Imperial College London

MASTER THESIS

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

IL2Coq: Automatic Translation of WebAssembly Specification To Coq

Author:
Diego Cupello

Supervisor:
Prof. Philippa Gardner

June 17, 2024

Submitted in partial fulfillment of the requirements for the Meng Computing of Imperial College London

Abstract

WebAssembly is a low-level bytecode language introduced in 2017 by Andreas Hossberg as a compilation target for low-level languages such as C/C++ and Rust [1]. One aspect of WebAssembly that sets it apart from other programming languages is that it boasts an incredibly formal and complete specification for all of its versions.

Writing formal specifications for each new version can be considered tedious and error-prone. It also limits the amount of people who are able to edit the specification. As such, Wasm Spectec was created to host a domain-specific language that, with the specification written in that language, is able to produce the Latex, prose, tests and even an interpreter. It acts as a single source of truth.

The problem with the Wasm Spectec toolchain is that it does not produce mechanized-ready inductive definitions. In the past, mechanizations of Webassembly have shown various issues with the specification itself and has made the specification more robust. IL2Coq acts as a proof of concept solution to this problem, giving a translation process between the intermediate language of the Wasm Spectec toolchain and Coq.

The IL2Coq solution was able to successfully produce the inductive definitions for specifically the WebAssembly 1.0 specification. With these definitions, a proof result of type preservation was given as evidence of the effectiveness of the translated inductive definitions.

Acknowledgments

I would first like to thank my supervisor, Prof. Philippa Gardner, for giving me an incredible amount of support and advice throughout the entire duration of the project. Thank you for keeping me in the correct path! It was a pleasure to work with you!

I would also like to thank the PhD student, Xiaojia Rao, who provided me with great advice and feedback while developing the solution and even through the proof. It was through your guidance that this project came to be!

I also would like to thank my friends and family, who gave me the guidance and perseverance to keep moving forward. Thank you for helping me through this project!

Contents

Chapter 1

Introduction

1.1 Motivation

WebAssembly is a low-level bytecode language introduced in 2017 by Andreas Hossberg as a compilation target for low-level languages such as C/C++ and Rust [1]. The goal was to make a safe, portable and efficient language for the Web, that enables high performance applications. Safety is absolutely essential as it is expected to be sent through the Web through different streams and as such it can easily be exploited if not designed correctly. It must also be portable as being part of the Web ecosystem means that it must be able to support many different types of hardware and systems. It should be able to be agnostic to these changes in order to succeed. It must also be efficient, both in time taken and space used, as current expectations of the Web is that any application should have low load times, and have as low bandwidth as possible through the network. WebAssembly at its current moment has met these goals, being accepted in the Web community and even has been supported by all the major Web browsers [2], having the official standard W3C be published in 2019 [3].

One unusual aspect of WebAssembly is that, before release, it had defined the formal semantics of the language fully. This enables complete mechanisation of the semantics, providing verification that the implementation is sound, and even be able extract a complete interpreter that functions as a source for correctness. In Watt and Rao's paper, they provide a mechanisation in two different proof assistants, Coq and Isabelle, for WebAssembly 1.0 [2].

However, while this may have been a good mechanisation for WebAssembly 1.0, in 2023, WebAssembly 2.0 has come out with new changes to the specification [4]. This would mean that the mechanisation would have to be adapted. The mechanization is done by hand and as such, this task involves modifying a large codebase, becoming increasingly more error-prone as versions keep getting added. To makes matters worse, this also applies to the reference interpreter and the specification itself, as they are largely done manually.

In order to resolve this problem, the WebAssembly community presented the idea of a WebAssembly domain-specific language, named Wasm SpecTec, officially presented in [5] (repository in [6]). The motivation of such a language is to have a single source of information and also allow anyone with sufficient knowledge of the domain-specific language to modify the WebAssembly language given new requirements. The information that it generates would be the latex, prose, tests and the interpreter. It had been recently implemented and has been able to successfully produce the expected artifacts and the interpreter has passed all of the original applicable tests [5]. It has also been tested with new implementations that are currently experimental, such as the addition of garbage collection [7] and threads [8].

The main problem at hand is that mechanisation ready definitions is not generated by Wasm Spectec yet. There is some initial work to use the internal representation of the domain-specific language and make several passes to it in order to transform it into a more proof-friendly way.

The main scope of the project is to translate the internal language of the DSL to inductive definitions and relations that define the WebAseembly language in the proof assistant Coq (targeting specifically WebAssembly 1.0), and then provide a manually written proof in Coq.

1.2 Contributions

This project makes the following contributions:

- Translate the Webassembly 1.0 DSL spec to Coq definitions: the Webassembly 1.0 spec written in Wasm Spectec has been automatically translated to completely compiled Coq inductive definitions and relations ready to be used for mechanisation. More specifically, the datatypes, typing rules and reduction rules are successfully translated to Coq. The translation process is described in section 4.
- Extend Wasm Spectec source: the Webassembly 1.0 spec was extended to include the soundness relations, in order to allow the support of proving type soundness. The extension is described in section 5.
- Type Preservation Proof: with the automatically generated definitions, type preservation has been shown to be true for all configurations defined in the Webassembly 1.0 spec. The proof is described in section 5.

The project can be found as a fork of the Wasm Spectec Github, made publicly available for anyone to see.

Chapter 2

Background

2.1 Language Standardization and Mechanisation

2.1.1 Language Standardization and Validation

Language standardization, in the context of programming languages, is the process of documenting the syntax and semantics of a certain language in a precise manner. It allows compilers to adhere to a single source of truth and build upon it. Types of language standardizations include: EMCAscript[9] for javascript, ANSI/ISO standard for C[10] and C++[11], and the formal semantics of WebAssembly seen in the specification[3].

Language standardization allow for syntax and semantics to be validated in a grand-scale, affecting all implementations of the languages such as compilers and interpreters. Validation in this case refers to the correctness of the syntax and semantics of a language, and the certainty that undefined behaviour is not possible. This can be done through test suites and unit testing which are easier to implement, but correctness properties such as type soundness like you can find in WebAssembly (i.e. the full mechanization in [2]) are often what is necessary to get guaranteed safety.

Validation is starting to become more important as there is a growing need for memory-safe languages and formal methods for all of our software, especially for correctness properties, as noted in a White house paper [12].

2.1.2 Mechanization Definition

Mechanisation, in the context of software verification, is the process of converting proof-checking by hand into one that can be run by the computer. When proving that a given piece of software or specification is correct, the proofs can start getting really large, up to the point that it is no longer sustainable to do by hand. As such, proof assistants were made.

A proof assistant, also known as interactive theorem provers (ITP), is regarded as a

formal proof management system that enables development of large-scale proofs. It involves human collaboration to guide the computer to reach a certain proof goal, making sure that every step of the proof is indeed valid. These are some known proof assistants and large-scale language specification proofs done in them:

- Coq[13]: software toolchain for the verification of C programs known as CompCert[14], mechanized specification of Javascript known as JSCert[15], and also the full mechanization of WebAssembly semantics, known as WasmCert-coq [2].
- Isabelle[16]: Formalization of C (C11)[17], formal verification of Java compiler[18], and also full mechanization of WebAssembly semantics, known as WasmCert-Isabelle[2].
- Lean[19]: mainly used for mathematical proofs.

This project focuses on automated mechanization of WebAssembly. Since there was already work done in Coq for WebAssembly (WasmCert-coq), we chose it to be the target for the automated inductive definitions.

2.1.3 Coq

Coq is a widely recognized proof assistant that has been used for many large-scale proofs. The main features of Coq include: defining and stating functions and mathematical definitions, interactively develop proofs through these theorems, perform machine checking of these proofs [13]. This section will give a small overview of the functionalities of Coq, and later on we will address some more technicalities when needed.

Figure 2.1 gives an inductive structure corresponding to a list. Inductive structures give certain conditions for how a certain type can be constructed. For example, A list can either be nil or cons an element to another list. It has to be the case that the list always ends at nil, or else it would just be an infinite chain of the constructor cons, which is never really the case for finite structures. Inductive types are the main essence of Coq, and all of the type utilized adhere to this, even numbers, as shown in the figure 2.1. Inductive types can also be dependent types, which means it has a certain type it depends on. A notable example is the list inductive type, which has X as a type it depends on.

Then, with these inductive types, it is possible to define function definitions, as shown with the length function by going through the entire structure and pattern matching for each case.

Finally, with these structures in mind, one can also produce lemmas, theorems, corollaries, like the lemma app_length, which basically states that the length of the concatenation of two lists is the same as the length of one list plus the other. Each proof step uses a proof command known as a tactic. The main ones used here are pretty straightforward: intros introduces all of the variables and hypotheses to the premise,

simpl simplifies the LHS and RHS of the goal, rewrite modifies the given side of the goal using a hypothesis or already proven fact, and induction goes through all of the cases for the structure, giving inductive hypotheses for the inductive cases (in this case, it was nil for the base case and cons for the inductive case). There are a vast number of tactics that are useful and are widely used (in fact one could define their own tactic), but these are out of the scope for this paper. These examples were adopted by the software foundations book as a guiding example [20].

```
Inductive list (X:Type) : Type :=
  | nil
  | cons (x : X) (l : list X)
Inductive nat : Type :=
  | zero
  | succ (n : nat)
Fixpoint length {X : Type} (xs : list X) : nat :=
  match xs with
    | nil => 0
    | cons x xs' => 1 + my_length xs'
  end.
Lemma app_length : forall (X:Type) (l1 l2 : list X),
    length (l1 ++ l2) = length l1 + length l2.
Proof.
  intros x l1 l2. induction l1 as [| x1 l1' IH].
    (*nil base case*)
    - simpl. reflexivity.
    (*inductive case l1 = cons x1 l1' *)
    - simpl. rewrite -> IH. reflexivity.
Qed.
```

Figure 2.1: Coq Example

2.2 WebAssembly

The following sections will give an overview on the core concepts of WebAssembly, as well as go through the compilation phases: Validation, Instantiation and Execution. During the Execution section, it will mainly go through the run-time semantics that WebAssembly inhibits. Finally, this will end with a brief look on how the syntax of WebAssembly is turned into bytecode in the binary format section. This section mainly goes through version 1.0 of WebAssembly, but the differences in versions are not too big. Most of the information is gotten from the current specification itself [3]. The binary format and instantiation sections are not very important for the translation process, and as such will only be discussed briefly.

2.2.1 Concepts

Webassembly is revolved around the following concepts [3]:

- Values: In WebAssembly, program number values are of type integer i32, i64 and float f32, f64. At all points of the execution of a WebAssembly program, the semantics expect any of these types in the stack. In addition to this, there is also a vector type v128 that represents many packed types. Finally, there is also opaque reference types which are not observable unlike the other types. We will omit v128 and the corresponding SIMD instructions as they are not necessary for the scope of the paper. The values are representated as follows in the abstract syntax:

$$\text{(value types) } t ::= i32 \mid i64 \mid f32 \mid f64$$
$$\text{(function types) } ft ::= t^* \rightarrow t^*$$

- Stack: WebAssembly is a stack-based language. This means that during the execution of the program, there is an implicit stack that holds an arbitrary amount of values. Instructions push and pop from the stack as needed for the operation to happen. There is no separate representation of the stack apart from the instruction sequence. As an example, the instruction t.add would pop two operands of value type t and push a new value of type t to the stack. The stack should be statically known from all program points. Figure 2.2 shows how the stack would be reduced during program execution. Each binary operation takes the previous operands, consumes them and pushes the expected value.

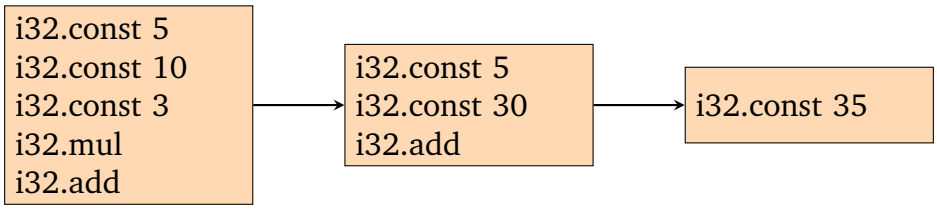

Figure 2.2: WebAssembly Stack Reduction

- Control Flow: WebAssembly follows a different approach to control flow as other stack-based languages. Normally what would be seen is the possibility of unconditional jumps or condition jumps, such as branching to any sort of label. This is risky and complicates control flow. WebAssembly provides a structured control flow, which restricts the program in the following ways: the instructions **if**, **block**, and **loop** must have an end label, and branches behave as a standard break seen in the C programming language when in an **if** or **block** sequence, and as a continue in the **loop** sequence. This gives Webassembly the property of being validated and compiled in a single pass. It also makes the program easier to read. Figure 2.3 shows how the **if** instruction would be used. It gets converted into a **block** statement once it has been decided what path to

take. Then the **block** gets reduced to a **label** statement, which gives a target for any branch instruction to index to. Then, the **label** statement reduces to the constant.

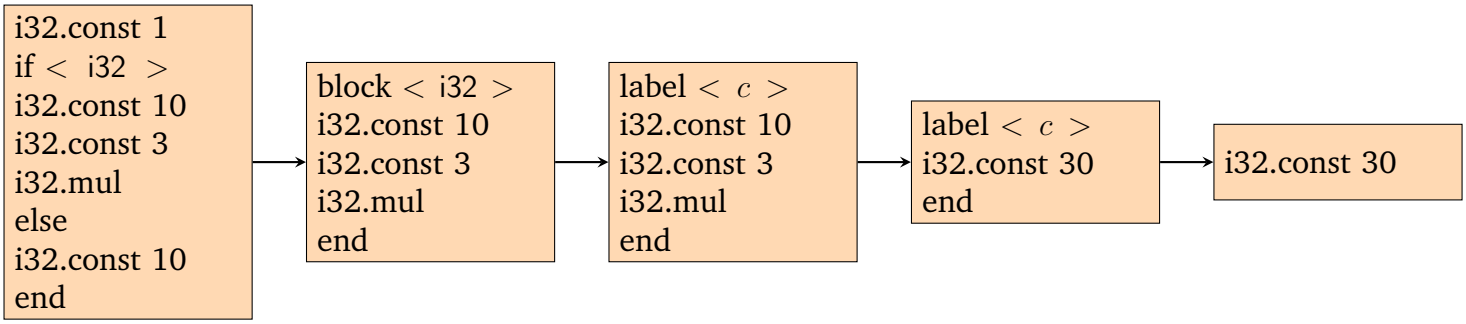

Figure 2.3: Control Flow Example

- Memory, Tables, and Globals: WebAssembly modules data consist of the global values, linear memory and function tables. Global values can be either mutable or immutable and accessed by any of the functions defined in the module. It can be statically instantiated. Memory is a linear stream of bytes, which can be imported. Programs can either load or store from this memory, and can dynamically grow the memory (with the **memory.grow** instruction) and can check the size of the memory at any moment (with the **memory.size**. Memory must specify a minimum size and can optionally have a maximum size. Function tables represent an opaque table of function pointers that can be used with dynamic dispatch to give the functionality of function pointers. The abstract syntax of all three is shown below, where *n, m* represent the minimum and maximum starting size, *mut* the mutability of the global value, *t* the value type and *e* the expression that the global value results to.

$$\text{(globals) } gb ::= \textbf{gb}\, mut\, t\, e$$
$$\text{(memories) } mem ::= \textbf{mem}\, n\, m$$
$$\text{(tables) } tab ::= \textbf{tab}\, n\, m$$

- Modules: WebAssembly is packaged in the form of a module, which consists of function types, definitions of functions, global values, tables and memories. It also consists of other concepts like elem and data, which represent static value instantiations of tables and memories, and also imports and exports, which specify what gets imported/exported in the module. These definitions can either be imported, which would specify the name and implementation, or exported, so it can be used outside of WebAssembly. The abstract syntax of a module is shown below. It is represented as a record of all the definitions described above. The term vec(t) stands for a vec of term t.

$$module ::= \{types :: vec(ft), funcdefs :: vec(func), globals :: vec(gb), memories :: vec(mem), tables :: vec(tab), data :: vec(data), elem :: vec(elem), imports :: vec(import), exports :: vec(exports)\}$$

- Traps: Whenever the program is executing and for some instructions, if some conditions fail to be met, then the program will produce a trap and it will be propagated outward until the final result is a trap. WebAssembly programs cannot handle a trap and must be handled by the outside environment. For example, when accessing the memory, if the instruction **t.unop**, which represents all unary operations for value type *t*, does not return a defined value, the semantics state that it should return a trap. [3]

With these concepts, it is starting to show how the abstract syntax of WebAssembly is shaped. Figure 2.4 shows the syntax of all the concepts outlined above, and some instructions for brevity. Some of the instructions that were not mentioned above but still interesting to note are **nop**, which does nothing and leaves the stack unmodified, **unreachable** which produces an unconditional trap, and **drop** which drops an operand from the stack [3]. These are special instructions that are necessary to produce specific control scenarios that ensure soundness.

