Symbolic Execution of WebAssembly

Bachelor Project in Computer Science,
DIKU, University of Copenhagen

Troels Korreman Nielsen
<xck773@alumni.ku.dk>

Supervisor: Ken Friis Larsen, <kflarsen@di.ku.dk>

November 22, 2024

Abstract

WebAssembly is a new language, set to become an integral component of web development as a cross-platform compilation target. Despite being primarily designed for the web, it may be useful as a base language for software delivery and other tasks. This project explores the possibility of performing symbolic execution on the WebAssembly language as a tool for cross-platform, language agnostic program testing and verification. A symbolic interpreter for WebAssembly is written and tested for various WebAssembly programs. The performance of the interpreter is measured by benchmarking its execution time and memory usage when running various functions and gradually increasing the depth of execution. The interpreter is found to exhibit poor performance, and optimization techniques are discussed and evaluated in relation to implementation in the interpreter and their synergy with WebAssembly as a whole. WebAssembly is found to be a promising language for symbolic execution. Future work could focus on identifying optimization techniques that are particularly well-suited for WebAssembly.

Contents

1 Introduction

1.1 The case for symbolic execution of WebAssembly

For compilers, there are trade-offs in the choice of program representation when performing optimizations, choosing a target language, and doing analysis. High level languages can often provide useful information and reasoning about a program, but contain language-specific complexity. Low level languages have neither the high level conceptual knowledge nor the associated complexity, but may introduce their own architectural complexity. For these reasons, compilers will often use an intermediate representation to bridge the gap between languages and platforms. This also avoids the redundant work of writing a compiler for every combination of language and target platform.

Program analysis faces the same trade-off. High level languages provide insight into the program structure, but require that tools are customized for every specific language. On the other hand, compiled low-level languages may be completely divorced from the source code, and are generally complex in other ways.

In the case of symbolic execution, we are generally trying to detect runtime errors in a program. The high-level insights can often be boiled down to some categories of errors which we do not need to check for, while low level languages can contain thousands of different instructions that may help concrete performance while hurting symbolic performance. An intermediate representation may mitigate this problem, but the design goals for such a representation can differ from those of a compiler.

This project explores the possibility of using WebAssembly as such an intermediate representation. WebAssembly is a supported target language for many languages, including C, C++, Rust. It is a very minuscule language, defining no more than 181 instructions as of its 1.0 specification. This number falls below 80 if overlapping instructions like `i32.add` and `i64.add` aren't counted separately. Despite its size, WebAssembly has all the capabilities that a low-level language needs in order to run modern software. The question we explore is whether WebAssembly strikes the right balance between expressivity and simplicity for it to be considered a good candidate for use in symbolic execution.

1.2 Notation

This report will contain code examples and symbolic expressions. To discern between the two, code is highlighted by a gray background while symbolic expressions are outlined by a box.
A symbolic expression looks like this: $\boxed{\texttt{2 + x * 3 / 5}}$
A piece of code looks like this: `mul x y = x * y`

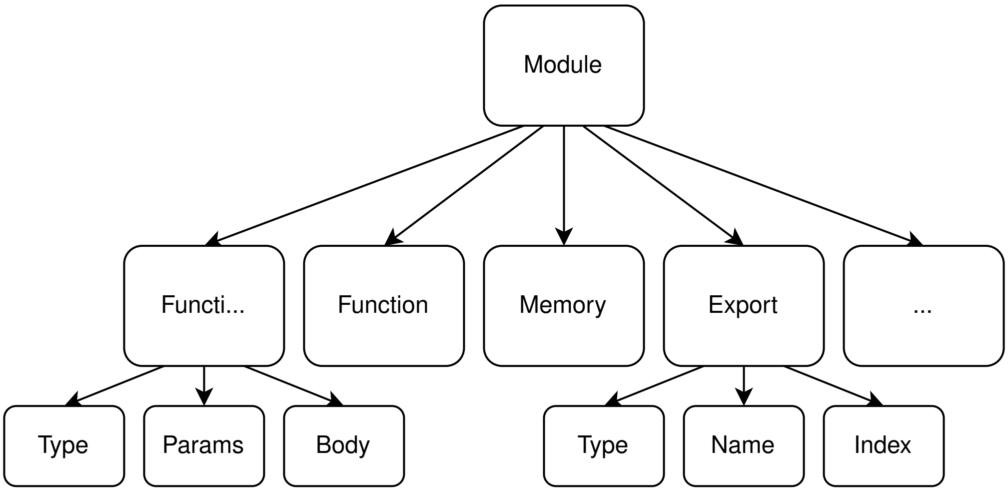

Figure 1: Example representation of a module

2 WebAssembly

WebAssembly (Wasm) is a language designed primarily to be a target language for cross-platform applications on the web. The goal of the language is to "Define a portable, size- and load-time-efficient binary format to serve as a compilation target which can be compiled to execute at native speed [...]" [4]. Wasm is defined by a normative specification which is currently at version 1.0 [6].

2.1 Syntax and structure

Wasm defines two syntaxes that parse to the same internal representation: a textual representation for human reading (has the extension `.wat`) and a binary representation for efficient transfer and parsing (with the extension `.wasm`). As they represent the same underlying structures, it is easy to convert between the two. Niceties like comments, names, and non-restricted ordering of top level constructs are lost when converting from the textual format to binary, however.

The textual representation of Wasm is built using S-expressions. A module is a collection of top-level constructs like functions, memory, imports, exports, globals, and more. These are each fully defined in their own sub-trees and may reference each other. Figure 1 shows an example of how a Wasm module may be structured.

2.2 Embedder

Wasm is designed to be embedded into other languages. It is explicitly structured so that one may call Wasm functions from an embedder in a parent environment.

An `export` construct tells the embedder to expose another named construct to the outside world. Through exports, one can expose functions, memory, globals, and function tables to the embedder context.

Conversely, an `import` construct requires the embedder to predefine certain constructs while initializing the module. One may require functions, memory sections, globals, and function tables to be defined before taking the module into use.

To use a Wasm module, it must be initialized by an embedder. In the initialization phase, external libraries, functions, and data can be exposed to the module through function imports. Functions exported from the module can then be called through the embedder. Exported module memory may also be read by the embedder for a low-overhead data transfer between the embedder and module. For example, one might write a list of elements to exported Wasm module memory, call an exported sorting function, then read back the sorted list directly from module memory again.

2.2.1 Labels

Many constructs in Wasm are referenced by index. To make textual code more readable, these indices can be replaced by symbolic identifiers. Symbolic identifiers are preceded by a `$`, and are translated back to indices when converted to bytecode and executed.

2.2.2 Functions

A function in Wasm consists of a set of parameters, a set of locals, a result type, and a code body. The code body is an expression, which in the context of Wasm is a sequence of instructions. The syntax of a typical function will look like this:

```
(func $identifier
    (param $arg1 i32) (param $arg2 f32)
    (local $lcl1 i64) (result i32)
  instr1
  instr2
  ...
)
```

Calling a function is straightforward: the arguments are saved in variables, and the instructions are executed in sequence. Wasm also has a `return` instruction in order to exit function execution from any point.

2.3 Runtime state

2.3.1 Values

Values in Wasm are typed with a simple system. They can be one of `i32`, `i64`, `f32`, or `f64`, corresponding to 32-bit words, 64-bit words, 32-bit floats, and 64-bit floats. Type checks are performed at compile time to ensure that no implicit reinterpretation of values takes place.

2.3.2 Stack

Wasm is a stack-oriented language, meaning that values are generally passed around by pushing to and popping from a stack. Almost every instruction that deals with data will interact with the stack in some way.

Unary operations pop a value from the stack, transform it, and push back the result.

Binary operations will pop two values instead, then push one resulting value to the stack. The bottom value becomes the first operand so that pushing operands follows an intuitive order.

When calling a function, a number of values corresponding to the amount of parameters are popped from the stack and passed to the function, with the bottom value being the first argument.

2.3.3 Locals and globals

Sometimes a stack doesn't quite cut it. For these cases, there are local and global variables, referred to just as 'locals' and 'globals'. Locals are defined for each function, and all parameters also become locals when the function is invoked. Globals are defined at the module level, being accessible from within every function.

Instructions like `local.get`, `local.set`, and `local.tee` will respectively retrieve a local value and push it to the stack, pop a value and write it to a local, and write the top value of the stack to a local without popping it.

2.3.4 Memory

Memory in Wasm is represented as a single array. The size can be set at initialization and expanded at a later time. It can be byte-accessed with an `i32` index, limiting its maximum size to 4GiB. In future versions of the specification, larger memory and multiple memories may be supported.

Wasm has quite a few instructions for loading and interpreting values from memory. As an example, the instruction `i64.load32_s 0 2` will:

1. Pop an `i32` value from the stack.
2. Add 2 to the value.
3. Treat this now-offset value as an address and load 4 bytes from the corresponding location in memory.
4. Sign-extend the loaded value from 32 bits to 64 bits.
5. Push the extended value to the stack.

The 0 is an alignment parameter which is purely there for performance optimizations.

2.4 Control flow

Block instructions enable control flow in Wasm expressions. Combined with branching instructions, they provide expressivity above typical imperative `if`, `while`, and `for`-statements, while still being more restrictive than direct `goto` jumps.

Block instructions each contain a sub-expression to execute. Validation rules and execution rules ensure that the current stack isn't modified by the sub-expression, and that it either leaves no new values or a single value on top of the stack when exited, according to its result type.

