DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

# A Verified, Analysis-Focused Interpreter for WebAssembly

Florian Märkl

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

# A Verified, Analysis-Focused Interpreter for WebAssembly

# Ein Verifizierter, Analyse-Fokussierter Interpreter für WebAssembly

| | |
|---|---|
| Author: | Florian Märkl |
| Supervisor: | Prof. Tobias Nipkow, PhD |
| Advisors: | Simon Rosskopf, Dr. Simon Wimmer |
| Submission Date: | 2021-09-15 |

I confirm that this master's thesis is my own work and I have documented all sources and material used.

| | |
|---|---|
| Florian Märkl | Place, Date |

# Acknowledgments

I would like to thank my advisors Simon Rosskopf and Simon Wimmer for their excellent guidance and continued support while working on this thesis. In addition, I would like to thank Conrad Watt for frequently joining our meetings and providing invaluable advice about WebAssembly-specifics, as well as being available for many interesting talks about compilers and languages. Furthermore, I would also like to thank Tobias Nipkow and Gerwin Klein for writing the terrific Concrete Semantics book, which first introduced me to the fascinating world of theorem proving.

# Abstract

WebAssembly is a standardized general-purpose bytecode language that primarily targets usage as part of web applications. Next to an English prose specification, it also has an official formal specification that describes the language semantics unambiguously in a natural deduction style. This motivates the development of verified tools for static analysis and optimization, since the formal specification can be used as a direct basis to verify against. However, this specification of execution by means of reduction rules has certain inconvenient properties such as mixing data and code, as well as not having any notion of addresses in a program. This makes the application of techniques such as Abstract Interpretation impossible in the sense that information should be determined and stored for specific program points. We overcome these limitations by developing a new interpreter that uses an explicit call stack and value stack, in addition to explicitly tracking a program counter on top of immutable code. This interpreter is developed directly on top of the WebAssembly mechanization developed by Conrad Watt in Isabelle/HOL, a generic theorem proving and logic system. We then present a proof for completeness of our interpreter, showing that it can indeed represent any execution allowed by the WebAssembly specification, which makes it a suitable basis for static analysis. As a side effect of this, we also show that real-world interpreters taking a similar approach using explicit stacks and program counters can also cover the entire WebAssembly language. In addition to the interpreter, we extend the mechanization by formalizing the specification of integer types and operations, as this part had still been missing. We also supply executable implementations for these integer operations and verify them against the specification.

# Contents

# 1 Introduction

## 1.1 WebAssembly

When JavaScript was introduced in 1995 by Netscape [Net95], it was primarily intended as an easy to write scripting language for adding dynamic behavior to HTML pages. Despite the name, the language itself has little to do with the Java programming language itself. Rather, it was designed for glue code that would bind Java applets containing potentially large and complex applications to the website they are embedded in.

With the advent of HTML5 [Wor14], providing features such as video playback natively, external plug-ins for dynamic content such as Java applets and Adobe Flash eventually became obsolete [Byl18] [Tea17] and focus shifted towards JavaScript as the sole target language for implementing complex applications as part of web pages. However, JavaScript's characteristics such as weak and dynamic typing are generally regarded as less suited for development of such large-scale applications. This resulted in the demand for projects such as TypeScript [Tur14], Emscripten [Zak11] and asm.js [HWZ], which reduce JavaScript to a mere intermediate language for delivering compiled platform-independent code, defeating its original purpose as a human-written high-level language without the need for a compilation step. Moreover, these projects still suffer from consequences of its design, such as increased code size, additional parsing overhead and potentially worse performance when compared against bytecode languages such as Java.

This situation motivated the creation of an entirely new language that would use a space-efficient binary format and be closer to real machine architectures, in order to reduce the workload of a just-in-time compiler or interpreter on a target machine. In contrast to native machine code, it should still be entirely platform-independent though. The result was the specification of WebAssembly in 2017 [Haa+17], aiming to fulfill these goals from the very beginning. The combination of a landscape of different browsers and runtimes combined with the requirement for code to always have the same intended behavior on every such platform implies that this language needs to be specified as unambiguously as possible. As such, in addition to English prose definitions, the WebAssembly specification also contains formal rules for the entire language semantics in a natural deduction style.

Such a specification provides an excellent starting point for formal reasoning about the language, as the given rules translate mostly directly to a formal logic system such as Isabelle/HOL. Indeed, Watt [Wat18] presents an Isabelle specification of the WebAssembly semantics, along with an executable interpreter that is proven to be sound with respect to this specification.

## 1.2 Static Analysis for Optimization

With the desire to run code as fast as possible comes the need for optimization. In the case of WebAssembly, one must distinguish between optimizations that happen before code is shipped as bytecode and ones that happen in the final runtime implementation.

The former stage is strictly bound to the possibilities of what can be expressed in the form of WebAssembly bytecode. It should thus not make too many assumptions of the runtime's performance behavior. For example, unrolling a loop that is somehow known to always perform a fixed number of iterations may avoid checks and aid the CPU to perform branch prediction. However, it will also increase code size, possibly leading to more memory having to be fetched and thus less effective usage of caches. As such, this stage should primarily perform optimizations that are guaranteed to eventually be beneficial, such as elimination of dead code. In a practical compiler infrastructure, most of these optimizations may already have been performed before WebAssembly code is emitted, for example when compiling through the LLVM framework [LA04]. However, there may be additional optimization passes tailored specifically to the target language, especially regarding code size, which depends on the exact binary serialization format. An existing project that is developed specifically to optimize WebAssembly is Binaryen [Web21a].

Since the actual runtime stage is aware of the machine it is running on, it can adjust its interpretation or code generation specifically for it, and even take measurements of code execution for profile-guided optimization. Moreover, it is not bound anymore to the exact instruction semantics that WebAssembly offers, as long as it can guarantee that the observed behavior from outside is still preserved. As a concrete example, WebAssembly instructions that access memory are given a dynamic offset into a block of bytes of limited size. According to the specification, the runtime should check at each such access if the offset exceeds this size, and if it does, fall into a trap state, i.e. crash the program. If it has been previously found out by analysis that, no matter what input the program receives, at a certain program point, the offset will always fall into a range that will be valid for access, the concrete check at this point can be avoided. This may be especially beneficial if this program point is hit frequently.

## 1.3 Approaching Abstract Interpretation

We will now sketch a concrete approach towards both dead code elimination and avoidance of bounds checks. In this introductory chapter, the word "variable" is used

```
// Input from outside: uint8_t anise;

// anise = [0, 256);
uint8_t wormwood = 5;
// anise = [0, 256); wormwood = [5, 6)
if (anise < 100) {
    // anise = [0, 100); wormwood = [5, 6)
    wormwood += 4;
    // anise = [0, 100); wormwood = [9, 10)
}
// anise = [0, 256); wormwood = [5, 10)
if (wormwood > 42) {
    // detected as dead code because ∀x∈[5, 10). x <= 42
}
```

Figure 1.1: Example of Abstract Interpretation. Comments describe the results of determined value intervals (the abstract domain) that overapproximate all possible values of variables at each program point, where [x, y) is the interval from x (included) to y (excluded).

in an abstract sense to refer to any storage containing a value that may change during execution. In the concrete case of WebAssembly, this may be for example a local variable, global variable or stack element, as detailed further in chapter 3.

Regarding dead code elimination, whether a block of code is hit is defined by the control flow of the program, which in turn is the result of the combination of input data and conditions that are checked in the code. If for all possible inputs, all conditions that would lead to a certain code block will be false, this block is considered dead code and may be removed safely without affecting the observable behavior of the program.

Whether a given condition is always false depends on the set of values that the variables the condition is made of can have at the program point of the check. As a consequence of Rices's theorem [Ric53], determining these exact sets automatically for any given program would be an undecidable problem. However, it is possible to overapproximate these sets, meaning that the results may consider values that can never occur in a real execution, but conversely, every value that may occur in a real execution is considered.

A method that is commonly used in program analyzers and compilers for calculating such an overapproximation of semantics is Abstract Interpretation [CC77]. It works by defining some abstract domain of values that form a complete lattice, of which every such value is mapped to a set of concrete values. The analysis will then successively execute the program on these abstract values and accumulate the results at each program point, which can be seen as fixpoint iteration of a monotone function over abstract program states. The Knaster-Tarski fixpoint theorem states that every such function has a least fixpoint. It can also be shown that this is the overapproximation of program states we are looking for. Figure 1.1 shows an example of a abstract interpretation on a C program. We will not further detail the inner workings of Abstract Interpretation, but instead refer to the corresponding chapter in the Concrete Semantics book by Tobias Nipkow and Gerwin Klein [NK14], which develops a verified Abstract Interpreter in Isabelle/HOL on the educational language IMP.

The main takeaway for us however is that in order to get the information we need to overapproximate the result of condition checks, we must be able to track information at each program point in a given WebAssembly program. At the same time, our final goal is to develop a verified analysis, implying we will eventually have to prove that this information we track is indeed correct with respect to the official WebAssembly specification. Unfortunately, as detailed further in section 3.4, the specification in its original form is unsuitable as a direct base to verify against, because, amongst other aspects, it has no notion of program points in our sense.

The interpreter developed by Watt [Wat18] shares these same properties and in addition does not cover non-determinism that WebAssembly explicitly allows. Thus, for example, it is not suitable to account for different targets having a different amount of memory available. The verification aspect of this interpreter only considers the soundness property, i.e. that every execution of it is covered by the specification, but not every execution allowed by the specification is covered by the interpreter. Thus, if we would verify our analysis against such an interpreter, the results would only be valid for this specific one and possibly not for other (sound) interpreters.

Motivated by this situation, we develop a new interpreter that explicitly tracks a program counter throughout a given WebAssembly program, as well as covers all possible non-determinism. We show that it is complete with respect to the mechanization of the WebAssembly specification from Watt [Wat18], meaning that it will overapproximate every possible sound WebAssembly interpreter and thus can serve as a base for static analysis. Furthermore, our interpreter tracks an explicit value and call stack, bringing it closer to real-world interpreters.

In addition to such an interpreter, in order to get meaningful results, concrete semantic definitions of integer types and operations are necessary. In our example of figure 1.1, we need to be able to compare numbers such as `5` and `9` to describe an interval `[5, 10)` that contains both of them. We also need to be able to determine that the operation `+= 4` applied to any value in the interval `[5, 6)` will produce a value in `[9, 10)`. Prior to this thesis, the mechanization of WebAssembly did not define concrete types for most kinds of integer values and all operations on them were only uninterpreted functions, making such detailed analysis impossible. We extend the mechanization by translating the respective part of the official specification to Isabelle/HOL, defining the concrete types and operations for integers based on the Word Library [Bee+16] and prove their correctness with respect to the specification.

## 1.4 Background and Notation

Isabelle is a generic logic system and interactive theorem prover that is able to host different logics. The most widely used one is Higher Order Logic (HOL) and generally referred to as Isabelle/HOL. The work that is presented in this thesis is built directly on top of the WebAssembly mechanization by Watt [Wat18], which is formalized entirely in Isabelle/HOL.

Any code in the following chapters that is written in `monospace` is such Isabelle code. For readability, we also make use of classical inference rule notation such as this:

$$\frac{\text{A} \qquad \text{B}}{\text{C}}\ \text{name}$$

If such a rule refers to a part of the formalization, the given name directly refers to the respective name in the Isabelle code.