(value types) $t ::= \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64}$
(packed types) $tp ::= \mathsf{i8} \mid \mathsf{i16} \mid \mathsf{i32}$
(function types) $ft ::= t^* \to t^*$
(instructions) $e ::= t.\textbf{const}\, c \mid t.\textbf{unop} \mid t.\textbf{binop} \mid \textbf{loop}\, ft\ e^* \mid \textbf{if}\, ft\ e^* \mid \textbf{block}\, ft\ e^* \mid \textbf{nop} \mid \textbf{unreachable} \mid \textbf{drop} \mid \textbf{local.get}\, i \mid \textbf{local.set}\, i \mid \textbf{memory.size} \mid \textbf{memory.grow} \mid \textbf{br}\, i \mid \textbf{br\_if}\, i \mid \textbf{br\_table}\, i^+ \mid \textbf{global.get}\, i \mid \textbf{global.set}\, i \mid t.\textbf{load}\, mem\_info \mid t.\textbf{store}\, mem\_info \mid ...$
(globals) $gb ::= \textbf{gb}\, mut\ t\ e$
(memories) $mem ::= \textbf{mem}\, n\ m$
(tables) $tab ::= \textbf{tab}\, n\ m$
(modules) $module ::= \{types :: vec(ft), funcdefs :: vec(func), globals :: vec(gb), memories :: vec(mem), tables :: vec(tab), data :: vec(data), elem :: vec(elem), imports :: vec(import), exports :: vec(exports)\}$

Figure 2.4: Abstract Syntax

2.2.2 Validation

For a WebAssembly module to be instantiated, it must first be validated. For it to be validated, a type system of the abstract syntax of a module is formed. The typing system revolves around building a valid typing judgement of shape $C \vdash e^* : ft$ where *C* is the typing context. The context is a record of all of the typing entries of all entities that are accessible at that given point of the program [1]. For this reason, the context should have information on all functions types in the module, the global types, the local types (that are defined at that program point), memory types, table types, block label types, and return types. With the context, it should be enough to prove that a given instruction sequence *e* has type *ft*. Figure 2.5 showcases some of the typing rules that are in the specification of WebAssembly, to illustrate how the type system is formed. They have the following behaviour:

- **nop**: This instruction simply takes no operands from the stack and outputs nothing. This is reflected by the empty brackets $[\,]$.

- *t*.**unop**: Unary operations must syntactically have a type *t* that must correspond to the input operand type and output operand type. Crucially, the typing rule states that it must only consume one operand from the stack.

- i32.**add**: This specific binary operation already has a specific value type (*i32*), so it must have two input operands from the stack of that type, and it will output i32 to the stack.

- **local.get**: This local variable instruction first has the precondition that the context must have $C.locals[x] = t$ defined, as in, the function body must have a local value accessed by index *x* with type *t*. This precondition implicitly implies that the index is bound. We then have the type consist of consuming no operands and outputting the same type *t*.

- **drop**: This parametric instruction just simply removes an operand from the stack. As there is no indication of what the type *t* should be, this instruction is value-polymorphic, which means that one or more of the stack operands is unconstrained. The type can either be chosen arbitrarily during type-checking, as long as it fits with the entire static checking process [3].

- **func**: This typing rule is not an instruction, but a set up of the module, specifically for the function definitions. The first precondition consists of requiring the context to have the type accessed by index *x* (provided by the syntax), and have the corresponding function type. The next precondition then appends all of the corresponding types to their respective places to the context, having locals have the parameters and local variables types, labels have the result type of the function (as labels are represented by their result type), and return also have the result type. With this new context, it must be the case that *expr* can be proven to have the result type of the function. With these preconditions true, then the entire function definition definable with context *C* must then have the correct type.

$$\text{NOP}\ \frac{}{C \vdash \textbf{nop} : [\,] \to [\,]} \qquad \text{UNOP}\ \frac{}{C \vdash t.\textbf{unop} : [t] \to [t]} \qquad \text{ADD}\ \frac{}{C \vdash \text{i32}.\textbf{add} : [\text{i32}\,\text{i32}] \to [\text{i32}]}$$

$$\text{LOCAL.GET}\ \frac{C.locals[x] = t}{C \vdash local.get\,x : [] \to [t]} \qquad \text{DROP}\ \frac{}{C \vdash \textbf{drop} : [t] \to [\,]}$$

$$\text{FUNC}\ \frac{C.types[x] = [t_1^*] \to [t_2^*] \quad C, locals\,t_1^*\,t^*, labels\,[t_2^*], return\,[t_2^*] \vdash expr : [t_2^*]}{C \vdash \{type\,x, locals\,t^*, body\,expr\} : [t_1^*] \to [t_2^*]}$$

Figure 2.5: Typing Rules

With figure 2.5 in mind, it should be clear now that the WebAssembly type system is simple enough to be able to go through a single pass during type checking.

2.2.3 Instantiation

Once a module has been validated, the module can be instantiated. Instantiation refers to the phase before type-checking and execution that creates the WebAssembly module dynamically, creating a store that is the old store of module defined states, appended with all of the new instances [3]. It also initializes the memory, globals and tables. More importantly, it also checks the module imports are satisfied by checking that the types declarations make sense.
The store refers to the representation of all the instances of functions, tables, memories, globals, data segments and elem segments. Each one of these have their own grammar and intricacies that can be seen clearly in the specification. I will go briefly over functions to give an example.

Function instances are effectively represented as closures of the actual function. It contains its function type, the module instance to be able to resolve references from other parts of the module (for the function call instruction) and then the actual function. Additionally, one can also represent host functions by having its type and then the actual representation of the host function. It is represented mathematically as follows [3]:

$$\begin{aligned} funcinst ::= &\{type\ functype, module\ moduleinst, code\ func\} \\ &| \{type\ functype, hostcode\ hostfunc\} \end{aligned}$$

2.2.4 Run-time Semantics

Once the WebAssembly module has been validated and the module has been instantiated, it can then be executed. The specification utilizes operational semantics to specify the runtime execution of a WebAssembly module. Its behaviour is modelled in terms of an abstract machine which models the program state. The state consists of the implicit stack, the abstract store containing the global state of the machine [3].

The global state of the machine, more known as its configuration, more concretely refers to the store mentioned in the instantiation section, the call frame of the current function, and the current instruction sequence. Frames (known as activation frames), refer to a structure that gets put on the stack that contains the information of the local values and the module instance. This is done in order to for function calls to be able to return to the original state after the function returns. This configuration is normally represented as a tuple $(S; F; e^*)$).

As stated above, the runtime semantics are based on operational semantics. The reduction rules are in terms of small-step reduction, which are based on small, compact rules that clearly state what the instruction does in a single step [1]. The stack

is not explicitly shown in the reduction rules, but instead is shown in the format of *t*.const rules [2], as shown in the following mul example:

$$(\text{i32.}\mathbf{const}\, i)(\text{i32.}\mathbf{const}\, j)\ \text{i32.}\mathbf{mul} \hookrightarrow (\text{i32.}\mathbf{const}\, k)\ (if\ k \in mul_{\text{i32}}(i,j))$$

$$(\text{i32.}\mathbf{const}\, i)(\text{i32.}\mathbf{const}\, j)\ \text{i32.}\mathbf{mul} \hookrightarrow trap\ (if\ mul_{\text{i32}}(i,j) = \{\})$$

This reduction rule consumes the constants i and j and then returns k as long as it is defined, which in this case it will most likely be the case. If not, then it would produce a trap. The following example is a more complicated reduction, showing how the if instruction would get reduced:

$$(\text{i32.}\mathbf{const}\, i)\ if\ t^n\ instrs_1^*\ else\ instrs_2^*\ end \hookrightarrow block\ t^n\ instrs_1^*\ end\ (if\ i \neq 0)$$

The If rule consumes the constant i, checks that it is not equal to zero, and if so, it reduces the whole if statement of type t^n to the first instruction sequence $instrs_1^*$, wrapped in a block of the same type.

Another aspect of the run-time semantics worth mentioning is the usage of administrative instructions. These instructions are an extension to the original instruction set to aid with the reduction of calls, control and traps. One worth mentioning is the label instruction which was shown before for control instructions. It represents the label that it is implicitly on the stack [3]. It has a nesting structure that allows the preservation of the original structured flow.

The last aspect worth mentioning of the run-time semantics is Numerics. The way that numbers are represented are in their usual binary representation. Integers are represented as unsigned binary numbers, having mathematical operators use signed conversion to convert it to its signed format for the calculation and then returned back to its original representation. Floating-point numbers are represented by the IEEE-754 [21] representation. The bits are stored generally as little endian.

2.2.5 Binary Format

I will briefly go over the binary format of a WebAssembly module. It is encoded as its abstract syntax shown above, in a linear fashion. It is defined by an attribute grammar and it is also considered a well-formed encoding if and only if it is described by the grammar [3].

To give some examples of the grammar, figure 2.6 shows how it would be encoded and decoded as. The example for the block instruction shows how to utilize the grammar in its recursive nature. It also makes use of an explicit opcode (*0x0B*) which represents the term end. The example also shows us the one guiding principle in this grammar, which is that each grammar rule only produces exactly one synthesized attribute. This would be the byte sequence that is shown in the left hand side of the rule. This makes it quite simple to be able to decode[3].

It is also worth noting how modules are represented. The entire WebAssembly module is separated into different individual sections that match the structure of the module record shown in the abstract syntax. The only separation made is for function definitions which separates the type declarations and the actual implementation [3].

$numtype ::= 0x7F \Rightarrow \mathsf{i32} \mid 0x7E \Rightarrow \mathsf{i64} \mid 0x7D \Rightarrow \mathsf{f32} \mid 0x7C \Rightarrow \mathsf{f64}$
$instr ::= 0x00 \Rightarrow unreachable$
$\mid 0x01 \Rightarrow nop$
$\mid 0x02\ bt : blocktype\ (in : instr)^*\ 0x0B \Rightarrow block\ bt\ in^*\ end$
$\mid 0x0C\ l : labelidx \Rightarrow br\ l$
$\mid 0x6A \Rightarrow \mathsf{i32}.add$

Figure 2.6: Selected Grammar rules

2.3 WasmCert

With WebAssembly having its semantics formally specified, it would simply make sense for its specification to be completely mechanized. With this in mind, there are already complete mechanisations done in this language, shown in the paper done by Watt and Rao in ”Two Mechanisations of WebAssembly 1.0” [2]. In fact, Watt had already done some part of the mechanisation in 2018 in his paper ”Mechanising and Verifying the WebAssembly Specification” [22]. At the moment, the focus is on WasmCert, which came from Watt and Rao’s paper, but more specifically the Coq version of it, as it is more relevant to the scope of the paper. It closely follows the Isabelle version so it should be sufficient. When saying WasmCert from know on, it means the WasmCert-coq version of it, repository in [23].

2.3.1 DataTypes

The Coq mechanisation starts with declaring the abstract syntax of WebAssembly. It does so by creating many inductive types for each part of the module. Figure 2.7 shows how the instructions are implemented in Coq [2]. For some of the instructions, not much is needed besides a good name to indicate which one it is, as this is only the syntax. As shown in the Coq section, we can prove results with proof by induction with later it will prove useful. It is good to note the BI_loop example as it shows that it requires a function type, a list of instructions and it returns an instruction.

With all of these definitions well-formed, you can start to reason through these and derive some results, in which ultimately they reach the soundness result that is required in the WebAssembly specification. These definitions can be found in the datatypes.v file in WasmCert [2].

```
Inductive basic_instruction : Type := (* be *)
  | BI_unreachable
  | BI_nop
  | BI_drop
  | BI_select
  | BI_block : function_type -> list basic_instruction -> basic_instruction
  | BI_loop : function_type -> list basic_instruction -> basic_instruction
  | BI_br : immediate -> basic_instruction
  | BI_current_memory
  | BI_grow_memory
  | BI_const : value -> basic_instruction
```

Figure 2.7: Basic Instructions

2.3.2 Typing

For the typing rules, there is an induction definition which encompasses all of the typing rules for the instructions. Not only that, but there are additional rules that serve as wrappers for the store and frame typing, as well as useful functions for checking if the types agree with each other. The typing context was defined in the datatype part as a Coq record, which is a data structure that allows for static storage of many different types [13]. As a typing context has many components, it is rather the correct structure to use.

Figure 2.8 shows how many of the instruction rules would be written to give an example. The inductive definition be_typing takes a typing context, a sequence a basic instructions and a function type to return a proposition. Looking at the example of bet_get_local, it assumes that i, being the index to the associated type in the local values of the typing context, is smaller than the size of the context, and it also assumes it can find some type *t*. As you can see, these mimics the typing precondition of get local seen in the Validation section. If both of those are true, then it must be the case that we have a the correct typing with the instruction being [::BI_get_local i], where BI_get_local is the constructor for the instruction $local.get\, i$, and the type being (Tf [::] [::t]), where Tf is the constructor for function types. Clearly this represents $C \vdash local.get\, i : [\,] \to [t]$.

Another thing to note about the typing rules is the extension of the typing judgements in order to satisfy type soundness. In Watt and Rao's paper [2], they present some additional judgements in efforts of better defining type soundness. Figure 2.9 shows configuration validity, which defines the typing judgement for type soundness in the highest level. The precondition $\vdash_s S : ok$ means that the store has to be well-formed in terms of the size of the memories and tables, and the types have to agree with each other. The precondition $S; \epsilon \vdash_{loc} F; e^* : [t^*]$ defines the type of the instructions under the given function frame [2]. This judgement, named local

```
Inductive be_typing : t_context -> seq basic_instruction
    -> function_type -> Prop :=
| bet_const : forall C v, be_typing C [::BI_const v] (Tf [::] [::typeof v])
| bet_unop : forall C t op,
    unop_type_agree t op -> be_typing C [::BI_unop t op] (Tf [::t] [::t])
| bet_binop : forall C t op,
    binop_type_agree t op -> be_typing C [::BI_binop t op] (Tf [::t; t] [::t])
| bet_get_local : forall C i t,
  i < length (tc_local C) ->
  List.nth_error (tc_local C) i = Some t ->
  be_typing C [::BI_get_local i] (Tf [::] [::t])
```

Figure 2.8: Basic Typing

validity, is given in more detail in WasmCert. These judgements are necessary as they enforce that the given store, function frame, and instances are valid over the enclosing instruction sequence.

$$\frac{\vdash_s S : ok \quad S; \epsilon \vdash_{loc} F; e^* : [t^*]}{\vdash_c S; F; e^* : [t^*]}$$

Figure 2.9: Configuration Validity

2.3.3 Reduction

Reduction rules can now be defined. It follows the same format as the operational small-step semantics shown in section 2.2.4. It is separated into two different inductive types, reduce simple and reduce, where reduce simple do not need reference to the store, frame and host state. Figure 2.10 shows how reduce_simple is defined for the binary operations case as a small example. The inductive structure takes in the previous sequence of instructions and the next one and returns a proposition. The first case depicts a success case where it uses a helper function app_binop, (which makes use of Coq's option type to represent possibly undefined values) to check that it is defined in order to create the full reduction. If it is ill-defined it will make the failure case true. These rules can be found in the opsem.v file in WasmCert [2].

