UNIVERSITY OF
CAMBRIDGE
Department of Computer
Science and Technology

Extending a WebAssembly formalisation

Maja Trela

Trinity College

May 2022

Submitted in partial fulfillment of the requirements for the
Computer Science Tripos, Part III

Total page count: 52

Main chapters (excluding front-matter, references and appendix): 38 pages (pp 9–46)

Main chapters word count: 10975

Methodology used to generate that word count:

```
$ texcount -template="{SUM}\n" report.tex
10975
```

Declaration

I, Maja Trela of Trinity College, being a candidate for Computer Science Tripos, Part III, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed: Maja Trela **Date**: 21.05.2022

Acknowledgements

This project would not have been possible without its originator and co-supervisor Conrad Watt, who had a clear idea of the project's goals and scope, as well as the techniques to make it feasible. He is the main contributor to WasmCert-Isabelle and the author of the interpreters involved. We collaborated closely throughout the course of the project, with him occasionally extending or modifying my proofs as a consequence of updating the repository to reflect more recent revisions of the WebAssembly specification. Moreover, Peter Lammich has been of great help, having supplied us with examples of use of his separation logic library and teaching me how to use `sep_auto` effectively.

I'd also like to thank my friends for keeping my sanity by making me leave my room every now and then while simultaneously pressuring me to write down the project report in time.

Abstract

WebAssembly is a growing standard for executable content on the Web. Due to its relative importance, small size, and precise specification, it has already been subject to formal verification in order to ensure its safety [1]. This project expands the existing efforts in two distinct areas. The first area is the *instantiation* mechanism, responsible for loading a WebAssembly module before execution; a missing proof of safety of instantiation is completed. The second area, of a larger focus, is a verifiable interpreter with a good enough performance for practical use; this project proves its soundness by *refinement*, i.e. showing its equivalence to an earlier verified interpreter, utilising monadic state and separation logic for that purpose. As a result, the improved verified interpreter was adopted in practice by the Bytecode Alliance for use in its testing infrastructure.

Contents

Listings

Chapter 1

Introduction

Writing code is difficult for humans, and writing correct code especially so. Inevitability of bugs is thus commonly accepted; one can always release a patch fixing the newly-discovered issue after all. That approach is perfectly reasonable for most use cases, where malfunctions cause relatively little harm. There are however cases when the trade-offs shift; whether it's because the application is entrusted with a major responsibility, or in widespread use on billions of devices, an unknown issue surfacing at an inconvenient time, or, possibly worse, opening up a security vulnerability, may be catastrophic in its consequences. In those cases one can increase testing to reduce the probability of a bug slipping through to release, and write code more defensively to reduce the probability of a bug arising at all; can one however ever be *sure*?

Formal verification is a technique bringing us one step closer to certainty. The approach switches; instead of trying to show counterexamples to a program's correctness, maybe we can *prove* it correct? A proof of one's desired properties of the system, expressed in formal mathematics, could show that bugs violating those properties cannot possibly occur[1]. Moreover, proof assistant software such as Isabelle removes the need to trust that the humans writing the proof didn't make a mistake themselves.

One effective target for formal verification is *WebAssembly*, a bytecode format intended for executable content on the Web. It includes a precise specification of executing said bytecode which browsers and other platforms supporting it have to follow. One of the promises of WebAssembly is enabling both fast and safe execution; the specification prescribes a validation step, and claims that certain checks performed at runtime[2] can be safely skipped for validated bytecode, improving execution speed. Formally verifying that claim allows us to conclude that an implementation of WebAssembly is safe provided it follows the specification. In addition, one can write an interpreter of WebAssembly and formally verify that it conforms to the specification. Accordingly, both of those goals have

[1] Assuming perfect hardware and consistency of mathematics.

[2] Example: checking whether attempted write to an array is within the bounds of said array.

been carried out already [1]; successfully validated WebAssembly bytecode is known to be safe to run in the manner prescribed by the specification, and a verified interpreter for WebAssembly exists.

1.1 Project overview

This project is basing on WasmCert-Isabelle, a repository of formal proofs relating to WebAssembly which in particular contains the aforementioned proof of safety and the verified interpreter; naturally, the proof assistant used by this project is Isabelle.

The first, smaller part of the project is to complete the missing safety proof for a phase of WebAssembly execution called *instantiation*. Instantiation is the process of converting a module into its runtime representation, allocating the state in memory as to make it ready for execution. The goal is to show that the result of instantiation fulfills the properties sufficient for it to be safe to execute.

The second, larger part is to verify a WebAssembly interpreter fast enough for practical use. The interpreter in question was a previously unverified existing side project basing on WasmCert-Isabelle, with a similar structure to the existing verified interpreter. The goal is to verify the fast interpreter by proving that the two interpreters always return the same result.

Both goals of the project were achieved, with extra work being focused on verifying the soundness of further optimisations to the fast interpreter. Moreover, a bug was found and fixed in the fast interpreter in the process of proving its soundness.

Notably, as a result of this project, the improved verified interpreter was incorporated into Bytecode Alliance's WebAssembly testing infrastructure. The Bytecode Alliance is a partnership of corporations with an interest in advancing WebAssembly's ecosystem across the Web; its major members include companies such as Fastly, Mozilla, and Microsoft. We are thus boasting a rare claim of the result of a formal verification project being adopted by major industry players and used in practice to improve software reliability.

It is rare for formal verification projects to verify the entire process from source code to executable file. While this means the resulting program is not certain to be bug-free, formal verification nevertheless reduces the scope in which bugs can arise. In this manner we assume the correctness of the OCaml compiler, Isabelle's code extraction tools, and custom translations from Isabelle into OCaml.

Results of this work are planned to be included in a paper submission to the POPL 2023 conference. The submission is in preparation at the time of writing.

Chapter 2

Background

2.1 WebAssembly

WebAssembly [2] (sometimes abbreviated as Wasm) is an executable low-level bytecode standard, mainly intended for use on the Web (but not limited to). Prior to its introduction, JavaScript was de facto the only standard for client-side executable content (and at the time of writing it is still the dominant language for that purpose) and thus WebAssembly was designed from scratch to provide an alternative where JavaScript is flawed, stating safety, fast execution, portability, and low binary size as its design goals.

2.1.1 Overview

Architecture

WebAssembly, just as any other bytecode standard, specifies its execution model, also called the *virtual machine*. The actual machine code the bytecode is compiled to might diverge from the virtual machine for performance reasons, but the result of executing the bytecode has to be observably the same as if run on the virtual machine. In WebAssembly's case the virtual machine is a stack machine; instructions are defined in terms of manipulating the top of a stack of operands, rather than a set of registers. A list of instructions is called an *expression*; executing the instructions in order results in the expression being *evaluated*.

WebAssembly also has a concept of the type of an expression, defined as the list of types of operands popped from the stack and pushed onto it when the expression is evaluated. As a corollary, in well-typed programs the shape of the stack at any point in the program can always be determined statically; the actual WebAssembly engine in the browser thus can optimise away the stack using that knowledge.

Control flow, rather than provided by a `goto` equivalent as might be expected in a low-level language, is defined in a structured way, the specifics of which might however be somewhat unintuitive at a first glance. The `block` instruction defines an instruction block; the `br` ("branch") instruction allows exiting a block early (similarly to `break` in many programming languages). Branching on condition is provided by the `if` instruction, which specifies two blocks of instructions – one for "true" and one for "false". Moreover, the `loop` instruction makes constructing loops possible; executing `br` while within the `loop` block restarts the loop (similarly to `continue` in many programming languages), and letting execution reach the end of the `loop` block exits the loop.

Although statically checked, WebAssembly allows some instruction to *trap*, i.e. abort execution and hand control back to the host environment. Not all types of faults can be practically eliminated at compile time, e.g. there is no way of knowing in general whether a given division operation will end up performing division by zero or not. In those cases the trap mechanism provides a way to "fail safely".

Organisation

At its top level, WebAssembly code is organised into *modules*, as shown in Figure 2.1. The core content of a module are *functions*, *memories*, *tables*, and *globals*, explained in more detail below.

Figure 2.1: Formal definition of a module as written in WebAssembly specification.

$$
\begin{aligned}
\mathit{module} ::= \{\ & \textbf{types}\ \mathit{vec}(\mathit{functype}),\\
& \textbf{funcs}\ \mathit{vec}(\mathit{func}),\\
& \textbf{tables}\ \mathit{vec}(\mathit{table}),\\
& \textbf{mems}\ \mathit{vec}(\mathit{mem}),\\
& \textbf{globals}\ \mathit{vec}(\mathit{global}),\\
& \textbf{elems}\ \mathit{vec}(\mathit{elem}),\\
& \textbf{datas}\ \mathit{vec}(\mathit{data}),\\
& \textbf{start}\ \mathit{start}^{?},\\
& \textbf{imports}\ \mathit{vec}(\mathit{import}),\\
& \textbf{exports}\ \mathit{vec}(\mathit{export})\ \}
\end{aligned}
$$

Functions are the basic unit WebAssembly instructions are organised into. A function contains a typed expression pushing values onto the stack, and can both take multiple arguments (by popping them from the top of the stack and saving them as local variables) and return multiple values (by leaving them at the top of stack at the end). Moreover, it may contain a number of local variables aside from the arguments; the expression may contain instructions for reading from and writing to local variables.

Memories are raw arrays of bits with no additional type information. WebAssembly code can freely read from and write to a given memory, reinterpreting the raw bits as a numeric type. There might be multiple memories in a program.

Tables provide a mechanism for runtime polymorphism of functions. A table is a read-only array of function indices, initialised at runtime. An expression, instead of calling a function directly, might instead call a function in the table as specified by the index given at runtime. The caller needs to specify the function type of the function called; the function call only proceeds if the type of the function at the specified index in the table agrees with the type declared by the caller, otherwise a trap occurs.

Globals are variables which can be accessed anywhere in the program. An expression may contain instructions for reading from a global variable, as well as writing to it if the global variable is specified as mutable.

Aside from the above key concepts, a module specifies initializer expressions for each global variable, evaluated when the module is loaded. In a similar manner, a module may contain *element segments* and *data segments*, which are initializer expressions for fragments of tables and memories respectively. Additionally, a module may specify the execution entry point if it's intended to be executed directly rather than used as a library. Lastly, a module may contain *imports*, which are definitions to be imported from other modules and which can be used throughout the module, as well as *exports*, which are definitions visible to other modules for import. When loaded, the module becomes an *instance*.

2.1.2 Semantics

WebAssembly is a rare case of a language with precisely defined semantics in a formal manner. It is described in terms of operational semantics with a small step reduction relation, the overview of which is written below.

First, a few more concepts are introduced. A *store* is a collection of runtime representations of functions, memories, tables, and globals. In other words, the store holds global state. A *frame* contains a list of values of local variables as well as the reference to the current module instance.[1]

Moreover, the instruction set is now augmented with *administrative instructions*. Administrative instructions only exist for the ease of defining what happens during execution formally by unifying similar constructs; modules may contain only basic (i.e. non-administrative) instructions. One such particular instruction is `invoke`, with the meaning of calling a function without further indirection; this way the description of handling both

[1] One can think of a frame as analogous to a call stack frame of a method in an object-oriented language; the instance is the object the method is executed from.

direct and indirect (i.e. from a table) function calls doesn't repeat needlessly. Similarly, the `label` instruction is used to unify the semantics of `block`, `loop`, and `if`. A `label` contains both the instruction block as well as its *continuation*, expressed as the list of instructions to insert after the block when `br` is used; `block` reduces to a `label` with no instructions to insert, while `loop` contains a copy of itself.

