HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN
MÜNCHEN

FAKULTÄT FÜR INFORMATIK UND MATHEMATIK

Masterarbeit
zur Erlangung des Grades
Master of Science
im Studiengang Informatik

Analyzing a possible implementation of Bounded Model Checking for WebAssembly programs

Analyse einer möglichen Implementierung von Bounded Model Checking für WebAssembly-Programme

---

Autor:	Maximilian Reichl
Matrikelnummer:	29437222
Abgabedatum:	19. Dezember 2024
Betreuer:	Prof. Dr. Matthias Güdemann

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

München, 19. Dezember 2024 Maximilian Reichl

Contents

1 Introduction

In our modern world, software and technology are more present than ever in our lives. As software complexity grows, so does the potential for errors. Such errors can lead to severe consequences, particularly in safety-critical domains. Finding errors is a hard task and this is done in most cases through rigorous testing.

A different way of finding errors in systems is offered by a method called **bounded model checking**. This is a technique from the field of formal verification. Many formal verification methods try to prove using formal techniques that a system is correct or at least adheres to a formal specification.

Bounded model checking takes the reverse approach: It tries to prove that errors exist by finding inputs that lead to defined erroneous states.

The advantages of bounded model checking over testing are best shown with an example. Listing 1 shows a function written in the C language that takes the absolute value of an integer. The absolute value is the non-negative value of an integer without the sign.

```
1  int absoluteValue(int number) {
2      int returnValue;
3      if (number >= 0)
4          returnValue = number;
5      else
6          returnValue = number * (-1);
7
8      return returnValue;
9  }
```

Listing 1: *Absolute value function*

This code seems very sensible, but to make sure it works as intended we can write a unit test for it. Listing 2 shows how this unit test would look like, with two test cases covering each branch of the logic.

```
1  int testAbsoluteValue() {
2      assert(absoluteValue(7) == 7);
3      assert(absoluteValue(-3) == 3);
4  }
```

Listing 2: *Unit test for the absoluteValue function*

With the tests passing we are pretty sure that our function is error-free. We deploy the code in production and the application works. After some time, an error ticket got raised because the application broke down. Reviewing the logging output, we apparently see that the result of our function was a negative number, that caused trouble in further processing. After all, we did not

expect that this function returns a negative number. The result we see in our logging output is the number −2147483648.

The reason that has happened is that in C signed integer values are represented using two's complement, where the first bit indicates the sign. [1] The possible values that an int can have in C is ranging from −2147483648 to 2147483647, which means that there is one more negative value than positive values. The reason for this is that 0 has the sign bit not set and counts therefore as a positive number.

When our function is called with the input −2147483648, it tries to calculate $-2147483648 \cdot (-1)$, which is 2147483648. This number is *outside* the range of an int, so the program takes its best guess at representing this value, which results in −2147483648, a negative number. This could have had worse consequences, in C this failed calculation actually triggers undefined behavior [12, §6.5 5], which means that anything could have happened. This seems like a very rare case that will almost never happen with random input, but it could be that the input is dependent on some user action and there are malicous actors trying to compromise the system.

This shows how easy it is to create code with an undetected error in it that even slipped through testing. Such a type of error could have been found with bounded model checking. To demonstrate I will use a special program called CBMC, the C Bounded Model Checker. To check our function we need to define an erroneous state. This can be done with an assertion, as shown in line 8 of listing 3.

```
1  int absoluteValue(int number) {
2      int returnValue;
3      if (number >= 0)
4          returnValue = number;
5      else
6          returnValue = number * (-1);
7
8      assert(returnValue >= 0);
9      return returnValue;
10 }
```

Listing 3: *Erroreous state definition*

When we run CBMC on this code, it will show us that the assertion is violated and gives us immediately the input leading to that violation, in this case the input of −2147483648.

There exist bounded model checking tools for many languages, most notably CBMC for C and C++, JBMC for Java, ESBMC for C, C++, Python and Solidity and Kani for Rust.

A technology that has gained a lot of traction in recent years is WebAssembly. WebAssembly is a bytecode format that allows code written in many different languages to run on the web alongside JavaScript. The motivation behind this is that as the web becomes more and more targeted by application developers, JavaScript is not designed to run high-performance code. Whereas WebAssembly code can be executed at near-native speed, which expands the possibili-

[1] The representation of signed integers in C is actually implementation-defined, but for simplicity we assume it is two's complement. We also assume that on the particular system an int is 32 bits wide.

ties of web applications. It also allows developers to port their native applications to the web easily. Another advantage is that WebAssembly code to be run outside of the web in standalone runtimes, which makes WebAssembly able to run applications cross-platform at near-native execution speed.

In this thesis I am exploring the possibility of building a bounded model checking tool that targets WebAssembly bytecode. For this, I will use the existing toolchain from CBMC as a framework and find out what needs to done to extend it with a frontend for WebAssembly.

2 Background

2.1 WebAssembly

WebAssembly, or Wasm in short, is a standardized bytecode format. Its intended use is to run alongside JavaScript in the web, but because of its open design it can be used in other contexts.

The idea is that WebAssembly acts as a compilation target for fast languages such as C or Rust. If a web application needs near-native execution speed for a part of the program, then this part can be written in another language and compiled to WebAssembly.

Wasm code gets released as modules. Unlike for example Java, whose bytecode format shares similar ideas, Wasm modules are not intened to be standalone programs. Every Wasm module needs a host environment in which it runs, which is in most cases a web browser, but it can also run outside of the browser using Node.js or one of many available third-party runtimes. Wasm can exchange data and functionality with the host environment through the use of imports and exports. Therefore, there is no "standard library", any additional functionality needs to be provided by the host environment. With the increasing use of Wasm outside of web environments, there was a need to standardize common OS interaction such as input/output and filesystem access. This standard is known as WebAssembly System Interface (WASI). [1]

Wasm code is a bytecode that means it is a binary format containing instructions and data, similar to machine code. The big difference to machine code is that machine code only runs on the one CPU architecture it was compiled to, while bytecode is designed to be runnable on most systems. This is possible because bytecode does not target a physical CPU, it targets a virtual CPU. The running environment of the Wasm module simulates that virtual CPU while interpreting the Wasm bytecode. Usually to increase execution speed the Wasm module gets just-in-time-compiled to native machine code.

2.2 Program Verification

2.2.1 Introduction

The process to prove that a program works correctly is called **formal verification**. The concept of constructing a proof is borrowed from math, where mathematicians have been proving theorems for hundreds of years. In math, theorems are proven by logically deducting them from other theorems or axioms, which are postulates that defined to be true. There are theorems, that have been tested to be true for billions of numbers. But without a logical proof, it can only be assumed that the theorem is correct, since there might a number that has not been tried yet which would disprove the statement.

But how can we prove that a program is correct? How do we define what "correct" means? Intuitively, we can say that a program is correct, if it follows its specification. Usually the specification of a program is the sum of its requirements. One of the approaches to formal verification is to precisely define these requirements into a formal specification. Then we can

are able to prove it using mathematical and logical techniques. This can be done manually, or even automatically using programs called provers.

2.2.2 Model Checking

One of the approaches to automate verification is **model checking**. This got introduced by the works of E. M. Clarke and E. A. Emerson in the 1980s [6]. In model checking a system is modeled using a finite state machine. A finite state machine is a representation of a system as a graph, where each node represents one possible state. The edges between nodes represent allowed transitions, these are operations that lead to the program changing its state. The specification is written in a formal language called computation tree logic, which is a form of temporal logic. This is helpful when dealing with concurrent protocols that have a low number of defined states.

Earlier types of model checking focused on hardware protocols. Software verification was often unfeasible, due to even simple programs reaching exponentially high number of states. This is known as state explosion. This got improved later with **symbolic model checking** [4]. In symbolic model checking state machines are modeled with binary decision diagrams (BDDs). Modeling with BDDs increased the realistic number of states that can be checked significantly.

2.2.3 SAT solvers

To draw the bridge from symbolic model checking to bounded model checking, we first need to introduce SAT solvers. The boolean satisfiability problem, often called SAT, is the problem of whether or not a boolean formula is *satisfiable*. This means proving that for a boolean formula there are inputs that make the whole expression true, or if not, proving that such inputs do not exist.

A boolean formula is a formula where each variable can only have two possible states, true and false. Operations on boolean variables include negation ($\neg$), conjunction ($\wedge$, $a \wedge b$ evaluates true only if a AND b are both true) and disjunction ($\vee$, $a \vee b$ evaluates true if one of a OR b are true, or both).

This is an example formula that is satisfiable:

$$(a \vee b) \wedge (a \vee \neg b) \wedge \neg b \tag{2.1}$$

It is easy to spot that the formula is satisfiable if a is true and b is false.

Now another example:

$$a \wedge b \wedge (\neg a \vee \neg b) \tag{2.2}$$

This formula is not satisfiable. With two variables, this seems easy to solve but for bigger expressions using more variables, we can rely to solve. These programs are called SAT solvers and can solve formulas with hundreds of variables.

SAT solvers help with verification because we can theoretically reduce every program into a boolean formula. This works by thinking about data that a computer processes as a number of boolean variables, one for each bit. But computers can not only perform standard boolean

operations on data, they can do things like arithmetic. It is indeed possible to reduce complex arithmetic operations down to standard boolean operations, if it would not be possible, no computer would work, since electronic circuits cannot "calculate", they can only perform boolean operations using transistors.

As an example I show how addition can be translated into a boolean formula using a 1-bit half adder. This adds two 1-bit numbers together. Our 1-bit input numbers are represented in binary and can either be 1 or 0. 0 plus 0 equals 0, 0 plus 1 or 1 plus 0 equals 1, and 1 plus 1 equals 2 in decimal, which is 10 in binary. The last digit of 10 can be thought of as the result, and the 1 may be used as a carry if we would add longer numbers together. This gives us the table shown in table 2.1.

Input 1	Input 2	Result	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 2.1: *Result table for 1-bit addition*

From this table it is possible to derive how the result bit r and carry bit c can be calculated from inputs i_1 and i_2 using boolean expressions:

$$r = (i_1 \wedge \neg i_2) \vee (\neg i_1 \wedge i_2) \tag{2.3}$$

$$c = i_1 \wedge i_2 \tag{2.4}$$

A full adder works by adding the carry as an additional input, and from there on, longer numbers can be added by chaining full adders together.

In similar ways all other operation that computers and programs are able to perform can be reduced into boolean formulas. In the next section I explain how this can be useful for program verification.

2.2.4 Bounded Model Checking

The general idea of bounded model checking is not to prove that a program is fully correct. To prove that a program is correct, the program needs to have a formalised specification, and the program will only be as good as the specification. Instead, bounded model checking defines erroneous states that should not occur and uses SAT solvers to find inputs that lead to these states. This can help to find many bugs. This way of finding errors is called **falsification**, while verification is proving the absence of errors. Since the goal is not to have a full proof, bounded model checking does not replace any other verification methods. [3]

Bounded model checking is generally faster than other forms of model checking and therefore more suitable for software verification than other methods. The reason is because to prove the absence of errors, a model checking algorithm needs to check every possible state of the system, which can be a huge number even in small programs. When trying to find errors, bounded model checking keeps the number of possible states low by introducing a bound that limits the

execution length. If no errors are found, this bound will be gradually increased, until an error is found, all states are reached or it is no longer practicable.

Instead of BDDs, bounded model checking uses SAT solvers. We define conditions that indicate an errorful state. The system is transformed into a boolean formula and the SAT solver tries to satisfy the conditions that we have defined. If the formula is satisfied the SAT solver gives a possible value mapping of all variables in the formula, which can be traced back to find inputs that lead to errors.

Traditonally bounded model checking tools used SAT solvers, but they can now also be used with SMT solvers.

2.2.5 SMT solvers

SMT solvers are programs that can solve satisfiability problems using different theories than first-order logic that SAT solvers use. SMT stands for **Satisfiability Modulo Theories**, which is the name of the research field on these theories.

To showcase how an SMT solver operates, listing 4 shows an example of an SMT formula.

```
1 (declare-const a Int)
2 (declare-const b Int)
3 (assert (> a 1))
4 (assert (> b 1))
5 (assert (= 14351 (* a b)))
6 (check-sat)
7 (get-model)
```

Listing 4: *Example SMT formula*

This formula checks if a given number is a prime number. This works by declaring two integer constants a and b and then we add three constraints. These constraints are declared using the *assert* statement. Operators are placed using Polish notation where the operator comes before both arguments, so the first two constraints say that both a and b are greater than 1. Since they are integers they are at least 2. The third constraint defines that the product of a and b is be equal to the number we check, in this case 14351.

The solver tries to find values for the variables we defined so that all constraints are satisfied, or tries to prove that there exist no such values. In this case, if the number we test can be represented through a product where both factors are at least 2, it is not prime. On the other hand, if there are no such values, then the number is prime.

The *(check-sat)* command tells the solver to check for satisfiability, and the printed result is either `sat` or `unsat`. In this case the number is not prime, and the *(get-model)* command shows us example values for a and b that satisfy all conditions, as shown in listing 5.

This example uses the theory of integers, but in software verification we want to use the theory of fixed-size bitvectors instead. In this theory values are not integers in the mathematical sense, they are a sequence of bits with a specific length. The theory adds operators that transform these bitvectors the same way as many common CPU instructions do.

```
sat
(
  (define-fun b () Int
    127)
  (define-fun a () Int
    113)
)
```

Listing 5: *Solver output*

Translating a program into an SMT formula is therefore a lot more convenient than translating it into an SAT procedure. We do not have to model complex operations into boolean transformations, and do not have to seperate numbers into bits.

2.3 The CBMC framework

The CBMC framework and toolchain includes tools and libraries to perform bounded model checking on programs. The most prominent example is the C Bounded Model Checker CBMC which checks programs written in the C language. [5]

The framework has a modular architecture which consists of a frontend, which is language-specific, and a backend, which is language-agnostic. That means that the backend can be reused and new languages can be added by implementing a new frontend. Many existing projects such as JBMC [7] have taken this approach.

To implement a frontend the code needs to converted into a GOTO program, which a language-independent representation. All further transformations, analyses and checks are then done by the CBMC backend.

3 Related work

3.1 Bounded model checking for different languages

Bounded model checking is getting increasingly common in software contexts. There are projects for many programming languages to build a bounded model checker.

JBMC is built on top of CBMC and consists of a frontend for the Java language. [7] To be more precise, not Java, but the bytecode that is executed by the Java Virtual Machine. Therefore it can check other languages that compile for the JVM, such as Kotlin or Scala.

Kani is a verification and model checking tool for the Rust programming language. [16] This is done using annotations in the source code. It also uses some parts of CBMC for its backend.

The ESBMC project has been forked from CBMC and does bounded model checking using SMT solvers. [15] It includes frontends for various languages, such as C, C++ and Python [8]. It also includes a frontend for the Solidity smart contract language. [17]

3.2 Verification of WebAssembly programs

There is not much literature about the verification of WebAssembly code. Most publications talk about the verification of the formal WebAssembly semantics and specification itself, not the code it covers. [18] [19]

Verification of programs built in WebAssembly has not been discussed by any published literature. I have only found two master's theses that cover the topic. In 2019 a Uruguayan student developed an approach to detect runtime errors using static analysis of WebAssembly code. [11] In 2023 a Spanish student developed a formal specification for WebAssembly programs. [14]

4 Methodology

In this chapter I explain the method I used to achieve the goal of designing a tool that can do bounded model checking for Wasm programs. First I talk about the general approach and then I explain the tools and techniques that have been used to achieve this.

4.1 Approach

To do bounded model checking we have to transform the program into a formula that can be checked by a solver. This can be done by converting it into the CNF format for SAT solvers or to the SMT-LIB format for SMT solvers.

To make it easier, we use the CBMC framework. The key is that the framework, similar to a compiler, works in two parts. The first part, the frontend, translates the code that needs to be checked into a language-agnostic **intermediate representation**.

The backend then takes this intermediate representation and transforms it into a formula to be used with a solver. This approach makes it possible that the same backend works with different languages.

In CBMC, the internal representation is the **GOTO language**. How the GOTO language works and how CBMC reduces it into a solver formula is explained in chapter 5.