2.3.4 Type Soundness

Now that we have the building blocks of the WebAssembly specification defined (and some additional utility definitions/theorems to make the proof work which we omit for brevity), it is possible to prove type soundness.

```
Inductive reduce_simple : seq administrative_instruction
    -> seq administrative_instruction -> Prop :=
(** binop **)
  | rs_binop_success : forall v1 v2 v op t,
    app_binop op v1 v2 = Some v ->
    reduce_simple [::AI_basic (BI_const v1); AI_basic (BI_const v2);
        AI_basic (BI_binop t op)] [::AI_basic (BI_const v)]
  | rs_binop_failure : forall v1 v2 op t,
    app_binop op v1 v2 = None ->
    reduce_simple [::AI_basic (BI_const v1); AI_basic (BI_const v2);
        AI_basic (BI_binop t op)] [::AI_trap]
```

Figure 2.10: Reduction defined in Coq

Type soundness represents two properties: preservation and progress. Preservation in terms of type theory means that if a certain program has a specific type and it can be reduced to a different program, then the resulting program will preserve the original type. This is an important property to have as it enables us to determine the resulting type of the whole program while being reduced, without doing any reduction ourselves. As such, we can statically type check the program. More formally, preservation is given as follows:

$$\text{If } \vdash_c S; F; e^* : [t^*] \text{ and } S; F; e^* \hookrightarrow S'; F'; e'^*, \text{ then } \vdash_c S'; F'; e'^* : [t^*] \text{ and } S \prec_s S'$$

The property has been lifted to all for tracking of the store and frame which is assumed to be potentially modified during a reduction. The relation $S \prec_s S'$ refers to a restriction on the store for WebAssembly programs that state that the store can not be decrease in size. It also ensures that it preserves the original parts of the store [2].

The progress property is defined as a certain program always being able to make some sort of progress in the reduction. As in, it either ends up in a terminal state or it is able to be reduced further into a new state, provided it has a well-defined type. More formally, progress is given as follows:

$$\text{If } \vdash_c S; F; e^* : [t^*], \text{ then is-terminal}(e^*) \vee \exists S' F' e'^*.\ S; F; e^* \hookrightarrow S; F; e'^*$$

Chapter 3

Wasm Spectec

The following sections will give an overview on the WasmSpectec toolchain, as well as the internal language (IL) core features.

3.1 Wasm Spectec Toolchain

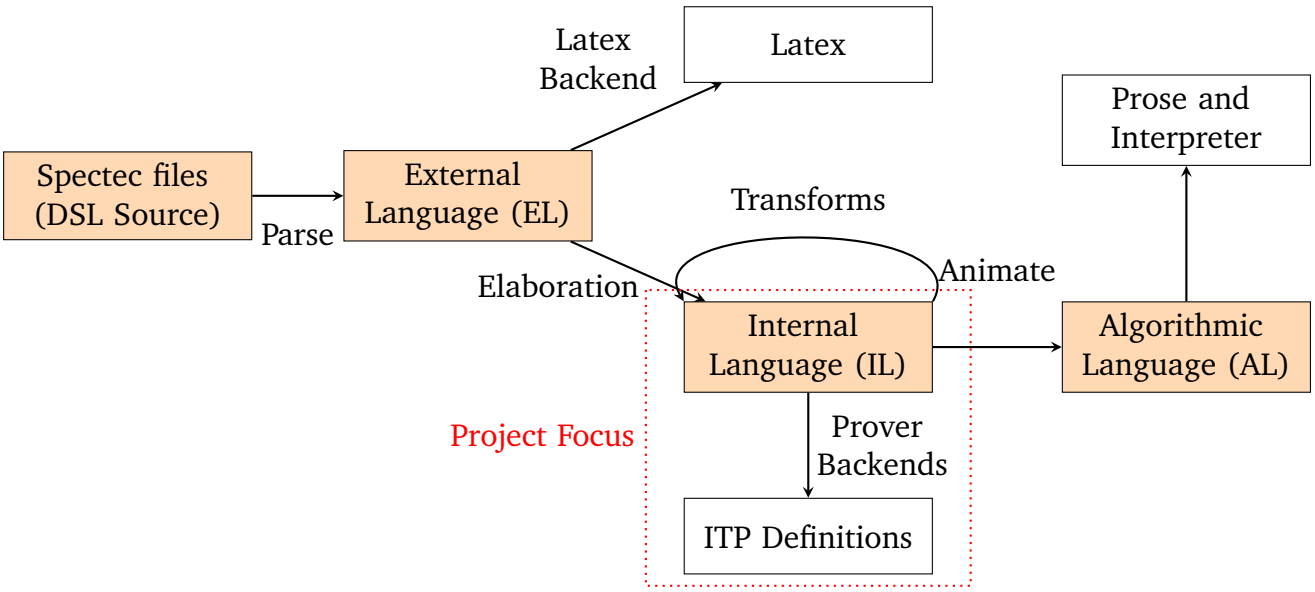

Figure 3.1: WasmSpectec Toolchain inspired by [24]

Figure 3.1 shows the overall tool chain for the DSL. The toolchain gives the ability to produce multiple backends from the input, also known as the DSL source [6]. The toolchain is written in Ocaml, a multi-paradigm, but mainly functional programming language [25]. The workflow starts with the parsing of the DSL, turning it into an abstract syntax tree known as the external language (EL). The EL can then be transformed into the internal language (IL). The IL is merely the EL with some additional information inferred such as the bindings of variables in definitions and rules.

Then, with the IL, one can either specify to generate ITP definitions, or transform the IL to a more algorithmic representation of the DSL source, known as Algorithmic Language (AL). AL is a more restricted version of the IL that enforces and algorithmic order of evaluation [24]. The AL is out the scope of the project, so only a high-level description is provided.

The toolchain then can use all of these languages to generate the following[24]:

- Latex: The latex backend translates the EL to the latex representation similar to the already established formal specification noted in [3]. This only gives the rules and definitions without the actual explanation of such rules.
- Prose: The prose backend provides the pseudo-algorithms for the reduction rules using directly the AL.
- Interpreter: The Interpreter backend interprets the AL which represents the Webassembly semantics.
- Prover Backends: The Prover backends transform the IL to inductive definitions and relations in ITPs that can later be used for mechanization.

The main focus of the project is to create a Prover backend for Coq for the Wasm-spectec toolchain. As such, the EL and AL are not going to be discussed in detail, as well as the other backends. However, the IL and its transformations will be discussed in the next section. All of the information derived from the IL comes from the code given in the repository: [6].

3.2 Internal Language Core Features

The main aim of this section is to give a thorough overview of the IL AST. The reason for this is because the documentation of the IL is not complete yet, and as such one would need to resort to the code to understand the IL. Furthermore, almost all parts of the IL are utilized in the translation process.

Figure 3.2 shows the AST for the IL language, based on the print output given in the Wasm Spectec toolchain (this is not a backend output). This gives the complete picture of how the IL is constructed. The only portion that is hidden is the annotations inside inductive types, definitions and relations, as they are not printed. The following subsections it will attempt to go through all of these components and explain them in more detail. Some of the sections are not absolutely necessary to understand the translation process. As such, I marked the optional sections with an asterisk at the end.

3.2.1 Annotations (Hints)*

Most of the Wasm Spectec toolchain operates with an AST which for the most part gives sufficient information for the given problem at hand. However, some infor-

(variable id) id ::= `lower-case-identifier`
(atoms) $atom$ ::= `upper-case-identifier` | `->` | `_|_` | ... (mixops) $mixop$ ::= $atom^{**}$
(iteration) $iter$ ::= $? \mid * \mid + \mid \texttt{\^{}}exp$
(annotations) $hint$ ::= $\{id, exp^*\}$
(binds) $bind$ ::= id : typ | syntax id
(number types) $numtyp$ ::= nat | int | rat | real
(types) typ ::= bool | $numtyp$ | text | $id(arg^*)$ | $typ_1 * typ_2 * ...typ_n$ | typ^{iter}
(user-defined types) $deftyp$ ::=
`(Type Alias)` typ |
`(Inductive)` $(\mid mixop\{bind^*\}(typ_1, typ_2, ..., typ_n)$ `--`$prem^*)^*$ |
`(Record)` $\{(atom\{bind^*\}\ typ$ `--`$prem^*,)^*\}$
(unary operators) $unop$ ::= $\sim \mid - \mid + \mid +- \mid -+$
(binary operators) $binop$ ::= $\wedge \mid \vee \mid$ `=>` | `<=>` | $+ \mid - \mid / \mid \backslash \mid$ `^` $\mid = \mid \neq \mid$ `<` | `>` | `<=` | `>=`
(expressions) exp ::= $id \mid bool \mid nat \mid text \mid unop\ exp \mid exp\ binop\ exp \mid (exp^*) \mid exp.i \mid$
$mixop\ exp \mid (exp)? \mid (exp)! \mid \{(atom\ exp)^*\} \mid exp.atom \mid exp$ `++` $exp \mid$ `[`$exp_1 ... exp_n$`]` $\mid$
$\mid exp \mid \mid exp$ `::` $exp \mid exp[exp_{idx}] \mid exp[exp_i : exp_j] \mid exp[path = exp_{val}] \mid exp[path =$
$.. exp_{val}] \mid id(arg^*) \mid (exp)^{iter}\ \{(id : typ)^*\} \mid exp$ `:` typ_1 `<:` typ_2
(paths) $path$ ::= $path[exp_{idx}] \mid path[exp_i : exp_j] \mid path.atom$
(premises) $prem$ ::= if $(exp) \mid$ otherwise $\mid (prem)^{iter}\{(id : typ)^*\} \mid id : mixop(exp)$
(args) arg ::= exp | syntax typ
(params) $param$ ::= $id : typ$ | syntax id
(Instance of type definition) $inst$ ::= syntax $id\{bind^*\}(arg^*) = deftyp$
(Inductive Relation Rules) $rule$ ::= rule $id\{bind^*\} : mixop\ exp$ `--`$prem^*$
(Auxiliary Definition Clauses)
$clause$ ::= $def\ \$id\{bind^*\}(arg_1, ..., arg_n) = exp$ `--`$prem^*$
(High-level definitions) def :=
`(Defining Single Type)` $inst$ | `(Defining Family Type)` syntax $id(param*)\ inst*$ |
`(Inductive Relations)` $relation\ id : mixop(typ\_1, ..., typ\_n)\ rule^*$ |
`(Auxiliary Definitions)` $def\ \$id(param_1, ..., param_n) : typ\ clause^*$ |
`(Recursive High-level defs)` rec $\{def^*\}$

Figure 3.2: IL AST

mation such as descriptions of certain types, and partial functions, doesn't really fit neatly into the AST. As such, the hint command is given to users to supply the information when needed.

Hints are defined as a record of two elements: the id, which represents the specific command you want for a certain element in the AST, and expressions, which are given as strings to be parsed later when needed. Figure 3.3 shows how the partial command may be used to represent partial functions. The IL representation after the totalize transform utilizes the partial keyword to transform partial functions into total functions using the optional iteration. Iteration and transforms are discussed later in the chapter.

```
def $signif(N) : nat hint(partial)
    def $signif(32) = 23
    def $signif(64) = 52

def $signif(N : N) : nat?
    def $signif(32) = ?(23)
    def $signif(64) = ?(52)
    def $signif{x0 : N}(x0) = ?()
```

Figure 3.3: The partial annotation and IL representation (after totalize)

3.2.2 Atoms and Mixops

The IL allows support for special kinds of identifiers known as atoms and mixops. Atoms are represented as upper-case identifiers or special identifiers such as bottom (_|_), representing falsity, or arrow (->), which is used a lot to represent a function type in a convenient manner).

Mixops are a list of a list of atoms, where each separation of the list of atoms denotes where a variable is placed in the DSL source. In the printing of the IL, these locations where variables are meant to be placed are denoted by the special character '%'. Figure 3.4 gives an example of how mixops are used to give the user freedom to represent certain identifiers in a expressive manner. The top identifier is how the IFELSE instruction is written in the DSL source for Webassembly 1.0 and the bottom is the IL representation (spaces were added between % and atoms for clarity, and variable bindings were omitted).

```
IF blocktype instr* ELSE instr* =>

`IF % % ELSE %`(blocktype : blocktype, instr* : instr*, instr*)
```

Figure 3.4: Mixops represented in DSL Source and IL

3.2.3 Bindings, Arguments, and Parameters*

In the IL, variables are placed in three different categories. Each category has a expression variant (which is either a normal variable or expression) and a generic type variant (indicated by the word syntax). The three categories are:

- Bindings in the form $bind ::= id \ : \ typ \mid \text{syntax } id$: this structure was added to the IL to give extra information on all variables that are utilized in the inner-portions of the AST, including free variables that do not appear in match arguments and parameters, but do appear in the premises.

- Arguments in the form $arg ::= exp \mid \text{syntax } typ$: these mainly represent values and variables that are allowed to also be expressions. As an example, in the case of matching, they can be match patterns instead of a simple identifier (such as list cons $X :: XS$). Another example would be dependent type arguments.

- Parameters in the form $param ::= id : typ \mid \text{syntax } id$: this is mainly used in high-level definitions declarations (such as function definition type declaration and relation type declaration), that do not allow any expression, and as such these parameters must be identifiers.

3.2.4 Basic Types and Iterations*

The IL provides support for various primitive types: different number types (integers (int), natural numbers (nat), rational numbers (rat) and real numbers (real)), booleans (bool) and strings and characters (text).
With these primitive types and user-defined types, the IL also supports n-ary tuple types of the form $(typ\_1 * ... * typ\_n)$ and iteration types of the form typ^{iter}. Iterations can be of four different forms:

- optional type of the form ?.

- any list of the form *.

- non-empty list of the form +.

- list of a specified size of the form ^exp.

These iterations all have an implicit subtyping in the straightforward way, such as optional being a subtype of all of the other lists, and the non-empty list being a subtype of the any list iteration.

3.2.5 User-defined types

Along with the basic types, The IL allows extensive support for three different types that users are allowed to define: inductive types, type aliases and record types. These can be of a single instance or a family type. User-defined types, when defined, can be specified in the form $id(arg*)$ where id is the name given through the definition, and arg^* is a list of type arguments for dependent types. We will go through how the three different types can be defined.

All of the three types are defined with an identifier representing its name and arguments representing the dependent type arguments. Then, each version has their own construction.

Inductive types are defined in the form of a list of constructors, with each of the entries having a mixop representing its name, a tuple type, a list of variable bindings utilized in the tuple type, premises constraining the constructor, and a list of hints. Premises are discussed later in section 3.2.9. Inductive types in the IL also allow parameters to the base type, allowing for dependent types to be constructed. Inductive types are also allowed to form recursive structures, and even have mutual recursion with other inductive types. Figure 3.5 gives an example of how the byte type is represented for Webassembly, the external type, and how lists are represented. The type byte has only one single constructor, and a natural value i which is constrained to be within the bounds of a byte. Externtype shows an example of multiple constructors. The list type gives an example of a dependent type argument X, which is utilized to represent the generic type of the elements of a list.

```
syntax byte =
    | `%`{i : nat}(i : nat)
        -- if ((i >= 0) /\ (i <= 255))
syntax externtype =
    | FUNC{functype : functype}(functype : functype)
    | GLOBAL{globaltype : globaltype}(globaltype : globaltype)
    | TABLE{tabletype : tabletype}(tabletype : tabletype)
    | MEM{memtype : memtype}(memtype : memtype)
syntax list{syntax X}(syntax X) =
    | `%`{X* : X*}(X*{X : X} : X*)
        -- if (|X*{X : X}| < (2 ^ 32))
```

Figure 3.5: Example of inductive types as represented in the IL

Type aliases are defined as simply a type it is an alias of. Figure 3.6 presents how result types would be represented in Webassembly 1.0, noting the restriction of the result always being an optional value instead of a list, a form utilized in the next versions of Webassembly.

```
syntax resulttype = valtype?
```

Figure 3.6: Simple example of a Type Alias

Record types are defined as a list of fields, with each field having a single atom as its name, variable bindings, a type, a list of premises and a list of hints. These records are interpreted similarly to other programming languages: as tuples of fields that can be accessed through dot notation. For example, figure 3.7 depicts how store would be represented for Webassembly 1.0.

```
syntax store =
{
  FUNCS{funcinst* : funcinst*} funcinst*,
  GLOBALS{globalinst* : globalinst*} globalinst*,
  TABLES{tableinst* : tableinst*} tableinst*,
  MEMS{meminst* : meminst*} meminst*
}
```

Figure 3.7: IL representation of the run-time representation of store record type

```
syntax unop_(valtype : valtype)
    syntax unop_{inn : inn}(INN_valtype(inn)) =
        | CLZ | CTZ | POPCNT
    syntax unop_{fnn : fnn}(FNN_valtype(fnn)) =
        | ABS | NEG | SQRT | CEIL | FLOOR | TRUNC | NEAREST
```

Figure 3.8: Example of family type IL representation

3.2.6 Family types

One last special case of user-defined types is the case of family types. Any of the previously mentioned user-defined types can come in groups, utilizing dependent types to differentiate between them and create a more general type.