Branching instructions can be used inside sub-expressions. They must specify a 'label', an integer representing how many levels to branch outward, where 0 branches to the current block. The behavior of a branching instruction depends on the block type targeted by the branch label. The instruction `br` will always perform a branch, while `br_if` will pop a value and branch if it is non-zero.

2.4.1 `block` and `if`

`block` is the simplest of the block instructions. Branching to its label will jump to the end of its sub-expression, ending execution of the expression early. `if` works the same way as `block`, but will start execution by choosing between two subexpressions to execute depending on whether a popped value is non-zero. Branching for these blocks is similar to the imperative `break` statement.

2.4.2 `loop`

Branching to the label of a `loop` block will restart the execution of the subexpression. All newly pushed values will be discarded when branching. This is similar to the imperative `continue` statement, but branching is required in order to keep a loop running in Wasm. When the subexpression finishes executing, the loop is exited by way of fallthrough.

2.5 Runtime errors

While the structure and validation of Wasm prevents many errors, programs can still encounter certain situations from which they cannot recover. Some of these are:

- Reaching an `unreachable` instruction.
- Division by zero.
- Out-of-bounds memory access.

When such an error is encountered, a trap is activated and execution is halted. These are the primary undesirable states we wish to be able to detect.

2.6 Example

Figure 2 shows a program which calculates the n'th number in the Fibonacci sequence using a loop-based method. A single parameter `$n` is defined, the result is an `i32`, and the function uses two local variables. Things to note:

- All locals are defined right after the function signature.
- Parameters are treated as locals in the code body.
- The `loop` instruction contains its own subexpression.
- The `br_if` instruction checks if the value of `$n` on the stack is non-zero, and jumps to the top of the loop if so.

```
(module
  (func $fibonacci (param $n i32)
                   (result i32)
                   (local $acc i32)
                   (local $acc_prev i32)
    i32.const 1
    local.tee $acc
    local.set $acc_prev
    (loop $add_loop
      local.get $acc_prev
      local.get $acc
      local.tee $acc_prev
      i32.add
      local.set $acc

      local.get $n
      i32.const 1
      i32.sub
      local.tee $n
      br_if $add_loop
    )
    local.get $acc
  )
  (export "fibonacci" (func $fibonacci))
)
```

Figure 2: A Fibonacci algorithm written in Wasm

- The return value is the value left on the stack after executing the function body.
- The function is exported with the name `"Fibonacci"` in order to be used by an embedder.

3 Symbolic execution

The core concept of symbolic execution is to execute a program using symbolic values instead of concrete ones. By performing analysis of these values, we can determine whether certain properties hold for all (or most of the) possible executions of a program. This can serve as an effective method for performing program analysis and validation instead of manually constructing hundreds to thousands of individual, distinct tests.

3.1 Symbolic values

Concrete values describe a single configuration chosen from a set of possibilities. Take the simple calculation: `2 * 3 = 6`. Here 2, 3, and 6 are all concrete values.

The use of symbolic expressions/values should be familiar as well. For example, in an equation like `x + y = 2 * x * y`, each side of the equation can be seen as a symbolic value. The symbols `x` and `y` represent unspecified numeric values. The equation imposes a constraint on the possible configurations of these symbols, and we may find assignments of concrete values to the symbols for which this constraint is satisfied (`x = 1` and `y = 1`). Note that symbolic expressions can contain concrete values, and an expression can be completely concrete if they don't contain any symbols.

In symbolic execution, symbols are given as inputs at the start of a program. When read, transformed, and written, all values in the runtime state are symbolic expressions. By transforming symbolic expressions, constraining them, and checking for satisfying value assignments, we can analyze how programs run and verify certain properties about them.

In this project, an SMT solver is used to represent, transform, constrain, and check symbolic values.

3.2 Constraints and conditionals

Symbolic values aren't an issue for predetermined state transformations. They do however present an issue for any conditional transformations. If the program encounters a conditional branch instruction, what should it do? A condition on symbolic values is symbolic in itself, so it is generally possible to find concrete assignments which evaluate the condition to both true *and* false.

As a solution, symbolic execution engines maintain a collection of constraints as part of the runtime state, which is referred to as the *constraint path.* When checking for satisfiability of a property, these constraints must be satisfied as well. Whenever a conditional transformation is encountered, we add the condition or its negation to the constraint path. By doing this we effectively choose a branch and rule out all value assignments that would lead to the other branch being chosen. To cover all possible executions of a program, it is necessary to fork the process and follow execution for both the condition and its negation. This can be accomplished through either backtracking or spawning multiple runtime instances to run concurrently. In either case, the runtime state must be copied. This method of forking the process for conditionals can also be generalized to conditional statements with more than two distinct outcomes.

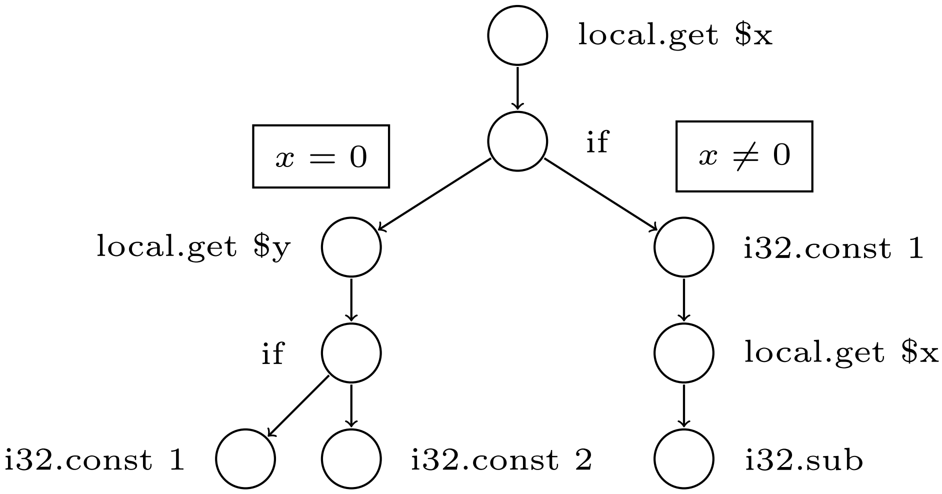

Figure 3: A short code snippet represented as a decision tree. Each vertex is labeled with the next pending instruction. Branches to the right represent a condition being true, branches to the left do the opposite.

We can visualize this process as a decision tree (see Figure 3). Vertices represent runtime state, and edges represent transformations. Conditional transformations can then be represented as a vertex having multiple outgoing edges, each with its own deterministic transformation and an additional path constraint. Symbolic execution can be viewed as a traversal of this decision tree, searching for vertices that satisfy some property or invalidate some assertion.

3.3 Properties

Being able to symbolically execute programs, we can use this to analyze programs in a number of ways. The most common objective is to verify properties about a program as a way to test for bugs and errors. These properties can be expressed in a variety of ways, for example:

1. Inline assertions in the program
2. Assertions on the output of a function
3. Assertions on the runtime state at any given point in execution

For this project we are focusing on the third example, as it is relatively flexible, and can stand in for the other two fairly well. Inline assertions can be constructed with a conditional block that contains some trap or error, and it is possible to construct assertions that only trigger at a certain code location, like the end of a function.

Assertions may take both symbolic values and structural runtime state into account. A basic assertion can state that the next transformation or instruction may not be some sort of `unreachable`. This only considers structural runtime state without actually inspecting any symbolic values, but symbolic execution will still serve as a way to explore and analyze branch logic. A more complex assertion could state that the second value on the stack may not be zero if the next transformation would use it as a divisor. This considers both the next instruction and a symbolic value.

If we wish to construct more complex assertions, we can either develop an assertion language language, or express them as inline code. Assertions can be converted to conditional checks which guard an `unreachable` instruction. This method opens up for expressing complex properties, but is of course limited by the language itself, and can only be used to form inline assertions.

3.4 Challenges to symbolic execution

[1] describes four main challenges to the feasibility of symbolic execution: path explosion, symbolic memory, environment integration, and constraint solving. Techniques for overcoming these obstacles will typically trade in some amount of soundness (identifying *all* property-breaking inputs) or completeness (property-breaking inputs actually do break the property) for performance and heuristic benefits. This is generally accepted, as the goal usually is to identify bugs in a program rather than to verify the integrity of the entire program. Many of these techniques are classified as peforming 'concolic execution', as they mix concrete and symbolic execution.

3.4.1 Path explosion

A major challenge to symbolic execution is the sheer amount of unique paths an execution can take through the control flow structures of a program. Every conditional transformation forks the process into at least two new states going forward. In the worst case, this results in an exponentially growing collection of paths, doubling the number of states for every branch. Exploring such a decision tree is one of those prohibitively expensive tasks which might run until the heat death of the universe.

Techniques to overcome this problem are centered around exploiting program structure and solver feedback in order to discard paths as irrelevant and/or redundant. For example, eager constraint evaluation will check the constraint path for satisfiability when pushing new constraints, nullifying branches where constraints are unsatisfiable.

3.4.2 Memory

Modeling memory as a symbolic array holding symbolic values is challenging. The problem lies in the symbolic addresses that create a layer of indirection. With unbounded symbolic addresses, writes can modify the value in any location, and reads can retrieve values from any location.

One way to model this is to fork the state for every possible address value. This obviously creates a *lot* of forked states, trading memory complexity for path explosion.

Another approach is to move the complexity to the solver. One could express read values as if-else expressions, and some SMT solvers directly support array theories.

3.4.3 Environment integration

Most programs will interact with their environment by way of API calls or system calls. The trouble for symbolic execution is that environments are generally large opaque collections of concrete state. Library APIs are often proprietary, so we cannot perform symbolic execution on these. It is generally required to concretize values for calls to outside functionality, and backtracking the environment transformations may be impossible.