As a result of this, it is only necessary to design a frontend that translates Wasm bytecode into this intermediate representation. This happens in chapter 6 where I take a detailed look at Wasm bytecode. I show the features of the language, how they work and how they may be converted. I also discuss what to look out for and if there are some areas where the CPROVER framework needs to be extended.

In chapter 7 I talk about some practical aspects of bounded model checking Wasm programs.

4.2 Tools and Techniques

In this section I want to explain which tools, programs and compilers I used for this work.

The main point of reference has been the CBMC program that includes CPROVER toolchain, which the content of this thesis builds upon. I used CBMC version 6.0.0 to explain its inner workings and to demonstrate its use. To showcase GOTO programs I ran CBMC with the `-show-goto-functions` option.

To generate WebAssembly code in most cases I used the C language with the Clang compiler version 14.0.0. Whenever the code needed standard library functions I used Emscripten in version 3.1.70. In some cases I wrote the Wasm code by hand using the WebAssembly text format. I also used this format for showing compiled Wasm code. To convert Wasm code between binary and text format I used the tools `wasm2wat` and `wat2wasm` from the WebAssembly binary toolkit.

In some cases I used the Rust programming language to show an alternate source of Wasm code when needed. For this I used version 1.73.0 of the Rust compiler and the Cargo build toolchain.

Also sometimes I use Java code to demonstrate JBMC, because Java bytecode shares some similarities with Wasm bytecode. To compile the Java code I use version 21.0.4 of the OpenJDK Java development kit.

5 CBMC and the GOTO language

5.1 Introduction

In this chapter I explain the CBMC backend, which we are going to use for our project. I will show how we are targeting it using an abstract syntax called GOTO programs. First I will explain the building blocks and features of GOTO programs. I will also shortly explain how the backend transforms the GOTO program into a formula that works as input for a solver.

5.2 The GOTO language

5.2.1 Introduction

CBMC needs to transform the code it checks into a machine readable form that can be used by a solver. Since it can be used with multiple languages, each frontend needs to transform the source program into a language-independent representation. In CBMC, this intermediate representation are GOTO programs. The namesake feature of GOTO programs is the *goto* statement. Each structured statement that common languages use such as *while*, *for* or *if* is translated into either a nonconditional or a conditional *goto* statement. GOTO programs are imperative, a program consists of functions. Each function contains a linear list of statements, because any kind of branching or loops have been replaced by the *goto* statement.

To see how this works we use the example code in listing 6, which calculates the factorial of a number.

```
1  int factorial(int n) {
2      int ret = 1;
3      for (int i = 1; i <= n; i++) {
4          ret *= i;
5      }
6      return ret;
7  }
```

Listing 6: *C function that calculates the factorial*

This will get transformed into the GOTO program shown in listing 7.

The program looks quite different, but I will explain what it does it line by line. The first thing that happens in line 3 is a variable declaration ret of type $sint32$, which is a signed 32-bit integer. The *ASSIGN* statement in line 4 assigns a value to ret, the constant $sint32$ of 1. One of the conditions of GOTO programs is that declaration and assignment always need to be separated. Then we declare a variable i in the same way and set it to 1.

When we look at line 7 we see that at the beginning there is a label definition. Labels get used as targets by *GOTO* statements, one of which comes already in the same line. In this case it is a conditional *GOTO*, there is condition after the *IF*, only if the condition is true, we execute the

```
1       FUNCTION factorial INPUT n: sint32 RETURN sint32
2
3       DECL ret: sint32
4       ASSIGN ret := const(1, sint32)
5       DECL i: sint32
6       ASSIGN i := const(1, sint32)
7   1:  GOTO 2 IF greater(n, i)
8       ASSIGN ret := mult(ret, i)
9       ASSIGN i := add(i, const(1, sint32))
10      GOTO 1
11  2:  DEAD i
12      SET_RETURN_VALUE ret
13      DEAD ret
14      END_FUNCTION
```

Listing 7: *Equivalent GOTO program*

GOTO. The condition is that the value of n is greater than the value of i. Instead of $n > i$ I have written it like I have to emphasize the fact that "greater" is an **operation** with two inputs, that produces a boolean value as a result. To explain what the program is doing, this is the condition when the original C code would leave the loop, so here we go to label 2 basically after the loop.

The next two lines are the logic that used to be inside the for loop: ret gets multiplied by i and i gets incremented by 1. Then it is clear how the for loop got translated into *goto* statements. In line 10 we jump back to line 7 to test the loop condition again.

Line 11 introduces the *DEAD* statement, this marks that the lifetime of the variable i has ended and it is no longer in use.

GOTO programs are a form of an **abstract syntax tree**, or AST. ASTs get used by a lot of software such as compilers, to translate the source code into an internal representation, which is easier for the program to do further processing on.

It is called tree because we can think of the structure like a tree: A function is a list of statements, and each statement may be a combination of smaller syntax elements. For example in line 9 of listing 7, the *ASSIGN* statement consists of two parts: The left hand side, which is the variable i, and the right hand side of the assignment. The right hand side is an additon operation, which takes two inputs, the two summands. The first summand is the variable i, and the second summand is a constant declaration, which consists of a value and its type.

Before I explain each statement in detail, I want to give definitons and explanations for some terms that I will use in the next sections and throughout the whole thesis.

Statement A statement is a construct in an imperative language that carries out an action, it therefore changes the state of the program in one way or another. In the GOTO language functions consists of a list of statements. There are different kinds of statements, which we will explore in section 5.2.3. When I show the syntax of GOTO progams, statements are always highlighted by writing them in UPPERCASE.

Symbol A symbol is an abstract way of defining a unit that can be the target of an assignment. It can be a variable, an array element, a member of a struct.

Expression An expression is a part of the syntax that can be reduced to a single value, which it evaluates to. Expression are part of many statements, for example the assignment statement needs an expression on the right hand side or the conditional *goto* statement needs an expression that evaluates to a boolean value. Expression can be nested, for example the addition expression needs two subexpressions that represent the summands. Expressions can be classified into the following subtypes:

- Operations: Types of expressions that require at least one subexpression as input which they transform into a single result
- Symbols: Expressions that reads the value of a symbol
- Constants: Expressions that returns a constant value, they consist of a literal and a type

When I write GOTO programs I try to differentiate expressions from statements by writing them in lowercase. Expression can have arguments, these are written using parentheses. This way they resemble functions, which they are in some way, since they produce a single result that is only dependent from the inputs. Arguments for expressions can either be subexpressions or types, exception is the constant expression that consists of a literal and a type. We take a close look at various expression types in section 5.2.4.

5.2.2 Types

GOTO programs can hold data of various types, in this section I will cover only the ones that we need to use in our Wasm frontend.

Boolean

The boolean type represents a boolean value, which can only have two states, true or false. This type represents abstract boolean values, which is different from technical representations, which treat boolean values either as bitvectors of size 1 or as integers.

Fixed-Size Bitvectors

A bitvector is a list of individual bits where each bit can be either set to 1 or 0. In GOTO programs we are only concerned about bitvectors of fixed size, so each bitvector is defined with a width n. Bitvectors can be of arbitrary width and the total number of states a bitvector can have is 2^n. Due to the data types that Wasm defines we will mostly use bitvectors of sizes 128, 64, 32, 16 and 8. Bitvectors are just an array of bits, but they can be interpreted in many different ways. Here is an overview of interpretations we are going to use:

- Uninterpreted: No interpretation, just raw data
- Unsigned integer: Binary representation of an integer that is zero or positive
- Signed integer: Representation of an integer that can be negative using two's complement
- Uninterpreted integer: Data represents an integer, either signed or unsigned

- Float: Data represents a real number using the IEEE 754 standard

Uninterpreted bitvectors are used when the representation of the bits is not needed or not important. In our syntax uninterpreted bitvectors are represented as bvn where n is the bit width.

Integers represent integer numbers in binary format. We have to differentiate between signed or unsigned integers. Unsigned integers treat every bit as a positive binary digit and therefore can represent the number range $[0; 2^n - 1]$.

Signed integers use a technique called two's complement. In two's complement, the most significant bit is used to determine the sign of the number, 0 means positive and 1 means negative. Positive numbers have the same bit pattern as they do in unsigned integers. To change the sign of a number, we invert all the bits and add 1.

For example representing the number 7 as an 8-bit signed integer results in 00000111. Inverting all the bits leads to 11111000, then we add 1 and so we get 11111001, which is the representation of -7. Signed integers can hold values in the range $[-2^{n-1}; 2^{n-1} - 1]$.

In many cases I will need to refer to the minimum or maximum value of a bitvector integer representation. To simplify for the reader I will call these values *INT_MIN* and *INT_MAX*.

I have specified the ranges because they are important and highlight the difference between normal arithmetic and bitvector arithmetic. What if an operation, for example multiplication, leads to a result that is outside the number range? The answer is if the result of a bitvector operation of width n can only be represented by a longer bitvector, the result is the last n bits of the longer bitvector. This is also called wrapping arithmetic, because the values "wrap around", that means if you add 1 to *INT_MAX*, you will get *INT_MIN*. This is therefore a form of modular arithmetic, so for unsigned integers we can define the result of an operation as the result $mod\ 2^n$.

There is another form other than wrapping arithmetic, which is called saturation arithmetic. In saturation arithmetic an integer as represented by the nearest possible representation. That means in other words that if the value would be greater than *INT_MAX*, it will be *INT_MAX*, and similarly, if the value is smaller than *INT_MIN*, it will be represented by *INT_MIN*. This is only used when it is explicitly demanded, otherwise wrapping arithmetic will be used.

Uninterpreted integers are integers that can be either unsigned or signed. These are used because many operations on integer bitvectors produce the same bitvector result regardless of signed or unsigned arithmetic.

The type symbols in the syntax I chose are $intn$ for uninterpreted integers, $sintn$ for signed integers and $uintn$ for unsigned integers of length n.

Floats or floating-point numbers are bitvectors interpreted as real numbers up to a certain precision. Floating-point arithmetic uses the IEEE 754 standard. In this standard, numbers are represented using a sign bit, a dedicated number of bits for the significand and a number of bits for the exponent. The number can be constructed as $sign * significand * 2^{exponent}$.

There also exist representations for special use cases in mathematics like negative zero and infinity. There are also special bit patterns called NaN, *Not a Number*, they represent the result of operations like $\frac{0}{0}$ that make no sense mathematically.

Floats cannot have an arbitrary width, the standard describes types of width 16, 32, 64, 128 and 256. We are only going to use floats of width 32 and 64. The notation of those types is $f32$ and $f64$.

Arrays

Arrays are a composite type, they hold a number elements of another data type. Technically there can be arrays of arrays but these will not be used in the next chapters. Arrays get initialised using a type and a size. Each element can be accessed individually using an index operator.

Arrays can be difficult to convert into boolean logic because they provide a new layer of abstraction. If each array index would be a constant value, then each array element can be dealt with in the same way as any other variable. But they do not have to be, one can index arrays using any expression they want, for example: arr is an array and i and k are integer variables. I can set an element $arr[i]$ and retrieve it with $arr[k]$.

This can be resolved in this example by an implication: If $i = k$, then the value set to $arr[k]$ is the value $arr[i]$. Otherwise it may be a different variable or undefined or an initial value.

Arrays do not correspond to a native data type in Wasm, but they will be used to model some language elements such as memories and tables.

Structs

Structs are a composite type that consists of elements of other data types. A struct definition needs to define all members, each member needs to have a name and a type. Each member can be accessed using the name of the variable and the name of the member.

Like arrays, structs are not a native type in Wasm, but they will be used to model the return values of functions that have multiple return values.

5.2.3 Statements

In this section I take a closer look at the different statements the GOTO language defines. I also explain shortly how the backend transform these statements into a formula used by a solver.

Unconditional Goto

Syntax: `GOTO <label>`

The *goto* statement gets followed by a label, which the control flow jumps to.

goto statements do not get transformed into the solver formula directly. Instead, they guide the way of a process called **symbolic execution**. This step starts at the beginning of the function and translates other statements into elements of the formula. When it reaches a *goto* statement, it continues at the target label.

If a *goto* statement jumps backwards, it creates a semantic loop. Statements in that loop will get executed multiple times during symbolic execution. There is a bound that limits the number of executions in a loop. If no counterexample can be found, the bound gets increased.

This is the defining feature of bounded model checking, in this way counterexamples can be found very quickly, but it may take very long or is not possible to determine that no counterexamples exist.

Conditional Goto

Syntax: `GOTO <label> IF <expression>`

The conditional *goto* statement gets followed by a label and an expression that needs to be of boolean type. If the expression evaluates to true, then execution continues at the target label. Otherwise the execution continues at the next statement.

```
1      (...)
2      GOTO 1 IF greater(n, const(5, uint32))
3      ASSIGN x := const(3, uint32)
4      GOTO 2
5  1:  ASSIGN x := const(4, uint32)
6  2:  (...)
```

Listing 8: *Example of a conditonal Goto*

To explain how this gets converted into a boolean formula I use the example in listing 8. Here we have a conditional *goto* statement in line 1. What happens during symbolic execution? The sybolic execution takes both paths and combines them with the condition, or the inverse of the condition. We know that if n is greater than 5, then x will be 3, otherwise x will be 4. So the boolean formula for this sequence of statements could look like this:

$$(n = 5 \wedge x = 3) \vee (n \neq 5 \wedge x = 4) \tag{5.1}$$

Assignment

Syntax: `ASSIGN <symbol> := <expression>`

The assignment statement changes the value of the symbol to the value of the expression. Both the symbol and the expression need to have the same type.

To the solver, assignment creates a condition of equality. We know that at that point in time, the symbol is equal to the expression. But what happens when we re-assign the same variable, or do something like this: $x := x + 1$

This would lead to a contradiction, and therefore the backend uses a technique called **single static assignment** (SSA). Each time we assign a new value to a variable in a symbolic execution path, we create a new variable. We can do this by "versioning" the variable, so $x := x + 1$ would be replaced by $x_2 := x_1 + 1$ during SSA transformation.

Assertion

Syntax: `ASSERT <expression>`

With the assertion statement you can define properties that you want to disprove. The condition needs to be boolean expression.

Ultimately the goal of bounded model checking is to find counterexamples. To do this we transform the program into a boolean formula and try to satisfy this formula with a solver. The goal is to find counterexamples, a counterexample is a combination of inputs that disprove the assertion condition.

Since we want to satisfy the formula to disprove each assertion, what we need to do is take the the formula for the rest of the program and invert each assertion condition.

```
1    DECL x: sint32
2    ASSIGN x: mult(n, n)
3    ASSERT greatereq(x, 0)
```

Listing 9: *Assertion example*

Listing 9 shows an example for an assertion. In this case we have a variable n that acts as input, and a variable x. We assign the value n^2 to x and assert that x is greater than or equal to 0.

To find an input n that disproves the assertion, we need to define the normal conditions for the program and invert the assertion. This would result in:

$$x = n \cdot n \wedge x < 0 \tag{5.2}$$

Because of overflow the result of a signed integer multiplying itself is not guaranteed to be positive. Plugging this formula into an SMT solver gives an example of $n = -1049601$.

Assumption

Syntax: `ASSUME <expression>`

The assumption statement looks similar to the assertion statement, after it follows a boolean expression as well. We can use this statement to define properties that we expect to be true. The solver treats these similar to assertions, the only difference is that we want to satisfy all assumption, so we do not invert the condition.

An example use for assumptions is to make it easier to translate certain operations. Listing 10 for example shows how unsigned integer division of a and b can be implemented using assumptions. [13, p. 146]

```
1    DECL q: uint32
2    DECL r: uint32
3    ASSUME equal(b, add(mult(a, q), r))
4    ASSUME less(r, b)
5    SET_RETURN_VALUE q
```

Listing 10: *Assumption example*

We define two variables q for the quotient and r for the remainder. Then we define two assumptions: $b = a \cdot q + r$ and $r < b$. The solver will try to find values for q and r that satisfy both conditions, and if they do, they are the quotient and the remainder.

