Semántica de WebAssembly en Maude
Semantics of WebAssembly in Maude

Trabajo de Fin de Máster
Curso 2024–2025

Autor
Rafael Morales Palacios

Directores
Ignacio Fábregas Alfaro
Rubén Rafael Rubio Cuéllar

Máster en Métodos Formales en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

22 de septiembre de 2025

Semántica de WebAssembly en Maude
Semantics of WebAssembly in Maude

Trabajo de Fin de Máster
Departamento de Sistemas Informáticos y Computación

Autor
Rafael Morales Palacios

Directores
Ignacio Fábregas Alfaro
Rubén Rafael Rubio Cuéllar

Convocatoria: *Septiembre 2025*
Calificación: *9.25 (Sobresaliente)*

Máster en Métodos Formales en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

22 de septiembre de 2025

Acknowledgements

First, I want to thank the teachers of this Master's program, who have sparkled in me an interest in this field, at a time when I was so let down with the industry.

Thanks to my friends, the new ones and the old ones, for their unconditional support and advice when I'm feeling lost.

And thanks to my family, who support me deeply and encouraged me to pursue this Master's.

Thanks. Thank you all.

Resumen

Semántica de WebAssembly en Maude

WebAssembly (Wasm) es un lenguaje de programación de bajo nivel diseñado, en un principio, para mejorar el rendimiento de aplicaciones web. A lo largo de los años, ha ganado popularidad en diferentes campos, incluyendo el Internet de las Cosas, la computación frontera y los contratos inteligentes. Una característica singular de este lenguaje ha sido definido formalmente con un conjunto de reglas semánticas y de validación para sus instrucciones.

En este Trabajo de Fin de Máster, desarrollamos una formalización de la semántica de WebAssembly en Maude, un lenguaje de especificación ejecutable basado en la lógica de reescritura. Esta formalización representa los programas y el entorno de WebAssembly como términos y la ejecución de sus instrucciones como reglas de reescritura, siguiendo las reglas semánticas de la especificación original. Además, la semántica se puede utilizar como un intérprete del lenguaje, que hemos comprobado que se comporta como el intérprete oficial mediante testing diferencial. Finalmente, mostramos cómo puede utilizarse para verificar propiedades sobre programas de WebAssembly.

Palabras clave

Semántica, WebAssembly, Maude, métodos formales, especificación de software, verificación de software, lógica de reescritura

Abstract

Semantics of WebAssembly in Maude

WebAssembly (Wasm) is a low-level programming language designed, at first, to improve the performance of web applications. Through the years, it has gained popularity in several other fields, including Internet of Things, edge computing, and smart contracts. A distinctive feature of this language is that it has been formally defined with a set of semantic and validation rules for its instructions.

In this Master's thesis, we propose a formalization of the semantics of WebAssembly in Maude, an executable specification language based on rewriting logic. This formalization represents WebAssembly programs and environment as terms and the exection of its instructions as rewrite rules, following the semantic rules of the original specification. In addition, the semantics can be used as an interpreter of the language, which we have verified that behaves like the official interpreter using differential testing. Finally, we show how it can be used to verify properties on WebAssembly programs.

Keywords

Semantics, WebAssembly, Maude, formal methods, software specification, software verification, rewriting logic

Contents

List of figures

List of Listings

Chapter 1

Introduction

"... j'observe, je déchiffre, je jouis d'un texte qui éclate de lisibilité par cela même qu'il ne dit pas"
— Roland Barthes, Fragments d'un discours amoureux [3]

In this first chapter, a general overview of the work will be introduced, starting from the motivation and goals of this Master's thesis, and detailing the work plan followed to reach the proposed goals.

1.1. Motivation

WebAssembly (Wasm) [40] is a low-level programming language that has gained popularity through the last years in a wide range of fields, namely: web applications, smart contracts, Internet of things (IoT), and edge computing [28, 30, 45]. Moreover, it has generated interest in the academic environments due to its numerous applications. It was initially designed as a high-performance client-side tool to ease computations on the web but, due to it being an open standard, several platforms and high-level compilers for Wasm have been developed for major programming languages. To name a few: Emscripten [11] from C/C++, Rust [32], and TinyGo [33] from Go. Its potential in a wide range of fields, and specially in IoT and edge computing, underlines the need for rigorous and correct software developed in this language. Moreover, one of the most interesting features of WebAssembly is that, unlike many other programming languages, it is formally defined [40]. This means its specification contains semantic rules for each of its functionalities, which makes it a perfect candidate for a semantic-driven verification. Previous work on the verification of programs in this language based on its semantics has been done with theorem provers in [37] and with other tools like $\mathbb{K}$ [31] in [19].

Although the semantics are already formalized in pen-and-paper small-step semantics, the novelty of this work resides on the tool used for the formalization and verification. The work in this Master's thesis implements a subset of the semantics of the language in Maude [6]. This tool implements rewriting logic, creating a framework to develop and formalize programs. As mentioned in its manual [6], Maude programs are formal executable specifications, and models over which the user can verify formal properties. This makes Maude an appropriate tool to specify formal semantics of programming languages, allowing their verification, checking their behavior against a formal specification. In addition, Maude allows separating functionalities in modules, which would help construct the specification incrementally, and ease the work for future extensions of the implementation.

1.2. Goals

The main goal of this project is to implement the semantics of a subset of WebAssembly 2.0 in Maude. This would allow us to analyze WebAssembly programs by means of the implementation of *executable* formal semantics. As an overview, the general goals of this project were:

- Study of WebAssembly and Maude
- Formalization of the semantics in Maude
- Verification of WebAssembly programs
- Documentation in the Master's thesis report

Internal goals were included throughout the development, taking into account the progress and implementation achievements reached along the way. At first, simple goals were set:

- Understand and implement the main objects of the WebAssembly execution environment.
- Implement simple instructions. Namely, numeric and variable-related instructions.

As the development advanced, more complex goals were introduced:

- Implement WebAssembly functions support
- Implement floating-point values support
- Implement full WebAssembly modules loading
- Load WebAssembly files using Maude
- Verify WebAssembly programs using the semantics

It is worth noting that some specification features have been left out from the implementation. Namely, complex value types and interaction between modules by means of component imports and exports. This is detailed in Section 6.2 as future work.

1.3. Work plan

This work started with a documentation-focused phase between November and December 2024, which resulted in a meeting with the Master's thesis tutors to share knowledge on the topic and plan the initial goals of the work. This first phase was mainly focused on understanding WebAssembly, the structures and components that would be needed to formalize the environment and how to implement the semantics.

Then, an initial implementation phase was centered in coding the fundamental structures used in WebAssembly specification to have a basic execution environment. This included an instruction list, a stack, and a key-value store to create small local environments. Later, these were changed to predefined Maude structures for consistency and convenience. This phase included basic instructions for handling variables in a `local` environment and numeric instructions.

Once a small working environment was implemented, the target was to improve it with more complex features, like control instructions (`block`, `if-then-else`, `loop`) and floating-point numbers. Later, the system was extended to handle function calls and custom type declarations. This allowed the system to process WebAssembly modules as a whole.

In the last implementation phase, the goal was set to develop an interface that allowed loading WebAssembly modules directly from files. This was achieved by using the Maude modules for external file manipulation and other rewriting logic properties, which will be detailed in Section 3.4.

The verification of WebAssembly programs was carried out along with the implementation of the semantics. Once the semantics formalized reached a significant subset, complex programs were verified. The testing of the semantics was performed along the development process but specially at the final phase, as the implemented subset allowed for more complex and meaningful tests.

During the development process, I took notes of the progress, problems and solutions that arose. Finally, I started the full documentation of the work in this report. However, while writing this document, some bugs and implementation problems were found when fully testing the system. Therefore, the final phase included documentation, testing and bug-fixing.

1.4. Structure of the work

This document is divided in five chapters. Chapter 1 corresponds to the introduction of the document, including the motivation, goals and work plan. Chapter 2 discusses the state of the art and preliminaries of the work, including an introduction to WebAssembly and Maude. Then, Chapter 3 includes a full description of the semantics. First, it describes how the WebAssembly environment is modeled in Maude, then it delves into each of the execution phases, including the semantic rules. Chapter 4 provides insight on the process followed for testing the implementation in Maude. After that, Chapter 5 shows how the system is used for program verification, as well as explaining how the system was tested to assert that the semantics are correctly implemented. And, finally, Chapter 6 describes possible extensions and future work, as well as discussing some conclusions.

The code developed for this Master's thesis is available in the following GitHub repository:

`https://github.com/rafamor-exe/wasm-maude`

Figure 1.1 shows the structure of the repository. `wasm-maude-semantics.maude` includes the Maude modules implementing the WebAssembly semantics. Moreover, a test module with simple examples is included in `wasm-test-mod.maude`. The directory `test-suite` contains the test files discussed in Section 4.1 and two subdirectories: `randomized`, with template tests to override values with random inputs, and `from-spec-suite`, containing adapted instances from the official test suite [42]. The directory `scripts` includes the scripts developed to evaluate the system (`.sh` Bash scripts) and compare the results with the official Wasm interpreter [41] (`wasmVSwasmMaude_test-eval.py`), as well as a Python script to transform WebAssembly modules to Maude modules, as discussed in Section 5.1.2. Finally, the file `calculator-example-mod.maude` contains the code for the example modules shown in Section 2.3.

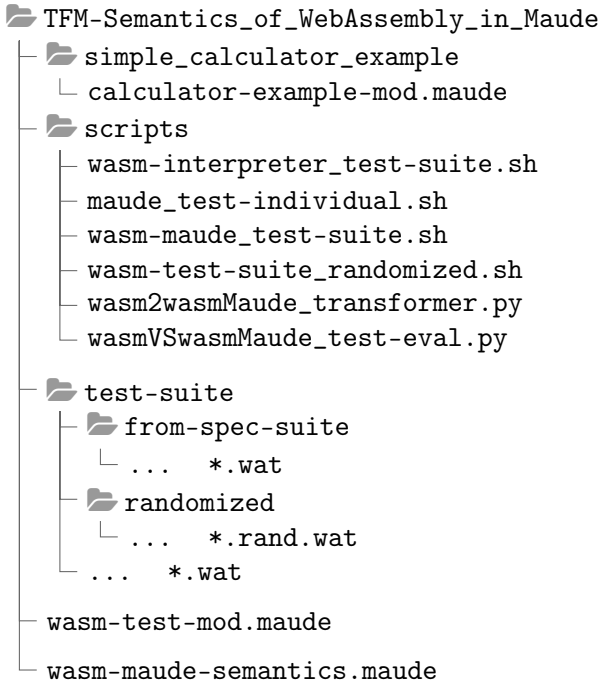

Figure 1.1: Code repository structure

Chapter 2

Preliminaries

This chapter contains the background and previous work found around the topic. It also serves as an introduction to WebAssembly and Maude, as the main subjects of this Master's thesis.

2.1. WebAssembly

WebAssembly (Wasm) is a low-level programming language presented around 2017 with the characteristic that it was formally specified from its creation [15]. The initial development of WebAssembly was carried out by engineers at Mozilla, Google, Microsoft, and Apple and their contributions arose from their previous attempts to build faster machine code for the Web. Specifically, these previous projects were ActiveX [26], by Microsoft, and Native Client (NaCl) [43], by Google. With the appearance of Wasm, reported issues regarding portability and security of these two tools were potentially solved, and it became a common ground to work from for fast Web projects.

In the specification of WebAssembly, the authors mention the main design goals of the language, which can be summarized in three key points [40]:

- **Fast**: Wasm is designed to execute client-side computations with near-native performance via fast decoding and compiling, potentially reducing web rendering times [37]. This is achieved thanks to its byte-code representation, which makes it easier to decode.

- **Well-defined**: As mentioned before, the language is formally specified, meaning that a small-step operational semantics have been written to specify its functionalities.

- **Portable**: Wasm is also designed to be cross-platform, in the sense that its code can be compiled to any hardware, used or deployed in any platform, and possibly compiled from any language as long as the specification is followed.

There exist several runtime environments that support WebAssembly, some the most popular ones [45] are: V8 [9] in Chrome, SpiderMonkey by Mozilla [8], wasmer [35], and wasmtime [36].

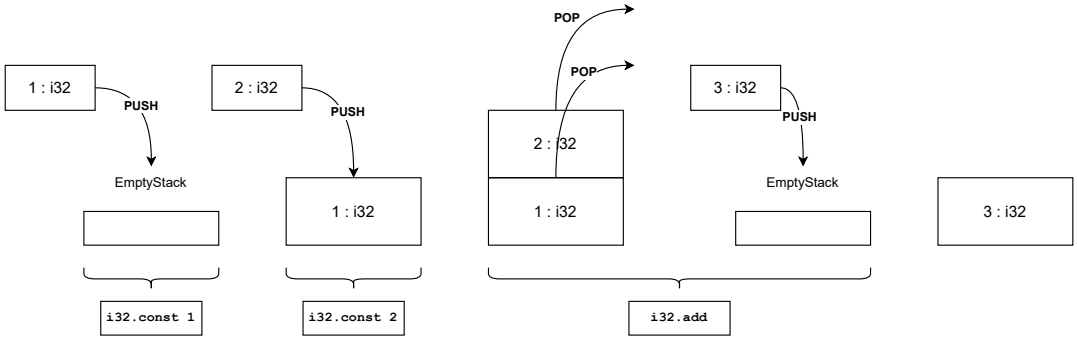

Figure 2.1: Stack usage example

2.1.1. Wasm design

First, Wasm supports two code formats: byte-code and text-based. The former is used in WASM (`.wasm`) files and includes the code as hexadecimal values. It has a "one-to-one" relation with the text-based format, which is used in WAT (`.wat`) files. As an example, the addition instruction `i32.add` in WAT format is equivalent to the hexadecimal value `0x6A` in byte-code. In this project, we have exclusively considered text-based format, as the semantics can be formalized for only one of them without loss of generality.

Wasm was designed to be portable, and it shall not be constrained by any platform-specific details. In order to establish a common interface for its execution across platforms, some simple components and structures are required.

The main data structure in the Wasm execution model is the runtime *stack*. Most instructions push or pop values to/from the stack to operate with them or perform control operations. A simple example would be the sequence:

```
i32.const 1
i32.const 2
i32.add
```

The execution of these instructions is illustrated in Figure 2.1. The first two instructions push values to the stack, and the third, pops two values, adds them, and pushes the result to the stack.

Moreover, the environment of Wasm also includes `local` and `global` variable stores, which can be used, respectively, to store and access values locally (during the execution of a single function) or globally (along the execution of the whole module). In other words, local variables are part of the local environment of the runtime of each function, and are reset once the function ends. On the other hand, global variables keep their values along the execution, but they are modifiable only if they are defined as mutable variables, otherwise they are constants.

Another data structure of the Wasm execution environment is `memory`, which is a binary buffer that instructions may use to load or store values. This component was not implemented in this work, but some ideas have been developed for possible future extensions. This component does not contain variables, however, it allows storing and accessing unstructured values encoded in their binary representations. Moreover, `memory` is shared across components in the system instantiating the module. As part of the Wasm system, it is bound to its environment and isolated from the host system, meaning that it does not expose memory from the host, which is a remarkable point for IoT systems [30].

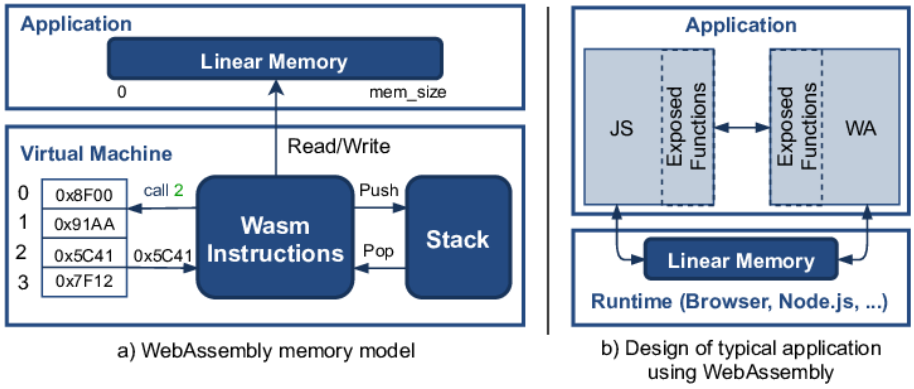

Figure 2.2: WebAssembly high-level architecture [5]

Figure 2.2 illustrates the architecture of the Wasm system, including the mentioned components. On the left, the virtual machine includes the instructions, encoded as hexadecimal values, the stack, which they use as explained before, and the memory as a separate entity. On the right, an example of application is shown, connecting the Wasm system and a JavaScript environment by exposed functions and memory.

2.1.2. Code

From a high-level view, Wasm code is organized in modules, which are self-contained units that define functions, types, tables, and global data/memory components, or variables. In web applications, these modules are usually loaded into the Wasm browser environment, where exported functions can be called from other environments (such as JavaScript).

Functions contain an explicit sequence of instructions and can be called both from inside the module or from external environments (for example, another module or a JavaScript program). In the latter case, the function must be "exported". Exporting a function, or any other component, requires defining an export component, indicating the object and the name it will be referred to in other environments.

Instructions can be of different types: numeric instructions, control operations, parametric, variable-related, table operations, or memory operations. They are executed sequentially and may interact with the stack, variable stores, or other module components.

- Numeric instructions: they compute numeric operations with numeric values that they get from the stack, pushing the result to it.
- Parametric instructions: type-agnostic operations over stack values.
- Variable instructions: they allow getting and setting values from variables.
- Control instructions: they affect the control flow of the program by means of instruction blocks, jumps or function calls.
- Vector instructions: operations with values in vectors.
- Table instructions: operations regarding tables for pointers and indirect function calls.
- Memory instructions: operations regarding the memory module component and preloaded binary data.

```
(module
  (func $mult (param $fst i32) (param $snd i32)
    local.get $fst
    local.get $snd
    i32.mul
    global.set $res)
  (export "mult" (func $mult))
  (global $res (mut i32) (i32.const 1))
)
```

Listing 1: Multiplication module example

As mentioned in the previous section, Wasm has both a byte-code and a text-based representation. Given that there is a "one-to-one" equivalence between both, without loss of generality, this work will be centered on the text-based representation semantics. The translation from text-based to byte-code and vice versa can be performed with the WebAssembly Binary Toolkit (WABT) [34].

