying out up to *two* accesses. As well as accessing data in memory, the instruction will also access a distinguished "length" location, to perform the bounds check. Both accesses are potentially subject to relaxed behaviour, since implementers wish to compile this bounds check as a bare architectural load with few ordering guarantees, for efficiency reasons.

We use this scheme as the basis for our formal specification of the relaxed behaviour of bounds checking, considering it the weakest discipline we are prepared to support. As detailed in Fig. 19, Wasm unord accesses correspond to bare architectural load/stores. We model the implicit bounds check on each load/store operation as an additional unord access to a distinguished len abstract location. The **memory.grow** and **table.grow** instructions are modeled with an atomic rmw increment, and explicit **memory.size** or **table.size** checks are modeled as atomic reads of the len location. This means that explicit length checks (for example to ensure that a **trap** does not occur later in the code) guarantee that subsequent instructions will observe the same (or greater) length, and all side-effects from instructions before the last growth. However, implicit observations about the length, through the success or failure of bounds checks, guarantee no synchronization whatsoever. This means that the example of Fig 12 is allowed to exhibit a non-SC behaviour.

This can be justified at two levels. First, any program fragment observing relaxed behaviour in its bounds checks must have a race between a **memory.grow**/**table.grow** in one thread and a regular access in another thread, such that this access is out-of-bounds "before" the grow but in-bounds "after". Even forgetting about relaxed behaviour, such a program is clearly wrong, and will exhibit executions which **trap**, depending on the interleaving of the two threads. We are not interested in providing strong guarantees for such programs when it may restrict our range of implementation choices and optimisations, or complicate the model. For an access which is out-of-bounds before a racing grow "commits", but is in-bounds after, our semantics makes it entirely non-deterministic whether the access will succeed or **trap**, independent of the success or failure of other accesses.

Second, this scheme, implemented on real architectures, genuinely exhibits some counter-intuitive relaxed behaviours. We give a example of this on the ARM architecture, in Fig. 13. The execution, a previously studied ARM litmus test, was verified using the rmem tool [Gray et al. 2015], which can explore and visualise the possible relaxed behaviours of program fragments in various architectures. It can be viewed as an abstraction (for brevity) of the compiled code of Fig. 12, sufficent to depict its memory consistency properties. The **memory.grow** operation is represented as a store of the new length, guarded by a barrier. Compilation of a real Wasm program would generate an atomic read-modify-write here, however only the initial barrier and the write are relevant to the consistency behaviour of this example. Thread 1 represents a bounds check-guarded unord load which is racing with the **memory.grow**. After the load of the memory size, there is a conditional branch to a trap label which will carry out error-handling in the case that the access is out of bounds. The precise condition of the branch is not relevant to the example's consistency behaviour, so we choose CBZ for brevity. The rmem tool shows that the ARM memory model allows

the execution depicted, where despite a barrier and control dependency, it is possible for access d to read 0.

Another interesting shape is depicted in Fig. 14. This example arises out of the lack of read-read coherence (CoRR) in the definition of our unord accesses (discussed in more detail in Section 5). Real architectures mostly guarantee the absence of CoRR violations, with a few exceptions [Alglave et al. 2015]. This behaviour could also arise out of other implementation schemes, which may be unable to guarantee that the entire growth is atomic. For example, (a) could be an especially large growth of multiple pages which must be allocated piecemeal, with indexes j and k falling far apart from each other.

Production engines implement efficient compilation schemes for bounds checks involving virtual memory manipulation and trap handlers. These implementations allow Wasm accesses to be compiled without explicit bounds checks, and instead rely on catching and handling OS/CPU faults to detect out-of-bounds accesses. This approach is expected by implementers to be at least as strong as the naïve strategy. However such implementations are difficult to reason about formally, as discussed in Section 7.1.

Thread 0:		Thread 1:
a: (**memory.grow** ...)	‖	b: (**i32.store** (**i32.const** j) ...)
	‖	c: (**i32.store** (**i32.const** k) ...)

Fig. 14. A coherence-read-read (CoRR) shape involving memory growth (assuming indexes j, k are out-of-bounds before the growth (a), and in-bounds after). Even if (b) successfully executes, it is not guaranteed that (c) will be in-bounds.

5 AXIOMATIC MEMORY MODEL

As discussed earlier, the top-level intuition for an axiomatic memory model is that the operational semantics generates a set of events, which is then subject to a consistency predicate that classifies whole executions as either valid or invalid. Fig. 15 and Fig. 16 define everything that is needed for our axiomatic memory model. Unlike the C++ semantics, but like the JavaScript one, our semantics does not introduce undefined behaviour.

Time Stamps. The C++ and JavaScript memory models capture the ordering of events by defining various *post-hoc* relations over the event set as part of a *candidate execution* (see Section 2). This results in an graph structure, where the vertices are memory events, with the operational semantics fixing some edges such as *sequenced-before*, while some others are non-deterministically picked in the candidate execution and are later constrained by the axiomatic memory model, such as *reads-from*.

We instead chose to adopt a more compact representation based on *time stamps*, in the style of the promising semantics [Kang et al. 2017] or the OCaml memory model [Dolan et al. 2018]. This is equivalent to defining an explicit graph, but has the advantage that the operational semantics does not need to manipulate graph edges. Our time stamps are drawn from an infinite set of abstract objects that is equipped with an *a priori* partial order (written $\prec_{\mathbf{hb}}$, pronounced "happens before") corresponding to the *happens-before* relation, as well as a total order ($\prec_{\mathbf{tot}}$, pronounced "tot"), corresponding to a *total memory order*, such that $\prec_{\mathbf{hb}} \subseteq \prec_{\mathbf{tot}}$. Our operational semantics (Section 4) assigns time stamps to events eagerly but non-deterministically. Individual threads keep track of the time stamp of their last emitted event, and force future same-thread events to be ordered later by $\prec_{\mathbf{hb}}$, mimicking the effect of explicit *sequenced-before* edges. Otherwise, the time stamp choices

are *constrained* by various ordering conditions imposed in the axiomatic memory model. Therefore, a Wasm candidate execution consists of a set of time stamped events, and is *valid* if the time stamps are chosen such that the axiomatic model is satisfied. In valid executions, all events have distinct time stamps.

