Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Designing and Maintaining an Efficient WebAssembly Mechanisation

Xiaojia Rao

July 16, 2025

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of Imperial College London and the Diploma of Imperial College London.

Declaration of Originality

This thesis constitutes a substantial body of research developed in collaboration with other researchers where I was the lead or co-lead of the work. I, Xiaojia Rao, declare that the specific contributions of my collaborators have been appropriately acknowledged at the end of each chapter. A complete list of references is provided in the bibliography.

Copyright

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution 4.0 International Licence (CC BY).

Under this licence, you may copy and redistribute the material in any medium or format for both commercial and non-commercial purposes. You may also create and distribute modified versions of the work. This on the condition that you credit the author.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Abstract

WebAssembly is a rapidly evolving binary format originally designed for efficient and safe execution of low-level languages like C and Rust on the web supported by all major browsers. Since its release in 2017, it has expanded to diverse execution environments and has grown extensively in its features. The importance of the language and its design with a formal specification makes it a compelling subject for programming languages and formal verification research.

This thesis focuses on my contributions to studies around the WebAssembly semantics and verification. I contributed a major part in the WasmCert-Coq mechanisation of the WebAssembly in Coq, starting from its inception which drew definitions from Watt's Isabelle/HOL mechanisation for a pre-1.0 version of WebAssembly, until my update of the semantics to Wasm 2.0 and beyond, which also produces a fully conformant interpreter as its extracted artifact. My work on the WebAssembly mechanisation has discovered and helped fix several errors in the WebAssembly specification, fulfilling the promise of the language mechanisation for refining an important non-mechanised specification.

In addition, through my work on WebAssembly mechanisation, several methods have been explored and implemented to reduce the effort of maintaining mechanised models of WebAssembly. In particular, my work on the progressful interpreters proposes a new method using a dependently-typed design to merge the interpreter, its correctness, and the progress property into one mechanisation artifact.

Besides designing the mechanisation of WebAssembly's semantics, I demonstrated the usefulness of language mechanisation to the formal verification by jointly developing the Iris-Wasm program logic using the Iris separation logic framework, which supports modular reasoning about WebAssembly programs, including host-WebAssembly interactions and module instantiation.

My efforts provide a foundation for efficient, scalable, and reliable verification of WebAssembly as it continues to evolve, ensuring its role as a secure and efficient platform for modern computation.

Acknowledgements

I gratefully thank my supervisor, Philippa Gardner, for introducing me to the evolving world of WebAssembly research and all the generous support and kind advice she has provided during my PhD, including but not limited to my research topics.

I would like to express profound appreciation to Conrad Watt, for the extensive and inspirational discussions we've shared throughout the years of my PhD. His insights and encouragement have been instrumental to every project I have undertaken.

It is my great pleasure to have worked with many of my collaborators and along many of my peers in Philippa's research group. In particular, I would like to thank Martin Bodin, who has guided me in my starting stages of the research; Sacha Elie-Ayoun and Petar Maksimovic, for their sharing of experiences in teaching and research; my technical collaborators across various projects, and my mentees from Imperial College's master projects through the years, where I have enjoyed the times we worked together to overcome technical challenges.

I would also like to thank many of my friends: Song, Tianhan, Yuchen, Shenghan and many others, who helped me through the lockdown time and motivated me through the difficult starting years of my PhD.

I deeply treasure the reliable and enjoyable accompaniment of my partner, Runcong Zhao, throughout our journey since the undergraduate time.

Finally, I express my utmost gratitude to my parents for their unwavering supports over my life, without which I could not have been in my position today.

Contents

1 Introduction **7**

1.1 Mechanisation of WebAssembly 8
1.2 Dependently-typed Progressful Interpreters 9
1.3 Modular Verification of WebAssembly Programs 10

2 Background **12**

2.1 An Overview of the Wasm Specification 12
2.2 WebAssembly's Runtime Structure 14
2.3 The Wasm 1.0 Instruction Set and Runtime Semantics 20
2.3.1 Numeric Instructions 20
2.3.2 Parametric Instructions 21
2.3.3 Variable Instructions 22
2.3.4 Memory Instructions 23
2.3.5 Control Flow Instructions 24
2.4 Wasm's Type System 29
2.4.1 Typing Contexts 30
2.4.2 Typing Rules for Wasm 1.0 Instructions 31
2.5 Administrative Instruction Typing and Validity Relations 35
2.5.1 Module Instance, Frame, and Thread Validity 35
2.5.2 Store Validity 36
2.5.3 Configuration Validity 37
2.6 Wasm's Type Soundness Property 37
2.7 Wasm Modules, Validation, and Instantiation 39
2.7.1 Validation of Wasm Modules 40
2.7.2 Instantiation of Wasm modules 43
2.8 Example: Fibonacci Module 44
2.8.1 Overview of the Module 44
2.8.2 Lifecycle of the Fibonacci Module 46

3 The WasmCert-Coq Mechanisation for WebAssembly **48**

3.1 Overview of the WasmCert-Coq Mechanisation 48
3.2 Data Types and Instruction Syntax 49
3.3 Operational Semantics 51

3.3.1 Representation of Host Function Invocations . . . 52
3.3.2 Formulation of Block Contexts . . . 53
3.4 Verified Interpreter . . . 55
3.4.1 Structure of the Interpreter . . . 55
3.4.2 Proof of Interpreter Correctness . . . 58
3.5 Type System and Soundness . . . 60
3.5.1 Preservation . . . 60
3.5.2 Progress . . . 67
3.6 Verified Type Checker . . . 69
3.7 Verified and Executable Module Instantiation . . . 72
3.8 Numerics . . . 74
3.9 Binary Format Conversion . . . 75
3.10 Updating WasmCert to WebAssembly 2.0 . . . 76
3.10.1 The WebAssembly 2.0 Feature Extensions . . . 76
3.10.2 Challenges of The WasmCert-Coq Update for Wasm 2.0 . . . 83
3.10.3 Specification Mistakes Found . . . 87
3.11 Testing . . . 89
3.11.1 The WebAssembly Test Suite . . . 89
3.11.2 Testing WasmCert-Coq . . . 90
3.11.3 Bugs and Inadequacies Discovered . . . 91
3.12 Related Work . . . 96
3.13 Acknowledgements . . . 97

4 Progressful Interpreters for Efficient Mechanisations **99**
4.1 Background . . . 99
4.2 Interpreter from Progress . . . 101
4.2.1 Methodology . . . 102
4.2.2 Limitations . . . 104
4.3 Progress from Progressful Interpreter . . . 110
4.3.1 Connection to Intrinsically Typed Languages . . . 111
4.3.2 Progressful Interpreter for WebAssembly 1.0 . . . 112
4.3.3 Optimising the Augmented Interpreter: Efficient Runtime Representation 116
4.4 Evaluation . . . 118
4.4.1 Runtime performance . . . 118
4.4.2 Proof engineering effort . . . 120
4.5 Related Work . . . 123
4.6 Acknowledgements . . . 124

5 Higher-Order Program Logic for WebAssembly **125**
5.1 Introduction . . . 125
5.2 Instantiating the WasmCert Semantics in Iris . . . 127
5.2.1 Key Concepts . . . 127
5.2.2 Pure Rules . . . 131

5.2.3 Control and Function Call Rules 132
5.2.4 Higher-Order Code with `call_indirect` 138
5.3 Higher-Order Programming Example in WebAssembly and Reentrancy 139
5.4 A Minimal Wasm-JS Host 142
5.5 Logical Relation and Robust Safety Examples 146
5.6 Mechanisation Summary 150
5.7 Related Work 151
5.8 Acknowledgements 152

6 Conclusion and Future Work **154**

Bibliography **157**

A WebAssembly Numeric Operations **166**
A.1 Unary Operations 166
A.2 Binary Operations 166
A.3 Test Operations 167
A.4 Relation Operations 167
A.5 Conversion Operations 167
A.6 Additional Operations in Wasm 2.0 167

B Wasm 1.0 Validity Relations **168**
B.1 Module Instance Validity 168
B.2 Limit Validity 169
B.3 Store Validity 170

C Iris-Wasm Proof Rules **171**
C.1 Pure Rules 171
C.2 Bind Rules 172
C.3 Function Call Rules 172
C.4 Stateful Rules 172
C.5 Control Rules 173

List of Figures

2.1 Different encodings of a module implementing a Fibonacci function 15
2.2 Indirections from module instance components to the Wasm store 17
2.3 Wasm's convention: the combined list representation of instructions 17
2.4 Structure of the Wasm configuration tuple 19
2.5 Core instruction syntax of Wasm 1.0 20
2.6 Top-level abstract reduction rules for Wasm numeric instructions 21
2.7 Reduction rules for parametric instructions 22
2.8 Reduction rules for variable instructions 22
2.9 Reduction rules for memory instructions 25
2.10 Reduction rules for control instructions (simple) 26
2.11 Reduction rules for control instructions (block constructs) 28
2.12 Reduction rules for control instructions (function calls) 29
2.13 Wasm 1.0 typing context 31
2.14 Typing rules of WebAssembly 1.0 (context-agnostic) 32
2.15 Typing rules of WebAssembly 1.0 (variables and memory) 32
2.16 Typing rules of WebAssembly 1.0 (control flow: block instructions) 34
2.17 Typing rules of WebAssembly 1.0 (function call instructions) 34
2.18 Typing rules of WebAssembly 1.0 (structural) 34
2.19 Illustration of the subsumption typing rule 35
2.20 Overview of Wasm validity relations 35
2.21 Validity relations and typing rule of **frame** administrative instruction 37
2.22 Structure of Wasm 1.0 modules 39
2.23 AST representation of the Fibonacci module (Revisit) 44
2.24 A C code fragment corresponding to the Fibonacci function 45
2.25 An example JavaScript code interacting with the Fibonacci module 46

3.1 Comparison between different versions of Wasm mechanisation 49
3.2 Selected WasmCert-Coq definitions for Wasm datatypes 50
3.3 Summary of the definition of operational semantics in WasmCert-Coq 52
3.4 Comparison between the implementations of block contexts in Watt's Isabelle mechanisation and the Coq mechanisation 54
3.5 Structure of the WasmCert-Coq one-step interpreter and relevant definitions . . . 56
3.6 Implementation of the one-step interpreter for **br** and **label** 57

3.7 Structure of typing relations for instructions and configurations of Wasm 1.0 . . . 61
3.8 Statements of the type soundness property in the WasmCert-Coq mechanisation for Wasm 1.0 . . . 61
3.9 Selected Typing Inversion Lemmas . . . 63
3.10 Illustration of type-checking a simple program . . . 70
3.11 Specification for the special cases in floating point multiplication . . . 75
3.12 Extended primitive types and values . . . 77
3.13 Reference and table operations . . . 78
3.14 Bulk table and memory operations . . . 81
3.15 Syntax extension of the wast format . . . 90
3.16 Binary format of Wasm 2.0 control instructions . . . 92
3.17 Specification of refs field of typing context . . . 92

4.1 Definitions of progress, preservation, and type inference as computable functions 102
4.2 Selected reduction and typing rules for **label** . . . 105
4.3 Typing term of a program with deeply nested **labels** . . . 106
4.4 Benchmark: $O(n)$ iterative Fibonacci with input n . . . 107
4.5 Pseudocode of the Wasm 1.0 progressful one-step interpreter . . . 114
4.6 Optimised runtime representation for Wasm interpreter and context-instruction composition . . . 117
4.7 List of interpreters compared . . . 118
4.8 Benchmark: Iterative Fibonacci with input n, log-scale . . . 119
4.9 Benchmark: Recursive Fibonacci with input n, log-scaled time axis . . . 119
4.10 List of interpreters compared . . . 120
4.11 Different attempts of implementing interpreter execution for **local.get** . . . 122

5.1 Points-to predicates for the store and the frame . . . 129
5.2 Selected core program fragment of the Fibonacci function . . . 133
5.3 A module implementing a stack library, and a client module. . . . 138
5.4 Host Syntax (definitions reference the grammar in Fig. 2.5) . . . 142
5.5 Robust safety example: applying map on an imported function . . . 147
5.6 Lines of code of the Iris development, as given by cloc . . . 150

A.1 Wasm 1.0 unary operations . . . 166
A.2 Wasm 1.0 binary operations . . . 166
A.3 Wasm 1.0 test operations . . . 167
A.4 Wasm 1.0 relation operations . . . 167
A.5 Wasm 1.0 conversion operations . . . 167

Chapter 1

Introduction

WebAssembly is an evolving binary format originally designed as a compilation target for low-level languages such as C/C++ and Rust for efficient and safe execution on the Web. It surpasses Mozilla's asm.js [10] and Google's Native Client [118], the two major initiatives from the early 2010s that aimed to enhance the efficiency and security of code execution in web browsers. These efforts emerged as JavaScript, originally designed in 1995, became increasingly unsuitable for newer computation-heavy use cases such as 3D visualization, video and audio streaming, gaming, and other web applications. Google's Native Client introduced sandboxing technology to safely and efficiently run x86 native code on the web, later expanding to support additional instruction sets. Meanwhile, asm.js provided a restricted subset of JavaScript with C-style manual memory management, enabling performance optimisations. However, neither technology achieved full support across all major browsers, leading to inconsistencies, as webpages relying on one of these tools might fail to render correctly or run efficiently in other browsers. Against this backdrop, WebAssembly was released in 2017 [44], with support from all four major browser vendors: Google (Chrome), Mozilla (Firefox), Apple (Safari), and Microsoft (Edge). Since its inception in 2017, WebAssembly has gained popularity beyond browser environments, expanding into use cases such as server-side applications.

WebAssembly is unique in that it was designed with a formal specification, which has been consistently maintained as the language has evolved over the years. The formal nature of WebAssembly's specification, coupled with its emergence as a next-generation web language, makes it a compelling and important target for research in programming languages and formal verification.

A critical aspect of formal verification in programming languages is the development of mechanised, machine-checked models. Traditional hand-written proofs are prone to human error and become increasingly tedious and difficult for third-party reviewers to verify manually, diminishing confidence in their correctness. Moreover, human-inspected certifications lack transferability, as they depend on the credibility and endorsement of individual reviewers.

The above challenge is particularly pronounced in programming language verification, where definitions are typically extensive and intricate. Proofs often involve exhaustive case analyses across numerous definition constructs—many of which may appear trivial—but a single overlooked mis-

take in a complex case can compromise the entire proof, undermining trust in the verification process.

Mechanised proofs address these challenges by reducing the trusted codebase to the specification tool's implementation itself. Additionally, the rigorous checking performed by mechanised tools ensures that even subtle errors in the language specification can be identified and corrected, further strengthening the reliability of formal verification. A well-maintained programming language mechanisation lays the foundations for further research on formal verification of the language itself.

However, language mechanisation is an onerous process, to the point that many mechanised semantics cover only a fragment of their language's full feature set. This challenge is particularly pronounced when the language continues to evolve, incorporating new features and extensions over time. In fact, many formalised programming language specifications are published once and never updated again. Given that WebAssembly is a relatively new and rapidly evolving language with frequent extensions to its specification, it is essential to explore methods that reduce the effort required to maintain and update its mechanisations.

This thesis provides a comprehensive discussion of various aspects of WebAssembly mechanisation, with a particular emphasis on these challenges, as outlined below:

- Design and implementation of a WebAssembly mechanisation in the Coq[1] [12] theorem prover;
- Methods for maintaining a theorem prover mechanisation of a language standard, particularly through the use of a dependently typed *progressful interpreter*;
- A specific application of the WebAssembly mechanisation – the development of the Iris-Wasm higher-order program logic.

1.1 Mechanisation of WebAssembly

In 2017, Watt implemented the first WebAssembly mechanisation [112] in the Isabelle/HOL theorem prover. This work implemented the entire core semantics of WebAssembly and the module system for a pre-1.0 draft version of the WebAssembly specification, with the corresponding type soundness proofs. It used external OCaml functions from Wasm's reference interpreter [39] to handle numerical operations and the decoding of Wasm's binary format. Through the mechanisation process of the specification, several errors in the targeted version of the Wasm specification were discovered and fixed [112], fulfilling the promise of mechanisation in detecting errors in an important non-mechanical specification.

It is highly beneficial to support multiple mechanisations of Wasm across various mainstream theorem provers. Theorem provers such as Coq, Isabelle, and Agda each have unique strengths

[1] The Coq theorem prover has announced a rename to Rocq in late 2024. In this thesis, the old name Coq is used throughout, as most work described in the thesis predates this change.

and are often tailored to specific conventional applications. Additionally, researchers from different fields may coalesce around key libraries associated with their preferred theorem provers. For instance, Isabelle is renowned for its powerful automation tactics, which reduce the need for low-level proof engineering in many cases. Agda, which can also be used as a programming language, excels in metaprogramming. Coq, being the least opinionated among these provers, features a vast community and an extensive set of libraries across diverse fields of application. For efforts related to WebAssembly mechanisation, Watt's Isabelle mechanisation leveraged the Isabelle-exclusive Sepref toolchain [61] to define a verified monadic interpreter for Wasm [115]. Similarly, Coq also offers several powerful libraries. For example, the Flocq floating-point library [19, 18] enables the CompCert verified C compiler [66, 67] to provide a set of verified numerics for Wasm numeric operations, thereby eliminating the need to rely on unverified OCaml implementations. Additionally, the Iris framework [53] in Coq serves as a versatile higher-order separation logic framework for designing program logics for programming languages.

WasmCert-Coq [114], a Coq-based mechanisation of WebAssembly, was designed and implemented in light of the above. It initially drew a part of its definitions from Watt's mechanisation of the pre-1.0 version in Isabelle/HOL. Both efforts were then unified under the WasmCert name. Subsequently, both mechanisations were updated to align with the W3C Wasm 1.0 specification and extended with additional soundness proofs, including the soundness of module instantiation and the correctness of the type checker.

The majority part of Chapter 3 describes the WasmCert-Coq mechanisation for Wasm 1.0, a small part of which originated from my Master project. A part of this chapter was briefly introduced – jointly with Watt's updated WasmCert-Isabelle mechanisation for Wasm 1.0 – in a previously published paper *Two Mechanisations of WebAssembly* (FM 2021) [114], authored by Conrad Watt, myself, Martin Bodin, Jean Pichon-Pharabod, and Philippa Gardner.

I have then updated the WasmCert-Coq mechanisation to the significantly larger Wasm 2.0 feature set using the design of the progressful interpreters from Chapter 4, which proved to be helpful in reducing the number of proof objects that need to be updated. In this process, I have discovered several mistakes in the Wasm 2.0 specification and reported them to the specification editor, some of which have been fixed as a consequence. Section 3.10 describes the update of WasmCert-Coq mechanisation to Wasm 2.0 using the design of the progressful interpreters from Chapter 4.

After my update to Wasm 2.0, I have also implemented the necessary infrastructure to run the official test suite on the extracted runtime from WasmCert-Coq. All tests were passed after fixing several mistakes in the mechanisation discovered in this process. Section 3.11 describes my work on WasmCert-Coq testing.

1.2 Dependently-typed Progressful Interpreters

During a discussion at the Verified Software workshop at the Isaac Newton Institute in 2022, Philip Wadler advertised the idea of generating a sound interpreter directly from a type soundness proof, building on some concepts from his earlier joint work in the Agda proof assistant [57].

This approach is particularly compelling as it offers the potential to merge three mechanisation artefacts – namely, the executable interpreter, the interpreter correctness proof, and the progress property – into a single piece of mechanised proof. Such unification could significantly reduce the maintenance effort required for updating proofs when the underlying mechanised definitions evolve.

I successfully implemented the aforementioned method in the WasmCert-Coq mechanisation, extracting an end-to-end executable interpreter from the type soundness proof – which was not produced in the original Agda-based work [57]. However, this method ultimately proved to be inadequate for practical use, as the interpreter generated directly from the existing type soundness proofs has significant performance limitations, which could not be improved sufficiently despite some optimisations that required awkward modifications to the proofs. Instead, I propose an alternative idea by designing a dependently-typed *progressful* interpreter. This interpreter encodes the progress property and the interpreter correctness property as part of its structure, which can be erased during extraction to achieve the same performance as the original interpreter. I also demonstrate the scalability of this approach by implementing an interpreter optimisation introduced in WasmRef-Isabelle [115]. Throughout this process, I was able to maintain the integrated soundness proofs without significantly increasing the size of the codebase, showcasing the efficiency and flexibility of the method.

Chapter 4 describes the work related to the type soundness interpreter, the progressful interpreter, and its optimisations. It draws from the previously accepted paper *Progressful Interpreters for Efficient WebAssembly Mechanisation* at POPL 2025 [92], authored by myself, Stefan Radziuk, Conrad Watt, and Philippa Gardner.

1.3 Modular Verification of WebAssembly Programs

Following Watt's initial mechanisation of WebAssembly in Isabelle, Watt et al devised a first-order encapsulated program logic of WebAssembly [113], which implemented a first-order subset of the Wasm core semantics, without higher-order dynamic function calls, the host-Wasm interoperation, or the module system and instantiation. However, this restriction heavily limits the scope of the programs that can be verified, since real-world Wasm programs are always instantiated by the host and interact with the host constantly.

Instead, the Coq-based mechanisation of WebAssembly allows the Iris separation logic framework to be used for constructing a higher-order program logic for WebAssembly. This approach captures the full feature set of the mechanised Wasm version, including the particularly intricate higher-order module instantiation operation.

Chapter 5 describes the higher-order program logic of the full Wasm 1.0 specification, including the module instantiation and an example host language that implements a minimal set of functions from the WebAssembly JavaScript Interface [28]. This work is designed using the Iris framework [53] in Coq, which is a higher-order separation logic framework used to design many program logics. This chapter draws text from a previously published paper *Iris-Wasm: Robust*

and Modular Verification of WebAssembly Programs (PLDI 2023) [91], authored by myself, Aïna Linn Georges[2], Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner and Lars Birkedal.

For all the works above, the contributions from my collaborators have been properly addressed in the acknowledgements section at the end of every chapter. In Chapter 4 and Chapter 5, I use the subject pronoun 'we' due to the collaboration involved in those works.

Unless stated otherwise, all of the works throughout this thesis have been fully mechanised in Coq. The actively-evolving WasmCert-Coq mechanisation is publicly available on GitHub [108].

[2] Aïna Linn Georges and myself contributed equally to this work.

Chapter 2

Background

This chapter describes the background of the Wasm specification, including an overview of its design and a description of its core semantics, focusing on the 1.0 version of the language. Nevertheless, as Wasm is an evolving language, I occasionally diverge from the old official Wasm 1.0 semantics and describe more forward-compatible design choices to avoid obsolete syntax and notations.

Section 2.1 and Section 2.2 provide an overview of the specification of WebAssembly and its runtime structure. The rest of the chapter, starting from Section 2.3, describes the details of various key parts of the semantics of the language, including the instruction set, type system, soundness, and the module system. Readers who are already knowledgeable about Wasm's semantics may skip these sections and continue reading from Chapter 3.

2.1 An Overview of the Wasm Specification

In this section, I present an overview of the Wasm specification by discussing several important concepts of the Wasm language and specification.

Scope of the Wasm Specification The core of Wasm is a virtual instruction set (virtual ISA) architecture which can be embedded in different environments, including but not limited to the major web browsers. The Wasm specification defines the abstract syntax tree (AST) of its instruction set, the validation process, and the execution semantics of Wasm programs. The specification does not define how Wasm programs are implemented by the hosts, nor how they are used by or interact with such hosts. Instead, each host can choose its own implementation of the Wasm runtime and design their interaction with Wasm by defining an application programming interface (API). One prominent example of such an API is the Wasm-JS API [28], which defines a JavaScript API that provides a set of interfaces for interacting with Wasm modules from a JavaScript program.

Model of Computation Wasm's execution is defined based on a stack machine model. The code of a Wasm function consists of a list of instructions, which are executed in sequence. The

instructions operate on a separate operand stack (also known as the *value* stack). Each instruction may consume a number of values from the operand stack and push a number of values back to it.

Four primitive value types **i32**/**i64**/**f32**/**f64** of variables are provided, corresponding to integers and IEEE-754 floats of sizes 32 or 64 bits. These are also the only available value types in Wasm 1.0.

Wasm uses structured control flow constructs including blocks, loops, and conditionals, where the branch instructions can only target such constructs. These control flow constructs are represented as instructions on the stack. Note that Wasm does not separately provide a boolean value type for conditional branches; instead, 32-bit integers are used for this purpose when required. Certain instructions may produce a *trap*, which represents an irrecoverable error within Wasm that needs to be handled by the host.

Wasm programs mainly use the *linear memory* for storage. A linear memory is a contiguous array of bytes, which is allowed to grow dynamically. Wasm programs can load values from or store values to a linear memory at any address. Besides the linear memory, another class of Wasm state for storage includes the *tables*, which are arrays of references that the Wasm programs can access dynamically by index during execution. In Wasm 1.0, a table can only hold untyped function references, which Wasm programs can invoke by using the corresponding instructions.

The Wasm specification defines the operational semantics of Wasm by a small-step semantics on Wasm *configurations*, which are syntactic descriptions of Wasm program states that include not only the current code in execution, but also information about the global and local states that have been allocated. I will discuss this in more detail in Section 2.2.

Most of Wasm's execution is deterministic, in that only one reduction rule is applicable to most program states. There are several exceptions, most notably when the host is involved (e.g. when additional memory is requested, but the host may not own sufficient resources), where the execution of an instruction can be non-deterministic.

Module System Wasm programs are distributed in *modules*, which are essentially large records that contain compact sets of definitions of Wasm states, including functions, tables, linear memories, and global variables. The modules interact with the host and potentially with other modules through an import-export system, where each module could specify a series of types of states that need to be provided to the module when the module is loaded (*imports*), and a series of states defined by the module that will be exported once the module loading is completed (*exports*). Each module can also optionally specify one function to be its start function, which is automatically invoked when the loading of the module is finished. The host is responsible for handling the loading – more formally known as the *instantiation* – of the modules, which includes handling how the states to be imported by the modules are provided, and how the exported states from the modules are stored and accessed by other runtime entities.

This design of the module interaction system is a key part of Wasm's safety guarantee of *module encapsulation*: Under the assumption that the host itself does not break this encapsulation property,

a Wasm module cannot accidentally access or modify the states of the other modules that have not been shared with the module.

Encodings of Wasm programs Besides the AST of Wasm semantics, the Wasm specification defines two additional encodings of Wasm programs. The *binary format* is a dense format for compact distribution of Wasm programs. The text format is readable and editable by humans and contains a lot of syntactic sugar to further increase the readability of Wasm programs.

Figure 2.1 displays the three different encodings of the same module implementing and exporting a function evaluating the nth Fibonacci number ($n \geq 1$). Throughout this thesis, the AST representation will be used for all formal definitions and discussions.

Semantic Phases The Wasm specification defines the processes for a host to handle its modules in three phases: decoding, validation, and execution.

Decoding is the process of converting a Wasm module from the dense binary format into an internal representation used by the implementation. In an industrial implementation, this internal representation could be direct machine code for maximum performance. However, in the Wasm specification, the reference interpreter, and various mechanisations, it takes the form of a human-readable abstract syntax.

Validation is the phase where a module and its definitions are checked against well-formedness conditions to ensure their safety. This process includes type-checking functions and their corresponding code within the module.

After validation, a module is *instantiated* by the host, where the specified imports are provided to the module, the states defined by the module are allocated by the host and initialised as specified by their initialisers, the start function of the module is invoked (if defined), and finally the exported instances of states are returned to the host, making them available for further use.

2.2 WebAssembly's Runtime Structure

The core of WebAssembly's runtime structure contains a small-step operational semantics over configurations *cfg*. A Wasm configuration is a tuple $(S; F; es)$ consisting of the store S, the current function call *frame* F, and the current instruction list es.

Store The *store* S collects the runtime representation, called *instances*, of all global states that have been created and can be accessed or modified by WebAssembly programs. The store is syntactically defined as a record containing a list of allocated instances for each category of Wasm states. This includes the following four categories:

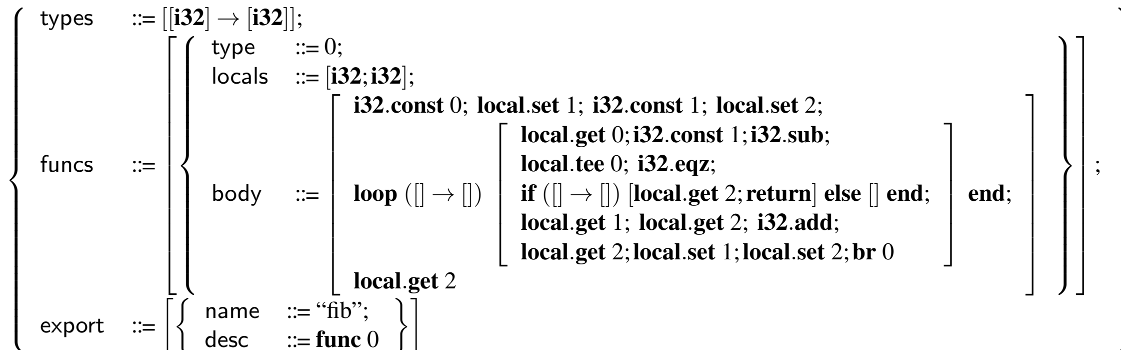
- A *function* instance of a Wasm function is essentially a *closure* over the original function definition, attaching to it with the module instance (introduced below in the *Frame* paragraph) of the module it was defined in and its *function type* signature, which specifies the types of the required arguments and results of the function.

```
(module
  (func $fib (export "fib") (param $n i32) (result i32)
    (local $x i32) (local $y i32)
    i32.const 0
    local.set $x
    i32.const 1
    local.set $y
    loop
        local.get $n
        i32.const 1
        i32.sub
        local.tee $n
        i32.eqz
        if
            local.get $y
            return
        end
        local.get $x
        local.get $y
        i32.add
        local.get $y
        local.set $x
        local.set $y
        br 0
    end
    local.get $y))
```

(a) Text format

```
00 61 73 6D 01 00 00 00 01 06 01 60 01 7F 01 7F
03 02 01 00 07 07 01 03 66 69 62 00 00 0A 2E 01
2C 01 02 7F 41 00 21 01 41 01 21 02 03 40 20 00
41 01 6B 22 00 45 04 40 20 02 0F 0B 20 01 20 02
6A 20 02 21 01 21 02 0C 00 0B 20 02 0B 00 1B 04
6E 61 6D 65 01 06 01 00 03 66 69 62 02 0C 01 00
03 00 01 6E 01 01 78 02 01 79
```

(b) Binary format (hexadecimal view)



(c) AST representation

Figure 2.1: Different encodings of a module implementing a Fibonacci function

Another source of function instances is the host[1]. As discussed in Section 2.1, the semantics of the host functions are out of the scope of Wasm semantics. Therefore, a host function instance only specifies its function type, and an opaque definition of a *host code*. The Wasm specification defines the behaviour of host function invocations to be non-deterministic up to certain constraints that preserve the well-formedness of the runtime structure, which will be discussed in Section 3.5.

- A *table* instance contains a list of table elements and an optional maximum size. In Wasm 1.0, the only type of table elements allowed is an option over the function address, which is a reference to a function in the store when defined, and interpreted as an uninitialised entry otherwise.

- A *memory* instance contains a list of bytes and an optional maximum size in pages (1 page = 2^{16} bytes). Wasm memories are always allocated by pages. It is therefore an invariant of the semantics that the length of the byte list of a memory is always a multiple of 2^{16}.

- A *global* variable instance (usually abbreviated as the global instance) contains a Wasm value and a mutability flag that indicates whether this global variable can be modified during execution.

Function frame and module instance The current function call *frame* F stores information specific to the current function invocation. In particular, it collects the values of the local variables and arguments of the current function call. More importantly, it holds a *module instance* of the module from which the function originates, which forms a substantial part of the call frame.

The module instance is the runtime representation of a module. It collects the runtime representations of the states in the Wasm store that can be accessed by the module. Syntactically, the module instance is a record containing a list of references to the above runtime representations for each of the four categories of Wasm states, a list of function types being used in the module, and a list of runtime representations of the module exports.

These simply refer to the corresponding locations in the store represented by addresses, where the actual runtime instances of the states are stored. The *export* instance similarly contains a reference to a state in the store, but is allowed to be a reference to any of the four categories of the Wasm states; it is also accompanied by an export *name*, which is a Unicode string used by the host to associate the corresponding store reference. For this reason, the store references are also collectively known as the *external values* in Wasm. This name is also used in the module import-export system.

Wasm code can only access the states in the store S via indirect references through the module instance. This design guarantees that the function will only have access to the same set of states accessible to the module where it is defined. This is important in providing the *module encapsulation* property for functions that are exported to the host.

[1] Also known as the embedder.

I display an illustration of the indirection process from the individual components of a module instance to the WebAssembly store in Figure 2.2.

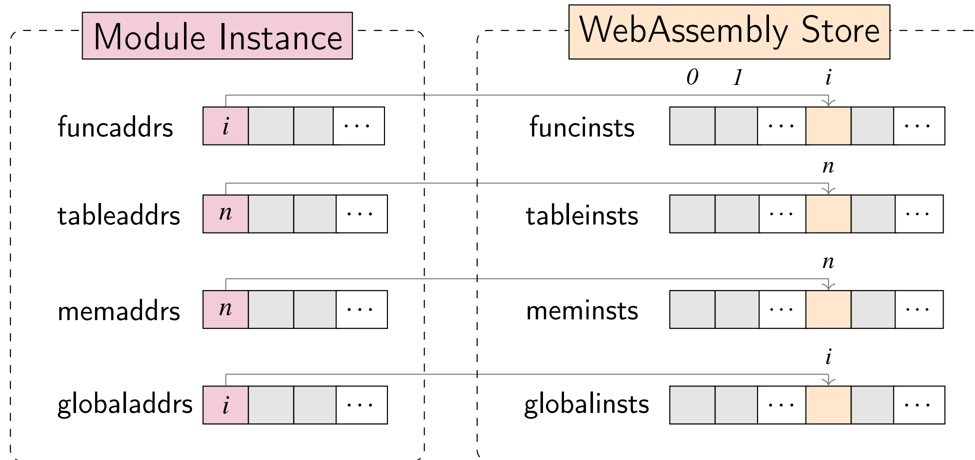
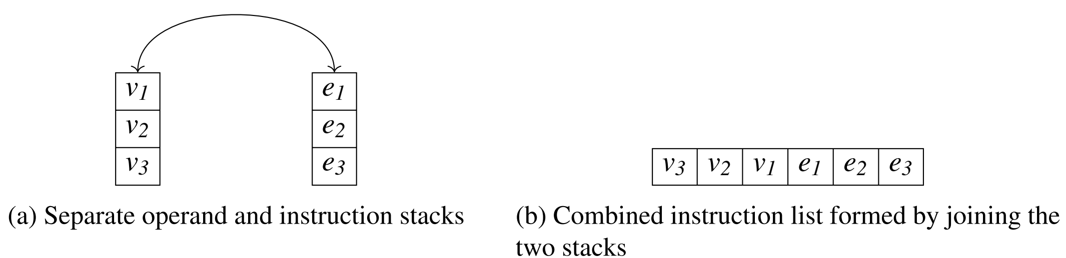


Figure 2.2: Indirections from module instance components to the Wasm store

It is noteworthy that the module instance was not part of Wasm's runtime configuration in early versions of the Wasm specifications [44], which only contained the local variable list but not the function frame. Instead, the module instance was a parameter of the reduction rules in the operational semantics. This is because Wasm's module instance is designed to be *immutable* throughout the execution of a function, as it serves only as an indirection to the corresponding runtime instances in the Wasm store. While these runtime instances in the store may be modified during execution, the addresses that can be accessed by the function do not. As a result, despite constituting a part of Wasm's runtime configuration, there is no instruction in Wasm that can modify the module instance.

Instruction list The instruction list *es* is the current function code which is being executed. Here, Wasm uses a convention where the instruction list *es* represents the combined operand and instruction stacks in the computation model. This is achieved by lifting the values to the corresponding **const** instructions, and joining the top of the two stacks together to form a list, with the operand stack preceding the instruction stack. Figure 2.3 presents an illustration of this convention.

Note that this is only a convention used by the Wasm specification to describe its semantics. Real-world efficient implementations do not need to adhere to this single-list representation, and are free to choose other equivalent representations for the operand and instruction stacks.



(a) Separate operand and instruction stacks

(b) Combined instruction list formed by joining the two stacks

Figure 2.3: Wasm's convention: the combined list representation of instructions

Execution takes place at the top of the instruction stack, interacting with the operand stack as specified by the operational semantics. For example, a reduction rule such as

$$[v_2; v_1; e_1] \hookrightarrow [v_1']$$

can be interpreted as: instruction e_1 executes by consuming v_1 and v_2 from the operand stack and pushing the value v_1' back to the operand stack.

In the formal syntax of the Wasm specification, the function call frame and the instruction list $(F; es)$ are together referred to as the current *thread*. Therefore, the configuration tuple can also be considered as the pair consisting of the store S and a thread. This structure makes it easier to extend Wasm's runtime semantics to allow multi-threaded execution in the future, where, for example, the configuration tuple could consist of a single store and a list of threads, each containing a function call frame and an instruction list. However, as of Wasm 2.0, Wasm still only supports single-threaded execution. As a result, the definition of the thread is currently only syntactic and has no semantic effect. Therefore, I usually omit the additional bracket over the thread tuple $(F; es)$ in the full configuration tuple $(S; F; es)$.

An overview of the structure of the configuration, with the definitions it depends on, is presented in Figure 2.4. Some identifiers or definitions in the figure are coloured gray; this indicates that their concrete definitions are left to be explained later.

Conventions In Figure 2.4, I adopt a number of conventions and notations, which will also be used for the rest of the thesis. Most of these conventions were originally used by the Wasm specification and adopted in the thesis.

- Xs (and sometimes X^*) represents a list of Xs[2];
- X^n represents a list of n Xs;
- $X^?$ represents an optional value X. When an empty (None) value of an option type needs to be specified explicitly (for example, when defining a partial function), the notation ε is used;
- $\{\mathsf{field1} :: val, \mathsf{field2} :: \mathsf{type}, \ldots\}$ represents a record definition with fields field1, field2, etc. In a record definition, there are two ways to specify the expected member of a field, as demonstrated in the example above: field1 contains values of the form *val*, which will be binders introduced earlier; field2 contains values of the type type, representing previously introduced types.

  When unambiguous, record instances are often presented as tuples consisting of the record's field values separated by ;. For example, the Wasm configurations are usually displayed as $(S; F; es)$.

[2] The Wasm specification uses the convention X^* only. However, the symbol $*$ can cause ambiguity in some contexts – e.g. with separating conjunction in separation logic used in Chapter 5. On the other hand, the suffix s is sometimes awkward when following certain binders or variable names. In this thesis, Xs is the default notation, and X^* is used when the default notation would cause awkwardness.

- The notations sN, uN, and iN represent the signed/unsigned/uninterpreted integers of bit-length N. This is also used in places where integer indices are required. For example, $u32$ values used as list indices in the current version of Wasm specification.

- The notation $f32/f64$ represents the type of floating-point values of 32/64-bit, formatted according to the corresponding binary format of floating points in the IEEE754 standard [47].

- $[e_1; e_2; e_3; \ldots]$ represents a list containing the elements $e_1, e_2, e_3, \ldots$;

- X with $\mathsf{field} = val$ represents a record update where the field field is updated to value val, with the other parts of the store remaining the same. This is frequently used in the description of the reduction rules for instructions which modify a part of the store S or other record structures in Wasm.

  This syntax is sometimes generalised where field is not only a field of the record X, but instead a *path*, which is a series of dereferences that ultimately refers to an entry of value contained deeply in the record. In such cases, this syntax refers to the update result where that entry is replaced with the new value val.

- $L[offset : len]$ represents the sublist starting from index $offset$ (0-indexed) of length len from the list L. This is used in the semantics of several memory instructions operating on a contiguous segment of bytes at once.

- Many types in Wasm are defined as (possibly non-recursive) inductive types with several constructors. In these cases, the constructor names will be presented in **bold**.

$$
\begin{array}{rlrl}
\text{(value type) } t & ::= \mathbf{i32} \mid \mathbf{i64} \mid \mathbf{f32} \mid \mathbf{f64} & \text{(function type) } ft & ::= ts \to ts \\
\text{(value) } v & ::= \mathbf{iN.const}\ uN \mid \mathbf{fN.const}\ fN & \text{(result type) } rt & ::= ts \\
\text{(instruction) } e & ::= \ldots & \text{(immediate) } i, n, min, max & ::= u32
\end{array}
$$

$$
\begin{array}{l}
\text{(addresses and indices) } addr, id ::= n \\
\text{(component addresses) } \{func/table/mem/global\}addr ::= addr \\
\text{(component indices) } \{func/table/mem/global\}id ::= id
\end{array}
$$

$$
\begin{array}{rl}
\text{(external value)} & externval ::= \mathbf{func}\ funcaddr \mid \mathbf{table}\ tableaddr \mid \mathbf{mem}\ memaddr \mid \mathbf{global}\ globaladdr \\
\text{(export instance)} & exportinst ::= \{\mathsf{name} :: chars, \mathsf{value} :: externval\} \\
\text{(module instance)} & moduleinst ::= \\
& \{\mathsf{types} :: fts, \mathsf{funcaddrs} :: funcaddrs, \mathsf{tableaddrs} :: tableaddrs, \\
& \mathsf{memaddrs} :: memaddrs, \mathsf{globaladdrs} :: globaladdrs, \mathsf{exports} :: exportinsts\}
\end{array}
$$

$$
\begin{array}{rl}
\text{(functions)} & func ::= \{\mathsf{type} :: typeid, \mathsf{locals} :: ts, \mathsf{body} : es\} \\
\text{(function instance)} & funcinst ::= \{\mathsf{type} :: ft, \mathsf{module} :: moduleinst, \mathsf{code} :: func\}^{\mathsf{Native}} \mid \\
& \qquad \{\mathsf{type} :: ft, \mathsf{hostcode} :: \ldots\}^{\mathsf{Host}} \\
\text{(table instance)} & tableinst ::= \{\mathsf{elem} :: (funcaddr^?)s, \mathsf{max} :: u32\} \\
\text{(memory instance)} & meminst ::= \{\mathsf{data} :: (byte)s, \mathsf{max} :: u32\} \\
\text{(global mutability)} & mut ::= \mathbf{const} \mid \mathbf{var} \\
\text{(global instance)} & globalinst ::= \{\mathsf{value} :: v, \mathsf{mut} :: mut\}
\end{array}
$$

$$
\begin{array}{rl}
\text{(frame)} & F ::= \{\mathsf{locals} :: vs, \mathsf{module} :: moduleinst\} \\
\text{(store)} & S ::= \{\mathsf{funcs} : funcinsts, \mathsf{tables} : tableinsts, \mathsf{mems} : meminsts, \mathsf{globals} : globalinsts\} \\
\text{(configuration tuple)} & \mathsf{cfg} ::= \{\mathsf{store} :: S, \mathsf{frame} :: F, \mathsf{instructions} :: es\}
\end{array}
$$

Figure 2.4: Structure of the Wasm configuration tuple

2.3 The Wasm 1.0 Instruction Set and Runtime Semantics

The Wasm 1.0 operational semantics has been thoroughly described in the Wasm specification [93]. The aim of this section is to describe the structure of the AST of Wasm 1.0 instruction set and introduce its design and corresponding execution semantics.

I first present an overview of the syntax of the instruction set in Figure 2.5, which is categorised in the same groups as given by the Wasm specification.

$$
\begin{array}{rl}
\text{(instructions) } e ::= & \\
\text{(values)} & v \mid \\
\text{(numeric instructions)} & t.\mathbf{unop}\ \mathit{unop} \mid t.\mathbf{binop}\ \mathit{binop} \mid \\
& t.\mathbf{testop}\ \mathit{testop} \mid t.\mathbf{relop}\ \mathit{relop} \mid t_2.\mathbf{cvtop}\ \mathit{cvtop}\ t_1\ \mathit{sx}^? \mid \\
\text{(parametric instructions)} & \mathbf{drop} \mid \mathbf{select} \mid \\
\text{(variable instructions)} & \mathbf{local}.\{\mathbf{get}/\mathbf{set}/\mathbf{tee}\}\ i \mid \mathbf{global}.\{\mathbf{get}/\mathbf{set}\}\ i \\
\text{(memory instructions)} & \mathbf{iN}.\mathbf{load}\ (\mathit{sz}, \mathit{sx})^?\ \mathit{memarg} \mid \mathbf{iN}.\mathbf{store}\ \mathit{sz}^?\ \mathit{memarg} \mid \\
& \mathbf{fN}.\{\mathbf{load}/\mathbf{store}\}\ \mathit{memarg} \mid \\
& \mathbf{memory}.\mathbf{size} \mid \mathbf{memory}.\mathbf{grow} \mid \\
\text{(control instructions)} & \mathbf{nop} \mid \mathbf{unreachable} \mid \\
& \mathbf{block}\ \mathit{ft}\ \mathit{es}\ \mathbf{end} \mid \mathbf{loop}\ \mathit{ft}\ \mathit{es}\ \mathbf{end} \mid \mathbf{if}\ \mathit{ft}\ \mathit{es}_1\ \mathbf{else}\ \mathit{es}_2\ \mathbf{end} \mid \\
& \mathbf{br}\ i \mid \mathbf{br_if}\ i \mid \mathbf{br_table}\ \mathit{is}\ i \mid \mathbf{return} \mid \\
& \mathbf{call}\ i \mid \mathbf{call_indirect}\ i \mid \\
\text{(administrative instructions)} & \mathbf{trap} \mid \mathbf{label}_n\{\mathit{es}_{\mathsf{cont}}\}\ \mathit{es}\ \mathbf{end} \mid \mathbf{invoke}\ \mathit{funcaddr} \mid \mathbf{frame}_n\{F\}\ \mathit{es}\ \mathbf{end}
\end{array}
$$

Figure 2.5: Core instruction syntax of Wasm 1.0

Administrative Instructions The *administrative* instructions form a special category of instructions in Wasm. They do not appear in the code of any Wasm module functions. Instead, they are auxiliary instructions defined in the specification to facilitate the reduction of various instructions in the other categories. Therefore, I introduce the respective administrative instructions along with the execution of the corresponding instruction categories. The only exception is the **trap** instruction, which I have already described earlier in Wasm's model of computation (Section 2.1): it represents an error that is irrecoverable within Wasm which should be handled by the host.

In the rest of this thesis, the set of the non-administrative instructions is referenced as the *basic* instructions.

2.3.1 Numeric Instructions

The numeric instructions provide basic operations for numeric values. The Wasm specification defines their syntax using *generic* numeric operators $\{\mathit{unop}/\mathit{binop}/\mathit{testop}/\mathit{cvtop}/\mathit{relop}\}$, representing the categories of unary operators, binary operators, test operators, convert operators, and relation operators. This choice of writing the definition avoids the repetition in the operational semantics (and later, in typing rules). For example, instead of explicitly stating one reduction rule for each unary operator, Wasm uses one single rule that defines the result of execution of a unary operator $t.\mathit{unop}$ to be an opaque mathematical function $\mathit{unop}_t(v)$ applied to the operand v from the stack. This allows the rules to be presented with better clarity, and also allows readers who are

not interested in the specifics of the numeric operators to skip the concrete specifications of the individual operators.

The top-level reduction rules for the numeric operations are therefore simple: five sets of rules are defined, with one or two for each set of numeric operators depending on whether the underlying operators are total or partial. In the cases where the partial operators are undefined for the operands on the stack, a **trap** is produced. These rules are displayed in Figure 2.6[3].

$$
\begin{array}{ll}
\text{(instructions) } e ::= & \\
 & \dots \mid \\
\text{(numeric instructions)} & t.\textbf{unop}\ \mathit{unop} \mid t.\textbf{binop}\ \mathit{binop} \mid \\
 & t.\textbf{testop}\ \mathit{testop} \mid t.\textbf{relop}\ \mathit{relop} \mid t_2.\textbf{cvtop}\ \mathit{cvtop}\ t_1\ \mathit{sx}^?
\end{array}
$$

$[t.\textbf{const}\ c;\ t.\textbf{unop}\ \mathit{unop}] \hookrightarrow [t.\textbf{const}\ c']$	$\mathit{unop}_t(c) = c'$
$[t.\textbf{const}\ c;\ t.\textbf{unop}\ \mathit{unop}] \hookrightarrow [\textbf{trap}]$	$\mathit{unop}_t(c) = \bot$
$[t.\textbf{const}\ c_1; t.\textbf{const}\ c_2;\ t.\textbf{binop}\ \mathit{binop}] \hookrightarrow [t.\textbf{const}\ c']$	$\mathit{binop}_t(c_1, c_2) = c'$
$[t.\textbf{const}\ c_1; t.\textbf{const}\ c_2;\ t.\textbf{binop}\ \mathit{binop}] \hookrightarrow [\textbf{trap}]$	$\mathit{binop}_t(c_1, c_2) = \bot$
$[t.\textbf{const}\ c;\ t.\textbf{testop}\ \mathit{testop}] \hookrightarrow [\textbf{i32.const}\ c']$	$\mathit{testop}_t(c) = c'$
$[t.\textbf{const}\ c_1; t.\textbf{const}\ c_2;\ t.\textbf{relop}\ \mathit{relop}] \hookrightarrow [\textbf{i32.const}\ c']$	$\mathit{relop}_t(c_1, c_2) = c'$
$[t_1.\textbf{const}\ c;\ t_2.\textbf{cvtop}\ \mathit{cvtop}\ t_1\ \mathit{sx}^?] \hookrightarrow [t_2.\textbf{const}\ c']$	$\mathit{cvtop}^{\mathit{sx}^?}_{t_1,t_2}(c) = c'$
$[t_1.\textbf{const}\ c;\ t_2.\textbf{cvtop}\ \mathit{cvtop}\ t_1\ \mathit{sx}^?] \hookrightarrow [\textbf{trap}]$	$\mathit{cvtop}^{\mathit{sx}^?}_{t_1,t_2}(c) = \bot$

Figure 2.6: Top-level abstract reduction rules for Wasm numeric instructions

Note that the reduction rules do not specify the outcome when the types or the number of values at the top of the operand stack does not match the expectation of the instructions. Instead, the execution would be *stuck* if no reduction rules can be applied. However, Wasm's type system and module *validation* ensure that no such stuck state will occur during execution, which is discussed later in Section 2.4 and Section 2.7.1.

The full set of numeric operators available in Wasm 1.0 is listed in Appendix A.

2.3.2 Parametric Instructions

The set of parametric instructions contains only two instructions: **drop** and **select**. In Wasm 1.0, they are stack-polymorphic, in the sense that they accept operands of any value type from the operand stack. Correspondingly, there is no type annotation along with these instructions.

The **drop** instruction simply pops the top value from the operand stack, while the **select** instruction retrieves one **i32** integer from the operand stack followed by the next two values from the operand stack, and uses the top **i32** integer as a boolean to decide which of the two values to preserve, which is then pushed back to the operand stack.

The operational semantics of the parametric instructions are described in Figure 2.7.

[3]Note that the rules displayed in the figure do not mention the store S or the frame F. I adopt this convention by the Wasm specification, which omits the component of the configuration tuple that is irrelevant in the rule when presenting a rule. Formally, the rules should be interpreted to be universally quantified over all stores S and frames F which remain invariant during execution.

(instructions) e ::=	
	... |
(parametric instructions)	**drop** | **select**

$[v;\ \textbf{drop}] \hookrightarrow []$	
$[v_1; v_2; \textbf{i32.const}\ c; \textbf{select}] \hookrightarrow [v_1]$	$c \neq 0$
$[v_1; v_2; \textbf{i32.const}\ c; \textbf{select}] \hookrightarrow [v_2]$	$c = 0$

Figure 2.7: Reduction rules for parametric instructions

2.3.3 Variable Instructions

The variable instructions in Wasm allow access and modification of the local and global variables.

The local variables are contained in the frame at F.**locals**, and the global variables are stored in the store at S.**globals**. Each of the variable instructions carries an argument of index i, specifying the index of the local/global variable it is trying to access. The **get** operations retrieve the corresponding value and push it to the operand stack, while the **set** operations consume a value from the operand stack and replace the variable at the specified index by the value. The **local**.**tee** operation is a special operation that performs a **set** operation without consuming the top value on the operand stack (essentially pushing a copy of the value to the stack first before proceeding to **set**).

The global variables are accessed through an indirect reference through the module instance F.module – specifically, the list of global addresses of the module instance, F.module.globaladdrs. As described in Section 2.2, this design ensures that a module function cannot access any global variable (and in general, any Wasm states) that are not accessible by the module itself.

Note that the reduction rules do not specify an execution path when the index specified by the instruction is out of bounds. Similar to the case of numeric instructions, the validation algorithm ensures that these situations do not occur by checking the static indices against the bounds of the corresponding lists before execution.

The operational semantics of the variable instructions are described in Figure 2.8.

(instructions) e ::=	
	... |
(variable instructions)	**local**.{**get**/**set**/**tee**} i | **global**.{**get**/**set**} i

$(F; [\textbf{local.get}\ i]) \hookrightarrow (F; [v])$	$F.\mathsf{locals}[i] = v$
$(F; [v;\ \textbf{local.set}\ i]) \hookrightarrow (F'; [])$	$F' = F$ with $\mathsf{locals}[i] = v$
$[v;\ \textbf{local.tee}\ i] \hookrightarrow [v;\ v;\ \textbf{local.set}\ i]$	
$(S; F; [\textbf{global.get}\ i]) \hookrightarrow (S; F; [v])$	$S.\mathsf{globals}[F.\mathsf{module.globaladdrs}[i]].\mathsf{value} = v$
$(S; F; [v;\ \textbf{global.set}\ i]) \hookrightarrow (S'; F; [])$	$S' = S$ with $\mathsf{globals}[F.\mathsf{module.globaladdrs}[i]].\mathsf{value} = v$

Figure 2.8: Reduction rules for variable instructions

2.3.4 Memory Instructions

Wasm 1.0's memory instructions consist of the **load** and **store** instructions, which provide read-/write operations from/to Wasm's linear memory. Besides these, Wasm 1.0 also provides two separate operations **memory**.{**grow**/**size**} which respectively grow a memory and retrieve the size of a memory.

Although Wasm's global store S can contain a general list of memories, Wasm 1.0 artificially limits the number of memories allowed per module to at most 1. Therefore, in contrast to the global variable instructions, none of the memory instructions carry the index of the memory being accessed, as there is at most only one allowed for every module; and all memory instructions by default interact with the only linear memory accessible to the module they live in.

Each of the **load**/**store** instructions carries the following arguments and a type annotation:

- The memory argument *memarg*, which contains the static *offset* that will be added to the dynamic address retrieved from the operand stack to form the starting address of the memory instruction's target; and the *alignment*, which is a number such that 2^{align} divides the target address being accessed. This is only used for optimisation purposes and does not affect the operational semantics.
- The type annotation t, which specifies the value type to be stored into the memory for **store** or the value type to be loaded from the memory for **load**. When no additional argument is specified, the type annotation also determines the number of bytes starting from the effective address that will be the *target segment* of the memory instruction. In particular, $\frac{|t|}{8}$ consecutive bytes starting from the starting memory address will be the target segment, where $|t|$ stands for the bit-width of the value type t (32 for **i32**/**f32** and 64 for **i64**/**f64**). If this segment falls out of bounds of the current memory, a **trap** is produced.
- For memory instructions on integer types, an optional storage size argument sz can be provided, which has to take one value among 8/16/32 and be smaller than the bit-width of the type t. When the storage size is specified, the memory instruction only targets the number of bytes specified by sz from the starting address on the memory, and extend or wrap the value to be loaded/stored accordingly. When a storage size is specified for a **load** instruction, a further *sign extension* flag sx needs to be provided, specifying how the extension from bit-size sz to $|t|$ is handled.

If the above arguments successfully determine a valid target segment, the t.**load** instruction reads the list of bytes from the target segment of the memory and converts it to a value of the annotated type t. The t.**store** instruction instead retrieves a value of type t on the operand stack, converts it to a list of bytes of the specified length, and stores it in the target segment of the memory. The Wasm specification defines a family of operators $\mathsf{bytes}_t(-)$ for the bytes-representation of values of type t in the little-endian order, which is a bijection for each value type t. I omit the details of it here and will come back to it in Chapter 3.

The instruction **memory**.**size** gets the size of the current Wasm memory as a **i32** value and pushes

the result to the stack.

Finally, the **memory**.**grow** instruction expands the module's memory by a number of *pages* (2^{16} bytes) specified by the top **i32** value on the operand stack and returns the *original* size of the memory (in pages) as an **i32** value.

The expansion of the memory instance is described by the partial function $\mathsf{growmem}(meminst, n)$:

$$\mathsf{growmem}(meminst, n) ::= \begin{cases} meminst \text{ with } \mathsf{data} = meminst.\mathsf{data} +\!\!+ \ (\mathsf{0x00})^{n \cdot 2^{16}} \\ \left(\begin{array}{r} len = n + \frac{|meminst.\mathsf{data}|}{2^{16}} \wedge \\ len \leq 2^{16} \wedge \\ (meminst.\mathsf{max} = \varepsilon) \vee (len \leq meminst.\mathsf{max}) \end{array} \right) \\ \varepsilon \, (otherwise) \end{cases}$$

which appends $n \cdot 2^{16}$ byte-zeros $0x00$ to the content of the linear memory $meminst.\mathsf{data}$, provided that the resulting size of the memory does not exceed the maximum size specified in the memory instance when such a maximum is specified.

However, **memory**.**grow** is allowed to fail even when the resulting size does not exceed the specified maximum size of the memory. This can be, for example, due to the host's limit on the resources available to Wasm. No **trap** is produced in this case; instead, the instruction simply returns an **i32**.**const** (-1), which can be captured and handled further within Wasm. Since this kind of failure is dependent on the host, Wasm models this possibility by allowing **memory**.**grow** to fail *non-deterministically*. This is one of the few exceptions where the reduction rule of a Wasm instruction is not deterministic.

The operational semantics of the memory instructions are described in Figure 2.9.

2.3.5 Control Flow Instructions

Wasm's control flow instructions can be further introduced in several sub-categories.

Simple Control Flow Instructions

This category contains the instructions **nop**, which does nothing; and the instruction **unreachable**, which produces a **trap**. Their semantics are displayed in Figure 2.10.

Structured Block Instructions

This sub-category contains the three instructions for block constructs

$$\textbf{block}\, \mathit{ft}\ \mathit{es}\ \textbf{end} \mid \textbf{loop}\, \mathit{ft}\ \mathit{es}\ \textbf{end} \mid \textbf{if}\, \mathit{ft}\ \mathit{es}_1\ \textbf{else}\ \mathit{es}_2\ \textbf{end}$$

and three variants of branch instructions **br** i | **br_if** i | **br_table** is i.