In the following sections, more details will be given on each of the functionalities of Wasm. Now, consider the simple Wasm module defined in Listing 1. This module defines and exports a simple function to multiply two input parameters of integer type (`i32`) and store the result in a global variable `$res`. As it is shown, the module declaration encapsulates the definition of the function, the global variable, and the function export declaration. The function definition declares the two statically-typed parameters and lists the sequence of instructions. Once this module is loaded, the function can be invoked externally with the name `"mult"`.

2.1.3. Execution phases

When a Wasm module is loaded into a Wasm environment, it executes three sequential steps: allocation, validation, and execution (or function invocation). The first two are in charge of checking whether all module components are ready to be used, and the latter allows the execution of module functions by other elements in the system.

2.1.3.1. Module decoding and allocation

In this phase, the system reads the module and processes each of its components. In a real implementation, this phase consists of decoding a WebAssembly program in binary format, compiling the module, and allocating its components in data structures. As in this project we exclusively consider text-based format, to avoid confusion with the term *decoding*, along this document this phase will be referred to as *allocation*.

2.1.3.2. Module validation

During the validation phase, the code in the module is proved to be valid according to the validation rules defined in the specification. These rules type-check the code according to the instructions semantics and the type declarations in program blocks and functions.

If any component in the module is not valid, the module itself is not valid and, therefore, it cannot be instantiated. The system raises an error indicating the validation fails and does not proceed with its instantiation.

As an example, the module in Listing 1 is valid. This can be proven by checking the type of the values produced and consumed in the stack by each instruction in the `$mult` function. Validation will be explained in more detail in Section 3.6.1 but, as an overview: each `local.get` gets a value from a target local variable and pushes it to the stack, `i32.mul` consumes two `i32` values from the stack and pushes their multiplication to it as another `i32` value, and `global.set` consumes a value from the stack and sets the value of a target global variable to it. The type annotation of `$mult` states that it takes two parameters and it produces no results over the stack. As the instructions satisfy its type annotation, the function is valid. This is the only function in the module, hence, the module is valid.

2.1.3.3. Module execution

The execution of a module is divided into two parts: instantiation and invocation. The first is executed immediately after the module is proven to be valid. Instantiating a module means that imported external elements are loaded into the system environment and, if a `start` function is defined, it is invoked.

During the invocation phase, the module is already set up and its functions can be invoked by module users within the system, for example, a JavaScript method.

2.1.4. Verification of WebAssembly programs

The formalization of WebAssembly semantics has already been addressed by previous works. First, in the Wasm specification repository, the authors provide an implementation of the semantics in OCaml [41]. This was first mentioned in the original Wasm paper [15] as a useful way to test Wasm programs.

Moreover, in [37, 39], the authors provide a proof of correctness of WebAssembly 1.0 semantics by a mechanization in Isabelle/HOL and Coq. A mechanisation is a way of proving well-definedness of the semantics by using theorem provers. After this work, different verification frameworks have been developed for WebAssembly. As mentioned in Chapter 1, developers at $\mathbb{K}$ actively maintain a production-ready framework based on the formal semantics of the language [19]. Also, Iris-Wasm [29] is an implementation of the semantics of Wasm in the Iris [18] framework, based on separation logic, which allows proving the correctness of programs relying on a proof assistant.

Furthermore, David Munuera Mazarro, a former student of this Master's program, developed for his Master's thesis a verification tool for a subset of the WebAssembly specification [27]. He follows an approach based on Hoare logic [16] to generate verification conditions, and relies on an SMT solver to check their satisfiability.

2.2. Rewriting logic

Rewriting logic is a formalism for the specification of concurrent and non-deterministic systems based on term rewriting.

A signature (S, Σ) is defined by a set of types (known as *sorts*) S and an $S^* \times S$-sorted family Σ of operations over sorted elements such that:

$$\Sigma = \{\Sigma_{s',s} \mid s' \in S^*, s \in S\}$$

An operator $f : s' \to s$, where s' is a (possibly empty) sequence of arguments $s_1 \cdots s_n$ with arity n, may construct terms in the specification. Terms are typed (sorted) and may

contain variables from an S-sorted family of variables $X = \{X_s \mid s \in S\}$. Then, the term algebra is an indexed S-family $T_\Sigma(X) = \{T_{\Sigma,s}(X) \mid s \in S\}$ inductively defined as:

- $X_s \subseteq T_{\Sigma,s}(X)$, i.e., the variables are terms
- If $f \in \Sigma_{s',s}$ where s' is the empty sequence, then $f \in T_{\Sigma,s}(X)$. These terms are called *constants.*
- If $f \in \Sigma_{s',s}$ where s' is not the empty sequence and, for i from 1 to n, the terms $t_i \in T_{\Sigma,s_i}(X)$ then $f(t_1, \ldots, t_n) \in T_{\Sigma,s}(X)$.

Moreover, an order-sorted signature is defined as a signature (S, Σ) where sorts can be ordered in a hierarchy with the relation $\leq_S$. We say that s_1 is a subsort of s_2 if $s_1 \leq_S s_2$, which means s_1 is the sort of a subset of elements with sort s_2. Although they will not be significantly used in this project, it is relevant to mention that *kinds* are the connected components of the subsort relation, and, for a sort s, its kind is denoted by $[s]$.

An order-sorted equational theory (Σ, E) can be built with an order-sorted signature Σ and a set E of equational formulas. Σ-equations are possibly conditional sentences of the form:

$$t = t' \quad \texttt{if} \bigwedge_i u_i = u_i'$$

where $t, t', u_i, u_i' \in T_\Sigma(X)$.

On top of this, membership equational theories (Σ, E) improve order-sorted equational theories by including in E possibly conditional membership axioms of the form $t : s$, which claim that a term t has sort s. Therefore, these theories have two kinds of formulas [23]:

$$t = t' \quad \texttt{if} \bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j \qquad \textit{and} \qquad t : s \quad \texttt{if} \bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j'$$

Notice that conditions for the formulas can be expressed by other equations or membership axioms. In addition, although they can be expressed as equations, it is worth mentioning that operators can have annotations for *structural axioms* such as commutativity, associativity and identity.

The set of *positions* of a term $t \in T_\Sigma(X)$ is defined as a set $\mathcal{P}os(t)$ of strings such that [2]:

- If $t = x \in X$, then $\mathcal{P}os(t) := \{\epsilon\}$, where ϵ is the empty string
- If $t = f(t_1, \ldots, t_n)$, then $\mathcal{P}os(t) := \{\epsilon\} \cup \bigcup_{i=1}^{n} \{i \cdot p \mid p \in \mathcal{P}os(t_i)\}$

In addition, we denote by $t|_p$ the subterm of t at position $p \in \mathcal{P}os(t)$, and by $t[u]_p$ the term that results from replacing the subterm at position $p \in \mathcal{P}os(t)$ in t by $u \in T_\Sigma(X)$ [2].

A *substitution* is a sort-preserving function $\sigma : X \to T_\Sigma(X)$ that maps variables to terms. It can be extended to a function over terms $\overline{\sigma} : T_\Sigma(X) \to T_\Sigma(X)$ that is inductively defined by $\overline{\sigma}(x) := \sigma(x)$ and $\overline{\sigma}(f(t_1, \ldots, t_n)) := f(\overline{\sigma}(t_1), \ldots, \overline{\sigma}(t_n))$ [2]. Moreover, the composition $\sigma'\sigma$ of two substitutions σ and σ' is defined by $\sigma'\sigma(x) := \overline{\sigma'}(\sigma(x))$ [2]. By convention, both σ and $\overline{\sigma}$ will be referred to as σ.

Then, the reduction step $t \to_E t'$ (t *rewrites* to t' under E) is defined if there is an equation:

$$u = v \quad \texttt{if} \ \bigwedge_i a_i = a'_i \wedge \bigwedge_j b_j : s_j$$

and there exist a position $p \in \mathcal{P}os(t)$ and a substitution σ that satisfies that, for all i, $\sigma(a_i) =_E \sigma(a'_i)$ and, for all j, $\sigma(b_j) \in T_{\Sigma,s_j}(X)$, such that $t|_p = \sigma(u)$ and $t' = t[\sigma(v)]_p$, where $u, v, a_i, a'_i, b_j \in T_\Sigma(X)$ and s_j is a sort [2]. The definitions of $\rightarrow_E$ and $=_E$ over terms in $T_\Sigma(X)$ are mutually recursive such that the latter is the reflexive, transitive, and symmetric closure of the former, i.e., the least equivalent relation that contains $\rightarrow_E$ [2]. We define the reduction $t \rightarrow_E^* t'$ as the reflexive and transitive closure of $\rightarrow_E$, so that t rewrites to t' in 0 ($t =_B t'$, where B is the set of structural axioms) or more applications of equations in E (steps). In the latter case, the reduction is of the form $t \rightarrow_E t_1 \rightarrow_E \cdots \rightarrow_E t_n \rightarrow_E t'$.

Besides, $\rightarrow_E$ must be *confluent*, meaning that: if $t \rightarrow_E^* t_1$ and $t \rightarrow_E^* t_2$, then there exist some term t' such that $t_1 \rightarrow_E^* t'$, and $t_2 \rightarrow_E^* t'$. Moreover, E must be *terminating*, which means that there is no infinite sequence of steps $t \rightarrow_E t_1 \rightarrow_E \cdots$ from any term t. When E is confluent and terminating, t can be reduced up to its irreducible form, also known as, *canonical form*. Finally, the quotient of $T_\Sigma(X)$ by the relation $=_E$ yields equivalence classes $[t]_E$ of a term $t \in T_\Sigma(X)$ modulo E. However, we will use t when referring to representative terms of the equivalence class, which may be in canonical form.

Definition 1 ([2]) *A rewrite theory $\mathcal{R}$ is composed by a tuple (Σ, E, R), such that R extends the equational theory (Σ, E) with possibly conditional rewrite rules acting over equivalence classes of terms in $T_\Sigma(X)$, modulo E. Similarly to $\rightarrow_E$, rewrite steps of the form $t \rightarrow_R t'$ are defined if there is a rule:*

$$u \Rightarrow v \quad \texttt{if} \ \bigwedge_i a_i = a'_i \wedge \bigwedge_j b_j : s_j \wedge \bigwedge_k w_k \Rightarrow w'_k$$

and there exist a position $p \in \mathcal{P}os(t)$ and a substitution σ that satisfies that, for all i, $\sigma(a_i) =_E \sigma(a'_i)$ and, for all j, $\sigma(b_j) \in T_{\Sigma,s_j}(X)$, and, for all k, $\sigma(w_k) \rightarrow_R^* \sigma(w'_k)$, where $w_k, w'_k \in T_\Sigma(X)$, such that $t|_p = \sigma(u)$ and $t' = t[\sigma(v)]_p$.

In [20, 24], the authors argue that rewriting logic can be seen as an equivalence between logic and computation in such a way that, defining a state of a system (whether it is sequential or concurrent), rewrite rules can be interpreted as inference rules, terms as formulas, and computations as proofs.

A relevant property of rewriting logic is *reflection*, i.e., this logic can represent itself. This means that, there exist a universal rewrite theory $\mathcal{U}$ that can represent, as terms, any finitely presented rewrite theory, their terms, sorts, equations, rules, and deductions [25]. This allows creating a *metalevel* in the logic, where the theory is an object as well and, therefore, can be used by operators of its own (or other) logic. Formally, Meseguer defines this in [25] as:

$$\mathcal{R} \vdash t \rightarrow t' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \rightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle$$

where $\mathcal{R} \vdash t \rightarrow t'$ is a rewrite step from term t to term t' under the rewrite theory $\mathcal{R}$. And $\overline{t}$, $\overline{t'}$, and $\overline{\mathcal{R}}$ are, respectively, the representations of t, t', and $\mathcal{R}$ in the rewrite theory $\mathcal{U}$. Besides, this property creates a sequence of representations of rewrite theories that allows proving metalogical properties and performing metalevel reasoning along the chain.

```
fmod CALCULATOR-OPS is
    protecting INT .
    sort Value .

    op val : Int → Value [ctor] .
    ops add sub div : Value Value → Value .

    vars I1 I2 : Int .
    var NzI : NzInt .
    eq add(val(I1), val(I2)) = val(I1 + I2) .
    eq sub(val(I1), val(I2)) = val(I1 - I2) .
    eq div(val(I1), val(NzI)) = val(I1 quo NzI) .
endfm
```

Listing 2: Maude functional module example

Rewriting logic has been applied to the verification of programming languages like C [10], Java [13], Erlang [12], and Verilog [22], among others, based on their formal specification. In [25], Meseguer reviews some of the studies carried by research groups around the world about the applications of this framework on programming languages. Examples of languages that support rewriting logic are ELAN [4], CafeOBJ [7], $\mathbb{K}$ [31] and, as mentioned before, Maude [6].

2.3. Maude

Maude is a programming and specification language based on membership equational logic and rewriting logic. It originates from the previous algebraic specification language family OBJ [14], and aims to provide a framework for simple, expressive and high-performance specification. More specifically, applications of Maude include sequential or concurrent software, specification and formalization of protocols, and logic systems. For that reason, in this project, we formalize the semantics of WebAssembly in Maude.

2.3.1. Structure and syntax

Maude is organized in modules, which can be functional, strategy, or system modules. The former specifies an equational theory and the latter extends its functionality to rewriting logic, following the definitions in Section 2.2. Therefore, both modules may declare sorts, operators, and equations, but only system modules may include rewrite rules. A simple functional module is shown in Listing 2.

First, this module imports `INT`, from the predefined modules in Maude, which includes the definition of integer values. Then, it defines several sorts for specifying a simple calculator, as well as their subsort relations. Operators are specified with the keyword `op`, and can be marked as constructors with the `[ctor]` attribute. In this case, the module defines constructors for the representation of internal values and the operators `add`, `sub`, and `div`, which define addition, subtraction, and division for terms of the sort `Value`, respectively. Then, several variables are created to specify equations. Equations are marked with the keyword `eq`.

The module can be extended by a system module with rewrite rules, modeling the behavior of the calculator. For this, consider the system module defined in Listing 3. The `CALCULATOR` module imports the functional module `CALCULATOR-OPS` described before

```
mod CALCULATOR is
    protecting CALCULATOR-OPS .
    sorts Calculator Op .
    sorts EmptyOpList NeOpList OpList .
    subsort EmptyOpList NeOpList < OpList .
    subsort Op < NeOpList .

    ops add_; sub_; div_; : Int → Op [ctor] .
    op null : → EmptyOpList [ctor] .
    op __ : Op OpList → NeOpList [ctor right id: null] .
    op < _ | _ > : OpList Value → Calculator [ctor] .
    op init : OpList → Calculator .

    var OpL : OpList .
    var I : Int .
    var V : Value .

    eq init(OpL) = < OpL | val(0) > .

    rl [add] : < add I ; OpL | V > ⇒ < OpL | add(V, val(I)) > .
    rl [sub] : < sub I ; OpL | V > ⇒ < OpL | sub(V, val(I)) > .
    crl [div] : < div I ; OpL | V > ⇒ < OpL | div(V, val(I)) >
        if I ≠ 0 .
endm
```

Listing 3: Maude system module example

and specifies the constructors for operations, lists of calculator operations, the calculator state, and three labeled rewrite rules: `add`, `sub`, and `div`. The state is composed by a list of calculator operations and a value to show the result. The rules defined operate over calculator states (terms of the sort `Calculator`), and implement the semantics of the calculator operations. The operations in the operation list are processed sequentially and affect the current value in the calculator at each state. For example, the first rewrite on a state `< add 1 ; sub 2 ; | val(0) >` would produce, with the equations defined before, `< sub 2 ; | val(0 + 1) >`, which reduces to `< sub 2 ; | add(val(0), val(1)) >`. Finally, the operator `init` takes as input a list of calculator operations and produces a calculator state with initial value 0 from where the execution starts.

Both equations and rewrite rules can be conditional, which means they are only applied if a condition is satisfied. The syntax for conditional equations is:

```
ceq <equation> if <condition> .
```

And for conditional rewrite rules is as follows:

```
crl [label] : <rule> if <condition> .
```

An example of conditional rewrite can be found in Listing 3 with the `div` rule. There can be several conditions in the same statement, joined by a conjunction symbol `/\`, and, according to the Maude manual [6], they can have the following forms:

- Equations of the form `t = t'`. When using Boolean equations, it is possible to abbreviate them with a term `t` of kind (general type) `[Bool]`, representing the equation `t = true`.
- Matching equations of the form `t := t'`, where `t` is a pattern and `t'` is a term that will be reduced to canonical form. Pattern `t` usually contains variables, which are instantiated with the matching terms in `t'`.

2.3.2. Commands

Maude includes several commands to execute the specifications declared given some input terms. In this section, some commands relevant for this project will be explained, but the complete list is available in the manual [6].