Our definition of $\prec_{\mathbf{hb}}$ also allows us to avoid the inter-thread synchronisation specification mechanisms of C++ and JavaScript, which rely on an *additional-synchronizes-with* relation to specify the effects of mutexes and other thread-blocking primitives on the axiomatic model [Batty et al. 2011]. For example, when thread A executes **notify'**, waking **wait'**-ing thread B, the respective reduction rule (Fig. 11, 3rd rule) specifies that thread B must advance its "last observed" time stamp to that of A, meaning that all subsequent events from B will observe previous events from A. In this capacity, time stamps function as a Lamport clock [Lamport 1978].

Traces. As seen in Section 4, our (global) small-step reduction relation is labelled with (possibly empty) events. An execution *trace* can be defined as the set of events generated by the *coinductive closure* of the reduction relation, as per the traces relation defined in Fig. 16. This relation is satisfied by all finite, terminated traces, but also by all infinite traces of non-terminating programs. The consistency predicate for a *valid* execution can then be defined over these traces.

Because we do not model garbage collection of store objects or terminated threads, the store s and the "list" of threads p^* in configurations considered in the fixpoint of a trace could become infinitely large. By slight abuse of notation, we take their "grammar" given in Fig. 3 to actually define these components as proper mathematical sets.

Auxiliary definitions:

$$
\begin{array}{lll}
\mathrm{ord}(\mathrm{rd}_o\ l\ w^*) := o & \mathrm{loc}(\mathrm{rd}_o\ l\ w^*) := l & \mathrm{size}(\mathrm{rd}_o\ l\ w^n) := n \\
\mathrm{ord}(\mathrm{wr}_o\ l\ w^*) := o & \mathrm{loc}(\mathrm{wr}_o\ l\ w^*) := l & \mathrm{size}(\mathrm{wr}_o\ l\ w^n) := n \\
\mathrm{ord}(\mathrm{rmw}\ l\ w_1^*\ w_2^*) := \mathrm{seqcst} & \mathrm{loc}(\mathrm{rmw}\ l\ w_1^*\ w_2^*) := l & \mathrm{size}(\mathrm{rmw}\ l\ w_1^n\ w_2^n) := n
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{read}(\mathrm{rd}_o\ l\ w^*) := w^* & \mathrm{write}(\mathrm{rd}_o\ l\ w^*) := \epsilon \\
\mathrm{read}(\mathrm{wr}_o\ l\ w^*) := \epsilon & \mathrm{write}(\mathrm{wr}_o\ l\ w^*) := w^* \\
\mathrm{read}(\mathrm{rmw}\ l\ w_1^*\ w_2^*) := w_1^* & \mathrm{write}(\mathrm{rmw}\ l\ w_1^*\ w_2^*) := w_2^*
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{addr}(act) := \mathrm{addr}(\mathrm{region}(act)) & \mathrm{region}(act) := \mathrm{region}(\mathrm{loc}(act)) & \mathrm{offset}(act) := \mathrm{offset}(\mathrm{loc}(act)) \\
\mathrm{addr}(l) := \mathrm{addr}(\mathrm{region}(r)) & \mathrm{region}(r) := r & \mathrm{offset}(r) := 0 \\
\mathrm{addr}(a_{fld}) := a & \mathrm{region}(r[i]) := r & \mathrm{offset}(r[i]) := i
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{range}(act) & := & [\mathrm{offset}(act), \mathrm{offset}(act) + \mathrm{size}(act)[\\
\mathrm{reading}(act) & :\Leftrightarrow & \mathrm{read}(act) \neq \epsilon \\
\mathrm{writing}(act) & :\Leftrightarrow & \mathrm{write}(act) \neq \epsilon \\
\mathrm{aligned}(act) & :\Leftrightarrow & \exists n.\ \mathrm{offset}(act) = n \cdot \mathrm{size}(act) \\
\mathrm{tearfree}(act) & :\Leftrightarrow & \mathrm{ord}(act) = \mathrm{seqcst} \vee (\mathrm{aligned}(act) \wedge \mathrm{size}(act) \leq 4) \\
\mathrm{same}(act_1, act_2) & :\Leftrightarrow & \mathrm{region}(act_1) = \mathrm{region}(act_2) \wedge \mathrm{range}(act_1) = \mathrm{range}(act_2) \\
\mathrm{overlap}(act_1, act_2) & :\Leftrightarrow & \mathrm{region}(act_1) = \mathrm{region}(act_2) \wedge \mathrm{range}(act_1) \cap \mathrm{range}(act_2) \neq \emptyset
\end{array}
$$

$$
\begin{array}{lcl}
ev_1 \prec ev_2 & :\Leftrightarrow & \mathrm{time}(ev_1) \prec \mathrm{time}(ev_2) \\
\mathrm{time}((act^*)_h) & := & h \\
\mathrm{access}_r((act^*)_h) & := & act' \quad \text{iff } \{act'\} = \{act \in act^* \mid \mathrm{region}(act) = r\} \\
f_r(ev) & := & f(\mathrm{access}_r(ev)) \\
f_r(Ev) & := & \{ev \in Ev \mid f_r(ev)\}
\end{array}
$$

Fig. 15. Axiomatic memory model, auxiliary definitions

Terminal configuration:

$$\begin{array}{rcl} p_{\text{term}} & ::= & v^* \mid \textbf{trap} \mid E[(\textbf{wait'}\ l\ q)] \\ conf_{\text{term}} & ::= & s; p^*_{\text{term}} \end{array}$$

Can synchronise with:

$$\text{sync}_r(ev_1, ev_2) \quad :\Leftrightarrow \quad \text{ord}_r(ev_1) = \text{ord}_r(ev_2) = \text{seqcst} \wedge \text{same}_r(ev_1, ev_2)$$