# 2 Integer Types and Operations

As elaborated in section 1.3, many useful instances of Abstract Interpretation need to reason about the semantics of values that are calculated in a program. Previously, the WebAssembly mechanization by Watt [Wat18] did not come with any specification for such values and operations were treated as mere uninterpreted functions.

The WebAssembly specification [Web21b] defines four basic numeric value types: `i32` and `i64` for integer and `f32` and `f64` for floating point values. We will only consider integers in this thesis as they are more substantial for analysis, given that for example memory addresses and Boolean conditions are always integers in this language. In the following chapter, we will thus translate the official WebAssembly specification of these types to Isabelle/HOL and supply verified implementations for all operations on them.

## 2.1 Specification

Both `i32` and `i64` describe integers of the indicated bit widths. Whether they are interpreted as signed or unsigned is not defined at the type level but merely a consequence of what kind of operations are then applied to them. Apart from conversions between each other, both types share the exact same specifications with the bit width given as a variable $N$, such that it is only formulated once on a type iN of arbitrary bit width.

This fact suggests implementing these types in Isabelle in a bit width-agnostic way too. In order to statically attach the non-zero value $N$ to a type, we make use of the `len` type class from HOL-Library. Its definition is shown in figure 2.1. We will use the `LENGTH('a) :: nat` syntax to retrieve the value attached to some type `'a`. This allows us to avoid manually passing $N$ and the assumption that it is not zero every time they are needed.

```
class len =
  fixes len_of :: 'a itself ⇒ nat
  — ... LENGTH('a) is set up as syntax for len_of TYPE('a) ...
  assumes 0 < LENGTH('a)
```

Figure 2.1: Shortened definition of the `len` type class in HOL-Library, which statically attaches a constant `nat` value to a type `'a`. After instantiating, `LENGTH('a)` conveniently refers to this value.

### 2.1.1 Integer Interpretations

At their basis, values of iN in the specification are interpreted as integer numbers in a mathematical sense in the range $[0, 2^N - 1]$. On top of this, such a number $i$ may be reinterpreted as a bit vector, that is, as a list of boolean values representing the number in base two:

$$\text{ibits}_N(i) = d_{N-1} \ \ldots \ d_0 \ (i = 2^{N-1} \cdot d_{N-1} + \ \ldots \ + 2^0 \cdot d_0) \tag{2.1}$$

Our direct translation to Isabelle/HOL, where `'a` is any given type in the `len` type class, is the following:

```
definition ibits :: 'a itself ⇒ int ⇒ bool list where          (2.2)
  ibits _ i ≡ THE l.
    length l = LENGTH('a) ∧
    i = (∑n ∈ {0..<LENGTH('a)}.
        (2 ^ (LENGTH('a) - n - 1)) * (if l ! n then 1 else 0))
```

By induction over `LENGTH('a)` we prove that for any input value in a sensible range, this is equal to the `bin_to_bl` function from Isabelle's Word Library, thus giving us access to all auxiliary lemmas that are defined for it:

$$\frac{\texttt{0} \leq \texttt{i} \qquad \texttt{i < 2 \^{} LENGTH('a)}}{\texttt{ibits N i = bin\_to\_bl LENGTH('a) i}}\ \text{ibits}$$

(2.2) already contains a formula for converting the list `l` back to its integer value `i` and we may use it to define the inverse function $\text{ibits}_N^{-1}(l)$ called `ibits_inv` in our formalization.

Next to the binary conversion is the interpretation as a signed number in twos complement, shifting numbers greater than or equal to $2^{N-1}$ into the negative range:

$$\begin{aligned} \text{signed}_N(i) &= i & (0 \leq i < 2^{N-1}) \\ \text{signed}_N(i) &= i - 2^N & (2^{N-1} \leq i < 2^N) \end{aligned} \tag{2.3}$$

Translated to Isabelle, this yields:

```
definition signed :: 'a itself ⇒ int ⇒ int where               (2.4)
  signed _ i ≡
    if 0 ≤ i ∧ i < (2^(LENGTH('a)-1)) then
      i
    else if 2^(LENGTH('a)-1) ≤ i ∧ i < 2^LENGTH('a) then
      i - (2^LENGTH('a))
    else
      0
```

The specification claims that this function is bijective on the given ranges and later refers to the inversion $\mathrm{signed}_N^{-1}(i)$. We show that this is indeed the case:

signed_bij
```
bij_betw (signed N) {0 ..< 2^LENGTH('a)}
  {-(2^(LENGTH('a)-1)) ..< 2^(LENGTH('a)-1)}
```

As such, we are also able to define the inverse function `signed_inv` and make use of it in later definitions.

## 2.1.2 Operations

Next, we are going to translate all specified WebAssembly integer operations to Isabelle. To be bit width-agnostic, they are specified as a type class extending `len`. This class, called `wasm_int_ops` and shown in Figure 2.2, introduces all operations syntactically,

```
class wasm_base = zero
class wasm_int_ops = wasm_base + len +
  — unops   ...
  — binops
  fixes int_add :: 'a ⇒ 'a ⇒ 'a
  fixes int_div_s :: 'a ⇒ 'a ⇒ 'a option
  ...
  — testops   ...
  — relops   ...
  — value conversions
  fixes int_of_nat :: nat ⇒ 'a
  fixes nat_of_int :: 'a ⇒ nat
begin
  abbreviation abs_int :: 'a ⇒ int where abs_int a ≡ int (nat_of_int a)
  abbreviation rep_int :: int ⇒ 'a where rep_int a ≡ int_of_nat (nat a)
  abbreviation abs_int_bits :: 'a ⇒ bool list where
    abs_int_bits a ≡ ibits TYPE('a) (abs_int a)
  abbreviation rep_int_bits :: bool list ⇒ 'a where
    rep_int_bits a ≡ rep_int (ibits_inv TYPE('a) a)
  abbreviation abs_int_s :: 'a ⇒ int where
    abs_int_s a ≡ signed TYPE('a) (abs_int a)
  abbreviation rep_int_s :: int ⇒ 'a where
    rep_int_s a ≡ rep_int (signed_inv TYPE('a) a)
end
```

Figure 2.2: Bit width-agnostic, purely syntactical definition of integer operations on some opaque type `'a`. Additional conversions to concrete representations are defined too for further specification of semantics.

alongside requiring the conversions `int_of_nat` and `nat_of_int` between `'a` and `nat`. On top of these, all previously defined representations are derived as abbreviations prefixed by `abs_int` and `rep_int`, to and from `'a`, respectively.

This type class is extended to another class `wasm_int` by adding all semantic specifications. As an example, let us consider the case of addition. The WebAssembly specification defines it as the following equation:

$$\mathrm{iadd}_N(i_1, i_2) = (i_1 + i_2) \mod 2^N \tag{2.5}$$

Using the `abs_int` and `rep_int` conversions to bring values of `'a` into the form that this expects and back, the equation directly translates to Isabelle:

```
class wasm_int = wasm_int_ops +                                    (2.6)
  ...
  assumes add: int_add (i₁::'a) i₂ =
    rep_int ((abs_int i₁ + abs_int i₂) mod (2^LENGTH('a)))
  ...
```

Some of the operations are allowed to fail and thus not return any value, for which these operations in Isabelle return `'a option` instead of `'a`. The WebAssembly specification uses a more general set notation to describe all possible result values or the absence of any. But as it actually only specifies either no or a single result for integer operations, our option type can sufficiently cover it.

A particularly interesting example of failure cases, which will also need special care in our concrete implementation, is the signed division $\mathrm{idiv\_s}_N$:

$$\mathrm{idiv\_s}_N(i_1, 0) = \{\} \tag{2.7}$$
$$\mathrm{idiv\_s}_N(i_1, i_2) = \{\} \qquad \textbf{if} \quad \mathrm{signed}_N(i_1)/\mathrm{signed}_N(i_2) = 2^{N-1} \tag{2.8}$$
$$\mathrm{idiv\_s}_N(i_1, i_2) = \mathrm{signed}_N^{-1}(\mathrm{trunc}(\mathrm{signed}_N(i_1)/\mathrm{signed}_N(i_2))) \quad \textbf{otherwise} \tag{2.9}$$

Here, division results may leave the integer domain and will be represented as rational numbers instead. A function `trunc :: rat ⇒ int` is defined that removes any decimal digits. Using it, signed division may be translated to Isabelle:

```
assumes div_s_0: i₂ = 0 ⟹ int_div_s (i₁::'a) i₂ = None               (2.10)
assumes div_s_nrep:
  i₂ ≠ 0
  ⟹ rat_of_int (abs_int_s i₁) / of_int (abs_int_s i₂) = 2^(LENGTH('a)-1)
  ⟹ int_div_s (i₁::'a) i₂ = None
assumes div_s:
  i₂ ≠ 0
  ⟹ rat_of_int (abs_int_s i₁) / of_int (abs_int_s i₂) ≠ 2^(LENGTH('a)-1)
  ⟹ int_div_s (i₁::'a) i₂ = Some (rep_int_s (
      trunc (of_int (abs_int_s i₁) / of_int (abs_int_s i₂))))
```

Besides the intuitive case of division by zero in (2.7) and `div_s_0`, the result of the specific calculation $(-2^{N-1})/(-1) = 2^{N-1}$ falls outside the range $[-2^{N-1}, 2^{N-1} - 1]$ and is thus not representable in twos complement. This special case is taken care of by (2.8) and `div_s_nrep`.

## 2.2 Implementation

This pure specification as a type class does not yet supply any concrete types or definitions of operations that may actually be executed. We will supply these now by reusing types and functions from Isabelle's Word Library [Bee+16] as much as possible and prove that they fulfill the specification.

### 2.2.1 Definitions

Concrete `i32` and `i64` types are defined as typedefs without further restrictions and may be directly brought into the `len` and `wasm_base` type classes:

```
typedef i32 = UNIV :: (32 word) set ..                                  (2.11)

instantiation i32 :: wasm_base begin
  lift_definition zero_i32 :: i32 is of_nat 0 .
end

instantiation i32 :: len begin
  definition len_of_i32 :: i32 itself ⇒ nat where len_of_i32 _ = 32
end
```

`i64` is defined analogously. While working towards also bringing both types into the full `wasm_int` type class, we again strive to avoid code duplication as much as possible. This approach does require a certain amount of boilerplate code in Isabelle, but the significant advantage is that all heavyweight definition and proof code may eventually be written down only once. The idea is to first abstract both typedefs as a locale `Wasm_Int_Word`, inside which all necessary definitions and proofs can be prepared to a degree that later instantiating `wasm_int` for the concrete types will be trivial:

```
locale Wasm_Int_Word =                                                  (2.12)
  fixes rep :: 'a :: {len, wasm_base} ⇒ 'n :: len word
  fixes abs :: 'n word ⇒ 'a
  assumes length[simp]: LENGTH('a) = LENGTH('n)
  and abs_rep: ⋀x. abs (rep x) = x
  and rep_abs: ⋀x. rep (abs x) = x
  and rep_0: rep 0 = 0
```

Here, `'a` is the concrete integer type and `'n` is the type argument given to the Word Library's `'n word`. With `rep` and `abs` being the respective `Rep_i32`/`Rep_i64` and `Abs_i32`/`Abs_i64` constants given by the typedefs, interpreting this locale for our types becomes straightforward.