Variable declaration and termination

Syntax: `DECL <symbol>: <type>`
Syntax: `DEAD <symbol>`

Each variable that will be used needs to be explicitly declared with a type. There is no concept of scopes in GOTO programs. A variable is valid as soon as it is declared until it is explicitly terminated using the *DEAD* statement. One could possible declare a variable inside a function and it would be available outside of it, but that it very uncommon. To make certain analyses easier, a variable declaration should be as close as the first use of the variable as possible. Similarly, a variable termination should be as close to the last use of the variable as possible. A declaration cannot be combined with an assignment, these need to be two different statements.

Statements regarding functions

Syntax: `CALL_FUNCTION <function>(<arguments>)`
Syntax: `ASSIGN <symbol> := <function>(<arguments>)`
Syntax: `SET_RETURN_VALUE <symbol>`
Syntax: `FUNCTION_END`

The function call statement calls a function and transfers control to the function. If the function accepts arguments they need to be passed. If a function call returns a value, the function call can be used as the right hand side of an assignment, but not as a subexpression. A function call is allowed to be recursive. The *SET_RETURN_VALUE* statement sets the return value of the current function. This does not necessary need to be at the end of a function. The end of function statement marks the end of the function.

5.2.4 Expressions

Symbol access

Syntax: `<symbol>`

The symbol access expression accesses the value of a symbol. The symbol can be any kind of symbol, for example a variable. The resulting type of the expression is equal to the type of the symbol.

Constants

Syntax: `const(<literal>, <type>)`

The constant expression defines a constant value. A constant value consists of a literal and its type.

Simple arithmetic operations

Syntax: `op(<expression>,<expression>), op = add | sub | mult | div | rem`

Arithmetic operations consist of addition, subtraction, multiplication and division. There is also a remainder operation, which calculates the remainder of a division. Each of these

operations take two expressions of the same type as input, which needs to be an integer or float type.

The type of the inputs determines the type of the output and also the semantics of the operation. Floating point numbers and integers behave very differently. Division and remainder operations also differ between unsigned and signed integers.

Comparison operations

Syntax: `op(<expression>,<expression>), op = equal | notequal | less | lesseq | greater | greatereq`

Comparison operations take two expressions of the same type as arguments. The result is a boolean value depending on if the condition is fulfilled or not. The equal and notequal operations work with any bitvector type. The other operations need an integer or float interpretation and differ in semantics between unsigned and signed integers.

Boolean operations

Syntax: `op(<expression>,<expression>), op = and | or | xor`
Syntax: `not(<expression>)`

Boolean operations take boolean expressions as inputs and produce a boolean result. The and, or and xor operations take in two inputs, while the not operation takes a single input an inverts the result.

Bitwise boolean operations

Syntax: `op(<expression>,<expression>), op = bitand | bitor | bitxor`
Syntax: `bitnot(<expression>)`

Bitwise boolean operations take one or two bitvectors as input and produce a bitvector result. The result gets constructed by performing the boolean operation for each bit. The inputs need to be of the same type and can be any bitvector type other than float.

Shift operations

Syntax: `op(<expression>,<expression>), op = shl | ashr | lshr | rotl | rotr`

Shift and rotation operations need two arguments, the first needs to be a non-float bitvector and the second an integer bitvector. Shift operations shift the bits from the first bitvector by a number of bits to the left or right that is equal to the second bitvector. Left shift and logical right shift fill the remaining bits with zeroes, while rotations fill the remaining bits the bits that got shifted out on the other side. The arithmetic right shift fills the bits on the left with zeroes or ones depending on the most significant bit of the original value.

Unary arithmetic operations

Syntax: `op(<expression>), op = abs | neg`

The unary arithmetic operations take a single integer or float bitvector as arguments. The abs operation calculates the absolute value, while the neg operation flips the sign. The input for the negation operation is not allowed to be an unsigned integer.

Unary bitwise operations

Syntax: `op(<expression>, op = popcount | ctz | clz)`

These operations take a non-float bitvector as input and return an unsigned integer. The popcount operation counts how many bits are set to 1. The ctz (count trailing zeroes) operation counts how many bits in a row are set to 0 from the end, while the clz (count leading zeroes) operation counts how many bits in a row are set to 0 from the start of the bitvector.

If expression

Syntax: `if(<expression>, <expression>, <expression>)`

The first expression is a condition and needs to be a boolean, the second and third expression can have any type but it needs to be the same. If the condtion is true, then the result of the operation is the second expression, otherwise the third.

Bit manipulation operations

Syntax: `extractbits(<expression>, <expression>, <type>)`
Syntax: `updatebits(<expression>, <expression>, <expression>)`
Syntax: `concat(<list[<expression>]>, <type>)`

The extractbits operation has three arguments, the first is a source bitvector, the second is an unsigned integer and the third is a type. The result is a bitvector of the specified type where the bit pattern is extracted from the source bitvector, starting at the index specified by the second argument.

The updatebits operation replaces some bits in the source bitvector, starting from the index specified by the second argument, with the bit pattern specified by the bitvector in the third argument.

In the extractbits and updatebits operations the index starts from the least significant bit.

The concat operation takes a list of bitvectors and a type as input and produces a bitvector of that type where all input bitvectors are concatenated together. The order of operands is that the most significant operand comes first.

Type cast operation

Syntax: `typecast(<expression>, <type>)`

The type cast operation converts the value from the first argument into the type specified by the second argument. The semantics of the casting are similar to the casting done by casts in C language.

Converting a bitvector to boolean results in false if the value is a float or integer representation of 0, or an uninterpreted bitvector where all bits are set to 0. Otherwise the result is true. Similarly,

converting a boolean to a bitvector results in the integer or float representation of 1 if the boolean is true, otherwise 0. Uninterpreted bitvectors are treated like integers.

Conversion between floats and integers tries to find the best possible representation in the target type of the closest numerical value. Conversion between uninterpreted bitvectors and any other bitvector of the same length does not change the bit pattern.

Conversion from a bitvector to a bitvector of longer width fills the new bits with zeroes. The exception is signed integers, here the bitvector gets sign-extended, that means that the value of the most significant bit, which determines the sign, is the value that the new bits are filled with.

Conversion from a bitvector to a bitvector of smaller width results in the most significant bits being dropped of.

Overflow detection expression

Syntax: `binaryoverflow(<expressiontype>, <expression>, <expression>)`
Syntax: `unaryoverflow(<expressiontype>, <expression>)`

The overflow detection expression returns a boolean value indicating if an integer operation would result in an overflow. The first argument is an expression type indicating which operation gets performed, and the other arguments are the operands of that operation. Valid options for the first arguments are addition, subtraction, multiplication, negation and the left shift operation. The type of the operation gets determined by the first operand, which needs to have the same type as the second, with the exception of the left shift operation.

This expression can be combined with an assertion statement to place before the actual operation happens. With this it is possible to automatically detect overflows, which is a source of error in many cases.

The isNaN expression

Syntax: `isNan(<expression>)`

The isNaN expression returns true if the float bitvector input is a NaN value, otherwise false.

6 Designing a Wasm frontend for CBMC

To build a frontend for Wasm, we need to convert the bytecode into the GOTO language. In this section, we take a deep look at Wasm bytecode and work out how this can be done. With JBMC, a frontend for another bytecode language already exists, so we can take advantage of that. When it is practical, we take a look into how JBMC works, but we cannot reuse a lot of it, because Wasm and JVM bytecode work fundamentally different in many aspects.

6.1 Introduction to WebAssembly Bytecode Format

WebAssembly bytecode gets released as a unit that is called a module. A module contains functions, code, memory, data, imports and exports. Code is organized in functions that contain a list of bytecode instructions. Functions can hold data in local variables, but individual instructions use an operand stack to process data.

There are four basic data types: Integers, floating-point numbers, vectors and references. Integers represent integer numbers, we take a closer look at those in chapter 6.3. Floating-point numbers represent real numbers, and we deal with those in chapter 6.4. Vector data is a type that can hold a number of shorter numeric data types and perform operations on those at the same time to increase performance. This concept is explained further in chapter 6.5. Reference data is used to point to another object. It serves as a memory-safe replacement for pointers. The types of references and its use are discussed in detail in chapter 6.9. There are other forms of types in Wasm like function types, but these are abstract types used for Wasm's strict formal type system, and cannot hold data.

Wasm uses linear memory, which is just an uninterpreted array of bytes. There are no memory addresses, just indices into the memory array. An in-depth look into memory is given in chapter 6.6.

Wasm modules do not serve as standalone applications, they are intended to run inside a host environment. Therefore a Wasm module can communicate to the outside by importing and exporting functions and data. A Wasm module is able to call host functions and vice-versa.

6.2 Functions

6.2.1 Introduction

Functions are the basic architectural element in which code is separated. Each function can have any number of inputs and any number of return values.

Functions contain code, and since Wasm is a bytecode, code is just a list of instructions. Each instruction performs a specific action. Instructions can perform calculations, read or write data, or change the control flow.

6.2.2 Operand stack

WebAssembly is a bytecode, that means it can't run natively on an operating system and need some kind of runtime environment. Contrary to machine code, bytecode does not target a physical processor, it targets a virtual processor with a virtual instruction set. WebAssembly creators designed Wasm to be a **stack machine**. [10] A stack machine is a machine model that stores inputs and outputs of instructions on a stack. This is contrary to a register machine, where instruction data is stored in registers, like in many modern CPU architectures. Stack in this case means a first-in-first-out data structure, with two defined operations:

- Push: Add data on top of the stack
- Pop: Retrieve data from the top of the stack

Instructions for a stack machine pop their arguments off the stack and push their results on the stack. To illustrate that with an example, we use the ***t*.add** instruction. This instruction adds two numbers together. *t* denotes the type, there is the same instruction for each type, but we look at that later in detail. Like one would expect, the **add** instruction has two inputs, and one result. When the **add** instruction gets executed, the two values from the top of the stack get popped, and the result of the addition gets pushed to the top of the stack. Here is an example of the stack manipulation for simple addition:

$$\begin{pmatrix} 19 \\ 7 \\ \dots \end{pmatrix} \rightarrow \begin{pmatrix} 26 \\ \dots \end{pmatrix} \tag{6.1}$$

For each individual instruction it is defined how many inputs and outputs they have, which can be any number. There are instructions with no inputs like ***t*.const** that pushes a constant on the stack, and there are instructions with no outputs like the ***t*.store** instruction, that takes a value from the stack and stores it in memory.

This stack-based approach is different from the GOTO language. Instead of storing intermediate results on a stack, the GOTO language stores intermediate values in variables, or combines multiple operations together in a tree of expressions. What we need to do is to "compile" the stack away, similar to how a just-in-time compiler would do.

The GOTO language consists of statements and expressions. There are instructions like most arithmetic instructions that have some inputs and push a single result to the stack. This is the same as an expression in the GOTO language — it evaluates to a single value. Other instructions change the state of the program, for example instructions that set local variables or store data in memory. These would be translated into statements.

Listing 11 shows an example Wasm code to explain the next steps. We have not talked about Wasm code in depth so I have annotated each instruction. One neat property of Wasm bytecode is that the operand stack is statically determinable. That means we can determine how the operand stack looks at any point of execution without actually executing the code. This means we can "simulate" the stack to help us transform the code into expressions. This is called **symbolic execution**.

```
local.get 0   ;; push the first parameter on the stack
i32.const 10  ;; push the constant 10 to the stack
i32.add       ;; add two values from the stack together
local.get 1   ;; push the second parameter on the stack
i32.mul       ;; multiply two values from the stack
return        ;; return whatever is on the stack
```

Listing 11: *Example WebAssembly code*

Looking at the example code we have code of a function that has two parameters and returns a value. If one knows the basics of how the operand stack works, it can be easy worked out that an imperative representation would look like, as shown in listing 12.

```
return arg1 * (arg0 + 10)
```

Listing 12: *Imperative pseudocode*

The way we can transform our Wasm code into expressions is by simulating the stack. We have to do this instruction by instruction. But we do not actually execute the program, we would not know the values anyway. Instead we place expressions on the stack. We know that whenever an instruction returns one result, it might me treated as an expression. All the instructions in our example do this, bar the return. So we translate each instruction to an expression, and put this result on our symbolic stack.

Instructions like **add** and **mul** consume inputs from the stack. When we do this, we create a new expression and link together the previous ones we consume from the stack. In the GOTO language, an addition or multiplication expression takes two arguments, both of which are expressions themselves.

Figure 6.1 shows how the symblic stack looks after each instruction.

This seems like a nice way to chain together our expressions. However we left out the last line because the return instruction is a statement. Whenever we have an instruction that translates to a statement, we need to emit a statement for our GOTO program. In this case, we set the return value to the expression on top of our symbolic stack.

The result that we want after transformation is a linear list of GOTO instructions. Therefore we start with an empty list, where we add each statement which we encounter. We do this for every instruction in our function, and with this algorithm we can transform our Wasm bytecode into GOTO programs, which will be processed further by the CBMC backend. How we process individual each individual instruction and how we handle special cases will be treated in subsequent chapters.

6.2.3 Local Variables

The operand stack is not always sufficient enough to hold all the data needed for function execution. That is why functions can define a list of local variables that can store values. These

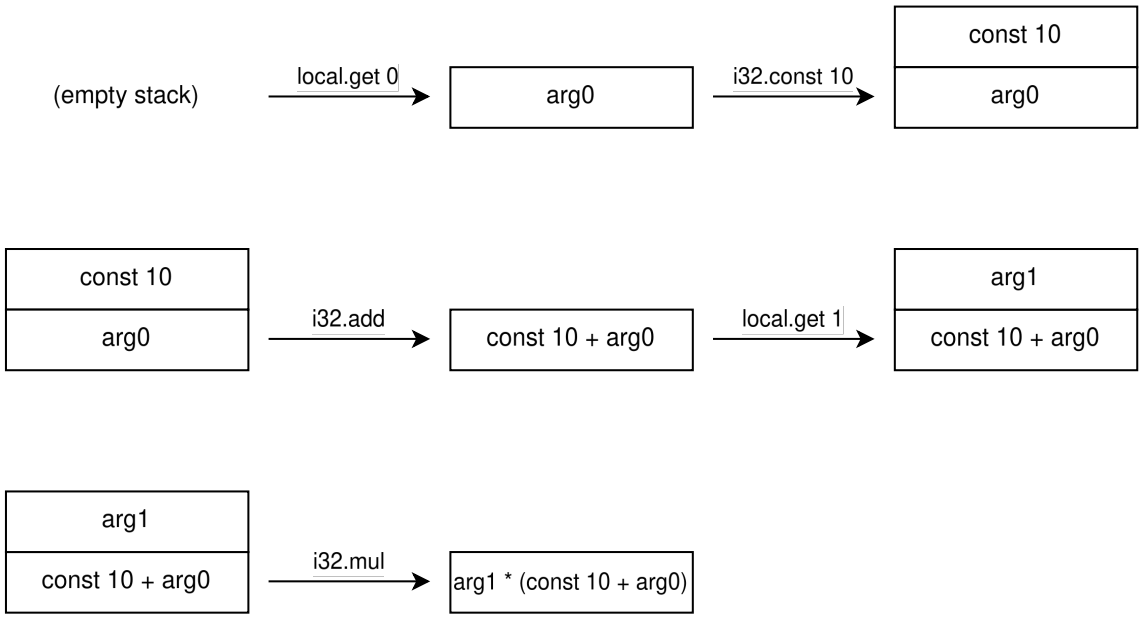

Figure 6.1: *Symbolic stack execution*

values are only accessible from inside the function and are accessed using an index. Index 0 is used for the first local variable, index 1 for the second and so on.

When functions have input parameters, they get stored in the first local variables. Local variables holding parameters do not have to declared, the first declared local variable starts at the smallest index that does not hold a parameter. When a function has for example two arguments, then the first two local variables store the values of the arguments. The variable at index 2 would be the first declared local variable. Parameters only hold their value initially and can be subsequently overwritten. Local variables that do not hold parameters get initialised with a default value, which is 0 for any numeric type and null for references.

There are three local variable instructions, **local.get**, **local.set** and **local.tee**. Each of them includes a static index after the instruction to specify which variable is affected. The **get** instruction accesses a variable and pushes the result to the operand stack. the **set** instruction takes a value from the stack and stores it in a local variable. The **tee** instruction was added for convenience, it sets a local variable and leaves the value on top of the stack.