2.3.2.1. match

This command uses the pattern matching feature in Maude to match a term against a specified pattern. Patterns are terms built from constructors and variables. Matching can have conditions and a number of desired matches can be specified. In the example below, the command tries to match the calculator term

```
< add 1 ; sub 5 ; add 2 ; | val(0) >
```

against the specified pattern `< operations:OpList | v:Value >`:

```
Maude> match < operations:OpList | v:Value >
            <=? < add 1 ; sub 5 ; add 2 ; | val(0) > .
Decision time: 0ms cpu (0ms real)

Matcher 1
operations:OpList --> add 1 ; sub 5 ; add 2 ;
v:Value --> val(0)
```

2.3.2.2. reduce

This command exhaustively reduces a term with the equations defined in the module. As an example, with the module `CALCULATOR-OPS` defined in Section 2.3.1, one can execute the command execution:

```
Maude> reduce add(val(2),val(3)) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Value: val(5)
```

2.3.2.3. rewrite

Unlike `reduce`, the `rewrite` command first normalizes the terms, then it applies the rewrite rules from the modules one after the other until it reaches a final state. As an example consider the following execution:

```
Maude> rewrite init(add 3 ; sub 1 ; add 10 ;) .
rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
result Calculator: < null | val(12) >
```

2.3.2.4. erewrite

This command performs rewrites a term, similarly to the previous command, but allowing interaction with external objects. This command is useful for programs that load data from external files or communicate with processes.

2.3.2.5. search

This command performs a search in the rewrite graph defined with the rewrite rules in the modules. It takes an initial term, from where to start the search, and a final pattern that it will try to match. Moreover, it allows including conditions on the pattern variables, making it a powerful tool to check invariants [6, Chap. 11]. There are several types of search available, including: only one step allowed (`=>1`), any number of steps (`=>*`), one or more steps (`=>+`), only final states (`=>!`), and only states where branching occurs (`=>#`). As an example consider the following execution:

```
Maude> search init(add 1 ; sub 5 ; add 2 ;) =>! c:Calculator .

Solution 1 (state 3)
states: 4  rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
c:Calculator --> < null | val(-2) >

No more solutions.
states: 4  rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
```

In this case, the search is over final states and matches any calculator state. It finds a solution where the resulting value is -2. From this result, one can execute the command `show path labels 3`, where 3 is the last state, to get the labels of the rewrite rules applied until that solution:

```
Maude> show path labels 3 .
add
sub
add
```

Now, consider the case where the target is any state where the value is negative. For that, one would have to specify a pattern that allows checking conditions over the current value:

```
< OpL:OpList | val(I:Int) >
```

And the search must include a condition that reflects that `I < 0`. If one wants to prove a property stating that a certain sequence shall never produce a negative value in the calculator, then, if a search with that condition does not find a solution, the property is proven correct. As an example, the sequence `add 1 ; add 2 ;` shall never produce a negative value:

```
Maude> search init(add 1 ; add 2 ;) =>* < OpL:OpList | val(I:Int) >
            such that I < 0 .
No solution.
states: 3  rewrites: 16 in 0ms cpu (0ms real) (~ rewrites/second)
```

On the other hand, the sequence `add 1 ; sub 5 ; add 2 ;` does reach states with negative values, and the search finds them:

```
Maude> search init(add 1 ; sub 5 ; add 2 ;)
            =>* < OpL:OpList | val(I:Int) >
```

```
              such that I < 0 .

Solution 1 (state 2)
states: 3  rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
OpL --> add 2 ;
I --> -4

Solution 2 (state 3)
states: 4  rewrites: 14 in 0ms cpu (0ms real) (~ rewrites/second)
OpL --> null
I --> -2

No more solutions.
states: 4  rewrites: 14 in 0ms cpu (0ms real) (~ rewrites/second)
```

2.3.3. Meta level

Maude effectively implements the rewrite logic reflection property in the predefined module `META-LEVEL` [6]. This module allows metarepresenting modules and theories, as well as executing meta-commands over them. Metarepresentations are terms, which can be metarepresented again, and, as mentioned in Section 2.2, they can create a sequence of reflection levels. In Maude, one can go up or down in that sequence by using the functions `upTerm` and `downTerm` in the `META-LEVEL` module. The first, takes a term and returns its metarepresentation, and the second, takes a metarepresentation $\overline{t}$ and a term t'. If there exists a term t such that its metarepresentation $\overline{t}$ is in the kind of t', it returns t, otherwise, it returns t'. As an example, we can import `META-LEVEL` in the `CALCULATOR` module. Then, consider the following executions:

```
Maude> red upTerm(add 1 ;) .
reduce in CALCULATOR : upTerm(add 1 ;) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result GroundTerm: 'add_;['s_['0.Zero]]
```

The result of `upTerm` with the addition operation is the canonical form of the term. One can get that term and use it as parameter in `downTerm` along with a term of the same sort, obtaining the original addition operation:

```
Maude> red downTerm('add_;['s_['0.Zero]], opVar:Op) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Op: add 1 ;
```

Furthermore, the `META-LEVEL` module declares functions `metaReduce`, `metaRewrite` and `metaSearch` to execute the analogous commands with metarepresentations of terms over a module.

To metareduce a term `add(val(2),val(3))`, the metarepresentations of the module and the term are given as parameters. The metarepresentation of the module can be obtained with `upModule`, which takes as parameters the module identification and a Boolean value that controls whether module importations are expanded or just indicated. As it is shown below, the result is given in a pair containing the resulting term in canonical form and its sort.

```
Maude> red metaReduce(upModule('CALCULATOR, false),
                      upTerm(add(val(2),val(3)))) .
rewrites: 5 in 0ms cpu (0ms real) (~ rewrites/second)
result ResultPair: {'val['s_^5['0.Zero]], 'Value}
```

To metarewrite a term `init(add 1 ; add 2 ;)`, additionally to the module and term, the descent function `metaRewrite` gets as parameter a `Bound` for the number of rewrites, which is set to `unbounded` for this example. As shown below, the result is given by a pair of the same form as for `metaReduce`.

```
Maude> red metaRewrite(upModule('CALCULATOR, false),
                       upTerm(init(add 1 ; add 2 ; )),
                       unbounded) .
rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
result ResultPair: {'<_|_>['null.EmptyOpList, 'val['s_^3['0.Zero]]],
                       'Calculator}
```

Notice the command used is `red`, the abbreviation of `reduce`, but it is still applying rewrite rules. This is evidence that the search is being performed internally, at the meta-level.

For the case of metasearching, the example presented below corresponds to the last example in Section 2.3.2.5. The search obtains any state such that the value in the calculator at that point is negative. The `metaSearch` function takes as parameters: the module, the initial state, the resulting state, the type of the search, the bound, and the number of the solution (starting at 0). Therefore, to obtain the first solution, one could execute:

```
Maude> red metaSearch(upModule('CALCULATOR, false),
                      upTerm(init(add 1 ; sub 5 ; add 2 ; )),
                      upTerm(< OpL:OpList | val(I:Int) >),
                      upTerm(I < 0) = 'true.Bool,
                      '*, unbounded, 0) .
rewrites: 15 in 0ms cpu (0ms real) (~ rewrites/second)
result ResultTriple: {'<_|_>['add_;['s_^2['0.Zero]],
                                'val['-_['s_^4['0.Zero]]]],
                      'Calculator,
                      'I:Int <- '-_['s_^4['0.Zero]] ;
                      'OpL:OpList <- 'add_;['s_^2['0.Zero]]}
```

The result is a triple containing: the resulting term, its sort and the corresponding substitutions on the resulting term (for `I` and `OpL`). If no solution is found, then the term is result to the `failure` constant. This also happens when the solution number indicated is greater than the number of solutions of the search. In this example, we know from the previous section there are only 2 solutions, so, if the third is selected (with number 2), the result is as follows:

```
Maude> red metaSearch(upModule('CALCULATOR, false),
                       upTerm(init(add 1 ; sub 5 ; add 2 ; )),
                       upTerm(< OpL:OpList | val(I:Int) >),
                       upTerm(I < 0) = 'true.Bool,
                       '*, unbounded, 2) .
rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
result ResultTriple?: (failure).ResultTriple?
```

Chapter 3

Description of the semantics

This chapter contains a detailed description of the semantics developed for the Master's thesis. It will show the main aspects of the implementation, explaining the intuition behind the design decisions, as well as discussing key ideas from the specification of WebAssembly.

3.1. Overview of the semantics

The goal of the semantics presented in this work is to formally specify the behavior of WebAssembly programs in Maude, and to create a framework in which these programs can be executed and analyzed according to the specification. The first step to do so is to represent WebAssembly modules as terms in Maude. For this, we define sorts and constructors in Maude to represent different components in a module (type definitions, functions, and global variables), and the elements that compose them (e.g., instructions). Besides, we also represent the components of the Wasm execution environment as terms, including data structures and the execution state, with which we are able to model the execution of Wasm programs.

As described in Section 2.1.1, Wasm requires a value stack, and stores for variables and allocated components. The latter are represented with `Map` structures, which are dictionary structures in Maude. Moreover, Wasm values are represented in Maude using the predefined Maude sorts `Int` and `Float`, for integer and floating-point values, respectively. A detailed description of the representation of the data types and data structures is shown in Section 3.2.

The execution state represents the environment where WebAssembly modules are processed and executed. It includes the allocated module components, the stack, and the execution frame. The specification of the execution state is detailed in Section 3.3. Also, Wasm modules can be directly specified as Maude terms or read from WAT files using a module parser described in Section 3.4.

WebAssembly semantics are presented in the official specification document [40] as a small-step semantics, defining the behavior of each instruction with one (or more) rules. Each rule defines only one execution step and is specified as a reduction rule of the form [40]:

$$state \quad \hookrightarrow \quad state'$$

where the left-hand side of the rule is an execution state that, when matched, reduces to the state in the right-hand side.

As an example, consider the `i32.div_u` instruction, which corresponds to the division of two unsigned integers. As it is a binary operation, its semantics are specified by the following rules [40]:

$$\begin{array}{lcll} (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop & \hookrightarrow & (t.\mathsf{const}\ c) & (\text{if } c \in binop_t(c_1, c_2)) \\ (t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop & \hookrightarrow & \mathsf{trap} & (\text{if } binop_t(c_1, c_2) = \{\}) \end{array}$$

where $(t.\mathsf{const}\ c_1)$ $(t.\mathsf{const}\ c_2)$ are the top two values in the stack, and $binop_t(c_1, c_2)$ corresponds to the binary operation specified in the instruction, in this case the division of the unsigned representations of c_1 and c_2 with type t. In the case of `i32.div_u`, the type t is `i32`. If the result is defined, it is pushed to the stack, otherwise (e.g. division by 0) it raises a trap (or execution error). Notice that neither the stores nor the execution frame appear in the state representations in the rule. This is because, they are omitted in rules that do not interact with them, and so, we do the same when describing their representation in Maude in the following sections.

In this work, each semantic rule is encoded as one (or more) rewrite rules in Maude, i.e., the reduction $\hookrightarrow$ is represented as a rewrite rule $\rightarrow_R$ under the rewrite theory implemented by the Maude modules. These rules are deterministic, as only one rule can be applied at each point in execution. WebAssembly programs consist of a set of instructions that are executed sequentially. Therefore, the execution of these programs in Maude is represented with a succession of applications of the rewrite rules for the instructions, until a state is reached where no rules can be applied. That state will be referred to as a *final state*.

Furthermore, the process from a declaring a module to its execution consists of three sequential phases:

- Allocation: when a module is loaded, its components are interpreted as Maude terms and indexed in the execution state with `Map` structures. This phase is described in detail in Section 3.5.
- Validation: the validity of a module is checked equationally based on validation rules from the WebAssembly specification [40]. A module must be valid to proceed with the execution and, for it to be valid, all its components must be so. Consider the validation of a function: its body must be valid with respect to its type annotation. To check that, all instructions in a function body are checked sequentially against their unique validation rules. As an example, consider again binary operations, whose validation rule is the following [40]:

  $$\overline{C \vdash t.binop : [t\ t] \rightarrow [t]}$$

  where t is the type prefix of the instruction. Therefore, a function containing a binary operation is valid given that:

  - before the instruction is executed, there are at least two values of type t at the top of the stack and,
  - the instruction sequence that follows $t.binop$ is valid with a value of type t on top of the stack

  Section 3.6 contains an in-depth description of the validation phase, including four significant examples of validation rules.
- Execution: a function may be invoked from the loaded module. Section 3.7 contains a detailed description of the semantic rules for each instruction.

This work provides a Maude operator `run-module-func` that, given a WebAssembly module and a function to invoke, generates an initial state from which the execution starts. Allocation and validation are performed equationally and, once they finish, the execution of the function is triggered.

Finally, the Maude code is separated into different modules. Most modules are defined as functional modules, specifying the sorts and operators to represent WebAssembly data structures in Maude. Moreover, the system module `WASMMOD` includes the definition of the execution phases, containing all the rewrite rules for the instructions. In addition, the module `WASM-FILE` extends the semantics to enable reading WAT files directly in Maude. In Appendix A, Figure A.1 shows the dependency graph among all the modules, including the ones imported from the *prelude* file in the Maude library.

3.2. Types, values, and basic data structures

This section describes the basic data structures and the rationale behind how Wasm types and values are represented in the specification of the semantics in Maude.

3.2.1. Types and values

WebAssembly defines 13 different data types: an index type, four numeric types, a vector type, two reference types, and five composite types. Of these, in this implementation only eight are fully implemented and another one is partially supported, which is the vector type. The reason that some are left unimplemented is that they are related with advanced features which are not considered, namely tables and memory. Numeric, vector, and reference types are known as value types. The syntactic annotations for a value types will be referred to as type tokens. Therefore, a value annotated with a type token is a value of the corresponding value type. The types considered here are:

- Index type
- Numeric types (`i32`, `f32`, ...)
- Three composite types: Function type, Result Type, Global type

Index types can use values corresponding to an unsigned integer of 32 bits (`u32`), so they are coded as natural numbers in Maude, using the sort `Nat`. In the context of a Wasm program, they are used to make a reference to variables (both local and global), types or functions. These indexes are used incrementally, for example, to reference the first and second variables defined in a local context and get their values, one would write:

```
local.get 0
local.get 1
```

In Maude, these are implemented as the sorts `ValRef` for indexes referring to variables and `Addr` for referencing internal addresses of module components, like types or functions.

Regarding numeric types, they are a generalization that comprises integer and floating-point numbers (as described in the IEEE 754 standard [17]). Furthermore, both can be represented with 32 or 64 bits. `i32` and `i64` represent integers of 32 and 64 bits respectively and, for the case of floating-point values, `f32` (single precision) and `f64` (double precision). These type tokens are used to mark the type a certain instruction supports or, simply, indicate the type of a value in the stack or in a store.

As we will see in Section 3.7, instruction operating with values of these types are prefixed by the type name. A simple example of this is the "addition" instruction: `i32.add`. In this case, it operates with two values of type `i32` and the result is pushed to the stack as another `i32`, as shown in Figure 2.1.

To refer to type annotations in the Maude implementation, we use the sort `TypeToken` as a general sort for numeric type tokens, `IntTypeToken` for integer type tokens (`i32` and `i64`) and `FloatTypeToken` for floating-point type tokens (`f32` and `f64`). As an example, a value marked with type token `i32` is a value of value type `i32`.

For Wasm values, these types are embedded in the `const` structure, which is composed of the literal value and its type. In the specification of WebAssembly, this is represented with the following syntax:

```
t.const c
```

In that structure, `t` corresponds to the type token and `c` to the literal value. This structure is also used as an instruction to push `c` to the stack. However, without losing generality, in our system, we will distinguish the type-carrying value wrapper from the `const` instruction and refer to this as:

```
const(t, val(c))
```

Here, `val(c)` corresponds to a value wrapper for `c` to generalize the parameter in `const` and its sort is `ValWrapper`. This value wrapper has been defined over the Maude predefined sorts `Int` and `Float`, and extended with special float value definitions like `infinity` and `NaN`. As an example, a value `const(i32, val(0))` represents the value 0 with value type `i32` (32-bit integer).

Maude only provides the `Float` predefined sort for representing floating-point values, which specifies 64-bit floats. A notable restriction in our representation is that, any floating-point value, whether it is marked as `f32` or `f64`, will use that type. The impact of this is the possibility that approximation errors may appear. However, support for `f32` instructions has been fully implemented considering the underlying 64-bit representation, so in future iterations on this work this could be adapted. Also, the function `correctF` has been defined to minimize this problem, which keeps the value within the range of the corresponding type. It also ensures the correct encoding of 0.0 as `zero(pos)`, where `pos` indicates the positive sign of the value. This is due to the standard IEEE 754 having two definitions for 0.0: $+0.0$ and -0.0, used by float operations. For that reason, in this implementation sign for floats is encoded as `pos` for `+` and `neg` for `-`.

Moreover, an `Unknown` type is defined strictly for the validation phase. As a general idea, it is used to represent polymorphism in the stack, but it will be explained in more detail in Section 3.6.4.

3.2.1.1. Function and block types

In Maude, function types are defined with the sort `ParamResultType`, which allows the optional declaration of parameter and result types, and result types are defined with the sort `ResultType`. Along this document, function types may appear with the notation $[t_1^*] \rightarrow [t_2^*]$. This is used in the specification of WebAssembly to indicate that a function (or code block) receives, as parameters, a list of values whose types are indicated in the list $[t_1^*]$ and returns to the stack a number of values whose types correspond to $[t_2^*]$. The asterisk expresses the possibility to have any number of parameters and/or results (including 0).

Composite types are constructed from numeric types. In the case of function and result types, they are represented with a structure indicating the list of types of the corresponding

input and output, as parameters and results over the stack. A complete definition of a function type follows the syntax:

```
(param TList) (result TList)
```

`TList` represents a list of type tokens, `(param TList)` is a parameter object and `(result TList)` is a result object. Types tokens contained in parameter objects represent the type of the values that are consumed from the stack as inputs to the function and types tokens contained in result objects represent the type of the values that are produced in the stack after its execution. With this, functions may have multiple results, which will be at the top of the stack when the execution of the function ends. It is possible to define a function type with any number of parameter or result objects but, in the case of result types, only a result object shall be present.

It is possible to specify the type of functions and instruction blocks with a reference to a type specified in the module. In fact, the type of a block (a `BlockType`) is translated to a type reference during the allocation phase, as will be described in Section 3.5.3.2. However, specifying a `BlockType` is optional and, if it is not specified, the system considers its type as an empty result type.