Inside the locale, concrete definitions are given for all operations on `'n word`. For example for addition the `(+)` operator from the Word Library is used directly:

```
definition int_add :: 'n word ⇒ 'n word ⇒ 'n word where          (2.13)
  int_add ≡ (+)
```

Let us again consider signed division as a more complex example. The Word Library provides an operator `(sdiv) :: 'n word ⇒ 'n word ⇒ 'n word` as a total function on `'n word`, mapping our cases (2.7) and (2.8) to some bogus values. To build the partial function defined by WebAssembly on top of it however, both specified failure cases have to be caught explicitly and mapped to `None`. This is done exactly as specified for division by zero by checking whether the right operand is equal to `0`. The unrepresentable case needs extra care though. Because the result of `(sdiv)` is already a valid `'n word`, it is impossible to determine from this final result alone whether the underlying division would have left the respective range. Instead, the operands are again checked to explicitly avoid the specific case $(-2^{N-1})/(-1) = 2^{N-1}$, leading to the final definition of `int_div_s`:

```
definition int_div_s :: 'n word ⇒ 'n word ⇒ 'n word option where
  int_div_s i₁ i₂ ≡
    if i₂ = 0 ∨ (i₁ = of_int (-(2^(LENGTH('n) - 1))) ∧ i₂ = of_int (-1))
    then None
    else Some (i₁ sdiv i₂)
```

### 2.2.2 Proof against Specification

With all definitions in place, the goal is to prove that they fulfill all properties given by the `wasm_int` type class. As we are still inside the `Wasm_Int_Word` locale with abstract type `'a`, we can not yet actually instantiate the class for this type yet. However, it is still possible to interpret the underlying (automatically generated) locale of `wasm_int`, in order to obtain a fact that will make future interpretations based on it trivial. More concretely, a `sublocale` relationship between `Wasm_Int_Word` and `wasm_int` can be established where all required constants are lifted through `abs` and `rep` to account for the typedef nature of `i32` and `i64`:

```
abbreviation lift2 ≡ map_fun rep (map_fun rep abs)
abbreviation lift2o ≡ map_fun rep (map_fun rep (map_option abs))

sublocale Int: wasm_int where                                    (2.14)
  ... and int_add = lift2 int_add and ...
  ... and int_div_s = lift2o int_div_s and ...
```

Wherever the definitions of operations given by the Word Library sufficiently match the WebAssembly specification, the proof goals are solved by only simplification. Other operations require additional effort, such as again signed division. Previously, while composing the definition to account for (2.8), we have blindly assumed that this case is hit exactly when the operand match the ones in our specific check. In other words, $i_1/i_2 = 2^{N-1}$ if and only if $i_1 = -2^{N-1}$ and $i_2 = -1$. And indeed, we show that this is the case:

```
lemma sdiv_nrep:                                                 (2.15)
  (i₁::'a::len word) = of_int (-(2^(LENGTH('a)-1))) ∧ i₂ = of_int (-1)
  ⟷ rat_of_int (sint i₁) / of_int (sint i₂) = 2^(LENGTH('a)-1)
```

With the proof for the `wasm_int` sublocale finished, we leave the `Wasm_Int_Word` locale context and may now easily instantiate the actual `wasm_int` class for `i32` and `i64` by making use of `Wasm_Int_Word`.

```
interpretation I32: Wasm_Int_Word Rep_i32 Abs_i32                (2.16)
instantiation i32 :: wasm_int begin
  ...
  lift_definition int_add_i32 :: i32 ⇒ i32 ⇒ i32 is I32.int_add .
  lift_definition int_div_s_i32 :: i32 ⇒ i32 ⇒ i32 option is I32.int_div_s .
  ...
instance
  ...
  by (rule I32.Int.wasm_int_axioms[...])
end
```

The instantiation for `i64` is carried out analogously, which wraps up our integer implementation.

# 3 Specification of Code Execution

The official WebAssembly specification [Web21b] describes the semantics of the language in the form of small step rules for all individual instructions. As mentioned previously, these rules are given in both an English prose form for intuitive understanding by a human reader, as well as a formal one in a natural deduction style. Henceforth, when referring to the formal specification, we will only refer to the existing Isabelle/HOL formalization from Watt [Wat18], as we will later directly build upon it. It follows the original specification exactly, with a few minor exceptions that are described in the respective paper.

WebAssembly is often loosely described as a stack-based language, meaning that a stack of operand values is tracked as the program is executed. Each instruction may pop a certain amount of entries from the stack, perform some operation using these values and push a result back onto the stack to be consumed by subsequent instructions. This concept has been well-known for a long time, for example as part of Java bytecode [Lin+15] or the Forth programming language [The80]. Figure 3.1 shows an example of a (simplified) WebAssembly prose specification, in this case for the `Binop` instruction that computes a binary operation such as addition or subtraction. It shows the explicit operations of interacting with the operand stack.

1. Assert: due to validation, two values of value type $t$ are on the top of the stack.
2. Pop the value $t$.const $c_2$ from the stack.
3. Pop the value $t$.const $c_1$ from the stack.
4. Let c be a possible result of computing $\text{binop}_t(c_1, c_2)$.
5. Push the value $t$.const $c$ to the stack.

Figure 3.1: Simplified prose specification for the `Binop` instruction. $\text{binop}_t(c_1, c_2)$ may for example be the addition of $c_1$ and $c_2$.

## 3.1 Configurations

Interestingly, the accompanying formal specifications do not have an explicit notion of a pure value stack and instead, operand values are recorded as part of a list of (nested) instructions. Concretely, the state of a program, called a configuration, is expressed as a tuple (|`s;f;es`|) where

`s :: s` denotes the store, i.e. an object that contains all available global state such as function closures, memory and global variables,

`f :: f` is called a frame, containing any state local to an invoked function, such as local variables, and

`es :: e list` is the list of instructions to be executed.

Figure 3.2 shows the according Isabelle definitions of these types. Notice also the added syntax `$` and `C` for converting a basic instruction to an administrative instruction and converting a value to a constant instruction, respectively. For example, `$C v` would expand to `Basic (EConst v)`. `$*` and `$C*` map these operations over entire lists. The syntax `at _> rt :: tf` denotes the function type `Tf at rt` of arguments and return values typed according to the lists `at` and `rt`, respectively.

`inst`, as part of the frame `f` contains types and indices into the global store. This is particularly important when multiple modules are loaded simultaneously into the same store. In such a case, the objects from each module will be appended one after the other to the respective lists contained in the type `s`, as part of a process called instantiation. Code which refers to these objects by index however is kept unchanged, so the instance `inst` is created as an indirection of indices from code to indices in the store.

All instructions that can be present in raw WebAssembly code are represented by `b_e`. For the instruction list `es` in a configuration however, these are extended to the type `e` by additional so-called administrative instructions. These are only created as part of the execution and are not present in the binary format that a module is typically loaded from.

A specialty of WebAssembly in contrast to other assembly-like languages is the fact that it employs structured control flow. Instead of an explicit goto, it uses the instructions `Block`, `Loop` and `If`, which themselves contain a list of instructions. Thus, a full program takes the form of a tree. The details of how this concept of control flow works in practice will be explained in section 3.2.1.

```
type_synonym i = nat

datatype — value types
  t = T_i32 | T_i64 | T_f32 | T_f64

datatype — function types
  tf = Tf t list t list (_ '_> _ 60)

datatype — values
  v =
    ConstInt32 i32
    | ConstInt64 i64
    | ConstFloat32 f32
    | ConstFloat64 f64

datatype — basic instructions
  b_e =
    Unreachable
    | Nop
    | Drop
    | Select
    | Block tf b_e list
    | Loop tf b_e list
    | If tf b_e list b_e list
    | Br i
    | Br_if i
    | Br_table i list i
    | Return
    | Call i
    | Call_indirect i
    | Get_local i
    | Set_local i
    | Tee_local i
    | Get_global i
    | Set_global i
    | Load t (tp × sx) option a off
    | Store t tp option a off
    | Current_memory
    | Grow_memory
    | EConst v (C _ 60)
    | Unop t unop
    | Binop t binop
    | Testop t testop
    | Relop t relop
    | Cvtop t cvtop t sx option
```

```
record inst = — instances
  types :: tf list
  funcs :: i list
  tabs :: i list
  mems :: i list
  globs :: i list

record s = — store
  funcs :: cl list
  tabs :: tabinst list
  mems :: mem list
  globs :: global list

record f = — frame
  f_locs :: v list
  f_inst :: inst

datatype e = — administrative instructions
  Basic b_e ($_ 60)
  | Trap
  | Invoke i
  | Label nat e list e list
  | Frame nat f e list

abbreviation to_e_list
  :: b_e list ⇒ e list ($* _ 60)
where
  to_e_list b_es ≡ map Basic b_es

abbreviation v_to_e_list
  :: v list ⇒ e list ($C* _ 60)
where
  v_to_e_list ves ≡ map (λv. $C v) ves
```

Figure 3.2: Basic types that make up a WebAssembly configuration (|s;f;es|).

## 3.2 Reduction Relation

Whether one configuration can step to another in the sense of small step semantics is described by the relation (|s;f;es|) $\rightsquigarrow$ (|s';f';es'|), of which an excerpt is shown in figure 3.3. Many instructions only affect es but leave s and f untouched. For these, the simplified relation (|es|) $\rightsquigarrow$ (|es'|) is used.

Notice how the rule for Binop from figure 3.1 is now represented as (|[$C v1, $C v2, $(Binop t op)]|) $\rightsquigarrow$ (|[$C v]|), assuming that v is the result of the binary operation op applied to v1 and v2. Intuitively, this means that a program consisting exactly of two constant instructions and one Binop instruction is reduced to the result of the respective computation.

Of course, practical programs will not consist only of a single operation and its operands, so these rules must be extended to the greater context of an entire program, which is the purpose of the label rule. The Lfilled predicate used by it is the key to establishing this context. Lfilled k lholed es les means that es :: e list is contained within les :: e list with its surrounding context defined by lholed :: Lholed. Or in other words: les is the list es with surrounding code defined by lholed added. If then the smaller es can be reduced to some es', the larger les can also be reduced to les', which is es' surrounded by the same context lholed.