Execution is described in terms of single steps (i.e. the *reduction relation*) performed on a *runtime configuration*, which consists of a store, a frame, and a list of (possibly administrative) instructions. Interestingly enough, the operand stack is implicitly contained within the instruction list; the `const` (pushing a constant onto the stack) instructions at the start of the list are treated as the stack itself. As an example, numeric binary operators are defined in a generic way as

$$\frac{}{(\texttt{const}\, c_1)\,(\texttt{const}\, c_2)\, binop \hookrightarrow (\texttt{const}\, c)} \text{ where } c \in binop(c_1, c_2) \qquad \text{(binop-reduce)}$$

i.e. a binary operator (such as `add` or `sub`) pops two values from the stack, and pushes a new value as a result. As another example, the `call` (direct function call) instruction contains the index of the function to call as defined in the module; therefore the conversion from `call` to `invoke` needs to look up the function address stored in the module instance.

$$\frac{}{F; (\texttt{call}\, x) \hookrightarrow F; (\texttt{invoke}\, a)} \text{ where } F.\texttt{inst.funcs}[x] = a \qquad \text{(call-reduce)}$$

The reduction for each instruction is defined in a similar manner, possibly using and/or updating the store and/or the frame, leading to the complete reduction relation on runtime configurations, denoted by

$$S; F; \texttt{instr}^* \hookrightarrow S'; F'; \texttt{instr}'^*.$$

2.1.3 Typing

As mentioned earlier, WebAssembly boasts a static type system, which allows it to skip some of runtime checks required otherwise in more dynamically typed languages like JavaScript, without compromising safety.

The basic types of WebAssembly are the *value types*. A value type can be 32 or 64 bit, integer or float. Subsequently, *function types* consist of the list of types of values popped from the stack and a list of those pushed onto it. In a well-typed WebAssembly program, each subexpression has a function type.

To define the typing relation rigorously, we need a *context* C, containing type information about functions and global state in the instance the expression is being executed in. Using

the earlier example, the typing rule for a binary operator on value type t is

$$\frac{}{C \vdash t.binop : [t\,t] \to [t]} \qquad \text{(binop-typing)}$$

i.e. it pops two values of type t and pushes a single value of type t in return. Meanwhile, the type of `call` is determined with the use of the context:

$$\frac{C.\texttt{funcs}[x] = [t_1^*] \to [t_2^*]}{C \vdash \texttt{call}\,x : [t_1^*] \to [t_2^*]} \qquad \text{(call-typing)}$$

Typing rules for expressions proceed in this manner.

Similarly, there are typing rules for defining that a store type-checks correctly, as well as a few others. Without diving into tiresome details, the end result is the typing relation

$$\vdash S; F; \texttt{instr}^* : [t^*]$$

which, broadly speaking, is intended to guarantee that repeated application of the $\hookrightarrow$ reduction rule on a runtime configuration $S; F; \texttt{instr}^*$ either is always possible or ends in an allowed end state; moreover, if that end state does not trap, the remaining stack forms a list of values of types t^*. This can be formalised in the classic form of Preservation and Progress theorems for a type system:

Theorem (Preservation)
If

$$\vdash S; F; \mathit{instr}^* : [t^*]$$

and

$$S; F; \mathit{instr}^* \hookrightarrow S'; F'; \mathit{instr}'^*$$

then

$$\vdash S'; F' \mathit{instr}'^* : [t^*].$$

Theorem (Progress) *If* $\vdash S; F; \mathit{instr}^* : [t^*]$ *then either*

- instr^* *is a single* `trap` *instruction*
- instr^* *contains only values (represented as* `const` *instructions)*
- *there exists a runtime configuration* $S'; F'; \mathit{instr}'^*$ *such that*

$$S; F; \mathit{instr}^* \hookrightarrow S'; F'; \mathit{instr}'^*.$$

In other words, those theorems claim that WebAssembly programs which pass typing checks don't do unpredictable things.

2.2 Isabelle

The goal of formal verification projects tends to be a proof of program's certain properties. However, a proof written and verified by humans only is not considered to be trustworthy enough – humans are notoriously fallible, and the theorems to be proven tend to be of tedious nature, further increasing the likelihood of glossing over important details and making it difficult for others to audit the proof. Instead, the goal is the have a proof that can be verified automatically by a computer.

One such toolset for formally verified proofs is Isabelle [3], a generic interactive proof assistant initially developed at the University of Cambridge and the Technical University of Munich. It enables writing and composing machine-verified proofs of formally-written statements. Moreover, Isabelle does not commit to a particular formal logic system; however, Isabelle/HOL (Higher-Order Logic) is most commonly used and thus assumed here.

As an example, Listing 2.1 shows an Isabelle proof of $(a+b)(a-b) = a^2 - b^2$ for $a, b \in \mathbb{N}$. One way to prove it is to apply distributivity laws; hence we can write `apply (simp add:nat_distrib)` to tell Isabelle to simplify the statement to be proved (also known as the *goal*), using distributivity of multiplication. Once we do so, pointing the cursor at line 2 within the GUI packaged with Isabelle helpfully reveals that simplification succeeded and that the current goal is $a \cdot a - b \cdot b = a^2 - b^2$. Adding that $x^2 = x \cdot x$ finishes the proof.

Listing 2.1: Isabelle example.

```
1 lemma "(a+b)*(a-b) = a² - b² " for a b::nat
2   apply (simp add:nat_distrib)
3   apply (simp add:power2_eq_square)
4   done
```

In general, proofs in Isabelle follow the structure of statements followed by their proofs, composed of applications of (possibly automated) *proof methods*. The automation facilities of Isabelle allow the human writing the proof to not write down every single step of a rigorous proof; ideally, only the interesting or non-trivial steps are written down explicitly, with automated proof methods filling the in-between. In particular, Sledgehammer [4] is an effective and versatile tool for finding proofs automatically with little user involvement. Other tools contained within Isabelle worth noting are Isabelle/Isar, a language for writing structured, human-like proofs (rather than a string of `apply` steps), and Isabelle/Eisbach, a scripting language for custom automated proof methods.

Isabelle has been used for a number of formal verification projects before. Likely the most famous such project involving Isabelle is machine verification of correctness of the seL4 microkernel [5], comprised of 8700 lines of C code. Another example is the Java-like programming language Jinja [6], formalised and proven to be type-safe in Isabelle,

including an Isabelle-verified interpreter as well. It goes without saying that WasmCert-Isabelle, the repository this project is building on, also is one of those projects.

As a particular interest to formal verification, a fragment of Isabelle/HOL can be treated as an ML-like functional language. Expressions written in that fragment can be exported as ML, OCaml, Haskell, or Scala code and executed there, a process called *code extraction*. Code extraction is of particular relevance to producing a verified interpreter since the source code of the interpreter can be native to Isabelle/HOL, and thus easy to reason about using Isabelle.

This report contains multiple listings of Isabelle code illustrating the work accomplished. To keep the report concise, Isabelle's syntax is not explained in detail, however the following points should be noted:

- Generally speaking, quotes signify the boundary between the language of logic and the outer proof language.
- TeX maths symbols are often used as generally accepted in mathematical notation. Moreover, one can easily define syntactic sugar for a particular concept in order to use mathematical symbols in place of the name of the function or predicate (e.g. note Listing 2.2 for $\rightsquigarrow$ and $\vdash$ as syntactic sugar for `reduce` and `b_e_typing`).
- Notation is ML-like: function application is written as `f x y` (rather than $f(x, y)$), `a ⇒ b ⇒ c` signifies a function taking arguments of types `a` and `b` and returning `c`, parametric types are written as `'a t` (e.g. `int list` for a list of integers). Higher-order functions (that is, taking a function as an argument) are common.
- Lists: `x#xs` denotes prepending the element `x` to a list `xs`; `xs@ys` denotes appending two lists `xs` and `ys`; `xs!i` denotes the `i`-th element of a list `xs`.
- `r.f x` denotes the field `f` of a record `x` of type `r`, and can be written as `f x` in absence of collisions with other names.

2.3 WasmCert-Isabelle

Similarly to the projects mentioned earlier, WebAssembly also has been subject to formal verification in Isabelle; [1] proved WebAssembly's type system to be sound, as well as included a verified interpreter. Furthermore, the formalisation has been updated and expanded to follow new developments in WebAssembly as well as reduce the unverified bits [7]. The result of that work is the WasmCert-Isabelle repository [8], which includes WebAssembly's specification ported to Isabelle, as well as the proof of soundness of the type system, the interpreter, and the interpreter's proof of soundness.

Listing 2.2: Reduction and typing rules in WasmCert-Isabelle

```
inductive reduce_simple :: "[e list, e list] ⇒ bool" ("(⟪_⟫) ↝ (⟪_⟫)" 60) where
[...]
| binop_Some:"⟦app_binop op v1 v2 = (Some v)⟧
    ⟹ ([$C_n v1, $C_n v2, $(Binop t op)]) ↝ ([$C_n v])"
[...]

inductive reduce :: "[s, f, e list, s, f, e list] ⇒ bool" ("(⟪_;_;_⟫) ↝ (⟪
    _;_;_⟫)" 60) where
[...]
| call:"(⟪s;f;[$(Call j)]⟫)
    ↝ (⟪s;f;[Invoke ((inst.funcs (f_inst f))!j)]⟫)"
[...]

inductive b_e_typing :: "[t_context, b_e list, tf] ⇒ bool" ("_ ⊢ _ : _"
    60) where
[...]
| binop:"binop_t_num_agree op t
    ⟹ C ⊢ [Binop t op] : ([T_num t,T_num t] _> [T_num t])"
[...]
| call:"⟦i < length(func_t C); (func_t C)!i = tf⟧
    ⟹ C ⊢ [Call i] : tf

definition reduce_trans where
  "reduce_trans ≡ rtranclp (λ(s,f,es) (s',f',es'). (⟪s;f;es⟫) ↝ (⟪s';f';es'⟫))"
```

As an example, definitions listed in Listing 2.2 reflect the (binop-reduce), (call-reduce), (binop-typing), and (call-typing) rules stated earlier when describing WebAssembly's semantics.

The `reduce_trans` predicate in Listing 2.2 is worth noting as it appears a few times in this document – it expresses the concept of evaluating an expression for multiple steps, being defined as the transitive closure of the reduction relation.

Similarly, the Preservation and Progress theorems are stated in WasmCert-Isabelle as shown in Listing 2.3.