Family types are formally defined as a tuple of: id representing its name, parameters representing the terms in which the base type depends on, and a list of user-defined types. The list of user-defined types would then have as their parameter the terms the base type depends on, but as a pattern to match to.

One notable example is the unop type in Webassembly, representing its unary operations. Since unop can be broken down into integer and float unary operations. Figure 3.8 shows this exact type expressed in the IL. It is clear that it is a family of inductive types, having valtype as the type to differentiate them. This gives the IL (and Wasm Spectec toolchain) even more expressive power as such a structure is convenient when handling a general type such as unop.

3.2.7 Expressions

Basic operations similar to what you would find in a programming language are provided as usual: unary operations such as not ($\sim$) and negation (-), binary operations such as basic arithmetic operations, logical operations, including equivalence ($<=>$) and implication ($=>$), and numerical comparison operations. Each numerical operation has their corresponding numerical type in order to differentiate between, for example, integer and real computation.

Primitive expressions are also supported such as variables, boolean, naturals, and strings. One thing to note is that there are only natural numbers as expressions supported at the moment, even if there are other numerical types.

Collection expressions are also supported:

- Tuple literals of the form $(exp_1\ exp_2\ ...\ exp_n)$
- List concatenation of the form $exp_1 :: exp_2$. This expression is overloaded with list cons, used extensively for matching.
- List literals of the form $[exp_1\ exp_2\ ...\ exp_n]$ with [] as empty.
- Option literal of the form $(exp)?$ with $()?$ as empty.
- List lookup of the form $exp[exp_{idx}]$ or $exp.i$ where i is the index.
- List slicing of the form $exp[exp_i : exp_j]$ where exp_j is not included.
- List length of the form $|exp|$
- List and option map of the form $(exp)^*\{id : typ\}$ and $(exp)?\{id : typ\}$ where id is the variable representing the entry of list id^* or option $id?$.
- List and option zip of the form $(exp)^*\{id_1 : typ_1, id_2 : typ_2\}$ and $(exp)?\{id_1 : typ_1, id_2 : typ_2\}$ where each id represents their entry of their corresponding list/option.
- Option conversion to normal type of the form $(exp)!$. This asserts that the optional type now must be required.
- Record literals of the form $\{atom_1\ exp_1\ ,\ atom_2\ exp_2,\ ...\ , atom_n\ exp_n\}$ where $atom_i$ is the field name.
- Record access of the from $exp.atom$ where exp must evaluate to a variable representing a record.
- Record composition of the from exp_{r1} @ exp_{r2} where exp_{r1} and exp_{r2} must be records. This expression handles the concatenation, if possible, of records. This is mainly used when all of the fields of a records are collections, such as the typing context.
- Record/list update of the form $exp_r[path = exp_{val}]$ where exp_r must be a record or list, $path$ resembles a list of operations involving record accesses, list lookup and slice lookup. Paths are their own inductive structure and the start of the path indicates what exactly gets updated in the top-level, while the end of the path indicates specifically what part of the structure gets updated. For example, figure 3.9 shows two different updates. The first one updates the runtime representation of the frame, specifically the local value at index x. Since the path ended with a list lookup, it ends up being a list update at the

end of the record path. The second one is more complex, having the runtime representation of the store be updated, specifically its lists of global instances is getting, at index idx, its record field VALUE_globalinst updated. Here idx was simplified for clarity.

- Record/list extend of the form $exp_r[path = .. \ exp_{val}]$. Similar to Record/list update, but instead of updating it extends the collection at the end of the path.

```
f[LOCALS_frame[x] = v]
s[GLOBALS_store[idx].VALUE_globalinst = v]
```

Figure 3.9: Record/list update example

It is good to note that the map and zip operations are the simple cases of that specific expression. In the general case, any number of lists/options are allowed, though they are largely unused due to their sheer complexity.

There are certain special expressions, namely the call expression, case expression and the sub expression. The call expression is of the usual form $id(arg_1 \ arg_2 \ ... \ arg_n)$, where id must be a definition.

The case expression is defined in the form $mixop \ exp$ where the $mixop$ represents the name of a constructor of an inductive type, while exp is the arguments that the constructor takes.

The subtyping expression is defined of the form $(exp \ : \ typ_1 \ <: \ typ_2)$, where typ_1 is the type the expression exp has, while typ_2 is the type the expression should convert into. The subyping expression is produced by the IL in order to explicitly state certain subtyping castings that occur while writing the DSL source. To ensure that these types can be casted, they must have the same atom identifier, and same typing. One notable example in the Webassembly 1.0 spec is the subtyping between instructions and Administrative instructions. Figure 3.10 shows this specific example, where it is clearly shown that instr is a subtype of admininstr, as they have the same constructors (i.e. CALL, CONST, CALL_INDIRECT, RETURN, etc.), but admininstr has more cases. Variable bindings are omitted in this figure for clarity.

3.2.8 Definitions and Relations

Along with types, users can define auxiliary definitions to simplify the specification, and inductive relations that represent the shape a certain judgement can form, such as reduction rules [6].

Definitions are defined in the IL as a tuple of: an identifier representing its name, a list of parameters (which could be either a variable of a certain type, or a generic

```
syntax instr =
  ...
  | CALL(funcidx : funcidx)
  | CALL_INDIRECT(typeidx : typeidx)
  | RETURN
  | CONST(valtype : valtype, val_ : val_(valtype))

syntax admininstr =
    ...
  | CALL(funcidx : funcidx)
  | CALL_INDIRECT(typeidx : typeidx)
  | RETURN
  | CONST(valtype : valtype, val_ : val_(valtype))
  | CALL_ADDR(funcaddr : funcaddr)
  | `LABEL_%{%}%`(n : n, instr* : instr*, admininstr* : admininstr*)
  | `FRAME_%{%}%`(n : n, frame : frame, admininstr* : admininstr*)
  | TRAP
```

Figure 3.10: Example of subtyping

type), its return type, and a list of clauses. The name, list of parameters and return type are known as the type declaration and the list of clauses are the actual definition. Each clause consists of variable bindings, a list of arguments representing the patterns their respective types match to, the resulting expression, and a list of premises. If the list of premises does not hold, then the pattern matching fails. Definitions are allowed to be self-recursive. To give a simple example, figure 3.11 shows how the function min would be represented in the IL. It can be clearly seen that the definition is recursive, however the IL makes this even more explicit and unambiguous by putting the rec keyword.

```
rec {
def $min(nat : nat, nat : nat) : nat
  def $min{j : nat}(0, j) = 0
  def $min{i : nat}(i, 0) = 0
  def $min{i : nat, j : nat}((i + 1), (j + 1)) = $min(i, j)
}
```

Figure 3.11: Simple Definition example

Inductive relations are defined in the IL as a tuple of: an identifier representing its name, a mixop representing the structure of such relation, a tuple type of arguments that the inductive relation can take, and a list of rules. The name, mixop and tuple type are known as the relation declaration, and the list of rules are the actual inductive rules encompassing the declaration. Each rule consists of an id, variable

bindings used in the premises and final expression, a list of premises, and the final expression. Premises are covered in section 3.2.9. The final expression represents the form the relation should take if all of the premises are fulfilled. Inductive relations are allowed to be self and mutually recursive with other inductive relations.

To give a simple example, figure 3.12 shows the inductive relation Instr_ok which represents the typing rules for single instructions. Rule nop has only the binding context, and its final expression is only the instruction NOP_instr and the empty function type. This means that when reasoning about Instr_ok, if the conclusion of a certain argument (say, in a proof), has the form given in the final expression, then one can reason that it is valid. Here the mixop (‘%|−%:%‘) represents the standard structure of typing judgements used in Webassembly, as discussed in section 2.2.2. Rule br shows a more complex example, where even if the conclusion is in the right form to apply the rule, the premises also need to be true.

```
relation Instr_ok: `%|-%:%`(context, instr, functype)
rule nop{C : context}:
    `%|-%:%`(C, NOP_instr, `%->%`_functype([], []))

rule unreachable{C : context, t_1* : valtype*, t_2* : valtype*}:
    `%|-%:%`(C, UNREACHABLE_instr,
    `%->%`_functype(t_1*{t_1 : valtype}, t_2*{t_2 : valtype}))

rule br{C : context, l : labelidx, t_1* : valtype*,
    t? : valtype?, t_2* : valtype*}:
    `%|-%:%`(C, BR_instr(l),
    `%->%`_functype(t_1*{t_1 : valtype} :: t?{t : valtype},
    t_2*{t_2 : valtype}))
    -- if (l < |C.LABELS_context|)
    -- if (C.LABELS_context[l] = t?{t : valtype})
```

Figure 3.12: Inductive relations example

3.2.9 Premises

The IL supports a wide range of premises (The let premise is omitted as it is unused and undocumented):

- Rule premise of the form (id : $mixop(exp)$): this premise constrains the given relation by stating that a specific rule of a different relation must also be true. The id represents the rule name, mixop the structure and the exp is the final expression noted in section 3.2.8.
- If premise of the form (if exp): this states that a specific expression exp must be true.

- Else premise of the form ($otherwise$): this states that any of the previous rules/-clauses premises must be false.
- Single iteration premise of the form ($(prem)^{iter}\ \{(id : typ)\}$): this states that for a given iteration of variable id, the premise must be true for each element of the iteration.
- Double iteration premise of the form ($(prem)^{iter}\ iteration\ \{(id_1 : typ_1),\ (id_2 : typ_2)\}$): this states that for a given iterations of two variables id_1 and id_2, the premise must be true for each zipped element of the two iterations. This means that the two iterations must be of the same size.

As it was the case for mapping and zipping, the iteration premise also allows for any number of iterations, but is unused for the most part except for the single and double case.

3.2.10 Transformations*

After the EL gets transformed into the IL, it is possible to run a few passes on the IL, transforming it and making more information available for the backends. The transformations used for the translation solution are: totalize, wild and sideconditions.
Totalize looks for the keyword $partial$ when going through all definitions and attempts to convert the definition into a total function by making the result type into the option iteration, and adding an additional case where the general pattern is given as the arguments, and the empty option literal as the resulting expression. Figure 3.3 presents a simple example of how it may transform the function $signif$ (representing the size of the significand of a floating point number). Totalize also changes any function call expression of this definition to assert that the value must be required through the option conversion expression ($exp!$).

Wild transforms the IL by attempting to remove wildcards from the syntax. It currently detects optional single constructor inductive types and empty option literals. This is motivated as many of the DSL source syntax has expressions be acutal relations which are allowed to fail and have multiple values, while theorem provers must denote exactly one value given another value and all free variables [6].

Sideconditions attempts to transform premises and final expressions in inductive relations to have explicit conditions. Some examples include: list access, which if the premise is true, then it must be the case that the index is smaller than the list; double iteration, which if the premise is true then it must be the case that the two iterations are of the same size.

Chapter 4

IL2Coq

4.1 IL2Coq Workflow

4.1.1 Structure

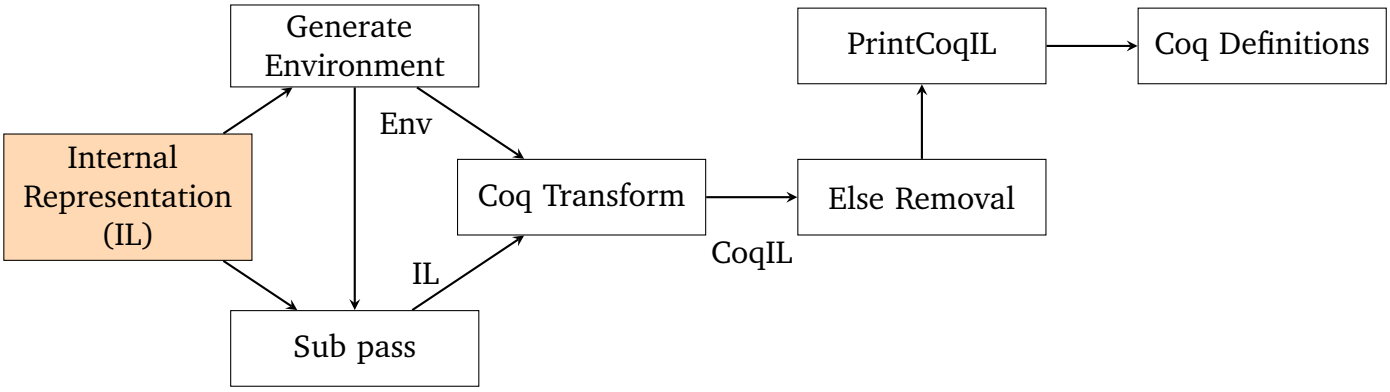

Figure 4.1: Design of Il2Coq

Figure 4.1 depicts the structure for the translation process between IL and Coq (IL2Coq). It is, as the Wasm spectec toolchain, written in Ocaml. This is a particularily good advantage as Ocaml syntax is similar to Coq syntax, making it easier to translate.

Coq Transform is the main core of the IL2Coq workflow, as it attempts to convert all of the IL to an AST that resembles the inner syntax of Coq more, named CoqIL. It attempts to disambiguate even more the information gathered from the IL, and make a more restrictive version, in order to comply with the Coq compilation process. This is discussed more in detail in section 4.2.

IL2Coq has three auxiliary passes. The first one is to generating an environment by going through the IL and gathering information on certain typings. It is then utilized in the transformation process to turn the IL to CoqIL. The second pass is known as the sub pass, which creates explicit subtyping conversion functions and sets up implicit coercion in Coq. The final pass is done after the CoqIL is created, known as

”else removal”, which is done in order to remove the ”otherwise” premise that Coq cannot handle. These three passes with be discussed more in detail in section 4.3.

After all of the passes and transformations are done, the last thing left to do is to print the CoqIL by traversing the AST. This is discussed more in detail in conjunction with IL2Coq as their design are tightly coupled.

All of these workflows can be found in the github repository linked at the start, in the folder backend_coq.

4.1.2 Restrictions Imposed in the DSL Source

The DSL source, at the current moment, is very expressive and gives the user a large amount of freedom to write their definitions and rules. In order to simplify the translation process, some restrictions are applied to the writing of the DSL source.

The first restriction is in regards to family types. Any parameter used in the base type of a family type must have explicit structure. This means that if there is any subtyping between types in the list of user-defined types, it is made more explicit through inductive structures. One notable example can be seen in figure 3.8, which makes sure that the user-defined types inside the family type utilizes valtypes structure to differentiate them, instead of having inn be a subtype of valtype (which is the original version of the spec). This was done as Coq does not support subtyping (except for explicit conversions made beforehand, which is what sub pass does), and there is no possible way of going from the general type to the subtype besides matching. Since it is not possible to match in an inductive structure, then there is no easy way of allowing sub expressions to occur in this case.

The second restriction is to definitions and inductive types with premises. Any definition that has clauses with premises will be removed and treated as an axiom. In the case for the type preservation proof, some of these definitions were needed, and as such they were rewritten as relations in the DSL source. This is due to the fact that Coq does not handle conditional definitions well, and the DSL source expects for some of the variables to be quantified universally, which is simply not possible for executable definitions in Coq. Inductive types are kept intact but without the premises. These premises make types much more complicated to handle in proofs. In addition, when used in case expressions, the premise would then need to be shown to be true, making the expression much more messy.

The third restriction is to iterations. Option and general list are kept as usual, but the other types of lists have been degenerated to the general list type. This is simply due to the fact that it is much easier to just use a single type of list, as the others are not implemented in Coq.