During transformation we treat the local variables as normal variables. GOTO programs allows us to define them and assign values to them. Since Wasm bytecode does not store the names of local variables, we have assign them generic names like *local0*, *local1*, etc. The **get** instruction returns a value and does not change the state of the program, so it is an expression. The GOTO language provides us with a symbol access expression, which we can use for that.

The **set** instruction gets translated into an assignment statement, since it assigns a variable. The left hand side of the assignment is the name of the variable, and the right hand side is the expression that we take from the stack. The corresponding type class is *code_assignt*. The **tee** instruction is the same as the **set** instruction, but instead we leave the expression on the stack.

In GOTO programs we have to declare each variable we use and its lifetime. For this we use the *DECL* and *DEAD* GOTO instructions. The *DECL* statement does not contain an assignment,

it just declares a variable and its type. GOTO programs mandate that the lifetime of a variable should be as short as possible. That means that each declaration should be as close to the first use as possible, and each *DEAD* should be as close as possible to the last use of the variable. To place the declarations correctly, we can keep a flag in our data structure that we use to hold the local variables. Initially it is set to false for all non-argument variables. When an instruction uses the variable and the flag is still false, we know that we use it for the first time. Each time this happens we add a declaration statement. If the first use of a variable appears to be a **get** instruction, which is possible, we add an assign statement after the declaration that initialises the variable to its default value.

After we processed the whole function we need to end the lifetimes of the variables. For this we use a similar approach. We iterate through the GOTO statement list backwards and add an end of lifetime statement for each variable after its last use. If the last use is a **set** instruction, we could even optimize it away and delete it, since an assignment without use is a form of dead code.

6.2.4 Return values

Functions in Wasm can have an arbitrary number of return values. When a function has ended execution either through reaching the end of the code or through executing a return statement, the values sitting on top of the stack are the values the function returns. The types and amount of return values are determined by the type signature of the function. If there are any other values left on the operand stack, they are discarded. The return values are then pushed to the calling function's stack.

In GOTO programs the function return value is set using a special statement. An issue is that in GOTO programs each function can only have one return value. To circumvent this restriction, we have to combine the return values in a single structure. This is possible GOTO language supports creating a complex type that has named members, like a struct in C or an object in Java.

So when we have a function with a single return value, we can set it directly. Whenever a function has more than one return value, we build a structured type out of all return values. In the calling function we create a variable out of this complex type that gets this data back through the function call statement.

We can then use the values by using member access on this variable.

6.3 Integer Data

6.3.1 Introduction

Wasm defines two integer data types, $i32$ and $i64$, which represent uninterpreted 32-bit and 64-bit integers. The reason for not having signed and unsigned integers is that many instructions, like normal addition, behave semantically the same between regardless of the signedness of the value, the result has the exact same bit pattern in all cases. Whenever this is not the case then there are two variants of an instruction, one with signed and one with unsigned semantics.

Signed interpretation uses two's complement, which is the same as signed integers in GOTO programs.

Therefore in GOTO programs, we have to use uninterpreted integers of 32 and 64 bits to directly represent $i32$ and $i64$ types. If an instruction requires a signed or unsigned value, we have to add a type cast to a signed or unsigned integer.

6.3.2 Shorter integer types

Wasm also indirectly supports 16-bit and 8-bit integers. They are useful for many things like text encoding. One cannot declare values directly with this type, but there are many instructions that convert between these types and longer types, for example for writing to memory. For example the instruction **i32.load8_s** reads an 8-bit signed integer from memory and extends it to 32 bits, preserving the value. There is a similar instruction that stores only the last 8 bits of an $i32$ value to memory.

Also Wasm supports shorter integers in packed types. Packed types are interpretations of vector data, which will be covered in detail in chapter 6.5. There exist shapes $i8x16$ and $i16x8$, which means that the 128-bit vector data gets interpreted as 16 8-bit integers or 8 16-bit integers.

Shorter integer arithmetic can be emulated by using a longer type and ignoring the first bits. This is possible because ignoring the first bits is equivalent of interpreting each result $mod\ 2^n$, where n is the bit width of the shorter type.

6.3.3 Overflows and undefined results

A very common source for bugs regarding integer values are overflows. These occur when operations result in values outside of the number range of the specified type. In many languages these type of errors do not interrupt the program or send any kind of warning, because checking every operation for overflow comes at a runtime cost. Overflowing an integer can result in defined results using wrapping arithmetic, but in some cases even undefined behaviour can occur. [12, §6.5 5]

GOTO programs are equipped with an expression type that can detect overflows. We are able to assert that no overflow occurs by adding an assertion to our program as shown in listing 13.

```
1         c = a + b
2         becomes
3         ASSERT not(binaryoverflow(add, a, b))
4         ASSIGN c := add(a, b)
```

Listing 13: *Overflow assertion*

In the C frontend, it places overflow checks before signed integer arithmetic operations, because a signed overflow is undefined behavior in C. Unsigned overflow checks can be added with a flag. In Java, overflow behavior is defined using wrapping arithmetic, so JBMC does not add overflow checks in front of arithmetic operations.

In Wasm, the overflow semantics are like Java well defined in all cases using wrapping arithmetic. The operands are treated as unsigned integers and the result is the mathematical result $mod 2^n$, where n is the bit width.

It is difficult to place overflow checks in Wasm bytecode, because the type system does not differ between unsigned and signed integers. The reason is that signed and unsigned integers have different number ranges, and therefore different overflow semantics. To demonstrate this, I show an example using 8-bit integers for simplification:

$$11111111 + 00000111 = 00000110 \quad \text{(bit pattern)}$$

$$-1 + 7 = 6 \quad \text{(signed integers)}$$

$$255 + 7 = 6 \quad \text{(unsigned integers)}$$

This clearly shows that in the example an unsigned overflow occurs, but not a signed overflow. Since additon has no signed or unsigned variants, we cannot presume that additon uses signed or unsigned integers. Therefore it is impossible to add reliable overflow checks for addition, the same principle applies to subtraction and muliplication.

Some arithmetic operations lead to undefined results, because the result of the operation is undefined in mathematics. A classic example is the division of an integer by 0. This is an error in any case and leads to program termination. We have to explicitly check for these cases and add assertions to find ensure that these cases do not occur.

6.3.4 Integer operation instructions

In this section we look at integer operations that are defined by Wasm and determine how we can convert them into GOTO programs, and what we have to look out for. In Wasm, an operation is an instruction that has the form $t.op$ where t is the type and op is the operation. Whenever there is an operation that has different semantics for signed and unsigned integers, the notation is $t.op\_s$ and $t.op\_u$. The type for the following instructions can be either $i32$, $i64$ or a packed integer type.

Packed integer types are vector interpretations and are explained in detail in section 6.5. They consist of a base type and a number of lanes. The operations included in this section perform operations on each lane individually, which is semantically equivalent to performing them on scalar types.

The characteristic of these operations is that they have no side effect, they pop their inputs from the operand stack and push their results back to the operand stack.

Arithmetic operations

The **add**, **sub** and **mul** operations perform addition, subtraction and multiplication of two integers. They do not differ between signed and unsigned values, so we can use them even with uninterpreted integers.

The **div** and **rem** operations perform integer division and the remainder operations. The remainder operation calculates the remainder of a division, for example: $rem(11,3) = 2$

The division and remainder operations have variants for unsigned and signed semantics. These get converted into the same expression type, because CBMC stores the signedness with the type and not the operation. We therefore have to make sure that integers have the correct type, and if not we have to add a type cast before the operation.

The division and remainder operations have inputs that result in undefined values. In this case, the result will be a trap and the execution terminates. The first case happens when the second operand of the division and remainder operations is 0. The other case is when in a signed division the first operand is INT_MIN and the second operand is -1. The division $\frac{-2^{n-1}}{-1}$ results in 2^{n-1}, which is outside the value range. So we have to add an assertion before every division and remainder operation for the first case. If we have a signed division, we have to add an additional assertion for the second case.

Bit-manipulation operations include *and*, *or*, *xor*, left and right shift and left and right rotation. These operations interpret integers not as numbers, but as a list of bits. The *add*, *or* and *xor* operations construct the result by using boolean operations on each bit of both inputs. *add* performs conjunction, *or* performs disjunction and *xor* performs exclusive disjunction.

The left and right shift operations take the first operand and shift the individual bits to the left or the right. The amount of places to shift is determined by the second operand. One might think that the bit shift operations have a lot of cases where the result is not defined. That is because for example in C, if the second operand is longer than the bit width or negative, the behavior is undefined. [12, §6.5.7] Wasm bytecode resolves this problem by treating the second operand as modulo n, where n is the bit width. That means for example that a shift by -1 of a 32-bit integer performs a shift by 31 places. This implies that all results are well-defined and we do not have to place assertions.

In a left shift, the bits that get added from the right are zero bits. The right shift comes in two variants. The unsigned right shift or logical right shift fills the bits that get added from the left with zero bits. The signed right shift or arithmetic right shift fills the bits from the left depending on the leftmost bit of the original value, therefore keeping the sign.

The left and right rotation operations behave similar to the bit shift, but the bits that get removed from one side get added on the other side. Like the bit shift operations, the second operand is interpreted modulo the bit width, so all results are defined. All shifts can be directly translated into the expression types the GOTO language provides.

Unary integer operations take a single input and return a single result. There are three operations, **clz**, **ctz** and **popcnt**. The **clz** operation stands for *count leading zeroes* and counts how many of the first bits of an integer are 0. The **ctz** operation stands for *count trailing zeroes* and counts how many of the last bits are 0. The **popcnt** instruction performs the population count operation, it counts how many of the bits are set to 1. Even though there are no direct representations in C or Java for these operations, there exist expression types in CBMC for them.

Comparison instruction compare two values and output a boolean result. The result is always of type i32 and is either the value 1 if true and 0 if the comparison does not hold. There are 6 operators, they compare if the first operator is equal, not equal, greater, greater or equal, lesser and lesser or equal than the second operand. The equal and not equal operation work regardless of signedness, but the other instructions come in two variants each. In the GOTO language, there exists an expression class for all of them.

The remaining instructions are the **const** and **eqz** instructions. The **const** instruction pushes a static value on the operand stack. This can get converted into a constant expression. The **eqz** instruction returns 1 if the operand is 0, and 0 otherwise. This behavior is identical to the negation operator *!* in C. The GOTO language provides a *not* expression for this case. We have to be careful because the expression expects the input to be of type boolean. We therefore have to add a type cast from int to boolean.

The following instructions do not exist for i32 and i64 types, but can be performed on vector data that represents packed integers. Vector data gets explained more detailed in chapter 6.5. Because the following instructions work with packed integer data, it made sense to put them here.

The **abs** instruction calculates the absolute value for each lane, and there is an abs expression in the GOTO language, which we can use. The **neg** instruction performs lane-wise negation which is equal to multiplying by -1. We can convert it into the neg expression. Both of these instructions have an edge case where the input is *INT_MIN* and the theoretical result would be outside of the number range. In this case the operation is defined to overflow and the result would be *INT_MIN* and does not trap. Also both of the operations implicitly require the values to be signed.

Then there are lane-wise minimum and maximum instruction in both signed and unsigned variants. These can be transformed into a ternary if expression like this:

$$\min(a, b) = \text{if}(\text{less\_than}(a, b), a, b) \tag{6.2}$$

The next instructions we look at are lane-wise saturating addition and subtraction. In saturation arithmetic, there is no overflow, in this case the result will just be the maximum value. There is no corresponding expression in the GOTO language, and no operation in SMT-LIB that covers this case. As a result we have to implement the functionality ourselves.

The saturating operations return the maximum if the result of the operation would be greater than INT_MAX, in other words if the computation overflows, then the result will be INT_MAX. The same happens when the result is smaller than the minimum value, which would be an underflow, the result will be INT_MIN. We have to define the conditions for which an over- or underflow would happen.

Overflow in a signed addition of two numbers a, b happens when the following conditions are fulfilled:

$$a > 0 \wedge b > INT\_MAX - a \tag{6.3}$$

Similarly, underflow happens under the follwing condition:

$$a < 0 \wedge b < INT\_MIN - a \tag{6.4}$$

It seems easy to treat signed subtraction the same as signed addition with the second operand negated, but then we would run into a problem if the second operand happens to be INT_MIN. Signed subtractions of numbers a, b overflow when:

$$a > 0 \wedge b < a - INT\_MAX \tag{6.5}$$

The following condition leads to an underflow:

$$a < 0 \wedge b > a - INT\_MIN \tag{6.6}$$

Unsigned over- and underflows are a little bit easier. Unsigned addition of numbers a, b can only produce an overflow when:

$$b > INT\_MAX - a \tag{6.7}$$

Unsigned addition cannot underflow, and by the same logic unsigned subtraction cannot overflow. Unsigned subtraction of numbers a, b underflows under the following condition:

$$b > a \tag{6.8}$$

So depending on the operation we need to check for the conditions and return INT_MAX in the case of an overflow and return INT_MIN in the case of an underflow. Listing 14 shows how a transformation of the saturating signed addition may look like.

```
def add_sat_s(a: expr, b: expr) -> expr:
    return if(
        and(
            greater_than(a, 0),
            greater_than(b, minus(INT_MAX, a))
        ),
        const(INT_MAX),
        if (
            and(
                less_than(a, 0),
                less_than(b, minus(INT_MIN, a))
            ),
            const(INT_MIN),
            plus(a, b)
        )
    )
```

Listing 14: *Pseudocode transformation of add_sat_s*

The next instruction is the **avgr_u** instruction. This instruction takes two packed integers and computes the average for each lane. If the average is not an integer, the result will be rounded towards positive. This can be transformed into a chain of addition and division by 2. The tricky part is that first both operands need to be divided by 2 and then added together, otherwise the addition may overflow if the numbers are large, leading to the wrong result.

Probably the most obscure WebAssembly instruction is the saturating integer Q-format rounding multiplication **i16x8.q15mulr_sat_s**. The Q format is a fixed point format representing real numbers. Q format addition and subtraction can be achieved through normal saturating addition and subtraction, but multiplication is a bit more complicated. Wasm supports lane-wise multiplication of Q15 numbers stored as $i16$ values. The operation has the following formula:

$$\text{q15mulr\_sat}(a,b) = \text{sat}((a \cdot b + 2^1 4) >> 15) \tag{6.9}$$

The *sat* function represents the saturation – the result is clamped between the maximum and minimum value of the 16-bit signed integer. For the calculation the saturation function gets converted into a chain of if expressions, and the rest gets converted according to the formula.

6.4 Floating Point Data and Instructions

Another common primitive data type are floating-point numbers. Floating-point numbers represent real numbers, and the representation and behavior is specified in the IEEE 754 standard.

Wasm defines two floating-point data types f32 and f64. These are bitvectors of length 32 and 64 bits. The representation is equal to the float interpretation of bitvectors in the GOTO language.

Instructions for floating-point types overlap a bit with integer instructions. Since there is no unsigned floating-point arithmetic, there is always just one variant of every instruction. We can reuse the expression types for addition, subtraction, multiplication and division. This works because when CBMC transform the GOTO program into a clause for the solver, it checks the type of the values and when the type is a floating point number, it tells the solver to use floating point logic. The comparison instructions that compare two values can be reused in the same way.

Then there are many instructions native to floating point values. These mostly represent mathematical operations on real numbers, like the square root operation. The problem here is that these instructions do not have an existing expression type in CBMC. The reason is that neither C nor Java has these operations as language built-ins. Instead these operations are implemented using library functions. In C, there is the *math.h* header that defines a lot of mathematical functions. In Java, the *java.lang.math* class offers a lot of functionality.

C and Java code that performs these library calls can be checked by CBMC as well. This works because CBMC provides implementations of those library functions. For example the square root function in C could be implemented using an approximation algorithm, or by using inline assembly to perform the processor instruction directly, depending on the system and library. The CBMC implementation is written to work well with CBMC and automatically performs checks for example to assert that the input is positive. As a first option, we could do the same.

Even though Wasm has no standard library, we treat these instructions as if they were library calls and replace them with semantically identical functions that include only operations that we can transform.