3.2.1.2. Global types

Finally, global types represent the type of a global variable, which can have a numeric type and an indicator on whether it is modifiable (`mut` or `var`) or constant (`const`). Therefore, mutable global types are specified as `mut t` and constant global types as `const t`, where `t` is a type token.

Global types are implemented in Maude by defining a sort `GlobalType`, which, in case the variable is modifiable, it is specified with subsort `MutGlobalType`.

3.2.2. Labels

In the context of the WebAssembly specification, a label is a structure that marks a relevant spot in execution. They are introduced once an instruction block is executed or when a function is invoked, separating the block from the previous instruction list. Labels are pushed to the value stack along with values, and they contain information about the next instructions to be executed, so that execution can resume once a block is finished or skipped with a jump. There are specific mechanisms defined in the specification for those cases, which allow retrieving the instruction sequence from the corresponding label, as there might be more than one in the stack at a certain point in execution. In this implementation, they contain the next instructions to be executed after a jump, the arity of the results of the block (the number of result values) and a tag, marking the origin of the label (a block, a loop, or a function).

3.2.3. Basic data structures

This section details some of the data structures of the WebAssembly semantics in Maude. Namely, the most notable structures are the value stack, the value and index reference stores, and the module component stores.

3.2.3.1. Stack

A stack is a basic data structure commonly used in Computer Science. As an overview, it is characterized by following a LIFO (Last In First Out) approach and the two main operations over them are `push` and `pop`. The first, given a stack and a value, then the value is appended at the top of the stack, becoming the new top. With pop, the top of the stack is removed and the next element in the stack is the new top.

WebAssembly is stack-based in the sense that its instructions rely on a stack to perform their operations. Although Wasm also counts with stores and variables, its instructions pop values from a stack to use them as operands and push their results back into the stack. Therefore, the stack is a key structure in Wasm. Along this document, this stack will be referred to as "value stack", "execution stack" or, simply, "the stack".

To define the main Wasm stack in Maude, we specify the sort `StackElem` to represent stack elements, which can be labels and values such that the following sort relation is satisfied:

```
subsorts EmptyStack NeStack < Stack .
subsorts Value Label < StackElem < NeStack .
```

Then, the structure can be constructed either as an empty stack (`EmptyStack`) or non-empty stack (`NeStack`) with the following constructor operators:

```
op EmptyStack : → EmptyStack [ctor] .
op _$_ : StackElem Stack → NeStack [ctor id: EmptyStack] .
```

The first is just a constant corresponding to the identity element, and the second is constructed by a sequence of `StackElem` instances separated by the dollar sign (`$`). To push a stack element `E` to stack `S`, we use the second operator so that `E $ S` is a new stack containing the elements in `S` and `E`.

3.2.3.2. Stores

During execution, it is needed to be able to store variables and system objects in order to be able to invoke them, get their values, or update them. In Maude, we use the parametric sort `Map{X :: TRIV, Y :: TRIV}`, which is provided in the prelude file from the Maude library. This structure allows mapping keys of sort `X` to values of sort `Y`. Therefore, this allows inserting and updating values indexed by key and accessing values by key.

For this semantics, several stores are needed:

- Value stores: these stores map keys of sort `ValRef` to values of sort `Value` with `Map{ValRef, Value}`. They are used to store local variables and their current values in a local environment.
- Index stores: these stores map module component names to their corresponding reference index. Therefore, the definition of this map is `Map{String, Addr}`, where the name is a string and the reference address is of sort `Addr`.
- Component stores: in this case the indexes of the previous store type are mapped to the components themselves. These stores are used to store the complete information of the module in case they are referenced during the execution. An example of this would be a function whose type references a type defined at module level. The definition of these stores is `Map{Addr, ModuleContent}` for general module components, and `Map{Addr, GlobalInstance}` for global variables after processing.

3.3. Execution state

In order to perform the execution of the program following the semantics, first, we need to shape the state of the system into a structure which allows representing the changes that the semantic rules entail. The execution state is built incrementally with its components, starting from a local execution environment, adding global variables and, finally, including module components like types definitions and functions.

3.3.1. Local environment

A local environment defines the execution frame of a function. That is, where its instruction list is executed. Instructions are executed sequentially and, in this environment, they may interact with local variables and the execution stack. This environment contains

- An instruction list
- The execution stack
- A local variable store

However, a function may call another function (or itself, recursively). Consider the case of a function f_1 calling a function f_2. If we only used one local variable store, the function f_2 could access the local variables of the function f_1 or, if we reset the store when entering the execution frame of f_2, once the execution f_2 had finished and returned to finish that of f_1, we could not restore the variables for f_1. Instead, we define the sort `WASMLocalEnv` with a list of local variable stores to represent the local environment of functions and possibly stacked execution frames. Every time a function is called, a new local store is prepended to the list of local stores, representing the local store of the execution frame of the new function. And, when the execution of a function ends, its local store is removed from the list. This will be shown when discussing the semantics of the `call` instruction, in Section 3.7.4.7. In Maude, `WASMLocalEnv` is constructed with the following operator:

```
op <_|_|_> : List{Inst} Stack List{Map{ValRef, Value}}
          → WASMLocalEnv [ctor format (ni d d d d d d)] .
```

Instructions that do not interact with global variables, functions or other module components, operate exclusively at this level, as will be shown when discussing them in Section 3.7.

3.3.2. Global environment

A global variable instance is a structure containing the key characteristics of a global variable, namely, its type and current value. It is represented with a tuple of the form:

```
{type <GlobalType>, value <Value>}
```

where `<GlobalType>` constitutes the type of the global variable as defined in Section 3.2.1 and `<Value>` is the current value stored for the variable, which must be coherent with its type.

This environment extends the local environment to support global variables. To do this, it includes an index store for global variables names (maps names to indexes), and a component store where the indexes are mapped to global variable instances. This is organized in a structure named `GlobalInventory`, which is appended to the previous definition of `WASMLocalEnv`, creating the `WASMGlobalEnv` as shown below.

```
op _<->_ : Map{String, Addr} Map{Addr, GlobalInstance}
            → GlobalInventory [ctor] .
op _ # _ : WASMLocalEnv GlobalInventory
          → WASMGlobalEnv [ctor format (d ni ni d)] .
```

Within this environment, instructions which interact with global variables (`global.get` and `global.set`), can be executed.

3.3.3. Module components support

Component stores and their corresponding index stores are appended to the global environment. This allows referencing type definitions and calling functions during execution. In order to make results more readable, index stores are included in separated structures indicating the module component they are used for. These structures will be referred to as "context", and are defined with the form: `{<indicator> Map{String, Addr}}`, where `<indicator>` depends on the module object (type, function, ...). To help understand later references in this document, the indicator of functions is `funcIndCtx`, and for types is `typeIndCtx`.

The definition in Maude for this extension is shown below. Notice that `FuncIndCtx` and `TypeIndCtx` represent the index context structures mentioned before.

```
*** Functions + Global enviroment
op _ # _ : WASMGlobalEnv Map{Addr, ModuleContent}
          → WASMFuncGlobalEnv [ctor format (d ni ni d)] .

op _ _ : FuncIndCtx WASMFuncGlobalEnv
        → WASMFuncGlobalIndEnv [ctor] .

*** Types + Functions + Global enviroment
op _ # _ : WASMFuncGlobalIndEnv Map{Addr, ModuleContent}
          → WASMTypeFuncGlobalEnv [ctor format (d ni ni d)] .

op _ _ : TypeIndCtx WASMTypeFuncGlobalEnv
        → WASMTypeFuncGlobalIndEnv [ctor] .
```

3.3.4. Full state and auxiliary components

To complete the state, we include the module. Moreover, we define an auxiliary operator that will be used internally to initialize the execution.

The code below shows how the complete state is constructed from the previous definitions, where `WASMModule` represents a WebAssembly module. Then, `WasmTrigger` defines an extra structure that will be used to initialize the execution and trigger the starting function. The latter is composed by a `Starter` object, which indicates: the starting function with `start funcID`, where `funcID` is either an index or a function name; a list of counters, one for each module element type; and a sequence of initial instructions. The latter component is optional and, if not specified when executing the module, it is set to `nil`. `WASMEnvTrigger` constructs an internal state which triggers the execution of the module based on the corresponding `WasmTrigger`. However, its terms are reduced to `WASMEnv` terms after the validation phase.

```
op # _ # _ # : WASMModule WASMTypeFuncGlobalIndEnv
              → WASMEnv [ctor format (ni ni ni ni ni ni)] .

op {_, indctr _, initIL _} : Starter ComponentCtr List{Inst}
              → WasmTrigger [ctor] .
```

```
op _ _ : WasmTrigger WASMEnv
              → WASMEnvTrigger [ctor format (ni ni ni)] .
```

3.4. Module parser

We define a WebAssembly module parser as a complementary element to the semantics. The main idea is of this is to transform the contents of modules in WAT files to their representation in Maude, initializing a state, as described in Section 3.3. Modules can be written directly in Maude-readable syntax as will be detailed in Section 5.1.2, without the need for this parser. However, it is still recommended to read this section, prior to the semantic phases, to understand the differences between original modules and their representation in Maude, and how the Maude `META-LEVEL` is used in this project.

The parsing process is split in two steps: first, reading the contents of the file, then, interpreting it as Maude terms. The first step uses the `FILE` module, importing the interface needed to read files. The parser in this project is inspired by the example shown in the Maude manual [6] for copying files using this interface. The difference is that, instead of reading from a file and writing to another, the information read in each line is appended to a string. Once every line is in the string, the parser has to interpret the contents. To do so, the function `tokenize-wasm` splits the string into a list of quoted identifiers in Maude with the built-in function `tokenize`, and transforms the tokens to the syntax allowed by the modules implemented with some particular tokenization rules.

The main syntactic differences between a Wasm module and its representation within Maude are the following:

- Type prefixes in instruction names are separated from the rest of the name. As an example: `i32.const 0` is transformed to `i32 .const 0`, where `i32` is matched as an `IntTypeToken` (integer type token). This allows matching the type tokens easing the validation process for instructions that are restricted for certain types (which will be explained more in depth in Section 3.6.1). Also, it is useful to keep values within range of their precision.
- Analogously to the previous feature, when there are different operations of the same kind of instruction, instruction opcodes are also separated from the instruction name. This allows grouping semantic rules by type of instruction as the WebAssembly specification defines them. As an example: `i32.add` is transformed to `i32 . add`, where `add` is matched as a `BinopToken` (binary operation token).
- The underscore character (`_`) is substituted by the `~` character. This allows reading instructions such as `div_u` or `br_if` in Maude, because, by default, Maude reads that character as a matching placeholder for other tokens.
- Tokens starting with the dollar sign character (`$`) are enclosed between double quotations, to parse them as string values.

Usually, in Maude, every string of characters until a whitespace is considered a single token, but with `tokenize-wasm`, the aforementioned transformations are applied, as shown in the example below:

```
Maude> red tokenize-wasm("i32.const 1 i32.const 2 i32.add") .
rewrites: 169 in 0ms cpu (0ms real) (~ rewrites/second)
result NeQidList: 'i32 '.const '1 'i32 '.const '2 'i32 '. 'add
```

```
Maude> red downTerm(getTerm(metaParse(['WASMMOD],
    tokenize-wasm("
        (module
            (export \"$func0\" (func 0))
            (func $func0 (result i32 f32 i64 f64)
                i32.const 0
                f32.const 0
                i64.const 0
                f64.const 0
            )
        )
        "), 'WASMModule)), errorMod) .
rewrites: 778 in 0ms cpu (0ms real) (~ rewrites/second)
result WASMModule: module (export "$func0" func 0) (
func "$func0" (result i32 f32 i64 f64)
  (local nil)
i32 .const (0).Zero
 f32 .const (0).Zero
 i64 .const (0).Zero
 f64 .const (0).Zero
)
```

Figure 3.1: Meta-parsed module

When the tokens have the correct form, they are interpreted as Maude terms using the `META-LEVEL` functions `metaParse` and `downTerm`. Combined, they interpret the resulting token list from `tokenize-wasm` as terms defined in the module `WASMMOD`. Figure 3.1 illustrates the result of loading a module as a string. As it shows, the module is tokenized with the custom tokenizer function, and the result is the representation of a Wasm module in Maude.

3.5. Module allocation

As mentioned in Section 2.1.2, WebAssembly code is organized in modules, which contain the definition of certain components that may be used during the execution process. This phase occurs after the WAT file is parsed. Here, the module is allocated, meaning its components are stored in an initial execution state.

Allocation is performed equationally with a function `read-module` over the execution state, which declares the semantics of processing an object of each of the module component types in any order they are presented.

Once every component has been allocated and processed, they will be validated to ensure their formal correctness before execution.

3.5.1. Type definitions

A module can contain any number of type definitions, which can be referenced in the code. These definitions contain a reference index or type name, and the corresponding

```
Maude> match (type id:Id funcType:FuncType) <=?
            (type 0 (func (param i32 f32) (result i32))) .

Decision time: 0ms cpu (0ms real)

Matcher 1
id:Id --> (0).Zero
funcType:FuncType --> func (param i32 f32) (result i32)
```

Figure 3.2: Type definition example

function type definition.

3.5.1.1. Syntax

Below, an example of type definition is shown. In this case, it is a full definition, containing both the declaration of parameters and results. However, it would be possible to have a type definition which lacks either one or both of them. If that is the case, the system interprets the lacking component as an empty list of type tokens.

```
(type <identifier> (func (param <TypeTokenL>) (result <TypeTokenL>)))
```

where `TypeTokenL` are type token lists. The components that can be identified in the syntax for this structure are:

- Identifier: To identify a type and be able to reference it in the code, it is compulsory to declare an identifier in its definition. This can be either an index (i.e. natural number) or a name. The latter must be preceded by a dollar sign (`$`) in the WAT file, although it will be transformed and treated as a string internally.
- Function type: It defines a function type as described in Section 3.2.1. In this case, the only syntactic difference is that it is contained in a structure of the form: `(func <FunctionType>)`.

As an example, Figure 3.2 shows a correct type definition, which is checked with the built-in pattern matching command in Maude.

3.5.1.2. Allocation

When allocating type definition from a module, two cases have to be considered: when the identifier is an index (which is a straightforward step) and when it is a name (a `string`, internally).

For the first case, the type definition is directly inserted into the `TypeStore`, which will collect all the type definitions indexed by a counter. The counter corresponding to types is kept in the counter list mentioned in Section 3.3.4, and it is incremented when processing each new type.

For the case where the type is identified by a name, the same process is followed considering the store of the type and the counter update. However, it is also needed to store the identifier name mapped to the counter value in the index context for types.

```
Maude> match (global Id:String (GbType:GlobalType)
                 (ConstInst:ConstInst))
            <=? global "$gb1"(mut i64)(i64 .const (1).NzNat) .

Decision time: 0ms cpu (0ms real)

Matcher 1
Id:String --> "$gb1"
GbType:GlobalType --> mut i64
ConstInst:ConstInst --> i64 .const (1).NzNat
```

Figure 3.3: Global definition example

3.5.2. Global variables

Global variables are declared and initialized at module level, meaning that this type of variables cannot be created within code fragments. Moreover, the type of a global variable is immutable and defined with this declaration.

3.5.2.1. Syntax

Below, an example of type definition is shown. In this case, it is a full definition, containing both the declaration of parameters and results. However, as with type definitions, it would be possible to have a type definition which lacks either one or both of them. If that is the case, the system interprets the lacking component as an empty list of type tokens.

```
(global <identifier> (<GlobalType>) (<ConstInst>)))
```

The different components in this structure are:

- Identifier: as type definitions, global variable declarations must be associated to an identifier. Global variables are identified with names, which are preceded by a dollar sign (`$`) in the WAT file. In the same way as type definitions, these names are treated as string values for internal computations.
- `GlobalType`: is the type of the global variable, following the structure detailed in Section 3.2.1. It controls whether the variable is mutable or constant and the type of its values.
- `ConstInst`: this corresponds to a type of instruction, which is also a value representation as mentioned in Section 3.2.1: the `t.const` instruction. It is the initialization of the variable.

In the same way as with type definitions, Figure 3.3 shows the definition for a global variable in a Wasm module, destructured with the pattern matching command in Maude.

3.5.2.2. Allocation

Similarly to type definitions, the allocation process for global variables is defined by cases in the `read-module` function.

For this component, four cases are distinguished by considering the initial value, three of which consider the combinations for numeric values with numeric types and one for the vector type. For all cases the semantics indicates that:

- the identifier is mapped to the global variable counter
- a global variable instance is created from the variable definition, following the instance structure specified in Section 3.3.2. These instances are created with the type and initial value of the variable.
- the global variable counter value is mapped to the global variable instance
- the global variable counter is incremented

For each case specifically, we consider the possible values for numeric types.

- In the case of integer types, only unsigned integer values within its type range are allowed. That is, a value `IntN` with type `IntT` (either `i32` or `i64` as defined in Section 3.2.1), has to satisfy that:

$$0 \leq \texttt{IntN and IntN} < 2^{\texttt{bitWidth(IntT)}}$$

  where `bitWidth(IntT)` corresponds to the number of bits of a value of type `IntT`.
- Then, if an `integer` is defined with a signed value, the system must get its unsigned representation and store it. If it is out of range, then the execution fails.
- For the case of `float` types with value 0.0, the value is stored using the internal representation for floating-point positive zero (`zero(pos)`).
- Regarding `float` types with a float, the value is corrected and adapted to internal float representation with the function `correctF`.
- Finally, for `float` types declared with `integer` values, the value is converted to `float` and it is processed as such. In this case, the counter increase and instance storage is not directly performed, as it will be done when processing the value as a `float`.