Listing 2.3: Preservation and progress theorems in WasmCert-Isabelle

```
theorem preservation:
  assumes "⊢ s;f;es : ts"
          "(⟪s;f;es⟫) ↝ (⟪s';f';es'⟫)"
  shows "(⊢ s';f';es' : ts) ∧ store_extension s s'"

theorem progress:
  assumes "⊢ s;f;es : ts"
  shows "const_list es ∨ es = [Trap] ∨ (∃a s' f' es'. (⟪s;f;es⟫) ↝ (⟪
      s';f';es'⟫))"
```

This project uses WasmCert-Isabelle as its base, expanding on and contributing back to the repository, in a manner similar to software engineering. New proofs take advantage not only of the definitions of WebAssembly's syntax, semantics, typing, and interpreter, but also invoke lemmas proven previously in existing work. The result is a repository of formal proofs of WebAssembly's properties extended by this project's achievements.

Chapter 3

Instantiation

Similarly to other programming languages, WebAssembly provides a mechanism for code organisation in the form of *modules*, as shown in Figure 2.1 (Section 2.1.1). Naturally, a module needs to be converted to its runtime representation before its contents can be executed; the process of doing so is called *instantiation* in WebAssembly terminology, and results in a *module instance* (called just an "instance" from now on). In particular, instantiation is the stage which ensures that the module is well-typed and thus safe to run. Moreover, instantiation resolves module imports and exports, similar to linking in C; a module instance can therefore call external libraries, as well as import system-specific constants.

The WebAssembly standard provides a precise definition of WebAssembly semantics, including instantiation. Moreover, the WebAssembly specification defined typing relations, and claims that well-typed runtime configurations are safe to execute; a claim formally verified and contained in WasmCert-Isabelle (see [1], also Listing 2.3). However, the repository did not include the proof that performing instantiation produces a well-typed runtime configuration. That hole meant that while well-typed runtime configurations were known to be safe, there was no full guarantee that instantiating and executing a module is safe; it could be the case that the resulting instance is not well-typed, making the soundness theorems inapplicable.

One of the goals of this project was to formally verify that the result of instantiation is well-typed, closing the hole mentioned above. This chapter explains what was achieved as well as the technical challenges encountered in achieving that goal.

3.1 Specification

Instantiation is possibly the largest single definition in the WebAssembly specification. At a high level, instantiation takes a store, a module, and a list of values to be supplied

as imports to the module, and as a result updates the store and produces an instance corresponding to the module. More specifically, instantiation:

- Verifies that the module is well-formed and compatible with the provided imports.
- Allocates the module, i.e. adds the functions, memories, tables, and globals from the module to the store, supplying the imports where required.
- Initialises the tables and memories as specified in element and data segments.

The formal specification, as expressed in Isabelle, is explained in slightly more detail below. It is not essential to understand the definition in full; it is listed to show what the project has achieved in concrete terms.

3.1.1 `instantiate`

Listing 3.1: Definition of instantiation in WasmCert-Isabelle

```
1  abbreviation "reduces_to s f bes v ≡ reduce_trans (s,f,$*bes) (s,f,[$v])"
2  abbreviation "elem_in_bounds s inst off e ≡
3    nat_of_int off + length (e_init e) ≤ tab_size ((tabs s)!((inst.tabs
       inst)!(e_tab e)))"
4  abbreviation "data_in_bounds s inst off d ≡
5    nat_of_int off + length (d_init d) ≤ mem_length ((mems s)!((inst.mems
       inst)!(d_data d)))"
6  abbreviation "elem_to_init_tab inst off e ≡
7    Init_tab (nat_of_int off) (map (λi. (inst.funcs inst)!i) (e_init e))"
8  abbreviation "data_to_init_mem inst off d ≡ Init_mem (nat_of_int off)
     (d_init d)"
9
10 inductive instantiate :: "s ⇒ m ⇒ v_ext list ⇒
11       ((s × f × (e list)) × (module_export list)) ⇒ bool" where
12   "⟦module_typing m t_imps t_exps;
13     list_all2 (external_typing s) v_imps t_imps;
14     alloc_module s m v_imps g_inits (s', inst, v_exps);
15     f = ⦇ f_locs = [], f_inst = inst ⦈;
16     list_all2 (λg v. reduces_to s' f (g_init g) (C v)) (m_globs m) g_inits;
17     list_all2 (λe c. reduces_to s' f (e_off e) (C_n (ConstInt32 c)))
         (m_elem m) e_offs;
18     list_all2 (λd c. reduces_to s' f (d_off d) (C_n (ConstInt32 c)))
         (m_data m) d_offs;
19     list_all2 (elem_in_bounds s' inst) e_offs (m_elem m);
20     list_all2 (data_in_bounds s' inst) d_offs (m_data m);
21     (case (m_start m) of None ⇒ [] | Some i_s ⇒ [Invoke ((inst.funcs
         inst)!i_s)]) = start;
22     map2 (elem_to_init_tab inst) e_offs (m_elem m) = e_init_tabs;
23     map2 (data_to_init_mem inst) d_offs (m_data m) = e_init_mems
```

```
24      ⟧ ⟹ instantiate s m v_imps ((s', f, e_init_tabs@e_init_mems@start),
          v_exps)"
```

Listing 3.1 contains the top-level definition of WebAssembly instantiation, and is broadly explained below.

Instantiation, expressed using the `instantiate` predicate, takes as input the current store, a module, and a list of imports, and returns the runtime state (i.e. the store, frame, instructions tuple) as well as module exports. The predicate

```
instantiate s m v_imps ((s', f, init_es), v_exps)
```

asserts that the result of instantiating using the store `s`, the module `m`, and the imports `v_imps` is the runtime configuration triple `(s', f, init_es)` and the exports `v_exps`. Subsequently, the lines from 12 to 23 list the conditions for `instantiate` to hold.

First, the module needs to be validated. The `module_typing` predicate (line 12) expresses that the module `m` is well-typed, including the imports `v_imps` and their types `t_imps`. The main part of loading the module follows in line 14, using the `alloc_module` predicate, which is explained in more detail later.

Lines 16 to 18 are responsible for evaluating the initializer expressions for globals, element segments, and data segments. Subsequently, lines 19 to 20 verify that the resulting element segments and data segments fit within their respective table or memory.

As a side note, a careful reader would notice that the definition as written in Listing 3.1 is circular – the initial values for globals `g_inits` needed by module allocation are obtained using the frame `f`, which in turn contains the instance returned from module allocation. This is not a mistake in transcribing the definition into Isabelle; the official specification actually is written this way. Functions in Isabelle can't be readily defined in a circular manner without a larger departure from the specification; hence `instantiate` is implemented as a predicate, not a function. Nevertheless an executable version of instantiation can be proven equivalent to the circular one, as was accomplished in prior work on the verified interpreter.

Finally, lines 21 to 23 specify the list of instructions to be executed after instantiation; those comprise the initialisation of tables and memories in accordance to element and data segments, as well as the program entry point if specified by the module.

3.1.2 `alloc_module`

Listing 3.2: Definition of module allocation

```
1  inductive alloc_module :: "s ⇒ m ⇒ v_ext list ⇒ v list ⇒ (s × inst ×
       module_export list) ⇒ bool" where
2    "⟦inst = ⦇types=(m_types m),
```

```
3             funcs=(ext_funcs imps)@i_fs,
4             tabs=(ext_tabs imps)@i_ts,
5             mems=(ext_mems imps)@i_ms,
6             globs=(ext_globs imps)@i_gs⦈;
7       alloc_funcs s (m_funcs m) inst = (s1,i_fs);
8       alloc_tabs s1 (m_tabs m) = (s2,i_ts);
9       alloc_mems s2 (m_mems m) = (s3,i_ms);
10      alloc_globs s3 (m_globs m) gvs = (s',i_gs);
11      exps = map (λm_exp. ⦇E_name=(E_name m_exp), E_desc=(export_get_v_ext
            inst (E_desc m_exp))⦈) (m_exports m)
12       ⟧ ⟹ alloc_module s m imps gvs (s',inst,exps)"
```

Module allocation is the step in which the functions, tables, memories, and globals defined in the module are included in the store, as defined in lines 7 to 10 in Listing 3.2. An instance is returned which contains the addresses of functions, tables, memories, and globals used by the module (lines 2 to 6). The globals are initialised with their initial values, hence the `gvs` argument. The returned instance contains not only addresses of newly allocated items, but also the addresses of imports, as included by the `imps` argument.

3.2 Soundness

Soundness broadly refers to properties which well-typed programs are expected to hold. Despite there being no explicit claims in the WebAssembly specification on statements which should hold after instantiation, some desirable statements naturally arise given the rest of the specification:

- First of all, recall the Preservation and Progress theorems for WebAssembly's type system. The assumption they share is that the expression being evaluated is well-typed in regard to the store and frame. Therefore, the result of instantiation should fulfill that assumption.
- Moreover, the resulting module exports should be well-typed so that they can be imported by other modules.
- Lastly, typing judgments unrelated to the module being instantiated should be preserved. An easy way to achieve that is using the concept of *store extension*; broadly speaking, a store extends another store if can be obtained by appending additional data. WasmCert-Isabelle contains results on preservation of typing predicates under store extension, thus it is sufficient to prove that the new store extends the old one.

As for assumptions, it should be sufficient for the above to hold that the store is well-typed; the assumption should be preserved so that instantiation can be applied repeatedly. The resulting theorem is presented in Listing 3.3. It is worth noting that line 5 is exactly the assumption in Listing 2.3, connecting soundness of instantiation with soundness of execution.

Listing 3.3: Instantiation soundness theorem

```
1  theorem instantiation_sound:
2    assumes "store_typing s"
3            "instantiate s m v_imps ((s',f, init_es), v_exps)"
4    shows "store_typing s'"
5          "⊢ s'; f; init_es : []"
6          "∃tes. list_all2 (λv_exp te. external_typing s' (E_desc v_exp) te)
               v_exps tes"
7          "store_extension s s'"
```

3.3 Proof

The proof of instantiation soundness is not conceptually difficult; the main idea is to follow through each step of module instantiation and show that the typing invariants are preserved. However, due to a large number of definitions to work with it spans around one thousands lines of Isabelle proofs, which is too long to be presented fully.

The main technical challenge came from the allocations of functions, memories, tables, and globals (Listing 3.2, lines 7 to 10). Their definitions in WasmCert-Isabelle (shown in Listing 3.4 using memories as the example) follow the official specification, which defines them in terms of iterating over the list and updating the store each time. This definition however makes proving statements about the new store particularly unpleasant if used directly, in particular because of the need to use explicit induction on the recursive[1] definition every time. Induction is troublesome because Isabelle's automated proof methods can't[2] perform an inductive proof automatically; instead one needs to specify the inductive hypothesis explicitly, a bothersome process if done repeatedly. How to avoid excessive use of induction in this case then?

Listing 3.4: Allocation of memories as in the specification

```
1  fun alloc_Xs :: "(s ⇒ 'a ⇒ (s × i)) ⇒ s ⇒ 'a list ⇒ (s × i list)" where
2    "alloc_Xs f s [] = (s,[])"
3  | "alloc_Xs f s (m_X#m_Xs) = (let (s'', i_X) = f s m_X in
4                                 let (s',i_Xs) = alloc_Xs f s'' m_Xs in
5                                 (s',i_X#i_Xs))"
6
7  definition alloc_mem :: "s ⇒ mem_t ⇒ (s × i)" where
8    "alloc_mem s m_m = (s⦇s.mems := (mems s)@[mem_mk m_m]⦈,length (mems s))"
9
10 abbreviation "alloc_mems ≡ alloc_Xs alloc_mem"
```

[1] Iteration becomes recursion when expressed in a functional language.
[2] This is not a weakness of Isabelle – induction is the hard part in automated reasoning.