Before diving into the exact definitions of Lfilled and Lholed, we will trust our intuition on what they should do and show the idea by an example. Figure 3.4 shows a simple program that is being executed. For the first step, we notice that by rule binop_Some:

```
(|[$C 1337, $C 123, $Binop Add]|) ⇝ (|[$C 1460]|)                    (3.1)
```

In addition, we find some k and lholed that describe a prefix of [$C 1502] and a suffix of [$Binop (_Suc)]. We can then embed our two states from (3.1) to derive the Lfilled predicate for the respective full lists:

```
Lfilled k lholed                                                     (3.2)
        [$C 1337, $C 123, $Binop Add]
        [$C 1502, $C 1337, $C 123, $Binop Add, $Binop Suc]

Lfilled k lholed                                                     (3.3)
        [$C 1460]
        [$C 1502, $C 1460, $Binop Suc]
```

Finally, (3.1) (3.2) and (3.3) combined match the premises for the label rule from Figure 3.3, letting us conclude the full step performed in figure 3.4. The resulting program is the reduced further in figure 3.5. As soon as the instruction list consists either of only of constant instructions, or a single Trap instruction, the program is considered to have terminated, which is now the case.

$$\frac{}{\llparenthesis\texttt{[\$C v, \$Unop t op]}\rrparenthesis \leadsto \llparenthesis\texttt{[\$C (app\_unop op v)]}\rrparenthesis}\ \text{unop}$$

$$\frac{\texttt{app\_binop op v1 v2 = Some v}}{\llparenthesis\texttt{[\$C v1, \$C v2, \$Binop t op]}\rrparenthesis \leadsto \llparenthesis\texttt{[\$C v]}\rrparenthesis}\ \text{binop\_Some}$$

$$\frac{\texttt{length vs = n} \qquad \texttt{length t1s = n} \qquad \texttt{length t2s = m}}{\begin{array}{c}\llparenthesis\texttt{(\$C* vs) @ [\$Block (t1s \_> t2s) es]}\rrparenthesis \\ \leadsto \llparenthesis\texttt{[Label m [] ((\$C* vs) @ (\$* es))]}\rrparenthesis\end{array}}\ \text{block}$$

$$\frac{\texttt{length vs = n} \qquad \texttt{length t1s = n} \qquad \texttt{length t2s = m}}{\begin{array}{c}\llparenthesis\texttt{(\$C* vs) @ [\$Loop (t1s \_> t2s) es)]}\rrparenthesis \\ \leadsto \llparenthesis\texttt{[Label n [\$(Loop (t1s \_> t2s) es)] ((\$C* vs) @ (\$* es))]}\rrparenthesis\end{array}}\ \text{loop}$$

$$\frac{\llparenthesis\texttt{e}\rrparenthesis \leadsto \llparenthesis\texttt{e'}\rrparenthesis}{\llparenthesis\texttt{s;f;e}\rrparenthesis \leadsto \llparenthesis\texttt{s;f;e'}\rrparenthesis}\ \text{basic}$$

$$\frac{}{\llparenthesis\texttt{s;f;[\$(Call j)]}\rrparenthesis \leadsto \llparenthesis\texttt{s;f;[Invoke (sfunc\_ind (f\_inst f) j)]}\rrparenthesis}\ \text{call}$$

$$\frac{\begin{array}{c}\llparenthesis\texttt{s;f;es}\rrparenthesis \leadsto \llparenthesis\texttt{s';f';es'}\rrparenthesis \\ \texttt{Lfilled k lholed es les} \\ \texttt{Lfilled k lholed es' les'}\end{array}}{\llparenthesis\texttt{s;f;les}\rrparenthesis \leadsto \llparenthesis\texttt{s';f';les'}\rrparenthesis}\ \text{label}$$

$$\frac{\llparenthesis\texttt{s;f;es}\rrparenthesis \leadsto \llparenthesis\texttt{s';f';es'}\rrparenthesis}{\llparenthesis\texttt{s;f0;[Frame n f es]}\rrparenthesis \leadsto \llparenthesis\texttt{s';f0;[Frame n f' es']}\rrparenthesis}\ \text{local}$$

Figure 3.3: An excerpt of the reduction relation describing the small step semantics of WebAssembly. Rules that do not make use of the store or frame and are thus specified as $\llparenthesis\texttt{es}\rrparenthesis \leadsto \llparenthesis\texttt{es'}\rrparenthesis$ are lifted to the full form $\llparenthesis\texttt{s;f;es}\rrparenthesis \leadsto \llparenthesis\texttt{s;f;es'}\rrparenthesis$ by the `basic` rule.

```
[
  $C 1502,
  $C 1337,
  $C 123,
  $Binop Add,
  $Binop Sub
]
```

$\leadsto$

```
[
  $C 1502,
  $C 1460,
  $Binop Sub
]
```

Figure 3.4: Reduction of an addition instruction (yellow) embedded in a surrounding context (grey). Additional constants necessary in the mechanization have been elided for brevity.

```
[
  $C 1502,
  $C 1460,
  $Binop Sub
]
```

$\leadsto$

```
[
  $C 42
]
```

Figure 3.5: Reduction of an subtraction instruction (yellow) without any surrounding context.

#### Non-Deterministic Reduction Derivations

Keep in mind that the sequence of rules to apply in order to derive a given reduction is not unique. For example, instead of defining `k` and `lholed` as we did, we may first apply the `label` rule with a context describing only the prefix `[$C _ 1502]` and an empty suffix. Then we could apply the rule again with an empty prefix and the suffix `[$Binop _ (_Suc)]`, which eventually leads to the same reduction. It is even possible to apply this rule with an empty context, which makes it conclude the exact reduction given as a premise. By this, it is theoretically possible to derive any reduction by itself an arbitrary number of times before progressing further. With the exception of the `Trap` instruction, this does not affect which actual steps that can be taken by a program though. It is merely a detail regarding the ways to show that these steps can be taken.

### 3.2.1 Control Flow using Labels

Until now, we have only considered purely linear programs, that is, programs without any instructions affecting the control flow. As hinted before, WebAssembly's way of implementing control flow is quite special in the way it implements structured control flow. The three relevant instructions here are `Block`, `Loop` and `If`. Because `If` itself is turned into a `Block` in a first step at runtime, we only need to consider the execution of `Block` and `Loop` now.

Both of these consist of a function type specifying the number and individual types of input and output values, and a list of instructions contained in the block or loop.

```
[
  $C 42,
  $Block ([T_i32] _> [T_i32]) [
    C 1,
    Binop Add
  ]
]
```

$\rightsquigarrow$

```
[
  Label 1 [] [
    $C 42,
    $C 1,
    $Binop Add
  ]
]
```

Figure 3.6: Reduction of a `Block` to a `Label`. One constant instruction is moved into the label according to the argument list of the `Block`'s type.

```
[
  Label 1 [$Loop [C 1, Binop Add, Br 0]] [
    $C 5,
    $Br 0
  ]
]
```

$\rightsquigarrow$

```
[
  $C 5,
  $Loop [
    C 1,
    Binop Add,
    Br 0
  ]
]
```

Figure 3.7: Reduction of a `Br 0` instruction, which replaces the innermost label by its continuation, in this case a `Loop`, with appropriate arguments prepended.

Whenever an instruction `Block _ bs` or `Loop _ bs` is hit during execution, it is unfolded to a `Label _ es (($C* vs) @ ($* bs))` administrative instruction, where `vs` is a list of input values of the appropriate length taken from before the instruction. `es :: e list` is a special additional instruction list called the label's continuation. This process of reduction is illustrated in Figure 3.6. As soon as such a `Label` is present, it can be expressed as part of a context for the `Lfilled` predicate, which has the consequence that the instructions inside of it will be reduced in the following steps.

Complementing `Block` and `Loop` exists the `Br` instruction, which, depending on the situation, may be regarded as the equivalent of either the `break` or `continue` statement in classic imperative languages such as C. `Br i` means that the `i`-th label in the current hierarchy of nested labels should be "broken out of". Counting of labels starts from the inside, such that `Br 0` will target the innermost label directly surrounding the `Br`.

Breaking out of a `Label (n :: nat) (es :: e list) (les :: e list)` concretely means that the label currently containing `les` is replaced entirely by its continuation `es` prepended with `n` constants taken from just before the `Br`.

Figure 3.7 shows an example for this behavior. When a `Loop` is unfolded, the continuation of the label becomes a clone of the initial `Loop` instruction itself. This has the effect that when this label is targeted by a `Br`, the entire label is again replaced by the loop, only with updated input values. This causes this `Loop` to be entered yet again in the next step, giving the `Br` a meaning similar to `continue` in C.

In the case of a `Block`, the continuation is empty, like in figure 3.6. Thus, a `Br` would behave like `break` in C by dissolving the label into only some output values.

### 3.2.2 Establishing Context with Lfilled

With the knowledge of labels, we can finally explain the concrete definition of `Lholed`, as used by the `label` reduction rule. This recursive definition, as it appears in the mechanization, is shown in figure 3.8.

```
datatype Lholed =
  — L0 = v* [<hole>] e*
  LBase v list e list
  — L(i+1) = v* (label n e*  Li) e*
| LRec v list nat e list Lholed e list

inductive Lfilled :: nat ⇒ Lholed ⇒ e list ⇒ e list ⇒ bool where
  L0:⟦lholed = (LBase vs es')⟧
  ⟹ Lfilled 0 lholed es (($C* vs) @ es @ es')
| LN:⟦lholed = (LRec vs n es' l es''); Lfilled k l es lfilledk⟧
  ⟹ Lfilled (k+1) lholed es (($C* vs) @ [Label n es' lfilledk] @ es'')
```

Figure 3.8: Definitions of the `Lfilled` predicate and accompanying `Lholed` type, describing that a list of instructions `es` is contained within some larger list and surrounded by a context defined by `lholed`.

There are two cases how `Lfilled` can be established. As the base case defined by the `L0` rule, a given list may be prepended and appended by additional constants and instructions, respectively, using the `LBase` constructor to describe these surroundings. Building on top of that, additional labels can be added around using the `LN` rule. Here, an already `Lfilled` list is embedded in a label with additional prefix and suffix. The `LRec` constructor recursively contains the inner `Lholed` along with the additional `n` and `es'` values to fully define the `Label` instruction.

It has been shown by [Wat18] that this predicate is deterministic in its last argument, proving that this relation is a partial function:

$$\frac{\texttt{Lfilled k lfilled es les} \qquad \texttt{Lfilled k lfilled es les'}}{\texttt{les = les'}}\ \text{lfilled\_deterministic} \tag{3.4}$$

This, along with similar properties, plays a key role in reasoning about reductions of programs. In particular, for rule `label` in figure 3.3, if `k`, `lholed`, `es` and `es'` are given, then `les` and `les'` are uniquely defined.

## 3.3 Typing

In order to ensure that a program will never get stuck or otherwise reach an invalid state not covered explicitly by a `Trap`, WebAssembly is statically typed. In addition to other properties that must be fulfilled by a module to be considered well-typed, the specification describes a relation `C ⊢ bs : at _> rt` describing that a piece of code `bs` consumes a specific number of constants matching the types given by `at` and returns constants according to `rt`. This property is constrained to the context `C`, which contains all necessary information coming from outside the code block itself, such as the types of global and local variables.

For example, typing of a `Binop` instruction is defined by the following rule:

$$\frac{\texttt{binop\_t\_agree op t}}{\texttt{C} \vdash \texttt{Binop t op : [t, t] \_> t}}\ \text{binop}$$

This means that a `Binop` instruction consumes two constants and returns one. All these values must be of the same type `t` and this type must be compatible with the operation `op`. Such individual typing rules are also extended to arbitrarily structured code in an inductive way, similar to reduction.

The specification states that a module must be fully type-checked by the runtime before its code may be executed. This means that when analyzing code in a practical case, it is generally sufficient to only consider code that is already checked and known to be well-typed. Thus, all properties that can be derived from typing can be made use of for free.

## 3.4 Takeaways

The main takeaway of how execution is specified by means of the reduction relation is that in its original form, WebAssembly has no notion of program addresses or a program counter. As soon as the first reduction has been applied to a given program, any direct correspondence between the program and the current state is immediately lost, making it impossible to attach certain properties to positions in the program by means of Abstract Interpretation.

This is why in the next chapter we will present an interpreter that overcomes these limitations and show that it covers the entire WebAssembly execution specification.