3.5.3. Functions

Finally, Wasm modules may define functions. They have a functional type (parameters and results), local variable definitions, and a list of instructions as its body.

3.5.3.1. Syntax

Similarly to the previous components, the syntax of a function definition is shown below. In that schema, all components are included, but it should be noted that all of them, except for the identifier, are optional. This is due to different functions having different numbers of parameters and results, as well as different number of local variables.

```
(func <identifier> <FuncTypeDef>
    <Locals>
    <InstList>
)
```

```
Maude> match (func Id:String Type:FuncTypeDef
                Locals:LocalsRef
                InstList:List{Inst}
            )
            <=?
            func "$mult" (type "$type1")
                (local i32)
                local.get 0   local.get 1   i32 . mul   .

Decision time: 0ms cpu (0ms real)

Matcher 1
Id:String --> "$mult"
Type:FuncTypeDef --> type "$type1"
Locals:LocalsRef --> local i32
InstList:List{Inst} --> local.get 0
 local.get 1
 i32 . mul
```

Figure 3.4: Function definition example

Formally, we can separate the components in the following way:

- Identifier: As for other components, function identifiers are compulsory and must be preceded with a dollar sign (`$`) in the module definition. Internally, analogously to other identifiers, they are treated as Maude string values.
- FuncTypeDef: Similarly to the `FuncType` component present in type definitions, this component declares the type of the function based on its inputs (parameters) and outputs (results), if any. In particular, the type of a function can be defined explicitly with a `FuncDef` expression, or by reference to a previously defined type. The latter is defined in the specification as `TypeRef`. This component is optional, if it does not appear then, empty lists are implicitly assumed for both parameters and results.
- Locals: An optional component which defines the types of the local variables used in the local environment of a function. These variables are additional to the parameters defined.
- InstList: This component defines the list of instructions in the body of the function.

As before, Figure 3.4 shows an example of a simple function definition, destructured with the built-in pattern matching commands in Maude against the syntax defined here. Also, as mentioned for global variable definitions, this specification considers the all the components of the functions are represented with Maude terms.

3.5.3.2. Allocation

Function allocation is, again, performed with the `read-module` function. In this case, this step is simpler than for other module components due to the fact that functions will

be validated more thoroughly later. Firstly, the semantics of allocating a function require that the type of the function is coherent. This means, it is correctly allocated if one of these three cases is satisfied:

- The function type is defined explicitly.
- The function type is defined by reference to a previously defined type.
- The function type is defined both explicitly and by reference to a type which matches the explicit definition.

If the type is coherent, then the function is defined in two ways, depending on whether it is identified by a number or by a name. In the first case, the semantics are straightforward and the function is directly inserted to the function store (`FuncStore`). In the latter case, the procedure is similar to that of the global variables:

- Map the name to the function counter
- Map the function definition to the current counter value

In both cases, the function counter is incremented, to correctly allocate the next function (if any). Moreover, for every block instruction (`block`, `loop`, and `if`) in the function body, if their type is expressed as an explicit functional type, as described in Section 3.2.1.1, it is substituted by a type reference to an equivalent type. If such a type does not exist, it is generated and allocated, updating the type definition counter. This is a relevant modification as we will only have to consider type references when validating and executing those instructions, as shown in Sections 3.6.3 and 3.7.4.2.

3.6. Module validation and instantiation

Once the module information is loaded into the state term, the system checks the validity of the module according to the rules defined in the specification. From that point, it will be safe to execute code from the model, and the instantiation process will start.

Module instantiation involves handling module imports and exports, including and initializing external components in the execution environment, as well as starting the execution of the module code. Since the interaction between modules is not considered in this implementation, instantiation has been reduced to invoking the start function.

3.6.1. Validation

In the validation phase, the system considers the module context and type-checks the function bodies according to the typing rules for the instructions. This context refers to a structure that stores relevant information regarding components and references of a module [40]. If the module is not valid, the system must not continue to the instantiation step.

Consider a module context **C** containing types, functions, tables, memory components, and global variables, both imported and internal. In addition, **C** contains the module element segments, data components, index reference contexts, and empty lists for local variable types, arity of labels, and return types. These last lists will be significantly used along this section. The first one corresponds to the types of the local variables of an

execution frame, the second one is the list of result types of labels generated in the code, and the last is the list of result types of functions.

For the scope of this implementation, we consider the validation for types, global variables and functions under the context **C**.

According to the specification, every function type is valid, with any number of inputs and outputs. And, regarding global variables, their validity relies on three conditions [40]:

- The type of the definition must be a valid type
- It shall be initialized with an expression with result type equivalent to one value type.
- The expression shall be a constant instruction.

Considering that our implementation in Maude already checks those three points when allocating global variables, we can ensure their definitions are valid at this point. The syntax for global variable definitions declared in Section 3.5.2.1 makes it clear that they can only be defined with constant instructions, which have a result type equivalent to one value type and their global type is valid according to the definition of a global type.

3.6.1.1. Function validation specification

To satisfy function validity, every function body must be consistent with their function type. This process consists of the following steps, executed under the context **C**:

- Get the explicit definition of the function type of the form $[t_1^*] \rightarrow [t_2^*]$
- Copy context **C** to some context $\mathbf{C}'$ containing:
  - A list of types of the local variables, including function parameters and local variable definitions, in that order.
  - The list for arity of the labels as result type $[t_2^*]$
  - A result list equal to result type $[t_2^*]$
- Check, under the new context $\mathbf{C}'$, that the body of the function ends with type $[t_2^*]$. In other words, considering the types of each of the instructions in the body, check that the sequence, as a whole, leaves a type list $[t_2^*]$ on the stack.

If that is satisfied, then the function is valid. If it is not, then it is notified and the module is not instantiated, preventing the execution of invalid code.

3.6.1.2. Function validation implementation

Right after the module allocation phase, the function `validate-funcs-module` is called with the state of the system as parameter. That function checks the validity of every function allocated in the execution state. If a function is not valid according to the specification, then the state rewrites to `Fail("Invalid module")` and the execution stops. On the other hand, if all functions are valid, then the system advances to the instantiation phase, from where the execution starts.

To check all the functions, `validate-funcs-module` makes use of the auxiliary function `verify-funcList` which takes as input the instruction list of the function and the several structures that compose context **C** as defined in Section 3.6.1. These structures are:

- Index contexts for types, functions, and global variables
- Stores for types, functions, and global variables

Then, for each Wasm function, the function `verify-IL` checks its instruction body under context $\mathbf{C}'$, as defined in Section 3.6.1.1. To construct context $\mathbf{C}'$ from $\mathbf{C}$ it needs the following structures as parameters:

- Locals: a list of type tokens formed by concatenating the parameter type list and the local variables type list. It uses an auxiliary function to get each of the token lists from the function definition. This will be referred to as `TList1`.
- Result: a list of type tokens corresponding to the result list of the function type. This list is obtained by another auxiliary function from the function definition. This will be referred to as `TListRes`.
- Labels: A list of result structures including the arity of each label in the program (referred to as `LabelsTList`). Initially, it takes the value of the first result type.

The function `verify-funcList` iterates through all module functions, checking that the instruction list is consistent with the type with which the function is defined, until the list is empty, where it returns `True`.

Now, `verify-IL` checks every instruction sequentially according to the validation rules in the specification. The system uses a temporal type token stack (which will be referred to as `TList` or type token stack) to execute their step-by-step semantics. This stack is initially empty as it simulates the behavior of the stack during the function execution. The function traverses the function body popping and pushing the type tokens corresponding, respectively, to the parameters and results of each instruction to the type token stack. When there are no more instructions left to check, it returns `True`. `verify-IL` is defined by cases, and if, at some point, it cannot execute any of the equations, it means that it is *ill-typed*, and returns `False`.

The specification defines a validation rule for each instruction to check that its type is satisfied. In the following sections, this notation will be used:

$$[t_x^*] \rightarrow [t_y^*]$$

It refers to a function over types, and it is specific for each instruction. A list of values of an specified combination of types ($[t_x^*]$) are consumed from the stack, and a list of values of another specified combination ($[t_y^*]$) are produced at the top of the stack. As a corollary, t_x^* is required to be at the top of the stack for the instruction to be executed. Another important remark is that, to resemble the stack structure defined in Section 3.2.3.1, the leftmost place in a list will represent the top of the stack.

For readability, and to avoid repetitive sections, the validation rules will be described in three groups. The first group includes the simplest instructions, whose validation rules just require an input list of parameters and produce a result list. The second group represents instructions containing an internal instruction sequence, which produce a label. And the last group includes stack-polymorphic instructions, which perform jumps.

3.6.2. Validation of simple instructions

Instructions in this group are numeric, parametric, and variable instructions. Nonetheless, control instructions also use this procedure, but, due to its complexity and extra features, they are explained in the next sections.

When `verify-IL` gets to an instruction of this group, it matches the parameters in the type of the instruction with the top of the type token stack. If the match is successful, they are popped and the result type tokens from the function type are pushed.

Consider the operation `t.const c` as an example, which pushes to the stack a value `c` of type `t`. This instruction is valid with type:

$$[] \rightarrow [\mathsf{t}]$$

This means that it does not consume any value from the stack, but it produces a value of type `t`. Therefore, when `verify-IL` reaches this kind of instruction, the reduction is as follows:

$$\texttt{verify-IL((t.const c) IL, TList,...)} \rightarrow \texttt{verify-IL(IL, t TList,...)}$$

Now, consider binary operations, whose type is defined as follows:

$$[\mathsf{t}\,\mathsf{t}] \rightarrow [\mathsf{t}]$$

In this case, two values will be consumed from the stack (as it is a *binary* operation) and one will be pushed to the stack as a result. The three of them must match type `t` for the instruction to be *well-typed*. In the implementation, two elements `t` are required at the top of the type stack to be consumed. After this instruction, the stack must have one type `t` at the top, while preserving everything but the tokens consumed. The implementation is as follows:

$$\texttt{verify-IL((t.binopT) IL, t t TList,...)} \rightarrow \texttt{verify-IL(IL, t TList,...)}$$

where `binopT` is any binary operation opcode. This token must be defined for the corresponding type `t`.

3.6.3. Block instructions validation

This section details the specification and implementation of the validation of instructions which encapsulate another instruction sequence. These instructions are `block`, `loop`, `if..else` and, in some way, `call`. The latter is included here because, even though it refers to calling a function, which involves different semantic steps than other block instructions, regarding validation, the main idea is to check the instruction sequence in the called function. The type of these instructions has been wrapped and indexed in a type definition during the allocation phase, as described in Section 3.5.3.2.

As an example, consider a `block` instruction, whose type is defined by type reference after allocation. The syntax for this is: `block TypeRef IL2 end`. The instruction list `IL2` inside the block must be valid with the referenced type. This means that, expanding the reference, it should be valid with type:

$$[\mathsf{t}_1^*] \rightarrow [\mathsf{t}_2^*]$$

where t_1^* is the parameters list and t_2^* the result list, defined by the type referenced by `TypeRef`. Therefore, t_1^* are the types of the values that are initially in the stack when validating the body of the block.

As an example, consider we have a module with a type `$type1` defined as the function type `(param i32) (result i32 i32)` (it gets a value of type `i32` as parameter and produces a result of two values of type `i32`), and we are validating the following sequence:

```
i32.const 0
block (type $type1)
    i32.const 0
end
```

where the *expanded type* of the `block` instruction is [i32]→[i32 i32]. The first instruction has type []→[i32], so it pushes a value of type `i32` to the stack. Then, the body of the `block` must be valid with type [i32]→[i32 i32]. This means that, at the end, there must be two values of type `i32` in the stack, considering there is, initially, a value of type `i32`. In the body of the block, the instruction `i32.const 0` pushes another `i32`, and the block ends. Finally, there are two values of type `i32` in the stack, so the sequence is valid. This example can be found in the test suite from the code repository of the project [21].

Then, the implementation requires expanding the type reference and extracting both the parameters and the result lists. Moreover, the type of the parameters of the block is used as the initial temporal type stack for the validation of the body of the block. The validation of this instruction is implemented as follows:

```
          TList' == getParamList(expT)
          and verify-IL(IL2, TList',...,getResultList(expT),
                ...,(getResultList(expT), LabelsTList))
-------------------------------------------------------------------------
verify-IL(block TypeRef IL2 end) IL, → verify-IL(IL,
  TList' TList,...                       getResultList(expT) TList,...
  LabelsTList)                           LabelsTList)
```

where `expT` corresponds to the expanded type. To get that representation, the system accesses the type index store and the type store. Therefore, `getParamList(expT)` and `getResultList(expT)` represent the parameters and result of the function type respectively.

3.6.4. Stack-polymorphic instructions validation

Finally, this section describes validation for instructions that perform program jumps. A special concept appears at this point in the official specification is *stack-polymorphism*, which defines unconstrained types both for inputs and outputs [40]. Therefore, some control instructions are valid with some form of the type:

$$[t_1^*] \rightarrow [t_2^*]$$

where both $[t_1^*]$ and $[t_2^*]$ are unspecified type lists. However, the whole sequence must still be well-typed, so these instructions must be typeable under some type coherent with the rest of the sequence [38].

Instructions of this type involve program jumps, like `br` i and `br_if` i. To show this with an example, consider the validation for `br` i, where i in an index. As it is polymorphic in the stack, the result type can be any type. Therefore, the sequence of instructions that follows it has to be validated considering a sequence of values of an unknown combination of types in the stack. First, this instruction is valid with type [40]:

$$[t^* \, t_1^*] \rightarrow [t_2^*]$$

where t^* is defined by the result type of the i-th label in the labels list, and both t_1^* and $[t_2^*]$ are unknown.

To be able to keep validating the rest of the instruction sequence in the function, the result $[t_2^*]$ shall be represented with some template type, which will be referred to with the `Unknown` token. It works as a wildcard for any combination of numeric types and `verify-IL` considers the cases where `Unknown` is in the temporal type stack for every instruction. In this implementation, the instruction will be valid with type:

$$[\mathsf{t}^*\ \mathsf{t}_1^*] \rightarrow [\mathsf{Unknown}]$$

Then, the condition for the instruction to be valid is that `Unknown` represents a type that allows the rest of the sequence to be valid. If the sequence cannot be valid for any type that `Unknown` can represent, then it is ill-typed. As an example, the following sequence is valid:

```
block (result i32)
    i32.const 0
    br 0
    i32.const 1
end
```

because t^* is defined by the result type of the `block` as [`i32`] (which forces the previous sequence to satisfy the result type of the `block` whose execution will break), t_1^* can be [], and $[t_2^*]$ (`Unknown`) can be []. Therefore, the type of `br 0` in this case would be: [`i32`]→[], and the sequence is valid with the type of the `block` being []→[`i32`]. However, the following sequence cannot be valid for any type of the instruction:

```
block (result i32)
    i32.const 0
    br 0
    f32.const 1
end
```

because t^* is still [`i32`] and, whatever the types of t_1^* and t_2^* (`Unknown`) are, `f32.const 1` will produce a value of type `f32` which cannot be consumed by `br 0`, and does not match with the type of the block. When running this example in the official Wasm interpreter [41], the result is a validation fail, indicating the module is invalid. More specifically, that the instruction `block` requires its result to be `[i32]`, but the stack has `[f32]`.

Both examples can be found in the test suite from the code repository of the project [21]. The implementation of the validation of `br` i is as follows:

```
                TListLabel == nthResultT(LabelsTList, i)
-------------------------------------------------------------------------
verify-IL((br i) IL, TListLabel TList, → verify-IL(IL, Unknown,
  ...,LabelsTList)                          ...,LabelsTList)
```

where `TListLabel` corresponds to $[t^*]$, that is the i-th type token result list stored in the labels list.

3.7. Execution: instructions and semantic rules

In this section the syntactic definition and the semantics of the implemented instructions will be detailed, not only explaining the semantics defined by the specification but,

also, focusing on the design decisions considered for the implementation in Maude. The semantics here are implemented using rewrite rules in Maude, which rewrite terms of the execution state.

After the validation phase, the execution starts. Given a function, it is called with the `call` instruction, an execution frame is created, and its body is executed sequentially.

In this phase, execution errors may appear, which shall not be confused with validation errors (discussed in Section 3.6.1). These errors are marked with execution traps, and indicate undefined operations in Wasm, i.e., an integer division by 0 or the square root of a negative floating-point value. In the following sections, undefined results will appear as the `und` token, and traps are defined as terms of the form `Trap(`*String*`)`, containing a string that indicates the reason of the trap.

3.7.1. Numeric operations

Numeric instructions comprehend the set of instructions that produce and operate with numerical values. The scope of these instructions is the local environment. Because of that, only that environment will appear in the rewrite rules.

3.7.1.1. t.const c

This kind of instruction pushes to the stack a constant value `c` of type `t`. To do this, it wraps the value as a `const` object and pushes it to the top of the current stack.

The semantics of this is divided into five cases: one for each type of integer, `float` and `vector` when the value is of the corresponding type, and two extra cases. These last case allow the declaration of a `float const` with an integer value and the declaration of integers as signed values, storing its unsigned interpretation (as long as there is any). These cases are specified in the same way global variables were allocated based on their type on Section 3.5.2.2.

Below, we show the five rewrite rules implemented for this instruction. The common idea among all of them is that the value `c` is pushed to the stack as a `const` object if its type corresponds to `t` (or is included in `t`, in the case of `integers` and `floats`). Notice that in the derivations `c` is either `IntN` or `FloatN`, which correspond to the Maude sorts for integer and `float` values respectively. Vectors `const` values are represented with integer values. Moreover, the function `unsigned` is used to get the reverse representation to the signed representation of an integer if the value is in range.