WebAssembly features structured control flows. The three block constructs **block**, **loop**, and **if** all take as arguments a function type, and an instruction list representing the body of the **block**

$sz ::= 8 \mid 16 \mid 32$
$sx ::= \mathsf{unsigned} \mid \mathsf{signed}$
$memarg ::= \{offset : u32, align : u32\}$
(instructions) $e ::=$

$\ldots \mid$
(memory instructions) $iN.\mathbf{load}\ (sz, sx)^{?}\ memarg \mid iN.\mathbf{store}\ sz^{?}\ memarg \mid$
$fN.\{\mathbf{load}/\mathbf{store}\}\ memarg \mid$
$\mathbf{memory}.\mathbf{size} \mid \mathbf{memory}.\mathbf{grow} \mid$

$$(S;F;[\mathbf{i32}.\mathbf{const}\ i; t.\mathbf{load}\ memarg]) \hookrightarrow (S;F;[t.\mathbf{const}\ c])$$
$$ea = i + memarg.\mathsf{offset} \wedge$$
$$a_{mem} = F.\mathsf{module.memaddrs}[0] \wedge$$
$$ea + \frac{|t|}{8} \leq |S.\mathsf{mems}[a_{mem}].\mathsf{data}| \wedge$$
$$\mathsf{bytes}_t(c) = S.\mathsf{mems}[a_{mem}].\mathsf{data}[ea : \frac{|t|}{8}]$$
$$(S;F;[\mathbf{i32}.\mathbf{const}\ i; t.\mathbf{load}\ (sz, sx)\ memarg]) \hookrightarrow (S;F;[t.\mathbf{const}\ c])$$
$$ea = i + memarg.\mathsf{offset} \wedge$$
$$a_{mem} = F.\mathsf{module.memaddrs}[0] \wedge$$
$$ea + \frac{sz}{8} \leq |S.\mathsf{mems}[a_{mem}].\mathsf{data}| \wedge$$
$$\mathsf{bytes}_{i_sz}(c') = S.\mathsf{mems}[a_{mem}].\mathsf{data}[ea : \frac{sz}{8}] \wedge$$
$$\mathsf{extend_}sx_{sz,|t|}(c') = c$$
$$(S;F;[\mathbf{i32}.\mathbf{const}\ i; t.\mathbf{load}\ (sz, sx)^{?}\ memarg]) \hookrightarrow (S;F;[\mathbf{trap}]) \quad \text{otherwise}$$

$$(S;F;[\mathbf{i32}.\mathbf{const}\ i; t.\mathbf{const}\ c; t.\mathbf{store}\ memarg]) \hookrightarrow (S;F;[])$$
$$ea = i + memarg.\mathsf{offset} \wedge$$
$$a_{mem} = F.\mathsf{module.memaddrs}[0] \wedge$$
$$ea + \frac{|t|}{8} \leq |S.\mathsf{mems}[a_{mem}].\mathsf{data}| \wedge$$
$$S' = S \text{ with } \mathsf{mems}[a_{mem}].\mathsf{data}[ea : \frac{|t|}{8}] = \mathsf{bytes}_t(c)$$
$$(S;F;[\mathbf{i32}.\mathbf{const}\ i; t.\mathbf{const}\ c; t.\mathbf{store}\ sz\ memarg]) \hookrightarrow (S;F;[])$$
$$ea = i + memarg.\mathsf{offset} \wedge$$
$$a_{mem} = F.\mathsf{module.memaddrs}[0] \wedge$$
$$ea + \frac{sz}{8} \leq |S.\mathsf{mems}[a_{mem}].\mathsf{data}| \wedge$$
$$S' = S \text{ with } \mathsf{mems}[a_{mem}].\mathsf{data}[ea : \frac{sz}{8}] = \mathsf{bytes}_{i_{sz}}(\mathsf{wrap}_{|t|,sz}(c))$$
$$(S;F;[\mathbf{i32}.\mathbf{const}\ i; t.\mathbf{const}\ c; t.\mathbf{store}\ sz^{?}\ memarg]) \hookrightarrow (S;F;[\mathbf{trap}]) \quad \text{otherwise}$$

$$(S;F;[\mathbf{memory}.\mathbf{size}]) \hookrightarrow (S;F;[\mathbf{i32}.\mathbf{const}\ sz])$$
$$|S.\mathsf{mems}[F.\mathsf{module.memaddrs}[0]].\mathsf{data}| = sz \cdot 2^{16}$$

$$(S;F;[\mathbf{i32}.\mathbf{const}\ n; \mathbf{memory}.\mathbf{grow}]) \hookrightarrow (S';F;[\mathbf{i32}.\mathbf{const}\ sz])$$
$$a_{mem} = F.\mathsf{module.memaddrs}[0] \wedge$$
$$|S.\mathsf{mems}[a_{mem}].\mathsf{data}| = sz \cdot 2^{16} \wedge$$
$$S' = S \text{ with } \mathsf{mems}[a_{mem}] = \mathsf{growmem}(S.\mathsf{mems}[a_{mem}], n)$$
$$(S;F;[\mathbf{i32}.\mathbf{const}\ n; \mathbf{memory}.\mathbf{grow}]) \hookrightarrow (S;F;[\mathbf{i32}.\mathbf{const}\ (-1)]) \quad \text{(non-det)}$$

Figure 2.9: Reduction rules for memory instructions

or **loop**, and two instruction lists in the case of **if**. Execution of a block construct proceeds by executing its body (in the case of **if**, the corresponding body determined by the truthiness of the top value on the operand stack) until one of the following is true:

- Only a list of values is left in the body, representing that the instruction list is empty and the operand stack contains values to be returned, in which case the block construct is exited and

(instructions) $e ::= \ \dots \mid \textbf{nop} \mid \textbf{unreachable}$

$[\textbf{nop}] \hookrightarrow []$

$[\textbf{unreachable}] \hookrightarrow [\textbf{trap}]$

Figure 2.10: Reduction rules for control instructions (simple)

the values are returned;

- A branch instruction is encountered, in which case a number of block constructs will be exited, and execution continues depending on the type of the next immediate enclosing block.

Each block construct also comes with a type annotation, which is a function type *ft*, indicating the types of values required as arguments to enter the block construct and the types of values produced at the end of executing the block construct[4]. Because of the similarity between these block constructs, the WebAssembly specification defines a **label** administrative instruction to handle the execution of the block constructs.

$\textbf{label}_n\{es_{\mathsf{cont}}\}\ es_{\mathsf{body}}\ \textbf{end}$ is a label with body es_{body}, continuation es_{cont}, and arity n. I explain these three components individually:

- The body of a **label** is where execution takes place. Each step of reduction of the body is lifted to a reduction of the **label** construct. In other words, the **label** construct

$$\textbf{label}_n\{es_{\mathsf{cont}}\}(_)$$

  behaves like an evaluation context with the hole at its body. When there is only a list of values left in the body, the label is exited.

- The continuation of a label is the code that will be executed when the label is selected as the target of a *branch* instruction within its body. As of Wasm 2.0, only the **loop** construct results in a label with a non-empty continuation, which is identical to the loop body itself.

- The arity of the label indicates the number of values to be provided to the label when it is selected as the target of a branch instruction, which is determined by the function type of the original block.

The **block** and **loop** instructions both reduce to a **label** instruction:

- **loop** reduces to a label with the loop body as its body, the continuation being the entire loop itself, and an arity equal to the number of values *consumed* by the type annotation of the loop;

- **block** reduces to a label with empty continuation and label body equal to the block body.

[4] In fact, the generalisation of block annotations to arbitrary function types was only added in Wasm 2.0. In Wasm 1.0, only a *result type* is allowed. As a result, at most 1 value is produced, and no arguments can be taken by the block constructs.

The arity is equal to the number of values *produced* by the type annotation of the block;

The **if** instruction reduces similarly to a **block** instruction with its body being one of the conditional bodies of **if**, depending on the top value on the operand stack.

Note that despite the name of the instruction **loop**, there is no mechanism in the resulting **label** that automatically loops over the label body. Instead, iterating over loops is handled by the branch instructions, which more generally provide structured control flow operations. When a branch instruction is executed, it selects the target label by its argument, which is a 0-indexed label index starting from the innermost **label** that encloses the branch instruction. Branching to a label retrieves a number of values specified by the arity of the target label from the top of the operand stack (of the innermost label whose body contains the branch instruction), exits from all the enclosing labels up until and including the target label, and executes the continuation of the target label. As a consequence, the canonical method of iterating over a loop is to include a **br** instruction of the appropriate index (for example, **br** 0 at the end of the loop body of the Fibonacci function in the example module introduced in Figure 2.1) inside the loop body, which targets the loop itself.

Three variations of the branch instructions are provided in Wasm:

- **br** k, which targets the kth-indexed label;
- **br_if** k, which is a conditional branch to the kth label if the top value on the operand stack is non-zero, and does nothing otherwise;
- **br_table** $ls\ l_N$, which retrieves the top value on the operand stack and targets the corresponding index in the list of label indices ls; if the top value is out of bounds of the length of the label indices provided, then the default label index l_N is selected as the branch target.

To formally define the reduction of the branch instructions within nested label blocks, Wasm defines the *block context*[5] – dependently-typed by its depth – as follows:

$$\begin{aligned} B^0[_] &= vs +\!\!+ [_] +\!\!+ es \\ B^{k+1}[_] &= vs +\!\!+ \textbf{label}_n\ \{es_{\mathsf{cont}}\}\ B^k[_]\ \textbf{end}\ +\!\!+ es \end{aligned}$$

The operational semantics of **br** can then be expressed by using a block context of the corresponding depth.

I display the operational semantics of the block constructs, branch instructions, and labels in Figure 2.11. Note that the last evaluation-context rule for $B^k[es]$ describes more than the nested labels. Since the rule also applies when $k = 0$, it also encodes a frame-like rule that allows us to concentrate on executing the top instruction of the instruction stack while leaving the unused operands and instructions in the frame.

Function Call Instructions

The last category contains two instructions for function calls **call** i | **call_indirect** i, and the **return** instruction that returns from a function call.

[5] Also known as the *label context* in some versions of Wasm specifications.

(instructions) e ::=

(control instructions) ... | **block** *ft es* **end** | **loop** *ft es* **end** | **if** *ft* es_1 **else** es_2 **end** | **br** *i* | **br_if** *i* | **br_table** *is i*

(administrative instructions) ... | $\textbf{label}_n\{es_{\mathsf{cont}}\}\ es_{\mathsf{body}}$ **end**

$(F; v^m \mathbin{++} [\textbf{block}\ (ts_1 \to ts_2)\ es\ \textbf{end}]) \hookrightarrow$ $(F; [\textbf{label}_n\{[]\}\ (v^m \mathbin{++}\ es)\ \textbf{end}])$	$\lvert ts_1 \rvert = m \wedge \lvert ts_2 \rvert = n$
$(F; v^m \mathbin{++} [\textbf{loop}\ (ts_1 \to ts_2)\ es\ \textbf{end}]) \hookrightarrow$ $(F; [\textbf{label}_m\{[\textbf{loop}\ (ts_1 \to ts_2)\ es\ \textbf{end}]\}\ (v^m \mathbin{++}\ es)\ \textbf{end}])$	$\lvert ts_1 \rvert = m$
$[\textbf{i32}.\textbf{const}\ c; \textbf{if}\ ft\ es_1\ es_2\ \textbf{end}] \hookrightarrow [\textbf{block}\ ft\ es_1\ \textbf{end}]$ $[\textbf{i32}.\textbf{const}\ c; \textbf{if}\ ft\ es_1\ es_2\ \textbf{end}] \hookrightarrow [\textbf{block}\ ft\ es_2\ \textbf{end}]$	$c \neq 0$ $c = 0$
$[\textbf{label}_n\{es_{\mathsf{cont}}\}\ B^k[(v^n \mathbin{++}\ [\textbf{br}\ k])]\ \textbf{end}] \hookrightarrow (v^n \mathbin{++}\ es_{\mathsf{cont}})$	
$[\textbf{i32}.\textbf{const}\ c; \textbf{br_if}\ k] \hookrightarrow [\textbf{br}\ k]$ $[\textbf{i32}.\textbf{const}\ c; \textbf{br_if}\ k] \hookrightarrow []$	$c \neq 0$ $c = 0$
$[\textbf{i32}.\textbf{const}\ i; \textbf{br_table}\ ls\ l_N] \hookrightarrow [\textbf{br}\ l_i]$ $[\textbf{i32}.\textbf{const}\ i; \textbf{br_table}\ ls\ l_N] \hookrightarrow [\textbf{br}\ l_N]$	$ls[i] = l_i$ $\lvert ls \rvert \leq i$
$[\textbf{label}_n\{es_{\mathsf{cont}}\}\ vs\ \textbf{end}] \hookrightarrow vs$	$\mathsf{is_val}\ vs$
$(S; F; B^k[es]) \hookrightarrow (S'; F'; B^k[es'])$	$(S; F; es) \hookrightarrow (S'; F'; es')$

Figure 2.11: Reduction rules for control instructions (block constructs)

The instruction **call** *i* simply calls the *i*th function (0-indexed) in the list of function instances accessible by the current module. This is resolved by looking up the corresponding entry of the module instance $F.\mathsf{module.funcaddrs}[i]$, which contains the address in the store for the function being called.

In contrast, **call_indirect** *i* is a dynamic function call which calls a function sourced from the function table at an index specified by the top **i32** value on the operand stack. The function references in the function table can be populated at runtime by the host. The argument *i* for **call_indirect** is an index to the list of function types $F.\mathsf{module.types}$ in the module instance, which specifies the expected function type of the function being called. If the index of the function is out of bounds of the table, or if the function type of the function in the table does not match the specified function type by the instruction, then a **trap** is produced.

In either case, Wasm uses the administrative instruction **invoke** *funcaddr* to handle the invocation of the function by an address in the store. This instruction handles the *invocation* of the function by looking up the function instance at the specified address from the store, and creates a function call frame on the stack by reducing to the following frame construct:

$$\textbf{frame}_n\ \{F\}\ [\textbf{label}_m\ \{[]\}\ es\ \textbf{end}]\ \textbf{end}$$

which contains a **label** enclosing the body of the instruction as the body of the frame with an arity *m* equal to the number of values produced by the function body (given by its signature). The module instance of the frame *F* and the body *es* are retrieved from the corresponding fields of

the function instance. The local variables consist of two parts: a leading list of values which are retrieved from the top of the operand stack from the **invoke** instruction, followed by a list of local variables defined by the function itself, initialised to the default values of the corresponding types, represented by $\mathsf{default}_{ts}$. Finally, the arity n of the **frame** describes the number of values to be returned from the function, which can be determined from the function type of the function.

Execution of a frame construct is similar to that of a label: the body is executed until it consists of entirely values, or a **return** instruction is encountered. There is a slight difference: although the $\mathbf{frame}_n\{F\}(-)\ \mathbf{end}$ also behaves similarly to an evaluation context, execution of the body uses the frame F instead of the frame in the configuration tuple of the **frame** instruction. This is captured by the following reduction rule:

$$\frac{(S;F;es) \hookrightarrow (S';F';es')}{(S;F_0;[\mathbf{frame}_n\ \{F\}\ es\ \mathbf{end}]) \hookrightarrow (S';F_0;[\mathbf{frame}_n\ \{F'\}\ es'\ \mathbf{end}])}$$

The **return** instruction exits from the enclosing **frame** instruction, along with all **label**s en route, taking a number of values equal to the arity of the frame from the operand stack where the **return** instruction is executed as the result returned.

The operational semantics of the function call instructions are described in Fig 2.12.

(instructions) e ::=	
	... |
(control instructions)	$\mathbf{call}\ i \mid \mathbf{call_indirect}\ i \mid \mathbf{return} \mid$
(administrative instructions)	$\ldots \mid \mathbf{invoke}\ \mathit{funcaddr} \mid \mathbf{frame}_n\{F\}\ es\ \mathbf{end}$

Rule	Condition
$(F;[\mathbf{call}\ x]) \hookrightarrow (F;[\mathbf{invoke}\ a])$	$F.\mathsf{module}.\mathsf{funcaddrs}[x] = a$
$(S;F;[\mathbf{i32.const}\ i;\mathbf{call_indirect}\ x]) \hookrightarrow (S;F;[\mathbf{invoke}\ a])$	$S.\mathsf{tables}[F.\mathsf{module}.\mathsf{tableaddrs}[0]].\mathsf{elem}[i] = \mathsf{Some}\ a \wedge$ $F.\mathsf{module}.\mathsf{types}[x] = S.\mathsf{funcs}[a].\mathsf{type}$
$(S;F;[\mathbf{i32.const}\ i;\mathbf{call_indirect}\ x]) \hookrightarrow (S;F;[\mathbf{trap}])$	otherwise
$(S;F_0;v^n +\!\!+\ [\mathbf{invoke}\ a]) \hookrightarrow$ $(S;F_0;\mathbf{frame}_m\ \{F\}\ [\mathbf{label}_m\ \{[]\}\ es_{\mathsf{body}}\ \mathbf{end}]\ \mathbf{end}$	$S.\mathsf{funcs}[a] = f \wedge$ $f.\mathsf{type} = t_1^n \to t_2^m \wedge$ $f.\mathsf{code} = \{x;ts;es_{\mathsf{body}}\} \wedge$ $F = \{f.\mathsf{module}; v^n +\!\!+\ (\mathsf{default}_{ts})\}$
$[\mathbf{frame}_n\ \{F\}\ B^k[v^n +\!\!+\ [\mathbf{return}]]\ \mathbf{end}] \hookrightarrow v^n$	
$(S;F_0;[\mathbf{frame}_n\ \{F\}\ es\ \mathbf{end}]) \hookrightarrow (S';F_0;[\mathbf{frame}_n\ \{F'\}\ es'\ \mathbf{end}])$	$(S;F;es) \hookrightarrow (S';F';es')$

Figure 2.12: Reduction rules for control instructions (function calls)

2.4 Wasm's Type System

As forward-referenced by several places in Section 2.3, Wasm's validation phase ensures that the execution of a validated Wasm program does not result in a stuck state before reaching the end of execution. The validation phase checks that Wasm modules are well-formed; in particular, the functions defined in Wasm modules are well-typed according to Wasm's type system.

The basic building blocks of the type system are a collection of typing rules for singleton Wasm instructions of the form $C \vdash e : ft$, which associates a Wasm instruction e with a *function type ft* under a typing *context* C, specifying the impact of the instruction e on the operand stack under an execution environment described by the context C. In particular, $C \vdash e : ts_1 \to ts_2$ can be interpreted as: under an environment specified by the typing context C, the instruction e consumes a list of values of types ts_1 from the operand stack and produces a list of values of types ts_2 to the operand stack.

These typing rules are then lifted and extended to define a *fragment* typing relation $C \vdash es : ft$ for general *program fragments es* via some structural typing rules, which then helps to define a typing relation for the entire Wasm configurations $(S;F;es)$ via some further auxiliary definitions.

Wasm's type system is designed in such a way that a well-typed Wasm configuration $(S;F;es)$ does not get stuck at a non-terminal state. This is further described in the Soundness section, and is one of the key properties verified in the mechanisation, namely the *type soundness* property.

2.4.1 Typing Contexts

The typing context C in Wasm collects the typing information from the environment of an instruction in the configuration tuple $(S;F;es)$. This includes:

- The types of local variables in $F.\mathsf{locals}$. This is used in the typing of instructions that access local variables;
- The types of states in the store S accessible by the current module, determined by $F.\mathsf{module}$. This is used in the typing of various instructions that access the global state in the store S;
- The return types of labels and frames 'reachable' by the current instruction list. This is used in typing, for example, the branching instructions within nested label blocks.

Formally, the typing context is defined as a record of the following form described in Figure 2.13, which also recalls some basic definitions previously introduced in Figure 2.4 and defines the types of the other Wasm states.

To briefly describe the types of Wasm 1.0 states in text:

- Function types are specified by the types of their arguments and results;[6]
- Table and memory types are specified by their size *limits*[7], each of which consists of a minimum size and an optional maximum size. In the case of memories, the sizes are specified in the unit of pages (2^{16} bytes);

[6]Note that Wasm 1.0 uses the same notion of *function types* for the type of instructions to describe their impact on the operand stack, and the type of functions to describe their arguments and results.

[7]The Wasm specification in fact defines the table type to be a product type of limits and a (table) element type. However, in Wasm 1.0, the element type is defined as an inductive with only one constructor funcref representing the type of function references, which is essentially a unit type. I therefore choose to omit it in the definition to avoid unnecessary notations. However, the full notation is used when I discuss the Wasm 2.0 extension which defines non-trivial element types in Section 3.10.

- Global variable types are specified by the value type accompanied by a mutability flag $mut ::= \textbf{const} \mid \textbf{var}$, representing that the corresponding global variable is immutable/mutable respectively.

$$
\begin{aligned}
\text{(value type) } t &::= \textbf{i32} \mid \textbf{i64} \mid \textbf{f32} \mid \textbf{f64} \\
\text{(function type) } ft &::= ts \to ts \\
\text{(result type) } rt &::= ts
\end{aligned}
$$

$$
\begin{aligned}
\text{(limits) } limits &::= \{\mathsf{min} :: u32, \mathsf{max} :: u32^?\} \\
\text{(table types) } tabletype &::= limits \\
\text{(memory types) } memtype &::= limits \\
\text{(global mutability) } mut &::= \textbf{const} \mid \textbf{var} \\
\text{(global type) } globaltype &::= \{\mathsf{mut} :: mut, \mathsf{type} :: t\}
\end{aligned}
$$

$$
\text{(typing context) } C \ ::= \left\{ \begin{array}{l} \mathsf{types} :: fts, \\ \mathsf{funcs} :: fts,\ \mathsf{tables} :: tabletypes, \\ \mathsf{mems} :: memtypes,\ \mathsf{globals} :: globaltypes, \\ \mathsf{locals} :: ts,\ \mathsf{labels} :: rts,\ \mathsf{return} :: rt^? \end{array} \right\}
$$

Figure 2.13: Wasm 1.0 typing context

For a given store S and a frame F, a *frame validity relation* of the form $S \vdash_{\mathsf{f}} F : C$ is defined, which associates a frame to a typing context under a store S. This typing context C is then used to check the type of the current program es in the Wasm configuration. I explain the details of the frame validity relation (alongside other validity relations) in Section 2.5.

2.4.2 Typing Rules for Wasm 1.0 Instructions

I now introduce the syntactic typing rules of Wasm 1.0 instructions. Similar to the operational semantics, I describe these typing rules in several categories.

Context-agnostic Rules

The typing rules in this category do not interact with the typing context at all. This includes the numeric instructions, parametric instructions, and the simpler control flow instructions **nop** and **select**. They are displayed in Figure 2.14. These rules are straightforward, except the rule for **unreachable**: this rule specifies that the **unreachable** instruction can be assigned with any function type, making it *stack-polymorphic*. Similar polymorphic rules would later be seen for some control instructions that perform conditional or unconditional branches, such as **br** and its variations, and **return**.

Variable and Memory Instructions

The typing rules for variable and memory instructions are presented in Figure 2.15 each requires some conditions on the typing context according to the part of the Wasm states they are trying to access. For the local variable instructions, the only condition is that the ith local exists in the current context, and the instruction is assigned to the corresponding function type according to the type of the local variable. For the global variable instructions, in addition to the existence

$$\frac{}{C \vdash t.\mathbf{const}\ c : [] \to [t]}\ \textsf{const} \qquad \frac{}{C \vdash t.\mathbf{unop}\ \mathit{unop} : [t] \to [t]}\ \textsf{unop}$$

$$\frac{}{C \vdash t.\mathbf{binop}\ \mathit{binop} : [t;t] \to [t]}\ \textsf{binop} \qquad \frac{}{C \vdash t.\mathbf{testop}\ \mathit{testop} : [t] \to [\mathbf{i32}]}\ \textsf{testop}$$

$$\frac{}{C \vdash t.\mathbf{relop}\ \mathit{relop} : [t;t] \to [\mathbf{i32}]}\ \textsf{relop} \qquad \frac{}{C \vdash t_2.\mathbf{cvtop}\ \mathit{cvtop}\ t_1\ \mathit{sx}^? : [t_1] \to [t_2]}\ \textsf{cvtop}$$

$$\frac{}{C \vdash \mathbf{drop} : [t] \to []}\ \textsf{drop} \qquad \frac{}{C \vdash \mathbf{select} : [t;t;\mathbf{i32}] \to [t]}\ \textsf{select}$$

$$\frac{}{C \vdash \mathbf{nop} : [] \to []}\ \textsf{nop} \qquad \frac{}{C \vdash \mathbf{unreachable} : ts_1 \to ts_2}\ \textsf{unreachable} \qquad \frac{}{C \vdash \mathbf{trap} : ts_1 \to ts_2}\ \textsf{trap}$$

$$\frac{C \vdash es : ts_1 \to ts_3 \quad C \vdash [e] : ts_3 \to ts_2}{C \vdash es \mathbin{++} [e] : ts_1 \to ts_2}\ \textsf{composition}$$

$$\frac{C \vdash es : ts_1 \to ts_2}{C \vdash es : ts_1 \mathbin{++} ts_3 \to ts_2 \mathbin{++} ts_3}\ \textsf{subsumption}$$

Figure 2.14: Typing rules of WebAssembly 1.0 (context-agnostic)

of the corresponding global variable in the context, the global variable also needs to be mutable for **global**.**set**. For the memory instructions which implicitly operate on the 0th memory, the only constraints include the existence of such a memory in the context, and that the alignment parameter is well-formed according to the definition.

$$\frac{C.\textsf{locals}[i] = t}{C \vdash \mathbf{local}.\mathbf{get}\ i : [] \to [t]}\ \textsf{local.get} \qquad \frac{C.\textsf{locals}[i] = t}{C \vdash \mathbf{local}.\mathbf{set}\ i : [t] \to []}\ \textsf{local.set} \qquad \frac{C.\textsf{locals}[i] = t}{C \vdash \mathbf{local}.\mathbf{tee}\ i : [t] \to [t]}\ \textsf{local.tee}$$

$$\frac{C.\textsf{globals}[i] = (-, t)}{C \vdash \mathbf{global}.\mathbf{get}\ i : [] \to [t]}\ \textsf{global.get} \qquad \frac{C.\textsf{globals}[i] = (\mathbf{var}, t)}{C \vdash \mathbf{global}.\mathbf{set}\ i : [t] \to []}\ \textsf{global.set}$$

$$\frac{0 < |C.\textsf{mems}| \quad 2^{\mathit{memarg}.\textsf{align}} \le \frac{|t|}{8}}{C \vdash t.\mathbf{load}\ \mathit{memarg} : [\mathbf{i32}] \to [t]}\ \textsf{load} \qquad \frac{0 < |C.\textsf{mems}| \quad 2^{\mathit{memarg}.\textsf{align}} \le \frac{|sz|}{8}}{C \vdash t.\mathbf{load}\ (sz, sx)\ \mathit{memarg} : [\mathbf{i32}] \to [t]}\ \textsf{load_packed}$$

$$\frac{0 < |C.\textsf{mems}| \quad 2^{\mathit{memarg}.\textsf{align}} \le \frac{|t|}{8}}{C \vdash t.\mathbf{store}\ \mathit{memarg} : [\mathbf{i32}; t] \to []}\ \textsf{store} \qquad \frac{0 < |C.\textsf{mems}| \quad 2^{\mathit{memarg}.\textsf{align}} \le \frac{|sz|}{8}}{C \vdash t.\mathbf{store}\ (sz, sx)\ \mathit{memarg} : [\mathbf{i32}; t] \to []}\ \textsf{store_packed}$$

$$\frac{0 < |C.\textsf{mems}|}{C \vdash \mathbf{memory}.\mathbf{size} : [] \to [\mathbf{i32}]}\ \textsf{memory.size} \qquad \frac{0 < |C.\textsf{mems}|}{C \vdash \mathbf{memory}.\mathbf{grow} : [\mathbf{i32}] \to [\mathbf{i32}]}\ \textsf{memory.grow}$$

Figure 2.15: Typing rules of WebAssembly 1.0 (variables and memory)

Control Flow Instructions (Block)

I first discuss the typing rules for the block-like control flow instructions, displayed in Figure 2.16. Each of these typing rules interacts with C.labels component of the typing context, which specifies

the required value types by the labels enclosing the current instruction, which can therefore be selected as the target of branch instructions.

The foundational typing rule in this category is in fact the rule for the administrative instruction **label** itself. This rule demonstrates the constraints that need to be satisfied for a label block to be well-typed with some function type $[] \to rt$:

- The continuation of the label needs to be typeable by consuming some list of types t_1^n (whose length agrees with the arity of the label) and producing rt;
- The body of the label needs to be of function type $[] \to rt$, under the context with t_1^n prepended to the label component of the same context.

To summarise, the types of values produced by the continuation need to agree with the types of values produced by the body, and the types consumed by the continuation need to agree with the arity of the label (which comes from the specified result type of the block instructions that reduce to the label).

The three block instructions **block**/**loop**/**if** are typed correspondingly to their intended control flow structure: each of the rules specifies that the type of the instruction matches the type(s) of their body/bodies, under a modified context with a new label of the corresponding type prepended to the original context. I provide some further explanations on why the specified typing rules agree with the operational semantics:

- The **block** instruction reduces to a **label** instruction with an empty continuation. Therefore, for the execution results of the continuation to match that of the label body, the inner branch instruction needs to provide the expected produced type rt;
- The **loop** instruction reduces to a **label** instruction with a continuation equal to the loop body itself, which will produce values of type rt by the other premise. Therefore, no additional value is required to be provided to this label when it is targeted;
- The **if** instruction is typed similarly to the **block** instruction due to having no continuation, although the types of its two branches need to match.

The typing rules for the branch instructions are also stack-polymorphic. They specify that the specified label must exist in the context, and that the top values on the operand stack need to agree with the requirement of targeting the corresponding label. For **br_table**, all result types of the labels that appeared in the parameters need to agree with each other; for otherwise the instruction could result in different function types when specifying a different label.

Control Flow Instructions (Function Call)

The typing rules for the function call instructions are displayed in Fig 2.17. I postpone the typing rules for the **frame** and **invoke** administrative instructions to Section 2.5, as they rely on several more complicated auxiliary definitions that are only used for the typing of administrative instructions and Wasm configurations.

$$\frac{C \vdash es_{\mathsf{cont}} : t_1^n \to rt \quad (C \text{ with } \mathsf{labels} = [t_1^n] +\!\!+ C.\mathsf{labels}) \vdash es : [] \to rt}{C \vdash \mathbf{label}_n \; \{es_{\mathsf{cont}}\} \; es \; \mathbf{end} : [] \to rt} \; \mathsf{label}$$

$$\frac{(C \text{ with } \mathsf{labels} = [rt] +\!\!+ C.\mathsf{labels}) \vdash es : [] \to rt}{C \vdash \mathbf{block} \; rt \; es \; \mathbf{end} : [] \to rt} \; \mathsf{block} \qquad \frac{(C \text{ with } \mathsf{labels} = [\varepsilon] +\!\!+ C.\mathsf{labels}) \vdash es : [] \to rt}{C \vdash \mathbf{loop} \; rt \; es \; \mathbf{end} : [] \to rt} \; \mathsf{loop}$$

$$\frac{C' = (C \text{ with } \mathsf{labels} = [rt] +\!\!+ C.\mathsf{labels}) \quad C' \vdash es_1 : [] \to rt \quad C' \vdash es_2 : [] \to rt}{C \vdash \mathbf{if} \; rt \; es_1 \; \mathbf{else} \; es_2 \; \mathbf{end} : [] \to rt} \; \mathsf{if}$$

$$\frac{C.\mathsf{labels}[i] = rt}{C \vdash \mathbf{br} \; i : ts_1 +\!\!+ rt \to ts_2} \; \mathsf{br} \qquad \frac{C.\mathsf{labels}[i] = rt}{C \vdash \mathbf{br_if} \; i : rt +\!\!+ [\mathbf{i32}] \to rt} \; \mathsf{br_if}$$

$$\frac{C.\mathsf{labels}[ld] = rt \quad C.\mathsf{labels}[l_i] = rt}{C \vdash \mathbf{br_table} \; ls \; ld : ts_1 +\!\!+ rt +\!\!+ [\mathbf{i32}] \to ts_2} \; \mathsf{br_table}$$

Figure 2.16: Typing rules of WebAssembly 1.0 (control flow: block instructions)

I highlight the **call_indirect** instruction, whose argument i refers to the C.types component for type-checking the function being referenced during execution.

The typing rule for **return** is similarly stack-polymorphic akin to the **br** instruction, except that it refers to the C.return component for the correct result type to be returned.

$$\frac{C.\mathsf{funcs}[i] = ts_1 \to ts_2}{C \vdash \mathbf{call} \; i : ts_1 \to ts_2} \; \mathsf{call} \qquad \frac{0 < |C.\mathsf{tables}| \quad C.\mathsf{types}[i] = ts_1 \to ts_2}{C \vdash \mathbf{call_indirect} \; i : (ts_1 +\!\!+ [\mathbf{i32}]) \to ts_2} \; \mathsf{call_indirect}$$

$$\frac{C.\mathsf{return} = rt}{C \vdash \mathbf{return} : (ts_1 +\!\!+ rt) \to ts_2} \; \mathsf{return}$$

$$\frac{S.\mathsf{funcs}[addr].\mathsf{type} = ts_1 \to ts_2}{S;C \vdash \mathbf{invoke} \; addr : ts_1 \to ts_2} \; \mathsf{invoke}$$

Figure 2.17: Typing rules of WebAssembly 1.0 (function call instructions)

Structural Typing Rules

Lastly, I introduce the two structural typing rules of Wasm 1.0 in Figure 2.18. These rules specify how the typing rules for singleton instructions are composed to construct typing relations for instruction lists.

$$\frac{C \vdash es : ts_1 \to ts_3 \quad C \vdash [e] : ts_3 \to ts_2}{C \vdash es +\!\!+ [e] : ts_1 \to ts_2} \; \mathsf{composition}$$

$$\frac{C \vdash es : ts_1 \to ts_2}{C \vdash es : ts_3 +\!\!+ ts_1 \to ts_3 +\!\!+ ts_2} \; \mathsf{subsumption}$$

Figure 2.18: Typing rules of WebAssembly 1.0 (structural)

The composition rule is similar to the sequence rule for languages with an explicit sequence con-

structor as part of the AST; it allows composition of typing relations to be performed at the tail of instruction lists. However, it is possible to prove by induction that a similar composition rule holds for arbitrary concatenations of two instruction lists.

The subsumption rule is special to the stack-machine model of Wasm's computation. It specifies that if an instruction list es can be typed as consuming values of types ts_1 and producing values of types ts_2, then it can also be typed as both consuming and producing a further list of value types ts_3. This is illustrated in Figure 2.19.

Figure 2.19: Illustration of the subsumption typing rule

2.5 Administrative Instruction Typing and Validity Relations

Section 2.4 describes the typing rules for all Wasm 1.0 basic instructions. To fully define Wasm's soundness property, it is necessary to specify the typing of all administrative instructions, as well as the validity of certain Wasm constructs such as Wasm configurations. However, the above requires a number of auxiliary relations to be introduced, which are otherwise scattered throughout different sections. In this section, I present these definitions altogether.

Figure 2.20 contains an overview of the relevant validity relations. This section will proceed to explain them in detail, followed by the typing rules of the administrative instructions that depend on these validity relations.

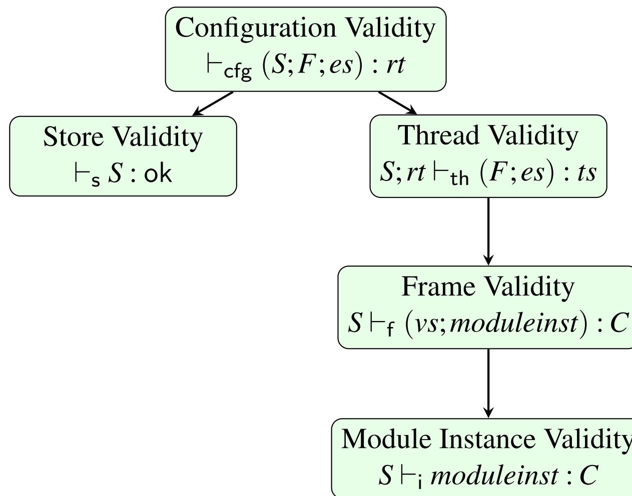


Figure 2.20: Overview of Wasm validity relations

2.5.1 Module Instance, Frame, and Thread Validity

Recall that all typing rules in Section 2.4 require a typing context C. However, the Wasm configuration tuple $(S;F;es)$ contains no such context inherently. Instead, Wasm defines the frame

validity relation

$$S \vdash_{\mathsf{f}} F : C$$

which associates the frame with a typing context under a Wasm store. This frame validity relation is then used in a separate thread typing rule to specify the typing context used for the program, and also for the typing of the **frame** administrative instruction itself.

The frame validity relation depends on the validity of the module instance

$$S \vdash_{\mathsf{i}} \mathit{moduleinst} : C$$

The module instance validity relation associates a type to each Wasm instance contained in the module instance in the corresponding entry of the typing context C. Recall that each entry in the module instance is a reference by index to the Wasm store. As a result, the store S is required as part of this validity relation, so that the types of the corresponding references can be determined.

The module instance validity relation checks the validity of each individual store reference in the module instance against the corresponding type in the typing context. This is possible since all runtime instances in the Wasm store carry sufficient information about their types. In particular, for functions and globals, their types are carried in their runtime representation. For tables and memories, the optional maximum sizes and their current sizes are used to determine if they match with the corresponding type in the typing context.

The above validity relation of individual store references is known as the *external validity relation* in the Wasm specification. This relation is originally defined for validating the module imports, where each import is also a store reference that needs to match a specific import type. The Wasm specification reuses this validity relation for typing the store references in the module instances.

The full formal definition of the module instance validity relation $S \vdash_{\mathsf{i}} \mathit{moduleinst} : C$ and the external validity relations are given in Appendix B.

The thread and frame validity relations allow the typing rule of the **frame** $\{F\}$ *es* administrative instruction to be defined by the typing of the corresponding thread $(F; es)$. This is included in Figure 2.21 at the end of the section.

2.5.2 Store Validity

The purpose of the store validity relation is to ensure that all runtime instances within the store – namely, the instances of functions, tables, memories, and globals – are consistent internally. For example, the **invoke** instruction is typed according to the function type *signature* that the corresponding function is annotated with, without verifying whether the body of the function matches this signature. Therefore, without other constraints and checks, Wasm's desired typing guarantees can be invalidated if a function in the Wasm store carries an incorrect type signature that does not match its body.

To prevent such inconsistencies, Wasm defines a *store validity* relation denoted as

$$\vdash_{\mathsf{s}} S : \mathsf{ok}$$

which includes the consistency conditions for all individual states in the Wasm store. These consistency conditions are given by the *validity* condition for each type of instance.

The full definition of the store validity relation and the validity of each component, are given in Appendix B.

2.5.3 Configuration Validity

Having defined the validity of threads and stores, a Wasm configuration $(S;F;es)$ is then associated with a result type rt if the store is valid, and the thread $(F;es)$ has the corresponding result type rt under the store S with no return type allowed (since the overall configuration is not within any function call to be returned from).

The rules discussed in this section are summarised in Figure 2.21.

$$\frac{S \vdash_{\mathsf{i}} moduleinst : C' \quad S \vdash vs : ts \quad C := C' \text{ with locals} := ts}{S \vdash_{\mathsf{f}} (vs;moduleinst) : C} \text{ frame_validity}$$

$$\frac{S \vdash_{\mathsf{f}} F : C \quad S;C \text{ with return} = rt \vdash es : [] \to ts}{S;rt \vdash_{\mathsf{th}} (F;es) : ts} \text{ thread_validity}$$

$$\frac{S;t^n \vdash_{\mathsf{th}} (F;es) : t^n}{S;C_0 \vdash \mathbf{frame}_n \; \{F\} \; es \; \mathbf{end} : [] \to t^n} \text{ frame}$$

$$\frac{\vdash_{\mathsf{s}} S : \mathsf{ok} \quad S;[] \vdash (F;es) : rt}{\vdash_{\mathsf{cfg}} (S;F;es) : rt} \text{ configuration_validity}$$

Figure 2.21: Validity relations and typing rule of **frame** administrative instruction

As a side note to conclude this section, several variants of typing and validity relations denoted by $\vdash$ accompanied by different subscripts are defined for several kinds of different validity relations. However, in most cases, the type of relation used can be inferred from the context and arguments. In the rest of the thesis, I sometimes omit these subscripts when there is no ambiguity.

2.6 Wasm's Type Soundness Property

One design of Wasm's type system and operational semantics is the soundness property that well-typed programs do not get stuck. In particular, execution starting from a well-typed Wasm configuration will not be stuck in a non-terminal state.

The Wasm specification states its soundness property using the standard progress and preservation properties [117] given as follows.

Proposition 2.6.1 (Preservation)**.**

$$\forall S, F, es, S', F', es'. ts. \left[(\vdash_{\mathsf{cfg}} (S; F; es) : ts \land (S; F; es) \hookrightarrow (S'; F'; es')) \implies (\vdash_{\mathsf{cfg}} (S'; F'; es') : ts \land S \preccurlyeq S') \right].$$

The preservation property states that the execution result of any well-typed Wasm configuration tuple has the same type as the original configuration; moreover, the new store S' is an *extension* of the old store S, denoted by the relation $S \preccurlyeq S'$.

Store extension The store extension relation in Proposition 2.6.1 is not a standard part of the traditional preservation property. It defines a set of invariants for the Wasm store during execution, for example, no allocated states are removed from the store, and no immutable objects are modified. These invariants ensure that a program that is well-typed in a given store S will always be well-typed (and of the same type) in any extension S' of S.

The execution semantics of all Wasm instructions respect the invariants described by the store extension relation as expected. The purpose of this definition is to describe the set of constraints which any host operation via host function calls must respect. Since the host semantics and functions are outside the scope of Wasm specification, Wasm semantics cannot enforce such conditions. Instead, the type soundness property is only satisfied when all host functions respect these invariants.

The set of invariants defined by the store extension relation $S \preccurlyeq S'$ is as follows, given by the types of the instance:

- All function instances of S' must be the same as the corresponding instances in S;
- The maximum sizes of table instances and memory instances of S' must be the same as the corresponding states in the old store S, and the length of their contents in S' cannot be smaller than the corresponding entries in the old store S;
- The mutability and the value type of globals cannot change. In addition, the values of globals cannot change if the global is immutable.

Note that the store extension property allows arbitrary new states to be allocated at the end of the original list of states. It only requires that the above conditions hold for the states corresponding to the old existing states (by address).

Proposition 2.6.2 (Progress)**.**

$$\forall S, F, es, ts. \left[\vdash_{\mathsf{cfg}} (S; F; es) : ts \implies (\exists S', F', es'. (S; F; es) \hookrightarrow (S'; F'; es')) \lor \mathsf{terminal}((S; F; es)) \right]$$

The progress property states that a well-typed Wasm configuration can either be reduced further in Wasm's operational semantics or it is of a pre-defied terminal form given by terminal predicate. In Wasm 1.0, a configuration is terminal if and only if its instruction list contains a single instruction **trap**, or consists of entirely values.

I describe our mechanised type soundness proof later in Section 3.5.

2.7 Wasm Modules, Validation, and Instantiation

With the semantics and type system of Wasm defined, this section explores the structure of Wasm modules for organising Wasm programs.

A Wasm module is formally defined as a record with shape displayed in Figure 2.22, consisting of several lists containing different categories of definitions of Wasm states, the type signatures used in the module, the table and memory initialisers (elem and data), the imports and exports, and an optional start function.

$$\text{(modules) } \mathit{module} \;::=\; \left\{\begin{array}{l} \mathsf{types} :: \mathit{fts}, \\ \mathsf{funcs} :: \mathit{funcs}, \\ \mathsf{tables} :: \mathit{tables}, \\ \mathsf{mems} :: \mathit{mems}, \\ \mathsf{globals} :: \mathit{globals}, \\ \mathsf{elem} :: \mathit{elems}, \\ \mathsf{data} :: \mathit{datas}, \\ \mathsf{start} :: \mathit{start}^?, \\ \mathsf{imports} :: \mathit{imports}, \\ \mathsf{exports} :: \mathit{exports} \end{array}\right\}$$

$$\begin{array}{rl}
\text{(functions) } \mathit{func} & ::= \{\mathsf{type} :: \mathit{typeid}, \mathsf{locals} :: \mathit{ts}, \mathsf{body} : \mathit{es}\} \\
\text{(tables) } \mathit{table} & ::= \{\mathsf{type} :: \mathit{tabletype}\} \\
\text{(memories) } \mathit{mem} & ::= \{\mathsf{type} :: \mathit{memtype}\} \\
\text{(globals) } \mathit{global} & ::= \{\mathsf{type} :: \mathit{globaltype}, \mathsf{init} :: \mathit{es}\} \\
\text{(element segment) } \mathit{elem} & ::= \{\mathsf{table} : \mathit{tableid}, \mathsf{offset} : \mathit{es}, \mathsf{init} :: \mathit{funcids}\} \\
\text{(data segment) } \mathit{data} & ::= \{\mathsf{mem} :: \mathit{memid}, \mathsf{offset} : \mathit{es}, \mathsf{init} :: \mathit{bytes}\} \\
\text{(start function) } \mathit{start} & ::= \{\mathsf{func} :: \mathit{funcid}\} \\
\text{(imports) } \mathit{import} & ::= \{\mathsf{module} :: \mathit{name}, \mathsf{entity} :: \mathit{name}, \mathsf{desc} :: \mathit{importdesc}\} \\
\text{(exports) } \mathit{export} & ::= \{\mathsf{name} :: \mathit{name}, \mathsf{desc} :: \mathit{exportdesc}\}
\end{array}$$

$$\begin{array}{rl}
\text{(import descriptions) } \mathit{importdesc} & ::= \mathbf{func}\ \mathit{typeid} \mid \mathbf{table}\ \mathit{tabletype} \mid \mathbf{mem}\ \mathit{memtype} \mid \mathbf{global}\ \mathit{globaltype} \\
\text{(export descriptions) } \mathit{exportdesc} & ::= \mathbf{func}\ \mathit{funcid} \mid \mathbf{table}\ \mathit{tableid} \mid \mathbf{mem}\ \mathit{memid} \mid \mathbf{global}\ \mathit{globalid}
\end{array}$$

Figure 2.22: Structure of Wasm 1.0 modules

Type Signatures The types field collects the function type signatures used in the Wasm module. As of Wasm 1.0, this component is only used by the functions of the module, as well as the **call_indirect** instruction, which refers to this component for type-checking the dynamic function calls. In Wasm 2.0, the block instructions can also refer to this component of the module as their function type signature.

Functions Each function in the funcs component is a record consisting of its function type signature represented by an index to the types component, a list of local variable types defined by the function, which is in addition to its arguments specified by the function type and appended to the input arguments, and the function body which is a list of (basic) instructions.

Tables, Memories, and Globals The definition of a table, memory, and a global in a module is similar: each definition only specifies their corresponding type (i.e. table type/memory type/-

global type, introduced in Figure 2.13). The definition of a global additionally carries an initialiser expression that is evaluated during module instantiation to provide the initial value of the global variable. The tables and memories are by default initialised to null references and zero-valued bytes respectively. The element and data segments can define further initialisers for tables and memories respectively in separate fields of the module.

Element and Data Segments Each element and data segment is a record specifying a specific table or memory location and a list of the corresponding type of data to initialise a contiguous segment starting from the location. The location is specified by the index of the table or memory and an offset given by an instruction list, which is evaluated to a value during instantiation.

Start Function A Wasm module can optionally specify one of its functions (by its index) as its start function. The start function of a module will be executed immediately after instantiating the module.

Imports and Exports Only the *descriptions* field of the imports and exports definitions have observable impact on the semantic behaviour within Wasm. Each import description specifies the expected *type* of the import by a category of Wasm state and a corresponding type of the state. The imported states will be added to the module's lists of available definitions during instantiation, prepended to the states defined by the module itself. Each export description, in contrast, specifies a category and an index to the state defined in the module which is to be exported to the host. It is allowed to export an imported state as well.

The *name* fields in the import and export definitions are UTF8-encoded Unicode strings. They are only used by the host, with the constraint that the export names must be unique within a module. The host is responsible for locating the corresponding states to be imported by a module and handling the storage of module exports. Both of these are host operations and are out of the scope of the Wasm specification.

2.7.1 Validation of Wasm Modules

The Wasm specification defines a typing relation for each category of the module components, assigning a module component with its corresponding type under a typing context C. However, note that the four Wasm states – functions, tables, memories, and globals – already carry their expected types, as displayed in Figure 2.22 (the types of functions indicated are represented by the index to the types field of the module). As a result, each of the typing relation is essentially defining the validity of the module components, specifying the condition necessary for each of the components to be valid (in the case of the four Wasm states, valid with their corresponding types).

A Wasm module is valid if all of its components are valid under an appropriate context corresponding to the collection of definitions the components are supposed to access, which I defined after the following descriptions of the typing relations first.

Functions and the Start Function A module function $\{\mathsf{type} : x, \mathsf{locals} : ts, \mathsf{body} : es\}$ is valid under a context C with its specified type signature, i.e. $C.\mathsf{types}[x]$, if the following conditions hold:

- The reference $C.\mathsf{types}[x] = ts_1 \to ts_2$ exists (not out of bound);
- The body is of function type $[] \to ts_2$ under a modified context from C:

$$C[\mathsf{locals} := ts_1 +\!\!+\ ts, \mathsf{labels} := [ts_2], \mathsf{return} := ts_2] \vdash es : [] \to ts_2$$

  The modified context represents the context that the body of the function will be executed in. The locals field captures the correct type of local variables accessible in the function (including the arguments and the locals defined by the function itself). The contents of the labels and return field reflect the fact that the function body is executed within an enclosing **frame** and a **label**, as introduced by the reduction rule of **invoke** in Figure 2.12.

The start function is optional and is an index to a function definition of the module if specified. As a result, it is valid as long as the index is not out of bound.

Tables and Memories Recall from Figure 2.22 that tables and memories definitions consist of only their types, and from Figure 2.13 that the types of tables and memories are given by their size limits. Therefore, the only constraint for a table or memory definition to be valid is the validity of their size limits. This is specified by a separate limit validity relation, which states that the minimum size min must not exceed the optional maximum size max if present.

In addition to the validity of the size limit, Wasm also specifies an upper bound on the maximum size that can be defined due to its 32-bit addresses. This upper bound is 2^{32} for table limits and 2^{16} for memory limits in Wasm 1.0. Note that the sizes of memories are given in the unit of pages where 1 page is equal to 2^{16} bytes, so Wasm 1.0 effectively sets the same bound of 2^{32} on the number of elements/bytes in tables and memories respectively.[8]

The Wasm specification defines the limit validity relation in a more general way to capture the above additional constraints. The full definition is included in Appendix B.

Globals For the definition of a global variable $\{\mathsf{type} :: \textit{globaltype}, \mathsf{init} :: es\}$ to be valid under a context C, its initialiser expression es needs to be of the same type as specified by the type signature of the global under the given context C. Furthermore, the instructions allowed to appear in the initialisers es is limited to a set of *constant expressions* defined by the Wasm specification, which includes only values of form t.**const** c, and the **global**.**get** instructions from immutable globals. This design ensures that there is no cyclicity in the definition of functions and global variables, so that the *instantiation* of modules can be done in one pass, as I will discuss later in Section 2.7.2. In Wasm 1.0, an additional constraint is imposed on the global initialisers, so that they are restricted to only referring to the imported globals, instead of using other globals in the module. This is enforced by using a different typing context C to validate the global definitions,

[8] A feature extension (Memory64)[73] including the extension of memory addresses to 64 bits is in progress. As of November 2024, this extension is still being standardised. Therefore, these bounds may change in a future version of Wasm when 64-bit memory addresses are introduced.

as I will introduce later in this section. This design helps to avoid cyclic dependency among the global initialisers.

Element and Data Segments Elements and data segments are valid under a typing context C if the index of table/memory they refer to exist in the context C (i.e. not out of bound), and the offset expression is of type **i32** in the given context. Furthermore, only constant expressions can be used in the offset expressions, similar to the constraint of the global initialiser expressions.

Imports and Exports Imports and exports defined by a module are valid if their corresponding import/export descriptions are valid.

For imports, as each description of the import is specified by their type, this type needs to be valid: this means the same conditions for the table and memory types as in the table and memory definitions. For functions, the import type is specified via an index to the C.types field, so this index must be within the bound. Global types are always valid.

For exports, each export description refers to a defined state of the module via an index. Therefore the only condition for an export description to be valid is that the specified index exists in the corresponding field of the module.

Typing contexts for Wasm module validation A Wasm module is valid if all of its definitions are valid under a context C corresponding to the types of the module definitions. For a module

$$module ::= \left\{ \begin{array}{l} \mathsf{types} : fts, \\ \mathsf{funcs} : funcs, \\ \mathsf{tables} : tables, \\ \mathsf{mems} : mems, \\ \mathsf{globals} : globals, \\ \mathsf{elem} : elems, \\ \mathsf{data} : datas, \\ \mathsf{start} : start^?, \\ \mathsf{imports} : imports, \\ \mathsf{exports} : exports \end{array} \right\}$$

The Wasm specification defines its corresponding typing context C as follows:

$$C ::= \left\{ \begin{array}{l} \mathsf{types} : fts, \\ \mathsf{funcs} : ts_{\mathsf{funcs_import}} +\!\!+\ ts_{\mathsf{funcs}}, \\ \mathsf{tables} : ts_{\mathsf{tables_import}} +\!\!+\ ts_{\mathsf{tables}}, \\ \mathsf{mems} : ts_{\mathsf{mems_import}} +\!\!+\ ts_{\mathsf{mems}}, \\ \mathsf{globals} : ts_{\mathsf{globals_import}} +\!\!+\ ts_{\mathsf{globals}} \end{array} \right\}$$

where $ts_{\mathsf{funcs/tables/mems/globals_import}}$ are the types of the corresponding components in the list of imports in the original order, and $ts_{\mathsf{funcs/tables/mems/globals}}$ are the types of the functions/tables/memories/globals defined in the module. The only exception to the above is the typing of

globals, which uses a slightly different typing context that only contains the imported global types, with all the other fields empty, i.e.

$$C' ::= \{\mathsf{globals} : ts_{\mathsf{globals_import}}\}$$

Note that there is an apparent circularity in the above definition: the typing of the components are defined under the context C, but the context C itself depends on the types of the components. However, each definition of a function/table/memory/global contains their expected signature as parts of the definition. Therefore, for an executable algorithm of validation, the typing context C can be constructed by inspecting the type signatures of the specified components first.

2.7.2 Instantiation of Wasm modules

Module instantiation is the phase where the definitions of a valid Wasm module are allocated in a Wasm store S.

The module instantiation phase checks that the given module is valid, and that the provided imports match the type of the imports requested by the module. The definitions of all the states in the module are then appended to the Wasm store, converting the module definitions to the corresponding runtime representations, i.e. instances. One particularly complex conversion here is the conversion of function definitions to function instances: recall that function instances are closures over the function definitions by the module instance. Note that the module instance can be fully determined at this point: it collects the addresses in the store where the module definitions are located in. Despite that these definitions have *not* been allocated into the store concretely, the addresses that they are going to be allocated in are known, since the new definitions will be appended to the existing states in the store S, and any address of the imported states are known. The tables, memories and globals are then initialised:

- For global variables, their initialiser expressions are simply evaluated with the resulting values filled to the global instances.
- The element and data segments are initialisers for tables and memories. Each of them specifies a location in a table or memory by the index, and an offset expression which evaluates to a **i32** value, representing a 32-bit address. The contents contained in the segments (in the init field) are then filled to the contiguous subarray starting at the evaluated offset of the specified table or memory.

  The effect of element and data segments are applied by their order in the module. It is therefore possible that a later initialiser segments overwrite some sections of tables or memories already initialised by some previous segments.

There is an apparent circularity in the definition of module instantiation in the Wasm specification here. Recall that the execution of any program requires a store S and a frame F. In the specification, these initialiser and offset expressions are evaluated under the module instance of the module and the post-instantiation store S. This choice renders the definition of the instantiation

operation to be a *relation* instead of a directly executable function, since the content of the post-instantiation store *depends* on the values of these initialiser and offset expressions. Nevertheless, recall from Section 2.7.1 that the initialiser and offset expressions are only allowed to use constant expressions, which include only values and **global.get** of immutables in Wasm 1.0. Furthermore, the **global.get** instructions used are only allowed to access the imported globals. Therefore, these values can be determined in one pre-pass before allocating the new states into the store.

After allocating the states defined by the module and obtaining the new store S, the start function of the module is invoked, if one is defined. This is represented by starting the execution in the following configuration

$$(S'; F; [\textbf{invoke}\ start])$$

where S' is the post-instantiation store and F is the module instance produced by the instantiation process.

I make a final comment that module instantiation is a responsibility of the host language. Although Wasm defines the effect of a module instantiation on the store, it does not specify how the imports or exports are handled, nor how the store S should be implemented in the host language. One can view the instantiation operation (or relation) defined by Wasm as the *specification* of the instantiation process, which the implementation of any host language must respect.

2.8 Example: Fibonacci Module

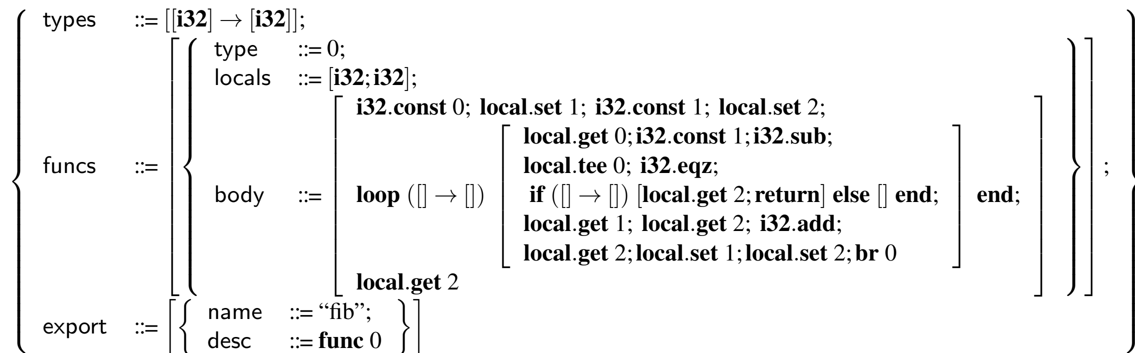


Figure 2.23: AST representation of the Fibonacci module (Revisit)

In this section, I revisit the example Fibonacci module in Figure 2.23 introduced at the start of the chapter and explain the lifecycle of the module, including the module instantiation, validation and execution of the exported function.

2.8.1 Overview of the Module

The Fibonacci module defines the following WebAssembly states:

- A single function type signature [**i32**] → [**i32**];
- A single function definition specified by the following:

  - Type index 0 – [**i32**] → [**i32**];
  - Two local variables of type, both of **i32**;
  - The function body, consisting of the instruction list displayed.