The final restriction has to do with user-defined number types. Since most of the

number types are dependent types with premises, must of the information is lost through the translation process. Hence it is much simpler to ignore them for now, as they aren't useful in the type safety proofs anyways, and they need to be handled with extra care to get the right meaning across.

4.2 Coq Transform and Printing

This section will talk about the design process of the core algorithm of IL2Coq: the transformation process from IL to CoqIL. It will go through firstly the motivation for having the CoqIL, a brief overview of the CoqIL, the transformation approach, and the transformation and printing of types and expressions.

4.2.1 Motivation for CoqIL

The CoqIL is a different AST generated from the IL that acts as a clean version of the DSL source that can be immediately represented in Coq. In earlier versions of IL2Coq, we did a direct translation (with passes) from the IL to Coq (following as inspiration earlier work done in [6]), but this lead to the code being much more complex and harder to work with. With the Coq Transform being only the transformation process, we were able to separate the complexity between different passes more effectively.

In addition, there are certain blocks of the IL AST that do not have enough information for Coq to handle correctly. For example, the else removal pass requires the negation premise that the IL simply does not have. Since the CoqIL is isolated from the other parts of the toolchain, it allows easier modification to the AST that would make the structure suit Coq better. It is also possible to add even more passes that are only needed for Coq.

4.2.2 Translation Approach

The guiding principle to constructing this translation between the IL and Coq was to mainly focus on targeting a specific version of WebAssembly. This would mean that the overall solution would be more of a proof of concept rather than a general solution that works on all inputs. That being said, while constructing the solution, most of the components have been made to be transformed in a general sense, but some parts are left out that could easily be extended if necessary. A notable example is inductive relations, which have a good clean translation between IL to Coq for all inputs, not only WebAssembly 1.0. The only exception to this is family types and dependent types in general, which require some modification of the IL to make them work better. This will be discussed more in detail in the evaluation.

4.2.3 Exported Code

In order to fully be able to translate most of the features given in the IL specification, we need to be able to extend our Coq environment.

The Coq environment first has to establish some type classes. Type classes are types that Coq can infer automatically through instances defined by the user [13]. With type classes, we can define a certain group of types that exhibit a specific behaviour. In this case, we want our types to have a default value. As such, the inhabited type class is created as shown in figure 4.4. This effectively makes it so any type T defined through Instance keyword can use the functions lookup_total (used for list lookup expression) and the (used for option conversion expression). With this setup, now any inductive type and record type created only needs to define their own instance.

To give a simple example of why the inhabited type class is necessary, figure 4.2 shows the typing rule for the call instruction. Its premise makes sure that the function type at address x is the same type as the call instruction. List Lookups in WebAssembly implicitly state that x is bound [3]. In the IL representation, due to the side conditions pass, this is added explicitly to help Coq in its proofs. However, Coq needs more information to deal with list lookups. Namely, it either must return an option (List.nth_error [13]), or the resulting type of the lookup must have a default value (as seen in figure 4.4 with lookup_total). The default value method is generally more favorable as the option case would make every lookup access have to be transformed to an option and it would have to be propagated further to the entire AST, making it a non-trivial problem. As such, the inhabited type class provides a simple solution that does not require modification of the expression.

```
rule Instr_ok/call:
  C |- CALL x : t_1* -> t_2?
  -- if C.FUNCS[x] = t_1* -> t_2?
```

Figure 4.2: Call Typing Rule in the DSL Source

The other type class needed is the Append type class which attempts to facilitate record composition. Since records can be made of multiple different types, in order to be able to append two records of the same type, we would need each field to also be appendable. The type class Append exists for this reason, so it is possible to easily append field types. We defined some instances beforehand, such as list and option, as these are used extensively in records. For fields that are of type user-generated inductive types and primitive types (such as booleans), we consider them as non-appendable and for the moment only take the first value and ignore the second one.

Functions that resemble the collection expressions behaviour are also made, such as list slice, option map, etc. We also made an option to list conversion function, to mimic the implicit subtyping of iterations that is utilized in the IL. To make this much simpler rather than having to add the function to all the cases where a list is

needed rather than an option, we utilize a feature known as coercion. The coercion keyword in Coq indicates an inheritance mechanism between types that doesn't really increase the expressive power of the language, but it instead offers convenience [13]. Figure 4.3 shows how it would be constructed, with the function_name being the conversion function, type_1 is the actual type, and type_2 is the type it is converting into. Whenever Coq encounters the second type required but only the first type is given, then the conversion function is applied automatically. This is really useful and the basis of how the sub pass is constructed.

```
Coercion function_name: type_1 >-> type_2
```

Figure 4.3: Type Coercion in Coq

```
Class Inhabited (T: Type) := { default_val : T }.
Class Append (A: Type) := _append : A -> A -> A.
Definition lookup_total {T: Type} {_: Inhabited T} (l: list T) (n: nat) : T :=
  List.nth n l default_val.
Definition the {T : Type} {_ : Inhabited T} (arg : option T) : T :=
  match arg with
    | None => default_val
    | Some v => v
  end.
Definition option_append {A: Type} (x y: option A) : option A :=
  match x with
    | Some _ => x
    | None => y
  end.
Global Instance Inh_nat : Inhabited nat := { default_val := 0 }.
Global Instance Inh_list {T: Type} : Inhabited (list T) :=
    { default_val := nil }.
Global Instance Append_List_ {A: Type}: Append (list A) :=
    { _append l1 l2 := List.app l1 l2 }.
Global Instance Append_Option {A: Type}: Append (option A) :=
    { _append o1 o2 := option_append o1 o2 }.
```

Figure 4.4: Type classes used and its corresponding functions

4.2.4 User-defined Types, Relations and Definitions

Inductive Types

Coq Inductive types are a reduced version of the IL version, defined in the form: name, list of arguments (if it is a dependent type), and a list of constructors.

Constructor names for Coq have to be unique in a Coq module, and as such the name of the base type is prepended to all constructors. Inductive types are also given a defined instance of the Inhabited type class, with the default val being the first constructor (with all of the argument types for a constructor taking their default value as well).

Record Types

Coq record types behave similar to the IL record type. In the CoqIL they are defined as an identifier representing its name and a list of record fields, each containing an identifier representing field name and a coq term as its type (it also has struct type but that is for the auxiliary pass).

Some restrictions and extensions are made when constructing a record type. Firstly, a restriction for Coq record types is that field names for a record type must be unique in the Coq module. This means that if a field name is reused in a different record, this would not be valid. This is due to the fact that projections (functions that allow access to a specific field) are made for each field and having the same name would clash the function name [13]. In order to overcome this, field names are extended to also include the record name, making it effectively unique. This can be seen in figure 4.5, with the fields being prepended with the name ”store”.

Secondly, records have to be extended to allow record composition and default records. With the aid of the type class Append created beforehand, we are able to define Append_store which goes inside the record and recursively calls the append_ function to effectively go inside the structure. A default record is also made through the Inhabited class, allowing lookup and option conversion capabilities.

Finally, records have to be extended to allow for automatically updating record fields. With the help of the external library for automating record updates (repository in [26]), we can facilitate the process in a generic manner. To set it up, we need to define the instance for the type class Settable, which takes the constructor made at the start (in the case of figure 4.5, it would be mkstore, the constructor that creates a record), and all of the field names, to set up and automatically infer what needs to be modified in the record. The notation of how to actually define the specific updates will be described in section 4.2.5.

Family Types

With the restrictions applied on family types, they can now simply be defined in the CoqIL as a tuple of a name and a list of either a coq_term for type alias group or a list of constructors for an inductive types group. In order to make this work for inductive types, we take the type that the base type depends on, and use its name to create a new inductive type. In this case, we take the constructor valtype__INN(inn)from figure 3.8 and utilize its inner type inn to form unop___inn. This only works because valtype was constructed in the specific way discussed in section 4.1.2. We then use

```
Record store := mkstore
{  store__FUNCS : (list funcinst) ; store__GLOBALS : (list globalinst)
;  store__TABLES : (list tableinst); store__MEMS : (list meminst)
}.
Global Instance Inhabited_store : Inhabited store :=
{default_val := {| store__FUNCS := default_val; store__GLOBALS := default_val;
  store__TABLES := default_val; store__MEMS := default_val|} }.
Definition _append_store (arg1 arg2 : store) :=
{|
  store__FUNCS := append_ arg1.(store__FUNCS) arg2.(store__FUNCS);
  store__GLOBALS := append_ arg1.(store__GLOBALS) arg2.(store__GLOBALS);
  store__TABLES := append_ arg1.(store__TABLES) arg2.(store__TABLES);
  store__MEMS := append_ arg1.(store__MEMS) arg2.(store__MEMS);
|}.
Global Instance Append_store : Append store :=
 { _append arg1 arg2 := _append_store arg1 arg2 }.
#[export] Instance eta__store : Settable _ :=
  settable! mkstore <store__FUNCS;store__GLOBALS;store__TABLES;store__MEMS>.
```

Figure 4.5: Full translation of a Record Type from input figure 3.7

these new inductive types created to make the base family type as an inductive type, having each constructor take each of the new inductive type and appending the word ”entry” to make the constructor unique. We also do not include the type it depends on, as this would make it much harder to deal with later if we keep it. This is seen clearly in the figure 4.6, having unop_ have two entries, each having an arg of type unop___inn and unop___fnn respectively.

For a type alias family, it would instead just make a constructor for each entry of the family, having the constructor take the type of the type alias. We also erase the dependent type in this family. For type aliases, we make sure that the type of any entry of the group respects subtyping coercions by making the constructor be the conversion function (this could also be done for inductive types, but is unnecessary for WebAssembly 1.0).

Figure 4.6 shows the inductive type val_, which is a family type alias, and iN and fN are what the resulting type the DSL wants val_ to convert into. val___inn__entry would be the conversion function between iN to val_.

Since we erase the dependent type, for any match statement in a definition, we need to infer the dependent type through the other match arguments. The IL makes sure that there needs to be the dependent type in the matching in order for this to work, so all we need to do is find the type it refers to. Figure 4.7 shows this specific example, where in order to add unop___inn__entry, we need to find the first arguments type and get that it matches to inn.

```
Inductive unop___inn  : Type :=
  | unop___inn__CLZ  : unop___inn | unop___inn__CTZ  : unop___inn
  | unop___inn__POPCNT  : unop___inn .
Inductive unop___fnn  : Type :=
  | unop___fnn__ABS  : unop___fnn  | unop___fnn__NEG  : unop___fnn
  | unop___fnn__SQRT  : unop___fnn  | unop___fnn__CEIL  : unop___fnn
  | unop___fnn__FLOOR  : unop___fnn | unop___fnn__TRUNC  : unop___fnn
  | unop___fnn__NEAREST  : unop___fnn .
Inductive unop_  : Type :=
  | unop___inn__entry (arg : unop___inn) : unop_
  | unop___fnn__entry (arg : unop___fnn) : unop_ .
Inductive val_: Type :=
  | val___inn__entry : iN -> val_
  | val___fnn__entry : fN -> val_.
Coercion val___inn__entry : iN >-> val_.
```

Figure 4.6: Inductive Family Type unop_ as Figure 3.8, and Family Type Alias val_

In addition, when a specific dependent type argument is already a concrete case (such as inn__i32), we rewrite it to be the sub-type. As an example, we would convert the IL representation val_(inn__I32) into simply iN. This is simply to mimic the sub-typing structure that the IL creates for family types.

```
Definition fun_unop (v_valtype_0 : valtype) (v_unop__1 : unop_)
    (v_val__2 : val_) : list__val_ :=
  match (v_valtype_0, v_unop__1, v_val__2) with
   | ((valtype__INN v_inn), (unop___inn__entry (unop___inn__CLZ )),
     (val___inn__entry v_iN)) =>
     [((fun_iclz (fun_size (valtype__INN v_inn)) v_iN) : val_)]
```

Figure 4.7: Inferring the Constructor for Family Types

Inductive Relations

Inductive relations in the CoqIL are the only high-level structure that mirror well what is depicted in the IL. It is defined in the form of a name, inductive relation arguments, and a list of rules. These rules are an extension of the inductive constructors, having also a list of premises and the end expression. Each of the arguments for the rules behave as universally quantified variables. Just like inductive types, the base name is prepended to all of the rule names to make them unique.

To give a small example of how inductive relations are translated, figure 4.8 shows how figure 3.12 is translated to Coq. Instr_ok is a relation that takes a context, in-

struction and functype and returns a proposition, allowing the use of such structure for proof handling.

```
Inductive Instr_ok: context -> instr -> functype -> Prop :=
 | Instr_ok__nop : forall (v_C : context),
   Instr_ok v_C (instr__NOP ) (functype__ [] [])
 | Instr_ok__unreachable : forall (v_C : context)
    (v_t_1 : (list valtype))
    (v_t_2 : (list valtype)),
    Instr_ok v_C (instr__UNREACHABLE ) (functype__ v_t_1 v_t_2)
 | Instr_ok__br : forall (v_C : context) (v_l : labelidx)
  (v_t_1 : (list valtype)) (v_t : (option valtype))
  (v_t_2 : (list valtype)), (v_l < (List.length (context__LABELS v_C)))
  /\ ((lookup_total (context__LABELS v_C) v_l) = v_t)
  -> Instr_ok v_C (instr__BR v_l) (functype__ (@app _ v_t_1 v_t) v_t_2)
```

Figure 4.8: Translated Inductive Relation Example

Definitions

Definitions in the CoqIL are a restrictive version of the IL, defined in the form of a tuple of name, list of arguments, return type and list of match clauses. There are no indication of premises in the CoqIL, as such it has lost information of the function definition itself. For match expressions, there are specific cases that make sure that matching is well defined. For example, list concatenation is overloaded with list cons, and as such when encountered in matching, we transform it to list cons in the CoqIL.

If the list of match clauses are empty, that means that there is only a definition declaration, and as such, the function definition is transformed into an axiom. Figure 4.9 shows an example of an axiom that appears in the WebAssembly 1.0 spec (due to numeric operations not being implemented yet). Axioms in Coq are just treated to hold for any input and output and as such is rather useful format for something that is not implemented yet [13].

```
Axiom fun_fabs : forall (v_reserved__N_0 : reserved__N) (v_fN_1 : fN), fN.
```

Figure 4.9: Axiom example

4.2.5 Transformation and Printing of Basic Types and Expressions

Most of the basic types and expressions shown in section 3.2.4 and 3.2.7 are translated in the straightforward way, as Coq already has builtin support for them.

One of the few exceptions is when, in a definition, a variable of type nat gets matched with $n + 1$. Coq, when matching, only considers constructors of inductive types as valid, and the plus operator is considered a notation for the function "add." As such, whenever a plus operation is encountered in a match statement, we replace it with the successor term in the CoqIL ("S") with as many as the value of the second term (in this case, one). Figure 4.10 shows how the min function given in figure 3.11 is translated to Coq. The successor term resembles the successor constructor for the inductive structure of natural numbers in Coq, and as such it succeeds in the match.

```
Fixpoint fun_min (v_reserved__nat_0 : nat) (v_reserved__nat_1 : nat) : nat :=
  match (v_reserved__nat_0, v_reserved__nat_1) with
    | (0, v_j) => 0
    | (v_i, 0) => 0
    | ((S v_i), (S v_j)) => (fun_min v_i v_j)
  end.
```

Figure 4.10: Translated min example, from input of figure 3.11

One of the notable modifications made were to list/record updates. The CoqIL attempts to keep the same structure of a path, however it makes it more compact by merging dot accesses together. In order to represent record updates and list updates, we make use of an external library named coq-record-update (repository in [26]), to facilitate updates. In order to represent a specific record access, figure 4.11 shows the notation made for updating records and an example seen in translated WebAssembly 1.0 spec (the same IL examples seen in figure **??**). In the example, we can see v_f, which is of record type frame, is getting its field frame__LOCALS updated through list update. In order to apply dot accesses together, we can simply put a semicolon. In the notation this simply means it is going to pipeline the accesses in order to get the right field to update.