There are generally two approaches to circumventing this problem: symbolically model the environment in order to properly simulate it, or compromise and concretize. Virtualization is often used to enable parallelization and backtracking.

3.4.4 Constraint solving

Constraint solvers are the backbone of symbolic execution, making it possible in the first place. However, there are still limitations that constraint solvers like SMT solvers struggle with, like non-linear arithmetic. Approaches to mitigating this problem generally look to reduce the constraints, reduce the number of queries, and cache results for reuse.

4 SMT solvers

SMT is short for Satisfiability Modulo Theories. As the name suggests, SMT solvers are proof engines that extend SAT solvers with modular theories at higher levels of abstraction. While the boolean primitives of SAT solvers can be used to build high-level logic, this is usually inefficient and tedious. SMT solvers provide efficient and accessible tools for proving properties about integers, reals, arrays, and similar high level primitives.

4.1 SMT-LIB and Logics

SMT-LIB [2] is a language specification that standardizes the interface to SMT solvers. Most solvers support this language, allowing one to decouple queries from the solvers, then use any solver as a backend (provided that it supports the subset of SMT-LIB that is utilized).

SMT-LIB defines a sorted (i.e., typed) first order language with which to construct terms. SMT-LIB also defines a set of *background theories*, theory modules that define the workings of integers, reals, arrays, etc. A combination of theories and a subset of the first order language language is referred to as a (sub)logic. With an explicitly chosen logic, the solver can work faster than if it were to support every theory and the entire first order language.

SMT-LIB is used through a command language where we can construct symbolic values, push and pop constraints, and check for concrete value assignments that satisfy the current constraints.

This project utilizes theories of integers, reals, and fixed-size bitvectors. These allow us to represent symbolic expressions that reflect the data types of WASM. To support memory, we would use a theory of arrays as well.

4.2 SBV

SBV is a Haskell library that interfaces with SMT-LIB. Rather than being a simple interface, it abstracts away SMT-LIB in favor of an API which treats symbolic values as regular Haskell values. Values have symbolic types like `SInt32`, `SDouble`, `SBool`, and so on. They implement various Haskell typeclasses so that one can perform arithmetic, boolean operations, and many other transformations on them as one would with basic Haskell types.

The values can only be constructed inside a `Symbolic` context, which is a typical stateful monad. When the function `runSMT` is applied to a `Symbolic` contextual value, SBV will convert this into an IO action which interfaces with SMT-LIB to compute a result.

Another stateful monad, `Query`, is also provided. This allows more fine-grained control of the solver, letting us push and pop constraints as well as check satisfiability in intermediate stages of the query. After construction, a `Query` value can be converted to a `Symbolic` value.

In practice, SBV lets us construct compound symbolic values from symbols and literals. Symbolic booleans can be pushed as constraints, and under these we can check for satisfiable assignments of values to symbols.

4.3 Example

As an example, let's use SBV in Haskell to solve quadratic equations. A typical equation will have a variable `x` and some parameters `a`, `b`, and `c`: $ax^2 + bx + c = 0$. We can construct this as a symbolic expression/constraint `a * x * x + b * x + c == 0`:

```
quad :: Float -> Float -> Float -> SFloat -> SBool
quad a b c x =
    literal a * x * x + literal b * x + literal c .== 0

x :: Symbolic SFloat
x = sFloat "x"
```

The parameters can be constructed as concrete literals, as `x` is our only variable. We can then construct `x` as a symbolic value with the name "x". By mapping and calling `sat`, we can check the satisfiability of our equation for different parameters, and get an assignment for `x` in the case of satisfiability. Here we attempt to find solutions to the equations $x^2 + 2x + 3 = 0$ and $5x^2 + 6x + 1 = 0$:

```
> sat (quad 1 2 3 <$> x)
Unsatisfiable
> sat (quad 5 6 1 <$> x)
Satisfiable. Model:
  x = -0.2 :: Float
```

5 wasym

The main product of this project is wasym, a symbolic execution engine for WebAssembly. It is written in Haskell and uses the SBV library to interface with the z3 SMT solver. This section describes the implementation and usage of this engine.

5.1 Implementation

The capabilities of a symbolic execution engine can be seen as a superset of the capabilities that a concrete engine possesses. Any symbolic value can be restricted so that it only has a single satisfying assignment, which in turn will restrict execution to a single possible path and concrete assignment, equivalent to performing concrete execution of a program. Inspired by this, I chose first to implement a concrete interpreter and then extend it to use symbolic values afterwards. This allowed me to divide the project into manageable parts and familiarize myself with WebAssembly before delving into symbolic execution.

Reflecting the approach, this section is divided into three parts: the creation of a concrete interpreter, extension of the interpreter with symbolic capabilities, and the creation of a validator that utilizes the symbolic interpreter.

5.1.1 Concrete interpreter

First, a concrete interpreter is written. A requirement for the interpreter is high modularity, in order to grant enough flexibility to extend it for its later use case. With this goal in mind, the engine is built around an `mtl`-style monad transformer stack (a style that defines and overloads families of monads, inspired by [5]), starting with a `State` monad that transforms a runtime state. By wrapping this in a `Reader` monad, we have access to configuration data and the syntax tree as well. This system allows the definition of primitive actions which can in turn be combined to construct larger macro-actions. We can define a synonym for this transformer stack type which the components are modelled on:

```
type Machine a = ReaderT Config (State WASMState) a
```

Where `Config` is a record of immutable globals (settings, input variables, the module's syntax tree) and `WASMState` is a record of mutable runtime state (instructions, globals, the value stack, the function frame, local variables, the block context, etc.).

At the lowest level, primitives are defined for interacting with the runtime state. This comprises components that perform actions like pushing and popping values, reading from and writing to locals, entering and exiting block contexts and frames, and so on. These primitives are defined along with the `Machine` type in `src/Machine.hs`.

Using these low-level primitives, it is possible to define how each instruction transforms the runtime state. As an example, the instruction `local.get` needs to read a local value, then push it to the stack. It can be implemented like so:

```
LocalGet idx -> do
    vars <- peekVars
    let v = vars V.! (fromIntegral idx)
    pushVal v
```

These instruction definitions reside in `src/Exec.hs` in the function `execInstr`. Execution of a program can essentially be achieved through repeated applications of `execInstr` to the runtime state.

5.1.2 Symbolic extensions

Now we can extend the engine to support symbolic values. The primary challenge is that values must correspond to expressions existing in the context of an SMT-solver. The `Symbolic` monad provided by `SBV` can be integrated into the transformer stack to support direct manipulation of symbolic values, resulting in the following type:

```
type Machine a = ReaderT Config (StateT WASMState Symbolic) a
```

The runtime state, low-level transformers, and instruction implementations are all sufficiently decoupled so that we only have to refactor relevant parts of our code.

A system for manipulating symbolic values is created with the wrapper type `SValue`, for which logic, arithmetic, and comparison operations are implemented. The runtime state is made to use this type. Initialization must now happen in a `Symbolic` context. This has no effect on low-level primitives, as they shuffle values around without inspecting them. Value-dependent instructions are modified to read and manipulate instances of SValue. Conditional instructions cannot be supported at this level, but will be re-introduced in the next section.

5.1.3 Search

Finally, we build a validator which utilizes our symbolic execution engine. Our validator is a search function which, given a predicate, will attempt to find an execution path for which the predicate is satisfiable. The predicate should be a negation of the assertion we wish to validate.

The `Symbolic` context is exchanged for the `Query` context, allowing us to control the pushing and popping of constraints as well as to check satisfiability mid-execution. The innermost type is also wrapped in a `MaybeT` transformer, resulting in the final type:

```
type Machine a =
    ReaderT Config (StateT WASMState (MaybeT Query)) a
```

`MaybeT` makes our `Machine` an instance of the `Alternative` typeclass. This gives us an 'alternative' operator, `(<|>)`, which in this instance will copy the runtime state and attempt to run multiple components on it. If any component returns a result (a satisfying assignment of concrete values to symbols), this result is chosen. The alternative operator is used for conditional instructions, forking the process and continuing execution with two opposing path constraints.

The search is a straightforward naive DFS. It can be described as follows:

1. Test if the predicate is satisfied on the current runtime state given the current constraints.

2. If so, stop execution and return the concrete assignments which satisfy the predicate.

3. If not, acquire the next instruction:
    (a) For conditional instructions, push the condition as a constraint and continue execution. If nothing is found, backtrack to this point and push the negation of the condition as a constraint.
    (b) Otherwise, simply execute the instruction.
4. If the end of the function has been reached, stop execution down this path.
5. Jump to 1.

Execution can be halted down a path in two other ways as well:

- If the path constraints alone are unsatisfiable.
- Optionally, if the depth has reached a specified level. That is to say, a maximum number of instructions have been executed in succession.

6 Examples

The final product is an engine which supports function calls, control structures, unsigned integer transformations, and unsigned integer conditionals.

6.1 Usage

wasym can be cloned from the following repository: `https://github.com:diku-dk/webassembly-symbolic-execution/`, and is located in the `wasym` folder. The code can also be viewed in appendix A. It can be built using the Stack toolchain [3], and requires the `z3` solver to be installed on the system in order to run.