The resolution of the problem is to notice that the explicit recursive definition can be shown to be equivalent to a definition expressed with the `map`[3] function. The benefit of doing so is that Isabelle's tools are natively aware of various properties of list manipulation, unlike user-defined ad-hoc recursion. Moreover, there is now only one store update at the top of the definition rather than deep inside auxiliary definitions, further simplifying reasoning.

In other words, this idea splits the proof of soundness of instantiation into two stages:

- Show that the recursive definition and the simplified definition are equivalent.
- Complete the rest of the proof using the simplified definition.

This way we get the best of both worlds: the base definitions can be easily checked manually to reflect the official specification, while the actual proof can avoid the troublesome parts of the original definitions by utilising more convenient but equivalent definitions.

Listing 3.5 shows the simplified definition, as well as the lemma showing that the original definition can be expressed in terms of the simplified one. As an example of usage, line 9 shows the lemma that the memories newly allocated in `alloc_module` can be expressed with the simplified definition.

Listing 3.5: Simplified allocation of memories

```
1  definition alloc_mem_simple :: "mem_t ⇒ mem" where
2    "alloc_mem_simple m_m = mem_mk m_m"
3
4  abbreviation "alloc_mems_simple m_ms ≡ map alloc_mem_simple m_ms"
5
6  lemma alloc_mems_equiv:"fst (alloc_mems s m_ms) = s⦇mems := mems s @
       alloc_mems_simple m_ms⦈"
7  [...]
8
9  lemma alloc_module_mems_form:
10   assumes "alloc_module s m v_imps g_inits (s', inst, v_exps)"
11           "mems s' = mems s @ ms"
12   shows "ms = alloc_mems_simple (m_mems m)"
```

The simplified definitions later came in use during the work on the verified interpreter as well, showing their utility over the original definitions.

In summary, proving soundness of instantiation was mostly an exercise in using Isabelle, taking a decent amount of work but requiring no substantial new ideas. Fortunately, the proof found no issues with the formal definition of instantiation. Nevertheless, the existence of the proof increases our confidence in WebAssembly's safety, extending the area covered by formal verification.

[3] `map f [x1, x2, ...]` is the list `[f x1, f x2, ...]`.

Chapter 4

Interpreter

Aside from verifying the soundness of specification, another area where formal verification enhances the safety of the language is having a verified interpreter. Even if not used to run programs in practice, a verified interpreter serves as a reference for other engines about the behaviour prescribed by the specification on a given input. Therefore, in theory, such an interpreter can be used to increase the confidence in engines more focused on other factors by running both on randomly generated input and checking if the output is the same for both.

WasmCert-Isabelle already includes a verified interpreter of WebAssembly [1]. However, its performance suffers outside of small test instances, meaning its utility for testing purposes described earlier is very limited. One particular issue is modelling memory as an immutable list, causing $O(n)$ access time instead of $O(1)$ and thus increasing the time complexity. Because of that and other factors, prior to this project another interpreter had been written as a side project on WasmCert-Isabelle, with lists replaced by arrays where relevant, as well as other changes. The interpreter, while yet unverified, was fast enough that it could hypothetically be used in practice as a test oracle.

In this work I complete a formal proof of *soundness* for the faster interpreter. Soundness in the context of an interpreter is the property that, if the interpreter terminates successfully, its result is consistent with the specification. It is a weaker property than total correctness, which in addition implies that the interpreter always terminates successfully. The proof of soundness is accomplished by refinement from the slower interpreter, i.e. by incrementally showing that each function in the new interpreter behaves in a way corresponding to the respective function in the old, verified interpreter. The challenge of using mutable arrays was solved by modelling them with the state monad for imperative data structures; reasoning about the program's properties was made possible by employing separation logic, avoiding the frame problem.

The interpreter is thus proven to be sound, and moreover had earlier been checked to pass the WebAssembly test suite. Those properties led to its adoption in practice by the

Bytecode Alliance for their WebAssembly testing infrastructure.

4.1 Imperative HOL

Isabelle/HOL, aside from being a formal proof system, contains a small, ML-like functional language as a subset. Additionally, Isabelle contains support for exporting functions written in Isabelle/HOL to OCaml[1], known as *code extraction*. Therefore, one can have a program written within Isabelle/HOL, with immediate support for proving its properties.

However, since logical statements have to be independent from program state, all functions in Isabelle/HOL are pure by necessity. In other words, the functional language in Isabelle/HOL does not contain mutable data structures such as arrays directly, unlike SML and OCaml. The solution to that is the use of the Imperative HOL library, which defines mutable data structures in terms of a monad, similarly to Haskell.

The core challenge comes from the need to handle the program's mutable state, also known as the *heap*[2]. Imperative HOL's approach comes from the observation that an imperative program is essentially a partial[3] function which, given the initial heap, returns the program's result as well as the updated heap. This leads to the `'a Heap` datatype, which is used to represent an imperative expression evaluating to type `'a`:

```
1 datatype 'a Heap = Heap "heap ⇒ ('a × heap) option"
2
3 primrec execute :: "'a Heap ⇒ heap ⇒ ('a × heap) option" where
4   [code del]: "execute (Heap f) h = f h"
```

A special case of a `Heap` expression is a pure expression, i.e. one that does not modify the heap and can be expressed in plain Isabelle/HOL. Therefore in Imperative HOL it becomes a function which returns the unmodified heap, giving rise to the `return` combinator which converts a pure expression to an imperative one:

```
1 definition return :: "'a ⇒ 'a Heap" where
2   [code del]: "return x = Heap (λh. Some (x, h))"[4]
```

Furthermore, one might wish to execute two `Heap` expressions one after each other, chaining them together. The second expression should also contain a way to capture the result of the first expression; a way to do that is to accept it as an argument to a function. This idea results in the `bind` combinator:

```
1 definition bind :: "'a Heap ⇒ ('a ⇒ 'b Heap) ⇒ 'b Heap" where
```

[1] OCaml is not the only supported language, but it is the language the interpreter is extracted to.
[2] A similar but slightly different concept to heap memory as contrasted with stack memory.
[3] The expression might crash or loop indefinitely; hence the function is partial, not total.
[4] The actual definition in `Heap_Monad.thy` is different; I present an equivalent, more readable definition

```
2    [code del]: "bind f g = Heap (λh. case execute f h of
3                    Some (x, h') ⇒ execute (g x) h'
4                  | None ⇒ None)"
```

Formally speaking, the `return` and `bind` combinators together form a monad. In order to make writing programs easier,

$$\texttt{do } \{ \texttt{ x} \leftarrow \texttt{a; e } \}$$

is syntax sugar for `bind a (λx.e)` and intuitively means "execute `a`, saving the result in `x`, and execute `e`"; it can be compared to writing `x = a; e;` in C.

In a similar manner, array operations can be defined to return a `Heap` value by modelling them as functions modifying the array in the heap.

By wrapping mutable state in the `Heap` monad, reasoning about expressions with mutable state is made possible by making the program state explicit.

An additional utility of Imperative HOL appears when extracting code containing `Heap` expressions. Instead of passing the heap around explicitly, the heap as formally modelled in the definition of the `Heap` monad becomes the implicit heap in the target language. Thus an Isabelle/HOL expression such as `Array.upd i x a`, despite being specified as a wrapped function taking a heap, becomes `a.(i) <- x` in OCaml, which modifies the heap in-place. This allows one to write a program in Isabelle/HOL which is nevertheless efficient when extracted to another language such as OCaml.

4.2 Separation logic

Separation logic is an extension of Hoare logic designed to reason about shared mutable data structures [9]. Hoare logic is a framework of specifying and proving programs' properties with the *partial correctness triple* (also called a *Hoare triple*) as its central concept. A partial correctness triple for a program `C` is written as

$$\{P\}\,\texttt{C}\,\{Q\}$$

for a precondition P and and a postcondition Q. That Hoare triple then informally means "if P holds initially, and `C` successfully terminates, then Q holds as a result of running `C`". It is called a *partial* triple because it does not show that `C` successfully terminates (in contrast to a *total* correctness triple which does require that from `C`). Hoare logic can support pointers by introducing a $x \hookrightarrow y$ predicate, meaning "at address x there is a value y".

However, plain Hoare logic is poorly equipped to deal with a large number of pointers to

distinct mutable data structures. As an example, consider two pointers a and b to two locations both containing 0, and a program writing 1 to the address b is pointing to. A naive attempt at a Hoare triple would write

$$\{a \hookrightarrow 0 \land b \hookrightarrow 0\}\, b := 1\, \{a \hookrightarrow 0 \land b \hookrightarrow 1\},$$

which turns out incorrect. The precondition does not exclude the possibility that a and b both point to the same location, in which case writing to b would also modify the value referenced by a. A correct triple then is

$$\{a \hookrightarrow 0 \land b \hookrightarrow 0 \land a \neq b\}\, b := 1\, \{a \hookrightarrow 0 \land b \hookrightarrow 1 \land a \neq b\}.$$

One can notice that as the number of pointers in the program grows, then so does the number of pairs of pointers we need to assert are not equal, quickly becoming unmanageable. Moreover, Hoare triples even for a small function need to be aware of all other pointers in the program to show that running the function does not change anything in a different part of the program, significantly violating modularity. Thus plain Hoare logic does not scale up at all to deal with imperative programs with pointers to mutable data structures.

Separation logic is designed to overcome that issue. It specifies a language of *assertions*, which act as descriptions of the heap. Separation logic assertions extend the base logic used in Hoare triples by additional predicate symbols:

- **emp**, which reads as "the heap is empty".
- $x \mapsto y$, which strengthens $x \hookrightarrow y$ by "the heap consists of a pointer x pointing to y and nothing else" .

The key innovation however is the introduction of *separating conjunction*, denoted by $*$. Given two assertions P and Q, the assertion

$$P * Q$$

describes the heap h if and only if it can be partitioned into two disjoint heaps h_1 and h_2 such that P describes h_1 and Q describes h_2.

To see how separating conjunction increases the expressive power, consider the earlier example of two pointers a and b. If instead of $a \hookrightarrow 0 \land b \hookrightarrow 0$ we write

$$a \mapsto 0 * b \mapsto 0,$$

we get $a \neq b$ for free, as by definition of separating conjunction they belong to distinct parts of the heap.

To complete the description of separation logic, we need to modify the semantics of partial correctness triples as well. Now,

$$\{P\}\,\mathtt{C}\,\{Q\}$$

takes a more complex description of:
if the initial heap h can be partitioned into two distinct heaps h_P and h_O,
and P describes h_P,
and `C` terminates,
then the resulting h' after the execution of `C` can be partitioned into two distinct heaps h_Q and h_O such that h_O is the same as earlier and Q describes h_Q.