The record update gets more complicated with many interleaving list accesses and record accesses. For every interleaving, we need to have create another record update notation. For list lookup, we need to utilize the predefined function list_update_func which instead of updating the list to a specific value, we apply a function to the value found in the list. This allows us to stack as many interleaving record and list accesses as we need. The second example in figure 4.11 shows exactly this.

```
(record name) <| (record field) := (value) |>
(v_f <|frame__LOCALS := (list_update (frame__LOCALS v_f) (v_x) (v_v))|>)
(v_s <|store__GLOBALS := (list_update_func (store__GLOBALS v_s) v_idx
  ((fun v_1 => v_1 <|globalinst__VALUE := v_v|>)
```

Figure 4.11: Translated record updates as example 3.9

4.3 Auxiliary Passes

4.3.1 Generating the Environment

In order to fulfill some of the translation goals of the sub pass and the Coq transform, we need to generate some mappings that give extra information of the types defined by the user.

We first need to know whether a specific type is either a type alias, inductive type, or record. This is because record types need to know whether a certain type is appendable, in order to define the instance. If that certain type is an inductive type, then it is considered not appendable.

Then, we need to also know the name of the variable type that the type alias structure aliases to, and the number of arguments. This is needed for case expressions, since there are cases where we only have the type alias name, but we actually need the inductive type name in order to prepend to the case. The number of arguments is needed to put holes, if the type alias definition is a dependent type. Holes are used so that Coq is able to automatically infer their value (if possible).

The final information needed is a mapping between the name of an inductive type and its constructors. This is used extensively in the sub pass in order to be able to match correctly between sub-type and super-type.

The structure of the environment is two mappings. The first one is a map between an identifier of any user-defined type and a tuple of the next identifier, number of arguments and structure type. The second one is a mapping of inductive type identifier to a list of constructors. Figure 4.12 shows the definition of the environment and structure type.

```
type struct_type = | Record | Inductive | TypeAlias | Terminal
module Env : Map.S
type var_typ = id * int * struct_type
type sub_typ = (id * mixop * typ) list
type env = { mutable vars : var_typ Env.t; mutable subs : sub_typ Env.t }
```

Figure 4.12: Definition of Environment in Ocaml

4.3.2 Creating Subtype Coercions

The sub pass is an intermediate stage between the IL and CoqIL in which attempts to look at any inductive relation (could be extended to other high-level structures) and find all sub-typing expressions. Then, we go through each sub-typing expression and go through the following procedure:

1. Find the constructors for the sub-type and the super-type. For this, we utilize the environment's map of id to list of constructors.
2. For each constructor in the sub-type, we go through all constructors of the super-type and check if they are the same. In order to be the same, the constructor name must match and each type inside the tuple type must be the same (this could be extended to also include sub-types instead of being strictly the same).
3. After the constructors are matched correctly, we create a CoqIL function definition, having each match clause be the constructors of the sub-type and super-type matched.
4. With the function definition, we create a coercion between the two types utilizing the function definition as the conversion function.

These are all created before the relation is created, and they are immediately made into CoqIL high-level structures. This is because coercions do not exist in the IL. Figure 4.13 shows this exact creation for the sub-type instruction and super-type administrative instruction.

With this in place, the sub expression then becomes simple: in the CoqIL it is of the form $(exp\ :\ typ_2)$ where typ_2 is a super-type. This forces Coq to convert the expression exp into the super-type, which then it implicitly adds the conversion function.

```
Definition fun_coec_instr__admininstr (v_instr : instr) : admininstr :=
  match (v_instr) with
  | (instr__NOP ) => (admininstr__NOP )
  | (instr__UNREACHABLE ) => (admininstr__UNREACHABLE )
  | (instr__CALL v_0) => (admininstr__CALL v_0)
  | (instr__CALL_INDIRECT v_0) => (admininstr__CALL_INDIRECT v_0)
  | (instr__RETURN ) => (admininstr__RETURN )
  | (instr__CONST v_0 v_1) => (admininstr__CONST v_0 v_1)
  ...
  end.
Coercion fun_coec_instr__admininstr : instr >-> admininstr.
```

Figure 4.13: Conversion Function between Instructions and Admin Instructions as example 3.10

4.3.3 Else Removal Pass

The else removal pass is a transformation that takes in the CoqIL and returns an extended version of the CoqIL. It is adapted from earlier work done in the Wasm Spectec repository [6], but modified a good amount to fit the CoqIL instead of the IL. It attempts to find an otherwise premise in an inductive relation and transform it

as the negation of all the rules that are similar enough to the rule where the premise is. For now it only works for binary relation (such as the reduction rules).

The procedure is as follows:

1. Go through all inductive relations and filter the binary relations.
2. For a rule, if we encounter an otherwise, then we take all of the previous rules, and apply a filtering process: we unarize the binary rule (taking the LHS), then we check that the previous LHS is the same as the current LHS.
3. We then take all of the filtered previous rules and form a new relation that has the previous rules as its own rules. The name is adjusted to make sure it is considered unique.
4. With this new relation, we put it before the normal relation and then continue for all the other rules. For the specific rule, we negate the new relation in place of where the otherwise premise was.

Figure 4.14 shows an example of how the otherwise premise found on the step reduction rule of call indirect trap is translated to Coq.

```
Inductive Step_read_before_Step_read__call_indirect_trap: config -> Prop :=
 | Step_read__call_indirect_call_neg : forall (v_z : state)
 (v_i : nat) (v_x : idx) (v_a : addr),
 (v_i < (List.length (tableinst__REFS (fun_table v_z 0))))
 /\ (v_a < (List.length (fun_funcinst v_z)))
 /\ ((lookup_total (tableinst__REFS (fun_table v_z 0)) v_i) = (Some v_a))
 /\ ((fun_type v_z v_x) = (funcinst__TYPE (lookup_total (fun_funcinst v_z) v_a)))
 -> Step_read_before_Step_read__call_indirect_trap
    (config__ v_z [(admininstr__CONST (valtype__INN (inn__I32 )) (v_i : val_));
    (admininstr__CALL_INDIRECT v_x)]).
Inductive Step_read: config -> (list admininstr) -> Prop :=
 | Step_read__call_indirect_trap : forall (v_z : state) (v_i : nat) (v_x : idx),
 (~(Step_read_before_Step_read__call_indirect_trap
 (config__ v_z [(admininstr__CONST (valtype__INN (inn__I32 )) (v_i : val_));
 (admininstr__CALL_INDIRECT v_x)])))
 -> Step_read
 (config__ v_z [(admininstr__CONST (valtype__INN (inn__I32 )) (v_i : val_));
 (admininstr__CALL_INDIRECT v_x)]) [(admininstr__TRAP )]
```

Figure 4.14: Example of generated relation for the Otherwise Premise

Chapter 5

Type Preservation Proof

The following sections will present the result of the type preservation proof, giving a small overview of how the proof was constructed, and the DSL Extension made to have all of the definitions required for soundness. The proof and generated definitions can be found in the file proofscoqil.v.

5.1 DSL Source Extension

The current Wasm Spectec repository contains the specification for Webassembly 1.0, which includes the syntax, typing rules, instantiation, numerics and reduction rules. However, these typing rules are generally insufficient to prove soundness. As shown in WasmCert and in section 2.3.2, we need a notion of what it means for the runtime configuration, and its components, to be justified in a typing judgement. As such, I extended the DSL source to include these typing judgements specifically for type preservation.

All of these extensions were made to reflect exactly the same as the specification behaviour depicts. Most of the definitions can be found in the WebAssembly 1.0 spec, under the section appendix ”A.5 soundness” [3]. We will go through some of the new additions, and give their translated versions as example. These definitions can be found in the folder spec/wasm-1.0-test, with the file named A-soundness.watsup.

5.1.1 Store Validity

For a store to be considered valid, we need each instance found in the store to also be valid. This means that each element of the lists of function, global, table, and memory instances must be valid. Figure 5.1 shows how store validity would then be implemented in the DSL source.

For brevity, we will only go through function instance validity. For a function instance to be valid, we need each part of the instance to be valid. This means that the module instance must be valid with the type being a typing context, and with such a context

then the core body of the function must be valid with the same functype described in the function instance. Function validity is seen in figure 5.1 as the first example.

```
rule Function_instance_ok:
  S |- { TYPE functype, MODULE moduleinst, CODE func } : functype
  -- Functype_ok: |- functype : OK
  -- Module_instance_ok: S |- moduleinst : C
  -- Func_ok: C |- func : functype

rule Store_ok:
  |- S : OK
  -- if S = {FUNCS funcinst*, GLOBALS globalinst*, TABLES
  tableinst*, MEMS meminst*}
  -- (Function_instance_ok: S |- funcinst : functype)*
  -- (Global_instance_ok: S |- globalinst : globaltype)*
  -- (Table_instance_ok: S |- tableinst : tabletype)*
  -- (Memory_instance_ok: S |- meminst : memtype)*
```

Figure 5.1: Store and function instance validity

5.1.2 Thread Validity

For a thread to be valid, its frame and instruction sequence must be valid. The instruction sequence takes the return value that is given in the configuration to be preprended to the context, along with the store, to show the the instruction sequence has function type $eps -> t?$. This is the main core of the preservation proof, as we can go through each instruction typing rule to show preservation.

A frame is known to be valid when its local values are valid with type t, along with the module instance being valid with context C. Then, with these both true, the frame instance has type C preprended with the local types t^*. This is necessary for the proof as there are some cases when the frame is modified (such as local get) and having these typing judgements provide aid to prove such cases. Figure 5.2 shows how both of these validities are depicted in the DSL source.

5.1.3 Configuration Validity

In section 2.3.2, we went through the typing judgement of the highest level of the WebAssembly structure: the run-time configuration. For the configuration to be valid, its store, frame and instruction sequence need to be valid.

Figure 2.9 depicts how configuration validity is supposed to behave. Figure 5.3 shows how this would be written in the DSL source.

```
rule Frame_ok:
  S |- { LOCALS val*, MODULE moduleinst } : C, LOCALS t*
  -- Module_instance_ok: S |- moduleinst : C
  -- (Val_ok: |- val : t)*

rule Thread_ok:
  S; rt? |- F; admininstr* : t?
  -- Frame_ok: S |- F : C
  -- Admin_instrs_ok: S; $($(C, RETURN rt?))
    |- admininstr* : eps -> t?
```

Figure 5.2: Thread and Frame Validity

```
rule Config_ok:
  |- (S; F); admininstr* : t?
  -- Store_ok: |- S : OK
  -- Thread_ok: S; eps |- F; admininstr* : t?
```

Figure 5.3: Configuration Validity

5.1.4 Store Extension

Along with the validities, for preservation one must also prove that through a reduction of a run-time configuration, the new store is an extension of the old store. To achieve this, we need each component of the old store, say for example the function instances, to be an extension of a portion equal in length of the new store. Figure 5.4 shows this exact behaviour (other components are omitted for simplicity), having funcinst_1'*, a portion of store_2, be an extension of funcinst_1*, the function instance list of store_1.

```
rule Store_extension:
  |- store_1 : store_2
  -- if store_1.FUNCS = funcinst_1*
  -- if store_2.FUNCS = funcinst_1'* funcinst_2*
  -- (Func_extension: |- funcinst_1: funcinst_1')*
```

Figure 5.4: Store Extension represented in the DSL source

5.2 Proof Structure

In order to show preservation, we follow the main structure of the proof found in WasmCert [2]. We will show all the type preservation theorem and the corresponding top-level lemmas needed to prove it. The proof along with the automatically

translated definitions can be found in the file proofsCoqIL.v.

Figure 5.5 shows how the preservation theorem would be represented with the automatic definitions. It takes as an assumption the reduction between c1 and c2 (both being universally quantified configurations), and configuration validity of c1 with type ts. The conclusion would then be that c2 is valid with type ts, so c1 and c2 have exactly the same result type.

```
Theorem t_preservation: forall c1 ts c2,
  Step c1 c2 -> Config_ok c1 ts ->
  Config_ok c2 ts.
```

Figure 5.5: Main Theorem

To start this proof, we would apply backwards reasoning and attempt to show store validity. However, to show store validity, we need to also show store extension at the same time. Figure 5.6 shows how the lemma would be represented. We would use the fact that the module instance, store and instruction instance is valid through our assumption that the configuration c1 is valid.

```
Lemma store_extension_reduce: forall s f ais s' f' ais' C tf loc lab ret,
  Step (config__ (state__ s f) ais) (config__ (state__ s' f') ais') ->
  Module_instance_ok s (frame__MODULE f) C ->
  Admin_instrs_ok s (upd_label (upd_local_return C loc ret) lab) ais tf ->
  Store_ok s ->
  Store_extension s s' /\ Store_ok s'.
```

Figure 5.6: Lemma to show Store Extension and Store validity

Now that we have store extension and store validity, we can go deeper into the configuration validity and attempt to prove thread validity. For the thread to be valid, we need the frame to be valid. Figure 5.7 and 5.8 shows the lemmas needed to show frame validity. The first lemma tells us that if we are given store extension and module instance validity of the old store, then we can prove module instance validity of the new store. We have already both store extension and module instance validity (through configuration validity of c1), and we know that the module instance does not change as it remains static throughout all execution. The second lemma is to prove that the list of local values in the frame are all valid with the same typing as the old list of local values.

```
Lemma module_inst_typing_extension: forall v_S v_S' v_i v_C,
  Store_extension v_S v_S' -> Module_instance_ok v_S v_i v_C ->
  Module_instance_ok v_S' v_i v_C.
```

Figure 5.7: Lemma to Show Module Instance Validity

```
Lemma t_preservation_vs_type: forall s f ais s' f' ais'
  C C' v_t1 lab ret t1s t2s,
  Step (config__ (state__ s f) ais) (config__ (state__ s' f') ais') ->
  Store_ok s ->
  Store_ok s' ->
  Module_instance_ok s (frame__MODULE f) C ->
  Module_instance_ok s' (frame__MODULE f') C' ->
  v_t1 = (context__LOCALS (upd_label
  (upd_local_return C (v_t1 ++ (context__LOCALS C)) ret) lab)) ->
  Forall2 (fun v_t v_val => Val_ok v_val v_t) v_t1 (frame__LOCALS f) ->
  Admin_instrs_ok s (upd_label (upd_local_return C
    (v_t1 ++ (context__LOCALS C)) ret) lab) ais (functype__ t1s t2s) ->
  Forall2 (fun v_t v_val0 => Val_ok v_val0 v_t) v_t1 (frame__LOCALS f')
  /\ length v_t1 = length (frame__LOCALS f').
```

Figure 5.8: Lemma to show Local Values Validity

With everything else now proven, the only thing left is to prove that the instruction sequence preserves the same type as the old instruction sequence. To prove this, we must go through each reduction rule in this case and prove this fact. Figure 5.9 gives the top-level lemma to prove instruction sequence type preservation.

```
Lemma t_preservation_type: forall v_s v_f v_ais v_s' v_f'
  v_ais' v_C v_t1 t1s t2s lab ret,
  Step (config__ (state__ v_s v_f) v_ais)
    (config__ (state__ v_s' v_f') v_ais') ->
  Store_ok v_s -> Store_ok v_s' ->
  Store_extension v_s v_s' ->
  Module_instance_ok v_s (frame__MODULE v_f) v_C ->
  Module_instance_ok v_s' (frame__MODULE v_f) v_C ->
  Forall2 (fun v_t v_val => Val_ok v_val v_t) v_t1 (frame__LOCALS v_f) ->
  Admin_instrs_ok v_s (upd_label (upd_local_return v_C
    (v_t1 ++ context__LOCALS v_C) ret) lab) v_ais (functype__ t1s t2s) ->
  Admin_instrs_ok v_s' (upd_label (upd_local_return v_C
    (v_t1 ++ context__LOCALS v_C) ret) lab) v_ais' (functype__ t1s t2s).
```