More comprehensive checks can be performed by using wasym as a Haskell library, but wasym does compile to an executable which will check assertions on a single function. As an example, the following commands check whether an `unreachable` instruction can be reached in the functions `unreach_good` and `unreach_bad` exported by the `examples.wat` Wasm module.

```
> cd wasym
> stack build
...
> stack run test/examples.wasm unreach_good 100 unreachable
assertions held

> stack run test/examples.wasm unreach_bad none unreachable
assertions failed with the following value assignments:
[I32Val 40026128,I32Val 37928963,I32Val 37928962,I32Val
    4374529]
```

While only assertions about `unreachable` instructions and division by zero are supported, we can use the `unreachable` instruction to construct more complex conditional queries inside a function in the Wasm module itself, by guarding it with conditional instructions.

6.2 Examples

The module `examples.wat` (appendix B.2) contains some sample functions that may either fail or succeed, denoted by their names. These can be executed as shown in the script `run_examples.sh`. The examples demonstrate that functionalities such as function calls, control flow structures, and arithmetic are supported. As an example, let's look at the `modulo_div_bad` function, shown in figure 4. The function calculates $\frac{z}{x \bmod y}$, but only checks that `y` is non-zero. The engine can solve to find concrete assignments so that `x` is divisible by `y`, resulting in a division by zero in the following operation.

The `math` Wasm module B.1 contains some more complex looping and recursive behavior, and contains functions for comparing the outputs of functions. These demonstrate the engine's ability to verify more complex loops and math.

```
(func $modulo_div_bad (param $x i32) (param $y i32)
                      (param $z i32) (result i32)
  local.get $y
  (if (result i32) (then
    local.get $z
    local.get $x
    local.get $y
    i32.rem_u
    i32.div_u
  ) (else
    i32.const -1
  ))
)
```

Figure 4: A WebAssembly example that computes a remainder and division.

7 Performance

7.1 Methodology

The current implementation of the engine will search through every path in a decision tree, no matter what. Though it speaks to the inefficiency of the engine, the presence of an assertion and code points which could break said assertion has no effect on the traversal of execution paths. The engine will still keep track of all values and all constraints, not considering whether they actually affect the reachability of any undesirable states. Out of the two possible assertions, `intDivZero` is the only assertion that takes a symbolic value directly into account. It performs an "equals-zero" check, which is the same type of check that every branch performs, so it doesn't represent a unique pattern to benchmark. The point is, the benchmarking the engine currently doesn't depend on reaching actual undesirable states, and error-less functions can be benchmarked just as well.

Non-looping, memory-less functionality can generally be boiled down to a set of equations. It requires large code bodies in order to stress the engine. It is easier to test the system using recursive and looping functions that perform the same branching instruction on repeat. Therefore, a `math` module (see appendix B.1) is constructed to test the performance of the engine. It contains functions that use recursion and loops to calculate values like Fibonacci numbers and triangular sums. The execution time and memory usage of these functions is benchmarked for increasing search depth limits. Figure 5 shows the results for execution times, while Figure 6 shows the results for memory usage.

7.2 Benchmarks

Looking at these benchmarks, we see both some expected and some surprising results. All loop-based functions perform better than their recursive counterparts, both in terms of execution time and memory usage. This matches the general concrete performance of loop-based methods compared to recursive ones. Just like in the concrete case, recursive functions must stack frames on top of frames for every iteration, resulting in greater memory usage and stack growth. Values cannot be dropped before the entire calculation finishes, and this goes for symbolic values as well. This hints that concrete performance patterns may carry over to symbolic execution to some extent.

In general, we see the expected pattern for execution time: very continuous polynomial or exponential growth.

The most confounding results come from the execution times of the greatest common divisor functions. Both of these functions have points where a slight increase in depth results in a drastic decrease in execution time. These results were validated by running the benchmark multiple times on an otherwise idle machine, producing the same general results every time. The reason for this is hard to deduce. Memory usage rises at at reasonably monotonic rate, so the drastic speed increase is not accompanied by a memory decrease. Logging the interactions with the SMT-solver, we can see an expected increase in total interactions. From these metrics we can reasonably rule out bugs or shortcuts taken in the engine itself. It is more likely that the SMT solver is saving time in some way or another, perhaps by reusing observations made at a deeper search depth in later queries as a form of solver caching.

Moving on to memory consumption, the patterns become much more irregular. The most dominating pattern is one of sudden sharp increases in memory consumption followed by short plateaus. Fibonacci, loop-based GCD, and the triangular sum seem to be increasing at a somewhat linear rate. However, as testing higher ranges is prohibitively time-consuming, we are not able to find patterns which can be extrapolated to greater search depths, aside from fact that usage will increase with hundreds of megabytes, if not thousands. As an example, the recursive GCD seems to show a linear increase until the 170-mark, where the usage suddenly spikes. We cannot confidently claim that the loop-based GCD wont exhibit the same behavior at, say, the 250 instruction mark. The memory usage results for the Collatz function tells the story clearly in regards to worst case memory performance. The usage may grow at a non-linear rate to half a gigabyte while we're still searching to a depth of only a few hundred instructions.

bench_results/fibo_loop_time.pdf

(a) Loop-based Fibonacci

bench_results/fibo_rec_time.pdf

(b) Recursive Fibonacci

bench_results/gcd_loop_time.pdf

(c) Loop-based greatest common divisor

bench_results/gcd_rec_time.pdf

(d) Recursive greatest common divisor

bench_results/collatz_loop_time.pdf

(e) Collatz conjecture

bench_results/triangular_loop_time.pdf

(f) Triangular sum

Figure 5: Execution times of various mathematical functions

bench_results/fibo_loop_memory.pdf

(a) Loop-based Fibonacci

bench_results/fibo_rec_memory.pdf

(b) Recursive Fibonacci

bench_results/gcd_loop_memory.pdf

(c) Loop-based greatest common divisor

bench_results/gcd_rec_memory.pdf

(d) Recursive greatest common divisor

bench_results/collatz_loop_memory.pdf

(e) Collatz conjecture

bench_results/triangular_loop_memory.pdf

(f) Triangular sum

Figure 6: Memory usage of various mathematical functions

8 Execution techniques & future work

This section covers some common execution strategies and evaluates their viability for usage with wasym or WebAssembly in general.

8.1 Eager versus lazy evalution

wasym already performs eager evaluation of constraints. Changing the algorithm to perform lazy evalution would be an easy and straightforward task. Constraint evalution could be set to either exclusively happen when checking for symbolic satisfiability, or could be set to happen at an interval or as a result of some check. These refactors can be done in a few lines of code.

8.2 Concurrency

The most straightforward technique to implement is to multi-thread the search. wasym is implemented in such a way that copying runtime state is trivial, and subdividing the decision tree is an easy task. There may be roadblocks when it comes to copying SMT solver state, however. Also, if paired with other techniques, one must take care to make sure that parallel threads are divergent and gain insight from each others work. Otherwise, one might end up running multiple threads that perform redundant work which could have been avoided.

8.3 Code-tagging strategies

Many techniques for dealing with path explosion will bind data to the syntax tree of a program. For example, a subpath-guided search tracks how often parts of a program have been executed and selects the least-traversed paths, while path subsumption adds new constraints to branch points when backtracking.

These types of techniques are hindered by the architecture of wasym. wasym treats the syntax tree as a static global and consumes the tree while navigating through it. Since Haskell maintains referential transparency, dynamically associating data with specific locations in the program is a challenge, but there *are* ways to modify a structure while traversing it. The most straightforward method is to include a modified structure in the return value. A more sophisticated method is to implement a Zipper pattern, which tracks the data needed to reconstruct the structure while delving deeper into it. Both methods run into issues when mixed with multi-threading. Using a language with references and fewer guarantees could enable code-tagging techniques with much more ease.

8.4 Concretization & concolic execution

Quite many techniques rely on concretization of values. There is not anything in the way of doing concretization in wasym. But wasym is not built to perform offline execution (where execution takes a single path and is restarted for every new branch), and there is no general strategy for incorporating concolic execution techniques. It could be beneficial to implement a general strategy or framework for performing concretization.

8.5 Symbolic memory

As WebAssembly supports memory as a single contiguous array, heap-based symbolic modeling of memory is out of the question. An advantage of Wasm's memory model is that memory is non-existent by default, a small size of 64KiB if declared, and only larger than that if explicitly requested to be. No implementation details of wasym get in the way of modeling this memory with an SMT array theory, address concretization, partial address concretization, or some if-then-else based approach. Which of these approaches works best should be decided experimentation.

8.6 Environment

When it comes to handling environments, WebAssembly has a similar advantage: it doesn't have any environment to interact with by default. If one wishes to interact with an environment at all, it must be done through function imports. Granted, functions can be imported to provide any interaction with the environment as needed, but this must be done as a conscious decision. The dividing line between internal and external code is made very clear.

Furthermore, a system interface called WASI is being developed for Wasm. This interface is forced to be generic, as it must behave the same way whether it is interacting with a browser sandbox or a kernel. It may be possible to implement a symbolic model of WASI, but evaluating the viability of such a model merits its own project.

Methods using concretization and virtualization are both viable strategies for representing environments in wasym, but this is entirely dependent on the APIs being exposed to a module. It may not be straightforward to interface with foreign functions in Haskell, but it certainly is possible.

9 Evaluation

9.1 wasym

The product of this project, wasym, serves its purpose well as a demonstration of symbolic execution. It is very limited in its capabilities, supporting only a subset of instructions which excludes many floating point and memory operations. The execution cost of running a query scales exponentially and irregularly with the search depth. Memory usage grows to hundreds of megabytes and could possibly reach the thousands if the engine wasn't bottlenecked by execution time. Despite these limitations, wasym does constitute an end-to-end symbolic execution engine with the capability to read, parse, execute and validate properties for a given WebAssembly module.