As an example, writing to a pointer can now be specified with a Hoare triple in separation logic as

$$\{p \mapsto x\}\, p := y\, \{p \mapsto y\}$$

which now not only says that the assignment modifies the value stored in p, but also that *the assignment doesn't change anything else in the heap.*

A consequence of the new semantics is the very powerful *frame rule*:

$$\frac{\{P\}\,\mathtt{C}\,\{Q\}}{\{P * F\}\,\mathtt{C}\,\{Q * F\}}$$

In other words, Hoare triples in separation logic can be extended with parts of the heap not affected by the expression. This provides the much-needed modularity, as the Hoare triple for each function and subexpression only needs to worry about the parts of the heap it touches and is unaffected by changes in other parts of the program.

The above informal description of separation logic is not exhaustive. The description of the $-\!*$ (separating implication, or "wand") operator is omitted, since it doesn't appear in the formal proofs this work reports on, and moreover various technical details have been glossed over in favour of a brief description.

4.3 Separation logic in HOL

We used Peter Lammich's library for separation logic for imperative structures in HOL [10]. The key concept in that library are assertions, represented by the type `assn`. An assertion is modelled as a predicate on partial heaps, subject to the restriction that it does not depend on the rest of the heap.

A common concept in formulations of separation logic for imperative languages is separating the *program variables* from *assertion variables*. Such separation is necessary in languages with mutable variables, as the value of a variable appearing in the program

is dependent on a given point in time. That distinction is however not necessary in Isabelle/HOL, as all variables are immutable, and hence the library makes no effort to distinguish between variables which appear in the program and those which do not.

The definition of separating conjunction follows as described earlier. There is however an interesting detail in how Hoare triples are implemented; in order to capture the result of executing the expression, the postcondition is a function which takes the result as the argument. In other words

```
<P> C <Q>
```

means "if `P` holds on a part of the heap before, and `C` terminates successfully with the result `r`, then executing `C` results in `Q r` describing the respective part of the heap while leaving the rest unchanged".

As an example, the Hoare triple for writing to a pointer looks as follows:

```
1 lemma update_rule:
2   "<p ↦r y> p := x <λr. p ↦r x>"
```

Since `p := x` doesn't return a meaningful value, the returned value is ignored in the postcondition. It can however be used in the postcondition when the type of the expression is non-trivial, as seen in the example of reading from a pointer:

```
1 lemma lookup_rule:
2   "<p ↦r x> !p <λr. p ↦r x * ↑(r = x)>"
```

where the $\uparrow(P)$ is the notation for the "P holds and the heap is empty" assertion, where P is a Boolean predicate.[5]

Lammich's main separation logic library works with total correctness triples. However, since partial correctness is sufficient for the purposes of this project, we use a modified version of the library with partial correctness triples instead. In particular, the expression is allowed to crash for the triple to still hold – the postcondition has to hold only if the expression terminates successfully without crashing. Because of that, some triples might look somewhat counterintuitive, as shown in Listing 4.1 on the example of array lookup.

Listing 4.1: The Hoare triple for array lookup

```
1 lemma nth_rule:
2   "<a ↦a xs>
3   Array.nth a i
4   <λr. a ↦a xs * ↑(r = xs ! i ∧ i<length xs)>"
```

One would naively expect that the `i<length xs` condition should appear in the precon-

[5]One might wonder if something like $p \mapsto_r x \wedge r = x$ wouldn't be simpler. However, the use of separating conjunction over Boolean conjunction allows the automated proof methods to do their work efficiently, as shown later.

dition, so that array lookup can take place safely. However, the program crashes if the index is outside of bounds, and crashes are not covered by the partial correctness triple; therefore `i<length xs` in the precondition is unnecessary. To the contrary, `i<length xs` appears in the postcondition – we know that the index must have been within bounds if the array lookup terminated successfully.

A key feature of the library is the `sep_auto` proof method. Given a Hoare triple to prove, `sep_auto` applies routine transformations working through the expression forwards, updating the precondition accordingly. Ideally, the entire expression is processed, simplifying the original Hoare triple to statements not involving separation logic, which can be proved afterwards with base Isabelle methods. This process is in general known as *verification condition generation*, and is essential for effective use of separation logic.

It is worth noting that `sep_auto` isn't magic; verification condition generation relies on being able to decide the underlying logic, which in most cases is simple enough but may not be trivial to automate. Because of that `sep_auto`, rather than generating the most general conditions possible, proceeds with a number of common-sense heuristics in order to be practical. In particular, it is essential for `sep_auto` to work well that preconditions and postconditions are expressed as a string of assertions joined together with separating conjunction. It might happen that `sep_auto` heuristics pick the wrong choice in verification condition generation; in that case tweaking[6] it manually is necessary. Finally, `sep_auto` does not work perfectly with higher-order functions, requiring more manual input.

4.4 Refinement proof

WasmCert-Isabelle already contains an interpreter proven to be sound [1]. Listing 4.2 contains its soundness theorems, expressed in words as "if the interpreter returns a list of values, that list of values is consistent with the specification" and "if the interpreter traps, trapping is consistent with the specification".

Listing 4.2: Soundness of the pure interpreter

```
1 theorem run_v_sound:
2   assumes "run_v fuel d (s, f, b_es) = (s', RValue vs)"
3   shows "(∃f'. reduce_trans (s,f,$*b_es) (s',f',v_stack_to_es vs))"
4
5 theorem run_v_sound_trap:
6   assumes "run_v fuel d (s, f, b_es) = (s', RTrap str)"
7   shows "(∃f'. reduce_trans (s,f,$*b_es) (s',f',[Trap]))"
```

[6]Example includes choosing the order of case splits and applications of `sep_auto` carefully.

That interpreter contains no mutable state, hence it's called the *pure interpreter* from now on. However, to simplify the proof of soundness, it models WebAssembly's containers (memories etc.) as lists, adding a polynomial factor to the time complexity and thus leading to unacceptably large execution overhead.

The new interpreter instead models the containers as arrays for constant-time lookup and update, using monadic state to do so, hence called the *monadic interpreter* from now on. Since the pure and monadic interpreter share the same structure, differing mostly only in the data structures used, the simplest way to prove the soundness of the monadic interpreter is to show that the pure and the monadic interpreter are, in a sense, equivalent – a technique known as *refinement*.

A simple example of the idea behind the refinement proof is included in Listing 4.3, which shows a Hoare triple indicating that `get_local`, the function responsible for fetching the value of a local variable onto the stack, and its monadic version `get_local_m`, are equivalent up to partial correctness.

Listing 4.3: Refinement of `get_local`

```
definition get_local :: "nat ⇒ f ⇒ v_stack ⇒ (v_stack × res_step)" where
  "get_local k f v_s =
     (if k < length (f_locs f)
        then (((f_locs f)!k)#v_s, Step_normal)
        else (v_s, crash_invalid)))"

[...]

definition get_local_m :: "nat ⇒ v array ⇒ v_stack ⇒ (v_stack ×
    res_step) Heap" where
  "get_local_m k loc_arr v_s =
     do {
       v ← Array.nth loc_arr k;
       return (v#v_s, Step_normal) }"

[...]

definition locs_m_assn :: "v list ⇒ v array ⇒ assn" where
  "locs_m_assn locs locs_m = locs_m ↦a locs"

lemma get_local_triple:
  "<locs_m_assn (f_locs f) f_locs1>
  get_local_m k f_locs1 v_s
  <λr. ↑(r = get_local k f v_s) * locs_m_assn (f_locs f) f_locs1>"
  unfolding locs_m_assn_def get_local_m_def get_local_def
  by sep_auto
```

One can easily notice that the `get_local` and `get_local_m` functions essentially do the

same thing; the main difference is that one fetches the value from a list, and the other from an array. The other difference is that the pure version crashes explicitly[7] with the use of a `crash_invalid` value if the index is out of bounds, while the monadic version crashes implicitly. However, because we are interested in partial correctness only, that difference is irrelevant. It is worth noting that partial correctness is sufficient for proving soundness.

To relate those functions together more precisely, we need to express something like "running `get_local_m` returns the same result as running `get_local` on the same arguments. Moreover, the monadic version, just as the pure version, doesn't modify anything".

How is that claim expressed? One way is to do it as follows:

- In the precondition, relate together the data structure in the monadic interpreter with one in the pure interpreter (line 21).
- In the postcondition, write that the result returned by the monadic expression is the same as the one which would have been returned by the pure one, using the pure data structure from the precondition (line 23). Moreover, copy and paste the precondition to express that nothing is modified.

This pattern, shown by the `get_local_m` example, repeats in every step of the refinement proof, even though it becomes more complex in other parts of the proof.

In this case, the Hoare triple is very simple to prove – it is sufficient to unfold the definitions and run `sep_auto`, an invaluable tool for the refinement proof. The `sep_auto` proof method allows to skip over parts which are identical or very similar, leaving only the more tricky parts to the human writing the proof. In fact, the refinement proof for 24 out of 39 WebAssembly's basic instructions has been handled entirely by specifying the definitions to unfold and running `sep_auto`.

4.4.1 Assertions on lists

Unfortunately, some refinement steps need a bit more work than the above. Listing 4.4 shows the case of `mem_size_m`, the function responsible for executing the instruction for returning the current size of a memory. It first obtains the index of the memory to answer the query about, and then looks up the memory of that index in the array of memories.

Listing 4.4: Refinement of `mem_size`

```
1  definition mem_size_m :: "mem_m array ⇒ inst_m ⇒ v_stack ⇒ (v_stack ×
       res_step) Heap" where
2    "mem_size_m ms i_m v_s =
3       do {
```

[7] The result of an out-of-bounds access to a list with `!` in Isabelle is an undetermined value, which may cause problems without an explicit bounds check.

```
4         j ← Array.nth (inst_m.mems i_m) 0;
5         m ← Array.nth ms j;
6         m_len ← len_byte_array (fst m);
7         return (((V_num (ConstInt32 (int_of_nat (m_len div Ki64))))#v_s),
              Step_normal)
8       }"
9
10  definition mems_m_assn :: "mem list ⇒ mem_m array ⇒ assn" where
11    "mems_m_assn ms ms_m = (∃A ms_i. ms_m ↦a ms_i * list_assn mem_m_assn ms
          ms_i)"
12
13  lemma mem_size_triple:
14    "< mems_m_assn ms ms_m * inst_m_assn (f_inst f) inst_m >
15      mem_size_m ms_m inst_m v_s
16    <λr. ↑(r = mem_size ms f v_s) *
17     mems_m_assn ms ms_m * inst_m_assn (f_inst f) inst_m >"
18    unfolding mem_size_m_def inst_m_assn_def mems_m_assn_def
          list_assn_conv_idx
19    apply (sep_auto split:prod.splits)
20     apply(extract_reinsert_list_idx "inst.mems (f_inst f) ! 0")
21    apply(sep_auto)
22    apply(simp add: app_s_f_v_s_mem_size_def smem_ind_def mem_size_def
23          mem_length_def mem_rep_length_def split: option.split list.split)
24    done
```

The difficulty is that now we have two layers of relating pure and monadic data structures. We first need to define the correspondence between a pure and monadic memory, and then relate a list of pure memories with the array of monadic memories, with the second step shown in `mems_m_assn`; the `list_assn P xs ys` assertion says "join the assertions `P x y` with `*` for each respective pair of `x` and `y` from `xs ys`".