- A single export definition, exporting the function at index 0 (the single function defined in the module) by the name "fib".

I explain several noteworthy details of the function body:

- Despite that the function only defines two local variables, three are referenced in the function body (**local**.**get** referencing indices $0/1/2$). This is because the function also takes an **i32** value as a parameter. As explained in Section 2.7, the local variables are appended to all the parameters to form the local variables list. Therefore, the local variable 0 would refer to the input of the function, whereas the local variables $1/2$ refer to the two local variables defined by the function.

- In this function, the local variables $1/2$ would record the current two values in the Fibonacci sequence, starting at $F_0 = 0$ and $F_1 = 1$ by the first four instructions. The function then proceeds with a loop. At the start of each iteration, the input parameter is subtracted by 1, stored back to the local variable store, and then compared against zero (**i32**.**eqz**). If the value is zero, the value of the local variable 2 is fetched and returned as the result of the function, finishing the execution. Otherwise, nothing is done in the **if** instruction (note the empty **else** branch of the **if** statement). The function then proceeds to update the values of the current two Fibonacci values. The two values are first fetched and added on the stack (**i32**.**add**). Before setting the value back to the local variable store, a copy of the current local variable 1 is retrieved to the stack, so that the two new values can then be overwritten to the local variable store.

To conclude the overview, I show a C code fragment corresponding to the body of the Fibonacci function in Figure 2.24.

```
int fib (int v0) {
    int v1 = 0;
    int v2 = 1;
label0:
    v0 --;
    if (v0 == 0) return v2;
    int tmp = v1 + v2;
    v1 = v2;
    v2 = tmp;
    goto label0;
    return v2;
}
```

Figure 2.24: A C code fragment corresponding to the Fibonacci function

Note that the corresponding C code fragment requires an additional local variable `tmp` to store the temporary result of the addition. This is not required for the Wasm function due to the stack

machine model of computation, as the intermediate results can be stored on the stack.

2.8.2 Lifecycle of the Fibonacci Module

Recall that Wasm modules are instantiated by hosts, which first decodes the Fibonacci module from the compact binary format in Figure 2.1(b) to an intermediate representation – the Wasm AST in our example – displayed in Figure 2.23.

The host proceeds to validate the module. The type signature definition is always valid. The single export is also a valid reference to the 0th function of the module. The declared type signature of the Fibonacci function is $\mathbf{i32} \rightarrow \mathbf{i32}$ (0th type signature). As described in Section 2.7.1, the function body needs to be of the function type $[] \rightarrow \mathbf{i32}$ under the following typing context:

$$C ::= \left\{ \begin{array}{l} \mathsf{types} : [\mathbf{i32} \rightarrow \mathbf{i32}], \\ \mathsf{funcs} : [\mathbf{i32} \rightarrow \mathbf{i32}], \\ \mathsf{locals} : [\mathbf{i32}, \mathbf{i32}, \mathbf{i32}], \\ \mathsf{labels} : [[\mathbf{i32}]], \\ \mathsf{return} : [\mathbf{i32}] \end{array} \right\}$$

which can be verified by going through the individual typing rules defined in Section 2.4.2. An executable version of this type-checking procedure is specified in the Wasm specification to allow type checking to be done in one pass. I explain this in more details later in Section 3.6 when I introduce our WasmCert-Coq mechanisation.

Having validated the module, the instantiation process can now load the Wasm states defined by the module to the Wasm store. The process is simple in this particular example, as the module contains only one function definition. If the instantiation is performed starting with an empty store, then the post-instantiation store becomes a store containing a function instance representing the function definition and all other fields empty. The instantiation process also produces an export representing the function with the name “fib”.

Finally, recall that Wasm modules are always embedded in a host environment. The handling and use of the exports now depend on the specific hosts. One typical example of the host is the Wasm-Javascript API [28]. Suppose the binary file of the Fibonacci module is stored in a file fib.wasm. An example of a JavaScript program that interacts with the Fibonacci module is displayed in Figure 2.25, where the JavaScript host loads and instantiates the Wasm module and invokes the exported Fibonacci function iteratively on the input from 1 to 10.

```
WebAssembly.instantiateStreaming(fetch("fib.wasm"), {}).then(
    (obj) => {
        const fib = obj.instance.exports.fib;
        for (let i=1; i<=10;i++) {
            console.log(fib(i));
        }
    }
);
```

Figure 2.25: An example JavaScript code interacting with the Fibonacci module

As demonstrated in Figure 2.25, the instantiation function takes the module binary and provides no import to it (as none is requested). A JavaScript Object is returned as the instantiation result, containing all exports of the module under the `exports` field. The example program above demonstrates how the exported function can then be called in the JavaScript host.

Chapter 3

The WasmCert-Coq Mechanisation for WebAssembly

This chapter describes the WasmCert-Coq [114] mechanisation of WebAssembly. Most of the content in this chapter is written in a way that is agnostic to a specific version of WebAssembly. When any particular part of the WebAssembly semantics is relevant, WebAssembly 1.0 – the version introduced in Chapter 2 – is the version being discussed, unless otherwise stated. At the end of this chapter, a section is included to discuss the update of the targeted version of semantics from Wasm 1.0 to 2.0 and the testing of the most recent version of the mechanisation.

The WasmCert-Coq mechanisation initially drew a core part of its definitions from Watt's Isabelle mechanisation [112] of a pre-1.0 version of the Wasm specification. The Coq mechanisation was later updated for the Wasm 1.0 specification, which was published together with the updated Isabelle mechanisation of Wasm 1.0 [114]. The Isabelle mechanisation of Wasm 1.0 was thoroughly described in Watt's PhD Thesis [111].

In both the Coq and Isabelle/HOL mechanisations, certain design choices were made to use a more forward-compatible design, which agrees with the semantics introduced in the background section Chapter 2, instead of sticking strictly to the 1.0-only features.

3.1 Overview of the WasmCert-Coq Mechanisation

The WasmCert-Coq mechanisation includes a set of mechanised definitions of the operational semantics and type system at its foundation, a mechanised proof of type soundness, a verified executable interpreter, type checker, and module instantiation. In addition, the Coq mechanisation takes a more ambitious approach to also mechanise the numeric operations using the CompCert's verified mechanised integer and float libraries [19, 18], and the binary decoding phases using the Parseque combinator library [6, 5], which were not formalised in the 2017 draft semantics nor mechanised by Watt in the original Isabelle mechanisation, and were not yet handled by WasmCert-Isabelle, except for integer operations which were later implemented by WasmRef-Isabelle [115] in 2023. Instead, the executable produced by Watt's Isabelle mechanisation dele-

	Watt 2018	WasmCert-Isabelle	WasmCert-Coq
Wasm 1.0 refactorings	✗	✓	✓
Wasm 2.0 refactorings	✗	✗	✓
type system	✓✓+	✓✓+	✓✓+
runtime semantics	✓✓+	✓✓+	✓✓+
type soundness proof	✓	✓	✓
binary decoding	✗	✗	✓+
numeric ops	✗	✗ (integers ✓+ [115])	✓✓+
instantiation	✗	✓✓+	✓✓+

Figure 3.1: Comparison between different versions of Wasm mechanisation

gates these operations to Wasm's reference implementation [39].

Figure 3.1 provides a summarised comparison of different versions of the Wasm mechanisations. ✗ indicates a feature or proof which is not included, ✓ marks a feature or proof which has been fully mechanised, ✓✓ indicates that a feature is accompanied by a verified executable definition capable of OCaml extraction, and a+ indicates additionally that the executable definition has been validated through full end-to-end execution of the Wasm test suite (Wasm 2.0 for WasmCert-Coq, Wasm 1.0 for WasmCert-Isabelle).

In the following sections of this chapter, I describe the design of the Coq mechanisation. I will explain in particular detail in the following two aspects:

- The design and implementation choices I have made in the mechanisations that do not correspond entirely with the official specification (in most cases, these choices are made in both the Isabelle and the Coq mechanisations);
- The design choices in the WasmCert-Coq mechanisation that differ from Watt's Isabelle counterpart due to differences between the theorem provers.

3.2 Data Types and Instruction Syntax

The AST of Wasm is defined in Coq using inductive types. Each definition in Figure 2.4, Figure 2.5, and others are represented by inductive types with a set of constructors that correspond to the official specification. The inductive definition of the instruction data type is displayed in Figure 3.2, along with some selected definitions that the instruction type depends on.

I describe several design choices where WasmCert-Coq does not follow the verbatim definitions of the Wasm specification, or where the specification does not indicate a clear choice.

Relaxation of Constraints by Wasm 1.0 Specification The most noteworthy design choice made in both the Isabelle and Coq mechanisations is the omission of restrictions that Wasm 1.0 made on the number of elements allowed for certain types. This is because the Wasm 1.0 specification stated explicitly that these restrictions may be lifted in the future. Therefore, I made the choice to model the more general version of the specification instead of making designs that rely on the temporal restrictions of Wasm 1.0.

```
Inductive value : Type := (* v *)
| VAL_int32 : i32 -> value
| VAL_int64 : i64 -> value
| VAL_float32 : f32 -> value
| VAL_float64 : f64 -> value
.
Inductive value_type : Type := (* t *)
| T_i32
| T_i64
| T_f32
| T_f64
.
Definition immediate (* i *) := N.
Definition result_type : Type := list value_type.
Inductive function_type := (* tf *)
| Tf : result_type -> result_type -> function_type
.

Inductive basic_instruction : Type := (* be *)
| BI_unreachable
| BI_nop
| BI_drop
| BI_select
| BI_block : function_type -> list basic_instruction -> basic_instruction
| BI_loop : function_type -> list basic_instruction -> basic_instruction
| BI_if : function_type -> list basic_instruction -> list basic_instruction -> basic_instruction
| BI_br : immediate -> basic_instruction
| BI_br_if : immediate -> basic_instruction
| BI_br_table : list immediate -> immediate -> basic_instruction
| BI_return
| BI_call : immediate -> basic_instruction
| BI_call_indirect : immediate -> basic_instruction
| BI_get_local : immediate -> basic_instruction
| BI_set_local : immediate -> basic_instruction
| BI_tee_local : immediate -> basic_instruction
| BI_get_global : immediate -> basic_instruction
| BI_set_global : immediate -> basic_instruction
| BI_load : value_type -> option (packed_type * sx) -> alignment_exponent
    -> static_offset -> basic_instruction
| BI_store : value_type -> option packed_type -> alignment_exponent
    -> static_offset -> basic_instruction
| BI_memory_size
| BI_memory_grow
| BI_const : value -> basic_instruction
| BI_unop : value_type -> unop -> basic_instruction
| BI_binop : value_type -> binop -> basic_instruction
| BI_testop : value_type -> testop -> basic_instruction
| BI_relop : value_type -> relop -> basic_instruction
| BI_cvtop : value_type -> cvtop -> value_type -> option sx -> basic_instruction
.
Inductive administrative_instruction : Type := (* e *)
| AI_basic : basic_instruction -> administrative_instruction
| AI_trap
| AI_invoke : funcaddr -> administrative_instruction
| AI_label : nat -> list administrative_instruction ->
    list administrative_instruction -> administrative_instruction
| AI_local : nat -> frame -> list administrative_instruction -> administrative_instruction
.
```

Figure 3.2: Selected WasmCert-Coq definitions for Wasm datatypes

The list of restrictions includes the following:

- The number of tables and memories per module (Wasm 1.0 only allows at most 1). In both mechanisations, a general list of tables and a general list of memories are allowed, without any limit on the lengths;
- The number of values in a return value (Wasm 1.0 only allows at most 1); this is reflected by the fact that the execution result is defined as an optional value, the result type is defined as an optional value type in the Wasm 1.0 semantics, plus well-formedness check on function types in some places that limit the number of values produced to be at most 1. In both mechanisations, the result type is defined as a general list of value types, and no additional validity checks on function types are added. The above has further implications on other parts of the semantics, such as the arity of labels and frames (which can be an arbitrary natural number instead of at most 1); these were handled in the mechanisation as well. The typing rules are generalised to allow lists of value types in the affected places.

In fact, in Wasm 2.0, almost all of the above restrictions have been removed: The restriction on the number of tables was removed by the reference types feature extension, and the restriction on the number of values returned was removed by the multi-value feature extension. The Wasm 2.0 update is described at the end of this chapter in Section 3.10. The only restriction remaining is on the number of linear memories, which will also be removed in the multi-memory proposal[79] in the upcoming version 3.0 of the Wasm semantics.

Administrative Instructions vs Basic Instructions As of Wasm 2.0, the Wasm specification does not make a clear choice here on whether the administrative instructions and the basic instructions should be defined together as one "instruction" type. This is because the administrative instructions are only defined for expressing the intermediate states of the formal runtime semantics; they cannot appear in any Wasm module function code or binaries. However, in some places of the Wasm specification, the instructions being referred to are implicitly restricted to the basic instructions (such as label bodies). Therefore, the Coq mechanisation chooses to follow the design of the Isabelle mechanisation, which defines the type of administrative instructions on top of the type of basic instructions, and introduces a constructor `AI_basic` for the basic instructions.

Another advantage of splitting the subset of basic instructions as a separate definition is that the typing relation for basic instructions $C \vdash es : ft$ can be defined directly, which is the relation required for the Wasm type checker and module validation. In contrast, typing rules for administrative instructions are similarly only defined for the purpose of expressing the soundness property; they are not used in Wasm's validation procedure.

3.3 Operational Semantics

The operational semantics is defined as an inductive relation (`Prop`) between an extended version of Wasm configuration tuples that include a parametric type of host state.

I display the shape of the Coq definition of the operational semantics in Figure 3.3. The main

definition reduce_tuple is a wrapper over the actual definition of the reduction relation reduce, whose arguments are the individual components of the configuration tuples. The reduce relation contains one constructor r_simple, which depends on a reduce_simple relation that includes all reduction rules in the operational semantics that do not depend on the store or the frame.

```
Inductive reduce_simple : list administrative_instruction ->
    list administrative_instruction -> Prop :=
| rs_nop :
    reduce_simple [::AI_basic BI_nop] [::]
| ...
.

Inductive reduce : host_state -> store_record -> frame -> list administrative_instruction ->
                   host_state -> store_record -> frame -> list administrative_instruction -> Prop :=
  | r_simple :
      forall e e' s f hs,
        reduce_simple e e' ->
        reduce hs s f e hs s f e'

  (** calling operations **)
  | r_call :
      forall s f i a hs,
        List.nth_error f.(f_inst).(inst_funcs) i = Some a ->
        reduce hs s f [::AI_basic (BI_call i)] hs s f [::AI_invoke a]
  | ...
  (** structural rules **)
  | r_label :
      forall s f es les s' f' es' les' k (lh: lholed k) hs hs',
        reduce hs s f es hs' s' f' es' ->
        lfill lh es = les ->
        lfill lh es' = les' ->
        reduce hs s f les hs' s' f' les'
  | r_frame :
      forall s f es s' f' es' n f0 hs hs',
        reduce hs s f es hs' s' f' es' ->
        reduce hs s f0 [::AI_frame n f es] hs' s' f0 [::AI_frame n f' es']
.

Definition reduce_tuple cfg cfg' : Prop :=
  let '(hs, s, f, es) := cfg in
  let '(hs', s', f', es') := cfg' in
  reduce hs s f es hs' s' f' es'.
```

Figure 3.3: Summary of the definition of operational semantics in WasmCert-Coq

In the following part of this section, I discuss two non-trivial design choices in the mechanisation of the operational semantics.

3.3.1 Representation of Host Function Invocations

One of the main design choices in the operational semantics is specifying the behaviour of host functions, which is out of the scope of Wasm specification. Recall that the Wasm specification defines the following reduction rule for a host function invocation, where $hf(S;v^n)$ denotes the set of possible results of execution of the host function hf in the current store S with arguments v^n.

$$(S;F;v^n +\!\!+ [\textbf{invoke}\ addr]) \hookrightarrow (S';F;result)$$
$$S.\mathsf{funcs}[a] = \{t_1^n \to t_2^m, hf\} \wedge (S';result) \in hf(S;v^n)$$

A direct way to represent this rule in Coq is to define a parametric type of host functions, and a parametric host function application. However, real world embedders like JavaScript often also have their own states, and the behaviour of the host functions may very well depend on the state of the host as well and potentially modify the host state itself. As a result, an additional parametric type of host states is defined, and the reduce relation is defined over the Wasm configuration tuple augmented with the host state. The implementation can be summarised as follows:

```
Variable host_state: Type.
Variable host_function: Type.
Variable host_function_application: host_state -> store_record -> function_type ->
    host_function -> list value -> host_state * option (store_record * host_result).
```

Additional assumptions are then assumed for the result of host function applications for the soundness property to hold; this will be discussed in Section 3.5.

3.3.2 Formulation of Block Contexts

Another important definition in the operational semantics is the block contexts, which is used in formulating the reduction rules for the control flow instructions **br**/**return**, and the reduction within **label** blocks. Recall Wasm's definition of block contexts $B^k[_]$:

$$
\begin{aligned}
B^0[_] &= vs ++ [_] ++ es \\
B^{k+1}[_] &= vs ++ \textbf{label}_n \; \{es_{\mathsf{cont}}\} \; B^k[_] \; \textbf{end} \; ++ es
\end{aligned}
$$

A block context $B^k[_]$ contains precisely k nested labels; this is used in expressing the reduction rule for the branch instruction **br** i, which targets the ith label from the innermost label.

As Isabelle/HOL doesn't support dependent types, Watt's Isabelle mechanisation defined the type of block context to only include information of its structure while omitting the depth. Separately, Watt defined a relation lfilled that not only associates a block context and an instruction list in the hole to the instruction list obtained from filling the instruction list into the hole of the block context, but also includes an additional argument that represents the depth of the block context. The relation is defined in a way that ensures that this extra argument always agrees with the actual depth of the context.

The Coq mechanisation initially ported the design of the Isabelle mechanisation, which is summarised in Figure 3.4a.

However, this deviation from the Wasm specification is unnecessary in the Coq mechanisation, since Coq has sufficient support for dependent types to represent the definition in Wasm's specification faithfully. Therefore, the type of the block context is instead defined as a dependent type on the depth of the block, and a standard `Fixpoint` can be defined to represent the fill operation for the block context, without needing an adapted relation to represent the depth of the context. This results in an overall simplification of the mechanisation, as there is no longer a requirement to deal with two separate formulations of the context predicate and the conversions between them. This is displayed in Figure 3.4b.

```
Inductive lholed : Type :=
| LH_base : list administrative_instruction -> list administrative_instruction -> lholed
| LH_rec : list administrative_instruction -> nat -> list administrative_instruction ->
    lholed -> list administrative_instruction -> lholed
.

Inductive lfilledInd : nat -> lholed -> list administrative_instruction ->
    list administrative_instruction -> Prop :=
| LfilledBase: forall vs es es',
    const_list vs ->
    lfilledInd 0 (LH_base vs es') es (vs ++ es ++ es')
| LfilledRec: forall k vs n es' lh' es'' es LI,
    const_list vs ->
    lfilledInd k lh' es LI ->
    lfilledInd (k.+1) (LH_rec vs n es' lh' es'') es (vs ++ [ :: (AI_label n es' LI) ] ++ es'').
```

(a) Representation of block contexts following Watt's design choices in Isabelle/HOL

```
Inductive lholed : nat -> Type :=
| LH_base : list value -> list administrative_instruction -> lholed 0
| LH_rec {k: nat}: list value -> nat -> list administrative_instruction ->
    lholed k -> list administrative_instruction -> lholed (S k)
.

Fixpoint lfill {k} (lh : lholed k) (es : list administrative_instruction)
    : list administrative_instruction :=
  match lh with
  | LH_base vs es' => v_to_e_list vs ++ es ++ es'
  | LH_rec _ vs n es' lh' es'' => v_to_e_list vs ++ [:: AI_label n es' (lfill lh' es)] ++ es''
  end.
```

(b) Representation of block contexts using dependent types in Coq

Figure 3.4: Comparison between the implementations of block contexts in Watt's Isabelle mechanisation and the Coq mechanisation

3.4 Verified Interpreter

The WasmCert-Coq mechanisation includes an interpreter of Wasm, which is an executable version of Wasm's operational semantics. The structure of the initial version of the interpreter introduced in this section follows from Watt's Isabelle mechanisation and Wasm's reference interpreter. I describe the structure of the interpreter and some relevant non-trivial details in its implementation, the correctness property I proved for the interpreter, and lastly the extraction of the verified interpreter to an executable program in OCaml.

The recursive call via the run_step function, which is responsible for splitting up the instruction list into the operand stack and instruction stack, and will be introduced in the next subsection.

3.4.1 Structure of the Interpreter

The core definition of the interpreter in Coq is the one-step interpreter, which is defined by a mutual recursion between run_step and run_one_step, implemented as a mutual `Fixpoint` in Coq.

run_step is a small function whose main job is to split the instruction list in a Wasm configuration into three parts: the operand stack, the top instruction on the instruction stack (if non-empty), and the rest of the instruction stack. If there is no instruction left to execute, a result corresponding to the operand stack is returned. Otherwise, the run_one_step function is invoked, which contains the main logic of the interpreter. The goal of this function is to perform a *one-step* execution for the top instruction on the instruction stack and return the execution result. Several possible outcomes are defined for the execution result:

- RS_crash *error*, which represents a crash either due to a trap or other reasons;
- RS_normal *es*, which represents a normal execution result and returns a list of administrative instructions (potentially empty) for the execution result of the top instruction on the instruction stack;
- RS_break *k bvs*, which represents an active **br** instruction currently returning from labels;
- RS_return *rvs*, which represents an active **return** instruction currently returning from labels.

The structure of the one-step interpreter and its relevant definitions are displayed in Figure 3.5.

Implementing the structured control flows of Wasm

I explain the RS_break and RS_return in more detail. Recall that execution of a **br** *k* instruction requires selecting the *k*th label in the environment as the branch target, and takes an appropriate number of values (equal to the arity of the target label) from the top of the current operand stack. However, a direct implementation of this requires knowledge of all the outer labels in the runtime representations.

Instead, the one step interpreter takes a *bubbling-up* approach to execute the **br** and **return** instructions. Whenever a **label** instruction is at the top of the execution stack, the one-step interpreter

```
Inductive res_step : Type :=
| RS_crash : res_crash -> res_step
| RS_normal : list administrative_instruction -> res_step
| RS_break : nat -> list value -> res_step
| RS_return : list value -> res_step.

Definition res_tuple := (host_state * store_record * frame * res_step).

Fixpoint run_step (cfg : config_tuple) : res_tuple :=
  let: (hs, s, f, es) := cfg in
  let: (ves, es') := split_vals_e es in (** Framing out the operand stack. **)
    match es' with
    | [::] => (hs, s, f, crash_error)
    | e :: es'' =>
      if e_is_trap e
      then
        if (es'' != [::]) || (ves != [::])
        then (hs, s, f, RS_normal [::AI_trap])
        else (hs, s, f, crash_error)
      else
        let: (hs', s', f', r) := run_one_step (hs, s, f, (rev ves)) e in
        if r is RS_normal res
        then (hs', s', f', RS_normal (res ++ es''))
        else (hs', s', f', r)
    end
with run_one_step (cfg: host_state * store_record * frame * list value)
    (e: administrative_instruction): res_tuple :=
let: (hs, s, f, ves) := cfg in
match e with
    (* unop *)
    | AI_basic (BI_unop t op) =>
      if ves is v :: ves' then
        (hs, s, f, RS_normal (vs_to_es (app_unop op v :: ves')))
      else (hs, s, f, crash_error)
    | ...
end.
```

Figure 3.5: Structure of the WasmCert-Coq one-step interpreter and relevant definitions

```
Fixpoint run_one_step (cfg: host_state * store_record * frame * list value)
    (e: administrative_instruction): res_tuple :=
let: (hs, s, f, ves) := cfg in
    match e with
    | ...
    | AI_basic (BI_br j) => (hs, s, f, RS_break j ves)
    | AI_label ln les es =>
      if es_is_trap es
      then (hs, s, f, RS_normal (vs_to_es ves ++ [::AI_trap]))
      else
        if const_list es
        then (hs, s, f, RS_normal (vs_to_es ves ++ es))
        else
          let: (hs', s', f', res) := run_step (hs, s, f, es) in
          match res with
          | RS_break 0 bvs =>
            if length bvs >= ln
            then (hs', s', f', RS_normal ((vs_to_es ((take ln bvs) ++ ves)) ++ les))
            else (hs', s', f', crash_error)
          | RS_break (n.+1) bvs => (hs', s', f', RS_break n bvs)
          | RS_return rvs => (hs', s', f', RS_return rvs)
          | RS_normal es' =>
            (hs', s', f', RS_normal (vs_to_es ves ++ [::AI_label ln les es']))
          | RS_crash error => (hs', s', f', RS_crash error)
          end
    end
end.
```

Figure 3.6: Implementation of the one-step interpreter for **br** and **label**

performs a recursive call which executes the body of the label under the same store and frame. If the label body returns a normal execution result (without encountering a **br** or **return**, and without producing a trap), then the one-step interpreter returns a new **label** with the updated body, as the operational semantics specified. The interesting case is when a **br** instruction is encountered in the body of some **label**. The one-step interpreter will return a RS_break result, including information of the *entire* operand stack, and the index of the target label with respect to the current environment. During the backtracking, when a **label** is encountered, if $k > 0$, then the interpreter simply exits from the label, and subtracts 1 from the index k; this continues until a **label** instruction is encountered when $k = 0$, in which case the target label is reached. In this case, the top part of the returned operand stack with length equal to the arity of the label, together with the continuation of the label, are returned as the execution result of the current label.

The **return** instruction is executed similarly: in the bubbling-up version of execution, it is essentially exiting from an infinite number of **labels** until a **frame** instruction is reached, after which the execution exits from the **frame** and returns an appropriate number of values from the original operand stack of the **return** instruction.

Figure 3.6 demonstrates the interaction between the **label** instruction and the returning control flow instructions (**br** and **return**).

However, this implementation of the labels and branching execution, inherited from Watt's Isabelle mechanisation and Wasm's reference interpreter, is very slow when executing a program with deeply nested labels. This is because at every step of execution, the one-step interpreter needs to

recursively step into each label until the top instruction of a label is no longer a label, performs a one-step execution of the top instruction, and backtracks to the outermost label and returns the execution result. While this is a faithful representation to the Wasm's operational semantics, a more efficient runtime representation is required for executing real-world Wasm binaries. In Section 4.3.3, an optimised runtime configuration is discussed as part of a broader study of the different design of the interpreter, which avoids this inefficiency altogether.

Termination Check for the Interpreter

Note that the mutual Fixpoint defined in Figure 3.5 is quite complex; in particular, the implementation as displayed does not satisfy Coq's termination check for Fixpoints. Indeed, the functions without the fuel are also guaranteed to terminate, since they together form a recursion on the complexity of the instruction list, which strictly decreases at each recursion (due to entering a **label** or a **frame**). However, Coq is unable to reach this conclusion by itself.

Therefore, in the Coq mechanisation, both `Fixpoint`s of the mutual recursion take an additional *fuel* as a parameter; one unit of fuel is consumed at the start of each recursion. To ensure that the introduction of the fuel does not have impact on the execution of the one-step interpreter, a separate fuel-computation function is defined which computes an upper bound of the fuel that is sufficient to guarantee that the interpreter will never run into the exhaustion error, which is separately proved as part of the correctness result of the interpreter.

Nevertheless, the optimised runtime representation to be discussed in Section 4.3.3 keeps track of a separate *context* stack. As a result, it does not suffer from this issue, since no recursive call is required for the execution of the one-step interpreter.

3.4.2 Proof of Interpreter Correctness

In this subsection, let eval_1 be the one-step interpreter function that represents the mutual fixpoints discussed earlier, which takes the components of the Wasm configuration as an input and produces one of the several possible execution outcomes. For simplicity, the host state in the extended configuration is omitted.

The main correctness result proved in the mechanisation is the following statement:

Proposition 3.4.1. *If*

$$\mathsf{eval}_1(S;F;es) = (S';F';\mathsf{RS_normal}\ es')$$

then

$$(S;F;es) \hookrightarrow (S';F';es').$$

Similar to the core of the one-step interpreter being a case analysis on the top instruction on the instruction stack, the main part of the proof of correctness is a large case analysis by the top instruction on the instruction stack as well. For a RS_normal result to be produced by the one-step interpreter, the side-conditions required for the execution need to be satisfied; this then provides sufficient premises for the corresponding reduction relation in the operational semantics to be

constructed. Most of these cases are straightforward.

The interesting cases of the proof are the ones involving the structured control flow instructions, such as **br** and **label**. Note that although Proposition 3.4.1 only states that the corresponding reduction is satisfied upon a normal execution result, *some* propositions for the RS_break and RS_return results need to be established, because they can be intermediate results to the production of an RS_normal result during further backtracking out of **label**s.

The main lemmas regarding them are the following two, one for each outcome:

Lemma 3.4.2. *If*

$$\mathsf{eval}_1(S;F;es) = (S';F';\mathsf{RS_break}\ k\ ves)$$

then $S' = S$, $F' = F$, and es can be decomposed in the shape of

$$es = B^i[\mathsf{rev}(ves) +\!\!+ [\mathbf{br}\ (k+i)]]$$

for some $i \in \mathbb{N}$. Furthermore, if $k = 0$, then the above implies that, for all $n \le |ves|$ and arbitrary continuation es_{cont},

$$(S;F;[\mathbf{label}_n\ \{es_{\mathsf{cont}}\}\ es\ \mathbf{end}]) \hookrightarrow (S';F';\mathsf{rev}(ves[0:n]) +\!\!+ es_{\mathsf{cont}}).$$

Lemma 3.4.3. *If*

$$\mathsf{eval}_1(S;F;es) = (S';F';\mathsf{RS_return}\ ves)$$

then $S' = S$, $F' = F$, and es can be decomposed in the shape of

$$es = B^i[\mathsf{rev}(ves) +\!\!+ [\mathbf{return}]]$$

for some $i \in \mathbb{N}$. Furthermore, the above implies that, for all $n \le |ves|$ and arbitrary frame and F_0,

$$(S;F_0;[\mathbf{frame}\ \{F\}\ es\ \mathbf{end}]) \hookrightarrow (S';F_0;\mathsf{rev}(ves[0:n])).$$

recalling the notation $l[i:n]$ representing the sublist of the list l starting at index i (0-indexed) with length n.

The above two lemmas state the correctness of the RS_break and RS_return outcomes by stating the concrete shape of any program that can evaluate to RS_break 0/RS_return in one step and proving their behaviour when the corresponding input programs are plugged into the body of a corresponding **label**/**frame** block.

For example, if an instruction list *es* evaluates to a RS_break 0 *ves* result, then when the instruction list *es* is plugged in as the body of any **label** block with an arity not exceeding the number of values carried in the evaluation result *ves*, the entire label will reduce to a result according to the reduction semantics as if the body of the label contains a **br** 0 at the top of its instruction stack and *ves* as its operand stack.

The proofs to the above lemmas themselves can be done by a structural induction on the input instruction[1]. The case analyses are relatively simple in both of the proofs, because in the one-step interpreter, there are only two cases where each of the RS_break/RS_return outcomes can be generated: one from the **br**/**return** instruction themselves, and the other from backtracking from **label** instructions whose bodies evaluate to these outcomes (for the case of RS_break, the label index reduces by 1 at each step). As a result, a straightforward induction is sufficient to prove the above lemmas (although the mechanisation details are quite tedious).

3.5 Type System and Soundness

The core of the type system is an inductive relation be_typing among the typing context, the (basic) instruction list, and the function type. This is the typing relation used for Wasm's validation, since only basic instructions can appear in Wasm modules. A separate typing relation e_typing is defined for the typing of administrative instructions, which takes the store as an additional argument. As introduced in Section 2.5, this typing relation is defined in mutual recursion with the thread typing relation thread_typing. Besides the typing relation for instructions, the typing relation for module instances, frames, and the well-formedness condition of the store, and the typing for configurations are defined as straightforward `Definition`s in Coq corresponding to the specification, building on top of each other. The structures of these definitions are illustrated in Figure 3.7.

Having defined the typing relations for configurations, the type soundness property of Wasm defined in Section 2.6 can be expressed in the mechanisation. This is stated using the preservation and progress properties in Figure 3.8, including the definition of the terminal states, which are precisely when the instruction list is a list of values or a single trap.

Recall that the Wasm specification merely describes the statements of the soundness properties without the full proof details, although it has outlined enough auxiliary definitions for a structure of the proof to be sketched, such as the store extension relation. Watt's initial mechanisation in Isabelle [112] includes a mechanised proof of the type soundness property. The idea of the proofs in the WasmCert-Coq mechanisation should be similar, although no explicit comparison has been done against the Isabelle proofs.

The following two subsections describe the main structure of the proofs and several key auxiliary lemmas described in words. For the detailed implementation, refer to the Coq mechanisation [109].

3.5.1 Preservation

The proof of the preservation property is done by induction over the reduction relation from the operational semantics and a case analysis over all the reduction rules.

Recall from Section 2.5 that the configuration typing relation requires the store to be valid, and the thread to be well-typed under the store with some result type, where the thread typing relation

[1] The last version of Coq mechanisation including this proof used an induction on the fuel instead due to the existence of the fuel in the mechanised one-step interpreter; however, the main idea is equivalent.

```
Inductive be_typing : t_context -> list basic_instruction -> function_type -> Prop :=
| bet_nop : forall C, be_typing C [::BI_nop] (Tf [::] [::])
| bet_drop : forall C t, be_typing C [::BI_drop] (Tf [::t] [::])
| bet_select : forall C t, be_typing C [::BI_select] (Tf [::t; t; T_i32] [::t])
| bet_block : forall C tn tm es,
  let tf := Tf tn tm in
  be_typing (upd_label C (app [::tm] (tc_label C))) es (Tf tn tm) ->
  be_typing C [::BI_block tf es] (Tf tn tm)
| ...
.
Inductive e_typing : store_record -> t_context ->
    list administrative_instruction -> function_type -> Prop :=
| ety_a : forall s C bes tf,
  be_typing C bes tf -> e_typing s C (to_e_list bes) tf
| ety_composition : forall s C es e t1s t2s t3s,
  e_typing s C es (Tf t1s t2s) ->
  e_typing s C [::e] (Tf t2s t3s) ->
  e_typing s C (es ++ [::e]) (Tf t1s t3s)
| ...
with thread_typing : store_record -> option result_type ->
    thread -> result_type -> Prop :=
| mk_thread_typing : forall s f es rs ts C C',
  frame_typing s f C ->
  C' = upd_return C rs ->
  e_typing s C' es (Tf nil ts) ->
  thread_typing s rs (f, es) ts
.

Definition frame_typing (s: store_record) (f: frame) (C: t_context): Prop := ... .
Definition store_typing (s : store_record) : Prop :=
  match s with
  | Build_store_record fs tabs ms gs es ds =>
    (List.Forall (fun x => exists t, funcinst_typing s x t) fs) /\
    (List.Forall (fun x => exists t, tableinst_typing s x = Some t) tabs) /\
    (List.Forall (fun x => exists t, meminst_typing s x = Some t) ms) /\
    (List.Forall (fun x => exists t, globalinst_typing s x = Some t) gs)
  end.
Definition config_typing (s: store_record) (th: thread) (ts: result_type) : Prop :=
  store_typing s /\ thread_typing s None th ts.
```

Figure 3.7: Structure of typing relations for instructions and configurations of Wasm 1.0

```
Theorem t_preservation: forall s f es s' f' es' ts hs hs',
    reduce hs s f es hs' s' f' es' ->
    config_typing s f es ts ->
    config_typing s' f' es' ts.

Definition const_list (es : seq administrative_instruction) : Prop := ... .
Definition terminal_form (es: seq administrative_instruction) :=
  const_list es \/ es = [::AI_trap].
Theorem t_progress: forall s f es ts hs,
    config_typing s f es ts ->
    terminal_form es \/
    exists s' f' es' hs', reduce hs s f es hs' s' f' es'.
```

Figure 3.8: Statements of the type soundness property in the WasmCert-Coq mechanisation for Wasm 1.0

is based on the instruction typing relation $S;C \vdash e : ft$.

Therefore, the proof of preservation has two parts: proving that the new store is also valid, and that the new instruction list has the same type under the new context.

Store validity Recall the store extension relation from Section 2.6. Note that S' being an extension of a valid store S does not automatically guarantee that the store S' is valid: for example, the contents of tables and memories might have grown beyond their maximum size limits. However, in Wasm 1.0, these are also the only two cases where the extension of a valid store could potentially become invalid. This is because no Wasm 1.0 instructions could directly allocate new state instances in the store, the function instances have to stay the same as per the store extension relation, and the global instances are automatically valid in Wasm 1.0. In addition, no Wasm 1.0 instruction could modify tables. Therefore, under the assumption that the new store is an extension of the old store, I only need to prove that the memory-related operations do not expand the size of the memory beyond their maximum limits, which is true since the only instruction that grows the memory is **memory**.**grow**, which has the correct checks encoded in its execution semantics.

Store extension As for proving that the new store is indeed a store extension of the old store itself, all reductions that modify the store need to be considered. In the mechanisation, a lemma has been proved for each case where the store is modified by an instruction. This includes **store**, **memory**.**grow**, **global**.**set**, and all the structural rules (such as the rules for **label**s and **frame**s) that propagate the execution of their bodies.

Assumptions on Host functions As described in Chapter 2, Wasm does not specify any concrete semantics for host function invocations. However, several conditions on the results and effects of host function calls are required for the preservation property to hold. Concretely, for the following host invocation

$$\frac{S.\mathsf{funcs}[a] = \{ts_1 \to ts_2, \ldots\}^{\mathsf{Host}} \quad \mathsf{typeof}^*(vs_1) = ts_1}{(S;F;vs_1 \mathbin{+\!\!+} [\textbf{invoke } a]) \hookrightarrow (S';F;vs_2)} \ \mathsf{invoke_host}$$

the following properties are required to hold:

- The new store S' is a valid store;
- The new store S' is a store extension of the original store S;
- The values returned, vs_2, match the expected types ts_2.

Instruction typing One crucial design used in proving the instruction typing is a large set of *typing inversion lemmas*. These lemmas describe the set of constraints that the different arguments of a typing relation need to satisfy for the typing relation to hold. These constraints are then used to construct the typing relation for the new instruction list under the new context. These lemmas are proved for each instruction at the top of the instruction stack: I display several typing inversion

lemmas in Figure 3.9.[2]

$$\begin{aligned}
&(\text{empty})\ \forall ts_1, ts_2. && (\vdash [] : ts_1 \to ts_2) \implies (ts_1 = ts_2)\\
&(\text{local.get})\ \forall C, ts_1, ts_2. && (C \vdash [\textbf{local.get}\ j] : ts_1 \to ts_2) \implies (\exists t. C.\text{locals}[j] = t \land ts_2 = ts_1 +\!\!+\ [t])\\
&(\text{composition})\ \forall es_1, es_2, ts_1, ts_2. && (\vdash es_1 +\!\!+\ es_2 : ts_1 \to ts_2) \implies (\exists ts_3. (\vdash es_1 : ts_1 \to ts_3) \land (\vdash es_2 : ts_3 \to ts_2))\\
&(\text{values})\ \forall vs, ts_1, ts_2. && (\vdash vs : ts_1 \to ts_2) \implies (ts_2 = ts_1 +\!\!+\ (\text{typeof}\ vs))
\end{aligned}$$

Figure 3.9: Selected Typing Inversion Lemmas

This design is especially efficient for automating a large part of the preservation proofs and any properties that take a typing relation as part of their premises. Given a typing relation, I designed in Coq an `Ltac` tactic that repeatedly attempts to apply the typing inversion lemmas based on the shape of the instruction list in the typing relation. In most cases, the tactic is able to completely consume all of the typing relations and produce all the constraints to the proof environment which are available for later use.

Note that for the above `Ltac` automation design to work, the typing inversion lemmas must not lose any important information in their implications, since the `Ltac` automation needs to consume the original typing relation (otherwise, the tactic will be stuck in an infinite loop trying to apply the same typing inversion lemmas for the same typing relations). In fact, most of the implications proved in the typing inversion lemmas are equivalences – i.e. the constraints produced are not only necessary, but also sufficient conditions to derive the original typing relation. In particular, this is the case for all the selected lemmas displayed in Figure 3.9.

The following proof illustrates the one example case of the preservation property for **local.get**.

Proposition 3.5.1. *Given*

$$(S; F; [\textbf{local.get}\ j]) \hookrightarrow (S; F; [v])$$

and

$$S \vdash F : C, C \vdash [\textbf{local.get}\ j] : ts_1 \to ts_2$$

then

$$C \vdash [v] : ts_1 \to ts_2.$$

Proof. The reduction rule for **local.get** is:

$$\frac{F.\text{locals}[j] = v}{(F; [\textbf{local.get}\ j]) \hookrightarrow (F; [v])}\ \text{local.get}$$

From the definition of the frame typing relation and $S \vdash F : C$, I would obtain that

$$\forall i. C.\text{locals}[i] = \text{typeof}(F.\text{locals}[i])$$

[2] In the figure, the store and context are omitted for rules that do not explicitly refer to them.

Now, by the typing inversion lemma, I have

$$\exists t. C.\mathsf{locals}[j] = t \land ts_2 = ts_1 \mathbin{++} [t]$$

Substituting the above into the goal and apply the subsumption rule (recall Figure 2.18), I now only need to prove that

$$C \vdash [v] : [] \to [t]$$

where t is the witness for the typing inversion lemma above. This can be established since

$$t = C.\mathsf{locals}[j] = \mathsf{typeof}(F.\mathsf{locals}[j]) = \mathsf{typeof}(v).$$

□

The preservation property for the cases of reductions without store modifications are directly provable by the above design.

The reductions that involve store or frame modifications require several additional auxiliary lemmas, since it is not obvious that the typing context C given by the frame typing relation $S \vdash F : C$ will not change when the store S if modified.

I briefly describe the lemmas I prove in the mechanisation to facilitate the proofs to these cases.

Lemma 3.5.2. *If*

$$S \vdash F : C, \quad S \preccurlyeq S', \quad \vdash S : \mathsf{ok}$$

then

$$S' \vdash F : C.$$

The above lemma allows to prove that the parts of the typing context C corresponding to the frame F stays invariant throughout the execution, i.e.

Lemma 3.5.3. *If*

$$(S; F; es) \hookrightarrow (S'; F'; es'), \quad S \vdash F : C, \quad \vdash S : \mathsf{ok}$$

then

$$S' \vdash F' : C.$$

The proof of the second lemma requires an induction on the reduction relation. The case analysis is relatively easy, since the module instance of the frame is immutable, and the local variables can only be modified by the **local**.**get** instruction (aside from the **label** structural rule), which cannot change the type of the local variables.

The above two lemmas along with the previous typing inversion design are sufficient to prove all the non-inductive cases of the preservation property. For the inductive cases of the preservation proof – corresponding to the structural reduction rules for **label** and **frame**, several additional lemmas are required to reason about the typing of program fragments under store extensions. The

two most important lemmas required are the following:

Lemma 3.5.4. *If*

$$(S;F;es) \hookrightarrow (S';F';es'), \quad S \vdash F : C, \quad S;C \vdash es : tf, \quad \vdash S : \mathsf{ok}$$

then

- $\vdash S' : \mathsf{ok}$;
- $S \preccurlyeq S'$.

This lemma states that any reduction from a valid tuple under a valid store results in a new valid store which is an extension of the old store. This is again proved by induction over the reduction relation. The case analysis is extremely tedious (and accounts for almost half the LOC of the preservation proof): for each instruction that modifies the store, the validity and the extension relations need to be established.

Lemma 3.5.5. *If*

$$S;C \vdash es : tf, \quad \vdash S : \mathsf{ok}, \quad \vdash S' : \mathsf{ok}, \quad S \preccurlyeq S'$$

then

$$S';C \vdash es : tf.$$

This lemma states that a well-typed program fragment is still well-typed (and of the same function type) under a valid store extension.

The above lemmas provide enough tools to discuss the case of the preservation property for the **label** instruction that represents a step in its body, which is the most difficult case.

Proposition 3.5.6. *Given a reduction*

$$(S;F;[\mathbf{label}_n\ \{es_{\mathsf{cont}}\}\ es\ \mathbf{end}]) \hookrightarrow (S';F';[\mathbf{label}_n\ \{es_{\mathsf{cont}}\}\ es'\ \mathbf{end}])$$

Let C' be a typing context corresponding to the store S and frame F, with arbitrary labels ls_{label} on the context

$$S \vdash F : C, \quad C' = C \textit{ with } \mathsf{labels} := ls_{\mathsf{label}}$$

and the typing and validity relations

$$S;C' \vdash [\mathbf{label}_n\ \{es_{\mathsf{cont}}\}\ es\ \mathbf{end}] : ts_1 \to ts_2, \quad \vdash S : \mathsf{ok}$$

with the following induction hypothesis on the label body es:

$$\begin{aligned} \forall ft, C'', ls'_{\mathsf{label}}.\quad & C'' = C \textit{ with } \mathsf{labels} := ls'_{\mathsf{label}} \wedge \\ & (S;F;es) \hookrightarrow (S';F';es') \wedge \\ & S;C'' \vdash es : ft \\ & \implies S';C'' \vdash es' : ft. \end{aligned}$$

then

$$S';C' \vdash [\textbf{label}_n\ \{es_{\mathsf{cont}}\}\ es'\ \textbf{end}] : ts_1 \rightarrow ts_2.$$

Note that the same context C' is used for typing the original program fragment and the execution result; this is derived by the lemma that the typing context C is invariant throughout execution.

Proof. Recall the following typing rule for **label** from Figure 2.16, extended with the store S:

$$\frac{S;C \vdash es_{\mathsf{cont}} : t_1^n \rightarrow rt \quad S;(C \text{ with } \mathsf{labels} = [t_1^n] +\!\!+\ C.\mathsf{labels}) \vdash es : [] \rightarrow rt}{S;C \vdash [\textbf{label}_n\ \{es_{\mathsf{cont}}\}\ es\ \textbf{end}] : [] \rightarrow rt}\ \mathsf{label}$$

The typing inversion lemma for **label** provides a generalised version of the premises in the label rule above:

$$\begin{aligned}
&S;C' \vdash [\textbf{label}_n\ \{es_{\mathsf{cont}}\}\ es\ \textbf{end}] : ts_1 \rightarrow ts_2 \implies \\
&\exists rt, ts_{\mathsf{label}}.\quad ts_2 = ts_1 +\!\!+\ rt\ \wedge \\
&\qquad\qquad |ts_{\mathsf{label}}| = n\ \wedge \\
&\qquad\qquad S;C' \vdash es_{\mathsf{cont}} : ts_{\mathsf{label}} \rightarrow rt\ \wedge \\
&\qquad\qquad S;(C' \text{ with } \mathsf{labels} = [ts_{\mathsf{label}}] +\!\!+\ C'.\mathsf{labels}) \vdash es : [] \rightarrow rt
\end{aligned}$$

This is because the subsumption typing rule can be applied for a **label** construct.

The given reduction inverts to a reduction of the body, i.e.

$$(S;F;es) \hookrightarrow (S';F';es')$$

To prove the goal, apply the subsumption rule (framing ts_1 off the function type) followed by the **label** rule, which reduce the proof obligation to proving that both the continuation and the new body has the correct types. Concretely, the two proof obligations are as follows:

- For the continuation, I need to prove that

  $$S';C' \vdash es_{\mathsf{cont}} : ts_{\mathsf{label}} \rightarrow rt$$

  for some ts_{label} with length n. The witness ts_{label} obtained from the typing inversion lemma is used. The typing inversion lemma provides the desired typing relation in the old store S; therefore, it is sufficient to apply the lemma that establishes the preservation of function types for program fragments under valid store extensions.

- For the body, I need to prove that, for some t_1^n,

  $$S';(C' \text{ with } \mathsf{labels} = [t_1^n] +\!\!+\ C'.\mathsf{labels}) \vdash es : [] \rightarrow rt$$

  The typing inversion lemma similarly already produces a corresponding typing relation under the old store S. Therefore, this case is proved similarly with the same lemmas applied in the case above.

□

The case for the **frame** instruction is similar but easier, since there is no continuation to reason about.

3.5.2 Progress

The proof of the progress property is done by inducting over the typing relation for program fragments $S;C \vdash es : ft$.

The proof is considerably less tedious than the preservation proof. In fact, the main challenge is to correctly formulate the progress property for the fragment typing relation. Although the original progress property states that every well-typed non-terminal program can be reduced further, a well-typed program fragment may require operands according to its function type to execute. For example, the following typing relation for a **binop** instruction holds:

$$\vdash [t.\mathbf{binop}\ binop] : [t;t] \to [t]$$

but the **binop** instruction cannot be executed on its own; instead, two values of type $[t;t]$ are required on the operand stack. In addition, there are other types of program fragments which cannot be executed on their own even when provided with the correct types of operands, such as **br** and **return**: the execution of these control flow instructions require the corresponding enclosing **label** and **frame** contexts as specified by the typing context C.

The progress property for fragment typing relation I formulate and prove in the mechanisation is as follows:

Proposition 3.5.7 (Fragment progress)**.** *For arbitrary ls and rt, if*

$$S \vdash F : C, \quad \vdash S : \mathsf{ok}, \quad C' = (C\ \mathit{with}\ \mathsf{labels} := ls,\ \mathsf{return} := rt)$$

with the typing relation

$$S;C' \vdash es : ts_1 \to ts_2$$

and the program fragment is not a **br** *or* **return** *without sufficient contexts, i.e.*

$$\forall B,n,k.\ es = B^n[\mathbf{br}\ k] \implies k < n$$

and

$$\forall B,n.\ \neg(es \neq B^n[\mathbf{return}])$$

then one of the following holds:

$$\begin{array}{ll} (1) & \mathsf{terminal}(es); \\ (2) & \forall vs.\,[\mathsf{typeof}(vs) = ts_1 \implies \exists S',F',es'.(S;F;vs +\!\!+\ es) \hookrightarrow (S';F';es')]\,. \end{array}$$

I first explain why this formulation of fragment progress property implies the progress property for

the configuration tuple. Recall the configuration typing rule and its related rules from Figure 2.21:

$$\frac{S \vdash F : C \quad S;C \text{ with return} = rt \vdash es : [] \to ts}{S; rt \vdash_{\mathsf{th}} (F; es) : ts} \ \mathsf{thread}$$

$$\frac{\vdash_{\mathsf{s}} S : \mathsf{ok} \quad S;[] \vdash (F;es) : rt}{\vdash_{\mathsf{cfg}} (S;F;es) : rt} \ \mathsf{configuration}$$

The configuration typing relation requires the body to contain the full operands and contexts required for execution (which is sensible, as the configuration body represent the *entire* program), since the thread typing relation uses a context C without any labels (so any br instruction will need to be enclosed by sufficient number of **label**s to be well-typed), and the configuration typing rule asserts that no return type is allowed (so any return instruction in the program will not be well-typed unless enclosed by a **frame** instruction). Therefore, the conditions of the fragment progress property can be satisfied. Since the thread typing rule asserts that the program does not consume any operand types, the fragment progress property will provide a reduction path without consuming any additional values.

The case analysis in the proof of progress property is mostly straightforward using the same automation with typing inversion lemmas to extract the conditions from the typing relations. In the cases where a reduction is produced, the condition in the goal (2) that a list of values of the correct types are produced provides the appropriate operands.

The only difficult cases are the cases regarding **br** and **return**. Due to the premises, these instructions can only appear within the correct **label** or **frame** constructs. As a result, it only requires some technical lemmas to prove that when they do appear, they are also accompanied by sufficient values to be taken out to fulfill the arity of the **label** or **frame** targets. I state the two lemmas I proved for this purpose:

Lemma 3.5.8. *If*

$$es = B^n[\mathbf{br}\ (n+k)]$$

with the typing relation

$$S;C \vdash es : [] \to ts, \quad C.\mathsf{labels}[k] = ts_{\mathsf{label}}$$

then there exists a decomposition of e:

$$\exists vs, B'.\ es = B'^n[vs +\!\!+\ [\mathbf{br}\ (n+k)]]$$

with

$$|vs| = |ts_{\mathsf{label}}|.$$

Lemma 3.5.9. *If*

$$es = B^n[\mathbf{return}]$$

with the typing relation

$$S;C \vdash es : [] \to ts, \quad C.\mathsf{return} = \mathsf{Some}\ ts_{\mathsf{return}}$$

then there exists a decomposition of es:

$$\exists vs, B'. es = B'^n[vs +\!\!+ [\mathbf{return}]]$$

with

$$|vs| = |ts_{\mathsf{return}}|.$$

In fact, a stronger property regarding the operands vs can be proven (their types have to be equal to the corresponding $ts_{\mathsf{label/return}}$), but this is not necessary for proving the progress property.

These two lemmas are proved by inducting on the depth of the block context n. The main logic lies in the case where $n = 0$, where the program fragment is a **br** or **return** instruction on top of the instruction stack accompanied with some values in the operand stack and some further instructions. The typing rules for br and return (recall from Figure 2.16 and Figure 2.17 respectively) then allow to establish that the values corresponding to the types must exist on the operand stack.

3.6 Verified Type Checker

The type system described in Section 2.4 define the conditions required to be satisfied for a program fragment to be valid. This is in turn used in the module validity conditions described in Section 2.7.1, where the function types of function bodies, initialiser expressions and offset expressions are type checked. Therefore, the validation of Wasm module requires an executable version of the type system that checks whether a program fragment can be typed with a certain function type ft under a given context, i.e.

$$C \vdash es : ft$$

Note that the version without the store is used (corresponding to be_typing in the mechanisation): this is because, as mentioned previously, no administrative instruction can appear in the actual module definitions.

Most of the Wasm instructions carry their function type annotations. Therefore, the type checker for most instructions are straightforward to implement, and can be done in one pass of the program by simply keeping track of the cumulative types of operands on the stack and working through the effect of the current instruction on the stack. For an instruction of function type $\vdash e : ts_1 \to ts_2$, the type checker first tries to consume values of type ts_1 on the current stack; if there is a mismatch between the top of the stack and the types to be consumed, then the type checker fails; otherwise, pushes the value types ts_2 to be produced to the stack, and proceed to the next instruction.

As an example, consider the following program consisting of two binary operations of type **i32**

operating on an operand stack of three **i32** values:

$$[\textbf{i32.const}\ c_1; \textbf{i32.const}\ c_2; \textbf{i32.const}\ c_3; \textbf{i32.binop}\ \mathit{binop}_1; \textbf{i32.binop}\ \mathit{binop}_2]$$

Validation requires checking whether this program can be typed of function type $[] \to [\textbf{i32}]$. Figure 3.10 illustrates the type checking procedure, by starting with the consumed type in the desired function type (which is $[]$ in this case), going through the instructions in their order and maintaining a cumulative stack type at each step. Types produced by the instruction in the preceeding step are coloured in green. Types consumed by the instruction in the following step are coloured in red. Types produced in the preceeding step and consumed in the following step are coloured in yellow.

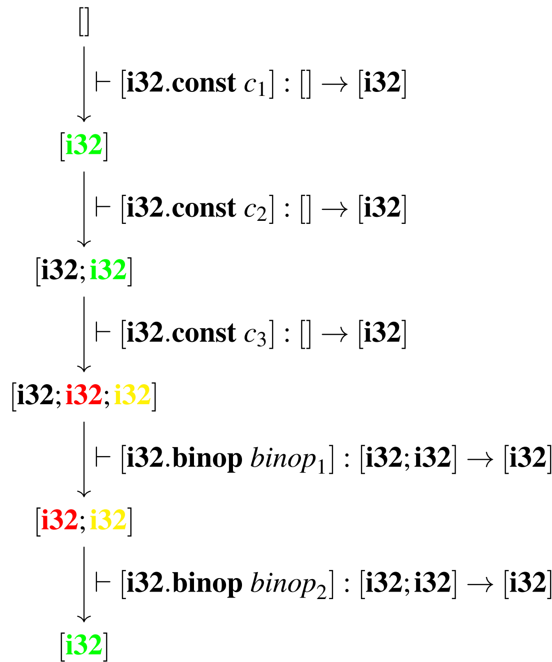


Figure 3.10: Illustration of type-checking a simple program

Therefore, the above procedure confirms that this program fragment can be assigned of function type $[] \to [\textbf{i32}]$.

Certain instructions require information from the typing context C. For example, the **local.get** k instruction has type $[] \to [t]$ where $t = C.\mathsf{locals}[k]$. Such information can be managed by keeping track of the current typing context C.

Certain instructions are *stack polymorphic*. For example, the **select** instruction consumes three values of types $[t; t; \textbf{i32}]$ from the stack for any t and produces $[t]$ back to the stack. This can be implemented by first consuming an **i32** type from the stack, followed by checking whether the next two types are equal.

Most of Wasm's instruction set can be type-checked in the above method. However, the control flow instructions **br** (and its variants), **return** and **unreachable** cannot be processed in this way. Their typing rules are fully stack-polymorphic and consume an arbitrary list of types (on top of some required types) and produce an arbitrary list of types, so that the function type of the

instruction can be compatible with the dead code remaining in the program or the extra values on the operand stack. The above simple type-checking process cannot predict in advance what type should be assigned to these instructions so as to make them compatible with the upcoming instructions. Note that it is not always straightforward to type-check the remaining part of the program first: as demonstrated by the **select** instruction, the type produced by certain instructions may depend on the existing types on the stack in a complicated way.

Instead, whenever an instruction producing arbitrary types is encountered and the values it needs to consume are satisfied, I clear the rest of the content of the stack, and push a wildcard symbol $*$ to the stack. Type-checking proceeds normally by continuing to the next instruction. Whenever some types need to be consumed and the stack contains only the $*$ symbol left, the consumption is considered to be satisfied, and the wildcard symbol $*$ stays in the stack. At the end of the type checking, the type checker tries to match the resulting stack type with the expected types to be produced, where the $*$ symbol matches with arbitrary types on the stack.

Mechanisation In the Coq mechanisation, instead of using an explicit wildcard symbol, I instead augment the current cumulative stack type by a boolean that indicates if there's a wildcard at the bottom of the stack.

```
(* Boolean flag for unreachable. *)
Record checker_type: Type :=
  { CT_type: list value_type;
    CT_unr: bool;
  }.
Notation "<< ts , unr >>" := (Build_checker_type ts unr) (at level 5).

Fixpoint consume (ct: checker_type) (cons : list value_type) : option checker_type :=
    ... .

Definition produce (t1 : checker_type) (t2: list value_type) : checker_type :=
    <<t2 ++ t1.(CT_type), t1.(CT_unr)>>.
```

The type checker in the mechanisation $\mathsf{be_type_checker}(C, es, ft)$ is defined using a mutually recursive `Fixpoint` with a function type-checking a single instruction and a function type-checking an instruction list; this design is similar to the mutual recursion in the executable interpreter, where one part defines the execution on the top instruction on the instruction stack, and the other defines the execution of a general instruction list. The type-checker returns a boolean indicating whether the typing relation specified by its arguments holds.