# 4 Interpreter

As previously mentioned, our newly-developed interpreter is designed to have the following characteristics:

- An explicit **program counter** is tracked on top of immutable code.
- Both the **call stack** and **value stack** are explicitly stored.
- The interpreter is **complete**, i.e. from a given interpreter state, all states that are reachable by means of reduction can also be reached by the interpreter. As a consequence of this and non-determinism being present in the reduction relation, the interpreter is **non-deterministic** too.

The high-level idea for verification is that every state of such an interpreter may be reconstructed to exactly one WebAssembly configuration. If, for any interpreter state `state`, if its reconstructed configuration ⦇`s;f;es`⦈ may be reduced to another configuration ⦇`reduced_s;f';es'`⦈, we are able to show that our interpreter can step from `state` to some `state'` whose reconstruction is exactly ⦇`reduced_s;f';es'`⦈, then we have proven completeness. This approach is illustrated in figure 4.1.

The following sections will present the design of data structures and functions composing this interpreter, as well as outline the proof for completeness.

### 4.0.1 One-to-one Correspondence

One final goal of the interpreter is to be able to provide a one-to-one correspondence between interpreter steps and reductions, which is necessary to directly relate the two

$$\begin{tikzcd}
\texttt{state} \arrow[r, "\text{step}"] \arrow[d, "\text{reconstruct}"'] & \texttt{state'} \arrow[d, "\text{reconstruct}"] \\
(\!|\texttt{s;f;es}|\!) \arrow[r, "\text{reduce}"'] & (\!|\texttt{s';f';es'}|\!)
\end{tikzcd}$$

Figure 4.1: Illustration of the relationship between interpreter steps and reduction of WebAssembly configurations. Any interpreter state can be reconstructed to a configuration. If this configuration can be reduced further, the interpreter can also step to a state that reconstructs to the same result.

processes in a proof. There are two kinds of incidents that would initially prevent this:

1. A single step of the interpreter corresponds to multiple reductions being applied after another. (4.1)
2. A single reduction requires multiple steps in the interpreter. (4.2)

The first case is tackled directly in the interpreter implementation. This is done by artificially introducing intermediate states that are more fine-grained than the most intuitive implementation of an interpreter would require. While they hold no concrete value for the interpreter itself, the reconstruction can make use of them and build the instruction list accordingly.

The second case will only be dealt with at a later point by simply stepping the interpreter multiple times whenever needed.

## 4.1 Implementation

### 4.1.1 State

The very core of the interpreter is formed by its state type `c_state`, which is shown in figure 4.2. It represents an entire state of a program being executed. The prefix `c_` is used here and in future definitions to disambiguate constants and types belonging to our interpreter from other entities of the same short names.

Identical to `s` in a WebAssembly configuration (|`s;f;es`|), a global store is saved in the field `c_s`. This is where the similarities end though. Our interpreter represents the local state of a single function invocation as a triple `(pc :: c_pc, locs :: v list, vs :: v list)` where

- `pc` is the program counter,
- `locs` contains all local variable contents and
- `vs` is the value stack.

```
record c_state =
  c_s :: s — store
  c_pc :: c_pc — program counter
  c_locs :: v list — local variables
  val_stack :: v list
  call_stack :: (c_pc × v list × v list) list — pc × locs × vs
  trapped :: bool — an instruction has trapped
  returning :: bool — special state just before returning from a function
  terminated :: bool — final result state, no further stepping possible
```

Figure 4.2: Data structure representing an entire state of the interpreter.

The triple of the current function is unfolded in `c_state` as the fields `c_pc`, `c_locs` and `val_stack` while triples of all functions currently on the call stack are saved in `call_stack`.

In addition, three Boolean flags are present:

- `trapped` indicates that some instruction has resulted in a runtime error, corresponding to having generated a `Trap` instruction in a WebAssembly configuration.
- `return` is a special case that is present just before the current call stack frame returns. It has no practical use on the level of the pure interpreter, but will reconstruct slightly differently and is necessary to correctly represent the respective WebAssembly reductions.
- `terminated` is true when execution has finished. No further stepping is possible then and the final results may be taken from the value stack.

### 4.1.2 Addressing

The type `c_pc` of the program counter, as shown in figure 4.3, is slightly more involved than one might initially expect. First, the `c_fi` field defines the current function as an index into the store. `c_ei` is then an index into the code of this function, pointing to the next instruction to be executed. Finally, `c_d` is used to explicitly track artificial intermediate states of instructions that can only be fully reduced in multiple iterations, as described by (4.1). The `c_d` type is a datatype with multiple constructors representing specific kinds of states, and one constructor `D_None` for when no such state is used.

```
record c_pc =
  c_fi :: nat — closure (function) index in the store
  c_ei :: nat — index of the instruction inside the function
  c_d :: c_d — intermediate state info for instructions that take up multiple steps
```

Figure 4.3: Data structure representing a program counter for a single function invocation.

Since execution will very commonly advance a program counter `pc :: c_pc` in either `c_ei` or `c_d`, additional syntax is introduced for these cases:

- `pc ▶ 1` increases `c_ei` by 1 and resets `c_d` to `D_None`.
- `pc ∇ d` keeps `c_ei` and replaces `c_d` by `d`.

As WebAssembly programs with code nested in `Block`, `Loop` and `If` instructions form a tree structure rather than a simple list, addressing the contained instructions by the single natural number in `c_ei` is non-trivial. Figure 4.4 shows an example of how addresses are assigned in our implementation by flattening the tree structure. Notice that every closing bracket of every (nested) instruction list is also given its own address. Thus, the position directly after the last instruction inside of the `Block` is not equal to the position after the `Block` itself. These brackets may also be conceived as implicit "end" instructions, as sometimes found in WebAssembly code listings.

```
  [
0   C 42,
1   Block ([T_i32] _> [T_i32]) [
2     C 1,
3     Binop Add
4   ],
5   Unop Popcnt
6 ]
```

Figure 4.4: Addressing of instructions in a nested WebAssembly program by natural numbers. In addition to all explicit instructions, closing brackets are assigned addresses too.

To determine the instruction for a given address in a list of instructions, a function `fetch :: b_e list ⇒ nat ⇒ f_e option` is implemented. It returns `None` if the address was outside the boundaries of the given list, or an instance of `f_e` shown in figure 4.5. This may be either a concrete basic instruction or a special value signifying the end of some list, possibly a contained sub-list.

```
datatype f_e =
  FB b_e — regular fetched instruction
  | FEnd — exactly at the end of some Block/Loop/If
```

Figure 4.5: Fetched instruction type, either a concrete basic instruction or the end of a (nested) list.

### 4.1.3 Stepping

Execution on a state is implemented by the function `c_step :: c_state ⇒ c_state set`, shown in figure 4.6. The fact that the return type is a set of states rather than a single state already indicates that stepping is non-deterministic.

If any of the specific Boolean flags is set, it takes precedence and stepping branches off accordingly. If the program counter is currently at the end of some list inside the current function, it is advanced by the `c_step_out` function. Otherwise, the step is performed by `c_step_d`, taking into account both the fetched instruction `b` and the optional intermediate state taken from `c_d`.

If no explicit intermediate state is currently stored in `c_d`, this will eventually result in a step depending on `b`.

```
definition c_step :: c_state ⇒ c_state set where
 c_step state ≡
   if terminated state then      {}
   else if returning state then  { c_return state }
   else if trapped state then    c_step_trap state
   else
     (case s_fetch (c_s state) (c_pc state) of
       FB b ⇒                    c_step_d (c_d (c_pc state)) b state
     | FEnd ⇒                    { c_step_out state })
```

Figure 4.6: Root stepping function of the interpreter, branching off to individual cases depending on the current state and program code.

### Numeric Instructions and Trapping

As an example of the stepping for an instruction, the implementation of `Binop` is the following:

```
c_step_op (Binop t op) state =
  (case val_stack state of v2 # v1 # vs ⇒
    (case app_binop op v1 v2 of
      Some v ⇒
        { state⦇                                   (4.3)
            c_pc := c_pc state ▶ 1,
            val_stack := v # vs
          ⦈ }
    | None ⇒
        { state⦇                                   (4.4)
            trapped := True,
            c_pc := c_pc state ▶ 1,
            val_stack := vs
          ⦈ })
  | _ ⇒ {})
```

Here, (4.4) is the implementation of `Binop_Some` as seen previously in figure 3.3. Instead of consuming two preceding constant instructions in some instruction list, these values are taken directly from the stack and the binary operation is applied to them. If the operation is successful, the resulting value is pushed back to the stack and the program counter is advanced by one.

In case `app_binop` fails and returns `None`, the `trapped` flag is set. At this level, this also differs fundamentally from the reduction rule, where a `Trap` instruction is generated in-place:

$$\frac{\texttt{app\_binop op v1 v2 = None}}{⦇\texttt{[\$C v1, \$C v2, \$Binop t op]}⦈ \leadsto ⦇\texttt{[Trap]}⦈}\ \text{binop\_None} \tag{4.5}$$

### Constant Instructions

Stepping over a constant instruction `C v` is defined explicitly too, by simply pushing `v` to the stack:

```
definition c_step_const :: v ⇒ c_state ⇒ c_state where          (4.6)
  c_step_const v state ≡ state(|
    c_pc := c_pc state ▶ 1,
    val_stack := v # val_stack state
  |)

...

c_step_op (C v) state = { c_step_const v state }
```

Interestingly, unlike for all other instructions, there exists no reduction rule for constants. This is only sensible though, because instructions such as `Binop` automatically consume the constants preceding them as their operands. In our classical stack machine however, it becomes necessary as operands are taken from the explicit stack. This is the only case where the interpreter must perform more steps than reductions would do, representing (4.2).