Listing 4.5: Definition of `list_assn`, courtesy of Peter Lammich

```
1  fun list_assn :: "('a ⇒ 'c ⇒ assn) ⇒ 'a list ⇒ 'c list ⇒ assn" where
2    "list_assn P [] [] = emp"
3  | "list_assn P (a#as) (c#cs) = P a c * list_assn P as cs"
4  | "list_assn _ _ _ = false"
```

However, the trouble with `list_assn` is that `sep_auto` generally doesn't know how to process it automatically. Thus, when an assertion from inside of `list_assn` is needed to progress, it needs to be extracted from `list_assn` manually, and most of the time reinserted back afterwards. It is a common enough process that we defined a custom proof method for it called `extract_reinsert_list_idx`. Thus the proof in lines 19 to 24 consists of the initial application of `sep_auto`, extracting and reinserting the relevant assertion in the place where `sep_auto` can't progress, and finishing off the rest.

The proof in Listing 4.4 is also a simple example of the main workflow used to carry out

the refinement proof.

- Apply `sep_auto`.
- Figure out where it got stuck, and what it needs to proceed. Sometimes an extra fact is needed, and sometimes dissecting `list_assn` is necessary. Tweak and repeat.
- When you reach a goal that doesn't involve separation logic, apply base Isabelle proof methods to finish (similarly as in line 22 with `simp`).

Unfortunately, as much as clean short proofs are preferable, some of the refinement steps were notably more complex than the above, with multiple steps of extraction from and reinsertion to `list_assn`. One might ask whether at the point of 10 to 20 `apply` statements wouldn't it have been better to write an Isar proof for more structure instead, however in case of reasoning about chunks of code any intermediate lemmas tend to be too large to write down manually. The capacity of automated methods to process the intermediate steps implicitly is invaluable, despite occasional troubles.

4.4.2 Shared references

The next challenge comes when defining the assertions relating the pure and monadic data structures. While for the most part those assertions are easily defined, a tricky part appears. Both frames and function closures contain a reference to the current instance, thus a naive assertion relating the pure and monadic frame or closure would include `inst_m_assn` as well. However, two different closures or frames might point to the same instance, which means they couldn't be joined with separated conjunction in the naive approach, as they both "claim" the instance.

An elegant solution to this problem would have been to utilise either fractional or counting permissions in separation logic [11], since they are specifically suited for the case of multiple references with the "right" to access but not modify. However, the separation logic library used here does not support that, and switching it to one that does would have involved considerable effort. Instead, after some thought, the option I chose works as follows:

- Define `inst_assocs` to be the pair of lists[8] of monadic instances in use and their pure counterparts.
- Assert that the pure and monadic instances are equivalent to each other in the `inst_assocs`.
- Pass the `inst_assocs` as an argument to assertions on data structures containing references to instances, which in turn assert that the instance they refer to is

[8] A list of pairs would make more sense in an explanation, but in this proof a pair of lists turned out to be easier to manage.

contained within the `inst_assocs`.

Listing 4.6: Definition of `inst_assocs`

```
1 type_synonym inst_assocs = "(inst list × inst_m list)"
2
3 definition inst_assocs_assn :: "inst_assocs ⇒ assn" where
4   "inst_assocs_assn ≡ λ(insts, inst_ms). list_assn inst_m_assn insts
      inst_ms"
5
6 definition inst_at :: "inst_assocs ⇒ (inst × inst_m) ⇒ nat ⇒ bool" where
7   "inst_at ≡ λ(insts, inst_ms) (inst, inst_m) j. j < min (length insts)
      (length inst_ms)
8   ∧ insts!j = inst ∧ inst_ms!j = inst_m"
```

Listing 4.6 shows the implementation of this idea. Written this way, the assertions for data structures can be freely joined with separating conjunction. When reasoning about the referred instance, all that needs to be done is extracting the assertion about the instance from the `inst_assocs_assn`, and reinserting it back to the lists of instances when finished.

An example is shown in Listing 4.7, where we need to assert the equivalence of pure and monadic instances both in the current frame (`f_inst f`, `f_inst2`) and in function closures (`funcs_m_assn`). Since those instances might overlap, the equivalence needs to be expressed as `inst_assocs_assn` separately. Then the `inst_assocs` can be passed through to `funcs_m_assn`, asserting the equivalence of pure and monadic closures. Moreover, `f_inst f` and `f_inst2` can be related together simply by adding the assumption that they can be found in the `inst_assocs` at some index, using the `inst_at` predicate.

Listing 4.7: Refinement of `call_indirect`

```
1 lemma call_indirect_triple:
2   assumes "inst_at i_s (f_inst f, f_inst2) j"
3   shows
4   "<tabs_m_assn ts ts_m * funcs_m_assn i_s fs fs_m * inst_assocs_assn i_s>
5   call_indirect_m k ts_m fs_m f_inst2 v_s
6   <λr. ↑(r = call_indirect k ts fs f v_s)
7 * tabs_m_assn ts ts_m * funcs_m_assn i_s fs fs_m * inst_assocs_assn i_s>"
```

Since the proof of the proposition is relatively complex due to the large number of definitions involved, it is not shown in Listing 4.7. Despite that, it follows the structure shown earlier of repeatedly applying `sep_auto`, tweaking it manually to process the entire Hoare triple, and finishing off with Isabelle's proof methods.

4.4.3 Memory structure (and a bug)

The key difference between the monadic and pure interpreter is the use of arrays in place of lists. However, the monadic interpreter went one step further for improved performance, defining memories in terms of a custom array, exported as specialised OCaml primitives. This meant that the largest divergence between the two interpreters occured in handling of the `load` and `store` instructions, and subsequently the relevant proof of equivalence is possibly the largest part of the refinement proof.

This is also where the proof uncovered a bug in the monadic interpreter, shown in Listing 4.8, with line 6 and 8 showing the original and fixed version respectively. `store_packed` is a function that implements writing a number of bits to memory other than 32 or 64 bits; since WebAssembly supports only 32 and 64 bit numeric values directly, `store_packed` takes a value and writes only a part of its bits to memory. Since a failed write due to index out of bounds should trap, `store_packed` checks first whether the location to be written to is contained within bounds. Here comes the bug: the pure interpreter, as well as the specific, trap only if the bits which are actually being written to memory extend past its end, while the monadic interpreter in its original version traps if writing the entire numeric value to memory would extend past its end, even if the bits actually being written would not.

Listing 4.8: A bug in the monadic interpreter.

```
1  definition store_packed_m_v :: "mem_m ⇒ nat ⇒ off ⇒ v_num ⇒ tp_num ⇒
       (unit option) Heap" where
2    "store_packed_m_v m n off v tp =
3       do {
4         m_len ← len_byte_array (fst m);
5         (* original *)
6         (if (m_len ≥ (n+off+(t_length (typeof v)))) then do [...]
7         (* fixed *)
8         (if (m_len ≥ (n+off+(tp_num_length tp))) then do [...]
9          else return None)
10      }"
```

It is a bug which arises in a very specific edge case and thus not readily caught. It is unknown whether it would have been caught by testing, and we haven't investigated whether the bug would have had actual serious consequences. However, it is a clear example of formal verification ensuring correctness in action.

4.4.4 Interpreter top-level

After completing the Hoare triples for each basic instruction, the pure and monadic interpreters can be finally related together at a higher level, as shown in Listing 4.9. Both

interpreters work by keeping a representation of the program state (`config` in pure interpreter, `config_m` in monadic), and executing the instructions one by one. Thus we first show the Hoare triple that execution of a single instruction is equivalent in both interpreters, using `cfg_m_assn` (listed in full in Listing A.1 in the appendix) to relate them.

The proof in Listing 4.9 follows a case split on the instruction, dealing with each case individually. Line 16 illustrates as an example that the `Get_local` case is solved by calling `sep_auto`, passing `get_local_triple` (Listing 4.3) to its knowledge base.

Listing 4.9: Refinement of `run_step_b_e`

```
1  abbreviation cfg_m_pair_assn where
2    "cfg_m_pair_assn i_s ≡
3      λ(cfg, res) (cfg_m, res_m). cfg_m_assn i_s cfg cfg_m * ↑(res = res_m)"
4
5  lemma run_step_b_e_m_triple:
6      "<cfg_m_assn i_s cfg cfg_m>
7      run_step_b_e_m b_e cfg_m
8      <λr. cfg_m_pair_assn i_s (run_step_b_e b_e cfg) r>t"
9  proof −
10 [...]
11   show ?thesis
12   proof (cases b_e)
13   [...]
14     case (Get_local k)
15     then show ?thesis unfolding unfold_vars_assns
16       by (sep_auto heap:get_local_triple)
17 [...]
```

Subsequently, the `run_iter_m` is the function which specifies the execution loop; `run_iter_m n cfg` means "execute `n` steps[9] starting from the state `cfg`". It is defined recursively, decreasing the number of steps passed as the argument in each step. Therefore, this time the Hoare triple for `run_iter_m` is proved by induction on the number of steps, as shown in Listing 4.10. To prove the inductive step, the inductive assumption (denoted as `Suc`) is provided to `sep_auto` in order to be able to process the recursive call.

Listing 4.10: Refinement of `run_iter`, proved inductively

```
1  lemma run_iter_m_triple:
2      "<cfg_m_assn i_s cfg cfg_m>
3      run_iter_m n cfg_m
4      <λr. cfg_m_pair_assn i_s (run_iter n cfg) r>t"
5  proof(induct n arbitrary: i_s cfg cfg_m)
6    case 0
7    show ?case unfolding 0 by sep_auto
```

[9] In Isabelle all functions must be *total*, i.e. always terminate. The easiest way to express a loop in presence of that requirement is to add an argument representing the number of steps to execute.

```
next
  case (Suc n)
  [...]
  show ?case
  [...]
       apply(sep_auto heap:Suc)
  [...]
  done
qed
```

Finally, the refinement proof reaches the top level of the interpreter in `run_v_m`, one of the functions extracted to OCaml. Instead of starting from the internal representation as defined by `config`, `run_v_m` starts from the runtime configuration (i.e. the store, frame, list of instructions triple) as defined in WebAssembly semantics.

Listing 4.11: Refinement of `run_v`

```
lemma run_v_m_triple:
  assumes "inst_at i_s (f_inst f, f_inst2) j"
  shows "< s_m_assn i_s s s_m * inst_assocs_assn i_s * locs_m_assn (f_locs
      f) f_locs1 >
  run_v_m n d (s_m, f_locs1, f_inst2, b_es)
  <λ(s_m', res_m). let (s', res) = run_v n d (s, f, b_es) in
  ↑(res_m = res) * s_m_assn i_s s' s_m' * inst_assocs_assn i_s >t"
```

4.4.5 Instantiation in the interpreter

While that shows that the pure and the monadic interpreter return the same answer given some initial preconditions, one might question whether that precondition is actually attained when executing WebAssembly code. For that reason, we need to show that the interpreters agree also in the instantiation stage.

Listing 4.12: Refinement of `interp_instantiate`

```
lemma interp_instantiate_m_triple:
  "< s_m_assn i_s  s s_m * inst_assocs_assn i_s  >
  interp_instantiate_m s_m m v_imps
  <λ(s_m', res_m). let (s', res) = interp_instantiate s m v_imps in
  ∃A i_s'. ↑(res_inst_m_agree i_s' res res_m) * s_m_assn i_s' s' s_m' *
      inst_assocs_assn i_s' >t"
```

Listing 4.12 shows the Hoare triple stating that instantiation in the monadic interpreter returns the same result as instantiation in the pure interpreter, and that the invariant in the precondition is preserved.