For the project, I chose to write in Haskell using SBV as an SMT backend. This did have some positive outcomes. For one, the flexible and modular approach to building the interpreter was provided by the stateful monad transformer stacks that Haskell supports. Having SMT queries be type-checked and represented as values was also a large gain, as it did a lot of the heavy lifting needed to construct queries and parse results. However, Haskell is not very friendly to those that wish to profile and optimize memory usage, and the layer of abstraction in the SBV API is removed from the SMT-LIB specification to such a degree that I lost some control over the queries. The mapping from SBV component to SMT query was weaker and less intuitive. For example, when an attempt to implement memory using array theory and bit-vector slicing wasn't working, the resulting error messages were undocumented and internal to SBV. Instead of turning to the larger SMT community for resources documenting arrays, I could only scour through sparse discussions and Github issues on the topic in the context of SBV. Going with a systems language such as Rust and interfacing more directly with SMT-LIB might have had a larger upfront cost, but could have paid off towards the end of the project.

9.2 WebAssembly for symbolic execution

WebAssembly does seem to be a suitable target for symbolic execution. The fact that the language is designed to be embedded in browser applications means that Wasm is a simple and sandboxed, but capable language. Interaction with the environment is entirely dictated by the embedder, and while any API can be exposed to a module, the target platform of the web encourages writing programs with few external dependencies. The structure of the language is designed with a structure and validation rules such that there are very few ways for a valid WebAssembly module to fail while executing. Moreover, control flow is explicitly stated; loops are all enclosed in a `loop` block, if-then-else has its own control structure, local variables are declared at the very start of a function, and so on. This makes analysis of the program predictable, paving the way for techniques such as loop unrolling and state merging.

The simplicity of WebAssembly comes at a cost, however. Complex error systems, heap-based memory management, garbage collection, and similar high level language features are not natively supported and must be implemented as runtime libraries. The result is that we lose the ability to model these concepts symbolically in a language-agnostic fashion. For example, different languages might use different allocators with differing APIs. If we wish to symbolically model the heap, we may need to tailor this feature to multiple different allocators. With static compilation and optimizations, the boundary between allocator and program may be broken, and this would make reliable heap simulations near impossible. This goes for all such high-level features. If the WebAssembly specification was extended to provide opt-in native support for some of these high-level features, it would be possible to model them universally.

10 Conclusion

This project has explored the possibility of performing symbolic execution of the WebAssembly language. The product of the project is a symbolic execution engine, wasym, which supports a subset of the WebAssembly language. The engine is inefficient, but demonstrates the feasibility of the project subject. WebAssembly is deemed to be a promising language for symbolic execution, due to its simplicity, portability, rigorousness, and separation from any environment. There is plenty of room for future work in testing the effectiveness of execution techniques with WebAssembly.

References

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3), 2018.

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard*, 2.6 edition, 7 2017.

[3] Stack contributors. *The Haskell Tool Stack*, 2020. `https://docs.haskellstack.org/en/stable/README/`.

[4] WebAssembly Community Group. Webassembly home page. `https://webassembly.org/`. Accessed: 2020-03-31.

[5] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. 1995. `http://web.cecs.pdx.edu/~mpj/pubs/springschool95.pdf`.

[6] Andreas Rossberg. *WebAssembly Specification*. WebAssembly Community Group, 1.0 edition, 3 2020.

A Appendix - wasym code

A.1 Main.hs

```
import ParserWASM
import AST
import Assertions
import Machine
import Search

import Data.Word (Word64)
import Data.Function ((&))
import Data.SBV (z3, runSMTWith)
import System.Environment (getArgs)
import qualified Data.ByteString as BS
import Text.Read (readMaybe)

usageString :: String
usageString = "\
\usage: wasym MODULE FUNCTION DEPTH_LIMIT [ASSERTION+]\
\\n\
\\nlimit can be either \'none\' or an integer\
\\npossible assertions:\
\\n    none\
\\n    unreachable\
\\n    intDivZero"

main :: IO ()
main = do
  args <- getArgs
  case args of
    modulePath : funcName : limit : asserts -> do
      mdl <- readModule modulePath
      let funcIdx = getFunc mdl funcName
      let parsedLimit = readMaybe limit
      let assertion = parseAsserts asserts
      case funcIdx of
        Nothing -> putStrLn ("function not found:'" ++ funcName ++ "'")
        Just idx -> execute mdl idx parsedLimit assertion
    _ -> putStrLn usageString

execute :: Module -> Idx -> Maybe Word64 -> Predicate -> IO ()
execute mdl funcIdx limit p =
    (
      makeConfig mdl funcIdx p >>= \ cfg ->
      startState >>= \ st ->
      evalMachine (searchFunc funcIdx) st (cfg {maxDepth = limit})
    )
    & runSMTWith z3 >>= \ smtRes -> case smtRes of
      Nothing -> putStrLn "assertions held"
      Just ass ->
        putStrLn "assertions failed with the following value assignments:" *>
        putStrLn ass

readModule :: String -> IO Module
readModule filename = do
  file <- BS.readFile filename
  let parseResult = parseWASM filename file
  case parseResult of
    Left err -> error err
    Right m -> pure m
```

A.2 AST.hs

```
module AST where

import Data.Word
import qualified Data.Vector as V

type Byte = Word8
type Idx = Word32
```

```haskell
type Vec = V.Vector

-- This module implements an abstract syntax tree in accordance with the
-- structure section of the WASM spec 1.0
-- To be used as static data for the interpreter

-- TODO:
-- * Support symbolic identifiers from the textual format, for user-friendliness
-- * Are external types necessary?

-- * This type, FuncRef, figure out what it is and should be:
-- I'm guessing it's for foreign function interfaces.
data FuncRef = Undefined deriving (Show)

data Module = Module {
  types   :: Vec FuncType,
  funcs   :: Vec Func,
  tables  :: Vec TableType,
  mems    :: Vec MemType,
  globals :: Vec Global,
  elems   :: Vec Elem,
  dat     :: Vec Data,
  start   :: Maybe StartFunc,
  imports :: Vec Import,
  exports :: Vec Export
} deriving (Show)

{----- Component types -----}
data Func = Func {
  fType :: FuncType,
  locals :: Vec ValType,
  body :: Expr
} deriving (Show)

data Global = Global {
  myType :: GlobalType,
  globalInit :: Expr
} deriving (Show)

data Elem = Elem {
  elemRange :: Idx, -- only current valid index is 0
  elemOffset :: Expr, -- must be constant
  elemInit :: Vec Idx
} deriving (Show)

data Data = Data {
  dataRange :: Idx,
  dataOffset :: Expr,
  dataInit :: Vec Byte
} deriving (Show)

-- Function index of start/initialization function
type StartFunc = Idx

data Export = Export Name ExportDesc deriving (Show)

data ExportDesc =
    ExportFunc Idx
  | ExportTable Idx
  | ExportMem Idx
  | ExportGlobal Idx
  deriving (Show)

data Import = Import {
  moduleName :: Name,
  importName :: Name,
  importDesc :: ImportDesc
} deriving (Show)

data ImportDesc =
    ImportFunc Idx
  | ImportTable Idx
  | ImportMem Idx
```

```
  | ImportGlobal Idx
  deriving (Show)

{----- Lesser types -----}
type ValType = (NumType, BitWidth)
data NumType = NumInt | NumFlt deriving (Show, Eq)
data BitWidth = W32 | W64 deriving (Show, Eq)
data Sign = SN | US deriving (Show, Eq)

type ResType = Maybe ValType

type Name = String

data FuncType = FuncType {
  funcParams :: Vec ValType,
  funcResult :: Maybe ValType
} deriving (Show)

data Limits = Limits {
  min :: Word32,
  max :: Maybe Word32
} deriving (Show)

type MemType = Limits

data TableType = TableType Limits ElemType deriving (Show)
type ElemType = FuncRef

data GlobalType = GlobalType {
  mutable :: Bool,
  globalType :: ValType
} deriving (Show)

type Expr = Vec Instr

{--
  It is possible to create malformed instructions, but we leave it to the
  parser not to do so.
--}

data Instr =
-- Control Instructions
   Unreachable
 | Nop
 | Block ResType Expr
 | Loop ResType Expr
 | If ResType Expr Expr
 | Branch Idx
 | BranchIf Idx
 | BranchTable (Vec Idx) Idx
 | Return
 | Call Idx
 | CallIndirect Idx
-- Parametric Instructions
 | Drop
 | Select
-- Variable Instructions
 | LocalGet Idx
 | LocalSet Idx
 | LocalTee Idx
 | GlobalGet Idx
 | GlobalSet Idx
-- Memory Instructions
 | MemorySize
 | MemoryGrow
 | Load ValType MemArg
 | Store ValType MemArg
 | IntLoad8 BitWidth Sign MemArg
 | IntLoad16 BitWidth Sign MemArg
 | IntLoad32 Sign MemArg
 | IntStore8 BitWidth MemArg
 | IntStore16 BitWidth MemArg
 | IntStore32 MemArg
```

```
-- Numerical Instructions
 | Constant Value
 | IUnary BitWidth IUnOp
 | IBinary BitWidth IBinOp
 | ICompare BitWidth IRelOp
 | FUnary BitWidth FUnOp
 | FBinary BitWidth FBinOp
 | FCompare BitWidth FRelOp
-- More Numerical Instructions
 | WrapI64
 | ExtendI32 Sign
 | IntTruncFlt BitWidth BitWidth Sign
 | FltDemote
 | FltPromote
 | Flt2Int BitWidth BitWidth Sign
 | Reinterpret ValType ValType
 deriving (Show, Eq)

data Value =
    I32Val Word32
  | I64Val Word64
  | F32Val Float
  | F64Val Double
  deriving (Show, Eq)

data MemArg = MemArg {
  memOffset :: Word32,
  memAlign :: Word32
} deriving (Show, Eq)

data IUnOp =
  IClz | ICtz | IPopCnt | IEQZ
  deriving (Show, Eq)

data IBinOp =
  IAdd | ISub | IMul | IDiv Sign | IRem Sign |
  IAnd | IOr | IXOr | IShiftL | IShiftR Sign | IRotL | IRotR
  deriving (Show, Eq)

data IRelOp =
  IEQ | INE | ILT Sign | IGT Sign | ILE Sign | IGE Sign
  deriving (Show, Eq)

data FUnOp =
  FAbs | FNeg | FSqrt | FCeil | FFloor | FTrunc | FNearest
  deriving (Show, Eq)

data FBinOp =
  FAdd | FSub | FMul | FDiv | FMin | FMax | FCopySign
  deriving (Show, Eq)

data FRelOp =
  FEQ | FNE | FLT | FGT | FLE | FGE
  deriving (Show, Eq)

--------------------------------------------------------------------------------
-- * Utility
--------------------------------------------------------------------------------
unconsV :: V.Vector a -> Maybe (a, V.Vector a)
unconsV vec =
  if V.null vec
    then Nothing
    else Just (V.head vec, V.tail vec)

getFunc :: Module -> String -> Maybe Idx
getFunc mdl funcName =
  (\ (Export _ (ExportFunc idx)) -> idx) <$>
  V.find (matchExportFunc funcName) (exports mdl)

matchExportFunc :: String -> Export -> Bool
matchExportFunc funcName (Export name (ExportFunc _)) | funcName == name = True
matchExportFunc _ _ = False
```