### Artificial Intermediate States

Let us now see how `c_d` is used to introduce intermediate states to exactly correspond to single reductions, in order to avoid (4.1). One instruction where this is necessary is `Tee_local j`, which can be used to set the contents of a local variable to a value taken from the stack. But unlike `Set_local j`, it keeps this value on the stack, rather than removing it. In fact, its reduction rule is defined simply by duplicating the respective constant and then transforming into a `Set_local` instruction:

$$\frac{}{\text{(|[\$C v, \$Tee\_local i]|)} \rightsquigarrow \text{(|[\$C v, \$C v, \$Set\_local i]|)}}\ \text{tee\_local}$$

$$\frac{\text{length vi = j}}{\begin{array}{c}\text{(|s;(|f\_locs = (vi @ [v] @ vs), f\_inst = i|);[\$C v', \$Set\_local j)|)} \\ \rightsquigarrow \text{(|s;(|f\_locs = (vi @ [v'] @ vs), f\_inst = i|);[]|)}\end{array}}\ \text{set\_local}$$

To simulate this behavior on top of immutable code, the interpreter duplicates the topmost stack element and then descends into an intermediate `D_Set_local j` state:

```
c_step_op (Tee_local j) state =                                         (4.7)
  (case val_stack state of
    v # vs ⇒ {state⦇
      c_pc := c_pc state ∇ D_Set_local j,
      val_stack := v # v # vs
    ⦈ }
```

Finally, both `Set_local` and `D_Set_local` may share the same stepping code:

```
c_step_op (Set_local j) state = c_step_set_local j state                (4.8)
c_step_d (D_Set_local i) b state = c_step_set_local i state
```

### Non-determinism

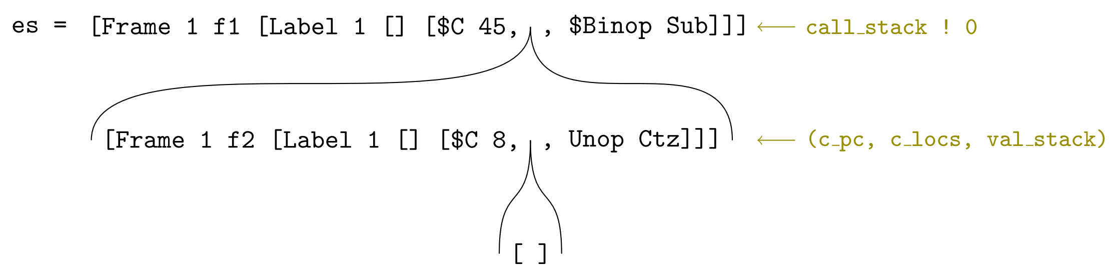
As elaborated previously, one of the primary goals of our interpreter is to cover fully all possible non-determinism allowed in WebAssembly. One example of where non-determinism is present is the `Grow_memory` instruction, which may succeed or fail, but WebAssembly does not put any restriction on when it is allowed to fail. It can be seen as similar to how `malloc` in C is always allowed to fail and return `NULL` any time. That is because the available amount of memory may differ significantly depending on the target device. The relevant part of the implementation in our interpreter is the following:

```
c_step_op Grow_memory state =                                           (4.9)
   ...
      (case mem_grow (mems (c_s state)!j) (nat_of_int c) of
        Some mem' ⇒ { state⦇ ... ⦈ } ...)
      ∪ { state⦇ — failure
            c_pc := c_pc state ▶ 1,
            val_stack := ConstInt32 int32_minus_one # vs
          ⦈ }
    ...
```

Notice how the union is always taken of the succeeded state and a failure state.

## 4.2 Reconstruction

We will now outline the idea of how a WebAssembly configuration ⦇`s;f0;es`⦈ is reconstructed from an interpreter state of type `c_state`. `s` is taken directly from the state's `c_s` field. The `f` of this configuration may in fact be chosen arbitrarily here, as the only reduction rule that may be applicable on our reconstructed configuration on



```
es = [Frame 1 f1 [Label 1 [] [$C 45, , $Binop Sub]]] ⟵ call_stack ! 0

      [Frame 1 f2 [Label 1 [] [$C 8, , Unop Ctz]]]   ⟵ (c_pc, c_locs, val_stack)

                                 [ ]
```

Figure 4.7: Illustration of nested `Frame` instructions reconstructed from the interpreter state's call stack. Function code is recursively filled at the current program counter with the next inner frame.

the outermost level will be `local`, shown in figure 3.3, which ignores the contents of `f0` entirely. This leaves us with the (administrative) instruction list `es`. In the Isabelle code, reconstruction of this is implemented by the function `c_es :: c_state ⇒ e list`. The algorithm that it uses is outlined below.

### 4.2.1 Call Frames

The instruction list `es` will contain any local information of all current call frames. The idea for how it is structured is illustrated in figure 4.7. Starting from the outermost triple on the call stack, `Frame` instructions are created recursively, containing the respective function code that has already been partially reduced according to the current state. In each such triple, the current program counter is exactly the point where the next inner function will return to when it has finished. Thus, this is the position where the next inner triple's `Frame` will be placed.

When the innermost frame has been reached, which is given by the triple `(c_pc state, c_locs state, val_stack state)`, the next filling list is simply an empty list in most cases, terminating the recursion. Only in the case when the state has trapped, a single `Trap` instruction will be filled in here, according to rules such as (4.5).

This part of the reconstruction is implemented by the function `c_es_frame :: ... ⇒ (c_pc × v list × v list) list ⇒ e list` by recursing over the given list of triples.

On a side note, WebAssembly defines every `Frame` to contain a single additional `Label` at the root with the actual code contained inside of it. This frame is only dissolved in the very last step just before the entire frame returns, unfolding its contained constants or `Trap` instruction. This special state is what the `returning` flag in our interpreter state is used to describe.

### 4.2.2 Local Code

With the high-level recursion of call frames in place, what is left is to develop an algorithm that, given an original, immutable function code, generates exactly the list of instructions that this code would have reduced to for a given program counter and value stack. The approach described in the following sections is implemented in the function `c_es_at :: ... ⇒ e list` in the Isabelle code.

### Pure Linear Code

For code without `Block`, `Label` and `If` instructions, this task is easy. Let us consider the following original function code:

```
    [                                   (4.10)
0     C 34,
1     C 13,
2     Binop Sub
3     C 2,
4     Binop Mul
5   ]
```

We assume that the `c_ei` part of our program counter points is 4, `c_d` is `D_None` and the the list describing the value stack is `[2, 21]`. The resulting reduced instruction list is generated by prepending the part of the program that is yet to be executed by the value stack converted to constant instructions in reverse order:

```
[                                       (4.11)
  $C 21,
  $C 2,
  $Binop Mul
]
```

Notice that, as indicated by the `$` prefixes, this list now contains administrative instructions rather than pure basic instructions as in the original code.

### Descending into Instructions

Going back to the `Tee_local` instruction from section 4.1.3, we have introduced the special `D_Set_local j` value in `c_d` to indicate its intermediate reduction state. Consider the following program code:

```
    [                                   (4.12)
0     C 5,
1     Tee_local 1
2     Nop
3   ]
```

Now, we assume that the value stack is currently `[5, 5]` and the program counter contains 1 in `c_ei` and `D_Set_local 1` in `c_d`. This would be the result after executing the respective steps defined in (4.6) and (4.7). The reconstruction then again takes the rest of the program starting at index 1, but replaces the first instruction inside it by `Set_local 1`, as specified by the current `c_d` value. The final result will then be the following:

(4.13)

```
[
  $C 5,
  $C 5,
  $Set_local 1
  $Nop
]
```

## Labels

As soon as `Block`, `Loop` and `If` instructions are present in the program, reconstruction becomes significantly more involved. The abstract idea of combining the rest of the program starting at the program counter with the value stack contents is still being followed. The difference now is that the program counter may point inside a nested instruction list as part of one of these control flow instructions and thus the entire list can not simply be split at this position anymore.

Instead, in accordance with the reduction relation, the control flow instructions forming the chain that the program counter currently points into are transformed into appropriate `Label`s. Ignoring the value stack for a moment, the contents of these labels are then simply the rest of the respective sub-lists at the respective positions. For example, the skeleton for reconstructing the following program at program counter 4 would look like this:

(4.14)

```
  [
0   C 13,
1   C 34,
2   Block ([T_i32] _> [T_i32]) [
3     Get_local 0
4     Binop Add
5   ],
6   Drop
7 ]
```

(4.15)

```
[
  ?

  Label 1 [] [
    ?
    $Binop Add
  ],
  $Drop
]
```

Finally, the question arises how to distribute the value stack over these labels correctly to fill the indicated gaps. The `block` rule from figure 3.3 dictates that when when a `Block` is entered and transformed into a `Label`, exactly `n` labels are moved from before the `Block` into the label, where `n` is the number of arguments given by the `Block`'s type. This information alone however is not sufficient because it only provides the number of constants inside the label directly after it has been entered, but the state may have already stepped further, such that the number could have changed. In our example, the `Get_local` instruction would have raised this number from 1 to 2.

The solution is to walk over every single instruction from the function start to the program counter and keep track of the number `vs_n` of stack values that are expected in the current label. At the point where a control flow instruction is descended into, this value is split up according to the given type, resulting in the final number of constants that should be taken from the tail of the stack list and put directly before the generated label, as well as the `vs_n` value to be tracked further into the inner list. When the final program counter is hit, `vs_n` itself is the number of constants to put there.

This kind of tracking in our program (4.14) would yield the following results:

```
  [  vs_n = 0                                                            (4.16)
0   C 13,  +1  ⇒  vs_n = 1
1   C 34,  +1  ⇒  vs_n = 2
2   Block ([T_i32] _> [T_i32]) [  1 arg  ⇒  outside = vs_n - 1 = 1; vs_n = 1
3     Get_local 0  +1  ⇒  vs_n = 2
4     Binop Add
5   ],
6   Drop
7 ]
```

From this analysis, we have determined that, starting from the tail of the stack, one value should be put before the label and two inside of it. Now assuming the value stack at the program counter 4 is `[0, 34, 13]`, we get the following reconstruction by completing the skeleton from (4.15):

```
[                                                                        (4.17)
  $C 13
  Label 1 [] [
    $C 34
    $C 0
    $Binop Add
  ],
  $Drop
]
```

Following the respective reduction rules, this is indeed the correct result.

## 4.3 Skipping over Constants

In section 4.0.1, we have discussed two issues that may prevent a one-to-one correspondence between interpreter steps and reductions. While (4.1) has already been solved by how the interpreter is implemented, (4.2) is still open.

The only case where multiple iterations of `c_step` are necessary to replicate a single reduction is stepping over constant instructions as elaborated in section 4.1.3. For example, let us consider the following program, along with its reconstruction for an empty value stack at program counter 0:

```
  [
0   C 32,
1   C 10,
2   Binop Add
3 ]
```
(4.18)

```
[
  $C 32,
  $C 10,
  $Binop Add
]
```
(4.19)

Now we let our interpreter take two steps on this state. The program counter will be 2 and the value stack will be `[10, 32]`. When this is reconstructed, we still get the exact same result as above. In contrast, reduction will automatically detect the `Binop` instruction with its two preceding constant instructions and execute the addition in one step. This prevents us from stating that any reduction on a reconstruction is reproduced by a single application of `c_step`.

We solve this issue by introducing a new stepping function that does not share this property. In particular, the new function `c_step_nc`, shown in figure 4.8, simply steps over all available constants immediately before applying one final `c_step`.

With the amount of constants to be stepped over before reaching a different instruction or the end of the program being always finite, properties on `c_step` can be lifted to `c_step_nc` by induction on this list of constants.

```
function c_step_nc :: c_state ⇒ c_state set where
 c_step_nc state = (
   if c_at_const state then
     case s_fetch (c_s state) (c_pc state) of (FB (C v)) ⇒
       c_step_nc (c_step_const v state)
   else
     c_step state)
```

Figure 4.8: Extended stepping function to skip over any available constants. In contrast to `c_state`, this behavior always corresponds exactly to applying a single reduction rule.

## 4.4 Completeness Proof

We now want to prove that our interpreter is able to replicate any possible sequence of reductions on the reconstruction of a given state, as previously illustrated in figure 4.1. Formally, for a given `state :: c_state`, we assume:

$$\text{(|c\_s state; f0; c\_es state|)} \rightsquigarrow \text{(|reduced\_s; f1; reduced\_es|)} \tag{4.20}$$

Having `c_step_nc`, we are able to also formally state the goal that we eventually want to prove:

```
∃state'. c_s state' = reduced_s ∧                                    (4.21)
         c_es state' = reduced_es ∧
         state' ∈ c_step_nc state
```

In words: there is some `state' :: c_state` that can be reached by `c_step_nc` and also has the same reconstruction as given by the assumed reduction.