Trace:

$$\frac{}{\emptyset\ \text{traces}\ conf_{\text{term}}} \qquad \frac{conf \hookrightarrow^{ev} conf' \qquad tr\ \text{traces}\ conf'}{tr \uplus ev\ \text{traces}\ conf}$$

Valid trace:
(Note: by construction, $\prec_{\textbf{hb}} \subseteq \prec_{\textbf{tot}}$)

$$\frac{\forall r, \vdash_r tr\ \text{valid}}{\vdash tr\ \text{valid}}$$

$$\frac{\forall ev_R \in \text{reading}_r(tr), \exists ev^*_W, tr \vdash_r ev_R\ \text{reads-each-from}\ ev^*_W}{\vdash_r tr\ \text{valid}}$$

$$\frac{\begin{array}{c} \left|ev^*_W\right| = |\,\text{read}_r(ev_R)| \\ \forall i < |ev^*_W|, tr \vdash^i_r ev_R\ \text{reads-from}\ (ev^*_W[i]) \\ \vdash_r ev_R\ \text{no-tear}\ ev^*_W \end{array}}{tr \vdash_r ev_R\ \text{reads-each-from}\ ev^*_W}$$

$$\frac{\begin{array}{lll} ev_R \neq ev_W & tr \vdash^{i,k}_r ev_R\ \text{value-consistent}\ ev_W & \\ ev_W \in \text{writing}_r(tr) & tr \vdash^k_r ev_R\ \text{hb-consistent}\ ev_W & tr \vdash_r ev_R\ \text{sc-last-visible}\ ev^*_W \end{array}}{tr \vdash^i_r ev_R\ \text{reads-from}\ ev_W}$$

$$\frac{\begin{array}{c} \text{read}_r(ev_R)[i] = \text{write}_r(ev_W)[j] \\ k = \text{offset}_r(ev_R) + i = \text{offset}_r(ev_W) + j \end{array}}{tr \vdash^{i,k}_r ev_R\ \text{value-consistent}\ ev_W}$$

$$\frac{\begin{array}{ll} \neg(ev_R \prec_{\textbf{hb}} ev_W) & \forall ev'_W \in \text{writing}_r(tr), \\ \text{sync}_r(ev_W, ev_R) \Rightarrow ev_W \prec_{\textbf{hb}} ev_R & ev_W \prec_{\textbf{hb}} ev'_W \prec_{\textbf{hb}} ev_R \Rightarrow k \notin \text{range}_r(ev'_W) \end{array}}{tr \vdash^k_r ev_R\ \text{hb-consistent}\ ev_W}$$

$$\frac{\begin{array}{l} \forall ev'_W \in \text{writing}_r(tr), ev_W \prec_{\textbf{hb}} ev_R \Rightarrow \\ \quad ev_W \prec_{\textbf{tot}} ev'_W \prec_{\textbf{tot}} ev_R \wedge \text{sync}_r(ev_W, ev_R) \Rightarrow \neg\, \text{sync}_r(ev'_W, ev_R) \\ \quad ev_W \prec_{\textbf{hb}} ev'_W \prec_{\textbf{tot}} ev_R \Rightarrow \neg\, \text{sync}_r(ev'_W, ev_R) \qquad (\dagger) \\ \quad ev_W \prec_{\textbf{tot}} ev'_W \prec_{\textbf{hb}} ev_R \Rightarrow \neg\, \text{sync}_r(ev_W, ev'_W) \qquad (\ddagger) \end{array}}{tr \vdash_r ev_R\ \text{sc-last-visible}\ ev_W}$$

$$\frac{\text{tearfree}_r(ev_R) \Rightarrow |\{ev_W \in ev^*_W \mid \text{same}_r(ev_R, ev_W) \wedge \text{tearfree}_r(ev_W)\}| \leq 1}{\vdash_r ev_R\ \text{no-tear}\ ev^*_W}$$

Fig. 16. Axiomatic memory model

Validity. The valid predicate encodes the conditions under which a trace is considered a valid execution. Our top-level quantification over regions, r, captures not only that reads-from related events must access the same memory, but also that they must also access the same *field*. For regular memories, the two possible fields are data, representing the values of memory locations, and len, representing the memory's current length. Analogously, tables have an elem field representing the array of reference values and len for their current lengths. Globals only have a value field, representing the current value they hold. All these fields are handled uniformly by the semantics.

Our per-region validity predicate specifies, via reads-each-from, that the values read by all read events must be associated with an appropriate list of write events, with each write event in the list being the source of one store value (i.e., an individual byte in the case of memories). The reads-each-from predicate enforces the well-formedness condition that a read event of n store values must be associated with a list of precisely n write events.

The reads-each-from predicate's sub-conditions are named so as to be analogous to the prose JavaScript model we present in Section 2.2. The value-consistent relation enforces some basic well-formedness over the indexes of read-write pairs (i.e., a store value can only be read from the same index it was written). The hb-consistent predicate captures a core part of the model; the treatment of $\prec_{\mathbf{hb}}$ as a *strong ordering*, which means that all read-write pairs must be consistent with it. Observations made by individual unord reads do not guarantee $\prec_{\mathbf{hb}}$, and therefore confer no ordering guarantees to the rest of the program. By contrast, a seqcst read confers additional guarantees (*synchronisation*, implying $\prec_{\mathbf{hb}}$) if and only if it reads from a seqcst write of equal range. If this condition is not fulfilled, such a read-write pair is treated identically to the unord case. In JavaScript, synchronization is recorded with an explicit *synchronizes-with* relation. We inline this as the condition $\text{sync}_r(ev_R, ev_W) \Rightarrow ev_W \prec_{\mathbf{hb}} ev_R$. The sync predicate expresses the circumstances under which a pair of accesses may restrict which writes are observable in the rest of the program. Note that this predicate requires both accesses to be seqcst *and* access the same location range (discussed below).

Read-write pairs satisfying sync are additionally constrained by sc-last-visible. This ensures that same-range seqcst events will act in a sequentially consistent way (see Section 6), and requires these events to respect the weaker $\prec_{\mathbf{tot}}$ ordering. Finally, the no-tear predicate replicates the no-tear condition of JavaScript.