Listing 15 shows an example of an implementation of the **f32.min** instruction.

```
(func $min (param f32 f32) (result f32)
    local.get 0       ;; first argument
    local.get 1       ;; second argument
    f32.gt            ;; if first > second
    if (result f32)
    local.get 1     ;; return second argument
    else
    local.get 0     ;; else return first argument
    end
)
```

Listing 15: *Wasm implementation of i32.min*

This could be one approach. Another idea is to actually model these as new expressions. Since CBMC transforms GOTO programs into an SMT formula, we can have a look if the solvers understand these operations natively. SMT-LIB2 is the standard interface for SMT solvers. [2] If we look at the FloatingPoint theory, we see that there are functions for square root, negation, absolute value, minimum and maximum. The Wasm instructions **floor**, **ceil**, **trunc** and **nearest** could be transformed into the same *fp.roundToIntegral* SMT-LIB2 function with a different rounding mode. This would only leave the **copysign** instruction without direct SMT support. This instruction takes two operands, and returns the value of the first operand with the sign of the second one. For this we would have to take the first approach and reimplement it in Wasm, but this should be trivial.

So there are two possible ways to handle this. The first is to reimplement the instructions as functions, and the second is to directly translate those into SMT instructions. The first approach would have the advantage of fitting in line with the way that CBMC and JBMC work. There would be no changes to the CBMC backend and it would work independently of the solver being used. The downside is that it would need twice the code because each instruction need to modeled in f64 and f32. With the second approach, we outsource the work to the solver which could be more performant. Also it is more in the spirit of Wasm since those instructions are direct operations and no library functions. But we need to make backend changes and we might need extra code for other types of solvers, for example SAT solvers.

Here is an overview of floating point operations and their representations in either CBMC expressions or SMT-LIB2 functions. For each of these instructions exist two variants, one for f32 and one for f64 values.

Unlike integers, floating point operations are well defined for all cases and do not trap. If a value is divided by zero, the result is either positive infinity or negative infinity, depending on the sign of the value and of the zero. If zero is divided by zero, the result is the NaN value, the same result happens when taking the square root of a negative number. There is also no real

concept of overflow, if a result of an operation is greater than the number range, which is very big, then the result will be an infinity.

6.5 Vector data and SIMD instructions

6.5.1 Introduction

The next data type we look at is the vector data type. Currently only one variant exists, the type **v128**. Vector data is a data type that is designed to hold a packed set of values of a smaller type. Special SIMD (single instruction, multiple data) instructions can be used to perform operations on those values at the same time.

These instructions interpret the v128 type as a specific shape *t*x*n*. A shape consists of of type *t* and a number of lanes *n*. For example the shape i32x4 consists of 4 lanes of i32 values. The underlying data is just a 128 bit vector, but the instructions operate on the data of a specific shape. The following shapes are possible: i8x16, i16x8, i32x4, i64x2, f32x4 and f64x2. **Packed data** is a general term when describing vectors that hold data that is interpreted in such a shape. There are also some instructions that interpret the vector as v128, these interpret the data either as a 128-bit integer or just as a bit pattern.

The vector type was added because one of the goals of Wasm is high performance. There are hardware registers and instructions on many modern CPU architectures that can do these kinds of operations. With these, the Wasm code can gain a lot of performance by doing multiple integer or floating-point operations in parallel.

To show an example, we have to compile the C code with the compiler option *-msimd128*, at least when using Clang.

```
1  void multiply_arrays(int32_t* out, int32_t* in_a,
2                       int32_t* in_b, int32_t size) {
3      for (int32_t i = 0; i < size; i++) {
4          out[i] = in_a[i] * in_b[i];
5      }
6  }
```

Listing 16: *Code that uses SIMD instructions for optimization*

Listing 16 shows a function that gets two arrays of integers as input. This code was taken from a blog post of the WebAssembly implementation in the V8 JavaScript engine. [9] The arrays are assumed to be the same size and for each element the numbers get multiplied and put into a new array. The resulting Wasm code is too long to put it here and explain it in detail. But in general, it tries to pack the integers into v128 types as good as possible. This is cheap, because in an array the integers are already stored next to each other in memory. This results in the costly multiplication instruction only being done once for four integers at a time. Instead of **i32.mul** the instruction **i32x4.mul** will be used.

6.5.2 Implementation considerations

To implement this in CBMC, first we need to think which type class we need. Since it is represented as 128 bits, it is clear that it should be modeled as a bitvector with a width of 128. This is the easy part, now we need to figure out how we treat it as a specific shape when instructions execute operations on that shape. For example one instruction may treat the vector as i16x8, and the next as i32x4. Whether or not it makes sense to do this does not matter, it would be an allowed piece of code.

The best way might be to leave it as an uninterpreted 128-bit bitvector whenever possible, and only cast it into the specific shape whenever needed. When we cast it to a shape, we need to cast it to a sequence of values. For example when we perform the instruction **i32x4.add**, the vector would need to be cast into 4 i32 integers to perform the individual operations. To do this, we can use the extractbits expression. This expression has 3 inputs, a source value, a start index and a target type. The result of the expression is a bit pattern of the target type, that was extracted of the source value at the start index.

```
1        # Conversion from v128 to i32x4
2        i32_1 = extractbits(vector, 0, i32)
3        i32_2 = extractbits(vector, 32, i32)
4        i32_3 = extractbits(vector, 64, i32)
5        i32_4 = extractbits(vector, 96, i32)
```

Listing 17: *Extractbits example*

Listing 17 shows how a conversion from v128 to i32x4 would look like in pseudocode.

To transform the shape back into 128-bit bitvector, we need to use the concatenation operator. The GOTO language supplies a concatenation expression type for that. That takes an arbitrary amount of input bitvectors and a target type, and concatenates them.

This gives us a manual how we need to transform most of the vector instructions that operate on a shape txn:

- Cast the m input vectors into sequences $S_1, ..., S_m$ of n values of type t
- For each $k \in \{1, ..., n\}$, perform the operation op on $S[k]$ of each sequence
- concatenate the results together

With that in mind, we can now provide an example transformation for the **i32x4.add** instruction:

Listing 18 shows how such an instruction can be transformed. Note that in this example *concat*, *plus* and *extractbits* are not functions, they are constructors for expression types. That means that whatever the Wasm program does with the resulting v128 like just putting it on the operand stack, or storing it in memory, we have to replace that with this huge tree of expressions.

This can even be generalized, since there are similar instructions that work on multiple shapes.

Listing 19 shows an example of such a general transformation. *result_list* gets initialised with an empty list, where the result of the operation is appended for each lane.

It is also important to remember that the typing does not differantiate between unsigned and signed integers. Some instructions come in an unsigned and signed variant where it is clear to

```
def transform_i32x4_add(
    vector1: expr(bv128),
    vector2: expr(bv128)) -> expr(bv128):
    return concat(
        [
            plus(
                extractbits(vector1, 96, int32),
                extractbits(vector2, 96, int32),
            ),
            plus(
                extractbits(vector1, 64, int32),
                extractbits(vector2, 64, int32),
            ),
            plus(
                extractbits(vector1, 32, int32),
                extractbits(vector2, 32, int32),
            ),
            plus(
                extractbits(vector1, 0, int32),
                extractbits(vector2, 0, int32),
            )
        ],
        bv128
    )
```

Listing 18: *Pseudocode transformation of i32x4.add*

see how the integer data needs to be interpreted. But some instruction, for example negation, implicitly expect the integer data to be signed, that is important to consider when extracting the lanes.

It is clear that when multiple vector operations are done in succession, the expression tree gets exponentially larger, since each input gets copied n times into the resulting expression where n is the number of lanes. It is possible to decrease the decrease the size of that expression tree by performing a reduction. When we look back at the example in listing 18, we notice that when we replace one of the input vectors with such an expression, then we have many occurences of the *extractbits* constructor where the first argument is a concatenation expression.

Listing 20 shows how such a construct may look. In this example *exprn* is placeholder for an expression that evaluates to an i32 integer. This piece of code concatenates 4 i32 expressions, and takes the first 32 bits of the result. It is clear that in this case the *extractbits* and *concat* operations are redundant, in this case the whole expression is equal to just *expr1*.

We can check for this reduction step anytime we place the *extractbits* constructor. This only works when the concatenated expressions and the bit extraction are of the same type. But chained vector operations can be exptected to target the same shape, so this might be the case most of the time.

```
        def transform_simd_binary_instruction(
                        vector1: expr,
                        vector2: expr,
                        shape_type: type
                        shape_lanes: int
                        op: expr_type) -> expr:

            result_list = []
            for i in shape_lanes - 1 .. 0:
                result_list.append(op(
                    extractbits(vector1, i * sizeof(shape_type), shape_type),
                    extractbits(vector2, i * sizeof(shape_type), shape_type)
                ))

            return concat(result_list, bv128)
```

Listing 19: *Pseudocode transformation of general binary SIMD instruction*

```
        extractbits(concat([expr1, expr2, expr3, expr4], bv128), 0, i32)
```

Listing 20: *Reduction opportunity*

It is possible that this type of reduction may be done by the solver as well so it does not have any performance gains, but it still can reduce the size of some expression trees and help with performance and memory usage.

6.5.3 Vector instructions

Now we look at individual instruction and how it is possible to transform them into parts of GOTO programs. Many of those instructions have the pattern ***shape.op*** where *shape* is a shape of packed integers or floating point numbers and *op* is an operation that is already covered in chapter 6.3 or in chapter 6.4. These instructions get transformed using an algorithm described in the previous section.

In this section I want to cover the following types of instructions:

- Instructions that operate on v128 directly
- Instructions that convert between packed types

v128 Instructions

These include the **v128.const** instruction, which can be treated the same as the other constant integer instruction. The bitwise instructions **v128.and**, **v128.or** and **v128.xor** also work the same as corresponding integer instructions, just with a longer bit width. The instruction **v128.andnot** can be transformed into a bitwise *and* expression where second operand is run through a bitwise *not* expression. The **v128.bitselect** instruction takes 3 v128 operands. For each bit, if the third

operand is set, take the bit value of the first operand, otherwise the bit value of the second operand. This operation can be translated into a chain of logic operators, this is taken directly from the WebAssembly specification:

$$\text{bitselect}_N(i_1, i_2, i_3) = \text{or}_N(\text{and}_N(i_1, i_3), \text{and}_N(i_2, \text{not}_N(i_3))) \tag{6.10}$$

The instruction **v128.any_true** returns an i32 constant value of 1 if at least one bit is set, otherwise zero. This is semantically equal to comparing against a 128-bit 0 number, and negating the result.

Vector type conversions and operations

Wasm offers many instructions that convert between different packed types and between packed and non-vector types. There are also some operations where the shape of the input differs from the shape of the output, these are included here as well.

It is possible to extract and replace single lanes in a packed type. The instructions are called ***shape*.extract_lane** and ***shape*.replace_lane**. Both of these instructions take a static argument which is the index of the lane that will be extracted or replaced. The value to replace, or the result of the extraction, is taken from or pushed to the operand stack. It is easy to see that the **extract_lane** instruction can be transformed into the extractbits expression. The **replace_lane** instruction can be tranformed into a chain of concatenation operations, where all elements are extracted from the original vector except the one to be replaced, where we take the value from the stack.

Another way of converting a scalar type to a vector is with the introduction ***shape*.splat**. This consumes a scalar value from the stack and transform it into a vector where each lane is given that value. We can emulate this behavior in our program by using the concatenate expression.

The instruction **i32x4.dot_i16x8_s** performs lane-wise multiplication on i16x8 values. The result is sign-extended to 32 bits and adjacent lanes are added together. Listing 21 shows how this transformation can look. The transformation is only shown for the least significant 32 bits of the result to show the general idea, the rest is done in similar fashion and the four parts are concatenated to the result.

```
1         add(
2             mult(
3                 cast(extractbits(vector1, 0, i16), i32),
4                 cast(extractbits(vector2, 0, i16), i32)
5             ),
6             mult(
7                 cast(extractbits(vector1, 16, i16), i32),
8                 cast(extractbits(vector2, 16, i16), i32)
9             )
10        )
```

Listing 21: *Pseudocode transformation the i32x4.dot_i16x8_s instruction*

The instruction ***shape1*.narrow_*shape2*** takes two vectors of shape *shape2* and produces a new vector of shape *shape1*. The result vector is constructed by taking alternating lanes form the first and second input. To fit these into one vector, the bit width of *shape1* is half of *shape2*, so the individual lanes will be narrowed to half the bit width. This narrowing is saturating, because of that there are signed and unsigned variants of the instruction.

The instructions ***shape1*.extend_low_*shape2*** and ***shape1*.extend_high_*shape2*** where *shape1* and *shape2* are integer shapes extend one half of a smaller lane vector into a vector where the length of a lane is twice as many bits. This extension can either be signed or unsigned. The conversion can be made by extracting each lane of half the vector, extending using a type cast and concatenating the results.

Extended multiplication performs lane-wise multiplication of either the first or second half of the vector and extends the result so no overflow can happen. The name of the instruction is ***shape1*.extmul_low_*shape2*** and ***shape1*.extend_high_*shape2*** depending on which half of the input vector gets used. This can be converted by extracting each lane of half of the vector, extending them with a type cast. Then a lane-wise multiplication is performed and the results are concatenated. There are signed and unsigned variants of this because the number can be zero-extended or sign-extended.

The instruction ***shape1*.extadd_pairwise_*shape2*** takes a single vector as input. It extends lanes into wider lanes by adding neighboring lanes together. The extension can be either zero-extended or sign-extended. Again this can be implemented using extracbits, type cast, addition and concatenation expressions.

There are also normal conversions between integer and floating-point shapes. The instruction **i32x4.trunc_sat_f32x4** converts lane-wise each $f32$ into an $i32$ integer, either signed or unsigned. The conversion is saturating if the float value is ouside the integer range. Rounding gets done towards zero. Similarly the instruction **i32x4.trunc_sat_f64x2_zero** converts from $f64$ to $i32$. The two result lanes are the low lanes, the remaining integer lanes are zero. These instructions can be converted into type conversions and if-expressions.

6.6 Linear memory

6.6.1 Introduction

An important factor of a program and its logic is how it uses memory. Memory refers to the portion of main memory from a computer that is reserved for that program. Many language runtimes seperate between a stack that stores local variables, and a heap that stores complex data. Objects in the heap are pointed to by some kind of pointer or reference.

Memory and especially heap memory is a very common source of bugs, especially in languages that use manual memory management. Trying to read an invalid address can lead to a segmentation fault, causing program termination. Continously requesting memory without freeing can cause the machine to run out of memory. Other types of memory errors can cause unspecified program behaviour, like using memory after it has been freed.

6.6.2 Linear Memory Model in WebAssembly

Since Wasm is designed to be compatible with various programming languages, it needs a very flexible approach to memory. Local variables are stored normally within the function stack frame. The heap is modelled very simply, it is an untyped linear array of bytes.

Available memory needs to be defined in the module using units of page size. This page size is equivalent to 65536 bytes, since Wasm designers found this to be the lowest common page size on modern hardware. [10] The Wasm code can access all of the memory available.

There are load and store instruction for all the data types, that move data between the linear memory and the operand stack. Instead of a pointer data type, memory gets accessed using an *i*32 index, with the first byte of memory starting at 0. Therefore the memory can be thought of a global byte array. Currently, only 1 memory per module is allowed, so every memory access implicitly accesses memory with index 0. In future versions more than one memory per module can be declared.

Wasm has a very strict approach to avoid undefined behavior regarding memory. When a Wasm module gets instantiated, all the available memory gets initialised with zero bits. And each memory access gets checked at runtime if it is within bounds.

Now we need to think how we translate memory access to GOTO programs. The downside of Wasm's memory model is that we can never free any memory. Each section of available memory is always allocated and valid. It is possible for a Wasm program to write to a section in memory at the beginning and to read that section somewhere at the end of the program. The upside is that we do not need to use logic to keep track which section of memory is valid and which has been freed.

To explain how we proceed we use the code example from listing 22.

```
1   (func $mem_test (result i32)
2       i32.const 5
3       i32.const 42
4       i32.store    ;; store the i32 42 at memory index 5
5       i32.const 5
6       i32.load     ;; load the i32 at memory index 5
7       return       ;; should return 42
8   )
```

Listing 22: *Store and load example*

Here we have an example usage of the **i32.load** and **i32.store** instructions. The **i32.store** instruction takes two inputs, the first is the value we store in memory, and the second is the index. Remember that the operand stack is a stack, so the value we last pushed on it is the first value we take from it, which means that in this case we store the integer 42 at memory index 5. Then we load the value back from memory with the **i32.load** instruction, which consumes the memory index from the stack.