In the case of `integers`, the value is checked to be within range, according to the bit width of type `t`. Due to `float` values being allowed to take `infinity` value, a function `correctF` is defined for `floats`, which transforms a `float` value to the corresponding representation in the internal IEEE 754 representation (specified in Section 3.2.1). Moreover, a `float` declared with an `integer` value shall be equivalent to the `integer` converted to `float` in the Maude representation.

$$\frac{0 \leq \texttt{IntN and IntN} < (2^{\texttt{(bitWidth(IntT))}})}{\begin{array}{l}\texttt{< (IntT.const IntN) IL |} \Rightarrow \texttt{< IL |} \\ \quad\texttt{ST |} \qquad\qquad\qquad\quad \texttt{const(IntT, val(IntN)) \$ ST |} \\ \quad\texttt{S >} \qquad\qquad\qquad\quad \texttt{S >}\end{array}}$$

$$\frac{\texttt{ValW1 := unsigned(val(IntN), bitWidth(IntT))}}{\begin{array}{l}\texttt{< (IntT.const IntN) IL |} \Rightarrow \texttt{< IL |} \\ \quad\texttt{ST |} \qquad\qquad\qquad\quad \texttt{const(IntT, ValW1) \$ ST |} \\ \quad\texttt{S >} \qquad\qquad\qquad\quad \texttt{S >}\end{array}}$$

```
< (FloatT.const FloatN) IL |  ⇒  if FloatN =/= 0.0 then
  ST |                              < IL |
  S >                                 correctF(const(FloatT, val(FloatN)))
                                        $ ST |
                                      S >
                                else
                                  < IL |
                                    correctF(const(FloatT, val(zero(pos))))
                                      $ ST |
                                    S >
                                fi
```

```
< (FloatT.const IntN) IL | ⇒ < (FloatT.const float(IntN)) IL |
  ST |                         ST |
  S >                          S >
```

$$0 \leq \texttt{IntN and IntN} < (2^{(\texttt{bitWidth(IntT)})})$$

```
< (v128.const IntN) IL | ⇒ < IL |
  ST |                       const(v128, val(IntN)) $ ST |
  S >                        S >
```

3.7.1.2. t.unop

Unary operations are all represented with `t.unop`, where `t` is the type of the operand. Below, the general rewrite rule for the semantics of this kind of operation is specified.

```
< (t.unopT) IL |          ⇒ < IL |
  const(t, ValW) $ ST |      unop(t, unopT, ValW) $ ST |
  S >                        S >
```

It works as a template rule as `t` represents any type token, `unopT` any unary operation defined and `ValW` any value. If the top of the stack is a `const` whose type matches `t`, the value in `ValW` must be of that type, because it has already been validated.

Regarding the result of the unary operation, it is obtained by the function `unop`, which takes as input: the type (`t`), the operation name (`unopT`) and the value `ValW`. This function, first, verifies the operation is defined for the values, then calculate the result, which is finally pushed to the stack.

Finally, the unary operations implemented are the ones defined for `float` values, namely `abs`, `neg`, `ceil`, and `floor`. They are included in the `FOPS` module in the code [21].

3.7.1.3. t.binop

This kind of instruction represents binary operations, which compute their result with two operands of type `t`. First, the system pops two values from the stack, checking they are of the same type `t`. Then, if the operation is defined for the operand values, it computes the result and pushes it to the stack. If the operation is not defined, a `Trap` is raised and the system halts.

The rule below shows the semantics of a `t.binop` instruction for some type `t`.

```
< (t.binopT) IL |     ⇒  if binop(t, binopT, ValW1, ValW2) =/= und then
  const(t, ValW2)            < IL |
  $ const(t, ValW1)            binop(t, binopT, ValW1, ValW2 $ ST |
  $ ST |                       S >
  S >                      else
                             trap("Binop result undefined")
                           fi
```

`binopT` is a token which refers to the specific binary operation. Notice that, for the rule to be applied, the topmost two elements of the stack must be two values of type `t`. Then, the function `binop` calculates the result value of the corresponding operation `binopT`. If the operation is defined, the result would be something different to `und`, and so it is pushed to the stack. If it is not defined, a trap is raised with a message indicating the reason.

The binary operations implemented are the ones corresponding to the usual arithmetic operations: addition, subtraction, multiplication, division, and remainder. Moreover, bitwise operations `and`, `or`, and `xor` are implemented for integer values. And `min`, `max` and `copysign` are defined for `float` values. As an example, the addition of integer values is defined as follows:

```
op binop : TypeToken BinopToken ValWrapper ValWrapper → Value .
eq binop(IntT, add, ValW1, ValW2) = const(IntT, iadd(bit-width(IntT),
    ValW1, ValW2)) .
```

where the auxiliary function `iadd` is defined as:

```
op iadd : Nat ValWrapper ValWrapper → ValWrapper .
eq iadd(N, ValW1, ValW2) = (ValW1 + ValW2) rem 2 ^ (N) .
```

The specific definition of each operation can be found in the `IOPS` and `FOPS` modules of the code [21].

3.7.1.4. t.testop

Corresponds to the `eqz` operation over integers, which pops and compares an `integer` value from the top of the stack with 0 and pushes the result to the stack. The result of this operation is 1 if the popped value is 0, and 0 otherwise. As defined in the specification [40], there is only one `eqz` is the only operation of this kind for now.

The rule below specifies the semantics for this instruction, where `ieq` is an operation that checks whether two integers are equal. In this case, the value popped from the stack shall be compared with 0.

```
< (IntT.eqz) IL |          ⇒ <IL |
  const(t, ValW1) $ ST |       const(i32, ieq(ValW1, val(0))) $ ST |
  S >                          S >
```

3.7.1.5. t.relop

This kind of instruction pops two values of type `t` from the top of the stack and compares them according to some relation. Similarly to `t.testop`, the result of the comparison is coded in binary (0 if false, 1 if true) and it is pushed to the stack as an `i32` value. Different comparison operations are allowed depending on the type, but it was already checked, in the validation phase, that the combination of operation and type is defined for WebAssembly.

The rewrite rule below, shows the behavior for this kind of instruction. The function `relop` chooses the operation according the type and operator name in the opcode, and returns the result of the comparison.

```
< (t.relopT) IL |                    ⇒ < IL |
  const(t, ValW2) $ const(t, ValW1)      const(i32,
    $ ST |                                 relop(t, relopT, ValW1, ValW2))
  S >                                      $ ST |
                                         S >
```

The implemented comparison operations include "equal to", "not equal to", as well as "greater" and "less than", and their combination with the equality comparison. Moreover, for integer values, there are different operations to use their signed or unsigned representations in the comparison. The complete description of the operators can be found in the code [21], in modules `IOPS` and `FOPS`.

3.7.1.6. t_2.cvtop_t_1

All conversion instructions are of the form t_2**.cvtop_**t_1, and convert a value of type t_1 at the top of the stack, into a value of type t_2, which is pushed to the stack.

Three conversion instructions have been implemented as part of the `cvtop` function, which is called by the rewrite rule shown below.

```
< (t2.cvtopT_t1) IL |      ⇒ if cvtop(t1, t2, cvtopT, ValW1) =/= und then
  const(t1, ValW1) $ ST |      < IL |
  S >                            cvtop(t1, t2, cvtopT, ValW1) $ ST |
                                 S >
                             else
                               trap("Cvtop result undefined")
                             fi
```

Depending on the token `cvtopT` and the specified types, `cvtop` returns the corresponding value, if defined. If the operation is undefined, a trap is raised indicating it and the system halts. The conversion instructions implemented are `i32.wrap_i64`, `f32.demote_f64`, and `f64.promote_f32`. The complete definition of each operation can be found in the code repository [21], in the module `CVOPS`.

3.7.2. Parametric operations

Parametric operations in WebAssembly are `drop` and `select`. The rewrite rules defined for them only affect the local environment.

3.7.2.1. drop

The `drop` instruction pops the top of the stack, regardless of its type. Hence, it is represented by the simple rule shown below, where `Val` represents any value located at the top of the stack.

```
< (drop) IL | Val $ ST | S > ⇒< IL | ST | S >
```

3.7.2.2. select

This instruction pops three values from the stack in the following order:

1. An `i32` value used as selector.
2. A value, `val2`, of type `t`.
3. A value, `val1`, of type `t`.

Notice there is a constraint on the type of the values, `t` can be any type as long as it is the same for both values. The semantics of `select` specify that if the selector value is 0, then `val1` is pushed back to the stack, otherwise `val2` is pushed instead.

In the implementation, the semantics are coded with the rule shown below:

```
< (select) IL |                                  ⇒ < IL |
  const(i32, ValW2) $ Val2 $ Val1 $ ST |            selectVal(ValW2, Val1, Val2)
  S >                                                 $ ST |
                                                    S >
```

where `selectVal` is an auxiliary function that contains the selection logic as follows:

```
op selectVal : ValWrapper Value Value → Value .
eq selectVal(val(NzIntN), Val1, Val2) = Val1 .
eq selectVal(val(0), Val1, Val2) = Val2 .
```

and where `NzIntN` is a variable representing a non-zero integer.

3.7.3. Variable operations

Variable instructions can be classified in two categories depending on the scope of the variable: local or global. The former, which comprises the first three instructions (`local.get`, `local.set` and `local.tee`), affects the local environment in the state of the system, and the latter affects the global environment (`global.get`, `global.set`).

From this point on, it will be useful to understand the operator `Map[Index]`, where `Map` refers to a structure mapping indexes to values (`Map` structure in Maude), and `Index` is the index allowed in the map. Then, `Map[Index]` retrieves the value mapped to `Index` in `Map`. As an example, a map that appears in this section is `LocalSTR`, which represents the store for local variables and their values in the current execution frame.

3.7.3.1. local.get

This instruction gets the value of a local variable by reference, and pushes it to the stack. Below, we find the rule defined for the semantics of this instruction, where `ValR` corresponds to the reference index, and the value is retrieved from the corresponding local variable, stored in the local store `LocalSTR`.

```
< (local.get ValR) IL | ⇒ < IL |
  ST |                      LocalSTR[ValR] $ ST |
  LocalSTR S >              LocalSTR S >
```

3.7.3.2. local.set

As opposed to the previous instruction, `local.set` pops the top value from the stack and sets the value of the referenced local variables to it. The semantics of this instruction is shown below, where `ValR` is the reference to the local variable, `Val` is the value at the top of the stack and `LocalSTR` is the local variable store.

```
< (local.set ValR) IL | ⇒ < IL |
  Val $ ST |                ST |
  LocalSTR S >              insert(ValR, Val, LocalSTR) S >
```

3.7.3.3. local.tee

This instruction stores the value at the top of the stack in the referenced variable, while keeping it at the stack. To do so, it pops the value, then pushes it twice and, finally, executes `local.set` with the same variable reference that it was called with. The rule can be found below. The intuition is that `Val`, the value at the top of the stack, is popped, then pushed twice at the stack and, at the instruction list, instead of going with the rest, `local.set ValR` is added at the beginning. This forces it to be the next instruction to be executed, storing the top `Val` in the referenced variable, while keeping the second `Val` at the top of the stack.

```
< (local.tee ValR) IL | ⇒ < (local.set ValR) IL |
  Val $ ST |                Val $ Val $ ST |
  S >                       S >
```

3.7.3.4. global.get

Analogously to `local.get`, this instruction gets the current value from a global variable and pushes it to the stack. To do this, the instruction is executed under the global environment, accessing the global index and variable stores. The rule below illustrates the behavior of this instruction. To get the variable being referenced, the system accesses the address and global instance stores. With `GBStore[GBAddrs[Str]]`, it gets the global instance structure. Then, the function `gbVal` gets the value of a global variable from the global instance structure in the store.

```
< (global.get Str) IL | ⇒ < IL |
  ST |                      gbVal(GBStore[GBAddrs[Str]]) $ ST |
  S >                       S >
  # GBAddrs <-> GBStore     # GBAddrs <-> GBStore
```

3.7.3.5. global.set

In the same way `global.get` was analogous to `local.get` in the global environment, `global.set` is analogous to `local.set` in that environment. The intuition is that the top value of the stack is stored as the current value of the referenced variable. In this case, only mutable global variables are allowed to have their value changed. As mentioned in Section 3.2.1, mutable variables are marked with a type `mut t`, where `t` is any valid type. However, there is no need to check that at this point because if the module is valid, the global variable must be mutable. If it were not valid, it would have caused a validation fail in the previous phase. The rule for this instruction is as follows:

```
< (global.set Str) IL |  ⇒ < IL | ST | S >
  const(t, ValW1) $ ST |     # GBAddrs
  S >                        <-> insert(
  # GBAddrs <-> GBStore        GBAddrs[Str],
                               type (mut t), value const(t, ValW1),
                               GBStore)
```

The value of the corresponding global variable in `GBStore` is changed by inserting the new `const` pair for the instance, where the value is `const(t, ValW1)`. This insertion is performed on the index corresponding to the referenced variable, obtained by accessing `GBAddrs`.

3.7.4. Control operations

Control instructions affect the control flow of the execution, including loops, conditionals, jumps within the code, and function calls. They affect different environments depending on the specific needs of the instruction, but, in general, they use most of the elements in the system state.

3.7.4.1. nop

According to the specification, the `nop` instruction does nothing, except removing `nop` from the instruction list. This behavior is modeled by the following rule:

```
---------------------------------------
< (nop) IL | ST | S > ⇒< IL | ST | S >
```

3.7.4.2. block

The `block` instruction wraps and joins a list of instructions. It is used to encapsulate instruction sequences that are expected to be executed together, i.e., in a conditional or after a jump in the program. Blocks are marked with labels, which are pushed to the stack once the execution enters the blocks, as described in Section 3.2.2. The type of a `block` is indicated by a `BlockType`, which is translated to a type reference after allocation, as described in Section 3.5.3.2. Moreover, at this point in execution, the type of a `block` was already checked in the validation phase. The syntax for this instruction is `(block TypeRef) IL2 end`, where `TypeRef` is a structure referencing a type and `IL2` is the sequence of instructions contained in the block. The rewrite rule defined is as follows, where `expT` is the expanded type corresponding to `TypeRef`:

```
          ST' := getTopVals(ST1, getParamList(expT))
          /\ ST := st-popN(ST1, st-size(ST'))
          /\ TListRes := getResList(expT)
-------------------------------------------------------------------
{typeIndCtx IndTypeCtxSTR} ... ⇒ {typeIndCtx IndTypeCtxSTR} ...
  < (block TypeRef IL2 end)        < IL2 exitB(IL) |
    IL |                           ST' $
  ST1 |                            lab("b",
  S >                                nil,
  ... # TypeStore                    size(TListRes)) $
                                   ST |
                                   S >
                                   ... # TypeStore
```

This rule triggers the execution to enter the block. To resolve the reference, the system accesses the type and type index stores. The rule, then, gets both the parameters and the results from the referenced type. The state of the system must satisfy that, at the top of the stack, there are n values of the types corresponding to the parameters, which has been checked in the validation phase. It generates a label `L` with an empty list of instructions to execute after a jump and with arity equal to the size of the results list of the referenced type. Then, places `L` below the n parameters. Finally, `IL2` is placed in the

current instruction list, followed by `exitB`, an auxiliary operation which stores the next instructions and will handle the block exit.

Finally, when the execution exits a block, it executes operation `exitB`, which marks the end of the block. It pops the label from the stack and restores the original list of remaining instructions. This process still keeps the values on top of the label in the stack, as they are popped and pushed back, once the label is removed. The behavior of this is specified by the following rule, where `reset-st` is the operation that performs the aforementioned stack transformation.

```
---------------------------------
< exitB(IL) ⇒ < IL |
    IL |        reset-st(NeST) |
  NeST |        S >
  S >
```

3.7.4.3. loop

A `loop` instruction in WebAssembly is similar to a `block` which, at the end of its instruction sequence, resumes the execution at its beginning. Therefore, the rule defined for `loop` is quite similar to the one defined for `block`, as shown below:

```
          ST' := getTopVals(ST1, getParamList(expT))
          /\ ST := st-popN(ST1, st-size(ST'))
          /\ TListParam := getParamList(expT)
-------------------------------------------------------------------
{typeIndCtx IndTypeCtxSTR} ... ⇒ {typeIndCtx IndTypeCtxSTR} ...
  < (loop TypeRef IL2 end)         < IL2 exitB(IL) |
    IL |                           ST'
  ST1 |                            $ lab("l",
  S >                                loop TypeRef IL2 end,
  ... # TypeStore                    size(TListParam))
                                   $ ST |
                                   S >
                                   ... # TypeStore
```

The main difference between `block` and `loop` is that, in the latter, the label points back to the beginning of the `loop` instruction and its arity is, therefore, equal to the size of the parameter list of the block. It is worth mentioning that it does not perform the execution jump back to the beginning, which is considered in the semantics of instructions `br` and `br_if`.

3.7.4.4. if..else

The conditional statement `if..else` is composed of two blocks of instructions: one executed if the condition is satisfied and another executed otherwise. The complete definition of a instruction `if..else` is of the form `if BlockType IL1 else IL2 end`, where `BlockType` is the type of the instruction, and `IL1` and `IL2` are instruction lists. However, the `else` component is optional and, if it is not present, it is assumed to be the empty list `nil`.

The condition is defined by the value at the top of the stack, which must be an `i32` value. If it is not 0, then the first block (`IL1`) is executed, otherwise the second (`IL2`) is executed. To execute each of the blocks, the semantics involve executing a `block` statement equivalent to the case to be executed. The rule below shows the semantics for this instruction:

```
< (if BlockType IL1 else IL2 end) IL | ⇒ < if-else(ValW1, IL1, IL2,
  const(i32, ValW1) $ ST |                  BlockType) IL |
  S >                                     ST |
                                          S >
```

where the function `if-else` generates the corresponding `block` statement of the same `BlockType` specified for the conditional. The following code snippet shows the behavior of the `if-else` function:

```
op if-else : ValWrapper List{Inst} List{Inst} BlockType → List{Inst} .
eq if-else(val(NzIntN), IL1, IL2, BlockType) = (block BlockType IL1 end
    ) .
eq if-else(val(0), IL1, IL2, BlockType) = (block BlockType IL2 end) .
```

where `NzIntN` is a variable representing a non-zero integer.