Note that the memory model (like JavaScript), has no notion of *coherence*, an explicit same-location ordering which appears in some other memory models. Wasm unord accesses do not enjoy any coherence guarantees, and coherence for seqcst accesses is subsumed by sc-last-visible.

Mixed-size Behaviour. As discussed, accesses in Wasm are to ranges of bytes rather than to discrete locations. Several ordering guarantees of the model require that accesses have identical ranges.

The no-tear rule only applies to same-range accesses. For example, an 8-byte tear-free write may still appear to tear when observed by a 4-byte tear-free read.

Atomic seqcst accesses also only provide stronger ordering guarantees when interacting with other same-range accesses. Otherwise, they are essentially identical to unord accesses. For example, as discussed, same-range seqcst read-write pairs synchronize, represented by an inter-thread $\prec_{\mathbf{hb}}$ ordering. However, mixed-size pairs will not synchronize. Moreover, sc-last-visible allows mixed-size seqcst accesses to interact in a way that does not respect $\prec_{\mathbf{tot}}$.

One rule that constrains even mixed-size accesses is hb-consistent. The model treats $\prec_{\mathbf{hb}}$ as a strong ordering, and even mixed-size read-write pairs must respect it. This means that mixed-size accesses can still be productively used, given appropriate synchronization.

As explored in recent work [Flur et al. 2017], mixed size guarantees are under-explored, and surprisingly weak on hardware, so Wasm, like JavaScript, picks a maximally weak (but defined) semantics. However, as discussed by [Flur et al. 2017], some hyper-optimised low-level data structures rely on mixed-size consistency guarantees which our model does not currently provide. As our formal understanding of mixed-size accesses grows, it should become possible for us to give more guarantees.

Thin-Air Behaviour. Unlike C++, every well-typed Wasm program has a well-defined semantics. Racing non-atomics will not trigger undefined behaviour in the C++ sense. However, the defined semantics in this case is very weak, to the point that it is still recommended for Wasm programs to be race-free. For example, Wasm non-atomics are specified weakly enough to exhibit out-of-thin-air behaviours when racing [Batty et al. 2015]. These behaviours are known to impair modular reasoning about program properties [Batty et al. 2013; Ševčík 2011]. However, unlike relaxed atomics, the C++ primitive that results in out-of-thin-air executions, it is reasonable to expect that a Wasm program should contain no data races on non-atomics, since the source program it was compiled from will disallow this. At least for Wasm code generated from C++, a data race will trigger undefined behaviour at the source-level, meaning that all Wasm programs generated from well-defined C++ should already be data-race-free. Moreover, we guarantee that all observed references have actually been allocated previously (Fig. 11), while C++ is ambiguous as to the thin-air behaviour of pointers.

Even in the case that a data race does occur, such races are "bounded in space" by the Wasm semantics. This property, coined by Dolan et al. [2018], states (informally) that a data race cannot affect the results of computations involving only unrelated locations. This property is not true of the C/C++11 model, because a data race results in undefined behaviour, but is true of our model.

$$\frac{ev \in tr \qquad ev' \in tr \qquad \vdash_r ev \text{ data-race-with } ev'}{\vdash tr \text{ data-race}} \qquad \frac{\neg \text{sync}_r(ev, ev') \qquad \vdash_r ev \text{ race-with } ev'}{\vdash_r ev \text{ data-race-with } ev'}$$

$$\frac{\neg(ev \prec_{\mathbf{hb}} ev' \vee ev' \prec_{\mathbf{hb}} ev) \qquad \text{writing}_r(ev) \vee \text{writing}_r(ev') \qquad \text{overlap}_r(ev, ev')}{\vdash_r ev \text{ race-with } ev'}$$

$$\frac{\forall r, \forall ev_R \in \text{reading}_r(tr), \exists ev^*_W,\ tr \vdash_r ev_R \text{ seqcst-reads-each-from } ev^*_W}{\vdash tr \text{ is-seqcst}}$$

$$\frac{|ev^*_W| = |\,\text{read}_r(ev_R)| \qquad (tr \vdash^i_r ev_R \text{ seqcst-reads-from } ev_W)^*_i}{tr \vdash_r ev_R \text{ seqcst-reads-each-from } ev^*_W} \qquad \frac{\begin{array}{l} ev_W \prec_{\mathbf{tot}} ev_R \qquad ev_W \in \text{writing}_l(tr) \\ tr \vdash^{i,k}_r ev_R \text{ value-consistent } ev_W \\ \forall ev'_W \in \text{writing}_r(tr), \\ \quad ev_W \prec_{\mathbf{tot}} ev'_W \prec_{\mathbf{tot}} ev_R \Rightarrow k \notin \text{range}_r(ev'_W) \end{array}}{tr \vdash^i_r ev_R \text{ seqcst-reads-from } ev_W}$$

Fig. 17. Formulation of the SC-DRF property

6 SEQUENTIAL CONSISTENCY OF DATA-RACE-FREE PROGRAMS

Sequential Consistency of Data-Race-Free programs (SC-DRF) is considered by many to be *the* desirable correctness property for a relaxed memory model [Adve and Hill 1990; Boehm and Adve 2008; Gharachorloo et al. 1992; Lahav et al. 2017]. A data-race-free program is one which does not have two non-atomic accesses, at least one of which is a write, in a race condition on the same memory location. SC-DRF guarantees that a program lacking such races will exhibit sequentially consistent behaviour, in the sense that the program will appear to execute as a naïve sequential interleaving of the operations of each thread, regardless of how weakly-specified its non-atomics are.