We can reuse some ideas from chapter 6.2.3. We can assume that we need to translate the store instruction into some kind of assignment, and the load instruction into some kind of expression.

The first idea is to have a similar data structure like we have for local variables. For example we could make an indexable list of memory locations that we have used.

In this case, we would write a new entry at memory index 5, where we place whatever expression we took from the stack, in this case the constant literal expression with the value of 42. We cannot store the value directly, because it could be something else that we cannot evaluate during static analysis.

This approach has a critical issue, which we explain with another example.

```
1     (func $mem_test (result i32)
2         i32.const 5
3         i32.const 949679958
4         i32.store
5         i32.const 6  ;; important change
6         i32.load
7         return       ;; what does it return?
8     )
```

Listing 23: *Store and load example*

The code in listing 23 is very similar to the previous example with two key differences. First, we store a bigger value in memory and second, we read from memory index 6 instead of index 5. This raises the question: What value does the function return? At first glance, one might assume it could be 0, because all memory is initialised with 0 and we have not written to memory index 6 yet. However, this is not the case. An $i32$ value consists of 32 bits which means it has a length of 4 bytes. It is important to remember that memory in Wasm is an array of uninterpreted bytes. So if we write an integer to memory index 5, it starts at memory index 5, and the whole integer gets written to index 6, 7 and 8 as well.

Wasm stores multi-byte values into memory in little-endian order. That means that lowest significant bits get written into the first byte, which is the reverse way we usually think about numbers where the most significant digits usually are on the left. With that knowledge we are able to work out what our function returns. First, we convert our value into hexadecimal, which helps us because 2 hexadecimal digits equal 1 byte. 949679958 into hexadecimal equals 0x389AF756. With the knowledge that bytes are stored in little-endian order and the fact that bytes get initialised with zero, we can visualize the memory.

Memory index	5	6	7	8	9
Value	56	F7	9A	38	00

Figure 6.2: *Memory detail*

Figure 6.2 shows how each byte from memory index 5 to 9 is set at the end of the function. An access to memory index 6 would give the value 0x00389AF7 which is 3709687 in decimal. When we run the program, this is exactly the value that gets returned.

This example shows us that we need to treat each byte of memory individually. In the last example we would set each byte from index 5 to 8 with a value. But the question now is to which value do we set it? We cannot use a numeric value because in other cases it might not be a constant value that we can statically determine. Any store instruction consumes an expression from the stack, and we would need to take that expression and extract each individual byte. This can be done in CBMC possibly with the extractbits expression. The type constructor accepts a source expression, an index and a target type. The source expression would be the expression we take from the stack, the index would be 0 for the first byte, 8 for the second and so on, and the target type would be an uninterpreted 8-bit wide bitvector.

To model the memory in GOTO programs, we need an array that is defined in global scope. The size of the array is the size of the memory in byte. The data type in uninterpreted bitvector with a width of 8 bits.

During Instantiation all available memory gets initialised to 0. We could add this to our GOTO program at the start, but that would create significant overhead. Many applications use only a fraction of available memory, and initialising everything to 0 would be a needless task. Also initialising the memory is more of a safety feature than a language feature. Most code should not rely on the fact that the memory has been initialised, it should generally avoid unwritten portions of memory.

In CBMC, accessing an array element that is not initialised creates an undeterministic choice: The element can be of any value possible, similar to a program input. That means solver can choose any value to satisfy the condition it tries to prove. This can be also an option, especially if the memory is being imported. Maybe it would make sense to activate memory initialisation using a command-line argument.

6.6.3 Memory instructions

Memory instructions are all instructions directly related to memory. In this section I will show different memory instructions and how they can be translated into GOTO programs.

The most important instructions are load and store instructions. Each load and store instruction can have two optional static parameters, an offset and an alignment specifier. The offset gets added to the index, so for example if an instruction reads memory at index 10 with an offset of 4, the index that will be read is 14. The alignment specifier has no semantic meaning, but it can help an interpreter or JIT-compiler make optimizations about the alignment of memory access. We can ignore it.

The load instruction has the notation ***t*.load** where t is any number type. It takes an $i32$ input from the operand stack and reads the data at that index in memory, plus an optional offset. The result of type t will be pushed to the operand stack.

To convert a load instruction we need to read each byte seperately, because we can only access one element of the memory array using the index operator. Each byte then gets concatenated into the target type. Listing 24 shows an example transformation of the **i32.load** instruction. It

needs to be emphasized that the memory order is little-endian, but the concat operation accepts the most significant operand first, so the indexing needs be reversed.

```
        def transform_i32_load(
            index: expr(uint32)
            offset: uint32
            ) -> expr(int32):
            return concat(
                [
                    memory[add(index, offset, const(3, uint32))],
                    memory[add(index, offset, const(2, uint32))],
                    memory[add(index, offset, const(1, uint32))],
                    memory[add(index, offset)]
                ], int32
            )
```

Listing 24: *Pseudocode transformation of i32.load*

There exist special load instructions that load short integer types from memory and extend them into larger ones. The notation is ***t*.load*n*** where t is the target type and n is the bit width of the integer that is read from memory. Since the value gets extended into a bigger type, there is a difference regarding signed and unsigned valus. To convert this class of instructions, we need to surround the extraction with an appropriate type cast.

The store instructions have the form ***t*.store**. They consume an $i32$ index and a value of type t from the operand stack and store it in memory. This needs to be converted into GOTO code where each byte of memory gets assigned in an individual statement.

```
    ASSERT not(greatereq(
        add(index, offset, const(3, uint32)),
        mul(memory_size, const(65536, uint32))
        ))
    ASSERT not(less(add(index, offset), 0))
    ASSIGN memory[add(index, offset)]
            := extractbits(value, 0, bv8)
    ASSIGN memory[add(index, offset, const(1, uint32))]
            := extractbits(value, 8, bv8)
    ASSIGN memory[add(index, offset, const(2, uint32))]
            := extractbits(value, 16, bv8)
    ASSIGN memory[add(index, offset, const(3, uint32))]
            := extractbits(value, 24, bv8)
```

Listing 25: *Transformation of i32.store into GOTO code*

Listing 25 shows an example of such a transformation. $value$ is the value or expression that gets stored in memory, $index$ is the index and $memory$ is the name of the global memory array. At the beginning I have inserted 2 assertions, these perform a bounds check and report a failure

if the index is out of bounds. For that, we use the global variable $memory\_size$, which holds the current size of the memory.

Similar to load instructions, store instructions of integers can store a smaller integer type in memory. These kind of store instructions discard the most significant bits of the larger type. To convert this, it is not necessary to type cast, we just need to extract the relevant bytes.

The instruction **memory.size** returns the current size of available memory in the unit of page size. That requires us to hold a global variable that records the size of the memory in units of page size. When we convert this instruction, we just access it. Whenever we need the value in bytes, it just gets multiplied by 65536.

The instruction **memory.grow** consumes an $i32$ parameter from the stack and increases the available memory size by the amount of that value in unit of page size. In the background, the runtime allocates more memory and zero-initialises it. We need to resize the memory array. For this we need to redefine it and copy the contents. Then we increase the memory size global variable. Since we read this variable during bounds checks, these will be always correct. If we take the approach of memory initialisation, we need to initialise this new memory as well.

The **memory.fill** instruction fills memory sections with a single byte. It takes 3 $i32$ values as arguments. The first is the index that determines the first index of that memory section. The second is the value these memory bytes will be filled with. It has type $i32$, but it will be truncated to an 8-bit value. The third argument is the length of that memory section.

To convert this into a GOTO program, we need to convert it into some kind of loop. Each memory byte will be assigned using an assignment statement. The loop will be modeled using a GOTO statement. Listing 26 shows how it may look like. The variable $value$ is the value the bytes get filled, $index$ is the start index and $length$ is the length of that memory segment. The length will be decremented each loop iteration. If the length is 0, the loop will terminate.

```
1      ASSERT not(greater(
2          add(index, length),
3          mul(memory_size, const(65536, uint32))
4          ))
5      ASSERT not(less(index, 0))
6      DECL fillvalue: bv8
7      ASSIGN fillvalue := extractbits(value, 0, bv8)
8  1:  GOTO 2 IF equal(length, const(0, uint32))
9      ASSIGN memory[index] := extractbits(value, 24, bv8)
10     ASSIGN length := minus(length, const(1, uint32))
11     ASSIGN index := plus(index, const(1, uint32))
12     GOTO 1
13 2:  (...)
```

Listing 26: *Transformation of memory.fill into GOTO code*

The **memory.copy** instruction copies data from one segment in memory to another. The segments can be overlapping. It takes in three $i32$ arguments, the first one is the starting index of the destination segment. The second one is the starting index of the source segment and the third one is the amount of bytes that get copied.

To convert this, we can check if the memory segments overlap. If they do not, then we could assign the target memory from the source memory directly. Otherwise, the data can be copied into a temporary array, and then into the target segment.

6.6.4 Data segments

Usually memory gets initialised as with zero values, but modules can define data segments. Data segments consist of values that a specific portion of memory gets initialised with.

They can either be active or passive. Active data segments get copied into memory during module instantiation, while passive data segments get copied into memory with the **memory.init** instruction.

Converting the **memory.init** instruction adds assignments to the GOTO program. Each memory element gets filled with the value from the data segment.

Active data segments need to be copied into memory at the beginning of the GOTO program.

6.7 Structured code flow

6.7.1 Introduction

Any meaningful code cannot only contain a linear list of instructions that get executed in order. To model real-world use cases code needs to perform conditional logic or loops. In one scenario it needs to perform one block of code, and in other scenarios it needs to execute one other block of code. In other situations one may write a block of code that gets executed n times, with n not known at compile time.

Therefore any language needs to have constructs that help to branch control flow. On assembly language level this is most often specified as a relative address to the current instruction pointer. Higher-level languages have introduced code constructs that abstract this process into a more human-understandable form in the form of structured blocks. Most prominent examples include the *if condition then ... else ...* construct and the *while condition do ...* loop, where ... can be substituted by any block of code, which has a defined end point. Programming with these constructs is known as structured programming.

In model checking, we need a way to abstract these construct into a form that a solver can understand and solve. CBMC helps us with the last step, but we need a way to lower all the code into a GOTO program at first. Since guarded GOTO statements are the only way in GOTO programs to perform any sort of branching by definition, all the branch instructions and constructs need to be transformed into the GOTO instruction.

6.7.2 Control-Flow Instructions in WebAssembly

First we need to explore the possibilities Wasm offers in regards to branching and loops. For this, we cannot look at each instruction individually, since they are intertwined and work together. Suprisingly, there is no kind of *goto* instruction, even though Wasm is intended as a low-level bytecode close to machine level, where goto is just a direct jump to another instruction. Wasm is more structured than traditional assembly language, and offers constructs to define blocks. Then there are several branching instructions that break out of blocks.

The 3 instructions to define blocks are the **block**, **loop** and **if** instructions. The **block** and **loop** instruction define a block of code until a corresponding **end** instruction. The **end** instruction does not do anything, it just serves as an informational marker to end the block. It can be even questioned if it should be called an instruction, but it is defined as one and has an opcode. The **block** and **loop** instruction are technically identical when they are standalone, but the difference comes when we introduce branch instructions. Blocks can be nested, so that the outermost **block** or **loop** instruction corresponds to the outermost **end** instruction. Validation ensures that all blocks are well-defined.

The **if** instruction defines 2 blocks of code with corresponding **else** and **end** pseudo-instructions. The **if** instruction consumes an i32 value from the operand stack and tests if it is equal to 0. If it is not, then control flow executes the block between the **if** and **else** instructions, and after that continues after the **end** instruction. If the value is 0, then the block after the **else** instruction is executed. This makes it very similar to the higher-level *if condition then ... else ...* construct found in many programming languages.

Listing 27 shows an example code using the **if** statement. In the second line the code takes the decision, depending on the value of a local variable. If it is nonzero, then the code block from lines 3 to 5 gets executed, otherwise the code from line 7 to 9.

```
1   local.get 1
2   if (result i32)
3     local.get 0
4     i32.const 5
5     i32.add
6   else
7     local.get 0
8     i32.const 10
9     i32.add
10  end
```

Listing 27: *Example code using if ... else*

There are 4 branching instructions, **br**, **br_if**, **br_table** and **return**. When these instructions are executed, the control flow can break out of the current block. Blocks can forward- or backward-jumping.

Inside a block defined using the **block** and **if** instructions, control flow after branching continues after the block. This is called a **forward-jumping** block. The branching instruction can be thought of as a C-style *break* statement. Inside a block defined using the **loop** instruction, control flow after branching continues at the beginning of the block. This is called a **backward-jumping** block. In this case, the branching instruction can be thought of as a C-style *continue* statement.

Branching instructions need to explicitly define how many layers of blocks they break out of. Instead of using labels, a number needs to be specified after the instruction. 0 means that the branch only breaks out of its current block. Each number higher increases the breaking to one more level.

Listing 28 shows an example. Line 5 shows a branching instruction, that breaks out of 2 levels of nesting. I have annotated each block with the level of nesting, as seen from the perspective of

the branching instruction. It clearly highlights that in this case the code flow would continue in line 9.

```
1   block          ;; level 3
2     block        ;; level 2
3       block      ;; level 1
4         block    ;; level 0
5           br 2   ;; branching instruction
6         end      ;; level 0
7       end        ;; level 1
8     end          ;; level 2
9     (; ... ;)    ;; code continues here after branching
10  end            ;; level 3
```

Listing 28: *Example code showing a branching statement*

The **return** instruction does not need a level label, since it implicitly breaks out of the outermost block. The outermost block in all cases is the function body, which means the instruction ends the function, returning the operands that are left on the stack. The **br** instruction performs an unconditional branch. The **br_if** instruction consumes an i32 operand, and performs a branch if it is nonzero. The **br_table** instruction consumes an i32 operand, that acts as an index to a list of level specifiers, and performs a branch to that target.

When a branching statement gets executed, the stack needs to be unwound to look like at the beginning of the target block. This is because when breaking out of a block, the values that instructions from that block have pushed to the stack are not going to get needed.

This behavior is similar to a function call. Each block has its own stack, and can consume inputs or produce outputs. To indicate to the runtime that a block has inputs or return values, the block needs a signature similar to a function signature. This signature maps any number of input values to any number of result values. An example for this is in listing 27, where both code blocks in the blocks described by the **if** and **else** statements add one value to the operand stack. Because the code block changed the stack, it needs to have a signature indicating that the block adds one $i32$ value to the stack.

When the control flow gets to the start of the block, it pops its inputs off the current stack, creates a new stack and pushes those on to it. After the execution of the block is finished because it reached the **end** statement or a branch instruction, the result values get popped off the stack. The new stack gets deleted, and the result values get pushed onto the original stack. If a branching statement gets executed in a loop, the control flow jumps to the beginning of the loop. In this case, the inputs of the block need to be on the operand stack before the branching instruction, not the results.

6.7.3 Conversion into the GOTO language

Conversion of blocks and branches

We can now work out how we can represent these constructs in GOTO programs. We have to take into account that blocks and branching statements do not only change control flow, but they

manipulate the operand stack as well. Since the **br** instruction just jumps to a defined location, we can replace with a *goto* statement. Likewise, the **br_if** instruction can be represented by a conditional *goto* statement. The condition is taken from the operand stack, so it has to be the result of the last operation.

Now the question is where these *goto* statements jump to. We know that in a normal block control flow jumps to the end of the block, and in a **loop** block it jumps to the beginning of the block. Since all blocks are well-nested, we can keep track of that.

During GOTO conversion, we go through the code twice. We start at nesting level zero, which is the body of the function before any block definition. Then we go through the function one instruction at a time. We keep track of any instruction that starts and ends a block. Each time we enter a block, the nesting level increases by one, when we leave one, we decrease it by one.