3.7.4.5. br i

This instruction jumps the execution to the instructions after the i-th label in the stack. To perform the jump, first, the system searches the corresponding label in the stack, gets its arity and pops that number of values from the stack, let's call them `resList`. Then, it pops elements from the stack (values and labels) until it reaches the i-th label, which it pops. After this, the system pushes `resList` back to the stack and the instruction list becomes the one stored in the label followed by the continuation to the block, marking the end of the block. For this instruction, the following rule is defined:

```
(ST1 || lab(Str, IL1, N1)) := get-ith-stackandlabel(ST, i, EmptyStack)
    < (br i) IL2 exitB(IL) IL' | ST | S > ⇒< IL1 IL | ST1 | S >
```

It illustrates the process using the function `get-ith-stackandlabel`, which gets the i-th label and the new stack after all the transformations mentioned.

3.7.4.6. br_if i

The previous instruction defined an unconditional jump in execution, in contrast, this defines a conditional jump. This instruction requires an `i32` value at the top of the stack and the jump to the i-th label is performed only if the value at the top of the stack is not 0. If it is 0, then the execution continues with the next instruction. Actually, the jump is executed by the instruction `br`, as its rule shows:

```
< (br_if i) IL |                ⇒ if Int1 =/= 0 then
  const(i32, val(Int1)) $ ST |      < (br i) IL | ST | S >
  S >                             else
                                    < IL | ST | S >
                                  fi
```

3.7.4.7. call

This instruction is used to call a function, either by index reference or by name. Calling a function consists of initializing a new local environment and executing the instruction sequence of the desired function within that environment. To do so, the system must, first, get the corresponding structure containing the function information. Then, with the

aid of some auxiliary functions, it shall get the parameters, results, and the instruction sequence from the function (with `getParamList-func`, `getResList-func`, and `getIL-f` respectively). With this, it shall get the parameter values from the stack and assign them to the local variables in a new and empty local store. Moreover, it shall reserve as many variable slots as parameters plus locals are defined (with their default values).

This is shown in the following rules. The first rule corresponds to a function `call` referenced by index and the second, by name. The latter just gets the index reference of the function from its name and relies on the first to execute the actual semantics. Because of that, the behavior of the first will be explained more in depth.

```
                FuncDef := FuncStore[Addr]
                /\ TListRes := getResList-func(FuncDef)
                /\ TListPar := getParamList-func(FuncDef)
                /\ TListLoc := getLocList-func(FuncDef)
-------------------------------------------------------------------------
< (call Addr) IL | ⇒ < getIL-f(FuncDef) exitF(TListRes) exitB(IL) |
  ST |                 lab("f", nil, size(TListRes))
  S >                     $ popN(ST, size(TListPar)) |
  ... # FuncStore     init-store(TListPar TListLoc,
                        getTopVals(ST, TListPar), empty) S >
                      ... # FuncStore
```

```
{funcIndCtx IndFuncCtxSTR} ⇒ {funcIndCtx IndFuncCtxSTR}
  < ((call Str) IL | ... >      < ((call (IndFuncCtxSTR[Str])) IL | ... >
```

When a function is called by index reference, the system produces the following changes in the state:

- The instruction list becomes the instruction sequence from the function. The function `getIL-f` gets the sequence, and auxiliary operations `exitF` and `end` are appended to return from the function once it is finished.
- n values are popped from the stack with the function `popN`, where n is the number of parameters of the function. After this, label `L` is created with the instruction sequence to execute after a jump (the end of the function call) and arity the number of result values from the function.
- A new local store is prepended to the local store list. This helps create the new environment, while keeping record of the previous one. The number of variables with space reserved in the store is equal to the number of parameters plus the number of locals specified for the function. Their space is reserved in that order too. This is the initialization of the new local environment, and it is performed by the auxiliary function `init-store`.

Chapter 4

Testing the system

This chapter describes the tests performed over the system to check whether the semantics are correctly defined and implemented. First, the whole test suite is detailed, considering simple tests for each instruction and complex tests combining several instructions in a program. Then, the results of the tests are evaluated against the official Wasm specification interpreter in OCaml [41].

4.1. Test suite

The official Wasm test-suite [42] was considered to test this system, but due to their specific format were not contemplated as the main testing source. Those tests are in WAST format, which is used strictly to test the official interpreter, as it extends the WAT format to include testing features. WAST also presents modules with a folded design for the instructions, which was not considered in this system. Despite the format differences, some official tests were transformed to WAT format and considered to test control instructions.

Although it would have been interesting to fully test the system with the official test suite, an extensive test suite has been created to cover the possible outcomes for all the instructions implemented. Each instruction may have different operations, for example: integer binary operations can be additions, subtractions or signed divisions. Therefore, the tests for each specific operation are included in the file of the corresponding general instruction. In addition, the test suite includes tests for incorrect types and incorrect parameters, which shall rise a validation error.

Furthermore, we have designed some random tests for numeric instructions, which allows covering a significant number of cases in the range of the values, as any number of tests can be performed with new random values each execution. We have created WebAssembly modules with template values, which are substituted with random values in each execution by the Bash script `wasm-test-suite_randomized.sh`.

To consider relevant cases for control instructions, they are tested in combination with other instructions which are previously tested. Also, as mentioned before, some tests for these instructions from the official test suite were transformed and included in the test suite. These transformations do not alter the behavior of the program under test, only its format. Finally, some complex tests are included, which are composed of several instructions and they resemble real-world examples.

Examples of these tests are the modules shown in Figure 5.1 and Listing 4. The former corresponds to a test for the instruction `br_if` where the jump is performed, and the latter

corresponds to a complex test combining several instructions.

Overall, the test-suite contains 252 original tests, 8 template tests, and 31 transformed official tests.

4.2. Results

The results of the tests have been compared with the official interpreter for WebAssembly, which their authors define as specification-focused, meaning it implements the specification as is. However, the main limitation of this process is that the interpreter only returns the values in the stack, no information about the intermediate states nor the values in local or global variables is available. This makes the testing process similar to black-box testing.

While it would be interesting, specifically for variable or control instructions, to check against the official interpreter intermediate states in the execution, testing from unitary examples to more complex ones, constructing the latter with instructions already tested in the former, allows assessing the validity of complex instructions and instruction blocks.

As a result of the testing process, some bugs were found in the implementation, which led to a reconsideration of some implementation aspects. First, it helped to detect a misunderstanding of signed integers in the `t.const` instruction. Internally, Wasm uses unsigned integers to encode all integers. Therefore, only unsigned integers where allowed in the integer `t.const` instruction at first. However, in text format, it allows using signed integers to define values at high level and performs the conversion with a function `signed`. As an example, signed value `i32.const -1` is equivalent to unsigned value `i32.const 4294967295`, but both are represented internally as unsigned 4294967295.

Moreover, some significant fixes were developed for the correct execution of instructions involving labels. At first, labels included the whole instruction sequence that follows the block they tag, which is an interpretation of what the specification mentions in words for each of those instructions. Even so, with that approach, the `loop` instruction does not terminate because the follow-up instruction is itself. Therefore, to handle this, the formal definition of the semantics indicated that labels must contain the instruction sequence that is executed immediately after a jump, then, when exiting the block, the remaining instruction sequence is executed. This can be observed in the semantics for instructions `block` and `loop` in Sections 3.7.4.2 and 3.7.4.3 respectively.

Furthermore, it is important to mention that some tests involving `f32` values actually fail. This is because this implementation uses the 64-bit float type for both 32-bit and 64-bit float values, as explained in Section 3.2.1. This restriction affects the precision of the values, but not the range (values exceeding the representable range take value `inf`). This may lead to errors sliding and carried through the program if a Wasm module relies on `f32` precision for its calculations.

Finally, notice that the implementation in Maude offers information of the full state of the system, both for intermediate and final states. This gives the user relevant information for debugging Wasm programs, including the values of local and global variables at each execution step.

Chapter 5

Verification of programs

This chapter describes how to use the semantics presented in Chapter 3 to verify WebAssembly code through some Maude commands and the features developed for this project.

5.1. How to use the semantics

The semantics allows verifying different kinds of properties over Wasm programs. Using Maude commands, it is possible to do model checking of invariants with the `search` command, as described in Chapter 11 of the Maude manual [6, Chap. 11]. This command performs a search in the state space of the small-step semantics defined in Maude. One can analyze properties over final states or intermediate states. There are, at least, three main features that can be tested with this command on the system:

- Module validity. It is possible to check whether the final state is a `Fail` state, indicating that the module is invalid according to the specification validation.
- System properties. Additionally, one can check properties over the execution system, for example, if the stack size should not exceed a certain value at any point in execution or if the value of certain system variables should stay in some range.
- Program properties. Also, if the program to be verified has specific properties, they can be checked at any execution state. For example, if a function only returns -1 when there is an internal execution error, one could check that after the function execution, -1 is not at the top of the stack.

Our system can be used in two different ways, either by loading an external file containing a Wasm module or, by providing the input in the internal representation and reducing the module as a Maude term.

5.1.1. From external file

When loading a module from an external file, the parser module described in Section 3.4 is used. This extension of the code uses the Maude support for reading external objects, as well as meta representation and meta search. The functionality for reading external files is provided by the `file.maude` file of the Maude distribution.

The function `search-with-func-from-file` executes a meta search in Maude as described in Section 2.3.3, invoking a function defined in the Wasm module from the file. It has several parameters, including the path to the WAT file, the name of the function to be invoked, initial instructions to execute before the function, and the usual parameters of a Maude search. The latter includes the pattern to match as a goal, a list of conditions, the type of the search (marking the number of steps or whether it should match only with final states), the maximum depth, and the solution number to display (where 0 is the first).

As an example, consider Listing 4, which implements the factorial as a recursive function in a fully-functioning Wasm module. The module defines a function `start` that calls the factorial function with value 5. Then, in Maude, one can invoke function `"$func0` with the command:

```
erew search-with-func-from-file("test-suite/factorial.wat", "$func0",
                                "",
                                'w:WASMEnv, nil,
                                '!, unbounded, 0) .
```

where `test-suite/factorial.wat` is a file containing the module in Listing 4, `"$func0"` is the function to be invoked, `""` indicates no initial instructions should be loaded to the system, `'w:WASMEnv` the meta-representation of the pattern to be matched (a complete state of the system) and `'!` indicates the search is over final states. The remaining parameters `nil`, `unbounded`, and `0`, indicate the search has no conditions, it does not have maximum depth, and it should display the first solution, respectively.

The result of executing that command is an assignment of a `WASMEnv` structure to the term `w` such that, after applying the rewrite rules defined in the module, the stack only contains the value `const(i32, val(120))`, which corresponds to the factorial of 5.

However, this search is limited because the command `erewrite` controls the interaction, in the sense that the user can only see the final term and does not obtain other relevant information from it, like the rules applied or the complete search graph. Doing that would require to develop an internal interface that would capture the output as a term and execute meta-operations with the previous results. Alternatively, one could use the original search command over a pre-parsed module. This is discussed in the next section, which allows us to fully exploit the capabilities of Maude.

5.1.2. From testing module

To overcome the restrictions of the previous use case, one can load a Wasm module with the syntax of `WASMMOD` module. This allows taking the full advantage of the search analysis features in Maude by being able to use the interface in Maude over the result term.

The main problem that arises here is the need to transform Wasm modules into the syntax allowed in `WASMMOD`. The solution proposed in this case is: transforming the Wasm module into an ad-hoc Maude module that extends `WASMMOD` and defines a constant `test` of sort `WASMEnv` with the operator that the system specifies to run a function in the Wasm module, in the expected syntax. This is performed by the Python script `wasm2wasmMaude_transformer.py`. The input and output files are configurable, as well as the target function to test within the module. However, the function can be modified once the Maude module is generated, in case several functions in a module want to be tested.

Figure 5.1 shows an example of module transformation. On the left, the Wasm module to be transformed, and on the right, the Maude module ready to be loaded into the

```
(module
    (type $type0 (func (result i32)))
    (type $type1 (func (param i32) (result i32)))
    (export "$func0" (func 0))
    (func $func0 (type $type0)
        i32.const 5
        call $factorial
    )
    (func $factorial (type $type1)
        i32.const 0
        local.get 0
        i32.eq
        if (result i32)
            i32.const 1
        else
            local.get 0
            local.get 0
            i32.const 1
            i32.sub
            call $factorial
            i32.mul
        end
    )
)
```

Listing 4: Factorial function module

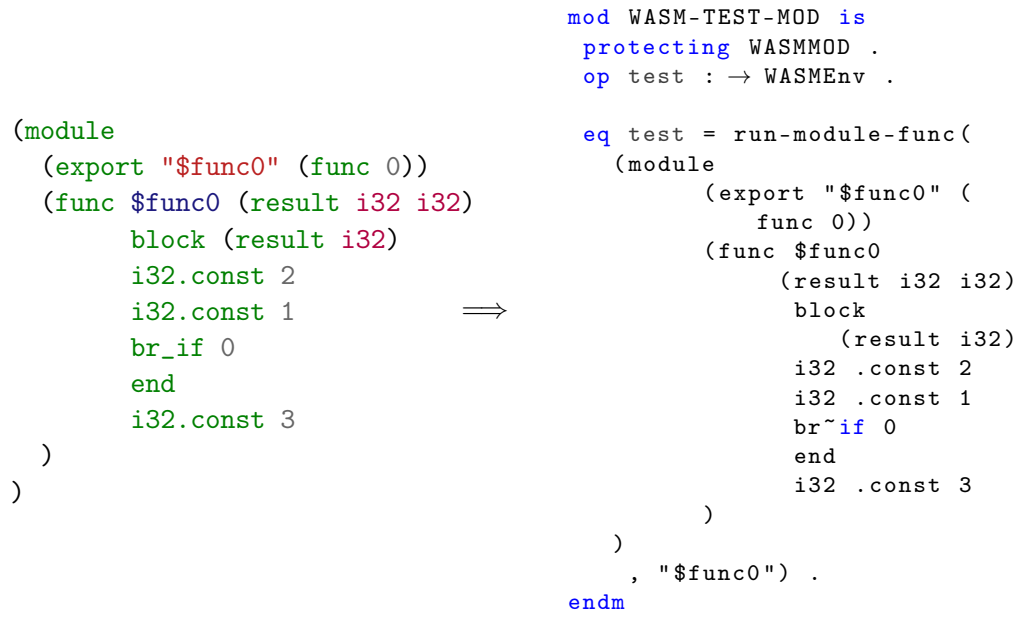

Figure 5.1: Module transformation

system. Notice the module syntax is slightly changed, namely: `i32.const` transforms into `i32 .const` and `br_if` into `br~if`.

Moreover, the output module creates a constant `test` that runs the target function in the Wasm module and can be used in Maude commands without restrictions. As an example, to analyze the state of the system at the point when the instruction `br_if` is executed, one could execute the command:

```
search [,4] test =>* w:[WASMEnv] .
```

Then, to obtain the rules that have been applied, it could be useful to execute the command `show path labels` at that state:

```
Maude> show path labels 4 .
call
block-tref
int-const
int-const
```

5.2. Verification of properties

In the previous Sections 5.1.1 and 5.1.2, simple execution examples were presented, where the main goal was to attest Wasm modules could be loaded and executed in the system. In this section, three kinds of properties verification are presented, based on the ones mentioned at the introduction of Section 5.1.

There are different auxiliary functions available in the `WASMMOD` module to look up the various components of the execution environment. They take as input a `WASMEnv`, which corresponds to the full state. As an example:

- `localEnv` gets the local environment of a state.
- `getGlobal` gets the value of a global variable, given the state and the variable name.
- `curr-inst` gets the instruction to be executed at each state.
- Regarding the stack, several functions are defined given a state: `stack` gets the stack itself, `st-size` gets the size of the stack, and `st-topVal` gets the value at the top of the stack.

Along the examples, the `META-LEVEL` function `upTerm` will be used to keep the syntax readable. This function takes a term in common syntax and returns its meta-representation, as explained in Section 2.3.3.

5.2.1. Module validity

First, consider again the factorial function module from Listing 4, to assert its validity one should check that it does not reach a `Fail` state as mentioned in Section 3.6.1.2. This can be done by performing the following test:

```
erew search-with-func-from-file(
                "test-suite/factorial.wat", "$func0", "",
                upTerm(Fail("Invalid module")),
                nil,
                '!, unbounded, 0) .
```

Notice it searches only on final states, as we know that, by design, a `Fail` state is always a final state. If there are no results (the output is `NoResult`) then the module must be valid. Otherwise, the system will show a state where the module is reduced up to a `Fail` state

5.2.2. System properties

Another kind of property to test are those regarding the execution system itself. As the Maude implementation includes the main components, one can set conditions over them. As an example, WebAssembly programs might be deployed on systems with minimal resources, so asserting they work with, for example, restricted stack size would be interesting.

To test this, consider the same factorial module from Listing 4. In this case, the program has to calculate the factorial of 100, but the stack cannot exceed a size of 250 elements. Therefore, the command directly calls the factorial function, with an initial instruction `i32.const 100`, which will load 100 as its parameter. In addition, to test the property, the condition is set to check if at any state the stack size exceeds 250. Finally, the symbol `*` is used in the search to allow any number of steps in the search.

```
erew search-with-func-from-file(
                "test-suite/factorial.wat", "$factorial",
                "i32.const 100",
                'w:`[WASMEnv`],
                upTerm(st-size(stack(w:[WASMEnv])) > 250)
                    = 'true.Bool,
                '*, unbounded, 0) .
```

In this case, it finds at least one solution, meaning that the execution does exceed the desired stack size. This is due to the recursive approach of the factorial implementation and, if it were a critical condition, this would suggest that the implementation should be modified. If no solutions were found, it would mean the restriction was satisfied.