6.1 Wasm is SC-DRF

The axiomatic model presented in Fig. 16 is SC-DRF:

Proposition 6.1 (Wasm is SC-DRF).
$(\vdash tr \text{ valid}) \wedge \neg(\vdash tr \text{ data-race}) \implies \vdash tr \text{ is-seqcst}$

A proof of this property can be found in the supplemental materials. The auxiliary relations data-race and is-seqcst are defined in Fig. 17. An execution is defined to have a data race if two events not related by $\prec_{\mathbf{hb}}$, at least one of which is a write, touch the same memory location with a non-seqcst consistency (denoted by the condition $\neg\, \mathrm{sync}_r(ev, ev')$). Note that this definition relies on the memory model to define what a data race is, in common with the SC-DRF proof of [Batty et al. 2012]. More recent work [Batty et al. 2015] advocates for a definition of data-race-freedom which is model-agnostic, and therefore simpler for programmers to reason about. We use the "model-internal" approach here, because it is the approach taken by the JavaScript specification in stating their SC-DRF guarantees [ECMA International 2018a] and, as discussed below, it is useful for us to be able to directly contrast the guarantees of the two models.

This definition considers seqcst accesses that overlap but do not have equal ranges as racy. This is consistent with the axiomatic model, which effectively degrades such accesses to unord. The is-seqcst condition requires that every read must observe the most recent write in the total order $\prec_{\mathbf{tot}}$.

6.2 JavaScript is not SC-DRF

The official specification for JavaScript claims that its relaxed memory model guarantees SC-DRF [ECMA International 2018a]. However, we have identified two reasons why this is not the case.

As previously discussed in Section 2, the JavaScript model suffers from the SC-DRF violations previously identified in a draft version of the C++ model by Batty [Batty 2014; Batty et al. 2011]. Moreover, merely adapting the C++ model's strengthening alone is not sufficient to enforce SC-DRF. The JavaScript model is additionally vulnerable to a novel counter-example which cannot be expressed in the formal C++ model, since it relies on an unord (non-atomic) read observing a seqcst write. Fig. 18 shows such a counter-example; a JavaScript program that is data-race-free, but not sequentially consistent. While both atomic writes are guaranteed to occur before any non-atomic read of `x[0]`, no sequential interleaving can explain the fact that both reads are allowed to take different values. We have confirmed the validity of this execution using the EMME tool [Mattarei et al. 2018], a model checker for the (uncorrected) JavaScript memory model. To the best of our knowledge, this execution is not observable on any real hardware because of the coherence guarantees between two same-location atomic writes, which force the second thread to observe them as totally ordered. We have verfied this in rmem for x86 and ARM, based on the compilation schemes laid out in Section 7.

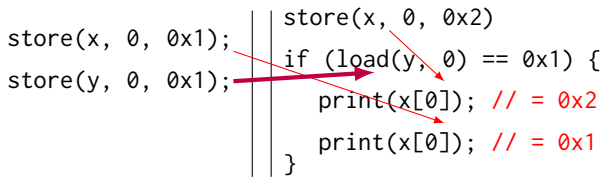

Fig. 18. A data-race-free JavaScript program that is not sequentially consistent; {load/store} abbreviates Atomics.{load, store}, the thick line represents *synchronizes-with*, which ensures that no data race occurs

6.3 Contrasting Wasm and JavaScript

Because Wasm and JavaScript are required to interoperate extensively on the Web, we must address how their memory models can be aligned. If the two conditions highlighted in Fig. 16, marked (†) and (‡), are removed from the Wasm model, this model becomes a superset of the uncorrected JavaScript one in the following sense:

Proposition 6.2. *Taking a model without* (†) *and* (‡)*, the data accesses of a Wasm program with no out-of-bounds* **trap** *errors will exhibit the same consistency behaviours as a JavaScript program carrying out equivalent accesses on a shared array buffer.*

However, such a model is clearly not SC-DRF. We have engaged with ECMA TC39, the standards body for JavaScript, about the possibility of amending the JavaScript model to include (†) and (‡) as a strict strengthening of the existing model. The correctness of the (†) condition has already been extensively investigated for C++ [Batty 2014; Batty et al. 2012, 2011], and we believe that the (‡) condition, as its dual, is also supported by current compilation schemes, given that real hardware disallows our counter-example. The standards body has provisionally agreed to accept our proposed changes in a future edition of the standard, and we continue to investigate more formal guarantees of their correctness.

Experimental Validation. We have implemented an SMT-based litmus checking tool for the Wasm memory model, with and without (†) and (‡). The tool accepts small fragments of Wasm code writen in an abstracted syntax, and computes and visualizes all valid executions. We expect that it will be useful in communicating the model to implementers and users, and can be used as an oracle for future testing of implementations.

We also implement a front-end that allows our tool to accept litmus tests written in (a subset of)[4] the syntax used by the EMME tool. This allows us to experimentally validate both Proposition 6.2 and our tool, by running the test in both tools, and checking that both tools generate the same set of visible behaviours for each litmus.

The EMME implementers provide a number of hand-written litmus tests. We observe that Proposition 6.2 holds for all 21 tests that our parser currently supports.

7 COMPILATION

In this section we discuss Wasm compilation both in its capacity as a "source" language compiled to hardware machine code by a Wasm engine (*consumers*), and as a target language for existing low-level languages like C/C++ and their compilers (*producers*). We motivate the correctness of Wasm to platform assembly schemes to the best of our ability, and describe several outstanding problems in the wider field of low-level relaxed memory research which will need to be solved in order to fully formalise a correctness proof. We show that compilation of C/C++ accesses to Wasm is correct as a direct consequence of our SC-DRF result.

[4]The EMME litmus syntax allows written bytes to be represented by integer or float literals. For now, we only support the integer syntax. Additionally, we do not support JavaScript-level constructs such as for loops.

Wasm instruction	JS operation	x86	ARMv7	AArch64
t.**load**	v[k]	MOV	ldr	LDR
t.**store**	v[k] = n	MOV	str	STR
t.**load.atomic**	Atomics.load	MOV	ldr; dmb ish	LDAR
t.**store.atomic**	Atomics.store	XCHG	dmb ish; str; dmb ish	STLR
t.**rmw.cmpxchg**	Atomics.compareExchange	[Sewell and Sevcik 2016]		

Fig. 19. Compilation of Wasm and JS memory accesses to selected platforms; atomics compilation follows C/C++11 SC atomics [Sewell and Sevcik 2016]; read-modify-write operations are generally not supported directly by platform instructions, and must be compiled to loops, which are omitted here for space.