Figure 5.9: Lemma to show Instruction validity

We will give a small example of a specific instruction preservation lemma and its DSL source version typing rules and reduction rules. Figure 5.10 gives the DSL typing and reduction rules for drop, to show how it is portrayed by the manually written lemmas for drop. Drop_typing attempts to extract the typing rule for drop, making sure that the input type must be the result type plus the type that is dropped. The preservation lemma shows that, given the instruction type validity of the instruction sequence before the reduction rule and the reduction itself, it maintains the instruction type

validity with the exact same type, showing preservation.

```
rule Instr_ok/drop:
  C |- DROP : t -> eps
rule Step_pure/drop:
  val DROP ~> eps
```

```
(* Lemma to extract the typing rule of drop *)
Lemma Drop_typing: forall v_S v_C t1s t2s,
    Admin_instr_ok v_S v_C (admininstr__DROP) (functype__ t1s t2s) ->
    exists t, t1s = t2s ++ [t].
(* Preservation Lemma *)
Lemma Step_pure__drop_preserves : forall v_S v_C (v_val : val) v_func_type,
  Admin_instrs_ok v_S v_C [v_val;(admininstr__DROP)] v_func_type ->
  Step_pure [v_val;(admininstr__DROP )] [] ->
  Admin_instrs_ok v_S v_C [] v_func_type.
```

Figure 5.10: DSL Typing and Reduction Rules, and Preservation Lemma for Drop

5.3 WasmCert Comparison

The main goal of the preservation proof is to show how similar the automated inductive definitions are compared to the manually designed inductive definitions of WasmCert. As noted in section 5.2, the main proof structure follows the main proof structure of WasmCert. Following the same proof structure allowed for parts of the WasmCert proof to be copied and adapted slightly to this form of the proof. We were even able to adapt some of the user-defined tactics used in WasmCert for this proof. To give a simple but notable example, figure 5.11 shows the comparison between both versions in abstracting certain type properties from the binop typing rule. Clearly, both of these lemmas resemble a good amount, and even the proof follows a similar direction, but it is omitted for brevity (if curious, the proof can be found in appendix B).

```
Lemma Binop_typing: forall v_S v_C v_t v_op t1s t2s,
  Admin_instr_ok v_S v_C (admininstr__BINOP v_t v_op) (functype__ t1s t2s) ->
  t1s = t2s ++ [v_t] /\ exists ts, t2s = ts ++ [v_t].

Lemma Binop_typing: forall C t op t1s t2s,
  be_typing C [::BI_binop t op] (Tf t1s t2s) ->
  t1s = t2s ++ [::t] /\ exists ts, t2s = ts ++ [::t].
```

Figure 5.11: Binop Typing in Translation Definitions Version (above) and WasmCert (below)

Similarities like these happen a large amount of the time, as the typing rules and reduction rules for the most part match the ones in WasmCert. However, there are some major differences that the proofs could not simply just be adapted and had to be altered in a significant way. This is mainly due to either: the way the DSL source is written doesn't match exactly the intentions of WasmCert, or the builtin relations/types used in the translation don't necessarily match the ones in WasmCert. For instance, some instructions such as branching and call address (known as invoke in WasmCert), and some higher-level structures such as store validity, store extension and local validity.

The branch instruction, in the current WebAssembly specification, utilizes the concept of block context to specify the recursive reduction of branches. WasmCert follows this faithfully by following this inductive structure, and performing induction on it to proof lemmas on branches. The existing DSL source takes a different approach which is to separate the two cases of the block context to different reduction rules. This actually makes the proof simpler as there is no unnecessary induction involved. Figure 5.12 shows this difference in lemmas (the translation version was modified to be clearer).

```
Lemma Step_pure__br_zero_preserves : forall v_S v_C v_n
  v_instr' v_val' v_val v_instr v_func_type,
  Admin_instrs_ok v_S v_C [(admininstr__LABEL_ v_n v_instr'
  (v_val' ++ v_val ++ [(admininstr__BR 0)] ++ v_instr))] v_func_type ->
  ((List.length v_val) = v_n) ->
  Admin_instrs_ok v_S v_C (v_val ++ v_instr') v_func_type.

Lemma Lfilled_break_typing: forall n m k
  (lh: lholed n) vs LI ts s C t2s tss,
  e_typing s (upd_label C (tss ++ [::ts] ++ tc_label C)) LI (Tf [::] t2s) ->
  const_list vs -> length ts = length vs -> lfill lh (vs ++ [::AI_basic (BI_br m)])
  e_typing s C vs (Tf [::] ts).
```

Figure 5.12: Branch Preservation in Translation Definitions Version (above) and WasmCert (below)

The translated version attempts to be as faithful as possible to the current specification. For store validity, this would mean that we have four different typing rules for each list of instances (function, tables, memories and globals). WasmCert takes a different approach and removes the check for global instances. This can be seen in figure 5.13, which shows the differences in the definition of store validity (the top inductive definition has some premises omitted for clarity. It is the translated version of figure 5.1). Mem_agree and table_agree behave similarly to Table_instance_ok and Memory_instance_ok, and cl_type_check_agree has Function_instance_ok along with extra information not yet in the DSL source (host functions). Thus, there is no reference to global instances. As such, when proving store validity, extra work had to be

done to prove this fact. In addition, the translated version utilizes Forall2 with the type being any type corresponding to the instance, which is vastly different to the WasmCert version, making it more complicated to prove.

Overall, the main preservation lemmas follow the same structure as WasmCert, and even the instruction preservation lemmas are very similar, both in lemma and proof. However, there are some exceptions that are mainly due to WasmCert taking a different approach from the current specification, which is reasonable as it still retains the same essence of WebAssembly while making it simpler. Nevertheless, the translated version still does a good job on providing similar definitions all things considered.

```
Inductive Store_ok: store -> Prop :=
  | Store_ok__OK : forall v_S v_funcinst v_globalinst v_tableinst
  v_meminst v_functype v_globaltype v_tabletype v_memtype,
  List.Forall2 (fun v_funcinst v_functype => (Function_instance_ok
  v_S v_funcinst v_functype)) (v_funcinst) (v_functype) /\
  List.Forall2 (fun v_globalinst v_globaltype => (Global_instance_ok
  v_S v_globalinst v_globaltype)) (v_globalinst) (v_globaltype) /\
  List.Forall2 (fun v_tableinst v_tabletype => (Table_instance_ok
  v_S v_tableinst v_tabletype)) (v_tableinst) (v_tabletype) /\
  List.Forall2 (fun v_meminst v_memtype => (Memory_instance_ok v_S
  v_meminst v_memtype)) (v_meminst) (v_memtype) -> Store_ok v_S.

Definition store_typing (s : store_record) : Prop :=
  match s with
  | Build_store_record fs tclss mss gs =>
  List.Forall (cl_type_check_single s) fs /\
  List.Forall (tab_agree s) tclss /\ List.Forall mem_agree mss
  end.
```

Figure 5.13: Comparions Between Translated Version of store validity (above) and WasmCert Version (below)

Chapter 6

Evaluation

The following sections will evaluate the three main contributions, presenting and at the same time answering a few research questions for each.

6.1 Evaluating IL2Coq: the Translation Process

6.1.1 Evaluating the Validity of the Translated Definitions

Chapter 4 gives an overview of the technicalities of the main solution of the project: the automatic translation of the DSL source to Coq. This presents us with a core research question: to what extent are the translated definitions a valid representation of the specification?

In section 4.1.2, we go through some restrictions that were imposed on the DSL source. While this made the translation simpler and able to be able to completed in time, this makes the translation process be an incomplete conversion. Having removed numerics and replaced them with a builtin number type removes some of the knowledge needed to construct an accurate representation of a reference interpreter. Additionally, inductive types and function definitions with premises lose information of their behaviour, ultimately being incorrect on the Coq side.

On the other hand, these inductive types with premises can be altered to only have the inductive type and have the premises be pushed to inductive relations where the types are used. As such, these can be alleviated by just restricting the expressiveness of the DSL source. This is a reasonable compromise as having targeting automated theorem provers already makes most of the language have to be restricted anyways. For numerics and function definitions with premises, they were removed simply to limit the scope of the project, as the main target is to prove that the typing rules and reduction rules of the WebAssembly specification behave well. Some function definitions can even be converted to inductive relations, allowing their use for the preservation proof. An example of this is the function grow_memory found in the file 5-runtime-aux.watsup.

The main evidence of the automated definitions being an accurate representation of the DSL source stems from the preservation proof being correct. Since the proof follows closely the work done from WasmCert (as described in 5.3), it is reasonable to conclude that at the very least the typing rules, datatypes used in the typing rules, and reduction rules strongly resemble the formal specification.

6.1.2 Evaluating the Potential for Expansion of the Translation Process and IL

As noted in section 4.2.2, the main focus of the translation was to specifically target WebAssembly 1.0. This is simply done to again limit the scope of the project and give a starting point for the creation of translation processes from IL to other interactive theorem provers. Thus, naturally the question comes to mind: to what extent can the current translation from IL to Coq be extended to handle arbitrary input (i.e. WebAssembly 2.0 spec or any proposal)?

Currently, some of the main high-level structures such as record types and inductive relations have a good representation in Coq and do not need to be modified. However, inductive types with arguments (known as dependent types) and family types need to be either fundamentally changed in the translation process or in the IL. This is due to the restrictions imposed in the DSL source, while being mainly good enough for WebAssembly 1.0, are not exactly good enough for arbitrary input. Also, most of the parts in the translation process were left to be extended to other cases when they become necessary for other specifications.

These problems mainly stem from dependent types just being too complex to handle in an automatic fashion, and even in proofs. After discussing this with active members of the WebAssembly community, it is possible to solve this problem through making a pass between the IL and CoqIL and remove the dependent types through a process called monomorphization. Monomorphization is defined to be the specialization of polymorphic functions (or dependent types) with respect to the type arguments that are actually used [27]. This will effectively remove the type argument and only supply instances of the type with concrete type arguments.

In addition, as future work it is also possible to use the CoqIL to extend the work to other interactive theorem provers. Since the CoqIL was made to be a separate abstract syntax tree decoupled from the IL, then it is indeed possible to make a general IL for interactive theorem provers, and have CoqIL connect to that one. This would make it possible to make passes that work for all interactive theorem provers.

To conclude, the CoqIL, while not fundamentally able yet to handle constrained arbitrary input such as other WebAssembly specifications, can be extended to other theorem provers and even has the potential to solve its problem through monomorphization.

6.2 Evaluating the DSL Source Extension

Having written the DSL source for the soundness typing judgements as noted in section 5.1, this main question is presented: how accurate is the DSL source extension compared to the formal WebAssembly specification?

One way to answer this question is to compare the latex generated from the latex backend with the formal specification, and check that they match. This is a reasonable approach and has been checked for all of the new additions to make sure it mostly has close resemblance to the formal specification. Appendix A has the generated latex of the soundness DSL source file, mainly put for curiosity.

The other conclusive evidence is the result of the preservation proof. Since the preservation proof was able to resemble WasmCert a large amount of the structure and be shown to be correct for the automated inductive definitions, it is mainly safe to say that the soundness extension closely resembles the formal specification.

There are some exceptions to this resemblance that have to do with the inconsistencies of the WebAssembly 1.0 formal specification. One notable example is the typing judgement for the Administrate instruction label validity, which can be seen in figure 6.1. This is automatically generated from the DSL source utilizing the latex backend. The main issue with this definition is that the result type $t_1^?$ is actually supposed to be t_1^n as in the WebAssembly specification [3]. The reason it is unable to be like this is because WebAssembly 1.0 restricts result types to be an option type, however in the formal specification it allows it to be of size n. This inconsistency is not really allowed in the wasm Spectec toolchain, which means that an extra premise was applied to the rule to have the size of result type $t_1^?$ be of same size as n.

$$\frac{C \vdash instr^* : t_1^? \to t_2^? \qquad S; C, \mathsf{labels}\ (t_1^?) \vdash instr^* : \epsilon \to t_2^? \qquad n = \mathrm{optionSize}(t_1^?)}{S; C \vdash \mathsf{label}_n\{instr^*\}\ instr^* : \epsilon \to t_2^?}\ [\textsc{Admin\_instr\_ok-label}]$$

Figure 6.1: Typing rule of Label Generated from Latex Backend

Another more serious example stems from how the external instances inside a module instance are valid in a typing judgement. For the formal specification version seen in figure 6.2, we can see that the minimum and maximum values of its table type must be the same as the one found in the store. However, if we have store extension and need to prove module instance validity, then it is not enough for the minimum value n to be exactly the same, as store extension only tells us that for tables, the minimum value n must not shrink. The generated version then instead makes it so that the external value only needs to be a limit sub of the one found in the store. This is the current definition found in WasmCert [23], and as such it has been kept that way in order for the type preservation proof to work.

$$\frac{S.\mathsf{tables}[a] = \{\mathsf{type}\ tt',\ \mathsf{refs}\ fa^{?*}\} \qquad \vdash tt' \leq tt}{S \vdash \mathsf{table}\ a : \mathsf{table}\ tt}\ [\textsc{Externvals\_ok-table}]$$

$$\frac{S.\mathsf{tables}[a] = \{\mathsf{max}\ m^{?},\ \mathsf{refs}\ fa^{?n}\}}{S \vdash \mathsf{table}\ a : \mathsf{table}\ \{min\ n, max\ m^{?}\}\ funcref}\ [\textsc{Externvals\_ok-table}]$$

Figure 6.2: Typing rule for External Table Values, generated example (above) and formal specification (below)

6.3 Evaluating the Preservation Proof

As noted in section 6.1.1, the results from the proof show that the translated definitions at the very least strongly resemble the formal specification. However, while constructing the proofs, some parts of not only the translated definitions, but the existing DSL source were shown to be inconsistent. Two examples of this seem to be typos, but due to these mistakes, the result has a completely different behaviour and thus preservation is not achieved. Both examples can be seen in figure 6.3. The first example states that the input should be of type $t_1^*\ t^?$, however, if we look at the reduction rule, the argument taken before the br_table instruction is a constant value of type I32. As such, the typing rule must be amended to have that that the end of the input (i.e $t_1^*\ t^?\ I32$) which is precisely what is seen in the formal specification [3]. The second example is more serious, as it makes instr_1 the single instruction instead of instr_2, which is again what how it is in the formal specification. In other words, the sequence rule in the DSL source goes recursively to the right instead of going to the left. This has some issues with the preservation proof and as such as been made to be of the other form. Also, the asterisk indicating it is a list is missing in the second premise, making it a singleton list in the IL.

```
rule Instr_ok/br_table:
  C |- BR_TABLE l* l' : t_1* t? -> t_2*
  -- if t? = C.LABELS[l']
  -- if (t? = C.LABELS[l])*
rule Instrs_ok/seq:
  C |- instr_1 instr_2* : t_1* -> t_3*
  -- Instr_ok: C |- instr_1 : t_1* -> t_2*
  -- Instrs_ok: C |- instr_2 : t_2* -> t_3*
```

Figure 6.3: Issues with DSL source

Through the result of the proof, these typos have been found and corrected. This clearly shows that the proof not only gave us the certainty that type preservation holds for the inductive definitions, but that the DSL source now has certainty that it adheres to type preservation just as the formal specification does.

Chapter 7

Conclusion and Future Work

In this project, we have presented IL2Coq, a proof of concept to the existing problem of automatically generating inductive definitions for mechanization. We firstly restricted the DSL source to simplify the translation process. Then, with various passes and transformations, we translate the intermediate language of the Wasm Spectec toolchain into a brand new AST, named CoqIL. With the additional flexibility of the new AST, we made some more passes to it and finally printed the result.

IL2Coq was mainly targeted to WebAssembly 1.0 specification, and has been successfully and completely compiled into Coq. With these definitions, in order to show their effectiveness, we extended the DSL source to include soundness inductive definitions, and then we presented a type preservation proof result with these automatically translated definitions.

From the exploration and evaluation of the entire project, we have effectively shown how much the IL2Coq solution covers the entire problem space and how much potential it has for future work. In addition, we have shown how effective the result of the proof and the DSL extension is, by showing how it revealed some inconsistencies in the DSL source and even inconsistencies in the formal specification itself.

In conclusion, IL2Coq serves as an effective starting point to the creation and realization of inductive definitions translated from the DSL source to interactive theorem provers. It can soon become part of the single source of truth that the Wasm Spectec toolchain is trying to achieve through the DSL source.

7.1 Future Work

As IL2Coq is mainly just a proof of concept and a starting ground for future interactive theorem provers translations, it has a lot of room for growth and improvement. Some of these are:

- Removing dependent types: as discussed in the evaluation section 6.1.2, there is a possibility to apply monomorphization to make the dependent type problem found in the IL2Coq solution much easier to solve.