Correctness The correctness property I proved for the type checker is the following equivalence, therefore including both soundness and completeness:

Proposition 3.6.1.

$$C \vdash es : ft$$

if and only if

$$\mathsf{be_type_checker}(C, es, ft) = \mathsf{true}.$$

The proof is done by proving the two implications separately. The forward implication is easily proved by an induction on the typing relation: this is because the type checker is a *computable* function, therefore in most cases it suffices to simply evaluate the type checker on the syntactic term given by the case analysis.

The backward implication that the type checker result implies the typing relation is more technical. Since the type checker is defined by mutual recursion, I prove this implication by a strong induction on the complexity of the program *es* to be typed; this ensures that an induction hypothesis will be available when required for typing the block constructs. Any complexity that strictly decreases when focusing on sub-programs of block constructs works, so there are many ways of defining such a complexity. WasmCert-Coq uses for the following simple definition:

```
Fixpoint be_size_single (be: basic_instruction): nat :=
  match be with
  | BI_block _ l => 1 + (List.fold_left addn (map be_size_single l)) 1 + size l
  | BI_loop _ l => 1 + (List.fold_left addn (map be_size_single l)) 1 + size l
  | BI_if _ l1 l2 => 1 + ((List.fold_left addn (map be_size_single l1) 1) + size l1) +
  ((List.fold_left addn (map be_size_single l2) 1) + size l2)
  | _ => 1
  end.
```

Note that while this design is certainly not minimal, it ensures the above decreasing property regardless.

Similar to the automation with the typing inversion lemmas, I devised a tactic that simplifies the type-checker valuation and extract the constraints that the arguments of the type checker function need to satisfy, and use them to construct the typing relation. The only tedious case is the case for the **select** instruction due to its interaction with a stack containing a wildcard $*$ due to the number of different subcases that need to be considered. Nevertheless, in general the proof is a straightforward strong induction with case analysis that is technical but does not require too many intermediate designs, unlike the preservation and progress property.

3.7 Verified and Executable Module Instantiation

I define the following instantiation predicate, which is based on the relational definition of the instantiation operation in the Wasm specification:

```
Definition instantiate (s : store_record) (m : module) (v_imps : list v_ext)
            (z : (store_record * instance * list module_export) * option nat) : Prop :=
  ... .

Definition interp_instantiate (s : store_record) (m : module) (v_imps : list v_ext)
  : option ((store_record * instance * list module_export) * option nat) := ...
```

Given the starting store S, a module m, and a list of imports v_imps, and a tuple of output containing the new store, module instance, list of module exports and the optional start function index, this instantiate relation defines whether the provided output is a plausible result of module instantiation for m.

Recall in Section 2.7.2 about the apparent circularity in evaluating the initialiser and offset expressions and that the relational version of instantiation is not directly executable. I therefore defined an executable version of the instantiation operation, $\mathsf{interp_instantiate}$, which computes the desired result of instantiation under the given constraints. The implementation is as described in Section 2.7.2, which makes a pre-pass to compute the various initialiser and offset expressions, and uses the results to allocate the states in the store. Note that the result is defined as an option type of the expected output: this reflects that instantiation may fail if the module is invalid, or if the types of the imports do not match with the requested module imports.

I proved the following two soundness results, one each for the relational version and the executable version of instantiation.

Proposition 3.7.1. *Given a valid store S ($\vdash S : \mathsf{ok}$) and a module, if*

$$\mathsf{instantiate}(S, m, v_{\mathsf{imps}}, (S', F, exps, start))$$

then

- $\vdash S' : \mathsf{ok}$*;*
- $S \preccurlyeq S'$*;*
- $\exists C.\ S' \vdash F : C$*.*

Moreover, the indices in the module exports and the start function are not out of bounds.

The proof of this lemma is purely technical by unfolding all the definitions. Proving that the new store is an extension of the old store is completely syntactic. To prove that the new store is also valid, I need to prove that all the new states allocated are valid, which are ensured by the corresponding validity relations of the module components. However, I also need to verify that the old components are still valid under the new store (with the newly added states). This is trivial for tables, memories, and globals, but requires more technical effort for function instances. In particular, I need to prove that all the function bodies are of the same instruction types in the new store after module instantiation. This holds since module instantiation only ever allocates new states and do not change the type of the old states; in fact, after proving that the resulting store of a module instantiation is always valid store extension of the old store, Lemma 3.5.3 can be applied to establish the invariant of typing of the function body.

The validity of the frame is obtained by verifying that the addresses recorded in the frame indeed correspond to the addresses of the imported states and the newly allocated states in the post-instantiation store. The parts about exports and the start function are directly obtained by unfolding the corresponding validity of the module exports and start function.

Proposition 3.7.2. *If*

$$\mathsf{interp_instantiate}(S, m, v_{\mathsf{imps}}) = \mathsf{Some}\ (S', F, exps, start))$$

then

$$\mathsf{instantiate}(S, m, v_{\mathsf{imps}}, (S', F, exps, start))$$

This proof is again an exercise of unfolding definitions. In fact, the executable version of instantiation mostly reuses the same set of definitions as the relational version (except for the places that deal with the initialiser expressions), so the proof is quite straight-forward for all the other parts. For the only non-trivial part that proves the correctness of the pre-pass that evaluates the initialiser expressions, the proof goes by noting that the initialiser expressions can only contain values (which are state-agnostic) and **global**.**get** of imported globals (which already exist in the old store S), so evaluating these expressions in the old store S yields the same result as evaluating them in the post-instantiation store S'.

3.8 Numerics

WasmCert-Coq implements Wasm numeric natively in Coq using CompCert's integers [66, 67] and Flocq's floating-point arithmetic [19, 18]. As both of these numerics come from verified libraries, this approach reduces the size of the unverified codebase compared to Watt's Isabelle mechanisation, which initially used external OCaml functions for both the integer and floating point operations, and later switched to a verified set of integer operations [115].

However, some wrapper functions are still required to connect from the operations of the numeric libraries, since Wasm defines its own set of corner case behaviours for numeric operations, such as the payloads of NaN and the signs of infinities and zeros, as illustrated in Figure 3.11 for floating point multiplications.

The custom wrapper functions perform the required case analyses and invoke the library functions in the corresponding cases. Therefore, although the underlying libraries are verified, there could still be bugs in the implementation of numerics in the wrapper functions due to the sheer number of cases of the definition. In particular, I have discovered several bugs in the initial version of the wrapper functions, mostly due to incorrectly assigning the signs for the results of arithmetic operations.

The above type of bugs are difficult to spot among tens of thousands of lines of code in the mechanisation. They also cannot be revealed by proving any of the traditional soundness properties (such as type soundness), since the type signatures of erroneous implementations can still be correct, despite the results themselves being incorrect. Moreover, contrary to the interpreter correctness proof, which proves that the behaviour of the executable interpreter matches the inductively defined operational semantics, a similar approach would be relatively ineffective for verifying the correctness of the numeric wrapper functions, for the main complexity of these functions lies in the case analysis itself. Therefore, a specification of them is prone to the same kind of human errors.

As a result, traditional bug-finding methods like testing are the most efficient in ensuring correctness of these implementations. All (except one) bugs in the numeric wrapper functions in

$\text{fmul}_N(z_1, z_2)$

- If either z_1 or z_2 is a NaN, then return an element of $\text{nans}_N\{z_1, z_2\}$.
- Else if one of z_1 and z_2 is a zero and the other an infinity, then return an element of $\text{nans}_N\{\}$.
- Else if both z_1 and z_2 are infinities of equal sign, then return positive infinity.
- Else if both z_1 and z_2 are infinities of opposite sign, then return negative infinity.
- Else if either z_1 or z_2 is an infinity and the other a value with equal sign, then return positive infinity.
- Else if either z_1 or z_2 is an infinity and the other a value with opposite sign, then return negative infinity.
- Else if both z_1 and z_2 are zeroes of equal sign, then return positive zero.
- Else if both z_1 and z_2 are zeroes of opposite sign, then return negative zero.
- Else return the result of multiplying z_1 and z_2, rounded to the nearest representable value.

$$
\begin{array}{lcl}
\text{fmul}_N(\pm\mathsf{nan}(n), z_2) & = & \text{nans}_N\{\pm\mathsf{nan}(n), z_2\} \\
\text{fmul}_N(z_1, \pm\mathsf{nan}(n)) & = & \text{nans}_N\{\pm\mathsf{nan}(n), z_1\} \\
\text{fmul}_N(\pm\infty, \pm 0) & = & \text{nans}_N\{\} \\
\text{fmul}_N(\pm\infty, \mp 0) & = & \text{nans}_N\{\} \\
\text{fmul}_N(\pm 0, \pm\infty) & = & \text{nans}_N\{\} \\
\text{fmul}_N(\pm 0, \mp\infty) & = & \text{nans}_N\{\} \\
\text{fmul}_N(\pm\infty, \pm\infty) & = & +\infty \\
\text{fmul}_N(\pm\infty, \mp\infty) & = & -\infty \\
\text{fmul}_N(\pm q_1, \pm\infty) & = & +\infty \\
\text{fmul}_N(\pm q_1, \mp\infty) & = & -\infty \\
\text{fmul}_N(\pm\infty, \pm q_2) & = & +\infty \\
\text{fmul}_N(\pm\infty, \mp q_2) & = & -\infty \\
\text{fmul}_N(\pm 0, \pm 0) & = & +0 \\
\text{fmul}_N(\pm 0, \mp 0) & = & -0 \\
\text{fmul}_N(z_1, z_2) & = & \text{float}_N(z_1 \cdot z_2)
\end{array}
$$

Figure 3.11: Specification for the special cases in floating point multiplication

WasmCert's implementation were discovered through testing against the official test suite, which will be described in Section 3.11.

3.9 Binary Format Conversion

WasmCert-Coq implements two executable functions for decoding and encoding between the binary format and the Coq AST of Wasm modules within Coq, using the Parseque [6] combinator library.

These definitions are currently used in the end-to-end executable extracted interpreter and remain unverified: verification for them faces the same challenge as the numeric implementations due to the lack of a reference semantics to be verified against. Nevertheless, certain properties of these conversion functions can be expected to hold. For example, since decoding and encoding provide bidirectional conversions between the Coq AST and Wasm binaries, it is reasonable to expect that they are some form of inverses to each other.

However, Wasm's specification of its binary format allows non-unique representations of the AST in the binary format. For example, for the conditional block construct

$$\textbf{if}\ \mathit{ft}\ \mathit{es}_1\ \textbf{else}\ \mathit{es}_2\ \textbf{end}$$

the binary encoding opcode of the else branch can be optionally omitted if the body of the else

branch is empty. As a result, the decoding function from the binary format to the Coq AST and the encoding function in the other direction are not inverse to each other. However, due to the uniqueness of the AST representation in Coq, the decoding function is expected to be the left inverse of the encoding function, which motivate the following correctness statement for all Wasm modules m:

Conjecture 3.9.1.

$$\mathsf{decode}(\mathsf{encode}(m)) = m.$$

The above statement is not proven in the WasmCert-Coq mechanisation. Instead, the binary format parser is thoroughly tested through the official test suite to be introduced Section 3.11. The binary format printer is currently used by the CertiCoq-Wasm project [71], from which several bug reports have been received and addressed.

3.10 Updating WasmCert to WebAssembly 2.0

Having seen the entire WasmCert-Coq for WebAssembly 1.0, this section describes the update of WasmCert-Coq to WebAssembly 2.0, which demonstrates how the model of the mechanisation adapts when a new version of the language semantics is released.

Wasm 2.0 is a major update over version 1.0, integrating multiple extension proposals that vastly extend the abstract syntax and semantics of the language. Due to the vast scope of this feature extension, some of the existing mechanisation designs need to be revamped in order to maintain it as a scalable artifact. I first describe the extension proposals introduced by Wasm 2.0, then describe the definitions and proofs affected, using some of the most impacted parts of the codebase as examples.

3.10.1 The WebAssembly 2.0 Feature Extensions

Wasm 2.0 incorporates the following feature extensions:

Reference Types

The reference types proposal introduces a new family of primitive types, *reference types*, into Wasm 1.0's type system which only included the four primitive numeric types. The main objective of this proposal is to allow functions, including both the native and host functions, to be directly used without needing to go through indirections through the Wasm tables.

In addition, this proposal also introduces new instructions that allow direct manipulation of function tables to a similar extent as the linear memory. In contrast, tables in Wasm 1.0 are essentially immutable in Wasm (without going through host function invocations), and are instantiated at module instantiation with the initial function references. In fact, besides **call_indirect** and host operations, there is no other instruction in Wasm 1.0 that interacts with the table at all.

With this proposal, tables now serve the role as a separate piece of storage for the reference values, much resembling the role of Wasm's linear memory as the storage of the Wasm 1.0 primitive values

(in bytes).

Concretely, the reference types proposal incorporates the following changes:

Extended set of primitives In addition to Wasm 1.0's primitive 32-bit and 64-bit integer and float number types, a new category of primitive types – *reference types* – is introduced. This consists of **funcref**, the types of function references; and **externref**, which is the type of references to host objects that can be passed into Wasm.

Values are similarly expanded by a new category of *reference values* with three constructors: **ref**.**null** t for null references of type t, **ref** x for a function reference to address x in the *store*, and **ref**.**extern** x for an external reference to address x in the storage of the host. The typing rules for the reference values are straightforward: the only rule with a premise is that of **ref** x, which merely requires the function address to be well-typed with respect to the store.

I summarise these additions in Figure 3.12.

$$\begin{array}{rcl} \text{(value type) } t & ::= & \mathit{numtype} \mid \mathit{reftype} \mid \ldots \\ \text{(number type) } \mathit{numtype} & ::= & \mathbf{i32} \mid \mathbf{i64} \mid \mathbf{f32} \mid \mathbf{f64} \\ \text{(reference type) } \mathit{reftype} & ::= & \mathbf{funcref} \mid \mathbf{externref} \\ \text{(values) } v & ::= & \mathit{numtype}.\mathbf{const}\ c \mid \\ & & \mathbf{ref}.\mathbf{null}\ \mathit{reftype} \mid \mathbf{ref}\ \mathit{funcaddr} \mid \mathbf{ref}.\mathbf{extern}\ \mathit{externaddr} \mid \\ & & \ldots \end{array}$$

$$\frac{}{\vdash \mathbf{ref}.\mathbf{null}\ t : [] \to [t]}\ \textsf{ref_null}$$

$$\frac{S \vdash \mathbf{func}\ a : -}{S;C \vdash \mathbf{ref}\ a : [] \to [\mathbf{funcref}]}\ \textsf{ref_func} \qquad \frac{}{S;C \vdash \mathbf{ref}\ a : [] \to [\mathbf{externref}]}\ \textsf{ref_extern}$$

Figure 3.12: Extended primitive types and values

Extended instruction set for reference operations A collection of new instructions that operate on reference types are introduced. These include two simple instructions **ref**.**func** x, which fetches the corresponding **ref** value in the store; and **ref**.**is_null**, which checks if the top reference on the operand stack is a **ref**.**null**.

Besides above, four new instructions related to tables – **table**.**get**/**set**/**size**/**grow** – are introduced. These instructions operate similarly to their *memory* counterparts. In particular, **table**.**grow** can either successfully grow the specified table by n elements filled with the default value v, and update the table type to the resulting type (represented by the growtable function in the figure), or it is allowed to fail non-deterministically (like **memory**.**grow**) for any other reason, e.g. if the host does not possess sufficient resources. A slight change to the upper bound of the table size limit from 2^{32} to $2^{32}-1$ is required as a side effect of **table**.**size** returning the size of the table as an **i32** integer (as well as **table**.**grow**), since the number 2^{32} would not fit in the range of 32-bit integers.[3]

[3]The memory sizes do not suffer from the same issue as they are stated in pages (2^{16} bytes). As a result, the allowed maximum of 2^{32} bytes correspond to only 2^{16} in terms of Wasm's memory size limit.

I display these additions in Figure 3.13.

Other changes Technically, the reference type proposal is the proposal that introduced the ability for a module to use multiple (>1) tables. However, as I discussed earlier this chapter, the WasmCert-Coq mechanisation for Wasm 1.0 already models it in this way.

A consequence of this change is that, the dynamic function call instruction **call_indirect** now carries an extra argument, specifying the index of the table that the instruction is referring to. Moreover, the module element segment now contains an additional part that specifies the table index.

The **select** instruction is augmented to optionally carry a value type to be selected, which must then match the respective value on the operand stack. This optional value type is allowed to be missing (for backward compatibility with Wasm 1.0), in which case the selected operand must be of a *numeric* or a *vector* type (to be introduced in the SIMD extension proposal subsection later).

These changes are also displayed in Figure 3.13.

$$\text{(instructions) } instr ::= \ldots \mid \textbf{select } (t^?) \mid \textbf{call_indirect } \mathit{tableidx}\ \mathit{typeidx} \mid \textbf{ref.func } x \mid \textbf{ref.is_null} \mid \textbf{table.get/set/size/grow } x$$

$$\frac{\mathsf{is_num}\ t \lor \mathsf{is_vec}\ t}{\vdash: \textbf{select} : [t; t; \textbf{i32}] \to [t]}\ \mathsf{select_default} \qquad \frac{}{\vdash: \textbf{select}\ t : [t; t; \textbf{i32}] \to [t]}\ \mathsf{select}$$

Reduction	Condition
$(F; [\textbf{ref.func}\ x]) \hookrightarrow (F; [\textbf{ref}\ a])$	$a = F.\mathsf{module.funcaddrs}[x]$
$[v; \textbf{ref.is_null}] \hookrightarrow [\textbf{i32.const}\ 1]$	$v = \textbf{ref.null}\ t$
$[v; \textbf{ref.is_null}] \hookrightarrow [\textbf{i32.const}\ 0]$	*otherwise*
$(S; F; [\textbf{i32.const}\ i; \textbf{table.get}\ x]) \hookrightarrow (S; F; [v])$	$S.\mathsf{tables}[F.\mathsf{module.tableaddrs}[x]].\mathsf{elem}[i] = v$
$(S; F; [\textbf{i32.const}\ i; \textbf{table.get}\ x]) \hookrightarrow (S; F; [\textbf{trap}])$	*otherwise*
$(S; F; [\textbf{i32.const}\ i; v; \textbf{table.set}\ x]) \hookrightarrow (S'; F; [])$	$S' = S$ with $\mathsf{tables}[F.\mathsf{module.tableaddrs}[x]].\mathsf{elem}[i] = v$
$(S; F; [\textbf{i32.const}\ i; v; \textbf{table.set}\ x]) \hookrightarrow (S; F; [\textbf{trap}])$	*otherwise*
$(S; F; [\textbf{table.size}\ x]) \hookrightarrow (S'; F; \textbf{i32.const}\ c)$	$\lvert S.\mathsf{tables}[F.\mathsf{module.tableaddrs}[x]].\mathsf{elem}\rvert = c$
$(S; F; [v; \textbf{i32.const}\ n; \textbf{table.grow}\ x]) \hookrightarrow (S'; F; \textbf{i32.const}\ c)$	$F.\mathsf{module.tableaddrs}[x] = a$ $\lvert S.\mathsf{tables}[a].\mathit{elem}\rvert = c$ $S' = S$ with $\mathsf{tables}[a] = \mathsf{growtable}(S.\mathsf{tables}[a], n, v)$
$(S; F; [v; \textbf{i32.const}\ n; \textbf{table.grow}\ x]) \hookrightarrow (S; F; \textbf{i32.const}\ (-1))$	*no condition*(non-det)

Figure 3.13: Reference and table operations

Bulk Memory and Table Operations

This proposal originally only added the bulk *memory* operations which support modification to a range of memory entries at once. The reference types proposal then extended this to include a corresponding set of instructions for tables as well.

As the design of the bulk memory and table operations are essentially the same, I first describe the added instructions for tables:

- **table**.**fill** *tableidx*: fills a range in the table *tableidx* to a reference value. The range and the value to be filled are specified by values on the operand stack, in the order of the length of the range, the fill value, and the starting offset in the table.
- **table**.**copy** *tableidx tableidy*: copies a range in the table *tableidx* to another range in the table *tableidy*. The source and destination ranges are specified by values on the operand stack, in the order of the length of the range, the offset in the source table, and the offset in the destination table.
- **table**.**init** *tableidx elemidx*: initialises a range in the table *tableidx* with the initialiser being a value in the elem segment *elemidx*. The range and the index in the elem segment being used are specified by values on the operand stack, in the order of the length of the range, the offset in the elem segment (source), and the offset in the table (destination).
- **elem**.**drop**: clears the content of an elem segment, preventing further uses.

The bulk memory operations contain a corresponding set of four instructions: **memory**.**fill**/**copy**/**init** and **data**.**drop**. They function in the same way as the table operations, except that they implicitly operate on the 0th memory, since Wasm 2.0 preserved the artificial restriction of only allowing up to one memory per module (as I discussed earlier this chapter). As a result, the memory instructions have one fewer arguments than their table counterparts, omitting the memory index argument. When the restriction on the number of memories is lifted in a future Wasm extension, I expect these arguments to be similarly augmented with the same argument specifying the memory index they are operating on. However, this will not cause additional mechanisation maintenance obligation, since the WasmCert mechanisation has always been modelling the generalised situation of multiple memories and tables since Wasm 1.0, which I preserve in the 2.0 update.

In the rest of this section, I will refer to the bulk memory and table operations as the *bulk operations* collectively for simplicity.

The operational semantics for each of the bulk operation is defined by iteratively assigning the corresponding state entries one by one (in contrast to the entire range being assigned in an atomic step). Therefore, these bulk operations could have been equivalently implemented using a loop iterating through the range of states to be assigned. Nevertheless, adding these instructions are beneficial for two reasons. First, they provide a shorthand for bulk operations, which are frequent in real world programs (e.g. compilation result of memcpy or memset), which helps to reduce the size of Wasm binaries produced. More importantly, these instructions can be easily optimised by industrial Wasm interpreters. This is crucial for the important design goal of Wasm of achieving near-native speed of execution in browsers and other virtual host environments. Optimisation is also the reason behind the addition of **elem**/**data**.**drop** as well: these instructions technically have no effect on the result of execution as long as the program does not use any of the segment that has been dropped. However, they allow interpreter implementations to perform some garbage collection and *free* the segments being dropped, as they should no longer be used in further execution.

A special note is required for the **memory**/**table**.**copy** instruction: it is possible that the source and destination ranges have a non-empty overlap. As the operational semantics unrolls the bulk copy instruction to individual **get** and **set** instructions, a careful case analysis is required to perform the copies in the correct order, by whether the source range is in front of the destination: this is to prevent the states being copied from getting overwritten. For example, when the source range is in front of the destination range, the copy instruction unrolls from the end of the range; and vice versa when the destination range is in front.

This proposal also introduces an additional passive mode to the elem and data segment, in contrast to the original mode which is denoted as the active mode. This distinguishes between the elem and data segments that will be automatically copied during module instantiation (the original active segments) and the segments that are used for dynamic initialisation by the bulk init instructions during execution (the passive segments).

I describe the above additions in Figure 3.14. I omit the operational semantics of the bulk memory operations in the figure as they are defined in the same way as their table counterparts (besides implicitly referring to memory 0 for all operations). Note that none of the fill/copy/init directly change the store S in the reduction rule. This is because, as previously mentioned, their operational semantics are defined as being unfolded to individual operations; thus the reduction result is only the unfolding one individual operation from the bulk operations.

Vector Instructions (SIMD)

The vector instructions proposal, also known as the SIMD (Single Instruction Multiple Data) proposal, further extended Wasm's primitive types to include a vector type **v128** which manipulates multiple numeric values in parallel. SIMD instructions are supported by hardware, which perform multiple computations over packed data in one instruction to help improve performance. Each piece of hardware, however, supports a large set of potentially different SIMD instructions. The SIMD proposal consists of a subset of instructions that correspond to most of the commonly used instructions on modern hardware. This allows Wasm implementations to take advantage of the commonly available SIMD instructions in hardware for native-speed execution [116], while fulfilling its design goal of being a portable format.

Other extension proposals and modifications

Apart from the three major extension proposals discussed above, Wasm 2.0 also includes several other smaller-sized feature extensions. These contain two added numeric instructions and a generalisation of result type to allow a general list of values.

Sign-extension operations A new numeric instruction **i32**/**i64**.**extendN_s** is added for performing sign extension with integer representations. These instructions extend a signed N-bit integer to a 32/64-bit integer.

[4] The choice of having two branches with side-condition marked *otherwise* is by the Wasm standard. Both of them refer to the other case of the side condition to the first reduction rule, and in this case the rule to be applied can be determined by matching the top **i32** value on the operand stack; but the fact that the combination of the three rules forms a well-defined semantic rule set for **table**.**fill** is not entirely obvious.

$$\begin{array}{lrl}
\text{(instructions) } \mathit{instr} & ::= & \ldots \mid \\
 & & \textbf{table.fill}\ \mathit{tableidx} \mid \textbf{table.copy}\ \mathit{tableidx}\ \mathit{tableidy} \mid \\
 & & \textbf{table.init}\ \mathit{tableidx}\ \mathit{elemidx} \mid \textbf{elem.drop}\ \mathit{elemidx} \\
 & & \textbf{memory.fill} \mid \textbf{memory.copy} \mid \\
 & & \textbf{memory.init}\ \mathit{dataidx} \mid \textbf{data.drop}\ \mathit{dataidx}
\end{array}$$

Rule	Condition
$(S;F;[\textbf{i32.const}\ i; v; \textbf{i32.const}\ n; \textbf{table.fill}\ x]) \hookrightarrow$ $(S;F;[\textbf{trap}])$	$i+n > \lvert S.\mathsf{tables}[F.\mathsf{module}.\mathsf{tableaddrs}[x]].\mathsf{elem}\rvert$
$(S;F;[\textbf{i32.const}\ i; v; \textbf{i32.const}\ 0; \textbf{table.fill}\ x]) \hookrightarrow$ $(S;F;[])$	*otherwise*[4]
$(S;F;[\textbf{i32.const}\ i; v; \textbf{i32.const}\ (n+1); \textbf{table.fill}\ x]) \hookrightarrow$ $(S;F;[\textbf{i32.const}\ i; v; \textbf{table.set}\ x; \textbf{i32.const}\ (i+1); v; \textbf{i32.const}\ n; \textbf{table.fill}\ x])$	*otherwise*
$(S;F;[\textbf{i32.const}\ d; \textbf{i32.const}\ s;$ $\textbf{i32.const}\ n; \textbf{table.copy}\ x\ y]) \hookrightarrow$ $(S;F;[\textbf{trap}])$	$s+n > \lvert S.\mathsf{tables}[F.\mathsf{module}.\mathsf{tableaddrs}[y]].\mathsf{elem}\rvert \vee$ $d+n > \lvert S.\mathsf{tables}[F.\mathsf{module}.\mathsf{tableaddrs}[x]].\mathsf{elem}\rvert$
$(S;F;[\textbf{i32.const}\ d; \textbf{i32.const}\ s; \textbf{i32.const}\ 0; \textbf{table.copy}\ x\ y]) \hookrightarrow$ $(S;F;[])$	*otherwise*
$(S;F;[\textbf{i32.const}\ d; \textbf{i32.const}\ s; \textbf{i32.const}\ (n+1); \textbf{table.copy}\ x\ y]) \hookrightarrow$ $(S;F;[\textbf{i32.const}\ d; \textbf{i32.const}\ s; \textbf{table.get}\ y; \textbf{table.set}\ x;$ $\textbf{i32.const}\ (d+1); \textbf{i32.const}\ (s+1); \textbf{i32.const}\ n; \textbf{table.copy}\ x\ y])$	*otherwise*, if $d \leq s$
$(S;F;[\textbf{i32.const}\ d; \textbf{i32.const}\ s; \textbf{i32.const}\ (n+1); \textbf{table.copy}\ x\ y]) \hookrightarrow$ $(S;F;[\textbf{i32.const}\ (d+n); \textbf{i32.const}\ (s+n); \textbf{table.get}\ y; \textbf{table.set}\ x;$ $\textbf{i32.const}\ d; \textbf{i32.const}\ s; \textbf{i32.const}\ n; \textbf{table.copy}\ x\ y])$	*otherwise*, if $d > s$
$(S;F;[\textbf{i32.const}\ d; \textbf{i32.const}\ s;$ $\textbf{i32.const}\ n; \textbf{table.init}\ x\ y]) \hookrightarrow$ $(S;F;[\textbf{trap}])$	$s+n > \lvert S.\mathsf{tables}[F.\mathsf{module}.\mathsf{elemaddrs}[y]].\mathsf{elem}\rvert \vee$ $d+n > \lvert S.\mathsf{tables}[F.\mathsf{module}.\mathsf{tableaddrs}[x]].\mathsf{elem}\rvert$
$(S;F;[\textbf{i32.const}\ d; \textbf{i32.const}\ s; \textbf{i32.const}\ 0; \textbf{table.init}\ x\ y]) \hookrightarrow$ $(S;F;[])$	*otherwise*
$(S;F;[\textbf{i32.const}\ d; \textbf{i32.const}\ s; \textbf{i32.const}\ (n+1); \textbf{table.init}\ x\ y]) \hookrightarrow$ $(S;F;[\textbf{i32.const}\ d; v; \textbf{table.set}\ x;$ $\textbf{i32.const}\ (d+1); \textbf{i32.const}\ (s+1); \textbf{i32.const}\ n; \textbf{table.init}\ x\ y])$	*otherwise*, if $v = S.\mathsf{elems}[F.\mathsf{module}.\mathsf{elemaddrs}[y]].\mathsf{elem}[s]$
$(S;F;[\textbf{elem.drop}\ x]) \hookrightarrow (S';F;[])$	$S' = S$ with $\mathsf{elems}[F.\mathsf{module}.\mathsf{elemaddrs}[x]].\mathsf{elem} = []$

Figure 3.14: Bulk table and memory operations

Non-trapping float-to-int conversions A new numeric instruction **i32/64.trunc_sat_f32/64_sx** is added as a saturated version for conversion from floating-point numbers to integers, which always returns an integer value instead of returning a trap when the conversion fails. The purpose of these additions include a better fit to the corresponding conversion in LLVM (which returns an undefined result instead of resulting in undefined behaviour), and to provide a convention for saturating existing operations. This convention is shared by the SIMD proposal, where the instructions are designed to better emulate SIMD-hardware, where no SIMD operations trap[83].

Multi-value proposal This proposal generalises the definition of *result type* from an optional value type to a list of value types. This means that blocks and function calls can now return multiple values instead of only at most one. However, as I discussed earlier this chapter, this is already implemented in the 1.0 version of the WasmCert-Coq mechanisation for forward compatibility.

Stack typing and validation Aside from the above feature extension proposals, Wasm 2.0 further extends the type system to define an *operand type*, which includes a bottom type representing the types used for unreachable code, along with the three concrete categories of value types (numerics/reference/vector types). As a result, instead of function types, the typing relation for program fragments now use a *stack type* instead, which is defined based on operand types instead of value types.

In addition, the Wasm 2.0 specification defines a *matching* relation between operand types $t_1 \leq t_2$, which is essentially a *subtyping* relation that describes if a type t_1 can be lifted to a supertype t_2. However, given the limited set of operand types in Wasm 2.0, the only valid matching relation except for the trivial case of reflexivity is the matching of $\bot$ to all other operand types. The matching relation is extended pointwise to a matching relation between stack types.

$$\begin{array}{rll} \text{(operand types)} & \mathit{opdtype} & ::= \mathit{valtype} \mid \bot \\ \text{(stack types)} & \mathit{stacktype} & ::= \mathit{opdtypes} \to \mathit{opdtypes} \end{array}$$

$$\frac{}{\vdash t \leq t} \qquad \frac{}{\vdash \bot \leq t}$$

In Wasm 2.0, the purpose of this modification is to better express the typing rules for **select** and **br_table**, in particular, in the presence of dead code. In addition, this modification also helps to prepare for the future Wasm 3.0, where a proper subtype lattice is introduced, adding some non-trivial subtype relations.

This modification has a big impact on the formulation of the type system and validation rules. Several places in the typing rules which used to state the equality between the type present in the context and the type required to be consumed are replaced by this matching relation. For example, the typing rule of the **br_table** instruction is updated to the following (compare to the old rule in Figure 2.16, where both of the matching relations $\leq$ were equalities instead):

$$\frac{\vdash ts \leq C.\mathsf{labels}[ld] \quad (\vdash ts \leq C.\mathsf{labels}[l_i])^*}{C \vdash \mathbf{br_table}\ ls\ ld : (ts_1 +\!\!+\ ts +\!\!+\ [\mathbf{i32}]) \to ts_2}\ \mathsf{br_table}$$

and the typing rule for **select** is updated to allow bottom type $\bot$:

$$\frac{t \le numtype \lor t \le vectype}{\vdash: \mathbf{select} : [t;t;\mathbf{i32}] \to [t]}\ \text{select_default} \qquad \frac{}{\vdash: \mathbf{select}\ t : [t;t;\mathbf{i32}] \to [t]}\ \text{select}$$

In Wasm 2.0, the above changes make a slight difference for the well-typedness of dead code which involves **select** and **br_table**. For example, the following program fragment

$$[\mathbf{br}\ 0; \mathbf{select}; \mathbf{ref.is_null}]$$

would be a valid program (under a typing context with suitable labels) by assigning the stack type $[] \to [\bot;\bot;\mathbf{i32}]$ to the **br** 0 instruction, which results in the overall program fragment being typeable with stack type $[] \to [\mathbf{i32}]$. This could not have been possible without the introduction of the bottom type $\bot$, since the **select** instruction always returns a numeric or vector type otherwise, while the **ref**.**is_null** instruction requires to consume a reference type.

In fact, there is a mistake in the initial update of WasmCert-Coq to Wasm 2.0 regarding **br_table** due to subtleties related to this change. This will be covered in Section 3.11.

Full type signatures for module instances Wasm 2.0 makes a modification to its runtime representation of the states, such that the instances of tables, memories, and globals in the store now contain their full table/memory/global types, in comparison to only the optional maximum size for tables and memories, or the mutability for globals in Wasm 1.0. The relevant execution semantics, typing rules, and various definitions in the soundness property are updated accordingly.

3.10.2 Challenges of The WasmCert-Coq Update for Wasm 2.0

Our update to the Wasm 2.0 specification shares a similar trusted computing base with the original WasmCert-Coq[114] mechanisation for Wasm 1.0 and the related verified interpreter from WasmRef-Isabelle[115]. In particular, the extraction process from Coq to OCaml and the OCaml tools themselves need to be trusted, and the vector instructions are similarly implemented as opaque instructions whose behavior agrees with the function type defined in the specification, with the concrete implementation left to be generated at the OCaml level, as there lacks a mature formalisation of the relevant machine-level vector operations. This mirrors the approach used by Watt et al. [115] for floating point operations in their Isabelle/HOL mechanisation of Wasm 1.0. Besides the above, I continue to use the verified CompCert[66] numerics for integer and the Flocq [19] floating point arithmetic, as well as the Parseque parser combinator library [6, 5] to generate the binary format parser.

In this section, I describe the challenges I encountered in updating the definitions and proofs for the Wasm 2.0 specification.

Instruction lifting for reference values The reference types proposal introduced several new constructors, including function references holding explicit addresses and external references

holding explicit external value addresses. Both of these refer to the locations in the store. Therefore, to satisfy Wasm's module encapsulation property, none of these reference values can be lifted to basic instructions; otherwise, they can be used in module functions, which potentially allow the function to access arbitrary store addresses that should not be accessible by the module.

Instead, these two reference values are lifted to the administrative instructions with the same constructor name (adding a prefix for administrative instructions). There are important implications of this addition. Firstly, the following hierarchy – which was assumed in Wasm 1.0 in some proof structures – is now broken:

$$\text{values} <: \text{basic instructions} <: \text{admin instructions}$$

As a result, some of the proofs have to be modified to account for this change. More importantly, as these reference values hold explicit store addresses, they are no longer guaranteed to be well-typed under arbitrary environment. In particular, the value typing function typeof, which was once a computable function on its own, now needs to involve the store S to check the bound for the reference addresses. In the WasmCert-Coq mechanisation, all of the typeof operations and value typing relations are replaced by a new value typing relation value_typing that now refers to the store as well. This adds further technicalities to some proofs which were originally much easier.

Stack types, typing inversion lemmas, and soundness proofs The type system, especially the part about the type soundness proofs, is greatly impacted by the introduction of the operand types and stack types. The most straightforward method is to faithfully replace all original uses of the function type in the program fragment typing relations by the stack types and change all the proofs accordingly. However, after checking the future extensions in Wasm 3.0 and consulting with the specification editor Rossberg, it comes to my knowledge that the system of operand types and stack types is only added as an intermediary type-checking mechanism in Wasm 2.0. In Wasm 3.0, the real bottom type $\bot$ will be introduced as an extended part of the value type instead of living in the operand type only. A new *instruction type* construct replaces the purpose of stack types, but its definitions are similar to the function types instead of using a separate operand type – i.e. being a map from result types to result types. A subtyping relation is introduced for each of the value types, result types, function types, and instruction types, introducing a proper type lattice. As a result, a faithful mechanisation of the Wasm 2.0 operand and stack type system is in fact poor for forward compatibility.

In light of the knowledge above, I opted for a more ambitious approach in the Wasm 2.0 update which tries to represent the genuine subtyping system in Wasm 3.0 for maximum forward compatibility. This design choice is similar to the choice of the WasmCert-Coq model for Wasm 1.0, which allowed the result types to be a general list of types without length restrictions, and allowed a module to contain a general list of tables and memories.

Concretely, I extend the value types directly by the bottom type. Furthermore, as specified by the upcoming Wasm 3.0, I extended the subtyping relation from a relation between value types and

result types to a subtyping relation between *function types*, defined as follows:

$$\frac{C \vdash ts_{21} \leq ts_{11} \quad C \vdash ts_{12} \leq ts_{22}}{C \vdash ts_{11} \to ts_{12} \leq ts_{21} \to ts_{22}}$$

That is, the subtyping relation for function types is covariant on the type produced and contravariant on the type consumed. A corresponding typing rule in the type system is added, which allows a program fragment to be associated with any supertype of a function type that the program can be associated with:

$$\frac{C \vdash es : ft' \quad C \vdash ft' \leq ft}{C \vdash es : ft} \ \textsf{subtyping}$$

The subtyping relation is further lifted to a subtyping between instruction types:

$$\frac{C \vdash ts_{21} \leq ts_{11} \quad C \vdash ts_{12} \leq ts_{22}}{C \vdash ts_{11} \to ts_{12} \leq_{\textsf{instr}} ts +\!\!+\ ts_{21} \to ts +\!\!+\ ts_{22}}$$

which introduces a sort of "frame"-like condition that allows the new subtyping rule and the old structural subsumption rule in Figure 2.18 to be combined.

However, this rule is not the most general relation between function types that can be weakened to the other. In particular, the common frame added to the consumed and produced types *ts* does not have to be identical. Instead, it suffices that the result type prepended to the consumed types is a subtype of the result type prepended to the type produced.

Nevertheless, the type system defined using the above instruction subtyping rule can still capture this scenario, since the above subtyping rule can be applied multiple times; therefore, the type system effectively defines a "real" subtyping relation which is the transitive closure over the above subtyping relation. Instead of using this approach, I spell out the most general relation fully in the Coq mechanisation as follows:

$$\frac{\begin{array}{r}\exists ts, ts', ts'_{11}, ts'_{12}. \quad ts_{21} = ts +\!\!+\ ts'_{11} \quad ts_{22} = ts' +\!\!+\ ts'_{12} \\ C \vdash ts \leq ts' \quad C \vdash ts_{11} \to ts_{12} \leq ts'_{11} \to ts'_{12}\end{array}}{C \vdash ts_{11} \to ts_{12} \leq_{\textsf{weaken}} ts_{21} \to ts_{22}} \ \textsf{weakening}$$

This relation describes the most general condition where the function type $ts_{11} \to ts_{12}$ can be weakened to the function type $ts_{21} \to ts_{22}$. The following general weakening rule for the fragment typing relation can then be proved as a lemma:

$$\frac{C \vdash es : ft' \quad C \vdash ft' \leq_{\textsf{weaken}} ft}{C \vdash es : ft} \ \textsf{weakening}$$

With the above design, I now have the tools to renovate the old typing inversion lemmas. Instead of the separate typing inversion lemmas in the design of the Wasm 1.0 type soundness proof, I define the following *principal typing relation*, relating a program fragment to a candidate *principal* type under a given typing context C:

```
Definition be_principal_typing (C: t_context)
```

```
  (be: basic_instruction) (tf: instr_type) : Prop :=
match be with
| BI_const_num c =>
    tf = (Tf nil [::T_num (typeof_num c)])
| BI_ref_null t =>
    tf = (Tf nil [::T_ref t])
| BI_ref_is_null =>
    exists t, tf = (Tf [::T_ref t] [::T_num T_i32])
| BI_ref_func x =>
    exists t, tf = (Tf [::] [::T_ref T_funcref]) /\
           lookup_N (tc_funcs C) x = Some t /\
           List.In x (tc_refs C)
| BI_br k =>
    exists tx ty ts,
    tf = (Tf (tx ++ ts) ty) /\
      lookup_N C.(tc_labels) k = Some ts
| ...
end.
```

Despite its name, the above relation in fact does not associate a program fragment uniquely with an instruction type due to the stack-polymorphic instructions such as **br** and **return**. In this sense, the name of the above relation may be considered a misnomer. However, the principal typing relation indeed describes the most general condition that needs to be satisfied. I prove the following typing inversion *scheme*, which states that for any typing relation to hold for an instruction, the instruction type associated to the expression must be a supertype of *some* principal type that can be associated to the instruction:

Proposition 3.10.1. *If*

$$C \vdash e : ft$$

then

$$\exists ft_{\mathsf{p}}.\, ft_{\mathsf{p}} \leq_{\mathsf{weaken}} ft \wedge \mathsf{be_principal_typing}(C, e, ft_{\mathsf{p}}).$$

This lemma is easily proved by a straightforward induction on the typing relation. Combined with the old typing inversion lemmas for program compositions, I can obtain a similar level of automation for extracting constraints from typing predicates as what the old `Ltac` tactic could achieve – with an even smaller lines of code as well, due to the compact structure of the principal typing relation.

Losing invariant on typing context Recall an important lemma Lemma 3.5.3 from Section 3.5, that the typing context corresponding to the store and module instance stays invariant throughout the execution. This property was crucial when I established the preservation property, as it allowed me to reason about the original program and the program after a step under the same typing context.

Unfortunately, this property is broken in Wasm 2.0. This is due to the change which added the full types of runtime instances into the store. In particular, the table and memory instances carry their table and memory types respectively. However, the table types and memory types are determined

by both the optional maximum size, and a minimum size which is defined to be equal to the current size of the content of the table or memory. As a result, whenever a table or a memory grows, their minimum sizes change accordingly, which affect their types. Since the typing context contains the information about the types of the tables and memories accessible by the current running module instance, the typing context is no longer an invariant during execution.

The breaking of this property was unintended. In fact, the Wasm 2.0 specification has a mistake (which I introduce in Section 3.10.3) which incorrectly assumed that the table and memory types would stay invariant, and defined the store extension property based on this assumption. After this discovery, the store extension property has to be relaxed to allow the table and memory types to only be an *extension* of the old types by allowing the minimum sizes to grow. Correspondingly, instead of proving the invariance of the typing context C, I define a *context extension* relation in the Coq mechanisation, which captures this possibility of extension of table and memory types in the context. This had a large impact on the preservation proof, since I can no longer use the same typing context for the program fragment before and after a step of reduction.

Fortunately, the main approach of the preservation proof is unaffected: since the minimum sizes are unused in any typing rules (as long as they stay below the maximum sizes), I am able to prove that the above extension of typing context preserves the typing relation of a program fragment. Nevertheless, a lot of additional technical work is required due to this mishap.

Prior to the update to Wasm 2.0, the progressful interpreter has replaced the original WasmCert interpreter and the progress proof, which will be introduced shortly in Chapter 4. As a result, the progress proof is unaffected by the above as it has been merged into the progressful interpreter. Overall, the subjective experience of updating the interpreter was smooth. In fact, the most tedious part of the update is to correctly formulate the new definitions introduced by the new version of Wasm and making sure they are organised in a sensible structure for the best forward compatibility.

Forward-compatibility is extremely crucial for the update process of WasmCert-Coq to remain smooth. However, the relevant insights and knowledge about the design choices can only be gained by discussing with the designers and editor of the specification. This motivates a world where the designers of the feature extensions could participate in the standardisation process more deeply, which I will come to a short discussion in Chapter 6 as part of the future work on the SpecTec DSL.

3.10.3 Specification Mistakes Found

Just as Watt [112] discovered errors in WebAssembly's draft type system, my extension of the existing WasmCert-Coq to the WebAssembly 2.0 feature set has uncovered several errors in the official specification.

Composition and Subsumption

The Wasm 2.0 specification introduces a new concept of subtyping between value types given by $\leq$, and attempts to refactor subsumption (with subtyping) and composition into a single composite typing rule, as follows:

$$\frac{}{C \vdash [] : ts \to ts}\ \mathsf{emp-poly}$$

$$\frac{C \vdash es : ts_1 \to ts_0 +\!\!+\ ts' \qquad ts' \leq ts \qquad C \vdash e_N : ts \to ts_3}{C \vdash es +\!\!+\ e_N : ts_1 \to ts_0 +\!\!+\ ts_3}\ \mathsf{seq-poly}$$

However this typing rule is erroneous. As a counter-example, consider the instruction ($\mathbf{block}\ \{ts' \to ts\}\ []$) — a block instruction with an empty body. If $ts' \leq ts$, then this instruction should successfully type-check, but the typing rules above fail to support this. This error in the official standard has been acknowledged by Wasm's specification editor. While an official fix is in progress, WasmCert-Coq instead uses more traditional subsumption and composition typing rules, drawn from earlier drafts of the type system.

Module typing

Wasm allows the pre-declaration of *active element segments*, which populate a module's function table with a list of pre-declared functions at startup time. With Wasm 2.0, the typing rule for these segments was re-written in anticipation of future features which would generalise their structure. However this revised typing rule omitted a key step in constructing a context representing the types of global module declarations, resulting in a rule which implied that every active element segment was ill-typed. This error was identified concurrently by both ourselves and an independent standards contributor, and has now been fixed.

Missing component of typing context

Wasm 2.0 introduces a new component of the typing context, $C.\mathsf{refs}$, representing a declared list of functions which are permitted to escape the boundaries of the module as dynamic references. However, due to an editorial oversight, $C.\mathsf{refs}$ was inconsistently propagated through the typing rules, making certain typing rules erroneously strict. These corrections have been adopted into the specification.

Memory soundness

The appendix of the Wasm specification includes auxiliary definitions necessary for the official statement of the intended soundness properties of the Wasm type system. However one of these auxiliary definitions declared that a Wasm memory has a runtime type based on its current size which remains unchanged during program execution, without correctly accounting for the possibility of memory size increasing due to the **memory**.**grow** instruction. After I discovered this error, a fix was adopted into the specification.

Missing subsumption rule for values

Another issue with the soundness appendix, the following subsumption rule on the runtime type of values was inadvertently elided but assumed to exist in other definitions.

$$\frac{S \vdash v : t \qquad \vdash t : \text{ok} \qquad t \leq t'}{S \vdash v : t'} \ \mathsf{val-sub}$$

3.11 Testing

To conclude this chapter, I discuss a complementary method of establishing trust for a codebase – testing.

As discussed in various places in the chapter, testing can be important even for a mechanised and verified codebase. The following summarises the main reasons for WasmCert-Coq:

- Verification by proofs is, in the end, proving properties within or against a set of *reference definitions* manually encoded by humans in theorem provers. It is possible that both the reference definitions and the verified functions contain errors in a "self-consistent" way, which allows the desired properties to be proved but in fact differing from the intended specification. This is especially the case when the reference definitions and other functions are written by the same group of researchers or developers (which is frequently the case for verification projects due to the smaller communities), since the same misinterpretation of the official or intended specification is likely to occur for both versions.

- Certain parts of implementations do not have any reference definitions to be tested against, or the reference definitions are facing essentially the same complexity of the functions being implemented. This is the case for the numeric operations and binary format parser, which have been discussed in Section 3.8 and Section 3.9.

- Verification provides *transferrable* certification of the desired properties for an implementation, thereby enhancing the trust of correctness from the other developers. However, testing is still the preferred method for many developers in the industry, which can also be seen as an orthogonal but important source of trust for a codebase. For industrial users of the extracted runtime from WasmCert-Coq, testing is an important prerequisite before adoption.

The WasmCert-Coq extracted runtime[5], which includes the full pass of binary parsing, module instantiation, and function execution, was tested against the test suite for WebAssembly in the official repository [85]. Several bugs were found in this process, which were mostly related to the unverified features, including the binary parser and the numeric wrappers. However, there were also two bugs in the semantically parts of the codebase, which were not captured by the soundness properties.

3.11.1 The WebAssembly Test Suite

WebAssembly's test suite contains a comprehensive collection of test files in the wast format, which is a syntax extension of WebAssembly's text format to include a set of *assertions* as testing obligations on top of text-format modules. This format is not formally specified as part of the

[5]The work on testing was done after the adoption of the progressful interpreter, which will be introduced in Chapter 4. Therefore, the extracted runtime tested is the one from the progressful interpreter.

WebAssembly standard, although a syntax description exists in a readme file in the repository containing the reference implementation. Figure 3.15 provides a selected set of the syntax extension of the wast format based on earlier definitions in Chapter 2, primarily from Figure 2.4.

$$
\begin{array}{rl}
\text{(action)} & \mathit{action} ::= \textbf{invoke}\ \mathit{name}^?\ \mathit{string}\ \mathit{vs} \mid \\
 & \qquad\qquad \textbf{get}\ \mathit{name}^?\ \mathit{string} \\
\text{(failure message)} & \mathit{failure} ::= \mathit{string} \\
\text{(numeric pattern)} & \mathit{num_pat} ::= \textbf{nan} : \textbf{canonical} \mid \textbf{nan} : \textbf{arithmetic} \\
\text{(expected result)} & \mathit{result} ::= v \mid t.\textbf{const}\ \mathit{num_pat} \mid \ldots \\
\text{(assertions)} & \mathit{assertion} ::= \textbf{assert_return}\ \mathit{action}\ \mathit{results} \mid \\
 & \qquad\qquad \textbf{assert_}\{\textbf{trap}/\textbf{exhaustion}\}\ \mathit{action}\ \mathit{failure} \mid \\
 & \qquad\qquad \textbf{assert_}\{\textbf{malformed}/\textbf{invalid}/\textbf{unlinkable}/\textbf{trap}\}\ \mathit{module}\ \mathit{failure} \\
\text{(command)} & \mathit{cmd} ::= \mathit{module} \mid \mathit{action} \mid \mathit{assertion} \mid \ldots \\
\text{(script)} & \mathit{script} ::= \mathit{cmds}
\end{array}
$$

Figure 3.15: Syntax extension of the wast format

Each wast test script consists of a sequence of commands, which can be one of the following:

- a module, requiring it to be instantiated;
- an action, which either invokes an existing function in the Wasm store or retrieves the value of a global variable;
- an assertion, which either asserts that a certain action returns a specific result, or asserts that an action or a module instantiation will fail due to various reasons.

It is important to note that the effects of all assertions, including the failure assertions, carry over to the next. For example, any function invocation that modifies the Wasm store result in the following commands being executed based on the modified store.

The official test suite has coverage over all Wasm 2.0 features as well as tests for each extension proposal that has not become part of the standard yet.

3.11.2 Testing WasmCert-Coq

WasmCert-Coq provides its own binary parser on top of module instantiation and executable interpreter to be tested. However, several features have to be additionally implemented to run the test suite:

- A minimal host that handles module imports and exports needs to be implemented in OCaml. This caters to the test scripts targeting module imports and exports, as well as inter-module function calls.
- A method for handling Wasm modules in text format is required. This is because the *module* construct can be given in either the binary or text format, while WasmCert-Coq only implements a binary format parser. This is resolved by converting all modules in text format to binary format using the functionalities from the reference implementation.

- Lastly, some OCaml implementation is required to verify whether each assertion has been fulfilled.

  The success assertions are relatively straightforward, except for those that assert that the return values must belong to specific categories of NaN values determined by their payloads (referred to as the *canonical* and *arithmetic* NaNs by WebAssembly).

  The failure assertions are less direct to implement, since the runtime from WasmCert-Coq only distinguishes among errors from different stages (parsing/instantiation/invocation), but not further details within each stage. As a result, the OCaml test suite runner accepts that an assertion has been passed as long as an error is returned in the correct stage.

  In addition, **assert_exhaustion** requires the corresponding function call to exhaust the function call stack. The corresponding tests usually involve some infinitely recursive function calls. However, the WebAssembly specification does not state explicitly the maximum size of the function call stack. WasmCert-Coq originally used a general linked list representation of the call stack, which allows the call stack to grow infinitely[6]. As a result, instead of returning a stack exhaustion error, WasmCert-Coq's runtime simply runs indefinitely. This resonates to the intention of the exhaustion tests, which force implementations to reliably return an error in these cases.

  To deal with these tests, an artificial limit of 256 on the depth of the call stack is imposed on the maximum size of the call stack in the extracted OCaml runtime. This limit is imposed for running the test suite only. While this seems overly restrictive (as it can rule out valid recursive programs that require more stack space), is taken from the corresponding limit in the reference implementation, therefore ensuring that the expected results can be observed while running the test suite.

3.11.3 Bugs and Inadequacies Discovered

The WasmCert-Coq extracted runtime was tested against the core test suite of Wasm 2.0, i.e. all tests except the SIMD tests. In this process, a handful of bugs in the WasmCert-Coq mechanisation were discovered, alongside several inadequacies of the extracted runtime, which is discussed in this section.

Parser and Numeric Wrappers Most of the bugs discovered are related to the binary format parser and the numeric wrappers, as mentioned earlier in the chapter. This is expected, given that they are the two major parts of the codebase that were not verified.

The parser bugs were mainly of the form of missing a validity check that allows some modules to be parsed where they shouldn't, which do not affect the behaviour of valid modules. However, there are also bugs that do.

One example is the one for the **call_indirect** instruction in Wasm 2.0. In Wasm 2.0, multiple tables are allowed to be present in a module. As a result, the instruction **call_indirect** *typeidx* is

[6] Strictly speaking, up to the size of RAM available.

extended to take an additional table index, instead of defaulting to operate on table 0. The new AST of the instruction becomes

call_indirect *tableidx typeidx*

However, the binary format places the two indices in the reverse order, as displayed in Figure 3.16.

instr	::=	0x00	⇒	unreachable
	|	0x01	⇒	nop
	|	0x02 bt:blocktype $(in$:instr$)^*$ 0x0B	⇒	block bt in^* end
	|	0x03 bt:blocktype $(in$:instr$)^*$ 0x0B	⇒	loop bt in^* end
	|	0x04 bt:blocktype $(in$:instr$)^*$ 0x0B	⇒	if bt in^* else ϵ end
	|	0x04 bt:blocktype $(in_1$:instr$)^*$ 0x05 $(in_2$:instr$)^*$ 0x0B	⇒	if bt in_1^* else in_2^* end
	|	0x0C l:labelidx	⇒	br l
	|	0x0D l:labelidx	⇒	br_if l
	|	0x0E l^*:vec(labelidx) l_N:labelidx	⇒	br_table l^* l_N
	|	0x0F	⇒	return
	|	0x10 x:funcidx	⇒	call x
	|	0x11 y:typeidx x:tableidx	⇒	call_indirect x y

Figure 3.16: Binary format of Wasm 2.0 control instructions

This reversal was missed in the initial coding of the binary parser, therefore resulting in observable differences in execution behaviour even for valid Wasm modules.

Type System Despite the soundness properties verified, there are two mistakes in the type system revealed by testing. The mistakes were due to misinterpretation of the specification in both cases. Both of these mistakes were in the Wasm 2.0 version only, partly due to the scale and complexity of the update, causing human misinterpretation to be more likely.

- The first mistake involves generating the C.refs field of the typing context for a Wasm module. The following description is given in the specification: However, it is up to the reader

  ◦ C.refs is the set funcidx($module$ with funcs $= \epsilon$ with start $= \epsilon$), i.e., the set of function indices occurring in the module, except in its functions or start function.

  Figure 3.17: Specification of refs field of typing context

  to correctly locate all possible locations where function indices could appear in a module. The initial implementation in WasmCert-Coq missed the potential occurrences in module memory initialisers and module exports.

- The second mistake involves the **br_table** instruction in Wasm 2.0. Recall the updated typing rule for **br_table** from Section 3.10:

$$\frac{\vdash ts \leq C.\mathsf{labels}[ld] \qquad \vdash (ts \leq C.\mathsf{labels}[l_i])^*}{C \vdash \textbf{br_table}\ ls\ ld : (ts_1 +\!\!+\ ts +\!\!+\ [\textbf{i32}]) \to ts_2}\ \mathsf{br_table}$$

  The two subtyping relations were originally equality relations in Wasm 1.0. WasmCert-Coq defines this rule faithfully in its inductive type system. The type checker for this rule

takes a different but necessary approach due to its executable nature. It first checks that all $C.\textbf{label}[l_i]$ are equal to the same value ts; if this is satisfied, then check that the corresponding types can be consumed from the stack.

In the update to Wasm 2.0, this consumability check was originally replaced by a check for the existence a common label type ts which can be consumed. However, this is incorrect, since the individual labels do not have to be equal to each other in Wasm 2.0. Instead, it is merely required that they have to be subtypes of the types ts to be consumed. The type checker correctness proof would normally have caught this mistake. However, the inductive typing rule is also updated incorrectly. This renders the proof becoming the correctness of the type checker against an incorrect underlying type system.

Inefficient Indexing Coq, like other theorem provers, defines its natural numbers and lists in simple inductive structures:

```
Inductive nat: Type :=
| O
| S : nat -> nat.

Inductive list (T: Type) : Type :=
| nil
| cons: list T -> list T.
```

These definitions resemble linked lists and are sound mathematically. However, the extracted programs based on these structures are terribly inefficient. For example, a simple lookup operation $arr[n]$ has time complexity $O(\mathsf{min}(n, |arr|))$. This method is therefore unrealistic for critical data structures such as linear memories, where the length of lists can be large.

In the context of a WebAssembly runtime, there are further complications that lead to inefficiencies even when the lists are small. Consider the **local**.**get** n instruction, which takes an **i32** integer n as an argument. Execution of this instruction looks straightforward: by the corresponding reduction rule of the operational semantics, it is only required to return the result of the lookup $F.\mathsf{locals}[n]$. However, since Coq's default list lookup function only takes nat as arguments, the index n is first converted from **i32** to nat before the lookup takes place. The subtle inefficiency is due to this conversion, which is itself $O(n)$. This is ostensibly a non-issue, because Wasm's type checking function prevents out-of-bound lookups from being executed. However, the type checking function itself performs a similar lookup in the typing context, which requires the same inefficient conversion from **i32** to nat. As a result, the extracted runtime hangs on tests that contain **local**.**get** for extremely large out-of-bound indices, instead of correctly returning a type error.