7.1 Compiling Wasm to Hardware

The expected mapping of Wasm (and JavaScript) memory accesses is given in Fig. 19. These compilation schemes are identical to C/C++11 [Sewell and Sevcik 2016]. Accesses to globals and tables can be compiled in a similar manner.

7.1.1 Mixed-size Access. Unfortunately, given the current state of the art in mixed-size concurrency research, it is difficult to fully investigate the correctness of these compilation schemes. Almost all research into platform assembly relaxed memory models concerns only a non-mixed-size fragment. No mixed-size axiomatic model exists for *any* architecture; this is a substantial open problem. To the best of our knowledge, the *only* existing formal work on the correctness of a mixed-size compilation scheme is [Flur et al. 2017]. This work presents mixed-size operational models for ARMv8 and Power, and sketches a mixed-size generalisation of a previous proof from non-mixed-size C/C++11 to Power [Batty et al. 2012; Sarkar et al. 2012]. Mixed-size C11 is less general than our model, as it only allows non-atomics to be mixed-size. Our model allows atomics to be mixed-size, although such accesses do not provide the same guarantees as non-mixed-size atomics (for example, mixed-size atomics are not related by sync in our model). Flur et al. [2017] state that creating an abstract axiomatic model, suitable for more involved proofs, is important future work.

We can still give some limited intuition regarding the correctness of the scheme, as a guide for future proof. Considering a non-mixed-size fragment of our model (i.e. no overlapping accesses), our unord accesses share a compilation scheme not only with C/C++ non-atomics, but also with C/C++ relaxed atomics, which are expected to be stronger, and our seqcst accesses share a compilation scheme with C/C++ sequentially consistent atomics, both of which must respect a total order. We expect that for such a fragment of Wasm, the correctness of our compilation scheme should be provable following a similar strategy to existing proofs of the correctness of a (non-mixed-size) C/C++ scheme.

Going beyond this fragment, our model's mixed-size accesses have a very weak behaviour, with mis-aligned accessses effectively treated by our no-tear rule as being decomposed into independant byte accesses. This fits the architectural models proposed by [Flur et al. 2017], where mis-aligned architectural loads and stores are treated as being decomposed in the same way. The main remaining concern is aligned, but mixed-size accesses; for example, 32-bit and 64-bit accesses to the same location. The architectural models of [Flur et al. 2017] guarantee that such accesses experience a form of coherence, but there are some edge-cases that warrant further investigation. Our model deliberately chooses a behaviour here that we expect to be far weaker than the behaviour of real architectures; mixed-size seqcst atomic accesses are effectively treated like unord non-atomic accesses for the purpose of sc-last-visible. This means two overlapping mixed-size accesses are not subject to coherence guarantees under any circumstances. There is room to strengthen this guarantee, but it would need to be motivated by additional investigation into the precise guarantees of mixed-size architectural models.

7.1.2 Bounds Checks. The discussion above has focussed purely on correctly compiling in-bounds accesses. We must also deal with the relaxed behaviour of access bounds checks. As previously discussed, our model supports an implementation where bounds checks are compiled as explicit non-atomic reads. In this case, compilation of bounds checks may be treated identically to compilation of data accesses, as the bounds check will be compiled as a bare architectural load (Fig. 19) followed by a conditional branch to code which handles the **trap** result. The correctness of this compilation scheme would therefore follow from the correctness of the compilation scheme for data accesses.

To the best of our knowledge, there is no existing concurrency-aware research that is capable of facilitating the verification of the more efficient "trap handler" implementations, since they rely on the concurrent (relaxed) semantics of memory protection behaviour in both the OS and the underlying architecture. However implementers are committed to ensuring that these implementations are at least as strong as the naïve strategy.

7.1.3 Wait/Wake. The **wait**/**wake** operations are not directly compiled as platform assembly, but are implemented using OS system calls. This is in common with other languages with these features such as Java, which guarantees similar synchronization. To the best of our knowledge, the formal correctness of these mappings has not been investigated in any language, but at least on Linux, suspending and waking a thread are documented as implying several strong barriers [Howells et al. 2019] which we expect to be sufficient to support our Wasm-level synchronization. Again, existing literature does not explore the relaxed behaviour of OS calls, which would be necessary for formal proof.

7.2 Compiling C/C++ to Wasm

The expected mapping of C/C++11 accesses to Wasm memory accesses is given in Fig. 20. (Thread-local storage will be compiled to Wasm globals, but we omit that case here.) The correctness of this mapping can be justified straightforwardly as follows. First, note that this mapping effectively treats weaker C/C++ atomic accesses (e.g. release/acquire, relaxed) as sequentially consistent (`memory_order_seq_cst`). Batty [Batty 2014] shows that C/C++ programs made up of such accesses are SC-DRF, and moreover that they admit *all* sequentially consistent executions. Since all valid C/C++ programs must be data-race-free, the compiled Wasm program will also be data-race-free, assuming that atomic locations are allocated to disjoint, aligned portions of the Wasm heap. Therefore, by our SC-DRF result (Section 6), the compiled Wasm program must have only SC executions, which must therefore be valid executions of the original C/C++ program.

Of course, this sketch only justifies a correctness result between the axiomatic parts of the C/C++ and Wasm memory access semantics. Full correctness of compilation relies on many operational aspects, such giving a scheme for C/C++ memory allocation (malloc/new) to be correctly implemented in Wasm. This is orthogonal to the relaxed memory model, and therefore not approached by this work. It should be noted that, given a correct implementation of C/C++ memory allocation in Wasm, the accesses of valid programs will always be in-bounds.