5.2.3. Program properties

Finally, it is possible to test properties of WebAssembly programs. These properties are specific to the programs themselves and require a specification of the expected behavior of the program.

Consider a new module which extends the module in Listing 4 to include a function that checks that the number to calculate the factorial from is positive. If the parameter is negative, it returns -1. Therefore, the new module includes a function `$func1` described as follows:

```
(func $func1 (param i32) (result i32)
    local.get 0
    i32.const 0
    i32.lt_s
    if (result i32)
        i32.const -1
    else
        local.get 0
        call $factorial
    end
)
```

Then, after invoking the function for a certain parameter, to check that the result is computed accordingly, the search must satisfy two conditions: that the execution is exiting the function, and that there is not a -1 value at the top of the stack. This must be checked at every intermediate state, therefore, the command is as follows:

```
erew search-with-func-from-file(
    "test-suite/factorial_withLt.wat", "$func1",
    "i32.const 3",
    'w:`[WASMEnv`],
    upTerm(curr-inst(w:[WASMEnv]))
        = upTerm(exit-func(i32))
    /\ upTerm(st-topVal(stack(w:[WASMEnv])))
        = upTerm(const(i32, unsigned(val(-1), 32))),
    '*, unbounded, 0) .
```

As the states we want to evaluate are at a point where the execution is exiting a function, we want to search for states where the current instruction is the function exit. Also, the top value of the stack must be -1. `unsigned(val(-1), 32)` indicates -1 in the unsigned 32-bit representation, which is the internal representation for integers in the system.

In that case, the function computes the factorial of 3, so there is no result. But, for any negative number, it will show a solution.

Chapter 6

Conclusions and future work

6.1. Conclusions

In this Master's thesis, the main goal was to model the WebAssembly semantics in Maude, which, as far as we know, it had not been previously done. We have successfully implemented a reasonable subset of the language, including its execution phases, which are formalized according to the official specification. Moreover, we have presented a way to verify WebAssembly programs in Maude by strictly following the formal semantics.

In Sections 3.2.1 and 3.3, we described how WebAssembly program components are represented in Maude. The model is organized in Maude modules, which will ease developing future extensions of the system without altering the whole implementation.

Moreover, the scripts developed for testing the system and transforming Wasm modules to Maude modules are a way of allowing users who may be unfamiliar with Maude to use the tool. They include everything needed to execute, debug, or verify Wasm programs in Maude with simple commands. Additionally, the module parser described in Sections 3.4 and 5.1.1 allows executing Maude commands directly over WebAssembly files. Its implementation can be taken as reference for other program verification systems in Maude, and could be extended to a more sophisticated verification tool, all within Maude.

Furthermore, in Chapter 5, we define a wide test suite (containing both custom and official tests) and compare the behavior of the implemented system against the official Wasm interpreter. The results obtained show that the implementation correctly implements the specification. The Maude implementation passes all the tests after fixing some divergencies, except for 32-bit floats. This issue is known and discussed in Section 4.2.

It is worth mentioning that using Maude for the implementation of the semantics has been an appropriate choice. The flexibility of Maude programs to resemble formal specifications has eased the translation of the semantic rules. The main difficulties that arose during the development phase were translating some of the nuances in WebAssembly internal operations to Maude. However, this work has also served as an extensive dive into the intricacies of Maude. Some of the functionalities of the implementation involve advanced Maude features that required a thorough study of the language.

Finally, there is clearly an important amount of possible extensions to this work, as included in Section 6.2. Even so, this work successfully implements a remarkable subset of WebAssembly operations and components in Maude, following the language semantics.

6.2. Future work

This section includes ideas that would be interesting to explore in the future or were left out from the implementation due to their complexity. They could be developed as extension modules to the ones described in this document.

6.2.1. Complex types and components

Implementing the semantics for the full set of types and components would be of great value to verify every kind of program. More specifically, vectors, memory, and tables would be the most useful features of the WebAssembly core left to be implemented.

6.2.1.1. Vectors

Although vectors have been briefly mentioned along this document, they are not fully implemented in the modules. Only the base for their definitions is included, like the type tokens, the `const` instruction, and some auxiliary functions to work with their shapes. However, it would be interesting to have a fully-functioning implementation of the representation and the semantics of vector operations. Vectors allow computations with several values at the same time with the SIMD (Single Instruction Multiple Data) paradigm, as well as storing them. The size of a vector is always 128 bits, and can be organized internally with different shapes.

6.2.1.2. Memory and data

WebAssembly supports a memory component, which is used during execution to load and store data directly with their binary representation. Moreover, data can be loaded from the module definition as predefined data, by using the `data` component.

6.2.1.3. Tables

WebAssembly tables are components that allow dynamic index handling. With them, a program is allowed to call functions indirectly, using a table as an index decoder, as if they were referenced by pointers.

6.2.2. Module interaction

Wasm modules are allowed to interact by exporting their components and importing from other modules. To expose them, the component `export` is used, which takes a string value and a reference to a component and exposes it, for other modules to import them. A module can export functions or global variables.

Although the implemented system allows, syntactically, a module having `export` components, the semantics of exposing the components externally have not been considered for this project. For this to take actual effect on the system, the full instantiation process defined in Wasm specification shall be implemented. It handles when and how exported elements are exposed, and imported elements are included in the system.

6.2.3. JavaScript extension

WebAssembly can be used as an extension to JavaScript code, where application written in the latter can use functions written in the former, and vice versa. To have a full

implementation of the system where a WebAssembly module is executed it would be very valuable to be able to interact with JavaScript code. To do this, we would have to extend the definition of the execution state and the allocation process to deal with imports in both environments. This extension would not need to implement the full semantics of JavaScript (although that would be a great contribution), only those functions that allow the interaction with WebAssembly.

6.2.4. Symbolic execution and SMT solver connection

Finally, another interesting extension to this work would be to explore the field of verification via symbolic execution. This would let us reason over symbolic structures within WebAssembly programs and verify more complex properties. This would shape a wider framework for program verification using Maude. Although the support for SMT solvers in Maude is limited, Maude-SE [44] is an extension which has already been used for verification [1], and might be a good start for this purpose.

Bibliography

[1] Jaime Arias, Kyungmin Bae, Carlos Olarte, Peter Csaba Ölveczky, and Laure Petrucci. A Rewriting-logic-with-SMT-based Formal Analysis and Parameter Synthesis Framework for Parametric Time Petri Nets. *Fundamenta Informaticae*, 192(3-4):261–312, 2024. DOI: 10.3233/FI-242195.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. DOI: 10.1017/CBO9781139172752.

[3] Roland Barthes. *Fragments d'un discours amoureux*. Éditions du Seuil, Paris, 1977. ISBN: 2-02-004605-9.

[4] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN: A logical framework based on computational systems. *Electronic Notes in Theoretical Computer Science*, 4:35–50, 1996. DOI: 10.1016/S1571-0661(04)00032-5. RWLW96, First International Workshop on Rewriting Logic and its Applications.

[5] Tiago Brito, Pedro Lopes, Nuno Santos, and José Fragoso Santos. Wasmati: An efficient static vulnerability scanner for WebAssembly. *Computers & Security*, 118:102745, 2022. DOI: 10.1016/j.cose.2022.102745.

[6] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Carolyn Talcott. *Maude Manual (Version 3.5.1)*. July 2025. URL: https://maude.cs.illinois.edu/manual.pdf.

[7] Răzvan Diaconescu and Kokichi Futatsugi. Logical foundations of CafeOBJ. *Theoretical Computer Science*, 285(2):289–318, 2002. DOI: 10.1016/S0304-3975(01)00361-9. Rewriting Logic and its Applications.

[8] Docs | SpiderMonkey JavaScript/WebAssembly Engine. URL: https://spidermonkey.dev/docs/ (visited on 08/10/2025).

[9] Documentation · V8. URL: https://v8.dev/docs (visited on 08/10/2025).

[10] Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 533–544, Philadelphia, PA, USA. Association for Computing Machinery, 2012. DOI: 10.1145/2103656.2103719.

[11] Emscripten. URL: https://emscripten.org/ (visited on 07/23/2025).

[12] Giovanni Fabbretti, Ivan Lanese, and Jean-Bernard Stefani. Generation of a reversible semantics for Erlang in Maude. Research Report RR-9468, Inria - Research Centre Grenoble – Rhône-Alpes, April 2022. URL: https://inria.hal.science/hal-03630407.

[13] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roşu. Formal Analysis of Java Programs in JavaFAN. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, pages 501–505, Berlin, Heidelberg. Springer Berlin Heidelberg, 2004. DOI: 10.1007/978-3-540-27813-9_46.

[14] Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. *Introducing OBJ*. In *Software Engineering with OBJ: Algebraic Specification in Action*. Joseph Goguen and Grant Malcolm, editors. Springer US, Boston, MA, 2000, pages 3–167. DOI: 10.1007/978-1-4757-6541-0_1.

[15] 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, pages 185–200, Barcelona, Spain. Association for Computing Machinery, 2017. DOI: 10.1145/3062341.3062363.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. DOI: 10.1145/363235.363259.

[17] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*:1–84, 2019. DOI: 10.1109/IEEESTD.2019.8766229.

[18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. en. *Journal of Functional Programming*, 28:e20, January 2018. DOI: 10.1017/S0956796818000151.

[19] KWasm: Semantics of WebAssembly in K. (Visited on 04/13/2025). https://github.com/runtimeverification/wasm-semantics.

[20] Narciso Martí-Oliet and José Meseguer. Rewriting Logic as a Logical and Semantic Framework. *Electronic Notes in Theoretical Computer Science*, 4:190–225, 1996. DOI: 10.1016/S1571-0661(04)00040-4. RWLW96, First International Workshop on Rewriting Logic and its Applications.

[21] Master's Thesis code - Semantics of WebAssembly in Maude. URL: https://github.com/rafamor-exe/wasm-maude (visited on 08/27/2025).

[22] Patrick Meredith, Michael Katelman, José Meseguer, and Grigore Roşu. A formal executable semantics of Verilog. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pages 179–188, 2010. DOI: 10.1109/MEMCOD.2010.5558634.

[23] José Meseguer. Membership algebra as a logical framework for equational specification. In Francesco Parisi Presicce, editor, *Recent Trends in Algebraic Development Techniques*, pages 18–61, Berlin, Heidelberg. Springer Berlin Heidelberg, 1998. DOI: 10.1007/3-540-64299-4_26.

[24] José Meseguer. Research Directions in Rewriting Logic. In Ulrich Berger and Helmut Schwichtenberg, editors, *Computational Logic*, pages 347–398, Berlin, Heidelberg. Springer Berlin Heidelberg, 1999. DOI: 10.1007/978-3-642-58622-4_10.

[25] José Meseguer. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming*, 81(7):721–781, 2012. DOI: 10.1016/j.jlap.2012.06.003. Rewriting Logic and its Applications.

[26] Microsoft ActiveX Controls. URL: https://learn.microsoft.com/en-us/cpp/mfc/activex-controls (visited on 04/18/2025).

[27] David Munuera Mazarro. Specification and verification of WebAssembly programs [Master's Thesis], June 2023. URL: https://oa.upm.es/75802/.

[28] Gaetano Perrone and Simon Pietro Romano. WebAssembly and security: A review. *Computer Science Review*, 56:100728, May 2025. DOI: 10.1016/j.cosrev.2025.100728.

[29] 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. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. DOI: 10.1145/3591265.

[30] Partha Pratim Ray. An Overview of WebAssembly for IoT: Background, Tools, State-of-the-Art, Challenges, and Future Directions. *Future Internet*, 15(8), 2023. DOI: 10.3390/fi15080275.

[31] Grigore Roșu and Traian Florin Șerbănuță. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. DOI: 10.1016/j.jlap.2010.03.012. Membrane computing and programming.

[32] Rust - The Rust Reference. URL: https://doc.rust-lang.org/stable/reference/ (visited on 07/23/2025).

[33] TinyGo - Go compiler for small places. URL: https://github.com/tinygo-org/tinygo (visited on 07/23/2025).

[34] WABT: The WebAssembly Binary Toolkit. URL: https://github.com/WebAssembly/wabt (visited on 04/18/2025).

[35] wasmer - Fast, secure, lightweight containers based on WebAssembly. URL: https://github.com/wasmerio/wasmer (visited on 08/10/2025).

[36] Wasmtime Docs. URL: https://docs.wasmtime.dev/ (visited on 08/10/2025).

[37] 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.

[38] Conrad Watt. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 53–65, Los Angeles, CA, USA. Association for Computing Machinery, 2018. DOI: 10.1145/3167082.

[39] Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. Two Mechanisations of WebAssembly 1.0. In Marieke Huisman, Corina Păsăreanu, and Naijun Zhan, editors, *Formal Methods*, pages 61–79, Cham. Springer International Publishing, 2021. DOI: 10.1007/978-3-030-90870-6_4.

[40] WebAssembly Specification. URL: https://webassembly.github.io/spec/core/ (visited on 04/18/2025).

[41] WebAssembly Specification - interpreter. URL: https://github.com/WebAssembly/spec/tree/main/interpreter (visited on 04/18/2025).

[42] WebAssembly Specification Core Semantics Tests. URL: https://github.com/WebAssembly/spec/tree/main/test/core (visited on 07/30/2025).

[43] 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. *Commun. ACM*, 53(1):91–99, January 2010. DOI: 10.1145/1629175.1629203.

[44] Geunyeol Yu and Kyungmin Bae. A Flexible Framework for Integrating Maude and SMT Solvers Using Python. en. In Kazuhiro Ogata and Narciso Martí-Oliet, editors, *Rewriting Logic and Its Applications*, pages 179–192, Cham. Springer Nature Switzerland, 2024. DOI: 10.1007/978-3-031-65941-6_10.

[45] Yixuan Zhang, Mugeng Liu, Haoyu Wang, Yun Ma, Gang Huang, and Xuanzhe Liu. Research on WebAssembly Runtimes: A Survey. *ACM Trans. Softw. Eng. Methodol.*, January 2025. DOI: 10.1145/3714465. Just Accepted.

Appendix A

Full module structure

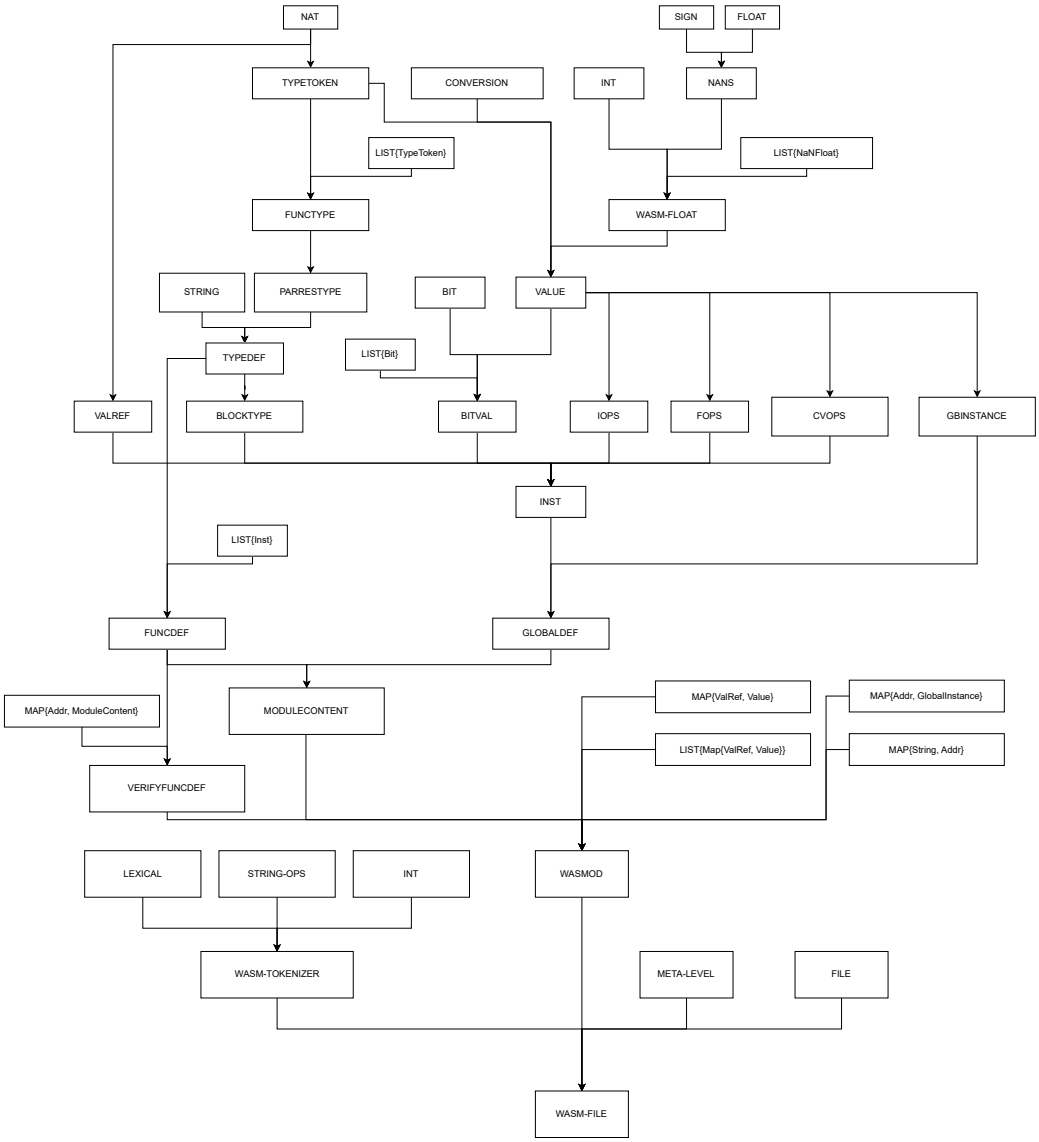

Figure A.1: Module structure