One additional trouble in proving the instantiation part compared to the rest of the in-

terpreter came from the more frequent need to use iteration. In order to avoid explicit recursion we used higher-order functions such as `list_all2_m`; `list_all2_m f_m xs ys` evaluates to `True` iff `f_m x y` returns `True` for each respective pair of `x` and `y` from `xs ys`, where the difference from `list_all2` is the support for mutable state in the form of `'a Heap`. Subsequently, by expressing `list_all2_m` as a Hoare triple[10] (Listing A.2 in the appendix), the need to work with explicit recursion every time was avoided. However, `sep_auto` was found to not deal with higher-order functions perfectly, unfortunately requiring more manual input for the proof to proceed. Despite that, the full proof was achieved.

Moreover, the invariant holds initially when the store is empty, as shown in Listing 4.13, thus concluding that (repeated) instantiation works exactly as expected.

Listing 4.13: Hoare triple for empty store

```
1 lemma make_empty_store_m_triple:
2   "<emp>
3   make_empty_store_m
4   <λr. s_m_assn ([], []) ⦇s.funcs = [], tabs = [], mems = [], globs = [] ⦈
       r * inst_assocs_assn ([], [])>"
```

In conclusion, whenever the monadic interpreter terminates successfully, the result is equivalent to the result from the pure interpreter. Together with the soundness of the pure interpreter ("whenever the pure interpreter terminates successfully, the result is consistent with the specification"), that concludes the soundness of the monadic interpreter.

4.5 Putting it all together

So far the theorems presented illustrate the equivalence up to partial correctness between the pure and the monadic interpreter. Those Hoare triples might be considered the top-level theorems; the functions for initialisation, instantiation, and execution are exported separately to be called by the host environment, as prescribed by WebAssembly.

One possible objection is that soundness is vacuously true if the interpreter always crashes. However, the monadic interpreter passes the WebAssembly test suite, with no known practical inputs where it crashes when it should not. Therefore the interpreter's soundness is a meaningful property.

The other, more serious objection is that the relation between the pure and the monadic interpreter only holds if the precondition is satisfied. In an extreme example, if the

[10] A most general Hoare triple for `list_all2_m` would be too unwieldy due to the need to represent induction; however, the assumption that the function does not modify state eliminates induction while being sufficient for our purposes.

precondition can never be satisfied, the equivalence becomes vacuous. That objection takes more effort to refute.

This section presents how the interpreter is used in practice, and shows said end-to-end usage to be sound, dispelling doubts about the preconditions.

4.5.1 Fuzzing

As mentioned in the introduction, the monadic interpreter found its practical use. Naturally, multiple considerably faster WebAssembly engines exist; moreover, the pure interpreter could be said to be verified to a strictly greater extent[11] than the monadic one. However, the monadic interpreter uniquely combines acceptable performance and formally proven soundness to occupy a yet-unfilled niche. Those two characteristics piqued the interest of the people involved in the Bytecode Alliance when presented with the interpreter, given their utility for a particular automated testing method.

In *fuzzing*, the inputs for the program being tested are being generated randomly, in particular to cover cases which might have been overlooked in manually written tests. The verified interpreter here comes in use for the variant where the output of the application being tested on random input is compared with another one meant to return the same results; in this case a performance-focused WebAssembly engine needs to be checked whether its output follows the specification. However, before the monadic interpreter there was no sure practical way to check whether it's the case – both the verified pure interpreter and the official (but unverified) reference interpreter are too slow to use on larger input. The verified monadic interpreter finally fits that purpose, and hence was adopted for use in Bytecode Alliance's WebAssembly testing infrastructure.

The interpreter by itself can't be used as a test oracle "out of the box". Instead, we present the function which is used in the testing infrastructure to obtain meaningful output from the interpreter. The version presented here contains light edits made with the purpose of making the specification more readable.

4.5.2 Specification

Let's specify what we want to run for a self-contained result. One way is to instantiate a given module from scratch, and execute a function exported by said module. This leads to the specification as written in Listing 4.14, which, in words, says

- Instantiate a module `m` with imports `v_imps` and run its initializer expressions.
- Have the `i`-th export be a function of type `t1 _> t2`.

[11] Firstly, the pure interpreter is proven to not crash whenever the monadic interpreter does not crash; secondly, the pure interpreter does not utilise custom-written translations to OCaml.

- Interpret raw `args_bytes` as typed arguments for the function.
- Run the function and return the result `vs`.

Listing 4.14: Specification of `run_fuzz`

```
inductive run_fuzz_spec :: "m ⇒ v_ext list ⇒ nat ⇒ bytes list ⇒ e list ⇒
    bool" where
  "⟦instantiate' ⦇s.funcs = [], tabs = [], mems = [], globs = [] ⦈ m v_imps
     ((s1, f1, es), v_exps);
   reduce_trans (s1, f1, es) (s2, f2, []);
   E_desc (v_exps!i) = (Ext_func j);
   external_typing s2 (Ext_func j) (Te_func (t1 _> t2));
   length args_bytes ≥ length t1 ∧ map2 wasm_deserialise args_bytes t1 =
      args;
   reduce_trans (s2, empty_frame, ($C* args) @ [Invoke j]) (s3, f3, vs)
   ⟧ ⟹ run_fuzz_spec m v_imps i args_bytes vs"
```

4.5.3 Implementation and soundness

Given the specification, we can now write `run_fuzz_m` which fulfills it; the full implementation is found in Listing A.3 in the appendix. Similarly as in the rest of the repository, the proof is now split into two parts: showing that the monadic and pure versions are equivalent, and that the pure version fulfills the specification, as shown in Listing 4.15. In the end those two parts can be combined and the Hoare triple unpacked to say: *if `run_fuzz_m` returns a list of values as the result, the result follows the specification.*

Listing 4.15: Soundness of `run_fuzz_m`

```
lemma run_fuzz_m_triple:
  "<emp>
  run_fuzz_m n d m v_imps i args_bytes
  <λr. ↑(r = run_fuzz n d m v_imps i args_bytes)>ₜ"

lemma run_fuzz_run_fuzz_spec:
  assumes "run_fuzz n d m v_imps i args_bytes = (RValue vs)"
  shows "run_fuzz_spec m v_imps i args_bytes (v_stack_to_es vs)"

theorem run_fuzz_m_soundness:
  assumes "execute (run_fuzz_m n d m v_imps i args_bytes) h
           = Some (RValue vs, h')"
  shows "run_fuzz_spec m v_imps i args_bytes (v_stack_to_es vs)"
```

In summary, the verified monadic interpreter can be shown to be sound in end-to-end usage, as displayed using the example of `run_fuzz_m`. This concludes that the relation between the pure and the monadic interpreter, and therefore the monadic interpreter's soundness, is meaningful.

Chapter 5

Related work

The idea of a proof by refinement of increasingly optimised versions of the program isn't new. This approach has been well utilised in the area of SAT solvers [12, 13]. IsaSAT [14], a SAT solver written and verified in Isabelle, is notable of having won in a competition of unverified SAT solvers [15]. The Imperative Refinement Framework [16], which IsaSAT is based on, is an Isabelle library which provides support for refinement proofs, utilising Imperative HOL and separation logic as the backend for generation of verified imperative code from a more abstract specification.

The tools used to reason about WebAssembly are not limited to Isabelle. WasmCert-Coq [17, 18] is a parallel formalisation written using the Coq proof assistant. Similarly to WasmCert-Isabelle, WasmCert-Coq includes a proof of WebAssembly's type safety as well as a verified interpreter.

Formal verification has been also used for languages other than WebAssembly. JSCert [19], a JavaScript formalisation project, can be considered a predecessor to WasmCert. It formalised the ES5 revision of JavaScript in Coq, and provided a verified interpreter. CompCert [20, 21] is an ongoing project aiming to formalise the semantics of C and produce a verified compiler in Coq; at the time of writing it covers almost all of ISO C 99 and generates code for multiple architectures, including ARM and x86. CakeML [22, 23] is a similar project utilising the HOL4 theorem prover to implement a verified, bootstrapping compiler for a subset of Standard ML[1]. Jinja [6, 24] is a Java-like programming language with a full formal specification in Isabelle, proof of type safety, and a verified compiler.

[1]Particularly relevant due to the wide use of languages in the ML family (SML, OCaml) in implementing proof assistants.

Chapter 6

Conclusions

The project achieved and exceeded its planned goals. As a result, the formally verified safety guarantees of WebAssembly have been extended to cover module instantiation, increasing the confidence in the specification. Furthermore, accomplishing the proof of soundness of the improved interpreter led it to be adopted by the Bytecode Alliance for testing purposes. It is rare for a formal verification project to have practical ramifications, which makes industry adoption due to formal verification's benefits a significant claim.

In this project we focused on partial correctness of the interpreter; knowing that the interpreter's output is correct if the interpreter didn't crash is more important for the use in testing than knowing that the interpreter doesn't crash. However, for completeness, one can further strengthen the proof to use total correctness Hoare triples, as a result showing that the interpreter does not crash.

Furthermore, while assuming a trusted computing base (which here in particular includes the OCaml compiler) is commonly accepted practice, one might want to complete an end-to-end proof. In that case one would have to transition to a complete verified ecosystem such as CakeML [23] as the backend for the interpreter; I have not tried to estimate the difficulty of doing so.

This project was feasible only thanks to the current state of formal verification tools: Isabelle with Sledgehammer [4] made it possible to make progress on the project straight away without substantial prior experience, and `sep_auto` reduced the amount of work by an order of magnitude. I expect that formal verification projects will become more and more common as capabilities and accesibility of the relevant tools improve with time. In particular, as invaluable as `sep_auto` has been, a significant part of its effective use is learning its idiosyncracies and limitations; a hypothetical improved version might reduce the time spent in places where the tool fails due to weird edge cases or unsupported modes of reasoning, leaving only the cases where genuine insight is required to progress. Moreover, Sledgehammer is not magic; throughout the project I frequently encountered fairly uninteresting goals which nevertheless were too complex for Sledgehammer to finish

off, and so required my closer attention.

Keeping a repository of formal proofs is in many ways similar to software engineering, and one aspect in particular relevant here is maintenance. It is likely that WebAssembly's specification will keep being updated in face of changing requirements; in particular we had to add support for vector operations, introduced in most recent versions of WebAssembly. This means that the formal proofs will have to be updated in the future accordingly in order to keep their relevance.

To summarise, this project further increased the confidence in WebAssembly's safety and security, both in terms of veryfing the specification itself, and enabling more testing techniques for WebAssembly engines used in practice. The WasmCert-Isabelle repository extended by this project, if maintained, will continue preventing issues that might otherwise occur as WebAssembly evolves during its spread, and improving confidence in their absence. Finally, hopefully this and other successful examples will further popularise the use of formal verification to guarantee security and safety.

Bibliography

[1] Conrad Watt. "Mechanising and verifying the WebAssembly specification". In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (2018). DOI: 10.1145/3167082.