One potential solution to this issue is to replace all list indexing functions in Coq by a custom version that directly takes an **i32** as an argument. However, this method is not only tedious to implement, but also means abandoning many convenient auxiliary lemmas for list operations from the standard library.

A better solution is to implement a custom efficient version of list lookup without replacing the

default list lookups in the mechanisation. Coq's extraction mechanism can be used to directly set the extraction target of the default list lookup function to the efficient version. To ensure correctness of this extraction target, a proof of equivalence between the efficient version and the default library function is implemented. This avoids the inefficiency in indexing while enjoying the benefit of using the auxiliary lemmas in Coq's standard library.

Inefficient Memory Implementation The method above caters to the tests with large out-of-bound indices. However, for valid indices, the lookup operation, or *random access*, still has a linear time complexity. Moreover, update operations required by the t.**store** instruction are also costly even for an update at a single point, as an entirely new copy of the list is created for each update. This does not scale to programs that perform extensive memory operations. This is exacerbated by the fact that Wasm memories are allocated in pages of $2^{16} = 65536$ bytes – therefore, the minimum size of memory that can be allocated is 2^{16} bytes, which is already a considerable burden for the inefficient runtime.

There are various methods to address this issue. In an extension to WasmCert-Isabelle [115], Watt et al implemented a completely separate monadic interpreter, whose stateful operations are represented by monadic updates in Isabelle with corresponding separation-logic specifications. This allows the actual stateful operations to be implemented in the target language of extraction – OCaml in their case, which is vastly more efficient. More importantly, the state, including both the Wasm store and the frame containing the local variables, is implemented in OCaml as well, avoiding the inefficient linked-list representations in theorem provers.

However, this approach has a shortcoming. Due to all stateful operations being implemented outside theorem provers and only the specifications remain available, the correctness of the interpreter is verified using separation logic instead. This leads to a complex verification process on top of the already non-trivial monadic interpreter, which is difficult to maintain when the underlying semantics evolve. Moreover, Watt's verification was performed with the aid of the Verification Condition Generator (VCG) of the Sepref library [61], which is also Isabelle-only and important in providing proof automation. Although there's a separation logic model Iris-Wasm (which will be discussed in Chapter 5), the model lacks such an automation mechanism. Therefore, verifying programs as large as an entire interpreter is excessively laborious.

Instead, WasmCert-Coq chooses a different approach which minimises the modification required to the existing codebase while achieving reasonable performance.

Before introducing the approach, it is important to understand why a direct extraction of Coq's lists to OCaml's efficient mutable arrays does not work. Consider the following higher-order function in Coq, which takes a list and two functions, and returns a tuple representing the results of applying the two functions to the list separately.

```
Definition apply2 {T: Type} (l: list T) (f g: list T -> list T) : list T * list T :=
    (f l, g l).
```

If Coq's list is extracted to OCaml's Array (mutable arrays), the function apply2 extracts to the following OCaml function:

```
(** val apply2 :
    'a1 Array -> ('a1 Array -> 'a1 Array) -> ('a1 Array -> 'a1 Array) -> 'a1
    Array * 'a1 Array **)
let apply2 l f g =
  ((f l), (g l))
```

However, this seemingly equivalent function behaves differently if f modifies the array, in which case g will compute based on the modified array l instead of the intended original version in Coq. One solution to this is to copy the content of l before applying the functions f and g; however, copying the array is inefficient and defeats the purpose of using arrays in the first place.

The proper solution without drastically changing the code in theorem provers (as opposed to what Watt did in WasmCert-Isabelle [115]) is to extract Coq's lists to efficient *persistent* arrays, where every update creates a new version of the array *without copying the entire array*. WasmCert-Coq uses the implementation of persistent array using "Baker's trick" from Coq's kernel implemented by Jean-Christophe Filliâtre[49], which consists of a set of parameterised definitions and properties for persistent array operations in Coq and a corresponding set of OCaml implementation. The method used was described by Conchon and Filliâtre [23], which maintains a mutable array with a *version tree*, each node representing a version of the array and tracking the difference between that version and the parent version. Each update operation adds a new node to the tree and *reroots* the tree to the most recently updated version, updating the version tree along the path and modifying the base mutable array as well. As a result, lookups and updates to the most recent version of the array are $O(1)$, and operations to the historical versions are $O(d)$, where d is the distance between the version to the most recent version in the version tree.

In WasmCert-Coq's extracted runtime, the linear memories are extracted to use the above implementation of persistent arrays. WasmCert-Coq's only operations that operate on the memories are load, store, and grow, none of which require keeping track of historical versions. Therefore, any historical versions of memories are only ephemeral within the functions that operate on states themselves, and all operations take the most recent version of the persistent memory as argument, resulting in excellent efficiency. It should be noted that the performance is still worse than Watt's monadic state solution, as the method based on persistent arrays cannot enjoy the optimisation benefits which might be possible for low-level memory operations such as memset. However, Watt's monadic state interpreter essentially requires a complete overhaul on all the stateful operations, whereas the method I presented here requires no adaptation to the existing codebase to be used.

The last remaining issue is that OCaml's mutable array cannot be inherently resized. On the contrary, the **memory**.**grow** instruction requires extending the array. Creating a new array and copying the content from the old one is too inefficient for **memory**.**grow**. Instead, WasmCert-Coq implements a dynamic array data structure on top of the parametric persistent array described above using the classical method of *geometric expansion*. In this method, whenever a resizing of the array is required, a new array of a fixed multiple (typically 2x) size of the old array is reserved, with the content from the old array copied over. This method provides an amortised $O(n)$ time complexity for inserting n elements to the array. Lastly, WasmCert-Coq also proves the correctness

of its implementation of dynamic vector based on the assumed properties of the persistent array operations.

With the above bugs fixed and inadequacies addressed, the extracted runtime of WasmCert-Coq passes all of the Wasm 2.0 core test suite in under 2 minutes in total[7].

3.12 Related Work

There is a wide body of existing work on the mechanised specification of programming language semantics [96]. Norrish presents a mechanisation of a fragment of C HOL [84]. The CompCert project [66, 67] has a mechanisation of a large fragment of C in Coq, called Clight [15], making simplifying assumptions regarding some details of the C memory model which are not relevant to the CompCert compilation correctness proof. Lee et al mechanise in Twelf [90] an "internal language" with "equivalent expressive power" to Standard ML, the semantics of which they formalise via elaboration [63]. CakeML [60, 107] includes a mechanised semantics in HOL4 for a large fragment of Standard ML (minus functors). OCaml Light [86] is a mechanised semantics in HOL4 of a core subset of OCaml. Jinja [56], Java$_S$ [106], and Featherweight Java [48] provide mechanised fragments of Java. JSCert [16] is a Coq-mechanised specification of a large subset of ECMAScript 5, handling all core constructs but leaving out "library objects" such as `Array` and `Number`. Guha et al give a JavaScript semantics through elaboration to a mechanised semantics in Coq of a core calculus [43]. These mechanisations provide machine checked models of the target specifications, and often verification of desired properties of the underlying semantics. Theorem provers – such as Coq [12], Isabelle/HOL [82], and Agda [20] – are popular for building mechanisations of programming languages.

It is usual for mechanised specification based on an interactive theorem prover to target an interesting language core or abstraction for mechanisation, due to the ambiguity, complexity, or size of the full language definition. For example, mechanisations such as CLight [15] formalises a large subset of C featuring most of the types and operators related to the verification of CompCert compliation, and JSCert [16] only formalises JavaScript up to ECMAScript 5. In contrast, WasmCert-Coq is able to closely follow the *entirety* of the Wasm semantics directly as stated in the standard, except for minor intentional adaptations for better forward compatibility. This is made tractable by its compact design and official formalisation. This is a double-edged sword: WasmCert-Coq mostly does not need to make its own interpretation of the standard due to its formality. On the other hand, the lack of room for choices due to closely following the standard also makes mechanisation and proofs more awkward than necessary sometimes.

Huang presents a partial Coq-mechanisation of Wasm in his M.Sc project, independently named WasmCert [46]. We have been given permission to use the WasmCert name. As already discussed, the initial port of WasmCert-Coq comes from Watt's Isabelle/HOL mechanisation [112].

Some lightweight approaches with more gentle learning curves than theorem provers are possible: e.g. the K framework [98] is used to define term-rewriting models of significant fragments

[7] Spec of the testing machine: Ubuntu 24.04, AMD Ryzen 7 5800X 3.80 GHz, 8GB RAM.

of C [29], Java [17], JavaScript [89], and PHP [31]; Cerberus [72], which defines an elaboration semantics for a large fragment of C with a core calculus defined in the Lem specification language [78]; and JaVerT, an analysis tool for JavaScript programs [100, 33], where the semantics of JavaScript is defined by an elaboration to a simpler intermediate language.

There are several downstream projects that use the WasmCert-Coq mechanisation, some of which I have been in contact with as the chief maintainer of WasmCert-Coq. CertiCoq-Wasm [71] is a WebAssembly backend for CertiCoq developed by Meier et al. This project provides a verified extraction from Coq to WebAssembly by targeting WasmCert-Coq's AST, followed by using WasmCert-Coq's binary format printer to convert the generated AST to Wasm binaries. Through their work, some mistakes in the binary format printer were found and fixed. The other significant downstream project is the Iris-Wasm program logic [91], which is my joint-work with others. Iris-Wasm provides a higher-order separation logic for Wasm using the Iris separation-logic framework. This will be discussed later in Chapter 5.

SpecTec[119] describes a Domain-Specific Language (DSL) for Wasm specification which acts as a single source of truth, which aims to automatically generate Wasm specification artefacts and mechanised semantics in the future, thereby providing a maintainable way to produce mechanised definitions for Wasm. In a longer future, it is expected that WasmCert-Coq will move to SpecTec, where the downstream projects can choose to move as well. This will be discussed in slightly more details in Chapter 6.

Aside from finding errors in the Wasm specification, WasmCert-Coq has also helped improve its dependencies. In particular, it helped locate a major inefficiency in the Parseque parser library where the `alt` combinator was not properly evaluated lazily, which I discovered while attempting to address an inefficiency of WasmCert-Coq's own binary format parser. The resolving of this issue, together with a refactoring of the module parsing process, brought the module parsing time from approximately 2s down to negligible (<10ms).

3.13 Acknowledgements

Besides my personal effort, several individuals have also contributed to the WasmCert-Coq project, mostly during its inception. In 2019, Martin Bodin and Jean Pichon-Pharabod, with consults from Conrad Watt, drafted an initial version of definitions for WasmCert-Coq, porting the definitions from Conrad Watt's WasmCert-Isabelle[112], including the definitions of the operational semantics, type system, interpreter and type checker. They also implemented the first version of numerics using CompCert's numerics [66, 19], and the first version of the binary format parser and printer using Parseque [6]. I completed the interpreter correctness and the type soundness proofs for that initial ported version of WasmCert-Coq, and updated the semantics from the Isabelle port to Wasm 1.0. This work was done during my Master project in 2020, under the technical supervision of Martin Bodin. Later in 2021, I completed the type checker correctness proof on top of a refactored version of the existing codebase with some bug fixes. Together with the Isabelle counterpart, this version of mechanisation has been published in 2021 as *Two Mechanisations of WebAssembly 1.0* [114].

From there onwards, Martin Bodin and Jean Pichon-Pharabod have moved on to other projects and were mostly inactive on the project, while I continued to act as the chief developer and maintainer of WasmCert-Coq, designing and implementing the rest of the contents discussed in this chapter while also revamping the original versions of almost all components of the codebase. These works have not been described in any publications in details, besides a brief description of the Wasm 2.0 update and the progressful interpreter (to be discussed in Chapter 4) in Rao et al. [92].

A master student Liqing Yang proved a partial version of module instantiation soundness for Wasm 1.0 in her Master project under my technical supervision, which establishes the well-typedness of the post-instantiation store, but does not include the other properties such as store extension or the frame typing relation. This was later refactored and completed by me to form the full module instantiation soundness result described in Proposition 3.7.1, and eventually incorporated into WasmCert-Coq.[8]

WasmCert-Coq has also received support from the research community. I consulted Conrad Watt and Andreas Rossberg regarding some particular choices during my implementation of the Wasm 2.0 update, while communicating the mistakes found in the specification. Wolfgang Meier helped discover some bugs in the binary format printer during their work on CertiCoq-Wasm introduced in Section 3.12, who also helped build the first version of continuous integration of WasmCert-Coq on GitHub. Other researchers including Bas Spitters, Karl Palmskog and Guillaume Allais have provided notices in deprecated dependencies or help in dealing with certain license issues. Despite not contributing to the mechanisation codebase directly, the members from the research community have helped influence WasmCert-Coq into a more well-maintained project.

[8] I was also the technical supervisor of another Master project of Henit Mandaliya related to WasmCert-Coq, who proved the type preservation property of a draft version of Wasm 2.0 based on the existing proofs of Wasm 1.0. However, this was eventually not used due to the refactorings and general restructuring of the proof designs required for Wasm 2.0.

Chapter 4

Progressful Interpreters for Efficient Mechanisations

Our mechanisation of the W3C WebAssembly (Wasm) standard [44], WasmCert-Coq [114], has so far been able to keep pace as the standard has evolved through various drafts up to the 2.0 release since its initially published 1.0 version [114]. Maintaining this mechanisation for future versions of Wasm is essential to ensure that new features continue to benefit from the same verification guarantees as those in the 1.0 feature set, as discussed in Chapter 3.

However, updating the mechanisation is a complex and labour-intensive process, requiring modifications across multiple definitions and proofs for each new feature extension. For Wasm, the jump from Wasm 1.0 to the forthcoming Wasm 2.0 standard is particularly large, almost doubling the instruction set of the language, as introduced in Section 3.10. It is therefore important to look for methods to alleviate the burden of performing such an update.

This chapter discusses this topic by investigating techniques and designs to combine various definitions and proofs within a single maintainable structure. WasmCert-Coq eventually implements a new design of a dependently-typed *progressful interpreter*, which unifies the interpreter's definition, its soundness proof, and the type progress proof into a single coherent structure.

4.1 Background

In the classic formalisation of a typed operational semantics, as followed by the W3C Wasm standard, both the type system and runtime operational semantics are defined separately as inductive relations, and a statement is made of the desired *syntactic type soundness* properties (*progress* and *preservation*) which relate them [117]. This leads to a natural set of core artefacts which a mechanisation of such a semantics will aim to support:

- the mechanisation of the language definitions themselves;
- a proof of the type system's progress and preservation properties;

- a definition of a simple executable interpreter, since the inductive operational semantics is not directly executable;
- a proof that the interpreter is sound with respect to the operational semantics.

Each of these items must be separately defined and maintained – as done in our WasmCert-Coq mechanisation of Wasm 1.0 [114] in Chapter 3.

Chapman et al. [21] and Kokke, Siek, and Wadler [57] observed in their Agda work that a constructive proof of progress over a small-step semantics essentially embodies a sound one-step interpreter, since given a well-typed input configuration it represents a computational "recipe" for obtaining an output configuration that is allowed by the language's small-step reduction relation. By composing this step with a constructive proof of preservation and iterating, a simple verified interpreter is obtained "for free". Since Wasm mechanisations already commit to maintaining an analogous type soundness proof, the idea that a sound interpreter may be derived "for free" from this proof, without needing to maintain a separate definition (and verification of this definition), is enticing.

In this chapter, we push the techniques of Kokke, Siek, and Wadler [57] to their limits by applying their approach to the WasmCert-Coq mechanisation of the W3C Wasm 1.0 standard introduced in Chapter 3. We convert its type soundness proofs to be fully constructive and therefore successfully derive an interpreter from these proofs. This approach allows us to eliminate WasmCert-Coq's separate interpreter definition and correctness proof. In addition, we execute this interpreter end-to-end by composing it with WasmCert-Coq's *verified* type checker and parser. In this process, we manage to identify multiple ways in which innocuous choices in constructing the type soundness proofs can severely degrade its runtime performance.

However, optimising the inefficiencies in the derived interpreter is challenging. One major limitation of the derived interpreter from type soundness proofs is that we lose direct control over the structure and internal representation of the interpreter. As a result, we can only indirectly optimise its performance by modifying the proofs. Moreover, the proof tree of the input term's typing judgement needs to be explicitly represented in the interpreter's memory and manipulated at each step, since the interpreter is only allowed to take repeated steps by constructing a proof that each successor term is well-typed. Ideally, a solution which maintains our proof maintenance benefits without runtime manipulation of any typing proofs is preferred.

Instead of the above approach, we propose a different design that smoothly augments the interpreter with its certification instead of deriving an interpreter from the certification. In particular, we show in Coq that a formalised one-step interpreter over untyped Wasm terms can, with only modest effort, be augmented with a dependent type result, certifying the *contrapositive* of the progress property ("if the interpreter fails to step soundly according to the operational semantics, the input must be ill-typed"). This design not only eliminates the need for a separate proof of progress, but also the need for a separate proof of the interpreter's soundness with respect to Wasm's operational semantics. We refer to such an interpreter as a *progressful* interpreter.

The interwoven parts of the one-step interpreter which prove the above contrapositive property can

be erased during its extraction to executable code, since subsequent iterations of the interpreter do not require this property as input. As a result, the extracted interpreter does not suffer from any performance penalty compared to a traditional interpreter.

Moreover, this interpreter-based design allows us the same level of fine-grained control over the interpreter's internal state as a traditional interpreter. As a result, interpreter optimisations can be implemented directly instead of requiring proof manipulation.

To test this design, we report on our experiences of extending WasmCert-Coq to Wasm 2.0, using this approach to save effort in updating what were previously separate definitions and correctness proofs. Separately, we have also implemented several optimisations to Wasm's representation of evaluation contexts based on the work of Watt et al. [115] while still significantly reducing the proof burden in comparison to the more traditional approach of separate type soundness and interpreter correctness proofs.

Previous work on proof maintenance has also explored a closely related method that defines the semantics of an intrinsically-typed language in terms of a dependently-typed definitional interpreter [11, 94, 8, 97], which then uses the interpreter as the normative definition of the semantics of a language, therefore saving a separate definition of the semantics. We show that many of the benefits are still achievable in WasmCert-Coq, despite the fact that Wasm is not intrinsically typed, and our interpreter cannot be definitional – due to the design choices of the official Wasm formal specification.

In summary, we establish the following takeaways:

- Kokke, Siek, and Wadler [57]'s approach for deriving a sound interpreter from a constructive progress proof scales to the industrial W3C Wasm 1.0 standard, but optimising the performance of this interpreter is challenging. Moreover, the requirement that the derived interpreter traverses a proof of well-typedness of the input term at runtime represents an unavoidable performance penalty. (Section 4.2)

- Our *progressful* interpreter similarly allows multiple verification artefacts to be produced from a single definition, but gives us more control over performance. Our approach is related to formal properties most often discussed in the context of intrinsically-typed languages, but we realise them in Wasm, which is extrinsically typed. (Section 4.3, Section 4.4)

- We extend WasmCert-Coq [114] to the substantially larger Wasm 2.0 feature set, updating all related verification artefacts smoothly with the aid of our progressful interpreter. Through the process, several errors in the Wasm 2.0 specification have also been uncovered and reported to the specification editor, reinforcing the value of language mechanisations in improving industrial language standards. (Section 3.10)

4.2 Interpreter from Progress

In this section, we describe the method given by Kokke, Siek, and Wadler [57] for constructing a sound interpreter from a proof of progress for a general language with a small-step, inductively-

defined reduction semantics and type system. We apply this method to the W3C Wasm 1.0 standard, by adapting our mechanised progress proof in WasmCert-Coq introduced in Section 3.5, thus demonstrating that this method can scale to a full industrial language. While the automatic generation of this interpreter saves significant proof effort, we report on fundamental limitations on its performance that arise from the natural structure of the type soundness proofs, motivating our new approach in Section 4.3.

4.2.1 Methodology

Kokke, Siek, and Wadler [57] report that a constructive proof of the progress property (Proposition 2.6.2) can be used as a one-step interpreter, also noting that this correspondence has appeared in scattered folklore. The progress proof takes a configuration cfg and a proof of the configuration's well-typedness as input, and concludes that either some cfg' must exist such that $\mathrm{cfg} \hookrightarrow \mathrm{cfg}'$ is allowed by the semantics, or cfg is terminal. In a constructive setting, the progress proof therefore computes some cfg' as a witness which, together with its conclusion that $\mathrm{cfg} \hookrightarrow \mathrm{cfg}'$ is allowed by the semantics, represents a sound one-step interpreter for the language.

One complication to using a constructively-proven progress theorem as a one-step interpreter is the requirement that a proof of well-typedness of the initial configuration must be additionally provided as input. Moreover, to iterate the one-step interpreter, a constructive proof of preservation must be used to re-establish the well-typedness of intermediate configurations before each step.

To emphasize the computable nature of these constructive proofs, we present the shapes of the required statements for this approach to be applied to Wasm in Figure 4.1. The progress function

```
Definition progress (cfg: config) (t: config_type) (HType: (|- cfg: t)):
    {cfg': config & (cfg --> cfg')} + (terminal cfg).

Definition preservation (cfg cfg': config) (t: config_type)
                          (HReduce: cfg --> cfg') (HType: (|- cfg: t)) :
    |- cfg' : t.

Definition infer_type (cfg: config) : option {t: config_type & (|- cfg : t)}.
```

Figure 4.1: Definitions of progress, preservation, and type inference as computable functions

acts as described above, taking a configuration cfg, a configuration type t, and a proof term `HType` which associates cfg with the type t. It produces either a new configuration along with a proof term $\mathrm{cfg} \hookrightarrow \mathrm{cfg}'$ representing the soundness of the reduction step, or a proof that the configuration is terminal. Similarly, the preservation function takes the old and new configurations, a proof term representing that a sound reduction was carried out, and a proof term representing the well-typedness of the old configuration, producing a proof term representing the well-typedness of the new configuration. The verified type inference function, `infer_type`, takes a configuration as input and, if the type inference was successful, returns a configuration type along with a proof term representing the correctness of the returned type with respect to Wasm's type system. For the specific application on Wasm, no inference of types is required, as Wasm programs are already

annotated by its type signature. As a result, it is only required to implement a type-checking of Wasm programs against its annotated type which also produce the corresponding typing proof.

With the above definitions, we can describe the algorithm for creating a one-step and multi-step interpreter from the type soundness proofs, with the aid of the verified type inference function.

The interpreter begins with a type inference on the input cfg and returns an error if it fails. If typing succeeds, the interpreter alternates between applying the progress and preservation functions as follows:

- Apply the progress property to the configuration cfg and the associated proof term $\vdash \mathsf{cfg} : t$, obtaining either a new configuration cfg' and a proof term $\mathsf{cfg} \hookrightarrow \mathsf{cfg}'$, or a termination result;
- If applying progress does not result in termination, apply the preservation property to the old and new configurations, the reduction proof term, and the old configuration typing proof term to produce a proof term $\vdash \mathsf{cfg}' : t$ for the new configuration, allowing iteration to continue.

Note that similar to the original mechanised interpreter introduced in Section 3.4, an additional argument representing the *fuel* is added to satisfy Coq's syntactic termination check, which specifies the number of iterations to compute in this case.

This method demonstrates that, given a constructive proof of type soundness and a verified type inference function, a sound interpreter can be directly extracted with minimal effort. This is particularly valuable for maintaining mechanized artefacts, as it eliminates the need to maintain a separate interpreter and its soundness proof, leading to a smaller overall codebase to maintain. In Chapman et al. [21], this approach is used to generate an interpreter for a small core language based on System F. However, the interpreter cannot be run end-to-end due to the lack of a verified type checker.

For WasmCert-Coq, the above approach can be applied to the full definition of Wasm 1.0 by reusing the existing progress and preservation proofs of WasmCert-Coq introduced in Section 3.5, with only minor changes to make them fully computable. In addition, the verified type checker introduced in Section 3.6 and the binary parser of WasmCert-Coq allows the extracted interpreter to be executed end-to-end, via standard Coq extraction mechanisms [68, 69] to OCaml. WasmCert-Coq originally included its own separately-verified interpreter in addition to its type soundness proofs, which can therefore be made redundant.

We explain some further detail on the adaptations required to make the existing Wasm type soundness proof of WasmCert-Coq fully constructive and executable.

- Overall, the proofs were already almost in the right form — the main change involved replacing occurrences of Coq's regular existential quantification (in `Prop`) with the corresponding sigma-type consisting of the witness and the proof term (in `Type`) throughout the proof. One such example is the goal of the progress statement itself, as displayed in Figure 4.1.

- An example where the constructiveness of the type soundness proof has a non-trivial impact is for the interaction between the **br** instruction and the **label** administrative instruction. Recall from Lemma 3.5.8, that if a program fragment es is of shape $B^n[\textbf{br}\ (n+k)]$ and is well-typed, then some further constraints on the block context B can be proved, which helps to provide the necessary conditions for establishing a reduction for **br**.

  In one branch of the proof for the **label** case of the progress property, the proof uses Lemma 3.5.8 by doing a case analysis on whether the program fragment can be decomposed into the block context decomposition, as stated in the premise of the lemma. In the successful case, a corresponding reduction can be obtained by applying the lemma. On the other hand, a contradiction with the well-typedness of the program fragment can be constructed in the failure case.

  Without the constructiveness requirement, the above case split can be done by applying the excluded middle (since the program fragment is either of the required shape or not). However, a decision function of the decomposition is required for the above proof to be constructive. Fortunately, the existing WasmCert-Coq proof has already implemented this decision function with the corresponding proofs. As a result, no additional effort was required.

The above adaptations were minimally-invasive and completed within a handful of hours.

4.2.2 Limitations

The method introduced so far in the section allows a constructive type soundness proof to be extracted to an interpreter relatively easily. However, there are several downsides to using this method for a real-world language like Wasm. The primary concerns are the inefficient performance of the interpreter and the difficulty of implementing optimizations.

Inefficient performance Let us first visit the concrete statement of the transformed progress property in WasmCert-Coq:

```
Theorem t_progress: forall s f es ts hs,
    config_typing s f es ts ->
    terminal_form es +
    {s' & {f' & {es' & reduce s f es s' f' es'}}}.
```

It can be seen that, the progress property in WasmCert-Coq – which should now be viewed as a function by the interpreter – takes input consisting of the constituents of the Wasm configuration tuple, a type signature of the configuration, and a typing *derivation* of the corresponding typing judgement (config_typing), and returns either a proof that the input program is terminal, or a new configuration with a proof of the reduction predicate from the input configuration to the output provided.

However, recall from Section 3.5.2 that the progress proof is performed via an induction on the typing derivation. Because the interpreter relies on the constructive progress proof as a one-step evaluation function, the proof term representing the well-typedness of the current configuration (config_typing) *must be included in the runtime representation of the interpreter's state*. This

leads to a severe performance penalty, as the progress and preservation functions may need to traverse substantial portions of this proof term to produce the next configuration. Additionally, certain proof constructions, which may seem innocuous, have surprising knock-on effects on the performance of the derived interpreter. In particular, all of the above is true in Coq's setting even *after* the interpreter is extracted to OCaml — the proof term remains concretely represented in memory, and is traversed by the code of the extracted interpreter. Ideally, the extracted runtime should not dependent on the full typing derivation, since its purpose in the progress proof is to assert that the input program must be of a certain shape by producing a *contradiction* otherwise.

Difficulty in optimizations It is challenging to directly control the computational behaviour of the interpreter extracted from type soundness, as any structural changes to the interpreter can only be indirectly effected by changing the underlying proof structure.

In the following part of this section, we discuss in detail the causes of inefficiency of the interpreter extracted from type soundness, and the extent to which they can be addressed through changes in the structure of the proofs. Ultimately, some performance issues are fundamental to this approach, motivating the alternative approach we describe in Section 4.3.

Inefficient Proof Term Traversals

As mentioned, the extracted interpreter relies on the progress proof to provide one-step execution results and the preservation proof to provide an associated typing term. At each iteration step, the progress and preservation proofs may need to traverse large portions of the typing or reduction proof terms in order to produce their results. As a result, the performance of the extracted interpreter is significantly impacted by the large sizes of these proof terms.

This issue is particularly pronounced for industrial languages like Wasm, which have complex structural typing rules. For instance, the reduction and typing rules for **label**, a structural instruction that models blocks of code on the stack, illustrate this complexity. Figure 4.2 displays some of the reduction and typing rules for **label**, where the reduction rules are a specialised version of the block context reduction rules introduced in Figure 2.11.

$$\frac{(S;F;es) \hookrightarrow (S';F';es')}{(S;F;\mathbf{label}_n\{es_{\mathsf{cont}}\}\ es\ \mathbf{end}) \hookrightarrow (S';F';\mathbf{label}_n\{es_{\mathsf{cont}}\}\ es'\ \mathbf{end})}\ \textsf{label_reduction}$$

$$\frac{}{[\mathbf{label}_n\{es_{\mathsf{cont}}\}\ vs\ \mathbf{end}] \hookrightarrow vs}\ \textsf{label_exit}$$

$$\frac{C \vdash es_{\mathsf{cont}} : t_1^n \to ts_2 \quad C' = C[\mathsf{label} := [t_1^n]\ +\!\!+\ C.\mathsf{label}] \quad C' \vdash es : [] \to ts_2}{C \vdash \mathbf{label}_n\{es_{\mathsf{cont}}\}\ es\ \mathbf{end} : [] \to ts_2}\ \textsf{label_typing}$$

Figure 4.2: Selected reduction and typing rules for **label**

The label_reduction rule states that a label instruction can be reduced if its body can be reduced, while the label_exit rule states that if the body is already a list of values, the label can be reduced

to that value list. The typing rule specifies that for a **label** instruction to be well-typed, its body must be typeable with the same type — the slightly different context is used to type control flow instructions which we do not describe in detail here.

Recall from Section 3.5 that the progress proof proceeds by induction over the typing derivation. In the **label** case, we must apply the induction hypothesis that, since the body of the **label** is also well-typed, it must either take a step or be terminal. if *es* takes a step, then the original **label** takes a step according to the label_reduction rule; if *es* is terminal, then the **label** takes a step by exiting the label according to the label_exit rule [1].

To see how the above proof structure gives rise to inefficiency of the interpreter, consider the typing term of the following small program in Figure 4.3 involving multiple nested labels.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\vdash [\mathbf{i32.const}\ c_1;\mathbf{i32.const}\ c_2] : [] \to [\mathbf{i32};\mathbf{i32}]}\ \textsf{composition} \qquad \dfrac{}{\vdash [\mathbf{i32.add}] : [\mathbf{i32};\mathbf{i32}] \to [\mathbf{i32}]}\ \textsf{add}}{\vdash [\mathbf{i32.const}\ c_1;\mathbf{i32.const}\ c_2;\mathbf{i32.add}] : [] \to [\mathbf{i32}]}\ \textsf{composition}}{\vdash [\mathbf{label}_0\{\}\ [\mathbf{i32.const}\ c_1;\mathbf{i32.const}\ c_2;\mathbf{i32.add}]\ \mathbf{end}] : [] \to [\mathbf{i32}]}\ \textsf{label_typing}}{\vdash [\mathbf{label}_0\{\}\ [\mathbf{label}_0\{\}\ [\mathbf{i32.const}\ c_1;\mathbf{i32.const}\ c_2;\mathbf{i32.add}]\ \mathbf{end}]\ \mathbf{end}] : [] \to [\mathbf{i32}]}\ \textsf{label_typing}
$$

Figure 4.3: Typing term of a program with deeply nested **labels**

As part of "executing" the progress definition, the application of the induction hypothesis in the label typing case translates to a recursive invocation on the label's body. The progress definition must be recursively invoked over each nested label body to produce one step of execution, backtracking through all nested levels to construct the execution result and the corresponding reduction proof term. Similarly, the preservation function must traverse the reduction term composed of nested label_reduction rules to form the typing term of the result configuration. Each time the interpreter steps, this recursion and reconstruction process is repeated. This process is highly inefficient and does not scale well to larger Wasm programs.

Improving the performance of this case is challenging — it arises directly due to the inductive structure of the progress proof over **label**. In fact, a similar, albeit less severe inefficiency can be observed in a traditional one-step interpreter defined in a recursive `Fixpoint` in Coq. As we will discuss in Section 4.3.3, we instead prefer direct control over the interpreter's representation of the evaluation context, rather than relying on the structure inherent in the naturally-arranged proof of progress.

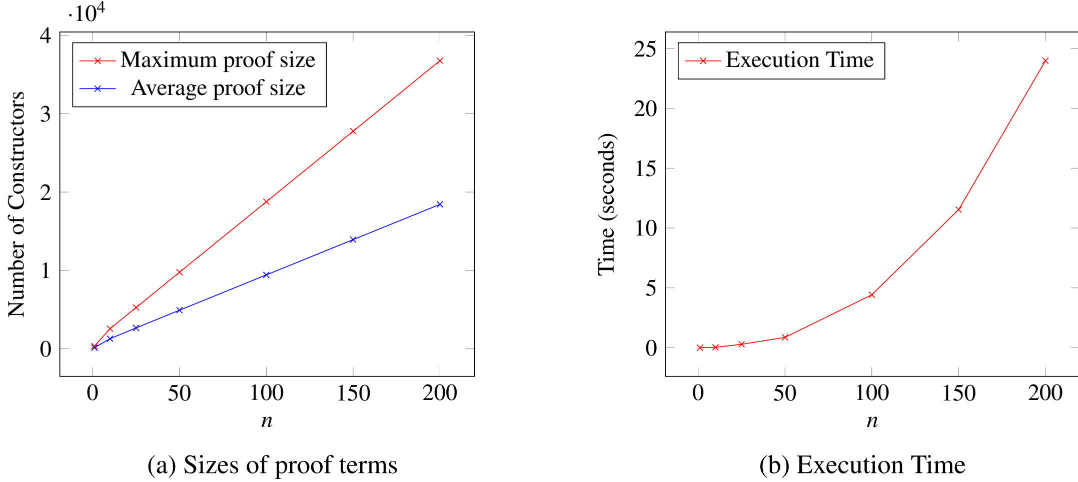
Proof Term Explosion

Another cause of inefficiency in the type soundness interpreter is the *explosion* in the size of proof terms, resulting from unexpected interactions between the proofs and some of Wasm's typing rules. Recall that our automatically-derived interpreter, even when extracted to OCaml, must explicitly represent the proof term of well-typedness of the configuration in memory as a full proof

[1] Special treatment is required when the body *es* contains a control flow instruction **br** at its hole, in which case the continuation is targeted. The corresponding case in the progress proof requires a decision process on decomposing the label body *es* at every step, which is also inefficient.

term, and traverse large portions of it at each step. This means that a larger proof term directly translates to lower performance of the interpreter. We encountered this issue while benchmarking the extracted interpreter and observed a super-linear complexity for a program that computes the nth Fibonacci number using a loop, which should theoretically have $O(n)$ time complexity. Importantly, this issue is orthogonal to the above issue with **label** contexts, as the iterative Fibonacci algorithm does not introduce deeply-nested labels.

Figure 4.4a shows the execution of an $O(n)$ iterative Fibonacci Wasm program that computes the nth Fibonacci number. By plotting the maximum and average sizes of the proof term (measured by the number of constructors in the proof term of the typing relation) during execution against different values of the input size n, we can observe how the proof term size grows linearly with the number of loop iterations. Since each execution step of the type-safety interpreter involves traversing a large part of the proof term, the complexity of execution per step is $O(n)$. On the other hand, $O(n)$ steps are required to compute the nth Fibonacci number. This results in an overall $O(n^2)$ complexity, which agrees with our observation in Figure 4.4b.



(a) Sizes of proof terms

(b) Execution Time

Figure 4.4: Benchmark: $O(n)$ iterative Fibonacci with input n

Our mechanised soundness proofs in Section 3.5, from which we automatically derive the interpreter, was not structured so as to minimise the size of the generated proof term. In particular, we identify that subtle choices in the use of the composition and subsumption typing rules can have massive impact on the size of the proof term.

To illustrate this problem, we recall the composition typing rule from Figure 2.18, which describes how a sequence of instructions is typed. We currently formulate Wasm's composition rule by allowing the splitting of one element at the end of the sequence each time as follows:

$$\frac{C \vdash es : ts_1 \to ts_3 \quad C \vdash [e] : ts_3 \to ts_2}{C \vdash es +\!\!+ \ [e] : ts_1 \to ts_2} \ \texttt{composition}$$

There are various other formulations that are equivalent to this choice from the perspective of the type system. For example, the composition rule could also be stated to split one element from the head of the list each time, or to simply allow general list concatenations. The formulation given above most closely follows the structure of industrial type checking algorithms for Wasm,

which operate in a single linear and incremental pass of the program. Note though our discussion in Section 3.10.3, where we report our discovery that the latest official Wasm specification includes an incorrect version of this rule.

This seemingly innocuous typing rule is a major contributor to the proof term explosion problem. Consider the following typing inversion lemma for the composition of instruction sequences introduced earlier in Figure 3.9:

Lemma 4.2.1.

$$\forall es_1, es_2, ts_1, ts_2. (\vdash es_1 \mathbin{++} es_2 : ts_1 \to ts_2) \implies (\exists ts_3. \vdash es_1 : ts_1 \to ts_3 \wedge \vdash es_2 : ts_3 \to ts_2).$$

The above lemma states that if a concatenation of two program fragments es_1 and es_2 has a function type $ts_1 \to ts_2$, then it must be the case where es_1 and es_2 are separately well-typed with some appropriate function types that match.

The proof is naturally conducted by induction on es_2 from the end. If es_2 is empty, the desired typing term for es_2 is obtained by the empty typing rule, and the desired typing term for es_1 is simply the original typing term given in the premise. Otherwise, let $es_2 = es' \mathbin{++} [e]$. The original premise can be rearranged to

$$\vdash (es_1 \mathbin{++} es') \mathbin{++} [e] : vs_1 \to vs_2$$

We apply an auxiliary lemma, which is a special case of original lemma to be proved when the second instruction list es_2 is a singleton list (proof omitted) to the above and obtain

$$\exists ts_3. \vdash es_1 \mathbin{++} es' : ts_1 \to ts_3 \wedge \vdash [e] : ts_3 \to ts_2$$

Now, by the induction hypothesis on es' we have

$$\exists ts_4. \vdash es_1 : ts_1 \to ts_4 \wedge \vdash es' : ts_4 \to ts_3$$

The first part of the conjunction provides the typing term for es_1, and the typing term for $es_2 = es' \mathbin{++} [e]$ can be applied by applying the composition typing rule with the typing terms for es' and $[e]$ we've obtained.

While the above proof is valid from the perspective of proving the lemma, and is in some sense the most "generic" formulation of the proof, the key observation is that it generates a typing tree with n composition rules for es_2 when $|es_2| = n$, instead of the $n-1$ required. This is because the induction's base case is $es_2 = []$, leading to a redundant application of the composition typing rule when the induction hypothesis is applied when $e' = []$ in the inductive case to construct a typing term for $\vdash [] \mathbin{++} [e]$ from that of $\vdash []$ and $\vdash [e]$.

While a redundancy of 1 can seem negligible, the above lemma is important in proving the preservation property, specifically in the case of the proof (done by induction on the reduction relation) where the top of the reduction term is a context reduction rule, and specifically the sequential

context which is described by the following reduction rule

$$\frac{(S;F;es) \hookrightarrow (S';F';es')}{(S;F;vs_c +\!\!+\ es +\!\!+\ es_c) \hookrightarrow (S';F';vs_c +\!\!+\ es' +\!\!+\ es_c)} \text{ context_sequence}$$

The preservation proof for this assumes a reduction term $(S;F;es) \hookrightarrow (S';F';es')$, along with a typing term for the previous configuration $(S;F;vs_c +\!\!+\ es +\!\!+\ es_c)$, and must construct a typing term for the result configuration $(S';F';vs_c +\!\!+\ es' +\!\!+\ es_c)$. The proof starts by first applying the typing inversion lemma (Lemma 4.2.1) twice to obtain the typing terms for vs_c, es, and es_c individually from the original typing term, introducing a redundancy of 1 to the size of the typing terms of es' and es_c. The proof then applies its inductive hypothesis to obtain a typing term for es' from the typing term of es and the reduction term; this means that the redundancy of 1 is inherited in this step for each application of the context_sequence rule in the reduction term. As a result, the preservation proof in fact introduces a redundancy equal to at least the number of times that context_sequence appears in the reduction term.

Exacerbating the problem, the reduction term produced by the progress proof contains the same number of context_sequence rule as the length of the program, as the natural induction over the typing term applies the context_sequence rule once per instruction (details omitted). Therefore, for larger programs, the typing term is quickly inundated with an insurmountable number of redundant composition rules for the interpreter to deal with.

Note that it is possible to modify the above proof to avoid the use of redundant type constructor. In particular, instead of proving the lemmas in the natural way of an induction with a base case (where the instruction list is empty) and an inductive case, we can additionally consider the case of a singleton instruction list as a base case as well and avoiding the application of the inductive hypothesis in that case. This avoids the redundant application of the composition rule and achieves an optimisation on the sizes of proof terms. However, from the perspective of completing the proof, such a method is completely unnecessary and awkward.

In fact, when we modify the inductive case of Lemma 4.2.1 to the above "optimised" version, the observed size of the proof terms in memory shrinks by two orders of magnitude. We report in Section 4.4 the performance of an "optimised" automatically-derived interpreter with as many of these lemma fixups as possible.

The sensitivity of the size of the proof term to these small changes, which are at odds with the most uniform approach to stating the relevant inductive proofs, creates an unfortunate tension between the performance of the interpreter and the natural structure of the proof. The root cause of all these performance issues is that the interpreter induced from progress requires the typing derivation to be taken as its input due to the natural way that a progress proof is constructed. This observation, combined with our observation about the inefficiency of **label** execution above, motivates a converse approach where the interpreter is directly defined, with proof following the structure of the interpreter instead of being taken as an input, which we will now detail.

4.3 Progress from Progressful Interpreter

In this section, we present an alternative approach that augments a one-step interpreter with dependently-typed certifications, enabling direct control over the interpreter's structure. We are motivated by the following high-level observation: in the approach of Section 4.2, the constructive progress proof gives rise to an interpreter and its soundness proof because of obvious structural similarities between their natural definitions. In fact, as we discuss in Section 4.3.1, in certain intrinsically-typed settings there is no distinction between the two. Our approach in this section seeks to invert this relationship between the progress proof and the interpreter — showing that, even in an extrinsically-typed setting, a dependently-typed interpreter can give rise to a proof of the classic progress property. That is, Section 4.2 focuses on getting a sound but inefficient interpreter "for free" from a proof of the progress property. Now, our *progressful* interpreter approach provides a proof of progress and more "for cheap" from the dependently-typed definition of an efficient interpreter.

By selecting the appropriate certifying proposition as part of the interpreter's type, this method allows us to combine the definition of the interpreter, the proof of the interpreter's soundness, and the proof of progress into a single definition — maintaining many of the proof maintenance benefits of the previous section's approach with two distinct advantages. First, we can arrange the interpreter so that these proof terms are erased upon extraction from Coq, and second, we can directly optimise the interpreter's internal state in order to improve its performance. As we discuss in Section 4.3.1, we make the decision to keep preservation as a separate lemma, for reasons related to the non-determinism of Wasm's inductive operational semantics.

Consider the standard version of the one-step interpreter, which takes an input configuration cfg and either returns a new configuration cfg' as a successful one-step execution result or produces an error (for example, in the case that the input configuration is ill-typed). Now consider extending the one-step interpreter to a dependently-typed function such that, in the successful case where a result configuration cfg' is produced, a proof term representing $\mathsf{cfg} \hookrightarrow \mathsf{cfg}'$ is also produced. Such an approach would mean that the implementation and successful typing of the function itself proves that the interpreter is sound. This approach of using dependently-typed functions to simultaneously build certification along with the definition is well-understood in previous work as shown by Chlipala [22].

Now, consider that we can additionally augment the error case with a proof term explaining why the interpreter failed to take a step given its input configuration cfg. For Wasm, we choose the error certification as follows:

$$\mathsf{terminal}(\mathsf{cfg}) \vee (\forall t. \vdash \mathsf{cfg} : t \implies \mathsf{False})$$

Our key observation is that this proof term (together with the already established proof term in the mutually-exclusive successful case) represents the *contrapositive* of the progress property - "if the interpreter fails to step soundly according to the operational semantics, the input must be ill-typed". This certification establishes that the interpreter is a sufficient witness for a constructive proof of the progress property, and therefore by extending our interpreter with these proof

terms, we have established the progress property of Wasm's original inductive semantics while certifying not just the soundness of the interpreter, but a stronger property that it will always successfully take a step if its input is well-typed. Throughout the rest of this thesis, we will describe an interpreter carrying such a certification of the progress property as *progressful*. In contrast to the more traditional approach of Watt et al. [114], which requires a separate interpreter definition, soundness proof of the interpreter, and proof of progress, in our setting all of these definitions and proofs are combined into a single dependently-typed interpreter definition. Since the proof term components of the interpreter are not required as input, they can be *fully erased* when extracting the interpreter to OCaml, in contrast to Kokke, Siek, and Wadler [57] which requires a runtime representation of the typing term. Moreover, comparing against Section 4.2, we demonstrate that we can directly optimise the internal representation of the interpreter while maintaining the benefits of having a unified definition. In this section we first present a fairly naïve interpreter design, before describing our optimisations in Section 4.3.3.

4.3.1 Connection to Intrinsically Typed Languages

Previous work by Bach Poulsen et al. [11] has described a deeply-related approach, centred around an intrinsically typed language definition combined with a dependently-typed *definitional* interpreter. This approach avoids the need for a separate type soundness proof, as the successful definition and typing of the interpreter itself in the host metatheory (e.g. Agda) embodies guarantees equivalent to progress and preservation. If the interpreter's (host) type signature guarantees that an intrinsically-typed input will result in a well-formed output (as opposed to some distinguished error value), this embodies progress. If the input and output configurations of the interpreter are specified in the interpreter's (host) type signature as having the same (language-level) intrinsic type, this embodies preservation. Similarly, since there is no separate inductive operational semantics, it is meaningless to ask whether the interpreter is sound or complete — it is simply the normative definition of the language's semantics. Their work argues that for a deterministic language there seems to be no inherent drawback in presenting a small-step semantics as a definitional interpreter rather than as an inductive relation.

We are able to capture much of the spirit of this intrinsically-typed setting in our work. In both settings, the computational structure of the interpreter itself is used to "share work" with proofs of a similar structure — the term manipulation performed by an interpreter in the intrinsic setting in order to successfully establish its host type certifying that its (intrinsically well-typed) input results in a non-erroneous output closely parallels the reasoning necessary for our progressful interpreter to establish the contrapositive certification in our error case. Moreover, since the structure of this computation and reasoning closely parallels the structure of both an interpreter soundness proof and a progress proof (as exploited in the other direction by Section 4.2), we can cheaply establish the necessary certification in the non-erroneous step case such that our interpreter's certification as a whole not only implies its own soundness with respect to Wasm's operational semantics, but also the progress property. As one distinction, all of our type-level reasoning can be erased during Coq extraction, which helps us make our interpreter as efficient as possible.

Had Wasm been deterministic, we could also incorporate preservation into this certification with

little effort, again paralleling Bach Poulsen et al. [11]. Non-deterministic language semantics present somewhat of a challenge to approaches based on a definitional interpreter, which must directly represent every outcome that is intended to be allowed, potentially requiring fiddly or computationally-inefficient manipulation of constructs such as choice monads. In particular Wasm, despite its design aiming for determinism wherever possible, is non-deterministic and non-confluent in several edge cases, which perhaps explains the official specification's decision to define its operational semantics in terms of an inductive relation. This motivates our decision to define a sound dependently-typed interpreter which embodies the progress property, while leaving preservation as a separate proof. In order to also incorporate preservation, we would require not only the definition of an interpreter with non-deterministic choice but *also* a proof that this choice mechanism is complete with respect to Wasm's inductive operational semantics. We note that this approach would be theoretically feasible, but feel that it conflicts with our goals of minimizing the maintenance burden of WasmCert-Coq's mechanisation.

4.3.2 Progressful Interpreter for WebAssembly 1.0

We now show how the above approach for a dependently typed progressful interpreter can be realised in the WasmCert-Coq mechanisation of Wasm. We describe our implementation of the progressful Wasm one-step interpreter, by starting with the original interpreter of WasmCert-Coq, and showing how its result type and body can be extended with a certification of progressfulness.

Recall from Section 2.2 that WebAssembly's operational semantics is defined as an inductive relation between Wasm's configuration tuples $(S;F;es)$. We also recall the ordinary one-step interpreter eval_1 from Section 3.4, which is a function taking a configuration tuple $(S;F;es)$ as its input argument, and returns one of the following results:

- $\mathbf{R}_{\mathsf{normal}}\ (S';F';es')$ – a normal step of computation returning a new configuration tuple;
- $\mathbf{R}_{\mathsf{value}}\ vs$ – a termination result that the instructions es to be executed in the input configuration is already a list of values vs;
- $\mathbf{R}_{\mathsf{trap}}$ – another termination result, indicating that the input program is already a trap;
- $\mathbf{R}_{\mathsf{error}}$ – an assertion failure that should only occur if the input configuration is ill-typed;
- $\mathbf{R}_{\mathsf{br}}\ k\ vs$ and $\mathbf{R}_{\mathsf{return}}\ vs$ – exceptional results used to represent WebAssembly's special structured control flow instructions **br** and **return** respectively.

The progressful interpreter takes a configuration $(S;F;es)$ as input and returns a dependently-typed result that includes the necessary progressful certifications for each case:

- $\mathbf{R}_{\mathsf{normal}}\ (S';F';es')\ (H_{\mathsf{reduce}} : (S;F;es) \hookrightarrow (S';F;'\,es'))$, a normal step of computation with a proof of the corresponding reduction;
- $\mathbf{R}_{\mathsf{value}}\ vs\ (H_{\mathsf{val}} : \mathsf{to_value}(es) = vs)$, a normal termination result with a proof that the input configuration is already a list of values vs;

- $\mathbf{R}_{\mathsf{trap}}$ $(H_{\mathsf{trap}} : es = \mathbf{Trap})$, a trap result with a trivial equality indicating that the input is indeed a trap;

- $\mathbf{R}_{\mathsf{error}}$ $(H_{\mathsf{error}} : \mathsf{fragment_illtyped}\ S\ F\ es)$, an error result with a proof that the input *program fragment* is ill-typed. Note that this is slightly different from the abstract overview of the error case for the progressful interpreter, where the error certification should reflect the ill-typedness of the input configuration:

$$\forall ts.(\vdash (S;F;es) : ts \rightarrow \mathsf{False})$$

However, recall from Section 3.4 that the input configuration could also represent a program *fragment* containing a **br** or **return** which are currently returning. In this case, the input configuration will also be ill-typed – but only because the input program fragment did not carry the full labels and function call frames within the runtime representation, which is separately dealt with by the $\mathbf{R}_{\mathsf{br}}$ and $\mathbf{R}_{\mathsf{return}}$ results. The error result should only capture program fragments which are genuinely ill-typed even when the potential missing labels and frames are recovered. As a result, the error certification we establish for the error case, fragment_illtyped, is stated as follows:

$$\forall ts, C, ts_{\mathsf{label}}, ts_{\mathsf{ret}}.(((\vdash_{\mathsf{s}} S : \mathsf{ok}) \wedge (S \vdash_{\mathsf{i}} F.\mathsf{inst} : C) \wedge \\ (S;C[\mathsf{label} := ts_{\mathsf{label}}; \mathsf{return} := ts_{\mathsf{ret}}] \vdash es : [] \rightarrow ts)) \implies \mathsf{False})$$

Note that the full configuration ill-typedness can be obtained from the above by applying the corresponding typing rules for config and thread introduced in Figure 2.21.

- $\mathbf{R}_{\mathsf{br}}\ k\ vs$/ $\mathbf{R}_{\mathsf{return}}\ vs$ $(H_{\mathsf{br}} : \mathsf{wf_\{br}\ k\ vs/\mathsf{return}\ vs\}\ S\ F\ es)$ are also augmented with well-formedness certifications corresponding to the correctness results for $\mathbf{R}_{\mathsf{br}}$ and $\mathbf{R}_{\mathsf{return}}$ given in Lemma 3.4.2 and Lemma 3.4.3. These results specify that the input can be decomposed as a block context containing a **br** or a **return**. The intuition for these certifications is similar to their corresponding lemmas, stating that a $\mathbf{R}_{\mathsf{br}}/\mathbf{R}_{\mathsf{return}}$ result can only be generated from a returning **br**/**return** instruction, with the additional constraint for the depth of the block in the case of **br**.

The full definitions are given as follows:

$$\mathsf{wf_br}\ k\ vs\ S\ F\ es ::= \exists i, B.\ es = B^i[vs +\!\!+\ [\mathbf{br}\ (k+i)]] \wedge \mathsf{empty_vs_base}(B)$$

$$\mathsf{wf_return}\ vs\ S\ F\ es ::= \exists i, B.\ es = B^i[vs +\!\!+\ [\mathbf{return}]] \wedge \mathsf{empty_vs_base}(B)$$

where the empty_vs_base predicate states that the preceding values in the innermost block must be empty, defined by the following recursion:

$$\begin{aligned} &\mathsf{empty_vs_base}(vs +\!\!+\ [_] +\!\!+\ es) && ::= (vs == []) \\ &\mathsf{empty_vs_base}(vs +\!\!+\ \mathbf{label}_n\ \{es_{\mathsf{cont}}\}\ B[_]\ \mathbf{end} +\!\!+\ es) && ::= \mathsf{empty_vs_base}(B) \end{aligned}$$

This additional constraint is required because the block decomposition $es = B^i[vs +\!\!+\ [_]]$ is

not unique if the innermost block could contain a non-empty list of preceding values. The empty_vs_base constraint ensures that all values preceding the hole in the innermost block are returned to the arguments of the interpreter results $\mathbf{R}_{\mathsf{br}}\ k\ vs/\mathbf{R}_{\mathsf{return}}\ vs$. This is necessary to reach a contradiction when the number of values returned are not enough to fulfill arity of the target label, as the constraint ensures that there's no more values to be found in the innermost block.

We present a pseudocode of our implementation in Figure 4.5. In the pseudocode, `split_vals` is a function that splits up the value stack vs and instruction stack from the input instruction list es. Execution then follows by looking up the top instruction e if there is one to execute and performs a case split and returns a terminal result otherwise. For each case of the instruction e, if the associated constraints are satisfied by the input, the interpreter returns a successful result $\mathbf{R}_{\mathsf{normal}}$ and constructs a successful certification H_{reduce} *in place*. Otherwise, the interpreter returns an error result $\mathbf{R}_{\mathsf{error}}$ with a certification H_{error} proving the ill-typedness of the input configuration.

Note on Line 1 of the pseudocode: only the minimal Wasm configuration is taken without any typing information, as opposed to the interpreter from type soundness introduced in Section 4.2. This is a key difference that allows the progressful interpreter to be much more efficient.

```
1:  Input : (S; F; es)
2:  match split_vals es with
3:  | (vs, []) ⟹ return R_value rev(vs) (H_value : ...)
4:  | (vs, e :: es_0) ⟹
5:      match e with
6:      | t.add :
7:         if vs = [t.const c_2; t.const c_1] ++ vs':
8:            return R_normal (S; F; rev(vs') ++ [t.const (c_1 + c_2)] ++ es_0) (H_reduce : ...)
9:         else
10:           return R_error (H_error : ...)
11:     | local.get j :
12:        if F.local[j] = Some v:
13:           return R_normal (S; F; rev(vs') ++ [v] ++ es_0) (H_reduce : ...)
14:        else
15:           return R_error (H_error : ...)
16:     ...
17:     end match
18: end match
```

Figure 4.5: Pseudocode of the Wasm 1.0 progressful one-step interpreter

We omit the concrete syntax of constructing the successful and error certifications in each case from the above figure. However, we will now explain the structure of our method in detail to demonstrate how ill-typedness proofs like the above can be constructed in a modular and scalable way.

In principle, each ill-typedness statement is an implication from a typing term to False, therefore the proof is performed naturally by inverting the structure of the typing term. However, Wasm's subsumption typing rule makes this proof slightly more difficult, as each code fragment es can be associated with different function types up to the subsumption rule. To design a scalable

infrastructure for these proofs, we utilise the set of *typing inversion* lemmas, which were originally established for proving the preservation property we introduced in Section 3.5.1. These typing inversion lemmas describe the corresponding set of constraints that various parts of the typing relation needs to satisfy for the typing relation to hold.

The proofs of ill-typedness proceed by proving a contradiction based on the information of the input configuration that goes into the error execution case.

As a specific example[2], we prove the error certification H_{error} for the case of **local.get** j.

Proposition 4.3.1. *If* $F.\mathsf{local}[j] = \texttt{None}$, *then*

$$\forall ts, (\vdash (S; F; vs +\!\!+ \ [\mathbf{local.get}\ j] +\!\!+ \ es') : [] \to ts) \implies \mathsf{False}).$$

Proof. We first expand the configuration typing relation to obtain the following typing relation for program fragment:

$$\ldots \land (S \vdash_{\mathsf{f}} F : C) \land (S; C \vdash (vs +\!\!+ \ [\mathbf{local.get}\ j] +\!\!+ \ es') : [] \to ts)$$

The frame validity relation (introduced in Section 2.5) would provide that

$$C.\mathsf{local}[j] = \texttt{None}.$$

Thus

$$\begin{aligned}
& S; C \vdash (vs +\!\!+ \ [\mathbf{local.get}\ j] +\!\!+ \ es') : [] \to ts \\
\implies & \exists ts_3.(S; C \vdash vs : [] \to ts_3) \land (S; C \vdash ([\mathbf{local.get}\ j] +\!\!+ \ es') : ts_3 \to ts) \\
\implies & (S; C \vdash ([\mathbf{local.get}\ j] +\!\!+ \ es') : (\mathsf{typeof}\ vs) \to ts) \\
\implies & \exists ts_3.(S; C \vdash [\mathbf{local.get}\ j] : (\mathsf{typeof}\ vs) \to ts_3) \\
\implies & \exists t.(C.\mathsf{local}[j] = \texttt{Some}\ t)
\end{aligned}$$

But the last line contradicts with the premise that $C.\mathsf{local}[j] = \texttt{None}$. □

We reiterate that, as displayed in the proof above, the error certifications are the only places where the typing predicate comes into relevance (in order to produce a contradiction), and the overall progressful interpreter does *not* require a typing derivation as an input. Since all certifications are erased during extraction, this means that the extracted progressful interpreter does not need to carry any typing derivation in its runtime, thereby avoiding the performance pitfalls discussed in Section 4.2.2.

The execution and certification for most of the other cases can be implemented similarly. The control-flow instructions **br** and **return**, and the related block-like instructions **label** and **frame**, require special treatment in the interpreter, where error certifications are propagated through exiting of the blocks, similar to the corresponding cases for the ordinary interpreter introduced in Section 3.4.