A.3 Machine.hs

```
module Machine where

import Data.Functor ((<&>))
import Control.Arrow ((>>>))
import Control.Applicative

import Data.Word (Word64)
import Data.Maybe (isJust)
import qualified Data.Vector as V

import Control.Monad.State
import Control.Monad.Reader
import Control.Monad.Trans.Maybe

import Data.SBV hiding (Predicate)
import Data.SBV.Control

import AST
import SValue

type Predicate = WASMState -> (Bool, SBool)
type Precondition = Config -> SBool

--------------------------------------------------------------------------------
-- * Configuration and runtime state
--------------------------------------------------------------------------------

type Machine a = ReaderT Config (StateT WASMState (MaybeT Query)) a

data Config = Config {
  wasmModule :: Module,
  symbols :: Vec SValue,
  maxDepth :: Maybe Word64,
  predicate :: Predicate
}

type Stack a = [a]
type Vars = Vec SValue

data WASMState = WASMState {
  frames :: Stack Frame,
  memory :: SArray Word32 Word8,
  depth :: Word64
}

data Frame = Frame {
  vars :: Vec SValue,
  contexts :: Stack Context,
  res :: Maybe ValType
}

data Context = Context {
  code :: Expr,
  stack :: Stack SValue,
  ctxLabel :: Instr,
  arity :: Bool
}

startState :: Symbolic WASMState
startState =
  newArray "memory" (Just 0) <&> \ mem ->
  WASMState {
    frames = [],
    memory = mem,
    depth = 0
  }

initMemory :: Int -> WASMState -> Symbolic WASMState
initMemory numBytes st@(WASMState {memory = mem}) =
  mkFreeVars numBytes <&> \ symBytes ->
  let writeFunc addr val mem = writeArray mem addr val in
```

```
  let modFunc = foldl (.) id $ zipWith writeFunc symBytes [0..] in
  st {memory = modFunc mem}

--------------------------------------------------------------------------------
-- * Machine operations
--------------------------------------------------------------------------------

-- Invokes a specific function. Can be called without prior setup
invoke :: Vars -> Idx -> Machine ()
invoke args idx =
  asks (wasmModule >>> funcs >>> (V.! fromIntegral idx)) >>=
  \ Func {fType = FuncType _ res, locals = lcs, body = bd} ->
  let frameVars = args <|> (mkZero <$> lcs) in
  let frameCtx =
       Context {
         code = bd,
         stack = empty,
         ctxLabel = Nop,
         arity = isJust res
       } in
  pushFrame $
  Frame {
    vars = frameVars,
    contexts = pure frameCtx,
    res = res
  }

-- | Prints a message using IO
mPrint :: String -> Machine ()
mPrint msg =
  lift . lift . lift . io $ putStrLn msg

-- | Retrieve the top frame from the execution stack
peekFrame :: Machine Frame
peekFrame =
  gets frames <&> \ fs ->
  case fs of
    [] -> error "Attempt to peek frame from empty stack"
    f:_ -> f

-- | Pop off the top frame from the execution stack and return it
popFrame :: Machine Frame
popFrame =
  state $ \ ws ->
  case frames ws of
    (f:fs) -> (f, ws {frames = fs})
    _ -> error "Attempt to pop frame from empty stack"

-- | Push a new frame to the execution stack
pushFrame :: Frame -> Machine ()
pushFrame f =
  modify $ \ ws -> ws {frames = f : (frames ws)}

-- | Modify the current top frame on the execution stack
modifyFrame :: (Frame -> Frame) -> Machine ()
modifyFrame fn =
  (popFrame <&> fn) >>= pushFrame

-- | Push a context to the execution stack
pushContext :: Context -> Machine ()
pushContext ctx =
  modifyFrame $ \ frame ->
  frame {contexts = ctx : (contexts frame)}

 -- | Pop the top context on the top frame.
popContext :: Machine Context
popContext =
  popFrame >>= \ frame ->
  case contexts frame of
    [] -> empty
    ctx : rest ->
      (pushFrame $ frame {contexts = rest}) <&> const ctx

```

```haskell
-- | Retrieve the top context on the top frame.
peekContext :: Machine Context
peekContext =
  popContext >>= \ ctx -> pushContext ctx <&> const ctx

-- | Pop an instruction from the code of the top context, on the top frame.
popInstr :: Machine Instr
popInstr =
  popContext >>= \ ctx ->
  case unconsV (code ctx) of
    Nothing -> error "Attempt to pop instruction from empty expr"
    Just (instr, expr) -> pushContext (ctx {code = expr}) *> pure instr

-- | Push a value, to the stack of the top context, on the top frame.
pushVal :: SValue -> Machine ()
pushVal v =
  (popContext <&> \ ctx -> ctx {stack = v : (stack ctx)}) >>= pushContext

-- | Pop a value, from stack of the top context, on the top frame.
popVal :: Machine SValue
popVal =
  popContext >>= \ ctx ->
  case stack ctx of
    [] -> empty
    v:rest -> (pushContext $ ctx {stack = rest}) <&> const v

-- | Retrieve the current value from the top of the stack
peekVal :: Machine SValue
peekVal =
  peekContext >>= \ ctx ->
  case stack ctx of
    [] -> empty
    v:_ -> pure v

-- | Retrieve the local values of the top frame.
peekVars :: Machine (Vec SValue)
peekVars =
  peekFrame <&> vars

-- | Modify the local values of the top frame with a given function.
modifyVars :: (Vec SValue -> Vec SValue) -> Machine ()
modifyVars f =
  modifyFrame (\ frame -> frame {vars = f $ vars frame})
```