In the second run, we look at the branching instructions. Since we kept track of where blocks start, end, and what nesting level they have, we can see where control flow continues after branching. Each branching instruction has an identifier n that declares how many levels of block the instruction breaks out of. $n = 0$ indicates jumping out of the current block, so we can that the block we jump out of for any branching instruction is the current nesting level minus n. If the block is forward-jumping, the target of the GOTO is the end of the block, if the block is backward-jumping, the target is the beginning of the block. We now have all the information to successfully lower the branching instruction into GOTO statements.

Listing 29 shows an example involving structured code. Every block and branching instruction is annotated with comments after the double semicolon. Two blocks are defined, one that is forward-jumping in line 5 and one that is backward-jumping in line 6.

The branching statement in line 10 performs a conditional branch by one level. Since we passed two block definition instructions, we are at that point at nesting level 2. The branch therefore breaks out of the block with nesting level 1, which is a backward-jumping block. The code flow continues after line 24.

The branching statment in line 22 performs an unconditional branch by zero levels, which means out of the current block. The current block is a backward-jumping block, so the control flow continues after line 6.

The next thing we have to think about is the unwinding of the stack. Detailed in section 6.2.2, we keep the stack as an internal data structure, in which we store expressions.

During symbolic execution, we can mark specific points on the stack. If we encounter the start of a block, we look at the block type and see if the block consumes any inputs. We know that these values will be consumed by the block, so we mark the point of the stack below the input operands. If a block has no input values, we mark the top of the stack.

This is the point to where we unwind to if the block ends. We also keep track of the nesting level, so we know how many levels of stack to unwind.

When we get to a branching statement, we determine what type of block we break out of. If we are in a forward-jumping block, we look if the block has any result types. If we are in a backward-jumping block, we look if the block has any input types it consumes. We pop these off the stack and remember them. Then we unwind the stack to the mark that is equal to the nesting level we jump to. After unwinding, we push the result or input expressions that we remembered back on our symbolic stack.

```
(func $f (param i32) (result i32)
(local i32)
i32.const 0
local.set 1
block  ;; start of block with nesting level 1
  loop  ;; start of loop block with nesting level 2
    local.get 0
    i32.const 2
    i32.lt_u
    br_if 1 ;; conditional branch
    local.get 0
    i32.const -1
    i32.add
    call $f
    local.get 1
    i32.add
    local.set 1
    local.get 0
    i32.const -2
    i32.add
    local.set 0
    br 0 ;; unconditional branch
  end
end
local.get 0
local.get 1
i32.add)
```

Listing 29: *Complex example for structured Wasm*

Conversion of the *if* statement

The **if** ... **else** ... **end** construct can be transformed into a *goto* statement as well. These also count as forward-jumping blocks, can contain branching instructions and consume or output values to the stack. Therefore we have to treat them like blocks and take everything from the previous paragraph into account. The **if** instruction works as a conditional *goto* statement as well. It consumes the last operand on the stack, and if is zero, it jumps to the **else** instruction. If not, it will execute the instructions in the block following the **if**, until the **else** instruction. So we have to add an additional *goto* statement right before the **else** instruction, that does an unconditional jump to the end of the block.

Conversion of the *br_table* statement

The last instruction we have not analyzed yet is the **br_table** instruction. This has been included because there is no unstructured *goto* in Wasm. C and derivative languages offer the *switch*

construct. This matches a value against a list options. The insteresting property about it is that every case needs to be ended with an explicit *break*, otherwise the control flow *falls through* to the next statement, even if it did not match the input. Therefore the *switch* statement cannot just be modeled with if and else, because there is the possibility to *fall through* to the next block.

To understand it better we look at the example C code in listing 30.

```
1  int switch_test(int a, int b) {
2      switch(a) {
3          case 0: b += 2; break;
4          case 4: b += 1;
5          case 7: b += 8; break;
6          default: b += 1;
7      }
8      return b;
9  }
```

Listing 30: *Switch example in C*

When we compile this into WebAssembly using Clang, we get the code shown in listing 31.

We see that the switch statement got replaced with four nested blocks. In the innermost block, the control flow quickly reaches the **br_table** instruction at line 8. This consumes an operand from the stack as input and uses it as an index into a list of values. The last value is always the default one, in case the index is out of bounds. The value that is getting indexed is the level of blocks the control flow out of, like in a normal **br** statement. There are 9 values because a value is needed at index 0, 4 and 7. This gives 8 indexable values plus 1 default.

We notice that after each block, the logic of one of the case statements is performed. I have annotated that with a comment before each part. At line 14, 26 and after the code finished at line 31 the functions returns. After line 20 there is no return, and control flow continues. This corresponds to the fall-through effect in the C code. The code block starting at line 16 is the block we reach when we break out by level 1, which corresponds to index 4 in the lookup table of the br_table instruction, which corresponds to the case-clause in the C code with no break.

So how can this be converted to a GOTO program. We just replace it with conditional GOTO instructions, one for each case. In our example that would be 9 GOTO instructions. 8 handling the cases when a is in each of $[0;7]$, and one for any other value. The targets of the GOTO statements have to be determined the same as any other branching instruction.

This is same way that CBMC uses when converting switch statements in C. It converts each case into its own code block, and the logic which block gets executed when is done using conditional *goto* statements.

6.8 Type conversions

We have four numeric non-vector data types, and Wasm offers instructions to convert values from one to any other, and even some more. The GOTO language provides us with a type cast

```
(func $switch_test (type 1)
    (param i32 i32) (result i32)
block
  block
    block
      block
        local.get 0
        br_table 0 3 3 3 1 3 3 2 3
      end
      ;; case 0: b += 2; break;
      local.get 1
      i32.const 2
      i32.add
      return
    end
    ;; case 4: b += 1;
    local.get 1
    i32.const 1
    i32.add
    local.set 1
  end
  ;; case 7: b += 8; break;
  local.get 1
  i32.const 8
  i32.add
  return
end
;; default: b += 1;
local.get 1
i32.const 1
i32.add)
```

Listing 31: *Converted Wasm code*

expression. This expression does type conversions in the backend like a cast operation in C would do. We need to see if this fits every case, or if we need to make changes and workarounds.

6.8.1 Integer to Integer

There are three instructions that convert between i32 and i64 variables.

The first one is **i32.wrap_i64**, this converts an i64 to an i32. The conversion is straightforward, it cuts off the first 32 bits and keeps the rest the same. This is the same behavior in C, so we can simply convert it into a type cast expression.

The instruction **i64.extend_i32_u** extends an unsigned i32 to an i64 by adding zeroes. On the other hand, the instruction **i64.extend_i32_s** extends a signed i32 to an i64 by keeping the sign.

Because integer values can be ambiguous, we have to make sure to convert them to the right signedness before performing the type cast. But other than that the behavior is the same as in C, and we do not need to add anything else.

Then there are instructions of type ***t*.extend*n*_s**. These instructions take a value of type *t* and sign-extend the last *n* bits to the whole length of the type. *n* can be 8, 16 or 32 as long as it is shorter than the target type. These instructions are helping to support arithmetic of shorter bit lengths within bigger types. These can be represented by a double type cast, first to the shorter signed type, then to the signed version of the original type.

6.8.2 Float to Float

The instructions **f64.promote_f32** and **f32.demote_f64** convert data between both floating-point types. These should behave according to the standard and will be handled correctly by a normal type cast.

6.8.3 Integer to Float

Conversions from integer to float are done using the ***t*.convert_i*n*_*s*** instruction. *t* denotes the float type, *n* the length of the integer type and *s* if it is signed or unsigned. All these combinations make for 8 individual instructions. This should also be handled properly by a standard type cast.

6.8.4 Float to Integer

Conversions from float to integer types can be made in two ways. The first is a normal conversion. This would try to find an integer value that is equal to the float value rounded towards zero. If the float value is outside the number range of the integer type or an infinity or NaN, then the operation results in a trap. The second way is a saturated conversion. This ensures that the operation does not trap. It simply finds the nearest integer value that is possible, for example positive infinity would result in *INT_MAX*. NaN gets converted by definition to 0.

The normal conversion instruction has the form ***t*.trunc_f*n*_*s***, with *t* being the integer type, *n* the width of the float type and *s* the signedness specifier for the resulting integer type. We have to add checks because if the float value cannot be represented as an integer. First we add an assertion to check that the value we convert is greater than or equal to the float representation of *INT_MIN*. We also have to assert that the value is smaller than or equal to *INT_MAX*. These two assertions also check for NaNs, since every comparison to a NaN is false per definiton. If the assertions do not cause verification to fail, then a normal type cast is sufficient.

The saturated float-to-int conversion has the form ***t*.trunc_sat_f*n*_*s***, with *t* being the integer type, *n* the width of the float type and *s* the signedness specifier for the resulting integer type. Like the normal conversion, this results in 8 individual instructions. Instead of adding assertions, we need some conditional expressions to cover all the cases. We can use nested if expressions to handle all the cases. If the value is higher than *INT_MAX*, then the result is *INT_MAX*. Otherwise, if the value is lower than *INT_MIN*, then the result will be *INT_MIN*. The comparisons will have to be done in the floating-point type, and for the integer values we can return constant expressions. Lastly, we have to check for NaN, and if it is NaN, then the result is 0. We can use the isNaN expression for this. If none of the conditons were satisfied, we perform a normal type cast.

6.8.5 Reinterpretation cast

Between an integer and a floating-point value of the same bit width we can perform a reinterpretation cast. This means that we do not change the data underneath and just interpret the bit pattern as the target type. In C, this is not a native operation and usually gets achieved by a pointer cast. We could transform this into a sequence of address-of expression, type cast expression and dereference expression, but it would be a very unelegant solution.

A better way is to cast to a generic bitvector, which is possible in CBMC. We then cast the generic bitvector into the target type.

6.9 Reference data

References are a data type in Wasm that references another object. It is intended to be a safe alternative to pointers.

A reference can either point to valid object or be null. To ensure reference safety, references can not be casted from or into other types. This also means that reference cannot be stored in linear memory, because then it would be possible to reinterpret the data. Therefore Wasm defines a data structure called a table that can hold references.

Wasm 2.0 supports two types of references: Function references and external references. Function references are references to functions. They are used whenever functions need to be as a first order type, for example passing functions as parameters. External references point to objects owned by the embedding environment. Wasm cannot interact with them directly, but they can be for example passed as an argument to an imported function.

There is a special value for references which is a null reference. A null reference can be created with the instruction **ref.null**.

References can be stored in tables. Tables are a replacement for memory, because references are an opaque data type and are not able to be casted into any other type. If references could be stored in memory, then it would be possible to obtain their bit pattern, cast them into other types and do things like pointer arithmetic. That is the reason why tables exist, and similar to memories it is possible to read and write references from and to them. A detailed look into tables and table instructions is given in section 6.9.4.

The interesting question here is to what type we convert references to in our program. We cannot use any existing CBMC type because any reference-like data type in the CBMC framework (like those that are used to model Java or C++ references) inherits from the pointer data type, which is modeled as a bitvector. In Wasm, we cannot obtain the bit pattern of references, so it does not make sense to model them as a bitvector. The best way seems to be to create a new type for Wasm references, and subtypes for each type of reference.

6.9.1 Function references

The data type for function references is **funcref**. Function references are needed whenever functions need to be called indirectly. Instead of a direct function call, the function can be called through its reference. This makes functions indirectly first-class values and it is for example possible to create functions that take other functions as input.

Function references can be created with the **ref.func** instruction. The instruction takes a single static argument, which is the index of the function that will be referenced. The resulting reference is then pushed to the stack.

Functions can be called through its references with the **call_indirect** instruction. This instruction has two static parameters, and one it takes from the operand stack. The first static parameter is the index of the table that holds the reference. A Wasm module can declare multiple tables, starting from index 0. The second static parameter is the index of the function type. This ensures type safety. The operand that gets taken from the stack is an $i32$ index into the specific table. The reference at that index gets dereferenced and the specific function is called.

First it is important that many things can go wrong and actually result in a trap that terminates the program. Similar to memory access, the table index can be out of bounds. Also the reference can be a null reference or the type of the actual function does not match the type parameter that is declared in the second static parameter. It is therefore a good idea that we add assertions that cover these three cases before any indirect function call.

The best way to transform this is by modeling the table as an array, similar to linear memory. When we transform the **call_indirect** instruction, the first part will be an access to the table array. After we have parsed the WebAssembly module, we know about all functions and their type signature. Whatever data structure will be used to store this data, pointers to this information can be stored in the table array.

Translating indirect function calls is difficult, because the GOTO offers no direct replacement for dereferencing a function reference or pointer. What CBMC does it replaces it with a chain of checks.

Suppose we have a C program with three functions, f, g and h, and a function pointer p. An indirect function call to p gets replaced by a chain of if statements, where the value of the function pointer is compared to an address of each defined function. If the value matches, then this function will be called. Listing 32 shows how that looks. I have left out any further checks, such as checking that the function pointer is valid.

```
1     void f();
2     void g();
3     void h();
4     void (*p)();
5
6     // ...
7
8     (*p)();
9     // gets replaced with:
10    if (p == &f) f();
11    else if (p == &g) g();
12    else if (p == &h) h();
13
```

Listing 32: *How CBMC replaces function pointers*

In our Wasm frontend, we can introduce a similar logic. We can test through every function and see if the function reference points to it. If the signature mismatches, it will introduce a trap. Therefore we only have to check the functions with the correct signature.

6.9.2 External references

External references in Wasm have the data type **externref**. These refer to objects owned by the hosting environment. In the case that Wasm runs inside a JavaScript environment, then external references may refer to any kind of JavaScript object.

Since their only real use is in imported functions, there is no need to convert them or include them in our program.

6.9.3 Null reference

The **null** reference is a special reference that indicates that this reference points to no object. It is the default value for all reference types and can be created using the **ref.null** instruction. To handle null references, we need to add internal information to the reference type we use internally whether the reference is null or not.

The instruction **ref.is_null** checks if a given reference is null. If it is, we return 1, otherwise 0. To do this in our program, we just read that internal information if the value is null.

Null references are important to verification as they are a common source of bugs. For example trying to call a function reference that turns out to be null triggers a trap.

6.9.4 Tables and elements

A table is a special memory that holds values of type reference. Each table can only hold one type of reference.

We can model tables using the same logic as we do with memories. Instead of holding bytes, the array consists of references.

A Wasm module can define elements. Elements are used to initialize table entries, similar to how data segments are used to initialize memory segments. Each element can be declared active or passive. An active element initializes its table entry during module instantiation, while a passive element does it on demand when executing the **table.init** instruction.

If we want to have the correct state of table entries, we need to add code at the beginning of our GOTO program that initializes table entries with active elements.

6.10 Global variables

Global variables are variables that can be accessed from anywhere in the program. They behave similar to local variables, but they never go out of scope.

Global variables need to be declared in a special section of the Wasm module. Each global variable has a type, a mutability qualifier and an initialisation value. The mutability qualfier specifies if the global variable is constant or mutable. We can ignore this because module validation ensures that there are no changes to mutable variables.

We can handle them similar to how local variables are handled, by storing them in an internal table. First we set the values to a constant expression that represents the initial value. The **global.set** instruction changes the value of a global variable with the value from the operand stack. In this case we replace the value in our table with the expression we take from the stack. The **global.get** instruction reads the value from the global variable and places it on the stack. Like with local variables, we read from our table and place the expression on our simulated stack.

6.11 Miscellaneous instructions

There are some instructions that do not fit in any of the above categories. These will be discussed here.

The **drop** instruction pops the top element from the operand stack. This is easy to implement, we just need to delete the topmost element in our symbolic stack.

The **select** instruction consumes three values from the operand stack. The result is either first one or the second one, depending on if the third argument is 0. The result gets pushed back to the stack.

In our conversion, we need to replace the three top elements from our symbolic stack with a ternary if expression. The condition is if the third argument is equal to 0. The other two expression are the second and third agrument to the if expression.

7 Practical considerations

In the last chapter we have deeply analyzed the Wasm bytecode and how to transform it to work with the CBMC backend. In this chapter, we look at issues that get relevant if the project would be implemented. This ranges from parsing, to variable naming and assertions. We look at each aspect individually and try to find solutions.

7.1 Parsing

Parsing is the step in our frontend where we read the Wasm binary file and convert it into an internal structure. It would be impractical to parse the binary directly into GOTO programs, because this is not a linear process, and we would have to read the binary back in from the beginning. Instead it is easier to parse the binary code into an abstract syntax tree, before we transform this into a GOTO program.