[2]For demonstration in the thesis, we only present the proof for the config ill-typedness. However, the full fragment ill-typedness is essentially done in the same way in the mechanisation.

With the above interpreter defined, we have constructed, in one go, a dependently-typed interpreter augmented with:

- A proof that its successful executions are sound;
- A proof that its erroneous executions arise from ill-typed inputs.

From these together, we can derive the progress property by contrapositive reasoning.

4.3.3 Optimising the Augmented Interpreter: Efficient Runtime Representation

One major advantage of our progressful interpreter in Section 4.3.2 over the interpreter derived from type soundness proofs is our ability to directly control the structure of the interpreter, making optimisations more feasible. In this section, we demonstrate an optimisation to the augmented Wasm interpreter by using a significantly more efficient runtime representation for evaluation contexts. This optimisation was first discussed in WasmRef-Isabelle by Watt et al. [115], though we employ a slightly different formulation in our work.

Recall the one-step interpreter in Figure 4.5. At every step of execution, the interpreter needs to decompose the input instruction list *es* into an evaluation context with a hole containing at most one instruction[3] $\mathcal{E}[e^?]$. This procedure requires traversing not only each nested **label** and **frame** context in the input instruction *es*, but also the linear instruction list (line 2, Figure 4.5) every time to locate the top instruction to be executed. As we discuss in Section 4.2, our automatically-derived interpreter suffers from a similar inefficiency.

The solution proposed by Watt et al. [115] addresses this inefficiency by switching to a more efficient representation of the evaluation context. Instead of naïvely defining the interpreter on Wasm's configuration tuple, this approach moves entered labels into a side data structure, avoiding re-recursing into them at each execution step.

The optimised representation defines three kinds of nested single-hole contexts, $\mathcal{E}_{\mathsf{stack}}$, $\mathcal{E}_{\mathsf{label}}$, and $\mathcal{E}_{\mathsf{frame}}$, as shown in Figure 4.6. Each of these contexts can be mapped to a regular Wasm term using the family of $\mathcal{E}[\![es]\!]$ functions described in Figure 4.6, which fill the context hole with the argument *es*.

The optimised one-step interpreter now takes a runtime configuration $(S; \mathcal{E}^*_{\mathsf{frame}}; \mathcal{E}_{\mathsf{stack}}; e^?)$ as its input and returns a tuple of the same shape as the output when successful. This representation efficiently records all the frame and label contexts in a separate stack in the runtime tuple. As a result, this method avoids traversing through all the evaluation contexts on the instruction stack at every step of execution. Instead, after every step of execution, the one-step interpreter can simply retrieve the next instruction to be executed from the context.

In WasmCert-Coq, a separate transformation function is implemented to transform an original Wasm configuration to its optimised interpreter runtime representation. This can be done by repeatedly pushing the **label** and **frame** contexts to the context stacks (similarly to how the original

[3]The top instruction may not exist when the current configuration represents, for example, a **label** or **frame** with an empty body. In such cases, the innermost context is exited via the label_exit or frame_exit rule at the next execution step.

(interpreter runtime configuration) $(S; \mathcal{E}^*_{\mathsf{frame}}; \mathcal{E}_{\mathsf{stack}}; e^?)$
(stack context) $\mathcal{E}_{\mathsf{stack}} := \textbf{stack_ctx}\ vs_{\mathsf{stack}}\ es_{\mathsf{stack}}$
(label context) $\mathcal{E}_{\mathsf{label}} := \textbf{label_ctx}\ n\ es_{\mathsf{cont}}\ \mathcal{E}_{\mathsf{stack}}$
(frame context) $\mathcal{E}_{\mathsf{frame}} := \textbf{frame_ctx}\ n\ F\ \mathcal{E}^*_{\mathsf{label}}\ \mathcal{E}_{\mathsf{stack}}$

$$(\textbf{stack_ctx}\ vs_{\mathsf{stack}}\ es_{\mathsf{stack}})[\![es]\!] = \mathsf{rev}(vs_{\mathsf{stack}}) +\!\!+\ es +\!\!+\ es_{\mathsf{stack}}$$
$$(\textbf{label_ctx}\ n\ es_{\mathsf{cont}}\ \mathcal{E}_{\mathsf{stack}})[\![es]\!] = \mathcal{E}_{\mathsf{stack}}[\![\textbf{label}_n\{es_{\mathsf{cont}}\}\ es\ \textbf{end}]\!]$$
$$(\mathcal{E}_{\mathsf{label}} :: \mathcal{E}^*_{\mathsf{label}})[\![es]\!] = \mathcal{E}^*_{\mathsf{label}}[\![\mathcal{E}_{\mathsf{label}}[\![es]\!]]\!]$$
$$(\textbf{frame_ctx}\ n\ F\ \mathcal{E}^*_{\mathsf{label}}\ \mathcal{E}_{\mathsf{stack}})[\![es]\!] = \mathcal{E}_{\mathsf{stack}}[\![\textbf{frame}_n\ F\ (\mathcal{E}^*_{\mathsf{label}}[\![es]\!])\ \textbf{end}]\!]$$
$$(\mathcal{E}_{\mathsf{frame}} :: \mathcal{E}^*_{\mathsf{frame}})[\![es]\!] = \mathcal{E}^*_{\mathsf{frame}}[\![\mathcal{E}_{\mathsf{frame}}[\![es]\!]]\!]$$
$$(S; \mathcal{E}^*_{\mathsf{frame}} +\!\!+\ [\textbf{frame_ctx}\ n\ F\ \mathcal{E}^*_{\mathsf{label}}\ \mathcal{E}_{\mathsf{stack}}])[\![es]\!] = (S; F; \mathcal{E}^*_{\mathsf{label}}[\![\mathcal{E}^*_{\mathsf{frame}}[\![es]\!]]\!])$$

Figure 4.6: Optimised runtime representation for Wasm interpreter and context-instruction composition

WasmCert interpreter traverses the current program to locate the instruction to be executed). This transformation can then be used to transform the Wasm configurations to be executed after instantiation to the optimised representation. An alternative method is to directly generate an optimised runtime representation as the result of module instantiation and invocation.

Aside from saving the context traversal at each step of interpreter execution, the optimised runtime representation also *simplifies* the structure of the interpreter. In particular, the auxiliary results $\mathbf{R}_{\mathsf{br/return}}$ are no longer required, since all the **label** and **frame** contexts are now directly contained within the runtime representation – as a result, the execution of **br** and **return** instructions can be performed by directly exiting from the appropriate contexts. In addition, the error certification no longer needs to work with *fragment* ill-typedness, but can instead be formulated directly in terms of ill-typedness of the whole Wasm configuration.

Concretely, the augmented, progress-deriving interpreter with the above optimisation now returns the following types of results:

- $\mathbf{R}_{\mathsf{normal}}\ (S'; \mathcal{E}'^*_{\mathsf{frame}}; \mathcal{E}'_{\mathsf{stack}}; e'^?)$, with certification

$$H_{\mathsf{reduce}} : (S; \mathcal{E}^*_{\mathsf{frame}})[\![\mathcal{E}_{\mathsf{stack}}[\![e^?]\!]]\!] \hookrightarrow (S'; \mathcal{E}'^*_{\mathsf{frame}})[\![\mathcal{E}'_{\mathsf{stack}}[\![e'^?]\!]]\!]$$

  Along with a proof that the context filling defined in Fig 4.6 and the decomposition from the Wasm configuration to the interpreter runtime configuration are inverse to each other, this implies the soundness of the interpreter with respect to the operational semantics.[4]

- $\mathbf{R}_{\mathsf{value}}\ vs\ (H_{\mathsf{val}} : \ldots)$, a terminating result indicating that the original input configuration is already a value. We omit the details of the certifying proposition as it also needs to account for the original decomposition procedure;

- $\mathbf{R}_{\mathsf{error}}\ (H_{\mathsf{error}} : \forall ts, (\vdash (S; \mathcal{E}^*_{\mathsf{frame}})[\![\mathcal{E}_{\mathsf{stack}}[\![e^?]\!]]\!] : ts) \implies \mathsf{False})$, an error result with a certification that the corresponding Wasm configuration of the input representation is ill-typed.

[4]Note that $(S; \mathcal{E}^*)[\![-]\!]$ is only well-defined when $\mathcal{E}^*$ is non-empty, as it needs to contain an outermost special frame that provides the overall frame in the restored Wasm configuration. This is always the case for any decomposition of a Wasm config tuple (whose detail we omitted here) as the original Wasm configuration always contains a frame. Our interpreter additionally proves (with trivial effort) that any successful result it returns preserves this as an invariant.

Due to the simplified process of dealing with control flow instructions as well as no longer needing to deal with fragment ill-typedness, the optimised progressful interpreter is not only more efficient in terms of execution time, but can also be implemented in a *smaller* code size. We will compare these characteristics of our different versions of interpreters in Section 4.4.

4.4 Evaluation

In this section, we compare several key metrics across the different interpreters we have discussed and implemented in this chapter of the thesis, including their runtime performances and the engineering efforts. As part of the discussion of the proof engineering, we discuss our experience of implementing our dependently-typed interpreter in Coq's *proof mode*, instead of directly constructing it functionally using the *convoy pattern* [22].

4.4.1 Runtime performance

In addition to the various versions of interpreters we implemented in this work, we have also fetched some external interpreters to provide more baselines for comparison. The interpreters we tested are listed in Figure 4.7. For external existing interpreters, we include a source of the interpreter in the figure. For interpreters implemented in this work, we note the sections where they are first discussed in the chapter. All interpreters from Coq mechanisations are extracted to OCaml as the target language. All benchmarks were run on a MacBook Pro (2019) with 2.3GHz Intel Core i9 Processor and 16GB RAM.[5]

Interpreter	Description	Source
Type Soundness Interpreter (Original)	Interpreter from WasmCert-Coq [114] type soundness proofs	Section 4.2
Type Soundness Interpreter (Optimised)	Type Soundness Interpreter with optimised proof trees	Section 4.2
Progressful Interpreter (Optimised)	Above optimised by methods from WasmRef-Isabelle [115]	Section 4.3.3
Reference Interpreter	Official Reference Interpreter from the Wasm Standard	[39]
WasmRef-Isabelle (Monadic State)	Monadic Interpreter from an Isabelle mechanisation of Wasm	[115]
Wasmtime	An industrial interpreter from Bytecode Alliance	[7]

Figure 4.7: List of interpreters compared

The first benchmark is a simple $O(n)$ iterative Fibonacci function, computing $\mathsf{Fib}(n)$ using a loop. The result is displayed in Figure 4.8.

We observe that the interpreter directly extracted from the WasmCert-Coq type soundness proofs quickly falls behind due to its super-linear performance[6]. In fact, running the interpreter on an input of $n = 10^3$ resulted in a stack overflow after approximately 20 minutes. An "optimised" version of the interpreter, which changes the structure of the type soundness proofs to minimize the size of the proof tree in memory as discussed in Section 4.2.2, achieves a linear runtime, although it takes $\sim$70 times longer than the OCaml reference interpreter. The optimised progressful interpreter has a runtime similar to the reference interpreter from the Wasm standard, while Watt's optimised interpreter in WasmRef-Isabelle is approximately twice as fast. Finally, the industrial

[5]The benchmark results were obtained by averaging from 3 executions and rounded to the nearest 0.01 seconds (with a floor of 0.01 seconds due to the logarithmic plot).

[6]Figure 4.8 is plotted on a logarithmic scale, so the super-linear performance is reflected by the gradient being larger than 1.

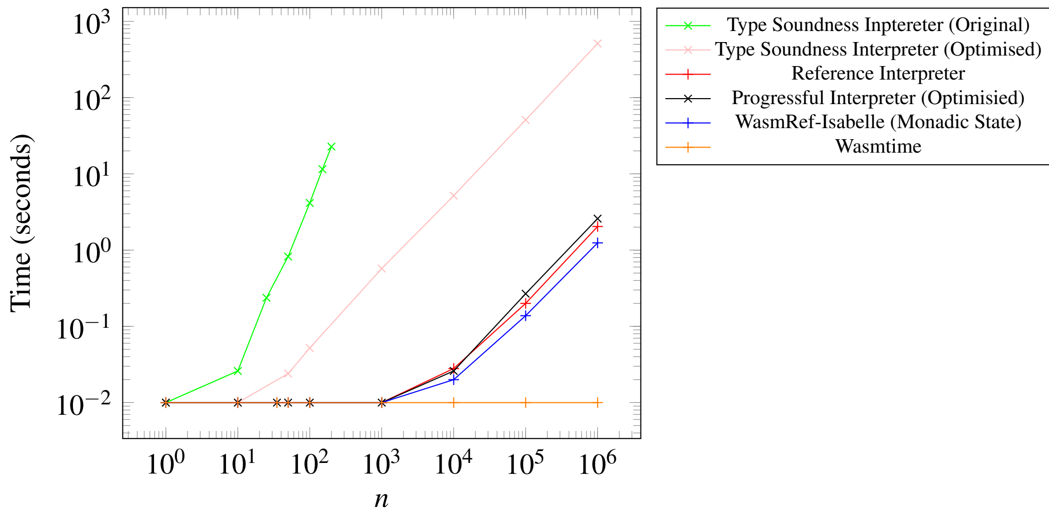


Figure 4.8: Benchmark: Iterative Fibonacci with input n, log-scale

heavily-optimised interpreter Wasmtime from Bytecode Alliance ran the largest test ($n = 10^6$) within 0.01 seconds.

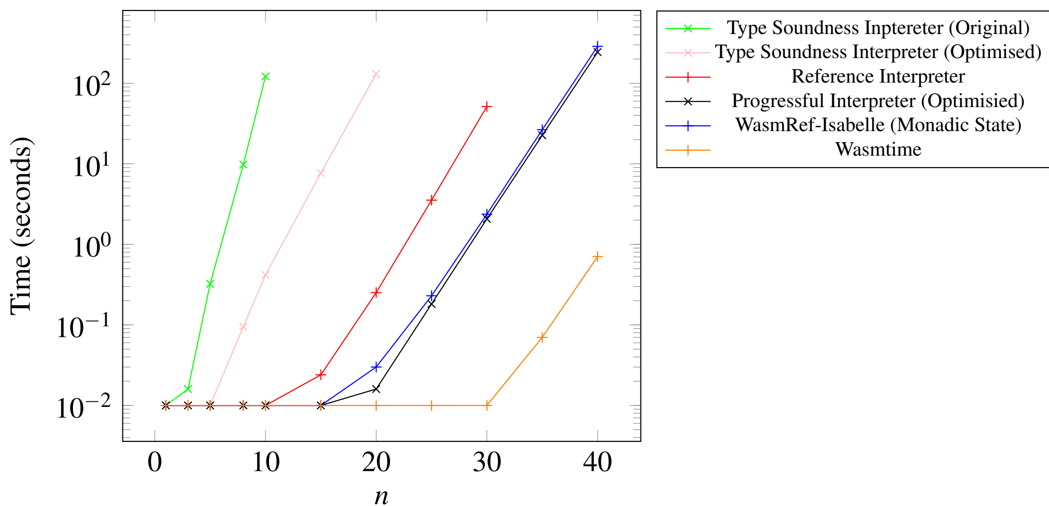


Figure 4.9: Benchmark: Recursive Fibonacci with input n, log-scaled time axis

The other benchmark we performed involved a recursive Fibonacci program that calculates the nth Fibonacci number using a recursive function, which has an exponential time complexity. This benchmark was similarly used in Watt et al. [115] to demonstrate the impact of the optimised runtime representation for evaluation contexts. Without this optimisation, the runtime configuration quickly grows in size and contains increasingly deep nested function call frames and labels. Naïve interpreters that use the native Wasm configuration tuples as runtime representation, such as the reference interpreter, will suffer from performance issues due to the need to traverse all the nested evaluation contexts at every step of execution, as explained in Section 4.3.3.

Figure 4.9 displays the benchmark result. All the interpreters without evaluation context optimizations (i.e. the type soundness interpreters and reference interpreter) can only be tested on inputs up to $n = 30$ or even less, either due to excessive runtime or crashing from stack overflow. Our

optimised progressful interpreter is slightly better than Watt's optimised WasmRef-Isabelle interpreter on all inputs, being approximately 20% faster. The industrial interpreter Wasmtime executes roughly 300 times faster than both of the verified and optimised interpreters.

We highlight that WasmCert-Coq's progressful interpreter is on par with the performance of Watt's WasmRef-Isabelle on the selected programs which the context optimisation targets. Together with the efficient persistent array extraction discussed in Section 3.11, WasmCert-Coq's interpreter is able to achieve reasonable performance without resorting to the approach required by WasmRef-Isabelle to create an entirely separate interpreter from scratch, which is therefore much more maintainable.

4.4.2 Proof engineering effort

Lines of code comparison We present a comparison of the lines of code used across different implementations of the interpreters in Figure 4.10, including a detailed breakdown on different components.[7] Auxiliary proofs and definitions, such as the typing inversion lemmas, automation tactics, and proofs to the preservation properties are omitted from the comparison table as they are required for all the approaches.

Components	WasmCert-Coq	Type Soundness Interpreter	Progressful Interpreter	Progressful Interpreter (Optimised)	Progressful Interpreter (Optimised, Wasm 2.0)
Interpreter	456	~300[8]	2396	1940	2463
Interpreter Soundness	1084	Embedded	Embedded	Embedded	Embedded
Interpreter Progressfulness	Not Proved	Embedded	Embedded	Embedded	Embedded
Progress	1039	1039	42 (trivial)	37 (trivial)	37 (trivial)
Total	2579	~1339	2438	1977	2500

Figure 4.10: List of interpreters compared

We note that the interpreter from type soundness comes with the shortest code: this is because the original type soundness proof of WasmCert-Coq is already almost constructive, so we only need to transform the Coq code from the `Prop` sort to `Type` to extract the proofs from Coq to OCaml. A wrapper function is then implemented to chain the progress, preservation, and type inference functions together accordingly.

We also observe that implementing the evaluation context optimisation from Watt et al. [115] does not result in a larger interpreter definition. A contributing factor may be that the optimised interpreter no longer needs to deal with the control flow instructions **br** and **return** using special auxiliary return results. Not all optimisations can result in simplifications as the above, but this nevertheless demonstrates the feasibility of performing structural optimisations to the progressful interpreter conveniently.

[7] The lines of code metric (LOC) is an unreliable basis for comparing different implementations because it is influenced by various factors, such as individual engineering practices and coding styles. In this case, the versions being compared were developed by different authors, so the LOC values shown in the table should be viewed as approximate indicators rather than precise measurements. They are intended only to offer a rough comparison of the various methods.

[8] This includes the estimated lines of code in the original proofs that had to be modified.

We report on our experience updating the WasmCert-Coq mechanisation to the Wasm 2.0 feature set in Section 3.10.1. For now, we note that the update to Wasm 2.0 results in a $\sim 30\%$ larger code size for the optimised interpreter, while the underlying number of instructions almost doubled.

Implementing Dependently-Typed Functions in Coq's Proof Mode As part of our experience of implementing the various interpreters in this chapter, we find Coq's *proof mode* especially convenient for implementing dependently-typed functions in Coq compared to using the traditional functional syntax and using convoy patterns for constructing the proof terms.

For a concrete example, we display an outline of the original WasmCert interpreter implementation for **local.get** j in Figure 4.11a. Transforming it to a progessful interpreter with pseudocode in Figure 4.5 requires adding a successful certification H_{reduce} to the successful R_{normal} result and an error certification H_{error} to the R_{error} result. We focus on H_{reduce}, which can be constructed as follows:

$$\dfrac{\dfrac{F.\mathsf{local}[j] = v_j}{(S;F;[\mathbf{local.get}\ j]) \hookrightarrow (S;F;[v_j])}\ \mathsf{local.get}}{(S;F;vs +\!\!+\ [\mathbf{local.get}\ j] +\!\!+\ es') \hookrightarrow (S;F;vs +\!\!+\ [v_j] +\!\!+\ es')}\ \mathsf{context}$$

The required proof term H_{reduce} can therefore be constructed by applying the two constructors for the context and local.get constructors, provided the knowledge that $F.\mathsf{local}[j] = v_j$. However, while this equality is valid within the specific case of the `match` statement, there is no way to extract this information directly from the original `match` syntax. Instead, we need to use a trick called the *convoy pattern* [22], which works by expanding the return type of the match to a function from the match equality to the original desired result. In this way, the match equality would then be available as an argument in constructing the match result. In the end, the function returned by the match is applied to a trivial reflexive equality of the match argument, thereby constructing the desired result. The shape of the resulting definition of the function is displayed in Figure 4.11b.

The above method is valid in itself. However, despite several shortcuts that further simplify the syntax slightly, we still find it tedious to completely adopt this pattern for every match case in our implementation. The breaking point that led us to abandon the functional syntax was that a direct definition of the interpreter does not satisfy Coq's strict syntactic termination check for `Fixpoint`, due to Wasm's structured control flow instructions. Therefore, the interpreter needs to incorporate a decreasing measure calculated from the structural complexity of the input configuration. This proved to be overly complicated to implement with the convoy pattern in the end.

As a result, we opted for an unusual method of defining the entire interpreter in the proof mode of Coq, essentially treating the definition of a function as a proof obligation depending on the input. This approach avoids the convoy pattern altogether and allows Coq's `Ltac` tactics to be directly used in the interpreter construction. Moreover, as displayed in Figure 4.11c, the certification for each case is constructed as a separate proof obligation after specifying the computation result. This is because constructors of inductive types in Coq can be equivalently used as lemmas that can be applied with the corresponding arguments. A partial application of a constructor means

```
Fixpoint run_one_step S F es :
    run_result S F es :=
...
match e with
| ... => ...
| local_get j =>
    match nth_error F.local j with


    | Some v_at_j =>
        R_normal
            (S; F; rev vs ++ [v_at_j] ++ es')

    | None =>
        R_error
    end
| ... => ...
end
...
```

(a) Original interpreter of WasmCert-Coq

```
Fixpoint run_one_step S F es :
    run_result S F es :=
...
match e with
| ... => ...
| local_get j =>
    match nth_error F.local j as nth_res
    return (nth_error F.local j = nth_res
            -> run_result S F es) with
    | Some v_at_j => (fun Hnth =>
        (R_normal ...
            (S; F; rev vs ++ [v_at_j] ++ es')
            (r_local_get ... Hnth)))
    | None => (fun Hnth =>
        (R_error ... (?HError)))
    end (eq_refl (nth_error F.local j))
| ... => ...
end
...
```

(b) Progressful interpreter using convoy patterns

```
Definition run_one_step S F es : run_result S F es.
Proof.
    ...
    destruct e as
    [ (* Other cases *) ... |
      (* local.get *) j |
      (* Other cases *) ... ].
    ...
    (* local.get *)
    {
        destruct (nth_error F.local j) as [v_at_j |] eqn:Hnth.
        - (* Success *)
            apply (R_normal ... (S; F; vs ++ [v_at_j] ++ es')).
          (* Proof obligation: prove the corresponding Wasm reduction *)
            ...
        - (* Error *)
            apply (R_error ...).
          (* Proof obligation: prove the ill-typedness of the input *)
            ...
    }
Defined.
```

(c) Progressful interpreter in Coq's proof mode

Figure 4.11: Different attempts of implementing interpreter execution for **local**.**get**

the remaining arguments (in our case, the certification) become proof obligations, which are then directly proven in the proof mode instead of being constructed functionally from the constructors.

To facilitate recursive calls, the main structure of the interpreter becomes an induction on the size of the input configuration, and recursive calls of the interpreter become applications of the inductive hypothesis, with the necessary premises conveniently established in the proof mode.

Overall, we found that this method of engineering eased the tediousness of implementing large dependently-typed functions as complex as our progressful interpreter in Coq.

4.5 Related Work

As I discuss in Section 4.3.1, Bach Poulsen et al. [11] present an alternative style for defining a deterministic programming language's semantics, using an intrinsically-typed representation combined with a dependently-typed definitional interpreter. The deep links between dependent types, definitional interpreters, and intrinsically-typed language definitions have also been widely discussed and explored by related work [95, 25, 87, 8, 94]. I believe our work is novel in mapping out the extent to which accepted benefits of the above setting can be transferred to "classical" inductively-defined semantics, without requiring all three of the above approaches to be simultaneously adopted. In particular, Wasm is non-deterministic, its configurations are not intrinsically typed, and our interpreter cannot be definitional. All of these constraints flow directly from the formalisation of Wasm's official standard, which I aim to stay close to wherever possible — any effort to (for example) define an intrinsically-typed Wasm or non-deterministic "definitional" interpreter would *increase* the maintenance burden of WasmCert-Coq, as a correspondence would need to be proved between this new definition and the faithful base mechanisation of the specification. I feel that our approach strikes the right balance in capturing many of the benefits of the intrinsically typed approach, without getting bogged down in these additional complications.

Chapman et al. [21] (drawing from Kokke, Siek, and Wadler [57]) is an interesting mid-point between our setting and the intrinsically typed + definitional interpreter setting, as it works with an intrinsically typed language but an inductive relational definition of the language's operational semantics. Such an approach causes the definition of the operational semantics to inherently embody *preservation* in its (host) type signature, but not *progress*, which is constructively proven separately. From this progress property, a sound interpreter can be automatically derived, as I discuss in Section 4.2. The authors of the above work are not able to execute their interpreter end-to-end as, in contrast to our work, they lack a verified type checker to initiate the interpreter loop. Therefore they do not investigate the relationship between the structure of their soundness proof and the runtime performance of the derived interpreter. Related to the need of a verified type checker, Adjedj et al. [1] describes a mechanised metatheory of the Martin-Löf Type Theory in Coq that could produce a certified and executable type checker from a decidability proof of type checking for their theory.

Youn et al. [119] describes a Domain-Specific Language (DSL) for Wasm specification, SpecTec, which aims to automatically generate Wasm specification artefacts and mechanised semantics, thereby providing a maintainable way to produce mechanised definitions for Wasm. This effort is neatly complementary to our work, as our approach tackles a different pain point in the mechanisation process of maintaining the proofs *on top of an already-produced model*. There is a minor overlap in the *meta-level interpreter* reported by SpecTec, as it could be seen as eliminating the need for any verified interpreter at all. However, SpecTec's approach is constrained in the optimisations that can be performed, since its intermediate representations must be derived automatically from the *original* Wasm definitions, while I can implement more ambitious optimisations while maintaining full trustworthiness through interactive proofs, as I reported earlier. Overall, our method would lessen the burden of maintaining relevant proofs when a SpecTec-derived mechanisation is extended.

Our challenge of proof maintenance is related to the *expression problem* (EP) [4], which describes the challenge of extending existing inductive definitions with new constructors while reusing the old proofs. The approach of *modular semantics* [76, 70] has been suggested to address this problem. With this approach, a language's syntax and semantic rules are given incrementally from reusable blocks, thereby avoiding the need of reformulation when further constructs are added to the language, and potentially allowing highly modularised proofs. This may be helpful in mechanising Wasm, as the feature proposals of Wasm can often be viewed as optional extensions of the semantics. Previous works have studied concrete implementations of extensible semantics in proof assistants such as Coq and Agda, where Delaware, S. Oliveira, and Schrijvers [26], Keuchel and Schrijvers [55], and Schwaab and Siek [101] followed the method of Swierstra [105] with workarounds for the respective proof assistants, and Jin, Amin, and Zhang [51] opted for a slightly different approach by directly extending the linguistic facilities of Coq. However, industrial language specifications such as Wasm are not always defined in a way that is amenable to modularisation. In particular, new features are simply added as inline patches to the specification text, and some features involve cross-cutting changes to Wasm's abstract representation and control flow. WasmCert-Coq places a high priority on its faithfulness to the original Wasm specification, and so careful consideration would be needed regarding the extent to which attempts at modularising the mechanisation would compromise this correspondence, and the extent to which such a modularisation would be robust against more cross-cutting changes.

4.6 Acknowledgements

This chapter draws text from the previously accepted paper *Progressful Interpreters for Efficient WebAssembly Mechanisation* (POPL 2025) [92], authored by myself, Stefan Radziuk, Conrad Watt, and Philippa Gardner. The idea of deriving an interpreter from a constructive type soundness proof described in Section 4.2 might have been around for a while, but first came to us via a series of popularisation by Philip Wadler in 2022.

Regarding mechanisation development, my co-author Stefan Radziuk implemented a large part of the initial unoptimised progressful interpreter and the adaptation of WasmCert's type soundness proof into the type soundness interpreter for Wasm 1.0. This work was implemented as his Master's project under my technical supervision. I then continued to implement the rest of the work described in this chapter, including the optimised runtime configuration from [115], and the update of the interpreter as part of the larger Wasm 2.0 update.

Chapter 5

Higher-Order Program Logic for WebAssembly

In this chapter, we discuss one particular application of our WasmCert-Coq mechanisation, the Iris-Wasm program logic for WebAssembly 1.0. This is a joint work with Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner and Lars Birkedal [91].

5.1 Introduction

Iris-Wasm is a mechanised higher-order separation logic for Wasm 1.0, which builds on the WasmCert-Coq mechanized specification of the Wasm 1.0 language standard. It is an interactive formal verification framework that exactly reflects the Wasm semantics. The result is a semantic and compositional characterisation of all Wasm definitions, which can be used to prove separation logic assertions about real Wasm programs, and which lays the foundation for rigorous investigations of the Wasm ecosystem.

Our implementation of the Wasm run-time semantics, with its difficult constructs such as complex control-flow commands, is given directly in Iris by using the mechanised semantics of WasmCert-Coq introduced in Chapter 3, instead of being translated into an existing intermediate Iris language. This choice requires considerable Iris engineering, but provides more trust in our mechanization, as it is line-by-line close to the Wasm semantics, and should lead to the mechanisation being comparatively straightforward to extend as the standard expands.

A major difficulty in designing the program logic is to handle the interaction between Wasm and the host. Recall from Section 2.7 that a Wasm program is expressed as a collection of higher-order ML-like *modules* composed together through a system of explicit imports and exports, and this process of composing Wasm modules into a full program is not performed within Wasm itself but handled by the host. In addition, the host is also responsible for providing several important capabilities not available to core Wasm code, including the module *instantiation* operation, which is a complex and inherently higher-order operation where the declared states of a WebAssembly module are allocated, the module's requested imports are satisfied, and the module's declared

exports are registered for use in satisfying further imports requested during subsequent instantiations. The Wasm standard defines instantiation in a host-agnostic way, to be then satisfied by the specific implementation of the embedders, such as the WebAssembly JavaScript Interface [28]. To achieve a similar level of independence from the host as the definition of the specification, we defined our program logic on top of a parametric host semantics. In particular, we provide a host-agnostic axiomatic characterization of Wasm module instantiation by establishing a lemma which lifts the complex W3C Wasm 1.0 instantiation predicate to our Iris-Wasm logic, describing the state before and after instantiation using our logical assertions. This lemma could therefore serve as the counterpart of the host-agnostic instantiation definition of the Wasm specification discussed in Section 2.7. We illustrate this instantiation lemma on a simple host language designed to capture the core functionality of the WebAssembly JavaScript Interface [28], and corresponding host program logic, where the soundness of our host instantiation proof rule is established using our instantiation lemma.

By capturing the semantics of the full Wasm 1.0 industrial standard directly, Iris-Wasm lays the groundwork for a wide range of future analyses. Iris-Wasm can be used to validate proposed extensions to Wasm such as MSWasm, a memory safe extension of Wasm [74]. It can be used to rigorously investigate compilers that either target Wasm or compile Wasm down to some low-level assembly language. Jacobs, Devriese, and Timany [50] demonstrate that Iris can be a useful tool to prove results such as full abstraction. Iris-Wasm sets the groundwork for similar results for realistic compilers involving Wasm.

We demonstrate our compositional higher-order reasoning about Wasm modules in our host language by developing a series of examples. Our main running example is a higher-order stack example comprising a stack module and a client module. The stack module defines and exports stack functions, including a higher-order map function for the stack. The client module imports and uses some of them, including map, in its main function. Using our Wasm program logic and a program logic for the simple host we introduce, we provide specifications for both modules: the stack module's specification contains specifications for all the stack functions, and the client module's specification depends on the stack module's specification. Finally, we verify a host program which instantiates the two modules in sequence, by modularly combining the proofs for the two module specifications. In addition, we demonstrate how to reason about *reentrancy* between the host and Wasm, by having the client module invoke a host function to modify the function table to provide a different input function for subsequent applications of map. The higher-order reasoning of the Iris framework provides an ideal environment to reason about Wasm modules. Nevertheless, it's a substantial task to apply Iris to a true industrial standard. Our implementation precisely follows the design decisions of the W3C Wasm 1.0 standard, and by using a rich logic such as Iris, we have laid the foundations for deep semantic investigations of WebAssembly and its future iterations.

In a case study, we investigate the intuitive coarse-grained encapsulation property of Wasm modules, stated in the standard: 'code from a module can arbitrarily affect its own state, but can only access the state of another module through the module's exports'. Several systems rely on this important property of Wasm to provide a form of sandboxing: for example, Fastly's 'Com-

pute@Edge' [45] platform and the RLBox tool [80]. Both depend on the encapsulation property of a module, regardless of behaviour of other modules, which are validated but not necessarily trusted. Reasoning about such modules necessarily involves the interaction between the known, verified code of one module against unknown, untrusted, and unverified code from other modules, something that cannot be done with a program logic. Building on top of Iris-Wasm, we define a relational interpretation of WebAssembly types through a unary logical relation, which is then used to verify specific *robust safety* properties of a known module, that hold even when composed with unknown modules. We demonstrate this by proving robust safety properties of our stack module composed with arbitrary clients. Our relational interpretation is entirely host agnostic, and can modularly be applied to any host language.

5.2 Instantiating the WasmCert Semantics in Iris

In this section, we introduce Iris-Wasm. We present several key concepts involved, and outline the process of instantiating the WasmCert-Coq semantics to the Iris. We present our proof rules for WebAssembly language features, and walk through how they are used to prove a specification for the fib function introduced in Chapter 3. For reasons of space, we only discuss selected proof rules; we stress that we have proved program logic rules for *all* of WebAssembly and used them to give full formal proofs of examples, including the stack module. A presentation of the Iris-Wasm proof rules is included in Appendix C.

5.2.1 Key Concepts

Instantiating the Wasm language in Iris It should be emphasised that Iris-Wasm has *proved* all its proof rules proposed in Coq, with respect to the Wasm 1.0 operational semantics introduced in Chapter 2 and Chapter 3. This differs from a traditional non-mechanised approach, where a series of derivation rules is proposed, with the soundness of the rules only justified by pen-and-paper arguments.

The Iris framework provides a streamlined process to instantiate a separation logic of a language definition. This is done in two steps: defining a language semantics in the particular way that Iris requires, and then defining the necessary predicates for Iris to produce a separation logic on top of the language.

For a language without concurrency like Wasm 1.0, Iris requires the following definitions to construct the semantics of a language, known as the `LanguageMixin` typeclass:

- A type of *states* σ, representing the stateful entities of a language. For Wasm, this would include both the store S and the frame F due to the frame containing the local variables.
- A type of *expressions* E, representing the program code of a language. For Wasm, this is precisely the list of administrative instructions *es*.
- A type of *values* V, representing the terminal results of execution, and *bidirectional conversions* $\mathsf{of_val} : V \to E$ and $\mathsf{to_val} : E \to \mathsf{option}\ V$ between values V and expressions E, where the conversion from expressions to values is partial, and the proofs that the composition of

these two conversions in either order is identity (up to partiality). This essentially requires that the values are in bijection to a subset of the expressions. For Wasm, the logical values defined above precisely form the type V.

- A small-step *reduction relation* of the form

$$(\sigma;E) \hookrightarrow (\sigma';E')$$

which defines the execution semantics of the language. For Wasm, this is the existing small-step reduction relation

$$(S;F;es) \hookrightarrow (S';F';es')$$

with a minor wrapper definition to ensure that it fits the required form of Iris due to the tuple $(S;F)$ being the state σ.

- A proof that values cannot be reduced further in the language. This is stated as the contrapositive in Iris, i.e. if an expression can be reduced further, then it is not a value:

$$\forall \sigma, \sigma', e, e'.(\sigma;e) \hookrightarrow (\sigma';e') \implies \mathsf{to_val}(e) = \mathsf{None}$$

Note that this is different from the progress property. In particular, Iris does not *inherently* deals with any syntactic type system. Instead, a set of *semantic typing relations* formulated based on assertions of the language can be defined; this is a well-known method [2] which is used in building logical relations for types.

Once the language semantics has been instantiated, it remains to define a *state interpretation* to represent the states σ using Iris assertions. Concretely, a state interpretation is a function from the type of states to the type of Iris assertions iProp. Iris provides a library gen_heap for generating this state interpretation for common heap representations modelled as finite maps. This fits Wasm's store well, since as introduced in Section 2.2, Wasm's store is a large record containing various categories of Wasm states, including functions, tables, memories, and globals. Each of the category can be modelled by a map from the corresponding addresses to the states – for tables and memories, finer modularity can be achieved by representing as a map from a 2-dimensional address (n,i) to the individual bytes or table elements. Each of these is then associated with a points-to predicate This is illustrated in the following figure Figure 5.1, which is an expanded version of Figure 2.2 that displays the Iris points-to predicates that correspond to each category of Wasm states. For example, $n \xmapsto{\mathsf{wm}}_i b$ asserts that the ith byte of memory n is b. These resource predicates can then be used later in the proof rules.

Weakest preconditions Our proof rules are phrased using Iris' *weakest precondition*, the wp predicate. In fact, Iris allows the weakest predicate to be customly modified by the user. However, for most languages, the default version provided by Iris is desired. Iris-Wasm uses the default version with a slight modification due to Wasm's control flow, which we will discuss later in Section 5.2.2 and Section 5.2.3. Intuitively, the default $\mathsf{wp}\, es\, \{w, \Phi(w)\}$ states that the program

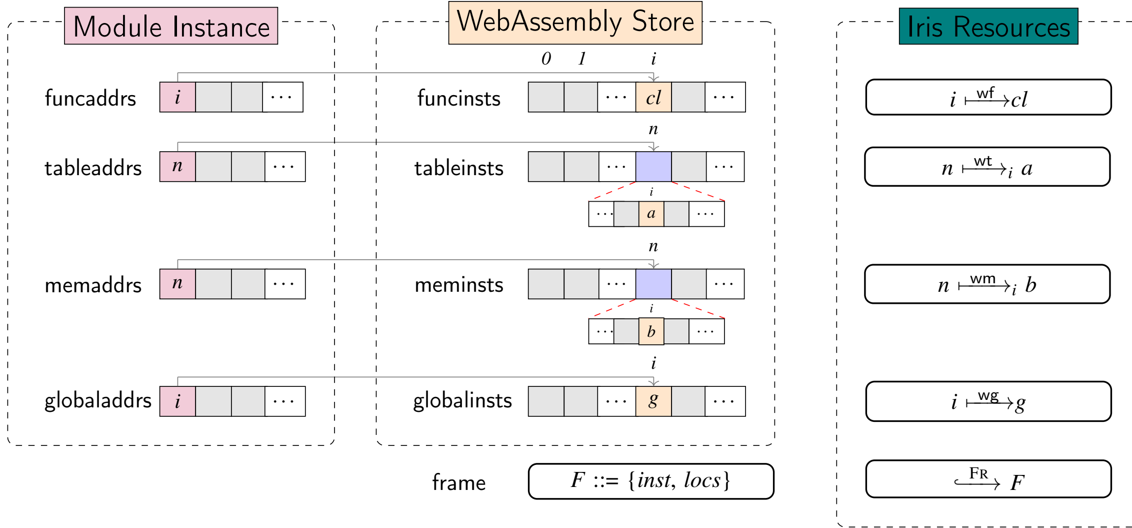


Figure 5.1: Points-to predicates for the store and the frame

fragment *es* computes safely[1], and, if it terminates with result w, predicate Φ holds of w. This construct is close to Hoare triples, as we have the following equality in Iris:

$$\{P\}\; es\; \{w, \Phi(w)\} = \Box(P \twoheadrightarrow \mathsf{wp}\; es\; \{w, \Phi(w)\})$$

The persistent modality $\Box$ indicates that the Hoare triple is a proposition that can be duplicated as many times as needed. This ensures that P indeed contains all the non-duplicable resources required for the weakest precondition.

Logical values Recall the design of the verified interpreter from Section 3.4, where intermediate results for the **br** and **return** control flow instructions are required for handling the execution of program fragments that may proceed further with the knowledge of its enclosing **label** and **frame** contexts. Similarly, because we reason about *fragments* of WebAssembly programs, the execution result needs to be extended more generally to a *logical value*:

$$LogVal \ni w \quad ::= \quad \textbf{immV}\; vs \;\mid\; \textbf{trapV} \;\mid\; \textbf{brV}\; i\; vh_i \;\mid\; \textbf{retV}\; lh_k \;\mid\; \textbf{call_hostV}\; tf\; hidx\; vs\; llh$$

which is one of the following:

- **immV** *vs*, the ‘normal’ result: a stack of WebAssembly values;
- a trap **trapV**, which represents that the program has encountered an error in its execution;
- a break (or branching) value **brV**, a return value **retV**, or a host call value **call_hostV**, which correspond to program fragments that are stuck as such, but can get unstuck when placed in an appropriate context. This is similar to the RS_break and RS_return intermediate results used in the initial version of the mechanised interpreter in Section 3.4. We explain their concrete definitions and the meaning of their arguments in Section 5.2.3.

[1] We’ll come back to a more detailed version of the meaning of this later in this section under ‘Proving the proof rules’.

Accordingly, in our proof rules, the postcondition Φ takes a logical value w as an argument.

Proof rules In this chapter, Iris-Wasm proof rules will be presented in the style of derivation rules, with the preconditions (premises) written over the derivation line and the wp predicate below the line. For example, a proof rule in the shape of

$$\frac{P}{\mathsf{wp}\, es\, \{w, \Phi(w)\}}$$

should be read as

$$\Box(P \twoheadrightarrow \mathsf{wp}\, es\, \{w, \Phi(w)\})$$

As an example, the proof rule we proved for the **local**.**get** instruction is as follows:

$$\texttt{WP_LOCAL_GET}\quad \frac{\ulcorner F.\mathsf{locals}[i] = v \urcorner \;*\; \triangleright\Phi(\mathbf{immV}\ [v]) \;*\; \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\ [\mathbf{local.get}\ i]\left\{w, \Phi(w) * \xhookrightarrow{\text{FR}} F\right\}}$$

which states that, with any function frame F with the ith local variable being v, if an arbitrary predicate Φ holds *later* for the value v, then this program fragment executes safely; moreover, if it terminates (which it does in this case), then Φ holds for the execution result w. This agrees with the operational semantics rule for **local**.**get**. The $\ulcorner P \urcorner$ notation lifts a pure proposition P in Coq (`Prop`) to the type of Iris propositions (`iProp`). This notation will be omitted for the remaining of this thesis.

We merely require that Φ holds after one step of execution, as expressed by the later $\triangleright$ modality of Iris [53]. One may choose to ignore this, but it is necessary in the presence of Iris' higher-order features, to avoid cyclicity.

Note that it is also possible to formulate a equivalent but slightly different-looking rule for **local**.**get** as follows, which readers unfamiliar with Iris might find it easier to understand:

$$\frac{F.\mathsf{locals}[i] = v \;*\; \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\ [\mathbf{local.get}\ i]\left\{w, (w = v) * \xhookrightarrow{\text{FR}} F\right\}}$$

The above formulation avoids the parametric assertion Φ and simply provides a postcondition stating that the result of executing the program fragment would be equal to v, up to termination. However, the first formulation is much easier to use in practice. This is because it allows any desired post condition to be proven to be transported directly to a premise that needs to be satisfied in the current proof environment. We will revisit this later.

Proving the proof rules With the Wasm language instantiated in Iris and the state interpretation defined, we can now discuss what it means to *prove* the proof rules we proposed in Iris with respect

to the underlying operational semantics. In Iris, to prove a proof rule of the form

$$\frac{P}{\mathsf{wp}\, e\, \{w, \Phi(w)\}}$$

it suffices to prove that, given any state σ satisfying P, if e is a value, then $\Phi(\mathsf{to_val}(e))$ holds in σ; otherwise, (σ, e) will execute safely – concretely, it will be able to take a step in the operational semantics[2] – and for *any* possible (σ', e') that (σ, e) can reduce to, $\mathsf{wp}\, e'\, \{w, \Phi(w)\}$ hold in that state σ' *later*. Note that the last part means that the wp predicate is recursive: indeed, it is defined as a *fixpoint* over the one-step definition.

Proving the wp_local_get proof rule is mostly straightforward. For any state $(S;F)$ satisfying the premise of the rule, i.e. with $F.\mathsf{locals}[i] = v$, the following reduction holds:

$$(S;F;[\mathbf{local}.\mathbf{get}\ i]) \hookrightarrow (S;F;[v])$$

which provides the reducibility witness.

Now, it might look trivial to conclude that the proof rule is proven: the postcondition is immediately satisfied in the above reduction because v is a value and $\Phi(v)$ is provided in the precondition. However, there's a catch: the wp predicate requires to prove that *all* possible reduction results satisfy the required condition. While it is true that the above reduction is the only possible one in Wasm, proving it requires far more effort, and replicating a similar proof for every individual proof rule would be extremely tedious and unscalable. Instead, recall from Chapter 3 that a large part of the Wasm 1.0 semantics is deterministic. In Iris-Wasm, we proved a result formulating a condition for a Wasm 1.0 program fragment to behave completely determinstically. This result can then be used in all proof rules, since the satisfaction of the post condition in the one reduction path can now lead to the conclusion of the proof rule, except for the non-deterministic cases such as **memory**.**grow**, which has to be separately proved.

In the rest of this section, we discuss the Iris-Wasm proof rules by their categories.

5.2.2 Pure Rules

Pure rules in Iris refer to those that do not interact with the state σ. For Wasm, this would include the set of all numeric, parametric, and variable instructions. One example is the following rule for a general binary operation **binop**:

WP_BINOP

$$\frac{[\![t.binop]\!](c_1, c_2) = c \;\ast\; \triangleright\Phi(\mathbf{immV}\,[t.\mathbf{const}\ c]) \;\ast\; \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\,[t.\mathbf{const}\ c_1; t.\mathbf{const}\ c_2; t.\mathbf{binop}\ binop]\left\{w, \Phi(w) \ast \xhookrightarrow{\text{FR}} F\right\}}$$

[2]Technically, the default wp construct in Iris also takes a "stuckness" flag which indicates whether this reducibility condition is required as part of the wp predicate. In our work, all the proof rules are proven with a universal quantifier over the flag. In other words, Iris-Wasm proves the harder version which requires the reducibility condition.

which essentially states that the evaluation result of the **binop** instruction is precisely defined by the $[\![t.binop]\!]$ proxy predicate.

Modification to Iris' default weakest precondition One unusual aspect of the rule is that the frame resources are included in both the premise and the postcondition despite the frame not being involved in the execution of t.**binop**. In fact, all rules corresponding to instructions that *take a step* require the frame resource in the precondition, and therefore produces one in the postcondition.

This is due to the modification in the Iris-Wasm weakest precondition from the default version Iris provides – in particular, the wp predicate is modified to require the existence of ownership of a frame resource at every step of execution. This modification allows a *frame switching* rule, wp_frame_bind, to be proved later for reasoning about function calls. We will mention this again in Section 5.2.3. This additional requirement in the wp predicate is sound with respect to Wasm's execution semantics: recall from Section 2.2 that the frame F is a local resource to the current thread. Even in a future version of Wasm where concurrency or multi-threading is introduced, each thread will have their own function frame F, which is not shared to other threads, so it is sound to assume the full ownership of the frame predicate when executing the program fragment in the current thread.

5.2.3 Control and Function Call Rules

We present an approach that allows us to reason about code fragments without needing knowledge of their environment. It improves over the approach taken in the earlier Wasm program logic [113] which does not scale to higher-order programs. In this section, we show how our rules capture this locality to make reasoning tractable.

Blocks, Labels, and Breaks

Recall from Section 2.3.5 that the operational semantics of the block-type instructions are expressed based on the **label** administrative instruction, where reduction steps can be taken in the body of the **label** instructions. As introduced in Section 2.3.5, WebAssembly defines its block evaluation contexts B_k, which describe stack environments consisting of k nested *labels* surrounding a *hole* $[_]$ where the next step of execution takes place:

$$B^0 ::= vs +\!\!+ [_] +\!\!+ es \qquad B^{k+1} ::= vs +\!\!+ \textbf{label}_n\{es_{cont}\}\ B^k\ \textbf{end} +\!\!+ es$$

Note how only (constant) values vs can be on the left of the hole and label instructions: this enforces that we can only 'zoom in' on the next expression to reduce.

As explained in Section 2.3.5, steps can be taken under an evaluation context: if es reduces to es', then $B^k[es]$ reduces to $B^k[es']$. Taking $k = 0$ yields the traditional sequencing rule.

Correspondingly, we prove the following proof rule, which reduces reasoning about a program fragment that can be decomposed as $B^k[es]$ to reasoning about $B^k[vs]$, that is, the result vs of

evaluating the expression to a list of constants, placed in the evaluation context. [3]

$$\frac{\texttt{WP_CTX_BIND} \quad \mathsf{wp}\, es \left\{ w, \mathsf{wp}\, B^k[w] \left\{ w', \Phi(w') \right\} \right\}}{\mathsf{wp}\, B^k[es] \left\{ w', \Phi(w') \right\}}$$

This rule leverages the fact that in Iris, weakest preconditions are propositions themselves, and can therefore be nested. Notice how we have implicitly cast w, a logical value, into an expression when plugging it into B^k. This is done in the intuitive way: **immV** vs is cast into vs, **trapV** is cast into the single administrative instruction [**trap**], etc. We omit the implicit casts between logical values and the corresponding expressions in the rest of this chapter.

While control flow in WebAssembly is structured, the presence of labelled breaks makes it slightly involved. A break targets a particular level of the evaluation context, and skips the rest. As a result, the default evaluation context rules provided by Iris are inadequate, and we have to build our own reasoning principles for contexts.

The **br** i instruction targets the ith label from the context. Crucially, breaking relies on the instruction **br** i being in an evaluation context B^k with $i = k$: the break index indicates what context depth is targeted. If $i > k$, the expression $B^k[\mathbf{br}\ i]$ is stuck and can only reduce if placed in a deeper context. Correspondingly, we introduce a new type of logical values: **brV** $i\ vh_i$, representing the program fragment $vh_i[\mathbf{br}\ i]$. The *breaking context* vh_i is similar to an evaluation context B^i, except that the meaning of the subscript i is that the context has depth *at most* i, instead of exactly i. If $i < k$, a **br** i nested in context B^k will only break out of the i first **label**s, and the result will be in the form $B^{k-i}[vs +\!\!+\ es]$. The break value **brV** allows to bind into any number of **label**s without needing to worry about getting stuck at a **br** i statement: when encountering such a statement, we simply bind back $i+1$ times to get a wp in a form where our rule for **br** can be applied.

Example: First-Order Sequential Programs

$$\mathsf{body}_{\mathit{fib}} ::= [\mathbf{local.get}\ 1; \mathbf{local.get}\ 2; \mathbf{i32.add}; \mathbf{local.get}\ 2; \mathbf{local.set}\ 1; \mathbf{local.set}\ 2]$$

Figure 5.2: Selected core program fragment of the Fibonacci function

To display the application of the bind rule and the other first-order Iris-Wasm proof rules, we revisit a simple, yet core program fragment of the example Fibonacci module introduced in Section 2.8. This program fragment performs the main evaluation of the next value of the Fibonacci sequence given the current two values and manage the storage of local variables, which is displayed in Figure 5.2. The following specification could be proved for this program fragment:

$$\begin{array}{l}\left(F.\mathsf{local}[1] = \mathbf{i32.const}\ a_1 \ast F.\mathsf{local}[2] = \mathbf{i32.const}\ a_2 \ast \xhookrightarrow{\text{FR}} F\right) \mathrel{-\!\!\ast} \\ \mathsf{wp}\ \mathsf{body}_{\mathit{fib}} \left\{ w.\ w = \mathbf{immV}\ [] \ast F'.\mathsf{local}[1] = \mathbf{i32.const}\ a_2 \ast F'.\mathsf{local}[2] = \mathbf{i32.const}\ (a_1 + a_2) \ast \xhookrightarrow{\text{FR}} F' \right\}\end{array}$$

This specification is straightforward: the only resource involved is the frame, and no values will

[3] In our Coq formalization, this rule in fact consists of several variants, where the rule with this exact name only applies to the case where $k > 0$. Some other variants of this rule covers sequencing, e.g. with $B^k[es]$ replaced with $B^k[es_1 +\!\!+\ es_2]$, which needs special consideration if the first part of the program fragment es_1 traps, since Wasm's **trap** absorbs other contents of the stack. In hindsight they could probably have been refactored better.

be added to the stack at the end of this program fragment.

Proving this specification is an exercise in application of the bind rules and the corresponding Iris-Wasm proof rules for the individual instructions. We demonstrate this for the first step of execution, which can be done by first binding into the context of the first instruction, which is the entire body_{fib} except for the first instruction, followed by applying the `wp_local_get` rule. We define the following shorthand before writing out the proof:

$$B_0 \triangleq [\mathbf{local}.\mathbf{get}\ 2; \mathbf{i32}.\mathbf{add}; \mathbf{local}.\mathbf{get}\ 2; \mathbf{local}.\mathbf{set}\ 1; \mathbf{local}.\mathbf{set}\ 2]$$
$$Q \triangleq (F'.\mathsf{local}[1] = \mathbf{i32}.\mathbf{const}\ a_2 \ * \ F'.\mathsf{local}[2] = \mathbf{i32}.\mathbf{const}\ (a_1 + a_2))$$

Apply the bind rule:

$$\frac{\mathsf{wp}\ [\mathbf{local}.\mathbf{get}\ 1] \left\{w, \mathsf{wp}\, B_0[w] \left\{w',\ w' = \mathbf{immV}\ []\ *\ Q * \xhookrightarrow{\text{FR}} F'\right\}\right\}}{\mathsf{wp}\, \mathsf{body}_{fib} \left\{w,\ w = \mathbf{immV}\ []\ *\ Q * \xhookrightarrow{\text{FR}} F'\right\}}\ \texttt{wp_ctx_bind}$$

Next we need to apply the `wp_local_get` rule. Note that the current post condition doesn't explicitly contain a frame resource $\xhookrightarrow{\text{FR}} -$ to match the post condition of the `wp_local_get` rule. However, the following implication, similar to *modus ponens*, holds in separation logic:

$$(P \twoheadrightarrow Q) \ * \ P \implies Q$$

Therefore, we can apply the above implication with $P = (\xhookrightarrow{\text{FR}} F)$ to match the post condition of `wp_local_get`:

$$\frac{(F.\mathsf{locals}[1] = v)\ *\ (w = \mathbf{immV}\ [v])\ *\ \left[\xhookrightarrow{\text{FR}} F \twoheadrightarrow \mathsf{wp}\, B_0[w] \left\{w',\ w' = \mathbf{immV}\ []\ *\ Q * \xhookrightarrow{\text{FR}} F'\right\}\right]\ *\ \xhookrightarrow{\text{FR}} F}{\mathsf{wp}\ [\mathbf{local}.\mathbf{get}\ 1] \left\{w, (\xhookrightarrow{\text{FR}} F \twoheadrightarrow \mathsf{wp}\, B_0[w] \left\{w',\ w' = \mathbf{immV}\ []\ *\ Q * \xhookrightarrow{\text{FR}} F'\right\}) * \xhookrightarrow{\text{FR}} F\right\}}\ \texttt{wp_local_get}$$

Now, recall our original precondition

$$\left(F.\mathsf{local}[1] = \mathbf{i32}.\mathbf{const}\ a_1 \ * \ F.\mathsf{local}[2] = \mathbf{i32}.\mathbf{const}\ a_2 * \ \xhookrightarrow{\text{FR}} F\right)$$

which provides the frame resource $\xhookrightarrow{\text{FR}} F$. The value of $F.\mathsf{local}[1]$ can then be unified, substituting v with $\mathbf{i32}.\mathbf{const}\ a_1$ and also w with the corresponding value. This results in the following proof obligation remaining:

$$\left(F.\mathsf{local}[1] = \mathbf{i32}.\mathbf{const}\ a_1 \ * \ F.\mathsf{local}[2] = \mathbf{i32}.\mathbf{const}\ a_2 * \ \xhookrightarrow{\text{FR}} F\right) \twoheadrightarrow$$
$$\mathsf{wp}\, B_0[\mathbf{i32}.\mathbf{const}\ a_1] \left\{w',\ w' = \mathbf{immV}\ []\ *\ Q * \xhookrightarrow{\text{FR}} F'\right\}$$

which is expected: $B_0[\mathbf{i32}.\mathbf{const}\ a_1]$ corresponds to the program fragment obtained after one step of execution (executing **local**.**get** 1; the post condition remains as the desired final state of executing the program fragment; and the resources owned haven't changed since **local**.**get** recovers the same frame as it needs, as specified by the `wp_local_get` proof rule. The rest of the proof can be performed in a similar fashion, binding into the corresponding block context at each step.

To conclude this example, we note two important observations from the above proof:

- Note again how the shape of `wp_local_get` allows the rule to be directly applied to the

relatively complex post-condition desired by matching with the parametric assertion $\Phi(w)$ in the rule. If the alternative version of the proof rule is used, then a separate step of implication is required to massage the shape of the post condition. With the current bind rule and the parametric shapes of the proof rules, one implication can be saved per proof step.

- Despite the above, the proof for this one step of execution is still tedious. This issue is exacerbated in Coq: at each stage of the proof in Iris, a set of Coq propositions and a set of Iris propositions are available in the context as the current premises/resources, each associated with a name in Coq. Applying the proof rules require carefully selecting the appropriate resources to be provided, which needs to also match exactly with the order stated in the proof rules. After applying a proof rule, the resources obtained in the post condition, formulated as a separating conjunction of several Iris propositions, need to then be split into individual Iris propositions and organised back to the "pool" of resources available for the future proof. In fact, a lot of the lines of code is devoted to the organising the resources well. Although Iris provides some automatic tactics to help this process, such as `iFrame` which automatically consumes any resources required by the current proof obligation, they are insufficient to cover all the need of resource management in Iris-Wasm. There is also no *proof mode* that helps deducing the current rule to be applied – which is obvious in most of the cases (bind rule followed by a proof rule corresponding to the current instruction). As a result, the users have to take delicate care of rule applications and resource managements themselves.

Functions

There are two ways to call a function in WebAssembly: statically with **call**, or by dynamically fetching a function from a table, with **call_indirect**. We focus on the simpler direct **call** here, and explain **call_indirect** in Section 5.2.4.