C/C++11 operation	**Wasm instruction**
Load (non-atomic)	*t*.**load**
Store (non-atomic)	*t*.**store**
Load Atomic (any consistency)	*t*.**load.atomic**
Store Atomic (any consistency)	*t*.**load.atomic**
Cmpxchg (any consistency)	*t*.**rmw.cmpxchg**

Fig. 20. Compilation of C/C++11 accesses to Wasm memory access (for appropriate Wasm type *t*)

Treating weaker C/C++ atomics as strongly as sequentially consistent atomics when compiling to Wasm loses some efficiency, as it implies additional barriers when the resulting Wasm is further compiled to platform assembly, compared to directly compiling the original C/C++. This strengthening has been accepted practice when compiling C/C++ to the Web platform since at least 2015, through Emscripten and asm.js [Herman et al. 2014; Jylänki 2015]. Compiler optimisations prior to the final Wasm generation may still take advantage of weaker consistency modes, but it is true that this approach, while semantically simpler, leaves some performance on the table. The Wasm Working Group is open to the possibility of adding weaker consistency modes to the language in the future, which would improve on this. Such an extended model would require a more involved proof of correctness for C/C++ compilation.

Similarly, higher-level languages such as Java [Cenciarelli et al. 2007] and Multicore OCaml [Dolan et al. 2018] define stronger relaxed memory models, so that their basic accesses are not supported by Wasm's **unord** consistency mode. In general, features for explicitly supporting efficient compilation of higher-level languages to Wasm are at an early stage of standardisation (e.g. the Garbage Collection proposal [Rossberg 2019]), and we expect that the specification of efficient consistency modes for this use-case will occur on a similar timescale.

8 RELATED WORK

Our memory model follows the existing definitional presentations of the axiomatic relaxed memory models of Java [Cenciarelli et al. 2007; Manson et al. 2005] and C++ [Batty et al. 2011; Boehm and Adve 2008]. These existing works, and those that build atop them, are limited by the fact that the wider normative specifications that they are embedded within are not formal, meaning that a significant part of subsequent work involves defining an appropriate formal specification for the concurrent operational semantics and motivating its correctness. This was the case with the JinjaThreads project [Lochbihler 2018], which was the result of Java formalisation work spanning over fifteen years. Because the Wasm operational semantics is fully formal, all definitional work is already incorporated into the normative specification, paving the way for mature formal analyses of the memory model in future work.

Recent work criticising the state of the art in axiomatic memory models has focused on the semantics of race conditions and out-of-thin-air. It is a well-known result that current axiomatic models must choose between admitting out-of-thin-air executions, and requiring a less efficient compilation scheme [Batty et al. 2015; Ševčík and Aspinall 2008]. The models of high-level languages such as OCaml [Dolan et al. 2018] and Java [Manson et al. 2005] have the freedom to choose a less relaxed semantics, as they are not chasing bare-metal performance. Lower-level languages such as C/C++ [Batty et al. 2015] admit out-of-thin-air executions in order to compile their weakest primitives to bare loads and stores [Batty et al. 2012; Sewell and Sevcik 2016; Vafeiadis et al. 2015]. Our model must be pragmatic in this regard, allowing out-of-thin-air executions for racing non-atomics. Due to the low-level nature of Wasm, implementers expect to compile its non-atomics to bare loads and stores. At the very least, we do not make racy non-atomics an undefined behaviour in the style of C/C++, and a program without racing non-atomics will not admit thin-air executions as a consequence of our SC-DRF result.

We have begun to see a new generation of models with the explicit aim of disallowing out-of-thin-air while preserving efficient compilation schemes [Kang et al. 2017; Pichon-Pharabod and Sewell 2016; Podkopaev et al. 2018]. As these models become more mature, it may be possible to use aspects of them to disallow our out-of-thin-air executions.

9 FUTURE WORK AND CONCLUSION

Our formal semantics for Wasm extended with shared memory concurrency anticipates future extensions such as shared globals, tables, and references. To achieve maximum generality and avoid preempting future design choices, this semantics supports all consistency modes for all stateful objects. We expect that concrete proposals to incorporate these features into Wasm in the future will make more specific choices, i.e., only support sequentially consistent access to tables.

We leave space within the operational semantics for additional consistency modes to be introduced. We expect that this will be necessary, for example, to efficiently support the compilation of programs that make use of so-called "low-level atomics" [Boehm and Adve 2008]. Moreover, future features such as memory protection may have to be integrated into the model.

The research trajectory of each language's relaxed memory model follows a predictable pattern. It is often a significant effort to even represent the memory model formally [Boehm and Adve 2008; Manson et al. 2005], let alone integrate it with the language's existing semantics. Even then, it will often be many more years of collaborative research effort before mature tooling and mechanised proofs over the model can be developed [Batty et al. 2011; Lochbihler 2018].

We believe that our presentation of the WebAssembly memory model lays a firm foundation for this further work. We present a fully mathematised specification of not only the axiomatic model, but the operational semantics, a significant improvement on the foundational presentations of the Java and C++ models [Boehm and Adve 2008; Manson et al. 2005]. Previous work has already mechanised the core of WebAssembly's sequential semantics [Watt 2018], and we expect that our work here will be the basis of future mechanisation of WebAssembly concurrency.

ACKNOWLEDGMENTS

We thank Shu-yu Guo, Lars T Hansen, and Peter Sewell for their valuable feedback. We thank the members of the WebAssembly Community Group, the WebAssembly Working Group, and ECMA TC39 for useful discussions. This work was partly supported by the EPSRC Programme Grant *REMS: Rigorous Engineering for Mainstream Systems* (EP/K008528/1). The first author was supported by an EPSRC DTP award (EP/N509620/1), and a Google PhD Fellowship in Programming Technology and Software Engineering.

REFERENCES

Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering — a New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*. ACM, New York, NY, USA, 2–14. https://doi.org/10.1145/325164.325100

Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 577–591. https://doi.org/10.1145/2694344.2694391

Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. 2009. The Semantics of Power and ARM Multiprocessor Machine Code. In *Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming*. ACM, New York, NY, USA. https://doi.org/10.1145/1481839.1481842

Mark Batty. 2014. *The C11 and C++11 Concurrency Model*. Ph.D. Dissertation. University of Cambridge.

Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library Abstraction for C/C++ Concurrency. *SIGPLAN Not.* 48, 1 (Jan. 2013), 235–248. https://doi.org/10.1145/2480359.2429099

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 283–307.

Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Philadelphia)*. 509–520. https://doi.org/10.1145/2103656.2103717

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*.

ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

Mark Batty and Peter Sewell. 2014. The Thin-air Problem. https://www.cl.cam.ac.uk/~pes20/cpp/notes42.html.

Hans-J. Boehm. 2005. Threads Cannot Be Implemented As a Library. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 261–268. https://doi.org/10.1145/1065010.1065042

Hans-J. Boehm. 2011. How to Miscompile Programs with "Benign" Data Races. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism (HotPar'11)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=2001252.2001255

Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 68–78. https://doi.org/10.1145/1375581.1375591

Hans-J. Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding Out-of-thin-air Results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC '14)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2618128.2618134

Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. 2007. The Java Memory Model: Operationally, Denotationally, Axiomatically. In *Proceedings of the 16th European Symposium on Programming (ESOP'07)*. Springer-Verlag, Berlin, Heidelberg, 331–346. http://dl.acm.org/citation.cfm?id=1762174.1762206

Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding Data Races in Space and Time. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 242–255. https://doi.org/10.1145/3192366.3192421

ECMA International. 2018a. ECMAScript 2018 Language Specification - Data Race Freedom. https://www.ecma-international.org/ecma-262/9.0/index.html#sec-data-race-freedom.

ECMA International. 2018b. ECMAScript 2018 Language Specification - Memory Model. https://www.ecma-international.org/ecma-262/9.0/index.html#sec-memory-model.

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg, FL, USA*. 608–621.

Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. Mixed-size Concurrency: ARM, POWER, C/C++11, and SC. *SIGPLAN Not.* 52, 1 (Jan. 2017), 429–442. https://doi.org/10.1145/3093333.3009839

Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. 1992. Programming for Different Memory Consistency Models. *J. Parallel and Distrib. Comput.* 15 (1992), 399–407.

Kathryn E. Gray, Gabriel Kerneis, Dominic Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. 2015. An Integrated Concurrency and core-ISA Architectural Envelope Definition, and Test Oracle, for IBM POWER Multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 635–646. https://doi.org/10.1145/2830772.2830775

Andreas Haas, Andreas Rossberg, Derek Schuff, Ben Titzer, Dan Gohman, Luke Wagner, Alon Zakai, JF Bastien, and Michael Holman. 2017. Bringing the Web up to Speed with WebAssembly. In *Principles of Programming Languages (POPL)*.

Lars T Hansen. 2017. Resizing details / underspecification. https://github.com/WebAssembly/threads/issues/26.

David Herman, Luke Wagner, and Alon Zakai. 2014. `asm.js`. http://asmjs.org/spec/latest.

Lisa Higham, LillAnne Jackson, and Jalal Kawash. 2006. Programmer-centric Conditions for Itanium Memory Consistency. In *Proceedings of the 8th International Conference on Distributed Computing and Networking (ICDCN'06)*. Springer-Verlag, 58–69. https://doi.org/10.1007/11947950_7

David Howells, Paul E. McKenney, Will Deacon, and Peter Zijlstra. 2019. Linux Kernel Memory Barriers. https://www.kernel.org/doc/Documentation/memory-barriers.txt.

Jukka Jylänki. 2015. Emscripten gains experimental pthreads support! https://groups.google.com/forum/#!topic/emscripten-discuss/gQQRjajQ6iY.

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 175–189. https://doi.org/10.1145/3009837.3009850

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. *SIGPLAN Not.* 52, 6 (June 2017), 618–632. https://doi.org/10.1145/3140587.3062352

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

Andreas Lochbihler. 2018. Mechanising a Type-Safe Model of Multithreaded Java with a Verified Compiler. *Journal of Automated Reasoning* 61, 1 (01 Jun 2018), 243–332. https://doi.org/10.1007/s10817-018-9452-x

Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. 2012. An Axiomatic Memory Model for POWER Multiprocessors. In *Proceedings of the 24th International Conference on Computer Aided Verification*. 495–512. https://doi.org/10.1007/978-3-642-31424-7_36

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 378–391. https://doi.org/10.1145/1040305.1040336

Cristian Mattarei, Clark Barrett, Shu-yu Guo, Bradley Nelson, and Ben Smith. 2018. EMME: A Formal Tool for ECMAScript Memory Model Evaluation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 55–71.

Paul E. McKenney, Alan Jeffrey, and Ali Sezgin. 2005. N4375: Out-of-Thin-Air Execution is Vacuous. *C++ Standards Committee Papers* (2005).

Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. 2016. An operational semantics for C/C++11 concurrency. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA, 18. https://doi.org/10.1145/2983990.2983997

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *Proceedings of Theorem Proving in Higher Order Logics, LNCS 5674*. 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

Jean Pichon-Pharabod and Peter Sewell. 2016. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg, FL, USA, January 20 - 22, 2016*. 622–633. https://doi.org/10.1145/2837614.2837616

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2018. *Bridging the Gap between Programming Languages and Hardware Weak Memory Models*. Technical Report. https://doi.org/10.1145/3290382 arXiv:arXiv:1807.07892

Andreas Rossberg. 2018. Reference Types Proposal for WebAssembly. https://github.com/WebAssembly/reference-types.

Andreas Rossberg. 2019. GC Extension. https://github.com/WebAssembly/gc/blob/master/proposals/gc/Overview.md.

Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 311–322. https://doi.org/10.1145/2254064.2254102

Peter Sewell and Jaroslav Sevcik. 2016. C/C++11 mappings to processors. https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html.

Ben Smith. 2019. Threading proposal for WebAssembly. https://github.com/WebAssembly/threads.

Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 209–220. https://doi.org/10.1145/2676726.2676995

Jaroslav Ševčík. 2011. Safe Optimisations for Shared-memory Concurrent Programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 306–316. https://doi.org/10.1145/1993498.1993534

Jaroslav Ševčík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP '08)*. Springer-Verlag, Berlin, Heidelberg, 27–51. https://doi.org/10.1007/978-3-540-70592-5_3

Conrad Watt. 2018. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*. ACM, New York, NY, USA, 53–65. https://doi.org/10.1145/3167082

WebAssembly Working Group. 2019. WebAssembly Specifications. https://webassembly.github.io/spec/.