A.4 Exec.hs

```haskell
module Exec where

import Data.Function ((&))
import Data.Functor ((<&>))
import Control.Applicative
import Control.Arrow ((>>>))

import Data.Maybe (isJust)
import qualified Data.Vector as V

import Control.Monad.State
import Control.Monad.Reader

import Data.SBV

import AST
import Machine
import SValue

-- | Ends finished blocks and returns from finished functions
execFallthrough :: Machine ()
execFallthrough =
  -- Get the contexts and top context
  gets frames >>= \ frames ->
  -- Block the execution path if no frames left on stack
  -- TODO: maybe put in ‘guardPath‘?
  if length frames == 0 then empty else
```

```haskell
  let ctxs = contexts (head frames) in
  let ctx = head ctxs in
  -- If the code for the current context has fully executed
  if V.length (code ctx) > 0 then pure () else
  (
    -- If there is only this context, return from the function
    if length ctxs == 1 then execInstr Return else
    -- Else, end the block without executing the label WRONG
    popContext <&> stack >>= \ st ->
    case st of
      [x] -> pushVal x
      [] -> pure ()
      _ -> error "stack too large when exiting block"
  ) *>
  -- Repeat in case there are more fallthroughs to perform
  execFallthrough

execInstr :: Instr -> Machine ()
execInstr instr =
  execInstr' instr *> execFallthrough

execInstr' :: Instr -> Machine ()
execInstr' instr = case instr of
  Nop -> pure ()

  Unreachable -> error "Unreachable instruction was reached"

  Drop -> () <$ popVal

-- | Pushes the specified value to the stack.
  Constant v ->
    case v of
      I32Val x -> pushVal . SW32 . literal $ x
      I64Val x -> pushVal . SW64 . literal $ x
      F32Val x -> pushVal . SF32 . literal $ x
      F64Val x -> pushVal . SF64 . literal $ x

-- | Pops a value, performs a unary operation, pushes the result.
  --IUnary tp op -> popVal >>= (evalIUnOp op >>> pushVal)

-- | Pops two values and performs a binary operation on them. Note that the
-- first operand is the second value to be popped!
  IBinary _ op -> (flip (evalIBinOp op) <$> popVal <*> popVal) >>= pushVal

-- | Pops two values and performs a comparison operation between two integers.
  ICompare _ op -> (flip (evalIRelOp op) <$> popVal <*> popVal) >>= pushVal

  FBinary _ op -> (flip (evalFBinOp op) <$> popVal <*> popVal) >>= pushVal

-- | Reads the specified local value and pushes it to the stack.
  LocalGet idx -> (peekVars <&> (V.! (fromIntegral idx))) >>= pushVal

-- | Pops a value and writes it to the specified local.
  LocalSet idx -> popVal >>= \ v -> modifyVars (V.// [(fromIntegral idx, v)])

-- | Peeks at the top value and writes it to the specified local.
  LocalTee idx -> peekVal >>= \ v -> modifyVars (V.// [(fromIntegral idx, v)])

-- | Constructs a new context and pushes it to the top
  Block res body ->
    pushContext $ Context {
      code = body,
      stack = empty,
      ctxLabel = Nop,
      arity = isJust res
    }

-- | Constructs a new context and pushes it to the top. The label is set to
-- another loop
  Loop res body ->
    pushContext $ Context {
      code = body,
      stack = empty,
```

```
      ctxLabel = Loop res body,
      arity = False
    }

  Branch n ->
    let idx = fromIntegral n in
    -- Create an action that will pop n + 1 contexts
    let popContexts = sequenceA $ replicate (idx + 1) popContext in
    -- Retrieve the last context to pop (inefficient lookup)
    -- Maybe we could pop the contexts into a vector to avoid the lookahead
    peekFrame <&> contexts <&> (!! idx) >>= \ ctx ->
    -- If the last context has a result value, pop a value, pop the contexts,
    -- push the value back on the stack
    if arity ctx
      then popVal <* popContexts <* execInstr (ctxLabel ctx) >>= pushVal
      else popContexts *> execInstr (ctxLabel ctx)

  Call idx ->
    -- Retrieve function by index, unpack
    asks (wasmModule >>> funcs >>> (V.! fromIntegral idx)) >>=
    \ Func {fType = FuncType params res, locals = lcs, body = bd} ->
    -- Pop values corresponding to the number of arguments
    sequenceA (V.replicate (V.length params) popVal) <&>
    -- Reverse and concatenate to n zero-values for every local
    V.reverse <&> (V.++ (mkZero <$> lcs)) >>= \ newVars ->
    -- Push a frame with the locals.
    pushFrame ( Frame {vars = newVars, contexts = [], res = res} ) *>
    -- Push a context with body and return type of the function
    execInstr (Block res bd)

  Return ->
    peekFrame <&> res >>= \ res ->
    if isJust res
      then popVal <* popFrame >>= pushVal
      else () <$ popFrame

  Load valType (MemArg offset _) ->
    case valType of
      (NumInt, W32) -> 4
      (NumInt, W64) -> 8
      _ -> error "float memory operations unsupported"
    & \ numBytes ->
    popVal >>= \ v ->
    gets memory >>= \ mem ->
    case v of
      SW32 addr ->
        vFromBytes valType (
          readArray mem <$>
          [addr + literal offset .. numBytes + addr + literal offset - 1]
        )
        & pushVal
      _ -> error "popped value wasn't an address"

  Store valType (MemArg offset _) ->
    case valType of
      (NumInt, W32) -> 4
      (NumInt, W64) -> 8
      _ -> error "float memory operations unsupported"
    & \ numBytes ->
    popVal >>= \ addr ->
    popVal <&> vToBytes >>= \ valBytes ->
    let writeFunc addr byte mem = writeArray mem addr byte in
    case addr of
      SW32 addr ->
        let writeToMem = foldl (.) id $ zipWith writeFunc [addr + literal offset ..] valBytes
        in modify (\ st @ (WASMState {memory = mem}) -> st {memory = writeToMem mem})
      _ -> error "popped value wasn't an address"

  _ -> error $ "unsupported instruction: " ++ show instr
```

A.5 SValue.hs

```haskell
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE DataKinds #-}

module SValue where

import Data.SBV

import AST

data SValue =
    SW32 SWord32
  | SW64 SWord64
  | SF32 SFloat
  | SF64 SDouble

--------------------------------------------------------------------------------
--- * Creation
--------------------------------------------------------------------------------

mkSValue :: ValType -> Symbolic SValue
mkSValue t =
  case t of
    (NumInt, W32) -> SW32 <$> free_
    (NumInt, W64) -> SW64 <$> free_
    (NumFlt, W32) -> SF32 <$> free_
    (NumFlt, W64) -> SF64 <$> free_

-- | Creates a concrete value of 0 corresponding to the given type
mkZero :: ValType -> SValue
mkZero t =
  case t of
    (NumInt, W32) -> SW32 0
    (NumInt, W64) -> SW64 0
    (NumFlt, W32) -> SF32 0
    (NumFlt, W64) -> SF64 0

--------------------------------------------------------------------------------
-- * Operations
--------------------------------------------------------------------------------
type UnOp = SValue -> SValue

type BinOp = SValue -> SValue -> SValue

--evalIUnOp :: IUnOp -> SValue -> SValue
--evalIUnOp op =
--  case op of
--    IClz ->
--    ICtz ->
--    IPopCnt ->
--    IEQZ ->

boolToNum :: SBool -> SValue
boolToNum v = SW32 $ ite v 1 0

getIBinOp :: ( SymVal a,
               Integral a,
               Bits a,
               SDivisible (SBV a)
             ) =>
             IBinOp -> SBV a -> SBV a -> SBV a
getIBinOp op =
  case op of
    IAdd -> (+)
    ISub -> (-)
    IMul -> (*)
    IDiv US -> sDiv
    IRem US -> sRem
    IAnd -> (.&.)
    IOr -> (.|.)
    IXOr -> xor

getIRelOp :: (SymVal a, Ord a) =>  IRelOp -> SBV a -> SBV a -> SValue
```

```
getIRelOp op x y =
  case op of
    IEQ ->    boolToNum (x .== y)
    INE ->    boolToNum (x ./= y)
    ILT US -> boolToNum (x .<  y)
    IGT US -> boolToNum (x .>  y)
    ILE US -> boolToNum (x .<= y)
    IGE US -> boolToNum (x .>= y)

evalIRelOp :: IRelOp -> SValue -> SValue -> SValue
evalIRelOp op a b =
  case (a, b) of
    (SW32 x, SW32 y) -> getIRelOp op x y
    (SW64 x, SW64 y) -> getIRelOp op x y

evalIBinOp :: IBinOp -> BinOp
evalIBinOp op a b =
  case (a, b) of
    (SW32 x, SW32 y) -> SW32 $ getIBinOp op x y
    (SW64 x, SW64 y) -> SW64 $ getIBinOp op x y
    _ -> error "type mismatch in binary int operation"

getFBinOp :: (IEEEFloating a) => FBinOp -> SBV a -> SBV a -> SBV a
getFBinOp op =
  case op of
    FAdd -> fpAdd sRoundNearestTiesToEven
    FSub -> fpSub sRoundNearestTiesToEven
    FMul -> fpMul sRoundNearestTiesToEven
    FDiv -> fpDiv sRoundNearestTiesToEven
    FMin -> fpMin
    FMax -> fpMax

evalFBinOp :: FBinOp -> BinOp
evalFBinOp op a b =
  case (a, b) of
    (SF32 x, SF32 y) -> SF32 $ getFBinOp op x y
    (SF64 x, SF64 y) -> SF64 $ getFBinOp op x y
    _ -> error "type mismatch in binary float operation"

--evalFRelOp :: FBinOp -> BinOp
--evalFRelOp op a b =
--  case (a, b) of
--    (SF32 a, SF32 y) -> SW32 $ getFRelOp op x y
--    (SF64 a, SF64 y) -> SW32 $ getFRelOp op x y

vToBytes :: SValue -> [SWord8]
vToBytes val =
  case val of
    SW32 x -> map fromSized . toBytes . toSized $ x
    SW64 x -> map fromSized . toBytes . toSized $ x
    _ -> error "cannot convert between symbolic bytes and floats"

vFromBytes :: ValType -> [SWord8] -> SValue
vFromBytes valType =
  case valType of
    (NumInt, W32) ->
      SW32 . fromSized . (fromBytes :: [SWord 8] -> SWord 32) . map toSized
    (NumInt, W64) ->
      SW64 . fromSized . (fromBytes :: [SWord 8] -> SWord 64) . map toSized
    _ -> error "cannot convert between symbolic bytes and floats"
```

A.6 Search.hs

```
module Search (searchFunc, evalMachine, makeConfig, makeVars, findSat) where

import qualified Data.Vector as V
import Data.Function ((&))
import Data.Functor ((<&>))
import Control.Applicative ((<|>), empty)

import Data.SBV hiding (Predicate)
import Data.SBV.Trans.Control