The instruction **call** i calls the ith function declared in the current module. Recall from Section 2.7 that function indexing starts at 0 with the imported functions, followed by the functions defined in the module itself. The store S keeps a list of the *function closures* (which we describe below) of *all* the instantiated modules. This means the ith function in the current module will not always be the ith function in the store: the *instance* in the function frame F is in charge of remembering that indirection. The instance also contains this indirection information for global variables, memories, and tables.

A **call** i retrieves the address $F.\mathsf{inst.funcs}[i] = addri$ of the relevant closure in the store from the frame's instance, and reduces to **invoke** $addri$. We prove the corresponding Iris-Wasm rule:

wp_call
$$\frac{(F.\mathsf{inst.funcs}[i] = addri) * \xhookrightarrow{\text{FR}} F * \triangleright \left(\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathsf{wp}\ [\textbf{invoke}\ addri]\ \{w, \Phi(w)\}\right)}{\mathsf{wp}\ [\textbf{call}\ i]\ \{w, \Phi(w)\}}$$

which requires ownership of the frame, not only because we are taking a reduction step, but also to know where to look up index $addri$.

The function closures cl (also called function instances *funcinst* in Figure 2.4) stored in the store S are of two kinds: native and host functions. We first focus on native closures, and come back to host closures at the end of this section.

Recall that the closure $\{ts_1 \to ts_2, (inst;ts), es\}^{\mathsf{Native}}$ describes a *native* function that was defined in a WebAssembly module with instance *inst* (this is the environment for the closure), which expects arguments of type ts_1, defines additional local variables of type ts for the computation of its body, yields results of type ts_2, and has body es. We also recall the operational semantic rules for **invoke** from Figure 2.12: when reducing **invoke**, we look up the closure in the store, and check that the stack contains the appropriate number of values to be passed as parameters to the function. If the closure is native, **invoke** is replaced with the body of the function. In order to properly encapsulate the function call, WebAssembly places the function body inside a **frame** administrative instruction, and inside a **block**, as captured by the following Iris-Wasm proof rule (we say more about **frame** and the meaning of F' further down):

wp_invoke_native

$$\frac{\begin{array}{c}|vs| = |ts_1| * cl = \{(ts_1 \to ts_2), (inst;ts), es\}^{\mathsf{Native}} * F' = \{\mathsf{locs} := vs +\!\!+ \mathbf{zeros}(ts); \mathsf{inst} := inst\} * \\ i \xmapsto{\mathsf{wf}} cl * \xhookrightarrow{\mathrm{FR}} F * \triangleright \begin{bmatrix} (i \xmapsto{\mathsf{wf}} cl * \xhookrightarrow{\mathrm{FR}} F) \mathrel{-\!\!*} \\ \mathsf{wp}\ [\mathbf{frame}_{|ts_2|}\ F'\ (\mathbf{block}\ ([] \to ts_2)\ es)\ \mathbf{end}]\ \{w, \Phi(w)\} \end{bmatrix}\end{array}}{\mathsf{wp}\ (vs +\!\!+ \mathbf{invoke}\ i)\ \{w, \Phi(w)\}}$$

Note that unlike for the function frame F, we do not assert ownership of the whole store S. Instead, the traditional separation logic points-to predicates are used to assert ownership of specific components, which were introduced in Figure 5.1. Particularly, in the rule above, the predicate $i \xmapsto{\mathsf{wf}} cl$ asserts ownership of $S.\mathsf{funcs}[i]$ in the store.

Encapsulation Let us return to why the function body is placed inside a **frame** and inside a **block**. The first of these is to provide proper encapsulation, as reduction of an expression nested in a **frame** takes place with respect to the nested frame of the **frame**: when reducing $[\mathbf{frame}_n\ F_1\ es\ \mathbf{end}]$, one reduces es with respect to frame F_1 rather than the current function frame F.

For our native invocation, the frame used will be F'. Note that the inst field of F' is the instance that was declared in the closure (to enforce static scoping), and that the local variables in F' are the function parameters from the stack, followed by a list of zeros corresponding to the types of local variables required by the function. We prove the corresponding proof rule for **frame**:

wp_frame_bind

$$\frac{\xhookrightarrow{\mathrm{FR}} F * \left(\xhookrightarrow{\mathrm{FR}} F_1 \mathrel{-\!\!*} \mathsf{wp}\, es \left\{w, \exists F_1', \xhookrightarrow{\mathrm{FR}} F_1' * \left(\xhookrightarrow{\mathrm{FR}} F \mathrel{-\!\!*} \mathsf{wp}\ [\mathbf{frame}_n\ F_1'\ w\ \mathbf{end}]\ \{w', \Phi(w')\}\right)\right\}\right)}{\mathsf{wp}\ [\mathbf{frame}_n\ F_1\ es\ \mathbf{end}]\ \{w', \Phi(w')\}}$$

which is reminiscent of wp_ctx_bind; the only reason this rule looks like more of a mouthful, is that the frame changes. As discussed above, this frame change is necessary for proper encapsulation. This rule is also the reason that Iris-Wasm has to use a slightly modified wp predicate from the default version provided by Iris: at the end of the function call, the existence of a frame resource is required to perform the frame switch from the inner frame back to the original frame F

of the outer closure. Note that there will always be a frame in the Iris state, since F is always part of the state σ that will never be consumed. This knowledge therefore needs to be encoded into the wp predicate used by Iris-Wasm. As a consequence, all proof rules now need to verify that it preserves the *existence* of the frame resource, which is true by Wasm's operational semantic rules.

Finally, the reason WebAssembly puts the function body in a **block** is to allow the function body to contain a **br** (with the right index) to exit the function-body's execution. Alternatively, a **return** instruction will work like a **br**, but target the closest **frame** instruction. The **return** instruction also has an associated logical value **retV** B^k, representing the expression $B^k[\textbf{return}]$.

Host functions WebAssembly is meant to be defined independently of the host language in which it is embedded. However, the way the WebAssembly standard is phrased assumes that it is given some operational semantics of the host language as input, and embeds it in the operational semantics of WebAssembly. This phrasing suffices for defining the semantics of WebAssembly alone, which is what the WebAssembly standard does. However, when providing the first formal integration of WebAssembly with a separately-defined host language, we identified that this phrasing is limiting, because it prevents formally giving the semantics of the combined host and embedded language as the integration of two concrete, separately defined language.

To account for this, we modify the presentation of the WebAssembly semantics (this is our only point of departure from the WasmCert-Coq formalization of Watt et al. [114]) so that the **invoke** of a host function reduces to a new **call_host** administrative instruction:

invoke_host
$$\frac{(S.\mathsf{funcs}[i] = \{ts_1 \to ts_2, hidx\}^{\mathsf{Host}}) * (|ts_1| = |vs|)}{(S; F; vs ++ [\textbf{invoke}\ i]) \hookrightarrow (S; F; [\textbf{call_host}\ (ts_1 \to ts_2)\ hidx\ vs])}$$

The closure $\{ts_1 \to ts_2, hidx\}^{\mathsf{Host}}$ represents a *host* function imported from the host language that expects arguments of type ts_1 and yields results of type ts_2. The argument *hidx* is an identifier that the host will use to determine what the desired function is. The **call_host** instruction remembers the function type *tf*, the 'host identifier' *hidx* that allows the host language to identify which function is being called, and the function arguments *vs*. A **call_host** is stuck, and can only be unstuck by the host language, which typically replaces it by the return value of the call, possibly changing the frame or the store in doing so. We say more about the host interaction in §5.4.

We prove the following Iris-Wasm proof rule:

WP_INVOKE_HOST
$$\frac{|vs| = |ts_1| * cl = \{(ts_1 \to ts_2), hidx\}^{\mathsf{Host}} * i \overset{\mathrm{wf}}{\mapsto} cl * \overset{\mathrm{FR}}{\hookrightarrow} F * \triangleright \left[\begin{array}{c}(i \overset{\mathrm{wf}}{\mapsto} cl * \overset{\mathrm{FR}}{\hookrightarrow} F) -\!\!* \\ \mathsf{wp}\ (\textbf{call_host}\ (ts_1 \to ts_2)\ hidx\ vs)\ \{w, \Phi(w)\}\end{array}\right]}{\mathsf{wp}\ (vs ++ [\textbf{invoke}\ i])\ \{w, \Phi(w)\}}$$

We introduce the **call_hostV** *tf hidx vs llh* logical value, representing the stuck value $llh[\textbf{call_host}\ tf\ hidx\ vs]$. This allows for seamless binding rules when we introduce the host language's logical rules in §5.4. Since a **call_host** instruction is also stuck if it is under a **frame** or under a **label**, we remember the context *llh* around the **call_host** as the fourth argument of

```
stack_module ≜
(module ;; "stack"
  (type $t1 (func (param i32) (result i32)))
  (table (export "tab1") 3 funcref)
  (memory 0)
  (func (export "map")
        (param $i i32) (param $stk i32)
   ...
   loop
    ...
    local.get $i
    call_indirect  $t1
    ...
   end ...)
```

```
client_module ≜
(module ;; "client"
  (import "stack" "tab1" (table 3 funcref))
  (import "stack" "map"
   (func $map (param i32 i32)))
  (elem (i32.const 0) $f0 $f1 $f2)
  (func $f0 (param $n i32) (result i32)
   ...)
  (func (export "main")
        (param $stk i32) (result i32)
   i32.const 0
   local.get $stk
   call $map
   ... ;; Rest of the code))
```

Figure 5.3: A module implementing a stack library, and a client module.

call_hostV. This context *llh* is a generalized version of B^k, that has a hole in nested **frame**s and **label**s. In the rule above, wp (**call_host** $(ts_1 \to ts_2)$ *hidx vs*) $\{\Phi\}$ is thus a weakest precondition on a value, and it thus suffices to show that Φ(**call_hostV** $(ts_1 \to ts_2)$ *hidx vs* [_]).

5.2.4 Higher-Order Code with `call_indirect`

As explained in Section 2.3.5, Wasm uses the **call_indirect** instruction to implement higher-order functions with the help of the host language. The instruction **call_indirect** *i*, where *i* is an index into the types field of the module instance in the function frame, takes one argument *k* from the stack, and uses it as an index to look up the function to call in the table. The table itself is located in the store. Like for function invocation, the instance in the frame *F* finds the store-index *ta* of the correct table (i.e. the one at the head of the tables field). Now the *k*th element *a* of the table indexed *ta* can be looked up, and used as the index in the function closures component of the store, to find the closure *cl* to execute. As a side condition, the type of the closure must match the one declared by index *i* (that **call_indirect** takes as an immediate). Finally, [**call_indirect** *i*] reduces to [**invoke** *a*], setting *cl* to be invoked in the next reduction step.

We prove the following program logic rule:

WP_CALL_INDIRECT_SUCCESS

$$\frac{\xhookrightarrow{\text{FR}} F * (F.\text{inst.tabs}[0] = ta) * (ta \overset{\text{wt}}{\mapsto}_k a) * (a \overset{\text{wf}}{\mapsto} cl) * (F.\text{inst.types}[i] = \text{typeof } cl) * \triangleright\Big((ta \overset{\text{wt}}{\mapsto}_k a) \mathrel{-\!\!*} (ta \overset{\text{wf}}{\mapsto} cl) \mathrel{-\!\!*} (\xhookrightarrow{\text{FR}} F) \mathrel{-\!\!*} \text{wp}\ [\textbf{invoke}\ a]\ \{w, \Phi(w)\}\Big)}{\text{wp}\ [\textbf{i32.const}\ k; \textbf{call_indirect}\ \ i]\ \{w, \Phi(w)\}}$$

Here, we use the points-to predicate for elements of the table: only ownership of the relevant *k*th element of the table is required. Notice how the rule passes the ownership of all three points-to predicates (frame ownership, table element ownership and function closure ownership) to the continuing weakest precondition.

5.3 Higher-Order Programming Example in WebAssembly and Reentrancy

Consider the WebAssembly snippet in Figure 5.3, which contains a module that works as a library implementing a stack of **i32**s (on the left), and a module that works as a client of that library (on the right). The library module, which the host language calls "stack" here, uses a **memory** (with initial size 0; some other function is in charge of allocating space for the stack) to implement a stack. The "stack" module exports a "map" function that maps a function over a stack. However, because WebAssembly is a first-order language, "map" does not take the function to map as an argument. Instead, "map" takes as argument an index, $i, into a table of 3 functions, "tab1", that this module creates and exports, and calls the function at that index in the table using **call_indirect**. The client module imports the same shared table of functions, and uses the **elem** directive to populate it (from offset 0) with functions it defines: $f0, $f1, and $f2. It also imports the "map" function from the "stack" module as $map, and its "main" function then calls the $map function with function index 0 as argument, which makes it map $f0 on the stack.

We now describe how Iris-Wasm can be used to give a modular specification of the stack module, and, in particular, the higher-order "map" function. A proof of the specification of the *instantiation* of the stack module is given at the end of Section 5.4. We emphasise that our logic supports verification of the client module relative to an abstract logical specification of the stack module; in other words, the encapsulation of the internal representation of the stack module is reflected in its specification.

Specification for the stack module functions We now outline what specifications for functions look like and, how they can be used by client modules. Take any function f. We write its specification in the general form:

$$\Box \exists cl\ P, \forall i\ vs\ xs, \quad \Psi(P, vs, xs) \twoheadrightarrow (i \overset{\mathsf{wf}}{\longmapsto} cl) \twoheadrightarrow \mathsf{wp}\ vs +\!\!+ [\mathbf{invoke}\ i]\ \{w, \Phi(P, w, xs)\}$$

with Φ and Ψ some predicates specific to the function f. Recall that the persistence modality $\Box$ simply indicates this specification can be duplicated as many times as needed; we omit this modality in every specification that follows, for simplicity. Note the existential quantifiers. The first one, cl, abstracts over the actual closure of function f; because it is hidden behind an existential, it is hidden from clients. The second one, P, allows the specification to reference some abstract representation predicate. In the case of the functions from the "stack" module, we will have an existentially quantified predicate **isStack**, which hides the data representation from clients. We put all specifications under one large existential $\exists cl_{\text{push}}\ cl_{\text{pop}}\ cl_{\text{map}}\ \ldots$ **isStack**, so that all specifications can share the predicate **isStack**.

The specification is thus a weakest precondition[4] on an **invoke**, with some precondition Ψ on the arguments vs given and some postcondition Φ. Both Ψ and Φ can mention the existentially quantified predicate P, as well as some universally quantified variables xs. The invocation address i is linked to the function f by the condition $i \overset{\mathsf{wf}}{\longmapsto} cl$, that asserts that the function body is stored

[4] In practice, we use the host weakest precondition $\mathsf{wp}_{\mathbf{HOST}} - \{-\}$ that we introduce in §5.4, as to allow functions to interact with the host via host calls. For functions that do not interact with the host, this makes no difference.

at address i. Let us give the concrete Φ and Ψ used for function "push":

$$\exists cl_{\text{push}}\ cl_{\text{pop}}\ cl_{\text{map}}\ \ldots\ \textbf{isStack}, \Big(\forall i\ v\ x\ s, \textbf{isStack}(v,s) \twoheadrightarrow (i \overset{\text{wf}}{\longmapsto} cl_{\text{push}}) \twoheadrightarrow \\ \text{wp}\ [\textbf{i32.const}\ x; v; \textbf{invoke}\ i]\ \{w, w = \textbf{immV}\ []\ * \textbf{isStack}(v, x :: s)\}\Big) * \ldots \text{(other specs)}$$

Given a specification written in this form, and given the resource $i \overset{\text{wf}}{\longmapsto} cl_{\text{map}}$,[5] a client can verify its code in the presence of a call to the imported map function: when arriving at the instruction **call** $map, **wp_call** reduces **call** to **invoke**, and now the specification shown above can be applied.

For example, when specifying the "main" function of the extended client module from Figure 5.3, one intermediate goal, when verifying the part of the code corresponding to the call to the host function $mut, would have the form $\text{wp}\, vs \mathbin{+\!\!+}$ **call** $mut $\{\Phi\}$, where vs represents the constant arguments we have pushed onto the stack prior to making the call. To prove this, one can simply apply rule **wp_call** to reduce **call** to **invoke**, and then rule **wp_invoke_host** to reduce the **invoke** to a **call_hostV** value. The computation is now reduced to a logical value, thus we now must prove that the postcondition Φ holds of the host call value. We cannot carry on to the rest of the code of the reentrant example if we stick at the WebAssembly level; this is in line with the nature of this call: it is a host call and needs interaction with the host to be unstuck. We will see in §5.4 how to reason about interaction with the host to prove the full specification of the reentrant example.

Higher-order specifications The higher-order "map" function of the stack module calls its argument function on each element in the stack by using **call_indirect**. Its modular specification can be presented as follows:

$$\begin{array}{ll} \exists cl_{\text{map}}\ \textbf{isStack}, \forall \Phi\ \Psi\ a\ v\ s\ F\ j\ k\ i, & (1) \\ \Box(\forall u. \Phi\ u \twoheadrightarrow \ldots \twoheadrightarrow \text{wp}\ (\textbf{i32.const}\ u; \textbf{invoke}\ a)\ \{v, \Psi\ u\ v * \ldots\}) \twoheadrightarrow & (2) \\ \quad \textbf{isStack}\ v\ s \twoheadrightarrow \text{stack_all}\ s\ \Phi \twoheadrightarrow & (3) \\ \quad\quad (\overset{\text{FR}}{\longleftrightarrow} F) \twoheadrightarrow (F.\text{inst.tabs}[0] = j) \twoheadrightarrow (j \overset{\text{wt}}{\longmapsto}_k a) \twoheadrightarrow \ldots \twoheadrightarrow (i \overset{\text{wf}}{\longmapsto} cl_{\text{map}}) \twoheadrightarrow & (4) \\ \quad\quad\quad \text{wp}\ [\textbf{i32.const}\ k; v; \textbf{invoke}\ i]\ \{w, \exists s'. \textbf{isStack}\ v\ s' * \text{stack_all2}\ s\ s'\ \Psi * \ldots\} & (5) \end{array}$$

Let us describe the specification line by line:

- As explained in Section 5.2.3, we existentially quantify over a closure cl_{map} and a predicate **isStack**, to hide our implementation of the stack and the body of the "map" function. We then universally quantify over many variables, including notably Φ and Ψ used in the specification of the mapped function, stressing this specification can be as general as needed.

- The first precondition is a specification for the mapped function; it uses two predicates Φ and Ψ to express that for any **i32** input u that satisfies Φ, the mapped function returns an **i32** result v such that Ψ relates u with v. We have used '…' to elide some predicates, which are simply a copy of some of the resources from line 4, so as to allow usage of those resources (like frame ownership) in the proof of the specification of the mapped function.

- Next, we describe the argument value v: it must represent a mathematical stack s, all ele-

[5] The name of the index i and ownership of this resource are provided by instantiation when the client does the import.

ments of which satisfy Φ. This is captured by the **isStack** $v\ s$ predicate.

- A points-to predicate for table j links the argument value k to the function index a (from the **invoke** in line 2). For brevity, we elide other side-conditions pertaining to typechecking the mapped function. At the end of the line, we have the function closure points-to predicate that links the index i of the invocation on line 5 to the "map" function closure.

- After running "map", we have a stack with logical state s' at location v, whose elements are related one-to-one to that of the previous logical state s by Ψ. For readability, we omit the second part of the postcondition, which simply gives back all of the resources from line 4.

To prove the above specification, the $stack module, who has access to the actual code of the "map" function, simply fills in the existential quantifiers with the actual closure of "map" and the definition of **isStack** reflecting the actual implementation. Then all that remains is a weakest precondition to prove, which is done by applying the rules in §5.2.3: `wp_invoke_native` using hypothesis $i \xmapsto{\text{wf}} cl_{\text{map}}$, then `wp_frame_bind`, to enter the function frame, etc.

Note that we rely on the fact that our ambient logic, Iris, is a higher-order separation logic, in which weakest preconditions are just usual propositions. We stress again that the user of "map" does not need to know how **isStack** is defined (and in fact, we hide it with an existential quantifier surrounding the specification of the stack module, again exploiting the higher-order logic of Iris) or the physical state of the stack representation in memory: they only need to reason about the mathematical state, s; for example, stack_all only refers to s.

This example demonstrates that Iris-Wasm can be used to prove specifications for modules that cleanly hide the heavy indirection and low-level details of WebAssembly.[6] The use of **call_indirect** for higher-order programming, to call an arbitrary client function, goes beyond the 'encapsulated' fragment of WebAssembly of Watt et al. [113], and yet is captured modularly in the first line of our specification. Our accompanying Coq formalization contains a formal proof that a simple implementation of the stack module meets the specification. We can then apply the specification to different clients.

We now consider a simple extension of this example to demonstrate the need for reasoning about reentrancy between WebAssembly and the host. To this end, we will let the "main" function, after the call to $map, dynamically modify the contents of the table to now contain a new function $f3 at index 0. Dynamic modification of the table cannot be performed in WebAssembly 1.0, as WebAssembly only has the **elem** directive available to statically provide an initial value for the elements of the table. WebAssembly code can, however, call functions defined by the host, and those may modify the state of the WebAssembly program. Thus we add an import (**import** "host" "mut" (**func** $mut (**param** **i32** **i32**))) to the preamble of the client module and then complete the code of the "main" function with 6 more instructions: **i32**.**const** 0; **i32**.**const** $f3; **call** $mut; **i32**.**const** 0; **local**.**get** $stk; **call** $map. The first three of these call the host function $mut that we assume will modify the function table at address 0, replacing the previous value ($f0) by $f3. The last three instructions are a call to $map identical to the one

[6]Indeed, the specification shown here is akin to the specification for a stack module implemented in an ML-like programming language in standard Iris [14].

(import variable) vi ::= nat (module variable) vm ::= nat (host action id) $hidx$ ::= nat

(declaration)	δ	::=	**inst_decl** $vis\ vm\ vis$ | **get_global** i
(host action)	a	::=	**nop** | **print** | **instantiate** δ | **call_wasm** | **table**.**set**
(import variable store)	I	::=	$vi \hookrightarrow export$
(host state)	H	::=	$\{\mathsf{store} : S,\ \mathsf{frame} : F,\ \mathsf{imports} : I,\ \mathsf{modules} : ms,\ \mathsf{actions} : as\}$
(host expression)	he	::=	$(es; \delta s)$ (host value) hw ::= $(vs; [])$ | $(\mathbf{trap}; [])$

Figure 5.4: Host Syntax (definitions reference the grammar in Fig. 2.5)

at the beginning of the body of "main" function (see Figure 5.3), but this time, when mapping the 0th function from the table onto the stack, it maps function $f3 instead of $f0 like it did during the first call to $map. Thus calling "main" on a value that represents stack $[x_0, \dots, x_n]$ will modify the stack so that the argument value now represents $[f_3(f_0(x_0)), \dots, f_3(f_0(x_n))]$.

This example illustrates how programs may take advantage of the stronger expressive power of the host. In Section 5.4, we introduce a simple host language and a program logic for it and show how it can be used in combination with our WebAssembly program logic to reason about complex interaction between WebAssembly code and the host language code that embeds it, including this example.

5.4 A Minimal Wasm-JS Host

In this section, we define a minimal host language featuring the core operations of the WebAssembly JavaScript Interface. The host fulfils two important roles; first, it embeds WebAssembly and defines the interoperability between WebAssembly and the host; and, second, it implements *module instantiation*, in which the host language handles the allocation of WebAssembly states. Our minimal host language also has the ability to mutate WebAssembly function tables.

We begin by introducing the syntax of the host language and selected proof rules, with a focus on the interoperability with WebAssembly. We then detail the rules for module instantiation.

The syntax of the host language is shown in Fig. 5.4. Host expressions are pairs of WebAssembly expressions and host-specific declarations; host values are pairs of WebAssembly values, and an empty list of declarations. Finally, the host state is a record of the WebAssembly store and frame, as well as host-specific state. Host specific state has three components. First, it includes a store of export objects, to store the exports of an instantiated module, and to feed the imports of future instantiations. Note that while we call them import variables, they are used both for imports and exports. Subsequently, an *export* object refers to any object passed from one module to another, either as import or export. Second, it keeps track of a list of WebAssembly modules. Finally, to maintain the generality of host calls, host actions are indirectly referenced by indices into a list of available host actions.

To illustrate the expressive power of a host, our minimal host language includes five different host actions. **nop**, **print** and **instantiate** δ are pure operations that do not depend on host or WebAssembly store. More noteworthy are the **call_wasm** and **table**.**set** operations: **call_wasm**

reduces to a WebAssembly call instruction, which opens up the possibility of reentrancy between the host and WebAssembly; **table.set** displays the expressive power of the host over the WebAssembly store, by mutating a given function table with a function from the WebAssembly store.

Declarations are either

- instantiations **inst_decl** *vis vm vis*, which consist of a list of import/ export variables to feed into the imports of a module (referenced indirectly by its index into the module store), whose exports are stored in the subsequent list of import/export variables, or
- load declarations for WebAssembly globals, to load the final output of a Wasm module's main function

.

The host operational semantics prioritises the reduction of WebAssembly expressions over that of instantiation declarations. We refer to the Coq formalization for a full account of the host operational semantics.

In the remainder of this section, we will discuss the proof rules of our new program logic for the host. We define our host logic using a weakest precondition predicate $\mathsf{wp}_{\mathbf{HOST}}\,(es;\delta s)\,\{hw,\Phi(hw)\}$, which intuitively means that the host expression $(es;\delta s)$ does not get stuck and, if it terminates with the host value hw, then the predicate Φ holds for hw.

While the host weakest precondition is not to be confused with the Wasm weakest precondition, it shares some similarities in its memory model. The memory model of the host program logic extends the memory model of the Wasm program logic, as it includes the Wasm store. We reason about the host-specific part of the host state using three new predicates:

- $vi \xmapsto{\mathsf{vis}} export$: a points-to predicate for the export object store;
- $vm \xhookrightarrow{\mathsf{mod}} m$: a points-to predicate for the module store;
- $hidx \xhookrightarrow{\mathsf{ha}} a$: a points-to predicate for the host action store.

We present the host program logic in two parts: first we discuss the rules that implement interoperability between WebAssembly and the host, and second we discuss module instantiation.

Interoperability The first key to WebAssembly and host interoperability is the WebAssembly lifting step. Any reduction in the WebAssembly part of a host expression corresponds to a step in the host expression, as captured by the following bind rule:

$$\frac{\mathsf{wp}\,es\,\{w,\mathsf{wp}_{\mathbf{HOST}}\,(w;\delta s)\,\{hw,\Phi(hw)\}\}}{\mathsf{wp}_{\mathbf{HOST}}\,(es;\delta s)\,\{hw,\Phi(hw)\}}\;\texttt{WP_LIFT_WASM}$$

Note that w may be a logical value, in particular a suspended host call from Wasm to the host, which can now be resolved via the host proof rules for **call_host**. Recall the definition of a stuck host call: the **call_host** *tf hidx vs* administrative instruction is considered stuck in any nested WebAssembly context *llh*, and is interpreted as the logical value **call_hostV** *tf hidx vs llh*, in

which *hidx* refers to the host action identifier which is storing the executing host action, *tf* refers to its type, and *vs* refers to the parameters of the invocation. Each host action is resolved via a different proof rule.

In particular, one such host action is a call in the other direction, from the host to Wasm. In that case, the inner **call_wasm** action, performed by the host function *hidx*, reduces to the WebAssembly instruction **call** as follows.

$$\frac{\texttt{WP_HOST_ACTION_CALL_WASM} \quad hidx \overset{\mathsf{ha}}{\hookrightarrow} \mathbf{call_wasm} \ * \ \triangleright (hidx \overset{\mathsf{ha}}{\hookrightarrow} \mathbf{call_wasm} \twoheadrightarrow \mathsf{wp}_{\mathbf{HOST}} \, (llh[\mathbf{call}\ i]; \delta s) \, \{hw, \Phi(hw)\})}{\mathsf{wp}_{\mathbf{HOST}} \, (llh[\mathbf{call_host}\ tf\ hidx\ [\mathbf{i32.const}\ i]]; \delta s) \, \{hw, \Phi(hw)\}}$$

Reentrant example We now have all we need to prove a specification for the extended (reentrant) client introduced in §5.3. This specification will be parametrized with specifications for all the functions from the stack module (and thus with all the existentials of those specifications, most importantly the **isStack** predicate), and can be *modularly* combined with a specification for the stack module.

Our specification could look like this:

$$\exists cl_{\mathrm{main}}, \forall v\ x_1\ \dots\ x_n\ i\ hidx, \quad \mathbf{isStack}\ v\ [x_1, \dots, x_n] \twoheadrightarrow i \overset{\mathsf{wf}}{\mapsto} cl_{\mathrm{main}} \twoheadrightarrow$$
$$\mathbf{OwnClosures}([\$\mathsf{f0}; \$\mathsf{f3}; \$\mathsf{map}]) \twoheadrightarrow \$\mathsf{mut} \overset{\mathsf{wf}}{\mapsto} \{[\mathbf{i32}; \mathbf{i32}] \to [], hidx\}^{\mathsf{Host}} \twoheadrightarrow hidx \overset{\mathsf{ha}}{\hookrightarrow} \mathbf{table.set} \twoheadrightarrow$$
$$\mathsf{wp}_{\mathbf{HOST}} \, ([\mathbf{i32.const}\ v; \mathbf{invoke}\ i], []) \, \{hw, \mathbf{isStack}\ hw\ [f_3(f_0(x_1)), \dots, f_3(f_0(x_n))] * \dots\}$$

The elided postconditions give back all the preconditions; **OwnClosures**(*fs*) asserts ownership, for all functions $f \in fs$, of a closure cl_f. For the function $map imported from the stack module, the closure is the one referenced in the specification of the stack module. In order to carry out our proof, we assume we are given specifications for functions $f0 and $f3 that reference cl_{f_0} and cl_{f_3}.

To prove this specification, we fill in the existential quantifier for cl_{main} with the actual code of the "main" function. Now we apply `wp_lift_wasm` to bring ourselves to proving a WebAssembly weakest preconditon: the postcondition now becomes $w, \mathsf{wp}_{\mathbf{HOST}}\, w \, \{hw, \Phi(hw)\}$ where Φ is the postcondition in the weakest precondition shown above. We can now begin the proof just like we proved all the specifications for the functions in the stack module: we apply `wp_invoke_native`, then `wp_local_bind`, etc.

As showcased in §5.2.3, the WebAssembly weakest precondition gets stuck on a value when it arrives at the host call: we now need to show that the postcondition holds of the **call_hostV** value, i.e. that

$$\mathsf{wp}_{\mathbf{HOST}}\, llh[\mathbf{call_host}\ tf\ hidx\ vs] \, \{hw, \Phi(hw)\}$$

where *llh* is the context in which the host call was, containing for instance all the code that follows the host call. To prove this, we have a rule `wp_host_action_table_set` similar to rule `wp_host_action_call_wasm` shown above, that, given our knowledge of $n \overset{\mathsf{wt}}{\mapsto}_0 \$\mathsf{f0}$, gives back $n \overset{\mathsf{wt}}{\mapsto}_0 \$\mathsf{f3}$, and brings us to prove a (host) weakest precondition statement on the code that follows the host call, with this new function at the 0th place in the table. We can prove this by lifting to WebAssembly and carrying out the proof in the WebAssembly program logic until the end.

Module instantiation While WebAssembly 1.0 does not depend on any particular host language, it does define a *specification* for module instantiation. Any host language is tasked with implementing instantiation according to that specification. We thus conceptually distinguish between the parts of module instantiation pertaining to the official WebAssembly specification, and the parts that deal with the host language. $\mathsf{Instantiate}(S, m, exportdescs, ((S', inst, exports), start))$ defines the specification for module instantiation. The full definition is quite elaborate; we refer to the Coq mechanization for all details, and provide an intuitive overview here. In essence, it states that *inst* is the result of instantiating module *m* while importing *exportdescs*, *exports* are the resulting exports, and S' is the resulting WebAssembly store, in which all the relevant state has been allocated.

The specification enforces various side conditions. First, the module must be well typed according to a list of relevant import and export types. Next, it asserts the necessary operational conditions on the allocated state and created instance; that all the fields of the instance are properly initialized (e.g. any function table is initialized with the proper elements as defined by the module), that all the initialized values are within the bounds of the initialized object, and finally that the start function is either empty, or refers to a function of the module of type $[] \to []$.

The instantiation specification specifies the outcome of module instantiation on the WebAssembly store. Note that the specification is host language agnostic. The semantic outcome of instantiation on the WebAssembly store ought likewise to be independent of the host language that implements it. The following lemma captures the effects of instantiation on the interpretation of the WebAssembly store as Iris resources, according to the host agnostic instantiation specification. The lemma is thus independent of any host language definition.

Lemma 5.4.1 (Module Instantiation Resource Allocation)**.**

$$\begin{array}{ll} \textit{If} & \vdash m : timps \to texps \ \wedge\ \mathsf{constInits}(m) \\ \textit{and} & \mathsf{Instantiate}(S, m, imports, ((S', inst, exports), start)) \\ \textit{then} & \mathsf{resourcesImports}(m, imports, timps, wfs, wts, wms, wgs) * \mathsf{stateInterp}(S) \\ & \Rrightarrow \mathsf{resources}(m, imports, timps, wfs, wts, wms, wgs, start, inst) * \mathsf{stateInterp}(S') \end{array}$$

For readability, we omit the technical details behind some of the above predicates. It suffices to know the following: $\mathsf{constInits}$ limits the global initializers and offsets to be constants, $\mathsf{resourcesImports}$ defines the points-to predicate associated with each import in $\mathsf{imports}$, and $\mathsf{resources}$ defines all the points-to predicates associated to the created instance *inst*, including those that were previously imported. The variables *wfs*, *wts*, *wms* and *wgs* are maps that summarise the values of functions, tables, memories and globals of the created instance. (The $\Rrightarrow$ modality is used in Iris to update ghost resources [53].) Using Lemma 5.4.1, we can then prove a host weakest precondition rule for host instantiation, that we will refer to as `wp_host_instantiate`.

Example The complete stack module is an instantiation declaration, which exports closures for push, pop, new_stack, is_empty, is_full, stack_length and map, as well as the function table

invoked by map. We recall that exports are passed via indices into the import variable store.

$$vm \triangleq \{0 \mapsto \mathsf{stack_module}\} \qquad host_program \triangleq ([], [\textbf{inst_decl}\ []\ 0\ [0,1,2,3,4,5,6,7])$$

The Iris-Wasm specification of the complete stack module from Section 5.3 is as follows (we elide the exporting of the table, for simplicity):

$$\exists \mathsf{stack_module}, \forall i\ js, (i \xhookrightarrow{\mathsf{mod}} \mathsf{stack_module}) \twoheadrightarrow \mathop{\circledast}_{j \in js} j \xmapsto{\mathsf{vis}} - \twoheadrightarrow$$
$$\mathsf{wp}_{\mathbf{HOST}}\ ([]; [\textbf{inst_decl}\ []\ \mathsf{i}\ js]) \left\{ \begin{array}{c} \exists cl_{\text{push}}\ cl_{\text{pop}}\ \ldots\ cl_{\text{map}}, \textbf{isStack}, \text{spec_push} * \text{spec_pop} \\ * \ldots * \text{spec_map} * \mathop{\circledast}_{j \in js} j \xmapsto{\mathsf{vis}} \mathsf{function_export}\ cl_j \end{array} \right\}$$

spec_push is the specification of the "push" method shown earlier. Likewise for the other specifications mentioned in the postcondition. Both the contents of the $stack module and the implementations of the stack operations are hidden from clients because of the existential quantifiers.

This stack module specification is proven by applying rule `wp_host_instantiate`, which populates the value import stores and gives ownership of all the resources necessary for the stack module operations, and then we apply the specifications for the stack operations shown in §5.2.4.

With this specification for the stack module and a similar one for the client module (parametrized by the specification of the stack), we verify the complete stack program (a sequence of instantiations) in our Coq formalization.

5.5 Logical Relation and Robust Safety Examples

One common application of an Iris program logic is the verification of *robust safety* examples. It should be noted that robust safety is not one single property that is stated like the traditional type safety property. Rather, it refers to any safety properties that can still be demonstrated even when interacting with unknown adverserial code. In an example later in this section, we'll demonstrate a stack-client example involving the interaction of the stack module, a client module, and an unknown adverserial module, where the desired invariants of the stack module can still be guaranteed due to Wasm's encapsulation at the module level – which illustrates the class of robust safety examples about module encapsulation that can be proven for Wasm modules.

Our goal is to leverage the coarse-grained encapsulation guarantees of WebAssembly modules to demonstrate robust safety of two scenarios involving some interaction between a known module and an unknown, potentially malicious, module. While the coarse-grained encapsulation properties granted by modules are relatively shallow (one module cannot interact with the internals of another unless explicitly shared), the reasoning principles are not: not only are we reasoning about unknown code, the desired robust safety property can be subtle, and highly specific to the particular implementation of a robustly safe module.

We emphasise that we do not seek to either define or prove module encapsulation as a meta-property. Rather, we define and apply a methodology to prove certain robust safety properties of specific example modules.

```
m_client ≜ (module ;; Another Stack Client
(import "adv" "f" (func $f (param i32) (result i32)))
(import "stack" "map" (func $map (param i32 i32)))
  ... ;; import global g and the remaining stack module
(elem (i32.const 0) $f) ;; populate table with imported function
(func $main (local $i i32)
  call $new_stack; ... ; const 4; call $push;
  local.get $i; const 2; call $push;
  local.get $i; const 0; call $map;
  local.get $i; call $stack_length; global.set $g))
```

```
stack_client ≜
inst_decl [] "stack" ["tab";...;"pop"]
inst_decl [] "adv" ["f"]
inst_decl ["f";"g";"tab";...;"pop"] "client" []
```

Figure 5.5: Robust safety example: applying map on an imported function

WebAssembly's modules are designed to allow trusted code to encapsulate its local state (e.g. variables and memory), by limiting what is shared with untrusted modules via imports and exports. This encapsulation is meant to hold no matter what other modules do, either by accident or by malice, and thus does not rely on compliance. Modules can take advantage of this encapsulation to guarantee various safety properties. To prove those properties formally, we may need to reason about the interaction between known, trusted code and unknown, untrusted code. We have thus far presented a program logic to reason about known code only. In this case study, we use the program logic to build a method to reason about the instantiation of unknown code, and use it to prove the *robust* safety of known code, that is, safety even when composed with adversarial code.

The methodology is based on a relational interpretation of WebAssembly types, built on top of our Iris-Wasm program logic, by defining logical relations for each WebAssembly type. The key idea is to interpret the types of primitives, functions, etc., all the way to module types, as propositions in Iris-Wasm. The methodology of defining logical relations in Iris is well known [59, 104, 52, 36], but here it is for the first time applied to the type system of a full industrial standard, namely the WebAssembly type system. We define semantic interpretations for all WebAssembly types. That includes all the internals of a module, and in particular it includes the types of exports and imports. We say that an import object is safe to share, or valid, if it is in the appropriate relation. All the results in this section have been formally proved in Coq. We give an overview here, and refer the reader to the accompanying Coq code for the full definition of the relational interpretation of WebAssembly types.

The interpretation of module types via the instance relation, denoted $\mathcal{I}[\![C]\!]$, is the keystone to derive specifications for unknown functions. The following key theorem states that the result of instantiating a well-typed module $\vdash m : \mathit{timps} \to \mathit{texps}$ produces a valid instance, given that all imports are valid according to *timps*.

Theorem 5.5.1 (Valid Instance Allocation). *If* $\vdash m : \mathit{timps} \to \mathit{texps}$*, and inst is the result of instan-*

tiating module m with imports imps, then

$$\mathsf{resources}(m, imps, timps, \cdots, inst) \twoheadrightarrow \mathsf{valid}[\![timps]\!](imps) \twoheadrightarrow \mathcal{I}[\![C]\!](inst)$$

where C is the module type, determined syntactically, $\mathsf{resources}(\cdots, inst)$ *corresponds to the ghost resources allocated by module instantiation as depicted by Lemma 5.4.1, and* $\mathsf{valid}[\![timps]\!](imps)$ *unfolds the list of imports, and applies the relevant relation on each import object.*

Proof. By unfolding the definition of module typing, inferring properties about the result of instantiating m, and component-wise proving the instance relation. Validity of imported types is established by the $\mathsf{valid}[\![timps]\!](imps)$ assumption, while the rest are established using the fundamental theorem of logical relations (FTLR). The FTLR, which roughly states that all well-typed programs are semantically well-typed, is a key non-trivial language property, and is proved by induction over the full type system. □

Applications of the Logical Relation Next we describe two scenarios, each involving our stack module interacting with some unknown function. In each case, the two modules interact via imported closures. We will therefore employ the closure relation $\mathcal{C}los$ as the principal logical relation in our reasoning.

The two applications highlight a conceptual distinction between two kinds of scenarios in which known code interacts with unknown code. In the first example, known code imports functions from an unknown module, and has a certain amount of control over how these are applied. The second example exports known code to an unknown module, and in that case, exported closures must carefully guard against misuse.

Figure 5.5 depicts a client of the stack module, which imports a closure "f" of type $[\mathbf{i32}] \rightarrow [\mathbf{i32}]$ from an unknown module. The client creates a new stack, pushes two values, then applies map using the imported unknown function, and finally computes the length of the stack by calling a function from the stack module. The stack module hides its internal representation from the context. Likewise, the host makes sure to hide the stack module operations from the unknown module. WebAssembly's coarse grained encapsulation thus guarantees that the integrity of the allocated stack is maintained, no matter what the unknown imported function does: as long as it does not trap, the final length operation succeeds and returns the original size of the stack, namely 2. We refer to imports and modules via names rather than indices, for the sake of readability. The following theorem expresses this example robust safety property for the stack and client module formally:

Theorem 5.5.2 (Top-level Host Specification). *If* $\vdash m_{\mathsf{adv}} : [] \; [\mathbf{func_e} \; ([\mathbf{i32}] \rightarrow [\mathbf{i32}])]$ *and the syntactic restrictions on* m_{adv} *hold, then*

$$\left\{\begin{array}{l} \text{"stack"} \xhookrightarrow{\mathsf{mod}} m_{\mathsf{stack}} * \text{"adv"} \xhookrightarrow{\mathsf{mod}} m_{\mathsf{adv}} * \\ \text{"client"} \xhookrightarrow{\mathsf{mod}} m_{\mathsf{client}} * \text{"g"} \xmapsto{\mathsf{vis}} \$g * \$g \xmapsto{\mathsf{wg}} - \twoheadrightarrow \\ [NaInv : \top] * \text{"f"}, \text{"tab1"}, \text{"map"}, \ldots, \text{"pop"} \xmapsto{\mathsf{vis}} - \end{array}\right\} \mathsf{stack_client} \left\{ hw, \begin{array}{l} (hw = ([];[]) \wedge \$g \xmapsto{\mathsf{wg}} 2) \vee \\ hw = (\mathbf{trap};[]) \end{array} \right\}$$

Proof. Once the host has allocated the unknown module, we apply Theorem 5.5.1 to conclude that

its instance is valid, which guarantees that each of its components, including the exported closure of type $[\mathbf{i32}] \to [\mathbf{i32}]$, is valid. As a result, we know that the unknown import of our client is in the closure relation $\mathscr{C}los$, which by definition of the relational interpretation includes a specification for the unknown function. Crucially, this specification does not depend on the stack internals, and thus we are able to prove that the stack size is maintained. □

Next we consider a scenario in which an unknown module imports operations from the stack module, namely new_stack, push and pop. The encapsulation of the stack module's internal state, alongside careful checks at the boundaries of each operation, which we will elaborate on below, should guarantee that the stack module memory indeed stores and maintains stacks, as defined by the **isStack** predicate, irrespectively of what the unknown module does. Henceforth we will refer to this as the representation invariant, denoted by $\mathsf{stackInvariant}(m)$, where m is the index of the encapsulated memory. Roughly, the representation invariant is an Iris (non-atomic) invariant containing a big separation of **isStack** predicates, one for each allocated stack.

The basic type system of WebAssembly guarantees that the adversary code does not get stuck. However, our goal is to reason about integrity of the data representation enforced by the module system. While the type system defines the typing of an individual module, it does not consider interweaving of module instantiations, since instantiation is handled by a host, typically written in untyped JavaScript. Therefore, the type system is too weak to capture the data abstraction enforced by the module system, which we are relying on here. As such, our interpretation of the type system does not capture the refined interpretation (with the representation invariant) of the stack module.

We use the standard type interpretation of the adversary module to reason about its execution. However, we want this interpretation to depend on the *refined representation invariant* of the stack module internals, rather than the *default interpretation granted by the logical relation*. Since each import must be valid when applying Theorem 5.5.1, we *manually* prove that, given the representation invariant, each exported function (new_stack, push, and pop) is in the closure relation.

As a result, we must now consider the case where a stack operation is applied on an arbitrary input value. Consider, for instance, push – it takes two arguments, one of which is a stack value, which is interpreted as a memory address. A malicious adversary could apply push to a masked stack value (a bogus memory address), thus breaking the expected internal behavior of the stack module. push must thus guard against such a situation by dynamically checking the validity of all safety-critical parameters. These dynamic checks ensure that no stack gets corrupted. Relying on those dynamic checks, we can then prove specifications that maintain the representation invariant:

Theorem 5.5.3 (Validity of Select Stack Module Operations). *If inst*.mems $= [m]$ *then,*

$$\begin{aligned}\mathsf{stackInvariant}(m) \to\ & \mathscr{C}los[\![[\mathbf{i32};\mathbf{i32}] \to []]\!](\{[\mathbf{i32};\mathbf{i32}] \to [], (inst, [\mathbf{i32}]), \mathsf{push}\}^{\mathsf{Native}}) \\ & *\mathscr{C}los[\![[\mathbf{i32}] \to [\mathbf{i32}]]\!](\{[\mathbf{i32}] \to [\mathbf{i32}], (inst, [\mathbf{i32}]), \mathsf{pop}\}^{\mathsf{Native}}) \\ & *\mathscr{C}los[\![[] \to [\mathbf{i32}]]\!](\{[] \to [\mathbf{i32}], (inst, [\mathbf{i32}]), \mathsf{new_stack}\}^{\mathsf{Native}})\end{aligned}$$

The representation invariant is allocated upon instantiation of the stack module, at which point

there are no allocated stacks. Theorem 5.5.3 is then applied on each of the relevant stack module exports, such that we can apply Theorem 5.5.1, and conclude with the standard type interpretation of the adversary module, while maintaining the now allocated representation invariant.

Finally, it should be noted that all the above robust safety examples require a host language implementation that respects Wasm's semantics and the module encapsulation, such as the one we introduced in Section 5.4. If the host language provides excessive permission to adverserial modules – such as arbitrary function invocation by absolute store addresses – then the robust safety examples cannot be proven. The proofs will fail at the point where the logical relation is constructed for the host language.

5.6 Mechanisation Summary

We implement and prove the Iris-Wasm proof rules in this chapter in the Iris framework in the Coq proof assistant. Iris was originally developed to reason about programs with complex concurrency; however, the same mechanisms have proven useful to reason about complex sequential programs such as the awkward example, as demonstrated for example by Georges et al. [36]. In this chapter, we focus our presentation on the novel, language-specific proof rules we introduce and prove, but our program logic also inherits many other logical constructs and proof rules from Iris which we make use of in our development. We have already mentioned the 'later' modality, $\triangleright$, which avoids circularities in the presence of the higher-order features of Iris, and which can be used to define guarded recursive predicates in Iris, as well as the 'persistence' $\square$ and 'update' $\Rrightarrow$ modalities. Other features we use include the frame rule, non-atomic invariants, ghost state, and other proof rules like Löb induction; for a thorough introduction to those, see Jung et al. [53].

We prove all our proof rules in Iris, with respect to the default definition of the weakest precondition predicate (with an extra requirement that the frame resource holds for every step of reduction) instantiated to refer to the Coq formalization of the official WebAssembly 1.0 operational semantics by Watt et al. [114].

The adequacy theorem of Iris [53, §6.4] then yields the final desired soundness theorem, which intuitively says that if a weakest precondition for a WebAssembly or host program has been proved in Iris-Wasm, then it does indeed mean that the program runs safely, according to the official WebAssembly 1.0 operational semantics, or the host language that embeds it. An example of the latter can be found in the Coq mechanization.

helpers	language	rules	instantiation	host	examples	logrel	stack	**total**
11836	3685	7123	6828	2339	2754	8145	8787	51497

Figure 5.6: Lines of code of the Iris development, as given by cloc

The size of the full Iris development is summarized in Fig. 5.6. The logrel folder contains a case study presented in the next section, and stack contains the full stack module and associated clients.

The stack module, with a binary size of 637 bytes, is defined in around 200 lines code in Coq, with the module type checking done in 300 lines of code using the type checker from Watt et al.

[114]. The module specification is fully verified using the Iris-Wasm logic in around 3800 lines of code in Coq, where 2100 lines are used to verify each of the module function specifications, and the remaining code is used to prove the top-level instantiation specification and auxiliary lemmas. Such a ratio between program and proof size may hint at a substantial verification effort. However, it's important to note that it reflects a version of Iris-Wasm without a bespoke proof mode; an interesting line of future work is to extend Iris-Wasm with various automation techniques, such as the proof search strategy of Mulder, Krebbers, and Geuvers [77], and use it to prove specifications of large real-world programs.

5.7 Related Work

Watt et al. [113] develop a mechanised first-order separation logic for what they call "encapsulated" WebAssembly, that is, code limited to a single module, with no exports or imports, and no uses of the **call_indirect** instruction or the host, and they do not handle instantiation. For their subset of the language, our proof rules are similar up to presentational details, except for the handling of breaks, where, as mentioned in Section 5.2.3, we use a novel approach with a bind rule which scales to higher-order programs, unlike the approach taken by Watt et al. [113].

WebAssembly provides coarse-grained memory safety, at the boundary of memory objects, and coarse-grained isolation, at the boundary of modules. Lehmann, Kinder, and Pradel [65] show that many of the classical attacks against memory unsafe languages, targeting a finer granularity, also work against Wasm programs that not specifically written to take advantage of module isolation. We show in our examples that, when Wasm programs are written with module isolation in mind, the language specification does indeed enforce expected isolation guarantees.

Recall that Iris-Wasm has to use a slightly modified version of the `wp` predicate due to the frame resource required for the frame switching rules. This makes the `wp` definition a bit more complicated, and also renders the proof rules more verbose. Wagner proposed a different solution to this problem by designing new *pop* and *push* modalities for the frame [9]. This method allows the frames to be properly encapsulated, rendering only the local function call frame accessible. However, this method has not been mechanised in Iris yet. It is therefore currently unknown whether there would be similar challenges in mechanising these modalities as we encountered in proving our Iris-Wasm proof rules.

In Iris-Wasm, we exploit the higher-order features of Iris. In particular, we use that weakest preconditions can be nested inside other weakest preconditions to specify higher-order functions and WebAssembly control structures, and quantification over predicates to get abstract modular specifications of modules. These features of Iris stem from earlier work on higher-order separation logic, starting with Biering, Birkedal, and Torp-Smith [13]. To show robust safety, we further rely on Iris' invariants, the idea of which can be traced back at least to recursively-defined Kripke models of type systems [3] and separation logics with hidden state [102].

MSWasm [74, 27] (Memory-Safe Wasm) is a proposed extension of WebAssembly that adds first-class support for CHERI-like [110] fine-grained runtime-checked memory capabilities. The log-

ical relation of Cerise [36, 37, 38, 35], mechanised in Iris, captures encapsulation for hardware capabilities in an idealized assembly model and may be used as a starting point to formalize the guarantees of MSWasm on top of Iris-Wasm. Indeed, Iris-MSWasm [64] is an extension on top of the Iris-Wasm program logic that extends WasmCert-Coq[114] to include the definitions of MSWasm[74] and a larger program logic covering the language constructs of MSWasm as well.

RichWasm [32] design a typed intermediate language built on top of Wasm which use an extended type system with capabilities to provide a fine-grained encapsulation for shared memories.

Kolosick et al. [58] use a logical relation to show that WebAssembly programs naturally compile to unsafe platform assembly in such a way that the compiled code obeys a safe calling convention and certain isolation properties with respect to the rest of the system. Narayan et al. [80] rely on this result to implement a sandboxing technique whereby C code is first compiled to WebAssembly which is then ultimately compiled to native assembly for linking. They use this technique to sandbox a number of Firefox libraries.

Many related works deal with the mechanised formalization of low-level languages. RockSalt [75] is a verified checker that validates code binaries against a sandbox policy, similar to that of Google's Native Client (NaCl). RockSalt is mechanically verified using a formalization of a subset of x86 in Coq. Kennedy et al. [54] use Coq to build a macro assembler for x86, while relating machine code to separation logic formulas suitable for program verification.

The Certified Assembly Programming (CAP) family of frameworks [81, 121, 120, 30] support the definition of second-order Hoare logics for verifying modular specifications of low-level assembly programs, using expressive features such as embedded code pointers, concurrency, and dynamic thread creation. As such, CAP focuses on features that are abstracted away by Wasm. Gu et al. [41] presents CertiKOS, an extensible architecture for certifying concurrent OS kernels. Using CertiKOS, Gu et al. [41, 42] develop and verify a concurrent OS kernel consisting of both C and x86 assembly code. By leveraging CompCertX [40], CertiKOS is able to reason about interactions between C and x86 assembly. As is the case with Iris-Wasm, the setup assumes that the two languages share the same memory model. The recent DimSum [99] framework supports reasoning about multilingual programs between languages with different memory models. However, while Iris-Wasm focuses on mechanizing the full language of a real industrial standard, the DimSum approach has only been applied to a simple high-level imperative language and an idealized assembly language so far.

5.8 Acknowledgements

This chapter draws texts from a previously published paper *Iris-Wasm: Robust and Modular Verification of WebAssembly Programs* (PLDI 2023) [91], authored by myself, Aïna Linn Georges[7], Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner and Lars Birkedal. In particular, Section 5.5 was adapted from the part of paper which was originally written by Aïna Linn Georges, who has also written an extended version in her thesis [34].

[7] Aïna Linn Georges and myself were joint-first authors of this paper.

Jean Pichon-Pharabod and Lars Birkedal were senior collaborators at Aarhus University working with Aïna Linn Georges and Maxime Legoupil[8]. Conrad Watt and Philippa Gardner were senior collaborators at University of Cambridge and Imperial College respectively, working with myself. The mechanisation itself was a joint-effort by myself, Aïna Linn Georges, and Maxime Legoupil, guided by the senior collaborators.

Regarding the mechanisation development, I implemented the instantiation of WasmCert-Coq semantics into Iris, the state interpretation for Wasm memory model with the necessary auxiliary results, and a large part of the first iteration of the proof rules, including those of the minimal host language described in Section 5.4, while having regular meetings with the collaborators. I also designed and developed the semantic characterisation result for module instantiation. Aïna Linn Georges and Maxime Legoupil from Aarhus University joined the developments of the proof rules later on. In particular, they developed variants of the bind and frame context rules which simplified the later developments of the stack example, and implemented a large part of the development related to the host-Wasm interoperation, including a tedious part related to extending the logical values to include the host call value, which required substantial effort in augmenting the *lh* context.

Besides above, Aïna Linn Georges designed and developed the codebase related to the logical relation for both Wasm and the host language, utilising her experience from her prior work [38, 35]. She also developed some examples using the Iris-Wasm proof rules including a store-recursion example known as Landin's Knot, among other examples.

Maxime Legoupil led the development of the robust safety stack example, with Aïna Linn Georges and myself also contributing later, adding a stricter version of the module with bound checks in the final iteration of the example. He also implemented a determinacy proof for a large part of Wasm 1.0's operational semantics, which is a substantial auxiliary result used in the codebase as described in Section 5.2.

[8]Maxime Legoupil was a master student affiliated with École Normale Supérieure France at the time. For simplicity, I categorise Maxime Legoupil together with Aarhus University in this acknowledgement, since he was working technically with Aïna Linn Georges in Lars Birkedal's group for a research internship at Aarhus University during the work.

Chapter 6

Conclusion and Future Work

WebAssembly has undergone significant growth since its initial release in 2017. Originally designed as a compilation target for web browsers, Wasm is now also utilized in non-web environments, such as JavaScript virtual machines and various sandbox environments. The Wasm 2.0 release in 2022 introduced several feature extensions, with additional major features, including garbage collection, scheduled for future releases. To address the challenges of maintaining mechanisations, our efforts have focused on streamlining the process, enabling the WasmCert-Coq mechanisation to remain largely aligned with the Wasm 2.0 standard. In the process, we identified and resolved several errors in the specification. Our work on progressful interpreters has also facilitated deeper theoretical exploration of the relationship between verified executable semantics and type soundness properties. Additionally, the Iris-Wasm program logic, a practical application of our mechanisation, has demonstrated its utility by formalizing and proving advanced security guarantees. Notably, it has demonstrated the module encapsulation property through verified examples – a key feature of Wasm that was previously only described informally in the specification.

Despite the successes above so far, there are still many future works to be done following the work discussed in this thesis. I bring up the most prominent ones in the following discussion:

- While my efforts have successfully kept the WasmCert-Coq mechanisation largely up to date with the Wasm 2.0 standard, maintaining synchronisation with the evolving specification remains a demanding task in the future. Without a more efficient solution, there is a significant risk of falling behind in future updates, which would impede downstream applications that rely on the mechanisation.

  One solution to the above is to introduce a new domain domain specific language (DSL) for writing specification for WebAssembly. The specification written this way should be viewed as a single source of truth for the WebAssembly specification, which all the other versions can be generated by individual *backends*. This includes the current Latex and prose specification which is human readable, and potentially the mechanisation definitions and a generated interpreter as well.

  The SpecTec DSL language [119] has been designed for this exact purpose, which aims at

providing the DSL and a toolchain that facilitates both the Wasm specification and the generation of related artifacts necessary to standardise new features. SpecTec contains an animation language (AL) that generates an interpreter from the DSL-defined semantics which passes all of Wasm's test suite. However, the animation language includes a considerable amount of manually coded parts. There has in fact been a mistake in a reduction rule related to frame propagation in the semantics, which the animation language did not use as it implements its own call stack. As a result, an interpreter generated from the mechanisation – and hence entirely from the reduction rules – is still important to verify the integrity of the operational semantics.

The WebAssembly Community Group has voted to adopt SpecTec as the tool for authoring future editions of the Wasm specification in 2025 [103]. An automated pass from the SpecTec DSL to the mechanisation is the next goal. In 2023, when the SpecTec DSL itself was less mature, I made an initial attempt at using the DSL to produce a Coq mechanisation of WebAssembly from the DSL's intermediate langauge (IL), which borrowed the designs from my collaborator Joachim Breitner's attempt of a similar pass that aims at producing a Lean mechanisation. This version produced only a prototype of the mechanisation including the datatypes and a very limited subset of the operational semantics and type system.