In this section I will explain the general structure and some technical aspects of binary Wasm modules and how these can be represented in an abstract syntax tree.

7.1.1 General structure of a Wasm binary

A Wasm module is a binary format, that means it is not easily human-readable with for example a text editor.

All the data that the module uses needs to be encoded. Integers are encoded using the LEB128 (little endian base 128) format. In this format, integers are split into groups of 7 bits, starting with the least significant bits. Each group starts with an additonal bit set to 1, except the last group, which starts with 0 to mark the last byte. Floating-point numbers are encoded directly into IEEE 754, and character sequences are encoded using UTF-8. These are needed for example to refer to imported symbols by name.

The binary always starts with the byte sequence 0x0061736D, which is the magic number. The magic number is used so that operating systems and programs can identify the file type from the first bytes. In ASCII code, the magic number can be read as the NULL character followed by “asm”.

The next four bytes contain the version of the binary, which is currently 1. If there will be backward-incompatible changes to the binary format the number will increase, but it is currently not planned.

After that the file is divided into sections. Each section starts with a 1-byte section id and an unsigned 32-bit integer value specifying the size of the section contents in byte. By utilizing this size information we can break down the binary into each section very quickly.

In the next subchapters I will look at the individual sections and explain why their contents are useful for our frontend.

Type section

The type section contains an array of function type definitions. A function type definition is a mapping of input types to result types.

We will need this information because some constructs refer to one of this type definitions using the index. An example would be indirect function calls, where the expected type is defined using an index into of this array.

Function section

The function section defines an array of function definition. Each function definition consists only of an index into the type section. The rest of the function, the local variable definitions and the actual code is declared in the code section.

Table section

The table section declares tables. Each table is declared with a reference type and a size.

This is helpful so that we can verify that access to tables using an index is not out of bounds.

Memory section

The memory section defines memories. A memory is defined using an initial size and an optional maximum size. In the current version of Wasm only one memory definition is allowed.

The useful information for us is the initial memory size, which we will need to use for bounds checks on memory access.

Global section

The global section defines global variables and their types. It may be possible to gain information about global variables out of the context where they are used in instructions because each access instruction uses a static index, but it is probably easier to read this section and store the information during parsing.

Start section

The start section defines an optional start function. The contents of this section is just an index into the array of functions. The start function gets executed when the module is instantiated, so it is important to know what it does because it can change for example global variables or memory contents, which we have to keep track of.

Element section

The element section defines elements, which are initial table entries. Elements can be active, that means they load themselves into tables during instantiation, or passive, that means they get copied into tables by calling an instruction. Parsing the element section is crucial to know some certain contents of tables, for example during indirect function calls.

Code section

The code section defines the local variables and the code for each function. The function signatures and indices are defined in the function section, and the code section has the same number of elements.

The code for each function consists of a list of instructions. Each instruction is differentiated using an opcode. The opcode is in most cases a single byte. Since that would limit the number of different instructions to 256, the special opcodes 0xFC and 0xFD are being followed by an unsigned 32-bit integer that adds an additional value to distinguish instructions.

Some instructions contain additional static parameters, these are encoded directly after the opcode.

Data section

The data section defines data items, which are used to initialize memory segments. Similar to elements, data items can be either active or passive, active data items get copied into memory during instantiation.

Knowing the contents of this section, we can keep track of the initial state of any memory.

Import and export sections

The import and export sections define functions, globals, tables or memories that are either imported from the host environment or get accessible to the host environment after the module has been instantiated.

This creates a problem: If for example a function that we want to check for errors calls an important function or uses imported memory, we cannot know what will happen. I discuss this issue in more detail in section 7.3.

7.2 User-defined assertions

Assertions are important in bounded model checking because they add constraints to a program. What we essentially look for is errors in a program and to find errors we have to define erroneous states. CBMC has the ability to add automatic assertions in many cases, for example when a division by zero could be possible.

But the real advantage of tools like CBMC is that the user can place constraints himself with the assert operation. The result of this that CBMC can not only check for language errors and undefined behavior, but also check the program against a specification. The program can be specified as detailed as the user wants using assertions.

```
1         int success = do_something();
2         assert(success);
```

Listing 33: *Example of a user-defined assertion*

Listing 33 shows an example of a user-defined assertion in a snippet of a C program. The procedure *do_something()* can fail and therefore returns a value indicating that the operation was successful. We then assert that the value does not indicate an error. If we input this code into a bounded model checker, it will try find states in which the assertion fails.

The assert statement is something that is common many programming languages. Usually it stops execution immediately. The semantics – defining an expected state – are perferct for usage in bounded model checking to define erroneous states.

In Wasm there is no statement directly equal to an assertion statement in C. The only instruction that is similar is the **unreachable** instruction. This instruction takes no arguments, it just unconditionally triggers a trap which results in immediate termination of the Wasm module. To mimic an assert statement, one could possibly use the **unreachable** statement together with an if-statement. This is impractical, since it would imply that the Wasm bytecode gets written by hand.

We can look at some compilers that target WebAssembly and see how they translate the assertion statement. The Emscripten toolchain translates the C assert statement into a call to an imported function. The imported function is defined in the compiled JavaScript code and throws an exception. The name of this function is "__assert_fail". We could detect this particular function call in our frontend and convert it to an assertion statement.

In Rust there is also an assert macro. The resulting Wasm code consists of internal functions that perform cleanup and debugging tasks. At the end of the function chain the **unreachable** instruction is executed, triggering program abortion. This can be converted into a GOTO program without any tweaks, and we could replace the **unreachable** statement with `ASSERT false`.

7.3 Shared data with host environment

Wasm modules can share data with its host environment through imports and exports. These can contain functions, memories, tables and global variables. If we import them, we cannot know its contents, so we cannot verify anything that it does. This is similar to calling a function in C that only has a forward declaration, but no definition. Another example would be variables in C declared with the *extern* keyword.

The best way to resolve this is to just assume these values or function can return anything. This can be done by imposing no constraints on that data, and then the solver tries to find the most suited value for its proof.

7.4 Symbol names

Since Wasm is a bytecode that is optimized for execution instead of readability, it does not refer to language elements with a symbolic name. Instead it keeps all language elements and structures and uses indices on that structure. This is impractical, since we want to have a trace of a possible counterexample and see how it affects the program logic.

The solution is that Wasm supports officially a section in the binary bytecode that is reserved for debugging information, including symbol names. This section gets created by compilers

using a flag that triggers the inclusion of debugging information. We can parse this section if available and have access to all available names.

8 Summary

I now want to summarize the previous chapter's contents. The project was to design a frontend for the Wasm bytecode language in the CPROVER framework.

We started by introducing the GOTO language as the target for our frontend. In chapter 5 we defined the semantics of the GOTO language, so that we had a formal base.

In chapter 6 we took a close look at Wasm bytecode and its components and how it can be translated into the GOTO language. We achieved the following results:

Operand stack Wasm bytecode is a stack machine, and we have to transfer that into the more imperative GOTO language. To do that, we need to have an internal symbolic representation of the operand stack, where we store intermediate expressions. As an example, the addition instruction pops two values off the stack and pushes the result back on it. What we do is we create an addition expression that replaces the two top expressions from the stack. Each of these expressions is in input to the addition.

Local variables Local variables can store data local to a function. We can convert them into variables in the GOTO program and keep track of them in an internal table.

Multiple return values The GOTO language does not support multiple return values for functions, but WebAssembly does. To circumvent this, we can combine the return values in a single structured type and deconstruct it in the calling function.

Integers Integers in Wasm are 32-bit and 64-bit wide bitvectors. The type system in Wasm does not differantiate between signed and unsigned interpretations. If an instruction has different semantics for unsigned and signed values, there exist two variants of that instruction. This prevents us from reliably placing overflow checks, but in Wasm, overflow is defined using wrapping arithmetic.

Division by zero and division overflow can produce a trap, which means the program can terminate. It is therefore a good idea to add assertions to detect that. Most integer operations can be converted into built-in expression types, while more complex operations can be converted using nested expressions.

Floating-point numbers Floating-point numbers in Wasm can be 32 and 64 bit wide and implement the IEEE 754 standard. In other languages, many operations on floating-point numbers are not built into the language, they are library functions instead. For these languages, CBMC provides its own implementation of common standard library features. In Wasm, there is no standard library so these functions, like taking the square root of a number, are built into the language. We can either provide "library" functions by implementing these operations using simpler instructions, or extend the GOTO language and let the solver resolve the functionality, which exists in some cases.

Vector data Wasm provides a data type for vector data, which consists of 128 bits. This bitvector can represent multiple *lanes* of data of a smaller type. There are instruction that perform operations on that data, multiple values at a time. These are called single-instruction, multiple-data instructions.

To convert those instructions, we need to decompose the vector into each lane. We then perform the operation on each lane. Since all these operations are independent of one another, it does not matter if we perform them at the same time or one after the other. Then we concatenate the indivual parts back into the vector.

To represent the vector we could have used arrays, like some existing CBMC functionality for vectors in the C language does. This would be practical in some cases, but impractical in others. I believe that representing them as a bitvector of width 128 is a better choice.

Linear memory The memory model of WebAssembly consists of a global array of bytes. There is no need for memory allocations, all available memory is permanently accessible.

To model memory we treat is an array of 8-bit wide bitvectors. Each instruction that reads from and writes to memory need to address each index individually, since memory accesses can potentially overlap.

An out-of-bounds memory access in Wasm causes a trap which terminates the program. It is important to avoid that, and before each memory access we should add assertions to the GOTO program to detect possible errors. For this, we need to keep a variable that remembers the current size of available memory.

Available memory gets initialised using 0 values. In some cases it may make sense to add this at the beginning of the resulting GOTO program, in some cases it does not. The best way may be to make it configurable.

Structured code flow WebAssembly offers structured code constructs. It is possible to define blocks, where special instructions called branching instructions “break out” of.

To convert this into the GOTO language, we need to reduce those constructs into *GOTO* statements. This can be done using static analysis.

Type conversions It is possible in WebAssembly to convert values between all numeric data types. In most cases, the semantics of the conversion is equivalent to the semantics of the type cast expression in the GOTO language, which is inspired by the semantics of casting in the C language.

In some cases we need to add assertions because type conversion can trap, especially conversions from floating-point to integer types.

A reinterpretation cast changes the type of a value while keeping the bit pattern the same. This can be done by casting the value to an uninterpreted bitvector, and then casting it to the target type.

Function references Function references can be used to call functions indirectly, similar to function pointers in other languages. To include them in the GOTO language we have to create

a new data type, because all reference types in the GOTO language are bitvectors, but Wasm defines that the bit pattern of references cannot be observed.

To convert indirect function calls into the GOTO language, we have to compare the value of the reference to each available function that matches that type signature. If it matches, then we perform a function call.

We have looked at some other discussion points not directly related to WebAssembly semantics in chapter 7. Here are the topics we talked about:

Parsing To to any kind of model checking on WebAssembly bytecode, we have to parse it in an internal structure first. The bytecode is specified in a binary format.

A binary Wasm module consists of multiple sections, for example types, imports, function signatures and code. We have to read each of these sections, because each one is important for our frontend in one way or another.

The code is encoded using opcodes. Each opcode represents a different instruction. Therefore to parse the code, we need to have some kind of table, that we can use to translate the opcodes into readable instructions.

User-defined assertions WebAssembly does not offer an assert statement like C or Java do. Therefore we have to do a work-around to let users define assertions in their code.

One way is to let users edit the Wasm code by hand and place **unreachable** instructions. Another way is to look at common WebAssembly compiler toolchains and see how they convert the assert statment from the source language. Whatever they put in the Wasm code, we detect and convert it to an assertion in the GOTO program.

Shared data with host environment WebAssembly code is not designed to run for itself, it is usually embedded into a host environment, such as a JavaScript engine. It can communicate with it by importing and exporting functions and data. That means we cannot know the value or contents of this data for sure.

We solve this by not putting any constraint on that data. It can be any value possible, so we do not assume it has a defined value.

Symbol names WebAssembly is a machine-readable bytecode and does by default not include symbol names such as function names. After verification, we want to understand how the program reached a state of error, and function names are very helpful.

To include symbol names in our output, we need compile the WebAssembly code with debugging information, and during parsing we need to process that information.

Bibliography

[1] B. Alliance. *Github - WebAssembly/WASI: WebAssembly System Interface*. 2024. URL: https://github.com/WebAssembly/WASI (visited on 12/19/2024).

[2] C. Barrett, P. Fontaine, and C. Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016.

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. "Symbolic Model Checking without BDDs". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by W. R. Cleaveland. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. ISBN: 978-3-540-49059-3.

[4] J. R. Burch, E. M. Clarke, D. L. Dill, and J.-N. Hwang. "Symbolic model checking: 1020 states and beyond". In: *Logic in Computer Science*. 1989. URL: https://api.semanticscholar.org/CorpusID:124743249.

[5] E. Clarke, D. Kroening, and F. Lerda. "A Tool for Checking ANSI-C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by K. Jensen and A. Podelski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. ISBN: 978-3-540-24730-2.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic verification of finite-state concurrent systems using temporal logic specifications". In: *ACM Trans. Program. Lang. Syst.* (1986).

[7] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik. "JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode". In: *Computer Aided Verification (CAV)*. LNCS. Springer, 2018. ISBN: 978-3-319-96144-6.

[8] B. Farias, R. Menezes, E. B. de Lima Filho, Y. Sun, and L. C. Cordeiro. "ESBMC-Python: A Bounded Model Checker for Python Programs". In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2024. Vienna, Austria: Association for Computing Machinery, 2024. ISBN: 979-8-40070-612-7. DOI: 10.1145/3650212.3685304. URL: https://doi.org/10.1145/3650212.3685304.

[9] D. Gandluri, T. Lively, and I. Stepanyan. *Fast, parallel applications with WebAssembly SIMD*. 2020. URL: https://v8.dev/features/simd (visited on 12/02/2024).

[10] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, et al. "Bringing the web up to speed with WebAssembly". In: *SIGPLAN Not.* 6 (June 2017). ISSN: 0362-1340. DOI: 10.1145/3140587.3062363.

[11] D. I. O. Herrera. "Verification of WebAssembly programs". In: (Feb. 2020). DOI: 10.25949/19444358.v1.

[12] ISO/IEC. *Programming languages - C, Committee Draft September 7, 2007*. en. Standard ISO/IEC 9899:TC3. International Organization for Standardization, 2007. URL: https://open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf.

[13] D. Kroening and O. Strichman. *Decision Procedures. An Algorithmic Point of View*. 2nd ed. Berlin, Heidelberg: Springer-Verlag, 2016. ISBN: 978-3-662-50497-0. DOI: https://doi.org/10.1007/978-3-662-50497-0.

[14] D. M. Mazarro. “Specification and verification of WebAssembly programs”. MA thesis. Universidad Politécnica de Madrid, 2023.

[15] R. Menezes, M. Aldughaim, B. Farias, X. Li, E. Manino, et al. “ESBMC 7.4: Harnessing the Power of Intervals”. In: 30^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'24)*. Lecture Notes in Computer Science. Springer, 2024. DOI: https://doi.org/10.1007/978-3-031-57256-2_24.

[16] K. Project. *The Kani Rust Verifier*. 2024. URL: https://model-checking.github.io/kani/ (visited on 12/18/2024).

[17] K. Song, N. Matulevicius, E. B. de Lima Filho, and L. C. Cordeiro. “ESBMC-Solidity: An SMT-Based Model Checker for Solidity Smart Contracts”. In: *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2022. DOI: 10.1145/3510454.3516855.

[18] A. VanHattum, M. Pardeshi, C. Fallin, A. Sampson, and F. Brown. “Lightweight, Modular Verification for WebAssembly-to-Native Instruction Selection”. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS '24. La Jolla, CA, USA: Association for Computing Machinery, 2024. ISBN: 9798400703720. DOI: 10.1145/3617232.3624862. URL: https://doi.org/10.1145/3617232.3624862.

[19] C. Watt. “Mechanising and verifying the WebAssembly specification”. In: Jan. 2018. DOI: 10.1145/3167082.