- Making a more general interactive theorem prover IL: with the existence of the CoqIL, one could abstract portions of the AST and make a more general version that would suit all interactive theorem provers. CoqIL would connect to it so there would need to be a transformation from the ITP IL to the CoqIL. This would enable certain passes that would work for all ITPs instead of having the same pass applied to many different ILs.
- Finish extending all of the final cases: IL2Coq's translation approach was to target WebAssembly 1.0. As such, most portions of the transformation process were left not done, as they were unnecessary for the proof of concept. These cases should not be too hard to extend to, but they were out of the scope of the project.
- Improve family type handling: the current version of IL2Coq handles family types in a very restricted manner. In section 4.1.2, we stated all of these restrictions, but they should eventually be either fundamentally modified in the CoqIL, or the IL should change how the family types work in order for Coq to be able to handle it properly.
- Automated lemma generation: after we made the CoqIL, we utilized the information of the DSL source to automatically generate some of the preservation lemmas. This can be found in the same folder as IL2Coq (named LemmaGen.ml). However, the solution relied too much on specific information of the DSL source, and could not be made into a general solution at the moment. In the future, the toolchain and languages could be extended to allow for lemma templates, which would behave as an inductive rule, where we have some given premises and an end result. An annotation can be added to specify which is the general premise which will be going through induction. In the case of instruction preservation, this would be the induction relation Step. Another alternative solution is to add annotations to tell the CoqIL which are the important inductive relations for preservation, and which type must be preserved. This would allow the existing lemma generation to be more general and with minimal changes to the DSL source.

7.2 Ethical Considerations

This project's ethical considerations can be considered negligible, as the main work involves handling data that have no direct impact on any users. No collection of user data or any processing has been done. The only aspect of ethical consideration of this project is through the inconsistencies found on the WebAssembly 1.0 spec. If any of these inconsistencies lead to bigger problems that can be found in WebAssembly compilers, then it could pose an issue. However, the inconsistencies found have already been discussed with people who are working in WebAssembly specification and no issues have been found.

Bibliography

[1] 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, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349888. doi: 10.1145/3062341.3062363. URL https://doi.org/10.1145/3062341.3062363. pages

[2] Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. Two Mechanisations of WebAssembly 1.0. In *FM 2021 - Formal Methods*, pages 1–19, Beijing, China, November 2021. URL https://hal.science/hal-03353748. pages

[3] Webassembly core specification, 2019. URL https://www.w3.org/TR/wasm-core-1/. pages

[4] WebAssembly Core Specification, 2022. URL https://www.w3.org/TR/wasm-core-2/. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf. pages

[5] Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. Bringing the webassembly standard up to speed with spectec. *Proceedings of the ACM on Programming Languages*, 8 (PLDI), April 2024. ISSN 2475-1421. 45th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI), PLDI 2024 ; Conference date: 24-06-2024 Through 28-06-2024. pages

[6] Wasm Spectec, 2024. URL https://github.com/Wasm-DSL/spectec/tree/main/spectec. pages

[7] WebAssembly Community Group. GarbageCollection, 2023. URL https://github.com/WebAssembly/gc. pages

[8] WebAssembly Community Group. 2023. Thread, 2023. URL https://github.com/WebAssembly/threads. pages

[9] ECMA Ecma. 262: Ecmascript language specification. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,*, 1999. pages

[10] Herbert Schildt. *The Annotated ANSI C Standard: American National Standard for Programming Languages: C*. 1990. ISBN 0-07-881952-0. pages

[11] ISO. *ISO/IEC 14882:1998: Programming languages — C++*. pub-ISO, pub-ISO:adr, September 1998. URL http://www.iso.ch/cate/d25845.html; https://webstore.ansi.org/. pages

[12] Mar 2024. URL https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/. pages

[13] The Coq Development Team. The coq proof assistant, jul 2023. URL https://doi.org/10.5281/zenodo.8161141. pages

[14] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006. URL http://xavierleroy.org/publi/compiler-certif.pdf. pages

[15] Philippa Gardner, Gareth Smith, Conrad Watt, and Thomas Wood. A Trusted Mechanised Specification of JavaScript: One Year On. In Daniel Kroening and Corina S. Pasareanu, editors, *Proceedings of the 27th International Conference on Computer Aided Verification (CAV'15)*, volume 9206 of *Lecture Notes in Computer Science*, pages 3–10. Springer, 2015. doi: 10.1007/978-3-319-21690-4_1. pages

[16] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002. pages

[17] Frédéric Tuong and Burkhart Wolff. Isabelle/c. *Archive of Formal Proofs*, October 2019. ISSN 2150-914x. https://isa-afp.org/entries/Isabelle_C.html, Formal proof development. pages

[18] Martin Strecker. Formal verification of a java compiler in isabelle. In Andrei Voronkov, editor, *Automated Deduction—CADE-18*, pages 63–77, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45620-9. pages

[19] Clara Löh. *The Lean Proof Assistant*, pages 7–20. Springer International Publishing, Cham, 2022. ISBN 978-3-031-14649-7. doi: 10.1007/978-3-031-14649-7_1. URL https://doi.org/10.1007/978-3-031-14649-7_1. pages

[20] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*, volume 1 of *Software Foundations*. Electronic textbook, 2023. pages

[21] Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi: 10.1109/IEEESTD.2019.8766229. pages

[22] Conrad Watt. Mechanising and verifying the webassembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 53–65, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355865. doi: 10.1145/3167082. URL https://doi.org/10.1145/3167082. pages

[23] M. Bodin, P. Gardner, J. Pichon, C. Watt, and X. Rao. WasmCert-Coq, 2019-2024. URL https://github.com/WasmCert/WasmCert-Coq/tree/master. pages

[24] Joachim Breitner, Philippa Gardner, Jaehyun Lee, Sam Lindley, Matija Pretnar, Xiaojia Rao, Andreas Rossberg, Sukyoung Ryu, Wonho Shin, Conrad Watt, and Dongjun Youn. Wasm spectec: Engineering a formal language standard, 2023. pages

[25] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system: Documentation and user's manual. pages

[26] Tej Chajed. Coq Record Update, 2024. URL https://github.com/tchajed/coq-record-update/tree/master. pages

[27] Akira Tanaka, Reynald Affeldt, and Jacques Garrigue. Safe low-level code generation in coq using monomorphization and monadification. *Journal of Information Processing*, 26:54–72, 2018. doi: 10.2197/ipsjjip.26.54. pages

Appendices

Appendix A

Generated Latex from Soundness DSL Source

Some typing judgements were removed as they were too long and not handled well:

- Store_ok
- Module_instance_ok
- Store_extension

$\boxed{\vdash val : valtype}$

$$\frac{}{\vdash t.\mathsf{const}\ c_t : t}\ [\textsc{Val\_ok}]$$

$\boxed{\vdash result : valtype^*}$

$$\frac{(\vdash v : t)^*}{\vdash v^* : t^*}\ [\textsc{Result\_ok-result}]$$

$$\frac{}{\vdash \mathsf{trap} : t^*}\ [\textsc{Result\_ok-trap}]$$

$\boxed{store \vdash externval : externtype}$

$$\frac{S.\mathsf{funcs}[a] = \{\mathsf{type}\ ext,\ \mathsf{module}\ minst,\ \mathsf{code}\ code_{func}\}}{S \vdash \mathsf{func}\ a : \mathsf{func}\ ext}\ [\textsc{Externvals\_ok-func}]$$

$$\frac{S.\mathsf{tables}[a] = \{\mathsf{type}\ tt',\ \mathsf{refs}\ fa^{?*}\} \qquad \vdash tt' \leq tt}{S \vdash \mathsf{table}\ a : \mathsf{table}\ tt}\ [\textsc{Externvals\_ok-table}]$$

$$\frac{S.\mathsf{mems}[a] = \{\mathsf{type}\ mt',\ \mathsf{bytes}\ b^*\} \qquad \vdash mt' \leq mt}{S \vdash \mathsf{mem}\ a : \mathsf{mem}\ mt}\ [\textsc{Externvals\_ok-mem}]$$

$$\frac{S.\mathsf{globals}[a] = \{\mathsf{type}\ (mut\ valtype),\ \mathsf{value}\ (valtype.\mathsf{const}\ val)\}}{S \vdash \mathsf{global}\ a : \mathsf{global}\ (mut\ valtype)}\ [\textsc{Externvals\_ok-global}]$$

$\boxed{store \vdash meminst : memtype}$

$$\frac{mt = [n..m] \qquad \vdash mt : \mathsf{ok}}{S \vdash \{\mathsf{type}\ mt,\ \mathsf{bytes}\ b^*\} : mt}\ [\textsc{Memory\_instance\_ok}]$$

$\boxed{store \vdash tableinst : tabletype}$

$$\frac{tt = [n..m] \qquad (S \vdash \mathsf{func}\ fa : \mathsf{func}\ functype)^{?*} \qquad \vdash tt : \mathsf{ok}}{S \vdash \{\mathsf{type}\ tt,\ \mathsf{refs}\ (fa^?)^*\} : tt}\ [\textsc{Table\_instance\_ok}]$$

$\boxed{store \vdash globalinst : globaltype}$

$$\frac{gt = mut\ vt \qquad \vdash gt : \mathsf{ok} \qquad \vdash v : vt}{S \vdash \{\mathsf{type}\ gt,\ \mathsf{value}\ v\} : gt}\ [\textsc{Global\_instance\_ok}]$$

$\boxed{store \vdash exportinst : \mathsf{ok}}$

$$\frac{S \vdash eval : ext}{S \vdash \{\mathsf{name}\ name,\ \mathsf{value}\ eval\} : \mathsf{ok}}\ [\textsc{Export\_instance\_ok}]$$

$\boxed{store \vdash funcinst : functype}$

$$\frac{\vdash functype : \mathsf{ok} \qquad S \vdash moduleinst : C \qquad C \vdash func : functype}{S \vdash \{\mathsf{type}\ functype,\ \mathsf{module}\ moduleinst,\ \mathsf{code}\ func\} : functype}\ [\textsc{Function\_instance\_ok}]$$

$\boxed{store; context \vdash instr : functype}$
$\boxed{store; context \vdash instr^* : functype}$
$\boxed{store; resulttype \vdash frame; instr^* : resulttype}$

$$\frac{}{S; C \vdash \epsilon : \epsilon \to \epsilon}\ [\textsc{Admin\_instrs\_ok-empty}]$$

$$\frac{S; C \vdash instr_1^* : t_1^* \to t_2^* \qquad S; C \vdash instr_2 : t_2^* \to t_3^*}{S; C \vdash instr_1^*\ instr_2 : t_1^* \to t_3^*}\ [\textsc{Admin\_instrs\_ok-seq}]$$

$$\frac{S; C \vdash instr^* : t_1^* \to t_2^*}{S; C \vdash instr^* : t^*\ t_1^* \to t^*\ t_2^*}\ [\textsc{Admin\_instrs\_ok-frame}]$$

$$\frac{C \vdash instr^* : functype}{S; C \vdash instr^* : functype}\ [\textsc{Admin\_instrs\_ok-instrs}]$$

$$\frac{C \vdash instr : functype}{S; C \vdash instr : functype}\ [\textsc{Admin\_instr\_ok-instr}]$$

$$\frac{}{S; C \vdash \mathsf{trap} : t_1^* \to t_2^*}\ [\textsc{Admin\_instr\_ok-trap}]$$

$$\frac{S \vdash \mathsf{func}\ funcaddr : \mathsf{func}\ (t_1^* \to t_2^*)}{S; C \vdash \mathsf{call}\ funcaddr : t_1^* \to t_2^*}\ [\textsc{Admin\_instr\_ok-call\_addr}]$$

$$\frac{C \vdash instr^* : t_1^? \to t_2^? \qquad S; C, \mathsf{labels}\ (t_1^?) \vdash instr^* : \epsilon \to t_2^? \qquad n = \mathrm{optionSize}(t_1^?)}{S; C \vdash \mathsf{label}_n\{instr^*\}\ instr^* : \epsilon \to t_2^?}\ [\textsc{Admin\_instr\_ok-label}]$$

$$\frac{S; t^? \vdash F; instr^* : t^? \qquad n = \mathrm{optionSize}(t^?)}{S; C \vdash \mathsf{frame}_n\{F\}\ instr^* : \epsilon \to t^?}\ [\textsc{Admin\_instr\_ok-frame}]$$

$$\frac{S; C \vdash instr : t_1^* \to t_2^*}{S; C \vdash instr : t^*\ t_1^* \to t^*\ t_2^*}\ [\textsc{Admin\_instr\_ok-weakening}]$$

$\boxed{store \vdash frame : context}$

$$\frac{S \vdash moduleinst : C \qquad (\vdash val : t)^*}{S \vdash \{\mathsf{locals}\ val^*,\ \mathsf{module}\ moduleinst\} : C, \mathsf{locals}\ t^*}\ [\textsc{Frame\_ok}]$$

$$\frac{S \vdash F : C \qquad S; C, \mathsf{return}\ rt^? \vdash instr^* : \epsilon \to t^?}{S; rt^? \vdash F; instr^* : t^?}\ [\textsc{Thread\_ok}]$$

$\boxed{\vdash config : resulttype}$

$$\frac{\vdash S : \mathsf{ok} \qquad S; \epsilon \vdash F; instr^* : t^?}{\vdash (S; F); instr^* : t^?}\ [\textsc{Config\_ok}]$$

$\boxed{\vdash funcinst : funcinst}$
$\boxed{\vdash tableinst : tableinst}$
$\boxed{\vdash meminst : meminst}$
$\boxed{\vdash globalinst : globalinst}$
$\boxed{\vdash store : store}$

$$\frac{}{\vdash funcinst : funcinst}\ [\textsc{Func\_extension}]$$

$$\frac{n_1 \leq n_2}{\vdash \{\mathsf{type}\ [n_1..m],\ \mathsf{refs}\ (fa_1^?)^*\} : \{\mathsf{type}\ [n_2..m],\ \mathsf{refs}\ (fa_2^?)^*\}}\ [\textsc{Table\_extension}]$$

$$\frac{n_1 \leq n_2}{\vdash \{\mathsf{type}\ [n_1..m],\ \mathsf{bytes}\ b_1^*\} : \{\mathsf{type}\ [n_2..m],\ \mathsf{bytes}\ b_2^*\}}\ [\textsc{Mem\_extension}]$$

$$\frac{(mut = \mathsf{mut}) \vee c_1 = c_2}{\vdash \{\mathsf{type}\ (mut\ t_2),\ \mathsf{value}\ (t_2.\mathsf{const}\ c_1)\} : \{\mathsf{type}\ (mut\ t_2),\ \mathsf{value}\ (t_2.\mathsf{const}\ c_2)\}}\ [\textsc{Global\_extension}]$$

Appendix B

Full Proof of Lemmas in Section 5.3

```
Lemma Binop_typing: forall v_S v_C v_t v_op t1s t2s,
  Admin_instr_ok v_S v_C (admininstr__BINOP v_t v_op) (functype__ t1s t2s) ->
  t1s = t2s ++ [v_t] /\ exists ts, t2s = ts ++ [v_t].
Proof.
  move => v_S v_C v_t v_op t1s t2s HType.
  gen_ind_subst HType.
  - (* Binop *) inversion H; subst; try discriminate.
    injection H3 as H1; subst.
    split => //. exists []. eauto.
  - (* Weakening *) edestruct IHHType as [? [??]] => //=; subst.
    split.
    - repeat rewrite <- app_assoc. reflexivity.
    - exists (v_t ++ x). by rewrite <- app_assoc.
Qed.

Lemma Binop_typing: forall C t op t1s t2s,
be_typing C [::BI_binop t op] (Tf t1s t2s) ->
t1s = t2s ++ [::t] /\ exists ts, t2s = ts ++ [::t].
Proof.
  move => C t op t1s t2s HType.
  gen_ind_subst HType.
  - split => //=. by exists [::].
  - by resolve_compose Econs HType1 IHHType2.
  - edestruct IHHType as [?[??]] => //=; subst.
    repeat rewrite -cat_app; repeat rewrite catA.
    split => //=.
    by eexists.
Qed.
```

Figure B.1: Full proof of Binop Typing in the translation version (above) and WasmCert version (below)