This effort was later extended to a major subset of the Wasm DSL 1.0 operational semantics and type system excluding the numerics and module instantiation, with a major part of the extension implemented by Diego Cupello in his Master's Project under my technical supervision [24], who has also proved the type preservation property as part of his project. However, the current version of the generated specification is still much more difficult and verbose to be used by human users, and do not include certain important parts of the specification, such as the numerics or the parsing stage. As a result, the artifacts produced in these early attempts remain far from matching the scope and quality of the manually developed mechanisations. No executable artifact is generated either from the mechanisation definition. Significant efforts are still required in this direction, including theoretic studies on converting the inherently dependently-typed SpecTec DSL to theorem provers – not limited to Coq – but also others such as Isabelle/HOL that do not support dependent types.

- Following the above point, it should be noted that standardisation is the last phase of process for a feature to be introduced into Wasm. As a result, although the generated mechanisations help provide correctness guarantees of the underlying DSL specification, it does not help the specification authors for these new features, since the specifications in the DSL have to be written before the theorem prover codes can be generated via the relevant backends.

  In my view, it is possible to bring the usefulness of mechanisation further upstream to help the specification authors as well. This does not contrast the current direction, but can in fact share some designs: a carefully designed, well-rendered theorem prover specification lays foundation for the verification research community; whereas a hasty, human-unfriendly version can still be used to connect to some other tools in theorem provers to catch errors in the specification as they are being written. It could be possible to connect a lightweight

version generation of the Coq inductive definitions to QuickChick [88, 62], which is a randomised property-based testing plugin for Coq that can test certain desired properties by automatically deriving generators from inductive semantics. If a pass in this direction can be implemented automatically, specification authors can then be noticed of errors in their specification drafts much earlier in the line of work. This can potentially help improve the efficiency of the specification authors for the new feature proposals.

- The Iris-Wasm program logic also has much future improvement to be done in its automation. As discussed in the example displayed in Section 5.2, despite Iris-Wasm having a strong expressive power thanks to the higher-order Iris framework, the Coq mechanisation is still rather tedious to be used, which has also affected the downstream works that follow it. A specialised *proof mode* for Iris-Wasm that helps the automation process, including resource managements and proof rule applications, would greatly help the feasibility of using Iris-Wasm to verify larger real-world programs.

  It can also be interesting to consider connecting the generation of Iris-Wasm proof rules to the SpecTec DSL. This could be possible for Wasm, since most of the Iris-Wasm proof rules are relatively straightforward, where the proof rule for each instruction has a correspondence to the respective operational semantic rules. The more complicated rules – mostly including the control-flow rules and the contextual bind rules – can still be stated and proved manually. The main benefit of this connection is to allow the verification researchers working on Iris to enjoy a better maintained and more trusted mechanisation foundation, instead of having to work on a fork from a historical version of the codebase.

Looking ahead, I envision a future where all new Wasm features are accompanied by automatically generated mechanised models, which can be rapidly verified with minimal effort. Achieving this would drastically reduce the burden of keeping the mechanisation in pace with the evolving specification. The specification authors of the feature extensions can also benefit from some early checks provided by the SpecTec DSL, therefore easing the standardisation process. My work has paved the way toward this vision by demonstrating how automation and principled design can make full-scale mechanisation of Wasm both feasible and maintainable.

Bibliography

[1] Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. "Martin-Löf à la Coq". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2024. London, UK: Association for Computing Machinery, 2024, pp. 230–245. ISBN: 9798400704888. DOI: 10.1145/3636501.3636951. URL: https://doi.org/10.1145/3636501.3636951.

[2] Amal Ahmed. *Semantics of Types for Mutable State*. 2004. URL: https://www.ccs.neu.edu/home/amal/ahmedthesis.pdf.

[3] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. "State-dependent representation independence". In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. Ed. by Zhong Shao and Benjamin C. Pierce. ACM, 2009, pp. 340–353. DOI: 10.1145/1480881.1480925. URL: https://doi.org/10.1145/1480881.1480925.

[4] Philip Wadler et al. *The expression problem*. 1998. URL: http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

[5] Guillaume Allais. "Agdarsec - total parser combinators". English. In: Publisher Copyright: © JFLA 2018 - Journees Francophones des Langages Applicatifs. All rights reserved. Sylvie Boldo, Nicolas Magaud. Journées Francophones des Langages Applicatifs 2018. Sylvie Boldo; Nicolas Magaud. Journées Francophones des Langages Applicatifs 2018, Jan 2018, Banyuls-sur-Mer, France. publié par les auteurs, 2018. ⟨hal-01707376⟩; Vingt-neuviemes Journees Francophones des Langages Applicatifs, JFLA 2018 - 29th French-Speaking Conference on Applicative Languages, JFLA 2018 ; Conference date: 24-01-2018 Through 27-01-2018. Feb. 2018, pp. 45–59.

[6] Guillaume Allais. *GitHub - Total Parser Combinators in Coq*. 2017.

[7] Bytecode Alliance. *GitHub - bytecodealliance/wasmtime: A fast and secure runtime for WebAssembly*. [Accessed 01-07-2024].

[8] Nada Amin and Tiark Rompf. "Type soundness proofs with definitional interpreters". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL '17. Paris, France: Association for Computing Machinery, 2017, pp. 666–679. ISBN: 9781450346603. DOI: 10.1145/3009837.3009866. URL: https://doi.org/10.1145/3009837.3009866.

[9] *An Adjoint Separation Logic for the Wasm Call Stack (Talk)*. [Accessed 15-07-2025]. 2025. URL: https://www.andrewwagner.io/assets/slides/adj-wasm-dagstuhl.pdf.

[10] *An extraordinarily optimizable, low-level subset of JavaScript*. [Accessed 11-07-2025]. URL: http://asmjs.org/.

[11] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. "Intrinsically-typed definitional interpreters for imperative languages". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158104. URL: https://doi.org/10.1145/3158104.

[12] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004. ISBN: 3540208542.

[13] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. "BI-hyperdoctrines, higher-order separation logic, and abstraction". In: *ACM Trans. Program. Lang. Syst.* 29.5 (2007), p. 24. DOI: 10.1145/1275497.1275499. URL: https://doi.org/10.1145/1275497.1275499.

[14] Lars Birkedal and Aleš Bizjak. *Lecture Notes on Iris: Higher-Order Concurrent Separation Logic*. Tech. rep. Aarhus University, 2017.

[15] Sandrine Blazy and Xavier Leroy. "Mechanized semantics for the Clight subset of the C language". In: *Journal of Automated Reasoning* 43.3 (2009), pp. 263–288.

[16] Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. "A trusted mechanised JavaScript specification". In: *SIGPLAN Not.* 49.1 (Jan. 2014), pp. 87–100. ISSN: 0362-1340. DOI: 10.1145/2578855.2535876. URL: https://doi.org/10.1145/2578855.2535876.

[17] Denis Bogdanas and Grigore Roşu. "K-Java: A Complete Semantics of Java". In: *SIGPLAN Not.* 50.1 (Jan. 2015), pp. 445–456. ISSN: 0362-1340. DOI: 10.1145/2775051.2676982. URL: https://doi.org/10.1145/2775051.2676982.

[18] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. "Verified Compilation of Floating-Point Computations". In: *Journal of Automated Reasoning* 54.2 (2015), pp. 135–163. URL: http://xavierleroy.org/publi/floating-point-compcert.pdf.

[19] Sylvie Boldo and Guillaume Melquiond. "Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq". In: *2011 IEEE 20th Symposium on Computer Arithmetic*. 2011, pp. 243–252. DOI: 10.1109/ARITH.2011.40.

[20] Ana Bove, Peter Dybjer, and Ulf Norell. "A Brief Overview of Agda – A Functional Language with Dependent Types". In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 73–78. ISBN: 978-3-642-03359-9.

[21] James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. "System F in Agda, for Fun and Profit". In: *Mathematics of Program Construction: 13th International Conference, MPC 2019, Porto, Portugal, October 7–9, 2019, Proceedings*. Porto, Portugal: Springer-Verlag, 2019, pp. 255–297. ISBN: 978-3-030-33635-6. DOI: 10.1007/978-3-030-33636-3_10. URL: https://doi.org/10.1007/978-3-030-33636-3_10.

[22] Adam Chlipala. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2022.

[23] Sylvain Conchon and Jean-Christophe Filliâtre. "A persistent union-find data structure". In: Oct. 2007, pp. 37–46. DOI: 10.1145/1292535.1292541.

[24] Diego Cupello. "IL2Coq: Automatic Translation of WebAssembly Specification to Coq". MA thesis. Imperial College London, 2024. URL: https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/2324-ug-projects/Cupello,-Diego-final_report-IL2Coq-Automatic-Translation-of.pdf.

[25] Nils Anders Danielsson. "Operational semantics using the partiality monad". In: *SIGPLAN Not.* 47.9 (Sept. 2012), pp. 127–138. ISSN: 0362-1340. DOI: 10.1145/2398856.2364546. URL: https://doi.org/10.1145/2398856.2364546.

[26] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. "Meta-theory à la carte". In: *SIGPLAN Not.* 48.1 (Jan. 2013), pp. 207–218. ISSN: 0362-1340. DOI: 10.1145/2480359.2429094. URL: https://doi.org/10.1145/2480359.2429094.

[27] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. "Position Paper: Progressive Memory Safety for WebAssembly". In: *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '19.

Phoenix, AZ, USA: Association for Computing Machinery, 2019. ISBN: 9781450372268. DOI: 10.1145/3337167.3337171. URL: https://doi.org/10.1145/3337167.3337171.

[28] Daniel Ehrenberg. *WebAssembly JavaScript Interface W3C Recommendation*. Tech. rep. W3C, Dec. 2019. URL: https://www.w3.org/TR/wasm-js-api-1/.

[29] Chucky Ellison and Grigore Rosu. "An executable formal semantics of C with applications". In: *SIGPLAN Not.* 47.1 (Jan. 2012), pp. 533–544. ISSN: 0362-1340. DOI: 10.1145/2103621.2103719. URL: https://doi.org/10.1145/2103621.2103719.

[30] Xinyu Feng and Zhong Shao. "Modular verification of concurrent assembly code with dynamic thread creation and termination". In: *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*. Ed. by Olivier Danvy and Benjamin C. Pierce. ACM, 2005, pp. 254–267. DOI: 10.1145/1086365.1086399. URL: https://doi.org/10.1145/1086365.1086399.

[31] Daniele Filaretti and Sergio Maffeis. "An Executable Formal Semantics of PHP". In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by Richard Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 567–592. ISBN: 978-3-662-44202-9.

[32] Michael Fitzgibbons, Zoe Paraskevopoulou, Noble Mushtak, Michelle Thalakottur, Jose Sulaiman Manzur, and Amal Ahmed. "RichWasm: Bringing Safe, Fine-Grained, Shared-Memory Interoperability Down to WebAssembly". In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). DOI: 10.1145/3656444. URL: https://doi.org/10.1145/3656444.

[33] José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. "JaVerT 2.0: Compositional Symbolic Execution for JavaScript". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290379. URL: https://doi.org/10.1145/3290379.

[34] Aïna Linn Georges. "Designing and Proving Robust Safety of Efficient Capability Machine Programs". PhD thesis. 2023. URL: https://iris-project.org/pdfs/2023-phd-georges.pdf.

[35] Aïna Linn Georges, Armaël Guéneau, Thomas van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. *Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code*. Tech. rep. Aarhus University, 2022. URL: https://cs.au.dk/~birke/papers/cerise.pdf.

[36] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. "Efficient and provable local capability revocation using uninitialized capabilities". In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–30. URL: https://doi.org/10.1145/3434287.

[37] Aïna Linn Georges, Armaël Guéneau, Thomas Van-Strydonck, Amin Timany, Dominique Trieu Alix Devriese, and Lars Birkedal. "Cap' ou pas cap' ?: Preuve de programmes pour une machine à capacités en présence de code inconnu". In: *Journées Francophones des Langages Applicatifs 2021*. Apr. 2021. URL: https://cris.vub.be/ws/portalfiles/portal/55081793/paper.pdf.

[38] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. *Le Temps des Cerises: Efficient Temporal Stack Safety on Capability Machines using Directed Capabilities*. Tech. rep. Aarhus University, 2022. URL: https://cs.au.dk/~ageorges/publications_pdfs/monotone-technical.pdf.

[39] W3C WebAssembly Community Group. *GitHub - WebAssembly/spec: WebAssembly specification, reference interpreter, and test suite.* [Accessed 09-07-2024].

[40] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. "Deep Specifications and Certified Abstraction Layers". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K.

Rajamani and David Walker. ACM, 2015, pp. 595–608. DOI: 10.1145/2676726.2676975. URL: https://doi.org/10.1145/2676726.2676975.

[41] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels". In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, 2016, pp. 653–669. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

[42] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. "Certified concurrent abstraction layers". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by Jeffrey S. Foster and Dan Grossman. ACM, 2018, pp. 646–661. DOI: 10.1145/3192366.3192381. URL: https://doi.org/10.1145/3192366.3192381.

[43] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. "The Essence of JavaScript". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 126–150. ISBN: 9783642141072. DOI: 10.1007/978-3-642-14107-2_7. URL: http://dx.doi.org/10.1007/978-3-642-14107-2_7.

[44] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. "Bringing the web up to speed with WebAssembly". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 185–200. ISBN: 9781450349888. DOI: 10.1145/3062341.3062363. URL: https://doi.org/10.1145/3062341.3062363.

[45] Pat Hickey. *How Fastly and the developer community are investing in the WebAssembly ecosystem*. May 2020. URL: https://www.fastly.com/blog/how-fastly-and-developer-community-invest-in-webassembly-ecosystem.

[46] Xuan Huang. "A Mechanized Formalization of the WebAssembly Specification in Coq". In: *RIT Computer Science*. 2019.

[47] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[48] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. "Featherweight Java: a minimal core calculus for Java and GJ". In: *ACM Trans. Program. Lang. Syst.* 23.3 (May 2001), pp. 396–450. ISSN: 0164-0925. DOI: 10.1145/503502.503505. URL: https://doi.org/10.1145/503502.503505.

[49] *Implementation of Persistent Array in Coq's Kernel*. [Accessed 20-06-2025].

[50] Koen Jacobs, Dominique Devriese, and Amin Timany. "Purity of an ST monad: full abstraction by semantically typed back-translation". In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (2022), pp. 1–27. DOI: 10.1145/3527326. URL: https://doi.org/10.1145/3527326.

[51] Ende Jin, Nada Amin, and Yizhou Zhang. "Extensible Metatheory Mechanization via Family Polymorphism". In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591286. URL: https://doi.org/10.1145/3591286.

[52] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: securing the foundations of the Rust programming language". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 66:1–66:34. DOI: 10.1145/3158154. URL: https://doi.org/10.1145/3158154.

[53] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *J. Funct. Program.* 28 (2018), e20. DOI: 10.1017/S0956796818000151.

[54] Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Évariste Dagand. "Coq: the world's best macro assembler?" In: *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*. Ed. by Ricardo Peña and Tom Schrijvers. ACM, 2013, pp. 13–24. DOI: 10.1145/2505879.2505897. URL: https://doi.org/10.1145/2505879.2505897.

[55] Steven Keuchel and Tom Schrijvers. "Generic datatypes à la carte". In: *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*. WGP '13. Boston, Massachusetts, USA: Association for Computing Machinery, 2013, pp. 13–24. ISBN: 9781450323895. DOI: 10.1145/2502488.2502491. URL: https://doi.org/10.1145/2502488.2502491.

[56] 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 (TOPLAS)* 28.4 (2006), pp. 619–695.

[57] Wen Kokke, Jeremy G. Siek, and Philip Wadler. "Programming language foundations in Agda". In: *Science of Computer Programming* 194 (2020), p. 102440. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2020.102440. URL: https://www.sciencedirect.com/science/article/pii/S0167642320300502.

[58] Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael Lemay, Deepak Garg, Ranjit Jhala, and Deian Stefan. "Isolation Without Taxation: Near-Zero-Cost Transitions for WebAssembly and SFI". In: *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, Jan. 2022. DOI: 10.1145/3498688. URL: https://doi.org/10.1145/3498688.

[59] Robbert Krebbers, Amin Timany, and Lars Birkedal. "Interactive proofs in higher-order concurrent separation logic". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 205–217. DOI: 10.1145/3009837.3009855. URL: https://doi.org/10.1145/3009837.3009855.

[60] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. "CakeML: a verified implementation of ML". In: *ACM SIGPLAN Notices* 49.1 (2014), pp. 179–191.

[61] Peter Lammich. "Refinement to Imperative HOL". In: *Journal of Automated Reasoning* 62.4 (2019), pp. 481–503. DOI: 10.1007/s10817-017-9437-1. URL: https://link.springer.com/article/10.1007/s10817-017-9437-1.

[62] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. "Generating good generators for inductive relations". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158133. URL: https://doi.org/10.1145/3158133.

[63] Daniel K. Lee, Karl Crary, and Robert Harper. "Towards a Mechanized Metatheory of Standard ML". In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '07. Nice, France: Association for Computing Machinery, 2007, pp. 173–184. ISBN: 1595935754. DOI: 10.1145/1190216.1190245. URL: https://doi.org/10.1145/1190216.1190245.

[64] Maxime Legoupil, June Rousseau, Aïna Linn Georges, Jean Pichon-Pharabod, and Lars Birkedal. "Iris-MSWasm: Elucidating and Mechanising the Security Invariants of Memory-Safe WebAssembly". In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (Oct. 2024). DOI: 10.1145/3689722. URL: https://doi.org/10.1145/3689722.

[65] Daniel Lehmann, Johannes Kinder, and Michael Pradel. "Everything Old is New Again: Binary Security of WebAssembly". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 217–234. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann.

[66] Xavier Leroy. "Formal verification of a realistic compiler". In: *Commun. ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: https://doi.org/10.1145/1538788.1538814.

[67] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. "CompCert-a formally verified optimizing compiler". In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. 2016.

[68] Pierre Letouzey. "A new extraction for Coq". In: *International Workshop on Types for Proofs and Programs*. Springer. 2002, pp. 200–219.

[69] Pierre Letouzey. "Extraction in coq: An overview". In: *Logic and Theory of Algorithms: 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008 Proceedings 4*. Springer. 2008, pp. 359–369.

[70] Sheng Liang and Paul Hudak. "Modular denotational semantics for compiler construction". In: *European Symposium on Programming*. Springer. 1996, pp. 219–234.

[71] Wolfgang Meier, Martin Jensen, Jean Pichon-Pharabod, and Bas Spitters. "CertiCoq-Wasm: A Verified WebAssembly Backend for CertiCoq". In: *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP '25. Denver, CO, USA: Association for Computing Machinery, 2025, pp. 127–139. ISBN: 9798400713477. DOI: 10.1145/3703595.3705879. URL: https://doi.org/10.1145/3703595.3705879.

[72] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. "Into the Depths of C: Elaborating the de Facto Standards". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pp. 1–15. ISBN: 9781450342612. DOI: 10.1145/2908080.2908081. URL: https://doi.org/10.1145/2908080.2908081.

[73] *Memory64 Proposal for WebAssembly*. [Accessed 28-11-2024].

[74] Alexandra E. Michael, Anitha Gollamudi, Jay Bosamiya, Evan Johnson, Aidan Denlinger, Craig Disselkoen, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, and Deian Stefan. "MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code". In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 425–454. DOI: 10.1145/3571208. URL: https://doi.org/10.1145/3571208.

[75] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. "RockSalt: better, faster, stronger SFI for the x86". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. Ed. by Jan Vitek, Haibo Lin, and Frank Tip. ACM, 2012, pp. 395–404. DOI: 10.1145/2254064.2254111. URL: https://doi.org/10.1145/2254064.2254111.

[76] Peter D Mosses. "Modular structural operational semantics". In: *The Journal of Logic and Algebraic Programming* 60 (2004), pp. 195–228.

[77] Ike Mulder, Robbert Krebbers, and Herman Geuvers. "Diaframe: automated verification of fine-grained concurrent programs in Iris". In: *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. Ed. by Ranjit Jhala and Isil Dillig. ACM, 2022, pp. 809–824. DOI: 10.1145/3519939.3523432. URL: https://doi.org/10.1145/3519939.3523432.

[78] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. "Lem: Reusable Engineering of Real-world Semantics". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: ACM, Sept. 2014, pp. 175–188. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628143.

[79] *Multi Memory Proposal for WebAssembly*. [Accessed 16-06-2025].

[80] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. "Retrofitting Fine Grain Isolation in the Firefox Renderer". In: *Proceedings of the 29th USENIX Conference on Security Symposium*. USA: USENIX Association, 2020. ISBN: 978-1-939133-17-5.

[81] Zhaozhong Ni and Zhong Shao. "Certified Assembly Programming with Embedded Code Pointers". In: *SIGPLAN Not.* 41.1 (Jan. 2006), pp. 320–333. ISSN: 0362-1340. DOI: 10.1145/1111320.1111066. URL: https://doi.org/10.1145/1111320.1111066.

[82] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN: 3540433767.

[83] *Non-trapping Float-to-int Conversions*. [Accessed 01-11-2024].

[84] Michael Norrish. *C formalised in HOL*. Tech. rep. 1998.

[85] *Official Test Suite for WebAssembly*. [Accessed 17-06-2025].

[86] Scott Owens. "A sound semantics for OCaml light". In: *European Symposium on Programming*. Springer. 2008, pp. 1–15.

[87] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. "Functional Big-Step Semantics". In: *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 589–615. ISBN: 9783662494974. DOI: 10.1007/978-3-662-49498-1_23. URL: https://doi.org/10.1007/978-3-662-49498-1_23.

[88] Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. "Computing correctly with inductive relations". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 966–980. ISBN: 9781450392655. DOI: 10.1145/3519939.3523707. URL: https://doi.org/10.1145/3519939.3523707.

[89] Daejun Park, Andrei Stefănescu, and Grigore Roşu. "KJS: A complete formal semantics of JavaScript". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2015, pp. 346–356.

[90] Frank Pfenning and Carsten Schürmann. "System Description: Twelf — A Meta-Logical Framework for Deductive Systems". In: *Automated Deduction — CADE-16*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 202–206. ISBN: 978-3-540-48660-2.

[91] Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. "Iris-Wasm: Robust and Modular Verification of WebAssembly Programs". In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591265. URL: https://doi.org/10.1145/3591265.

[92] Xiaojia Rao, Stefan Radziuk, Conrad Watt, and Philippa Gardner. "Progressful Interpreters for Efficient WebAssembly Mechanisation". In: *Proc. ACM Program. Lang.* 9.POPL (2025). DOI: 10.1145/3704858. URL: https://doi.org/10.1145/3704858.

[93] W3C Recommendation. *WebAssembly Core Specification 1.0*. [Accessed 11-11-2024]. 2019.

[94] Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser, and Peter Mosses. "Intrinsically-typed definitional interpreters à la carte". In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (Oct. 2022). DOI: 10.1145/3563355. URL: https://doi.org/10.1145/3563355.

[95] John C. Reynolds. "Definitional interpreters for higher-order programming languages". In: *Proceedings of the ACM Annual Conference - Volume 2*. ACM '72. Boston, Massachusetts, USA: Association for Computing Machinery, 1972, pp. 717–740. ISBN: 9781450374927. DOI: 10.1145/800194.805852. URL: https://doi.org/10.1145/800194.805852.

[96] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. "QED at Large: A Survey of Engineering of Formally Verified Software". In: *Found. Trends Program. Lang.* 5.2-

3 (2019), pp. 102–281. DOI: 10.1561/2500000045. URL: https://doi.org/10.1561/2500000045.

[97] Tiark Rompf and Nada Amin. *From F to DOT: Type Soundness Proofs with Definitional Interpreters*. 2016. arXiv: 1510.05216 [cs.PL]. URL: https://arxiv.org/abs/1510.05216.

[98] Grigore Roșu and Traian Florin Șerbănută. "An overview of the K semantic framework". In: *The Journal of Logic and Algebraic Programming* 79.6 (2010), pp. 397–434.

[99] Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. "DimSum: A Decentralized Approach to Multi-language Semantics and Verification". In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 775–805. DOI: 10.1145/3571220. URL: https://doi.org/10.1145/3571220.

[100] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. "Symbolic Execution for JavaScript". In: *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*. PPDP '18. Frankfurt am Main, Germany: Association for Computing Machinery, 2018. ISBN: 9781450364416. DOI: 10.1145/3236950.3236956. URL: https://doi.org/10.1145/3236950.3236956.

[101] Christopher Schwaab and Jeremy G. Siek. "Modular type-safety proofs in Agda". In: *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*. PLPV '13. Rome, Italy: Association for Computing Machinery, 2013, pp. 3–12. ISBN: 9781450318600. DOI: 10.1145/2428116.2428120. URL: https://doi.org/10.1145/2428116.2428120.

[102] Jan Schwinghammer, Hongseok Yang, Lars Birkedal, François Pottier, and Bernhard Reus. "A Semantic Foundation for Hidden State". In: *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by C.-H. Luke Ong. Vol. 6014. Lecture Notes in Computer Science. Springer, 2010, pp. 2–17. DOI: 10.1007/978-3-642-12032-9_2. URL: https://doi.org/10.1007/978-3-642-12032-9%5C_2.

[103] *SpecTec has been adopted*. [Accessed 14-07-2025]. 2025. URL: https://webassembly.org/news/2025-03-27-spectec/.

[104] David Swasey, Deepak Garg, and Derek Dreyer. "Robust and compositional verification of object capability patterns". In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017), 89:1–89:26. DOI: 10.1145/3133913. URL: https://doi.org/10.1145/3133913.

[105] Wouter Swierstra. "Data types à la carte". In: *Journal of Functional Programming* 18.4 (2008), pp. 423–436. DOI: 10.1017/S0956796808006758.

[106] Don Syme. "Proving Java Type Soundness". In: *Formal Syntax and Semantics of Java*. Ed. by Jim Alves-Foss. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 83–118. ISBN: 978-3-540-48737-1. DOI: 10.1007/3-540-48737-9_3. URL: https://doi.org/10.1007/3-540-48737-9%5C_3.

[107] Yong Kiam Tan, Magnus O Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. "The verified CakeML compiler backend". In: *Journal of Functional Programming* 29 (2019), e2.

[108] *WasmCert-Coq: A mechanisation of Wasm in Coq*. 2025. URL: https://github.com/WasmCert/WasmCert-Coq.

[109] *WasmCert-Coq: A mechanisation of Wasm in Coq, version 2.0.3*. 2025. URL: https://github.com/WasmCert/WasmCert-Coq/tree/v2.0.3.

[110] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization". In: *IEEE Symposium on Security and Privacy*. 2015, pp. 20–37. DOI: 10.1109/SP.2015.9.

[111] Conrad Watt. "Mechanising and evolving the formal semantics of WebAssembly: the Web's new low-level language". PhD thesis. Apollo - University of Cambridge Repository, 2021. DOI: 10.17863/CAM.76476. URL: https://www.repository.cam.ac.uk/handle/1810/329032.

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

[113] Conrad Watt, Petar Maksimovic, Neelakantan R. Krishnaswami, and Philippa Gardner. "A Program Logic for First-Order Encapsulated WebAssembly". In: *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*. Ed. by Alastair F. Donaldson. Vol. 134. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 9:1–9:30. DOI: 10.4230/LIPIcs.ECOOP.2019.9. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2019.9.

[114] Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. "Two Mechanisations of WebAssembly 1.0". In: *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 61–79. ISBN: 978-3-030-90869-0. DOI: 10.1007/978-3-030-90870-6_4. URL: https://doi.org/10.1007/978-3-030-90870-6_4.

[115] Conrad Watt, Maja Trela, Peter Lammich, and Florian Märkl. "WasmRef-Isabelle: A Verified Monadic Interpreter and Industrial Fuzzing Oracle for WebAssembly". In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591224. URL: https://doi.org/10.1145/3591224.

[116] *WebAssembly 128-bit packed SIMD Extension*. [Accessed 01-11-2024].

[117] A.K. Wright and M. Felleisen. "A Syntactic Approach to Type Soundness". In: *Information and Computation* 115.1 (1994), pp. 38–94. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1994.1093. URL: https://www.sciencedirect.com/science/article/pii/S0890540184710935.

[118] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. "Native Client: a sandbox for portable, untrusted x86 native code". In: *Commun. ACM* 53.1 (Jan. 2010), pp. 91–99. ISSN: 0001-0782. DOI: 10.1145/1629175.1629203. URL: https://doi.org/10.1145/1629175.1629203.

[119] Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. "Bringing the WebAssembly Standard up to Speed with SpecTec". In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). DOI: 10.1145/3656440. URL: https://doi.org/10.1145/3656440.

[120] Dachuan Yu, Nadeem Abdul Hamid, and Zhong Shao. "Building Certified Libraries for PCC: Dynamic Storage Allocation". In: *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*. Ed. by Pierpaolo Degano. Vol. 2618. Lecture Notes in Computer Science. Springer, 2003, pp. 363–379. DOI: 10.1007/3-540-36575-3_25. URL: https://doi.org/10.1007/3-540-36575-3%5C_25.

[121] Dachuan Yu and Zhong Shao. "Verification of safety properties for concurrent assembly code". In: *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*. Ed. by Chris Okasaki and Kathleen Fisher. ACM, 2004, pp. 175–188. DOI: 10.1145/1016850.1016875. URL: https://doi.org/10.1145/1016850.1016875.

Appendix A

WebAssembly Numeric Operations

In this section, we list the full set of numeric operators available in WebAssembly. We first present all the operators in Wasm 1.0 by their categories. In the last section, we introduce the new numeric operators added in Wasm 2.0.

A.1 Unary Operations

Name	Description	Int/Float-Only?
clz	Count of leading zero bits	Int
ctz	Count of trailing zero bits	Int
popcnt	Count number of bits set to 1	Int
abs	Absolute value	Float
neg	Negation	Float
sqrt	Square root	Float
ceil/floor/trunc/nearest	Rounding methods	Float

Figure A.1: Wasm 1.0 unary operations

A.2 Binary Operations

Name	Description	Int/Float-Only?
add	Addition	
sub	Subtraction	
mul	Multiplication	
div	Division	Int carries a sign option *sx*
rem_*sx*	Remainder	Int
and/or/xor	Bitwise and/or/xor	Int
shl	Bitwise left shift	Int
shr_*sx*	Bitwise right shift	Int
rotl/rotr	Bitwise left/right rotation	Int
min/max	Minimum/Maximum	Float
copysign	Copy the sign of second argument to first	Float

Figure A.2: Wasm 1.0 binary operations

A.3 Test Operations

Name	Description	Int/Float-Only?
eqz	Testing if input is equal to zero	Int

Figure A.3: Wasm 1.0 test operations

A.4 Relation Operations

Name	Description	Int/Float-Only?
eq	Equality	
ne	Non-equality	
lt/**gt**	Less/greater than	Int carries a sign option *sx*
le/**ge**	Less/greater than or equal to	Int carries a sign option *sx*

Figure A.4: Wasm 1.0 relation operations

A.5 Conversion Operations

The conversion operations are presented in a matrix. The elements in row t_1 and column t_2, if present, display the available list of the conversion operators from type t_1 to type t_2. An empty cell indicates that no such conversion operation is available for the corresponding pair of conversion.

	i32	**i64**	**f32**	**f64**
i32	-	**extend**	**convert**/**reinterpret**	**convert**
i64	**wrap**	-	**convert**	**convert**/**reinterpret**
f32	**trunc**/**reinterpret**	**trunc**	-	**promote**
f64	**trunc**	**trunc**/**reinterpret**	**demote**	-

Figure A.5: Wasm 1.0 conversion operations

The difference between the **reinterpret** operation and the **convert**/**trunc** operations of the same conversion type is that the **reinterpret** operation simply takes the interpretation of the bit content stored in the old numeric value and parse it as a value of the new type, whereas the **convert**/**trunc** operations convert an value of the old type to the same value (or a truncated value) in the new type.

The **extend**, **trunc**, and **convert** operations carry a sign option *sx*.

A.6 Additional Operations in Wasm 2.0

Wasm 2.0 added two additional numeric instructions, the sign-extension operation **extend** and the non-trapping float-to-int conversion operation **trunc_sat**. **iN**.**extend***M* is a signed extension operation that extends a signed integer value of shorter bit-length (8/16/32 bits) to one of a longer length (32/64 bits). **trunc_sat** is a conversion operation similar to the existing **trunc** operation but never returns a trap. Instead, whenever the existing **trunc** conversion would fail (e.g. attempting to convert float values such as NaN), the new non-trapping version simply returns a zero value of the target integer type.

Appendix B

Wasm 1.0 Validity Relations

B.1 Module Instance Validity

The module instance validity relation is a direct combination of the validity relations of each store reference (external values) in the module instance *moduleinst* and the corresponding element of the typing context C:

$$\frac{\begin{array}{l} S \vdash \mathsf{func}\ \mathit{funcaddrs} : \mathsf{func}\ \mathit{fts}' \\ S \vdash \mathsf{table}\ \mathit{tableaddrs} : \mathsf{table}\ \mathit{tabletypes} \\ S \vdash \mathsf{mem}\ \mathit{memaddrs} : \mathsf{mem}\ \mathit{memtypes} \\ S \vdash \mathsf{globals}\ \mathit{globaladdrs} : \mathsf{global}\ \mathit{globaltypes} \\ S \vdash \mathit{exportinsts} : \mathsf{ok} \\ \mathit{exportinsts}\ \mathsf{disjoint} \end{array}}{S \vdash_{\mathsf{i}} \left\{ \begin{array}{ll} \mathsf{types} & : \mathit{fts}, \\ \mathsf{funcaddrs} & : \mathit{funcaddrs}, \\ \mathsf{tableaddrs} & : \mathit{tableaddrs}, \\ \mathsf{memaddrs} & : \mathit{memaddrs}, \\ \mathsf{globaladdrs} & : \mathit{globaladdrs}, \\ \mathsf{exports} & : \mathit{exportinsts} \end{array} \right\} : \left\{ \begin{array}{ll} \mathsf{types} & : \mathit{fts}, \\ \mathsf{funcs} & : \mathit{fts}', \\ \mathsf{tables} & : \mathit{tabletypes}, \\ \mathsf{mems} & : \mathit{memtypes}, \\ \mathsf{globals} & : \mathit{globaltypes} \end{array} \right\}}$$

The validity relations for external values associate each external value with an *external type*, which is determined by the corresponding store reference specified by the external value. The external type consists of four constructors, each corresponding to a specific category of external values.

$$\text{(external type)}\ \mathit{externtype} ::= \quad \mathsf{func}\ \mathit{ft} \mid \mathsf{table}\ \mathit{tabletype} \mid \mathsf{mem}\ \mathit{memtype} \mid \mathsf{global}\ \mathit{globaltype}$$

The full definitions of the external validity relations are given as follows. For reference, the definitions of the Wasm state instances and the definitions of types of Wasm states can be recalled in Figure 2.4 and Figure 2.13 respectively.

$$\frac{S.\mathsf{funcs}[a].\mathsf{type} = ft}{S \vdash \mathsf{func}\ a : \mathsf{func}\ ft}$$

$$\frac{S.\mathsf{tables}[a].\mathsf{max} = tabletype.\mathsf{max} \quad |S.\mathsf{tables}[a].\mathsf{elem}| = tabletype.\mathsf{min}}{S \vdash \mathsf{table}\ a : \mathsf{table}\ tabletype}$$

$$\frac{S.\mathsf{mems}[a].\mathsf{max} = memtype.\mathsf{max} \quad |S.\mathsf{mems}[a].\mathsf{elem}| = memtype.\mathsf{min} \cdot 2^{16}}{S \vdash \mathsf{mem}\ a : \mathsf{mem}\ memtype}$$

$$\frac{S.\mathsf{globals}[a].\mathsf{mut} = globaltype.\mathsf{mut} \quad \mathsf{typeof}(S.\mathsf{globals}[a].\mathsf{value}) = globaltype.\mathsf{type}}{S \vdash \mathsf{global}\ a : \mathsf{global}\ globaltype}$$

Note that the minimum sizes of the external types associated to table and memory external values are determined by their current lengths. For memories, a multiplied of 2^{16} applies since the size of the memory is given in terms of pages.

The definitions of external validity relations are simplified in Wasm 2.0, because the state instances in Wasm 2.0 contain their full types instead of only the maximum types for tables/memories and the mutability for global variables. As a result, the external validity relations for them can be defined by a straightforward equality similar to the case of external function values.

B.2 Limit Validity

Instead of simply defining whether a limit is valid, Wasm defines a more general version of the limit validity relation against an integer argument k, known as the *range* in Wasm:

$$\frac{n \le k \quad m \le k \quad n \le m}{\vdash \{\mathsf{min} : n, \mathsf{max} : \mathsf{Some}\ m\} : k} \quad \frac{n \le k}{\vdash \{\mathsf{min} : n, \mathsf{max} : \mathsf{None}\} : k}$$

A limit satisfying the above relation against k is said to be within range k.

The purpose of this additional range parameter is for defining the validity of different types of limits. In the current version of Wasm, the limits are only used for tables and memories. As introduced in Section 2.7.1, the upper bounds for table and memory size limits are 2^{32} and 2^{16} respectively[1]. This can be specified by defining a table/memory limit to be valid only if it is within range 2^{32} or 2^{16} respectively.

[1] $2^{32} - 1$ for table limits in Wasm 2.0, as described in Section 3.10.

B.3 Store Validity

The store validity relation is similarly a combination of the validity relations of the individual components of the store:

$$\frac{\begin{array}{l} S \vdash \mathit{funcinst} : \mathit{ft} \\ S \vdash \mathit{tableinst} : \mathit{tabletype} \\ S \vdash \mathit{meminst} : \mathit{memtype} \\ S \vdash \mathit{globalinst} : \mathit{globaltype} \\ S = \left\{ \begin{array}{ll} \mathsf{funcs} & : \mathit{funcinsts}, \\ \mathsf{tables} & : \mathit{tableinsts}, \\ \mathsf{mems} & : \mathit{meminsts}, \\ \mathsf{globals} & : \mathit{globalinsts} \end{array} \right\} \end{array}}{\vdash_{\mathsf{s}} S : \mathsf{ok}}$$

We describe the conditions required for each type of state instance to be valid here:

- Recall that a (native Wasm)[2] function instance is a closure over the function definition by adding the module instance and type signature. Therefore, the validity of function instances is defined through the typing of module function definitions, which requires that the type of the function body matches with the function type signature, as defined in Section 2.7.

$$\frac{S \vdash_{\mathsf{i}} \mathit{moduleinst} : C \quad C \vdash \mathit{func} : \mathit{ft}}{S \vdash \{\mathit{ft}, \mathit{moduleinst}, \mathit{func}\}^{\mathsf{Native}} : \mathit{ft}} \; \texttt{funcinst_validity}$$

- For a table or memory instance to be valid, the length of their corresponding contents (elem for tables, data for memories) must not exceed the maximum limit max of the instance if defined. This is defined through the validity relation of the limits corresponding to the table and memory types. In addition, for table instances, each of its non-null function reference must be valid, i.e. within the bound of $S.\mathsf{funcs}$. This is specified through the external validity relation, similar to the module instance validity relation.

$$\frac{S \vdash \mathsf{func}\, \mathit{funcaddrs} : \mathit{fts} \quad \vdash \{\mathsf{min} : |\mathit{funcaddrs}|, \mathsf{max} : m\} : \mathsf{ok}}{S \vdash \{\mathsf{elem} : \mathit{funcaddrs}, \mathsf{max} : m\}} \; \mathsf{tableinst_validity}$$

$$\frac{\vdash \{\mathsf{min} : |\mathit{bytes}|, \mathsf{max} : m\} : \mathsf{ok}}{S \vdash \{\mathsf{data} : \mathit{bytes}, \mathsf{max} : m\}} \; \mathsf{meminst_validity}$$

- Global instances are always valid in Wasm 1.0[3].

[2] Host function instances are out of scope of the Wasm specification. However, certain assumptions about the behaviour of the host functions are required for the soundness property of Wasm to hold. This was discussed in Section 3.5.

[3] In Wasm 2.0, all Wasm's state instances are annotated by their full type of states. In particular, global instances are annotated by the global type, which includes both the mutability tag and the type signature of the enclosed value. Therefore, In Wasm 2.0, the type of the value needs to match its type signature for global instances to be valid.

Appendix C

Iris-Wasm Proof Rules

This section presents the core Iris-Wasm proof rules for WebAssembly 1.0. Note that in the Coq formalization, additional rules exist for convenience. Notably, many rules like `wp_call` have a corresponding `wp_call_ctx` rule for successively applying the bind rule and the `wp_call` rule.

C.1 Pure Rules

wp_unop

$$\frac{[\![t.\mathit{unop}]\!](c) = c' * \triangleright\Phi(\mathbf{immV}\ [t.\mathbf{const}\ c']) * \overset{\text{FR}}{\hookrightarrow} F}{\mathsf{wp}\ [t.\mathbf{const}\ c; t.\mathbf{unop}\ \mathit{unop}] \left\{w, \Phi(w) * \overset{\text{FR}}{\hookrightarrow} F\right\}}$$

WP_BINOP

$$\frac{[\![t.\mathit{binop}]\!](c_1, c_2) = c \quad * \quad \triangleright\Phi(\mathbf{immV}\ [t.\mathbf{const}\ c]) \quad * \quad \overset{\text{FR}}{\hookrightarrow} F}{\mathsf{wp}\ [t.\mathbf{const}\ c_1; t.\mathbf{const}\ c_2; t.\mathbf{binop}\ \mathit{binop}] \left\{w, \Phi(w) * \overset{\text{FR}}{\hookrightarrow} F\right\}}$$

WP_BINOP_FAILURE

$$\frac{[\![t.\mathit{binop}]\!](c_1, c_2) = \bot * \ \triangleright\Phi(\mathbf{trapV}) * \overset{\text{FR}}{\hookrightarrow} F}{\mathsf{wp}\ [t.\mathbf{const}\ c_1; t.\mathbf{const}\ c_2; t.\mathbf{binop}\ \mathit{binop}] \left\{w, \Phi(w) * \overset{\text{FR}}{\hookrightarrow} F\right\}}$$

(Other numerical rules are similar)

WP_UNREACHABLE

$$\frac{\triangleright\Phi(\mathbf{trapV}) * \overset{\text{FR}}{\hookrightarrow} F}{\mathsf{wp}\ [\mathbf{unreachable}] \left\{w, \Phi(w) * \overset{\text{FR}}{\hookrightarrow} F\right\}}$$

WP_DROP

$$\frac{\triangleright\Phi(\mathbf{immV}\ []) * \overset{\text{FR}}{\hookrightarrow} F}{\mathsf{wp}\ [t.\mathbf{const}\ c; \mathbf{drop}] \left\{w, \Phi(w) * \overset{\text{FR}}{\hookrightarrow} F\right\}}$$

WP_NOP

$$\frac{\triangleright\Phi(\mathbf{immV}\ []) * \overset{\text{FR}}{\hookrightarrow} F}{\mathsf{wp}\ [\mathbf{nop}] \left\{w, \Phi(w) * \overset{\text{FR}}{\hookrightarrow} F\right\}}$$

WP_SELECT_TRUE

$$\frac{n \neq 0 * \triangleright\Phi(\mathbf{immV}\ [t.\mathbf{const}\ c_1]) * \overset{\text{FR}}{\hookrightarrow} F}{\mathsf{wp}\ [t.\mathbf{const}\ c_1; t.\mathbf{const}\ c_2; i32.\mathbf{const}\ n; \mathbf{select}] \left\{w, \Phi(w) * \overset{\text{FR}}{\hookrightarrow} F\right\}}$$

`WP_SELECT_FALSE`

$$\frac{n = 0 * \triangleright \Phi(\textbf{immV}\ [t.\textbf{const}\ c_2]) * \xhookrightarrow{\textsc{Fr}} F}{\mathsf{wp}\ [t.\textbf{const}\ c_1; t.\textbf{const}\ c_2; i32.\textbf{const}\ n; \textbf{select}] \left\{ w, \Phi(w) * \xhookrightarrow{\textsc{Fr}} F \right\}}$$

C.2 Bind Rules

`WP_FRAME_BIND`

$$\frac{\xhookrightarrow{\textsc{Fr}} F * \left[\xhookrightarrow{\textsc{Fr}} F_1 \mathrel{-\!\!*} \mathsf{wp}\, es \left\{ w, \exists F_1', \xhookrightarrow{\textsc{Fr}} F_1' \right\} * \left(\xhookrightarrow{\textsc{Fr}} F \mathrel{-\!\!*} \mathsf{wp}\ [\textbf{frame}_n\{F_1'\}\ w\ \textbf{end}]\ \{w, \Phi(w)\} \right) \right]}{\mathsf{wp}\ [\textbf{frame}_n\{F_1\}\ es\ \textbf{end}]\ \{w, \Phi(w)\}}$$

`WP_CTX_BIND`

$$\frac{\mathsf{wp}\, es\, \{ w, \mathsf{wp}\, lh_i[w]\ \{ w', \Phi(w') \} \}}{\mathsf{wp}\, lh_i[es]\ \{ w', \Phi(w') \}}$$

C.3 Function Call Rules

`wp_call`

$$\frac{(F.\mathsf{inst.funcs}[i] = addri) * \xhookrightarrow{\textsc{Fr}} F * \triangleright \left(\xhookrightarrow{\textsc{Fr}} F \mathrel{-\!\!*} \mathsf{wp}\ [\textbf{invoke}\ addri]\ \{w, \Phi(w)\} \right)}{\mathsf{wp}\ [\textbf{call}\ i]\ \{w, \Phi(w)\}}$$

`WP_CALL_INDIRECT_SUCCESS`

$$\frac{\begin{array}{c} \xhookrightarrow{\textsc{Fr}} F * (F.\mathsf{inst.tabs}[0] = ta) * \\ (ta \xmapsto{\mathsf{wt}}_k a) * (a \xmapsto{\mathsf{wf}} cl) * (F.\mathsf{inst.types}[i] = \mathsf{typeof}\ cl) * \\ \triangleright \left((ta \xmapsto{\mathsf{wt}}_k a) \mathrel{-\!\!*} (ta \xmapsto{\mathsf{wf}} cl) \mathrel{-\!\!*} (\xhookrightarrow{\textsc{Fr}} F) \mathrel{-\!\!*} \mathsf{wp}\ [\textbf{invoke}\ a]\ \{w, \Phi(w)\} \right) \end{array}}{\mathsf{wp}\ [\textbf{i32.const}\ k; \textbf{call_indirect}\ \ i]\ \{w, \Phi(w)\}}$$

`wp_invoke_native`

$$\frac{\begin{array}{c} |vs| = |ts_1| * cl = \{(ts_1 \to ts_2), (inst; ts), es\}^{\mathsf{Native}} * F' = \{\mathsf{locs} := vs +\!\!+\ \textbf{zeros}(ts); \mathsf{inst} := inst\} * \\ i \xmapsto{\mathsf{wf}} cl * \xhookrightarrow{\textsc{Fr}} F * \triangleright \left[\begin{array}{c} (i \xmapsto{\mathsf{wf}} cl * \xhookrightarrow{\textsc{Fr}} F) \mathrel{-\!\!*} \\ \mathsf{wp}\ \left[\textbf{frame}_{|ts_2|}\ F'\ (\textbf{block}\ ([] \to ts_2)\ es)\ \textbf{end}\right]\ \{w, \Phi(w)\} \end{array} \right] \end{array}}{\mathsf{wp}\ (vs +\!\!+\ [\textbf{invoke}\ i])\ \{w, \Phi(w)\}}$$

`wp_invoke_host`

$$\frac{\begin{array}{c} |vs| = |ts_1| * cl = \{(ts_1 \to ts_2), hidx\}^{\mathsf{Host}} * \\ i \xmapsto{\mathsf{wf}} cl * \xhookrightarrow{\textsc{Fr}} F * \triangleright \left[\begin{array}{c} (i \xmapsto{\mathsf{wf}} cl * \xhookrightarrow{\textsc{Fr}} F) \mathrel{-\!\!*} \\ \mathsf{wp}\ (\textbf{call_host}\ [ts_1 \to ts_2]\ hidx\ vs]\ \{w, \Phi(w)\} \end{array} \right] \end{array}}{\mathsf{wp}\ (vs +\!\!+\ [\textbf{invoke}\ i])\ \{w, \Phi(w)\}}$$

C.4 Stateful Rules

`WP_LOCAL_GET`

$$\frac{F.\mathsf{locs}[i] = v * \triangleright \Phi(\textbf{immV}\ [v]) * \xhookrightarrow{\textsc{Fr}} F}{\mathsf{wp}\ [\textbf{local.get}\ i] \left\{ w, \Phi(w) * \xhookrightarrow{\textsc{Fr}} F \right\}}$$

`WP_LOCAL_SET`

$$\frac{i < |F.\mathsf{locs}| * \triangleright \Phi(\textbf{immV}\ []) * \xhookrightarrow{\textsc{Fr}} F}{\mathsf{wp}\ [v; \textbf{local.set}\ i] \left\{ w, \Phi(w) * \xhookrightarrow{\textsc{Fr}} F[\mathsf{locs} := F.\mathsf{locs}[i := v]] \right\}}$$

`WP_LOCAL_TEE`

$$\frac{\triangleright(\xhookrightarrow{\textsc{Fr}} F \mathrel{-\!\!*} \mathsf{wp}\ [v; v; \textbf{local.set}\ i]\ \{w, \Phi(w)\}) * \xhookrightarrow{\textsc{Fr}} F}{\mathsf{wp}\ [v; \textbf{local.tee}\ i]\ \{w, \Phi(w)\}}$$

WP_GLOBAL_GET

$$\frac{F.\mathsf{inst.globs}[i] = n * n \overset{\mathsf{wg}}{\longmapsto} \{mutability, v\} * \triangleright\Phi(\mathbf{immV}\ [v]) * \overset{\mathrm{FR}}{\hookrightarrow} F}{\mathsf{wp}\ [\mathbf{global.get}\ i] \left\{w, \Phi(w) * n \overset{\mathsf{wg}}{\longmapsto} g * \overset{\mathrm{FR}}{\hookrightarrow} F\right\}}$$

WP_GLOBAL_SET

$$\frac{F.\mathsf{inst.globs}[i] = n * n \overset{\mathsf{wg}}{\longmapsto} \{\mathbf{mut}, -\} * \triangleright\Phi(\mathbf{immV}\ []) * \overset{\mathrm{FR}}{\hookrightarrow} F}{\mathsf{wp}\ [v; \mathbf{global.set}\ i] \left\{w, \Phi(w) * n \overset{\mathsf{wg}}{\longmapsto} \{\mathbf{mut}, v\} * \overset{\mathrm{FR}}{\hookrightarrow} F\right\}}$$

WP_LOAD

$$\frac{\begin{array}{c} F.\mathsf{inst.mems}[0] = n * n \overset{\mathsf{wms}}{\longmapsto}_{(i+off)} bs * \\ \mathsf{deserialise}(bs) = v * \mathbf{typeof}(\mathsf{v}) = t * \triangleright\Phi(\mathbf{immV}\ [v]) * \overset{\mathrm{FR}}{\hookrightarrow} F \end{array}}{\mathsf{wp}\ [\mathbf{i32.const}\ i; \mathbf{load}\ t\ off] \left\{w, \Phi(w) * n \overset{\mathsf{wms}}{\longmapsto}_{(i+off)} bs * \overset{\mathrm{FR}}{\hookrightarrow} F\right\}}$$

WP_STORE

$$\frac{\begin{array}{c} F.\mathsf{inst.mems}[0] = n * n \overset{\mathsf{wms}}{\longmapsto}_{(i+off)} bs * \\ \mathbf{typeof}(\mathsf{v}) = t * |bs| = |t| * \mathsf{serialise}(v) = bv * \triangleright\Phi(\mathbf{immV}\ []) * \overset{\mathrm{FR}}{\hookrightarrow} F \end{array}}{\mathsf{wp}\ [\mathbf{i32.const}\ i; v; \mathbf{store}\ t\ off] \left\{w, \Phi(w) * n \overset{\mathsf{wms}}{\longmapsto}_{(i+off)} bv * \overset{\mathrm{FR}}{\hookrightarrow} F\right\}}$$

WP_MEMORY_SIZE

$$\frac{F.\mathsf{inst.mems}[0] = n * n \overset{\mathsf{mlen}}{\longmapsto} len * \lfloor len/64\mathsf{Ki} \rfloor = c * \triangleright\Phi(\mathbf{immV}\ [\mathbf{i32.const}\ c]) * \overset{\mathrm{FR}}{\hookrightarrow} F}{\mathsf{wp}\ [\mathbf{memory.size}] \left\{w, \Phi(w) * n \overset{\mathsf{mlen}}{\longmapsto} len * \overset{\mathrm{FR}}{\hookrightarrow} F\right\}}$$

WP_MEMORY_GROW

$$\frac{\begin{array}{c} F.\mathsf{inst.mems}[0] = n * n \overset{\mathsf{mlen}}{\longmapsto} len * \lfloor len/64\mathsf{Ki} \rfloor = c * \\ \triangleright\Phi(\mathbf{immV}\ [\mathbf{i32.const}\ c]) * \triangleright\Psi(\mathbf{immV}\ [\mathbf{i32.const}\ (-1)]) * \overset{\mathrm{FR}}{\hookrightarrow} F \end{array}}{\mathsf{wp}\ [\mathbf{i32.const}\ c; \mathbf{memory.grow}] \left\{ w, \begin{array}{c} \Big((\Phi(w) * n \overset{\mathsf{mlen}}{\longmapsto} (len + c \cdot 64\mathsf{Ki}) * n \overset{\mathsf{wms}}{\longmapsto}_{len} (0\mathsf{x}00)^{64\mathsf{Ki}}) \vee \\ (\Psi(w) * n \overset{\mathsf{mlen}}{\longmapsto} len)\Big) * \overset{\mathrm{FR}}{\hookrightarrow} F \end{array} \right\}}$$

The memory-related rules in this section used two additional types of points-to predicates omitted in the main text:

- $n \overset{\mathsf{mlen}}{\longmapsto} len$, indicating that the length of the memory at index n is len. This resource is required for instructions dealing with whole memories (e.g. **memory**.**size** and **memory**.**grow**);
- $n \overset{\mathsf{wms}}{\longmapsto}_{off} bs$, a shorthand for a number of consecutive $|bs|$ normal points-to predicates for individual memory bytes at memory n starting at offset off, with contents bs. In other words,

$$(n \overset{\mathsf{wms}}{\longmapsto}_{off} bs) \equiv \circledast_{i=0}^{|bs|-1} (n \overset{\mathsf{wm}}{\longmapsto}_{off+i} bs_i)$$

C.5 Control Rules

WP_BR

$$\frac{(lh_k[vs +\!\!+\ \mathbf{br}\ k] = les) * (|vs| = n) * \overset{\mathrm{FR}}{\hookrightarrow} F * \triangleright\left(\overset{\mathrm{FR}}{\hookrightarrow} F \twoheadrightarrow \mathsf{wp}\ (vs +\!\!+\ es)\ \{w, \Phi(w)\}\right)}{\mathsf{wp}\ [\mathbf{label}_n\{es\}\ les\ \mathbf{end}]\ \{w, \Phi(w)\}}$$

WP_RETURN

$$\frac{(lh_i[(vs +\!\!+\ \mathbf{return})] = les) * (|vs| = n) * \triangleright\left[\mathsf{wp}\ (vs +\!\!+\ es) \left\{w, \Phi(w) * \overset{\mathrm{FR}}{\hookrightarrow} F\right\}\right]}{\mathsf{wp}\ [\mathbf{frame}_n\ F_0\ les\ \mathbf{end}] \left\{w, \Phi(w) * \overset{\mathrm{FR}}{\hookrightarrow} F\right\}}$$

WP_BLOCK

$$\frac{|vs| = |ts_1| * \xhookrightarrow{\text{FR}} F * \triangleright \left[\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathsf{wp}\,[\mathbf{label}_{|ts_2|}\{[]\}\ (vs +\!\!+\ es)\ \mathbf{end}]\,\{w, \Phi(w)\}\right]}{\mathsf{wp}\,(vs +\!\!+\ [\mathbf{block}\ (ts_1 \to ts_2)\ es])\,\{w, \Phi(w)\}}$$

WP_LOOP

$$\frac{\begin{array}{c}|vs| = |ts_1| * \xhookrightarrow{\text{FR}} F * \\ \triangleright \left[\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathsf{wp}\,[\mathbf{label}_{|ts_1|}\{[\mathbf{loop}\ (ts_1 \to ts_2)\ es]\}\ (vs +\!\!+\ es)\ \mathbf{end}]\,\{w, \Phi(w)\}\right]\end{array}}{\mathsf{wp}\,(vs +\!\!+\ [\mathbf{loop}\ (ts_1 \to ts_2)\ es])\,\{w, \Phi(w)\}}$$

WP_IF_TRUE

$$\frac{c \neq 0 * \xhookrightarrow{\text{FR}} F * \triangleright \left[\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathsf{wp}\,[\mathbf{block}\,ft\ es_1]\ \{w, \Phi(w)\}\right]}{\mathsf{wp}\,[\mathbf{i32.const}\ c; \mathbf{if}\,ft\ es_1\ es_2]\ \{w, \Phi(w)\}}$$

WP_IF_FALSE

$$\frac{c = 0 * \xhookrightarrow{\text{FR}} F * \triangleright \left[\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathsf{wp}\,[\mathbf{block}\,ft\ es_2]\ \{w, \Phi(w)\}\right]}{\mathsf{wp}\,[\mathbf{i32.const}\ c; \mathbf{if}\,ft\ es_1\ es_2]\ \{w, \Phi(w)\}}$$

WP_BR_IF_TRUE

$$\frac{c \neq 0 * \xhookrightarrow{\text{FR}} F * \triangleright \left[\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathsf{wp}\,[\mathbf{br}\ \ i]\ \{w, \Phi(w)\}\right]}{\mathsf{wp}\,[\mathbf{i32.const}\ c; \mathbf{br_if}\ \ i]\ \{w, \Phi(w)\}}$$

WP_BR_IF_FALSE

$$\frac{c = 0 * \xhookrightarrow{\text{FR}} F * \triangleright \Phi(\mathbf{immV}\ [])}{\mathsf{wp}\,[\mathbf{i32.const}\ c; \mathbf{br_if}\ \ i]\ \left\{w, \Phi(w) * \xhookrightarrow{\text{FR}} F\right\}}$$

WP_BR_TABLE

$$\frac{iss[c] = j\ * \xhookrightarrow{\text{FR}} F * \triangleright \left[\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathsf{wp}\,[\mathbf{br}\ \ j]\ \{w, \Phi(w)\}\right]}{\mathsf{wp}\,[\mathbf{i32.const}\ c; \mathbf{br_table}\ iss\ i]\ \{w, \Phi(w)\}}$$

WP_BR_TABLE_LENGTH

$$\frac{|iss| \leq c * \xhookrightarrow{\text{FR}} F * \triangleright \left[\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathsf{wp}\,[\mathbf{br}\ \ i]\ \{w, \Phi(w)\}\right]}{\mathsf{wp}\,[\mathbf{i32.const}\ c; \mathbf{br_table}\ iss\ i]\ \{w, \Phi(w)\}}$$