### 4.4.1 Induction over WebAssembly Programs

Very frequently, we will have to show a property for either an entire list of instructions or for a program counter into such a list. This is often only possible by induction over the structure of the program, showing first that the property holds for example for the empty program and extending this to arbitrary programs. Because of the tree-like structure of WebAssembly, classic structural induction over neither the list nor the `b_e` type suffices.

Instead, we introduce multiple dedicated induction rules for these cases. They slightly differ in their purpose but the overall structure of all of them can be informally described as the following:

- If `P` holds for program counter 0 in `bs :: b_e list`
- and
  - assuming `P` holds for program counter `pc` in `bbs :: b_e list`,
  - `P` also holds for the program counter that points to the same relative position in `bbs` in any of
    * `[Block bbs, ...]`,
    * `[Loop bbs, ...]`,
    * `[If bbs _, ...]`
    * and `[If _ bbs, ...]`

- and
  - assuming `P` holds for program counter `pc` in `bs :: b_e list`,
  - `P` also holds for the program counter that points to the same relative position in `bs` in any `[b] @ bs`
- then `P` holds for any program counter `pc` in any `bs :: b_e list`.

### 4.4.2 Local Completeness

As a basis for proving completeness on arbitrary states, we first restrict ourselves to a single, local call frame. In particular, we make the following assumptions on a given `state :: c_state`:

1. `state` has not terminated yet.
2. The call stack is empty.

#### Cause of the Reduction

We want to learn more about the contents of `state`. Apart from our local restrictions, this depends entirely on the reduction rule and premises that initially induced (4.20). Thus, rule inversion on the inductively defined reduction relation is the approach to take.

Unfortunately, due to the behavior described in section 3.2, the derivation that caused the reduction may contain arbitrarily long chains of applications of the `label` rule shown in figure 3.3. This is rather inconvenient since rule inversion of the reduction relation in its current form would strictly require a possibly complex induction to be able to fully reason about any possible derivation.

In order to avoid this and ease our proof work, we define an alternative reduction relation `⦇s;f;es⦈ ↝| ⦇s;f;es'⦈`, which is defined exactly in the same way as the original one, but with the `label` rule removed. The leftover rules are all purely syntax-directed and thus provide a good basis for further reasoning about the structure of their inputs. Apart from the `local` rule considering the contents of a `Frame` instruction, it also contains no recursion anymore and we thus call it "atomic reduction".

We are then able to prove by a relatively simple induction over the original reduction relation that any reduction may also be traced back in the form of a single atomic reduction lifted to the full context by `Lfilled`. This lifting could be seen also as a single application of the `label` rule:

$$\frac{\texttt{(|s;f;les|)} \leadsto \texttt{(|s';f';reduced\_les|)}}{\begin{array}{l}\exists\texttt{es es' k lholed. (|s;f;es|)} \leadsto\texttt{| (|s';f';es'|)} \land \\ \qquad\qquad\texttt{Lfilled k lholed es les} \land \\ \qquad\qquad\texttt{Lfilled k lholed es' reduced\_les}\end{array}}\ \text{reduce\_atomic} \tag{4.22}$$

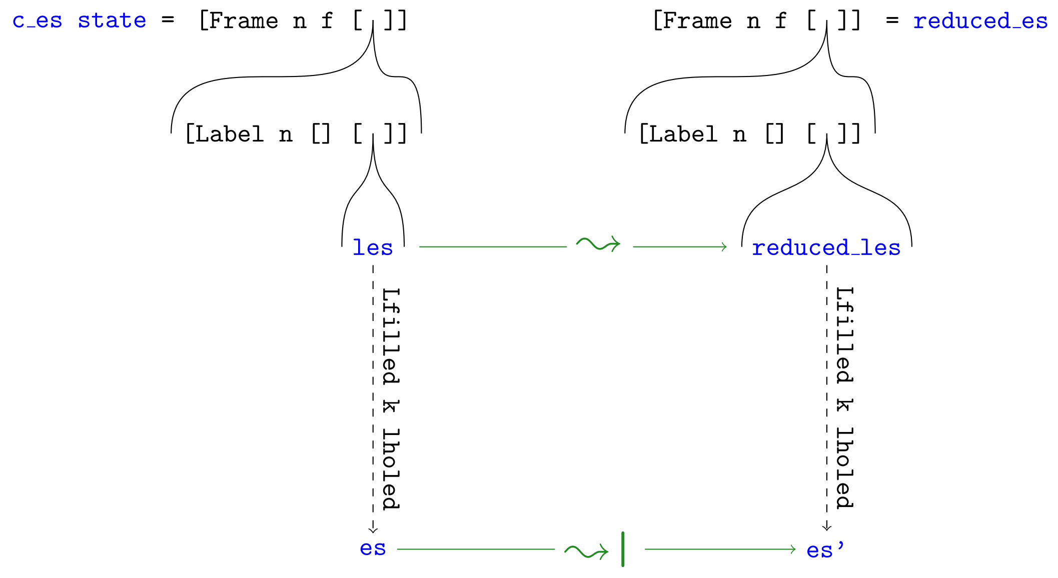


Figure 4.9: Illustration of the relationship between current and final instruction lists, as well as the reductions established for them.

Note that `les` and `reduced_les` are not equal to `c_es state` and `reduced_es`, but contained inside of them, respectively, with a `Frame` and `Label` instruction wrapped around. This is a consequence of the WebAssembly frame structure and the relevant relationships are illustrated in figure 4.9. However, each of them is uniquely defined by their wrapped or unwrapped counterpart.

With this atomic reduction being available, a rule inversion without induction on it gives us meaningful results. For example, from the inverted `binop_Some` rule, we obtain constants `v1`, `v2` and `v`, as well as a binary operation type `op` such that:

```
es = [$C v1, $C v2, $Binop op]                                  (4.23)
app_binop op v1 v2 = Some v
es' = [$C v]
```

Since `les` from (4.22) in our case was the list that our own reconstruction algorithm has generated from `state`, we can derive important properties that `state` must have had in order for `les` to be `Lfilled` in the sense of `Lfilled k lholed es les`.

Specifically, from `Lfilled k lholed es les` and `es' = [$C v]`, we are able to show by induction over the code of the current function in `state` that, after any constants have been skipped by `c_step_nc`:

- `[v2, v1]` is a prefix of `val_stack state`. (4.24)
- `c_pc state` points exactly to a `Binop op` instruction.

### Equality of the Final State

With this amount of information, we can fully evaluate `c_step_nc state` and obtain a `state'` that already fulfills the `state'` $\in$ `c_step_nc state` goal from (4.21). This leaves `c_s state' = reduced_s` and `c_es state' = reduced_es` to prove.

Due to how all reduction rules are specified, the structure of `reduced_s` is given already by the inversion of the atomic reduction. The same can be said about `c_s state'` because of how our stepping is implemented. Thus, `c_s state' = reduced_s` may generally be shown for all individual cases by simplification and similarly trivial methods.

For showing that `c_es state' = reduced_es`, we show that `stepped_les = reduced_les` where `stepped_les` denotes the appropriate reconstructed sub-list of `c_es state`. At this point, the only thing we know about `reduced_les` is `Lfilled k lholed es' reduced_les` from (4.22).

From the deterministic nature of `Lfilled` expressed in (3.4), we do know however that if for any other `xes`, `Lfilled k lholed es' xes` holds, then `xes = reduced_les`. Thus, we approach the problem by trying to show `Lfilled k lholed es' stepped_les`. Notice that these `k` and `lholed` are the same that are used in `Lfilled k lholed es les`. Both `les` and `stepped_les` are results of reconstruction on `state` and `state'`, respectively, with the only differences between the two states being known from how we evaluated `c_step_nc state`. By using this knowledge and `Lfilled k lholed es les` as a premise, we are able to prove `Lfilled k lholed es' stepped_les` for all individual, possible reduction cases by induction over the function code.

With this aspect being shown, we have proven completeness, given the restriction to an empty call stack.

### 4.4.3 Completeness for Arbitrary Call Stacks

We can now extend our findings to states with arbitrarily long call stacks. This is done by induction over the `call_stack state` list by starting with the empty list and appending a single element at the tail in every step. In WebAssembly terms, this corresponds to taking the current execution state of a program and putting it in a larger context, as if it was called from another function, rather than having been invoked directly. For the induction, we can use our completeness theorem for the empty call stack as the base case.

In the induction step, we may then assume that completeness holds for any state with a specific call stack `cs` and then show that it also holds for a state with call stack `cs @ [(pc, locs, vs)]`. So, we are given the following premises in order to show (4.21):

```
call_stack state = cs @ [(pc, locs, vs)]                                  (4.25)
(|c_s state; f0; c_es state|) ⇝ (|reduced_s; f1; reduced_es|)            (4.26)
```

From the way our reconstruction illustrated in figure 4.7 works, we know that a `Frame` instruction representing the entire call stack up to the end of `cs` is contained within `c_es state`. Let `istate` now denote `state(|call_stack := cs|)`. In fact, this contained `Frame` is now exactly the reconstruction `c_es istate`.

This information about the structure of `c_es state` lets us trace back the reduction that happened in (4.26) to the reduction of this inner `Frame` by ruling out any derivation that could not possibly match this structure. By doing so, we obtain `f'` and `reduced_ies` such that:

$$\texttt{(|c\_s istate; f; c\_es istate|)} \leadsto \texttt{(|reduced\_s; f'; reduced\_ies|)} \tag{4.27}$$

where `f` is derived from `(pc, locs, vs)` and `reduced_ies` is contained within `reduced_es` the same way that `c_es istate` is contained within `c_es state`. We now notice that `istate`'s call stack in combination with (4.27) perfectly matches our induction hypothesis, which is the completeness of states with call stack `cs`, so we can obtain a stepped state `istate'` to match `reduced_ies`:

```
 c_s istate' = reduced_s                                    (4.28)
c_es istate' = reduced_ies
     istate' ∈ c_step_nc istate
```

We then analyze all possible ways that `c_step_nc state` and `c_step_nc istate` could have been evalued. Because of the fact that `state` and `istate` differ only in their call stack, we find out that we can always obtain a `state'` such that:

```
c_s state' = reduced_s                                      (4.29)
    state' ∈ c_step_nc state
```

and in addition `c_es istate'` is contained within `c_es state'` the same way that `c_es istate` is contained in `c_es state`. We previously noticed that this exact property is also shared by `reduced_ies = c_es istate'` within `reduced_es`. Thus, again making use of the determinism of `Lfilled` (which established these relationships), we are able to prove that `reduced_es = c_es state`.

Finally, combining this with (4.29), completeness in the induction step is finished by stating:

```
 c_s state' = reduced_s                                     (4.30)
c_es state' = reduced_es
     state' ∈ c_step_nc state
```

Thus, as a result of the entire induction, completeness has been shown for arbitrary states. □

## 4.5 Caveats

Inside the above proofs in Isabelle, there are various cases where the behavior of how many elements are on the stack have to follow certain rules. For example, when a function returns to its outer frame, its return values will be pushed to the outer stack. Then, in the reconstruction, these return values must end up in exactly the correct spot where the inner function was initially invoked. This however can only guaranteed if the stack has the correct size, such that the constants will be distributed over all `Labels` in the expected manner.