[2] Andreas Haas et al. "Bringing the web up to speed with WebAssembly". In: *ACM SIGPLAN Notices* 52.6 (2017), pp. 185–200. DOI: 10.1145/3140587.3062363.

[3] *Isabelle*. https://www.cl.cam.ac.uk/research/hvg/Isabelle/index.html, accessed 2022-05-21.

[4] Lawrence Paulson. "Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers". In: *PAAR-2010: Proceedings of the 2nd Workshop on Practical Aspects of Automated Reasoning*. Ed. by Renate A. Schmidt, Stephan Schulz, and Boris Konev. Vol. 9. EPiC Series in Computing. EasyChair, 2012, pp. 1–10. DOI: 10.29007/tnfd.

[5] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09* (2009). DOI: 10.1145/1629575.1629596.

[6] Gerwin Klein and Tobias Nipkow. "Jinja is not Java". In: *Archive of Formal Proofs* (2005).

[7] Conrad Watt et al. "Two mechanisations of WebAssembly 1.0". In: *Formal Methods* (2021), pp. 61–79. DOI: 10.1007/978-3-030-90870-6_4.

[8] *WasmCert/WasmCert-Isabelle: A mechanisation of Wasm in Isabelle*. https://github.com/WasmCert/WasmCert-Isabelle, accessed 2022-05-21.

[9] J.C. Reynolds. "Separation logic: A logic for shared mutable data structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science* (2002). DOI: 10.1109/lics.2002.1029817.

[10] Peter Lammich and Rene Meis. "A Separation Logic Framework for Imperative HOL". In: *Archive of Formal Proofs* (2012).

[11] Richard Bornat et al. "Permission accounting in Separation Logic". In: *ACM SIGPLAN Notices* 40.1 (2005), pp. 259–270. DOI: 10.1145/1047659.1040327.

[12] Filip Marić. "Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL". In: *Theoretical Computer Science* 411.50 (2010), pp. 4333–4356. DOI: 10.1016/j.tcs.2010.09.014.

[13] Jasmin Christian Blanchette et al. “A verified SAT solver framework with learn, forget, restart, and incrementality”. In: *Journal of Automated Reasoning* 61.1-4 (2018), pp. 333–365. DOI: 10.1007/s10817-018-9455-7.

[14] Mathias Fleury. “Optimizing a verified SAT solver”. In: *Lecture Notes in Computer Science* (2019), pp. 148–165. DOI: 10.1007/978-3-030-20652-9_10.

[15] *EDA Challenge, Fixed CNF Encoding Race results*. https://www.eda-ai.org/results/, accessed 2022-05-25.

[16] Peter Lammich. “The Imperative Refinement Framework”. In: *Archive of Formal Proofs* (2016).

[17] Xuan Huang. “A Mechanized Formalization of the WebAssembly Specification in Coq”. In: 2019.

[18] *WasmCert/WasmCert-Isabelle: A mechanisation of Wasm in Coq*. https://github.com/WasmCert/WasmCert-Coq, accessed 2022-05-25.

[19] Martin Bodin et al. “A trusted mechanised JavaScript specification”. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2014). DOI: 10.1145/2535838.2535876.

[20] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Communications of the ACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814.

[21] *CompCert*. https://compcert.org/, accessed 2022-05-25.

[22] Ramana Kumar et al. “CakeML”. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2014). DOI: 10.1145/2535838.2535841.

[23] *CakeML*. https://cakeml.org/, accessed 2022-05-17.

[24] Gerwin Klein and Tobias Nipkow. “A machine-checked model for a Java-like language, virtual machine, and compiler”. In: *ACM Transactions on Programming Languages and Systems* 28.4 (2006), pp. 619–695. DOI: 10.1145/1146809.1146811.

Appendix A

Additional listings

Listing A.1: Relating pure `config` and monadic `config_m`

```
definition "inst_m_assn i i_m ≡
    inst_m.types i_m ↦a inst.types i
  * inst_m.funcs i_m ↦a inst.funcs i
  * inst_m.tabs  i_m ↦a inst.tabs i
  * inst_m.mems  i_m ↦a inst.mems i
  * inst_m.globs i_m ↦a inst.globs i"

type_synonym inst_assocs = "(inst list × inst_m list)"

definition inst_assocs_assn :: "inst_assocs ⇒ assn" where
  "inst_assocs_assn ≡ λ(insts, inst_ms). list_assn inst_m_assn insts
     inst_ms"

definition inst_at :: "inst_assocs ⇒ (inst × inst_m) ⇒ nat ⇒ bool " where
  "inst_at ≡ λ(insts, inst_ms) (inst, inst_m) j. j < min (length insts)
     (length inst_ms)
  ∧ insts!j = inst ∧ inst_ms!j = inst_m"

abbreviation "contains_inst i_s i ≡ ∃j. inst_at i_s i j"

definition cl_m_agree_j :: "inst_assocs ⇒ nat ⇒ cl ⇒ cl_m ⇒ bool" where
  "cl_m_agree_j i_s j cl cl_m = (case cl of
  cl.Func_native i tf ts b_es ⇒
    (case cl_m of
    cl_m.Func_native i_m tf_m ts_m b_es_m ⇒
    inst_at i_s (i, i_m) j ∧ tf = tf_m ∧ ts = ts_m ∧ b_es = b_es_m
  | cl_m.Func_host tf_m host_m ⇒ False)
| cl.Func_host tf host ⇒
    (case cl_m of
    cl_m.Func_native i_m tf_m ts_m b_es_m ⇒ False
  | cl_m.Func_host tf_m host_m ⇒ tf = tf_m ∧ host = host_m)
)"
```

```
31
32  definition "cl_m_agree i_s cl cl_m ≡ ∃j. cl_m_agree_j i_s j cl cl_m"
33
34  definition funcs_m_assn :: "inst_assocs ⇒ cl list ⇒ cl_m array ⇒ assn"
      where
35    "funcs_m_assn i_s fs fs_m = (∃_A fs_i. fs_m ↦_a fs_i *↑(list_all2
        (cl_m_agree i_s) fs fs_i))"
36
37  definition tabinst_m_assn :: "tabinst ⇒ tabinst_m ⇒ assn" where
38    "tabinst_m_assn = (λ(tr,tm) (tr_m,tm_m). tr_m ↦_a tr * ↑(tm = tm_m))"
39
40  definition "mem_m_assn ≡ λ(mr,mm) (mr_m,mm_m). mr_m ↦_baRep_mem_rep mr * ↑
      (mm_m=mm)"
41
42  definition mems_m_assn :: "mem list ⇒ mem_m array ⇒ assn" where
43    "mems_m_assn ms ms_m = (∃_A ms_i. ms_m ↦_a ms_i * list_assn mem_m_assn ms
        ms_i)"
44
45  definition tabs_m_assn :: "tabinst list ⇒ tabinst_m array ⇒ assn" where
46   "tabs_m_assn ts ts_m = (∃_A ts_i. ts_m ↦_a ts_i * list_assn tabinst_m_assn
       ts ts_i)"
47
48  definition "globs_m_assn gs gs_m ≡ gs_m ↦_a gs"
49
50  definition s_m_assn :: "inst_assocs ⇒ s ⇒ s_m ⇒ assn" where
51    "s_m_assn i_s s s_m =
52    funcs_m_assn i_s (s.funcs s) (s_m.funcs s_m)
53  * tabs_m_assn  (s.tabs s)  (s_m.tabs s_m)
54  * mems_m_assn  (s.mems s)  (s_m.mems s_m)
55  * globs_m_assn (s.globs s) (s_m.globs s_m)"
56
57  definition locs_m_assn :: "v list ⇒ v array ⇒ assn" where
58    "locs_m_assn locs locs_m = locs_m ↦_a locs"
59
60  definition fc_m_assn :: "inst_assocs ⇒ frame_context ⇒ frame_context_m ⇒
      assn" where
61    "fc_m_assn i_s fc fc_m = (
62    case fc of Frame_context redex lcs nf f ⇒
63    case fc_m of Frame_context_m redex_m lcs_m nf_m f_locs1 f_inst2 ⇒
64    ↑(redex = redex_m ∧ lcs = lcs_m ∧ nf = nf_m ∧ contains_inst i_s (f_inst
        f, f_inst2))
65    * locs_m_assn (f_locs f) f_locs1
66  )"
67
68  definition "fcs_m_assn i_s fcs fcs_m ≡ list_assn (fc_m_assn i_s) fcs fcs_m"
69
70  definition cfg_m_assn :: "inst_assocs ⇒ config ⇒ config_m ⇒ assn" where
71    "cfg_m_assn i_s cfg cfg_m = (
72    case cfg of Config d s fc fcs ⇒
```

```
  case cfg_m of Config_m d_m s_m fc_m fcs_m ⇒
  ↑(d=d_m)
  * s_m_assn i_s s s_m * fc_m_assn i_s fc fc_m * fcs_m_assn i_s fcs fcs_m
  * inst_assocs_assn i_s
)"
```

The Hoare triples for higher-order functions `fold_map` and `list_all2_m` are written in the format of *deconstruction rules* in order to be more usable with `sep_auto`.

Listing A.2: Deconstruction rules for higher-order functions

```
lemma fold_map_decon:
  assumes "list_all R xs"
  assumes "⋀x. R x ⇒ <P> f x <λr. Q x r * P>"
  assumes "⋀ys. list_assn Q xs ys * P ⇒_A Q' ys"
  shows
    "<P> Heap_Monad.fold_map f xs <Q'>"

lemma list_all2_m_decon:
  assumes "⋀ x y. <P> f_m x y <λr. ↑(r = f x y) * P>"
  assumes "P ⇒_A Q' (list_all2 f xs ys)"
  shows
    "<P> list_all2_m f_m xs ys <Q'>"
```

Listing A.3: Fuzzing using the monadic interpreter

```
fun run_fuzz_m :: "fuel ⇒ depth ⇒ m ⇒ v_ext list ⇒ nat ⇒ bytes list ⇒
    res Heap" where
  "run_fuzz_m n d m v_imps i args_bytes = do {
   init_s ← make_empty_store_m;
   i_res ← interp_instantiate_init_m init_s m v_imps;
   case i_res of
     (s', RI_res_m inst v_exps es) ⇒
      (case es of
        [] ⇒ (if i < length v_exps then
            case (E_desc (v_exps!i)) of Ext_func j ⇒ do {
               cl ← Array.nth (s_m.funcs s') j;
               case (cl_m_type cl) of
                 (t1 _> t2) ⇒
                    if length args_bytes     length t1 then
                   (let params = map2 wasm_deserialise args_bytes t1 in
                   do {
                   (s'', res) ← run_invoke_v_m n d (s', params, j);
                   return res })
                else return (RCrash (Error_invariant (STR ''not enough
                    arguments'')))
           } | _ ⇒ return (RCrash (Error_invariant (STR ''not a
               function'')))
             else return (RCrash (Error_invariant (STR ''out of
```

```
                    bounds'')))))
21        | _ ⇒ return (RCrash (Error_invalid (STR ''not run fully'')))))
22    | (s', RI_crash_m res) ⇒ return (RCrash res)
23    | (s', RI_trap_m msg) ⇒ return (RTrap msg) }"
```