In order to guarantee these properties, the interpreter tracks some invariants that are proven to be preserved by stepping, such as that the stack size always matches the expected number at the current program counter. It makes use of typing, as introduced in section 3.3, which guarantees for example that a stack underflow will never happen, to carry out these proofs.

In its current form however, it only reasons about the pure number of elements on the stack, but not their individual types. As a consequence, there are two specific cases where invariant preservation in its original form could not be shown:

1. Host functions, i.e. external functions that may be called from inside WebAssembly, only have a well-defined stack behavior if all argument types match exactly their specified type.
2. The `Set_global` instruction can break well-typedness of a store if its operand value has an incorrect type.

As host functions in the mechanization are defined by axioms on an uninterpreted constant, we have added a respective axiom that guarantees the expected stack behavior also for only the premise of being given the correct number of arguments, independently of their types.

For the `Set_global` instruction, a runtime type check was introduced in the respective reduction rule and the interpreter, such that no steps breaking well-typedness can be performed anymore.

Since we are only concerned with programs that are fully well-typed, both additions however are not necessary in theory, as the typing relation already provides the appropriate preconditions. Proving this is plausible, will however require significant additional effort, since all individual types for the entire call stack will have to be tracked as part of the invariant, rather than only their number.

# 5 Conclusion

## 5.1 Breakdown

The entire project for this thesis, which was developed over a course of six months, weighs in at about 19000 lines of Isabelle code. Splitting it up by individual files in the session, the distribution of this amount over different aspects of the project is the shown in figure 5.1. It indicates that the most significant portion of work was spent on the completeness proof, along with any definitions and properties that are necessary for it, such as reconstruction.

## 5.2 Results

By specifying and implementing integer operations, we have added one of the last missing pieces to the WebAssembly formalization from Watt [Wat18] to provide a mechanization of the entire WebAssembly standard. Our work on this has been merged into the official repository as of commit 2e2fa63 [MW21]. It is now used as part of the sound WebAssembly Interpreter inside this repository and has replaced the previous, unverified OCaml variant.

Our theory `Sshiftr.thy` implements various auxiliary lemmas about the right shift of words interpreted as signed numbers. Analogous lemmas for shifts of unsigned numbers had already been available as part of the official Word Library [Bee+16], but these specific ones had still been missing. Our lemmas have been merged into the development version of the Archive of Formal Proofs, which hosts the library, in commit 556e4a005c15 [KM21].

Finally, we have developed an interpreter for the WebAssembly language that is implemented as a classic stack machine and have proven that it can replicate any execution allowed by the WebAssembly specification on a well-typed program. This is not only of high significance for verified analysis of WebAssembly, but also shows that this concept, which is the common approach used in real-world interpreters such as Fizzy [BMB], indeed can cover the entire formal WebAssembly specification.

## 5.3 Future Work

With integer operations being finished, the only other missing piece of the WebAssembly mechanization are float types and operations. These are represented in WebAssembly

| **Auxiliaries** | | |
|---|---|---|
| 38 lines | `List_All_Pairs.thy` | Predicate on lists necessary for interpreter invariants |
| 1170 lines | `Wasm_Properties2.thy` | Various added properties of WebAssembly-specifics |
| **Integer Operations** | | |
| 133 lines | `Sshiftr.thy` | Lemmas for signed right shifts previously missing from the Word Library |
| 108 lines | `Power_Sum.thy` | Generic lemmas about sums of powers, such as the base two representation of a number |
| +469 lines | `Wasm_Type_Abs.thy` | Formalization of the WebAssembly integer specifications |
| 779 lines | `Wasm_Type_Word.thy` | Generic implementation of WebAssembly integers using the Word Library |
| **Interpreter** | | |
| 1174 lines | `Base.thy` | Elementary definitions for all parts of the interpreter |
| 470 lines | `Interpreter.thy` | Interpreter state and stepping |
| 2019 lines | `Interpreter_Properties.thy` | General properties of the interpreter |
| 1684 lines | `Stack.thy` | Stack behavior and properties concluded from typing |
| 347 lines | `Descend.thy` | Properties of the interpreter for `c_d` $\neq$ `D_None` |
| 210 lines | `Skip.thy` | Skipping over entire instructions, for trapping behavior |
| 2411 lines | `State_Valid.thy` | Invariants and their preservation |
| 2143 lines | `Reconstruct.thy` | Reconstruction from interpreter state to administrative instruction lists |
| 946 lines | `Step_NC.thy` | Stepping function `c_step_nc` with one-to-one correspondence to reduction |
| 110 lines | `Reduce_Atomic.thy` | Reduction relation with `label` rule removed |
| 4912 lines | `Completeness_Local.thy` | Completeness for an empty call stack |
| 401 lines | `Completeness.thy` | Completeness for arbitrary call stacks |

Figure 5.1: Breakdown of the developed theory files with their lengths and content description.

by the two types `f32` and `f64`, which both share most of the same specification, similarly to `i32` and `i64`. Thus, the most natural way to implement them would be to take the same bit width-agnostic approach that we used for integers.

Our interpreter opens up many possibilities in the sense of analysis. For example, a generic Abstract Interpretation framework could now be implemented on top of it, for the various use-cases described in chapter 1. In particular, one possible practical application would be to develop a framework similar to Binaryen [Web21a], but fully verified. Inside of it, the Abstract Interpreter would be able to overapproximate all possible values of variables. Based on this information, for example, `If` instructions whose conditions are guaranteed to evaluate to either True or False, could be simplified to a single block. This could replicate Binaryen's DeadCodeElimination pass. The resulting tool may then be used as a reliable optimizer of WebAssembly code before shipping.

In addition, as a counterpart to completeness, a proof for soundness of the interpreter could be approached. The specific goal to prove would be that every step the interpreter can take is also covered by a WebAssembly reduction rule. One could make use of the existing reconstruction algorithms for this. A convenient aspect is that the way the interpreter performs its stepping is defined as concrete equations rather than an inductive predicate. Thus, no complex rule inversions and inductions on the reduction rule should be necessary for this. Once the soundness proof is finished, in combination with the completeness proof, it would be shown that the interpreter does not only underapproximate or overapproximate WebAssembly, but implements it exactly.

Moreover, a soundness proof is also the necessary component to derive a verified, executable interpreter for real-world applications. Such an interpreter could be easily created by picking a single case from every non-deterministic step in our interpreter, and only executing that. Thus, the interpreter would be fully deterministic. One major advantage of such an interpreter based on our stack- and program counter-based approach over for example the verified interpreter by Watt [Wat18] is that no additional memory is needed to track the current reduction state of the program. In the extreme case, it could even be possible to load a raw WebAssembly binary and let the program counter point directly into this binary, as code is being treated as immutable. It is thus a good basis for further refinement towards highly-optimized WebAssembly interpretation.

# List of Figures

# Bibliography

[Bee+16] J. Beeren, M. Fernandez, X. Gao, G. Klein, R. Kolanski, J. Lim, C. Lewis, D. Matichuk, and T. Sewell. “Finite Machine Word Library.” In: *Arch. Formal Proofs* 2016 (2016). URL: `https://www.isa-afp.org/entries/Word%5C_Lib.shtml`.

[BMB] A. Beregszaszi, A. Maiboroda, and P. Bylica. *Fizzy*. URL: `https://github.com/wasmx/fizzy` (visited on 2021-09-07).

[Byl18] S. Bylokhov. *Release Note: Removal of appletviewer Launcher*. 2018-04-23. URL: `https://bugs.openjdk.java.net/browse/JDK-8202161` (visited on 2021-08-04).

[CC77] P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.” In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by R. M. Graham, M. A. Harrison, and R. Sethi. ACM, 1977, pp. 238–252. DOI: `10.1145/512950.512973`. URL: `https://doi.org/10.1145/512950.512973`.

[Haa+17] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien. “Bringing the web up to speed with WebAssembly.” In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by A. Cohen and M. T. Vechev. ACM, 2017, pp. 185–200. DOI: `10.1145/3062341.3062363`. URL: `https://doi.org/10.1145/3062341.3062363`.

[HWZ] D. Herman, L. Wagner, and A. Zakai. *asm.js Working Draft*. URL: `http://asmjs.org/spec/latest` (visited on 2021-08-04).

[KM21] G. Klein and F. Märkl. *afp-devel commit 556e4a005c15: sshiftr/bl lemmas by Florian Märkl*. 2021-04-19. URL: `https://foss.heptapod.net/isa-afp/afp-devel/-/commit/6b53a6d8121ef1088de9668d98061fb500e915e5` (visited on 2021-09-07).

[LA04] C. Lattner and V. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation.” In: San Jose, CA, USA, 2004-03, pp. 75–88.

[Lin+15] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java® Virtual Machine Specification*. 2015-02-13. URL: https://docs.oracle.com/javase/specs/jvms/se8/html/index.html (visited on 2021-08-10).

[MW21] F. Märkl and C. Watt. *WasmCert-Isabelle commit 2e2fa63: Specification and Implementation for Integer Ops*. 2021-06-11. URL: https://github.com/WasmCert/WasmCert-Isabelle/commit/2e2fa6358f01d7d87c4c8858b67de35b00e60077 (visited on 2021-09-07).

[Net95] Netscape. *Netscape and Sun Announce JavaScript, the Open, Cross-platform Object Scripting Language for Enterprise Networks and the Internet*. http://wp.netscape.com/newsref/pr/newsrelease67.html archived by the Internet Archive on 2007-09-16. 1995-12-04. URL: https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html (visited on 2021-08-04).

[NK14] T. Nipkow and G. Klein. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014. ISBN: 978-3-319-10541-3. DOI: 10.1007/978-3-319-10542-0. URL: http://concrete-semantics.org/.

[Ric53] H. G. Rice. "Classes of recursively enumerable sets and their decision problems." In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366.

[Tea17] A. C. Team. *Flash & the Future of Interactive Content*. 2017-07-25. URL: https://blog.adobe.com/en/publish/2017/07/25/adobe-flash-update.html (visited on 2021-08-04).

[The80] The Forth Standards Team. *Forth-79 Standard*. 1980-10.

[Tur14] J. Turner. *Announcing TypeScript 1.0*. 2014-04-02. URL: https://devblogs.microsoft.com/typescript/announcing-typescript-1-0 (visited on 2021-08-04).

[Wat18] C. Watt. "Mechanising and verifying the WebAssembly specification." In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*. Ed. by J. Andronick and A. P. Felty. ACM, 2018, pp. 53–65. DOI: 10.1145/3167082. URL: https://doi.org/10.1145/3167082.

[Web21a] WebAssembly Community Group. *Binaryen*. 2021. URL: https://github.com/WebAssembly/binaryen (visited on 2021-08-09).

[Web21b] WebAssembly Community Group. *WebAssembly Specification Release 1.1*. Ed. by A. Rossberg. 2021-08-03. URL: https://webassembly.github.io/spec/core/ (visited on 2021-08-10).

[Wor14] World Wide Web Consortium. *HTML5, A vocabulary and associated APIs for HTML and XHTML*. 2014. URL: `https://www.w3.org/TR/html5/` (visited on 2021-08-04).

[Zak11] A. Zakai. "Emscripten: an LLVM-to-JavaScript compiler." In: *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. Ed. by C. V. Lopes and K. Fisher. ACM, 2011, pp. 301–312. DOI: `10.1145/2048147.2048224`. URL: `https://doi.org/10.1145/2048147.2048224`.