```

```haskell
import Control.Monad.Trans
import Control.Monad.State.Lazy
import Control.Monad.Trans.Reader
import Control.Monad.Trans.Maybe

import AST
import Machine
import Exec
import SValue

type Result = String

-------------------------------------------------------------------------------
-- * Program execution
-------------------------------------------------------------------------------

searchFunc :: Idx -> Machine Result
searchFunc funcIdx =
  asks symbols >>= \ syms ->
  invoke syms funcIdx *> findSat

-- | Creates a config
makeConfig :: Module -> Idx  -> Predicate -> Symbolic Config
makeConfig mdl funcIdx searchTerm =
  let argTypes = funcParams . fType $ funcs mdl V.! fromIntegral funcIdx in
  makeVars argTypes <&> \ syms ->
  Config {
    wasmModule = mdl,
    maxDepth = Just 1000,
    predicate = searchTerm,
    symbols = syms
  }

-- | Unpacks the machine stack
evalMachine :: Machine a -> WASMState -> Config -> Symbolic (Maybe a)
evalMachine machine st =
  query .
  runMaybeT .
  (`evalStateT` st) .
  runReaderT (
    machine
  )

-- | Creates a list of symbolic variables given their names
makeVars :: Vec ValType -> Symbolic (Vec SValue)
makeVars types =
  setOption (ProduceUnsatCores True) *>
  (V.map mkSValue types & sequenceA)

-------------------------------------------------------------------------------
-- * State space search
-------------------------------------------------------------------------------

-- | Attempts to find input values which will eventually put a WASM machine into
-- a state satisfying the given predicate.
findSat :: Machine Result
findSat =
  -- Apply the predicate to our state and push the resulting constraint
  asks predicate <*> get >>= \ (concretePred, symbolicPred) ->
  if not concretePred then handleBranch else
  pushConstraint symbolicPred *>
  -- Query the SMT solver about the symbolic value
  checkPath >>= \ satPred ->
  if satPred
    -- If satisfied, return the assignments
    then getResult
    -- If not, move forward
    else popConstraint *> handleBranch

-- | Handles searching branches if necessary, otherwise just exec
handleBranch :: Machine Result
handleBranch =
  -- If a maximum depth is reached, stop the search
```

```
  gets depth >>= \ d ->
  asks maxDepth >>= \ maxd ->
  if maxd == Just d
    then {--mPrint "maximum depth reached" *>--} empty
    else modify (\ cfg -> cfg { depth = d + 1 }) *>

  popInstr >>= \ instr ->
  --mPrint (show instr) *>
  case instr of
    BranchIf n ->
      popVal >>= \ v ->
      case v of
        SW32 x ->
          (
            ( pushConstraint (x .== 0) *>
              guardPath *>
              execInstr Nop *>
              findSat
            ) &
            mapQ (<* pop 1)
          ) <|> (
            ( pushConstraint (x ./= 0) *>
              guardPath *>
              execInstr (Branch n) *>
              findSat
            ) &
            mapQ (<* pop 1)
          )
        _ -> error "attempt to branch on value that isn't i32"
    If res body1 body2 ->
      popVal >>= \ v ->
      case v of
        SW32 x ->
          (
            ( pushConstraint (x ./= 0) *>
              guardPath *>
              execInstr (Block res body1) *>
              findSat
            ) &
            mapQ (<* pop 1)
          ) <|> (
            ( pushConstraint (x .== 0) *>
              guardPath *>
              execInstr (Block res body2) *>
              findSat
            ) &
            mapQ (<* pop 1)
          )
        _ -> error "attempt to branch on value that isn't i32"
    _ -> execInstr instr *> findSat

-- | Checks the current constraints for satisfiability and nullifies this
-- execution path if it isn't feasible.
guardPath :: Machine ()
guardPath =
  checkPath >>= \ satPath ->
  if not satPath
    then (getUnsatCore >>= traverse mPrint) *>
         --mPrint "unsatisfiable branch" *>
         empty
    else pure ()

getResult :: Machine Result
getResult =
  asks symbols >>= sequenceA . V.map getSValue <&> show

getSValue :: SValue -> Machine Value
getSValue (SW32 x) = I32Val <$> getValue x
getSValue (SW64 x) = I64Val <$> getValue x
getSValue (SF32 x) = F32Val <$> getValue x
getSValue (SF64 x) = F64Val <$> getValue x

checkPath :: Machine Bool
```

```haskell
checkPath =
  checkSat <&> \ res -> case res of
    Sat -> True
    Unsat -> False
    Unk -> error "Satisfiability unknown."

pushConstraint :: SBool -> Machine ()
pushConstraint c =
  push 1 *> liftQ (constrain c)

popConstraint :: Machine ()
popConstraint = pop 1

mapQ :: (Query (Maybe (a, WASMState)) -> Query (Maybe (b, WASMState))) ->
        Machine a -> Machine b
mapQ = mapReaderT . mapStateT . mapMaybeT

liftQ :: Query a -> Machine a
liftQ = lift . lift . lift
```

A.7 ParserWASM.hs

```haskell
module ParserWASM (parseWASM, moduleP) where

import Text.Megaparsec
import Text.Megaparsec.Byte

import Data.Function ((&))
import Data.Functor ((<&>))
import Data.Bifunctor (first)
import Control.Monad (join)
import Data.Word (Word32, Word64)
import qualified Data.Vector as V
import Data.Either (fromRight)

import qualified Data.ByteString as B
import Data.Text.Encoding (decodeUtf8)
import Data.Serialize.IEEE754
import Data.Serialize.Get (runGet)
import qualified Data.Text as Text (unpack)

import AST
import ParserInstr

type Parser a = Parsec ParserWASMError B.ByteString a

data ParserWASMError = ULEB128 deriving (Show, Eq, Ord)

instance ShowErrorComponent ParserWASMError where
  showErrorComponent = show
--------------------------------------------------------------------------------
-- * Primitives
--------------------------------------------------------------------------------

unsignedLEB128 :: Integral a => Integer -> Parser a
unsignedLEB128 size =
  fromIntegral <$> anySingle >>= \ n ->
  if n < 128 && ((fromIntegral n) :: Integer) < 2 ^ size
    then pure n
  else if size > 7
    then (\ m -> 128 * m + n - 128) <$> unsignedLEB128 (size - 7)
  else customFailure ULEB128

u32 :: Parser Word32
u32 = unsignedLEB128 32 <?> "u32"

u64 :: Parser Word64
u64 = unsignedLEB128 64 <?> "u64"

f32 :: Parser Float
f32 =
  takeP Nothing 4 <&>
  runGet getFloat32le <&>
```

```
  fromRight (error "failed to read float")
  <?> "f32"

f64 :: Parser Double
f64 =
  takeP Nothing 8 <&>
  runGet getFloat64le <&>
  fromRight (error "failed to read float")
  <?> "f64"

idx :: Parser Idx
idx = u32 <?> "idx"

vec :: Parser a -> Parser (Vec a)
vec p =
  u32 >>= \ elemCount ->
  V.replicate (fromIntegral elemCount) p &
  sequenceA

vecTill :: Parser a -> Parser () -> Parser (Vec a)
vecTill p pEnd = V.fromList <$> manyTill p pEnd

byte :: Byte -> Parser ()
byte b =
  () <$ char b

name :: Parser String
name =
  u32 >>=
  takeP Nothing . fromIntegral <&>
  Text.unpack . decodeUtf8
  <?> "name"

--------------------------------------------------------------------------------
-- * Lesser types
--------------------------------------------------------------------------------

valType :: Parser ValType
valType =
  (NumInt, W32) <$ byte 0x7F <|>
  (NumInt, W64) <$ byte 0x7E <|>
  (NumFlt, W32) <$ byte 0x7D <|>
  (NumFlt, W64) <$ byte 0x7C
  <?> "valtype"

resType :: Parser ResType
resType =
  Nothing <$ byte 0x40 <|>
  Just <$> valType
  <?> "restype"

funcTypeP :: Parser FuncType
funcTypeP =
  byte 0x60 *> (
    FuncType
      <$> vec valType
      <*> (vec valType <&> (V.!? 1)) -- Just a single result in spec 1.0
  )
  <?> "functype"

limits :: Parser Limits
limits =
  byte 0x00 *> (Limits <$> u32 <*> pure Nothing) <|>
  byte 0x01 *> (Limits <$> u32 <*> (Just <$> u32))
  <?> "limits"

memType :: Parser MemType
memType = limits <?> "memtype"

tableType :: Parser TableType
tableType =
  flip TableType <$> elemType <*> limits
  <?> "tabletype"
```

```haskell

elemType :: Parser ElemType
elemType =
  Undefined <$ byte 0x70 -- FuncRef type seems to contain a lot of redundancy
  <?> "elemtype"

globalTypeP :: Parser GlobalType
globalTypeP =
  flip GlobalType <$> valType <*> (False <$ byte 0x00 <|> True <$ byte 0x01)
  <?> "globalType"

--------------------------------------------------------------------------------
-- * Instructions
--------------------------------------------------------------------------------

instr :: Parser Instr
instr =
  controlInstr
  <|> paramInstr
  <|> varInstr
  <|> memInstr
  <|> constInstr
  <|> numInstr
  <?> "instr"

controlInstr :: Parser Instr
controlInstr =
  Unreachable <$ byte 0x00 <|>
  Nop <$ byte 0x01 <|>
  Return <$ byte 0x0F <|>
  byte 0x0C *> (Branch       <$> idx)               <|>
  byte 0x0D *> (BranchIf     <$> idx)               <|>
  byte 0x0E *> (BranchTable  <$> vec idx <*> idx)   <|>
  byte 0x10 *> (Call         <$> idx)               <|>
  byte 0x11 *> (CallIndirect <$> idx) <* byte 0x00 <|>
  blockInstr <|>
  loopInstr <|>
  ifElseInstr

end :: Parser ()
end = byte 0x0B <?> "end"

blockInstr :: Parser Instr
blockInstr =
  byte 0x02 *> (
  Block
    <$> resType
    <*> vecTill instr end
  )

loopInstr :: Parser Instr
loopInstr =
  byte 0x03 *> (
  Loop
    <$> resType
    <*> vecTill instr end
  )

ifElseInstr :: Parser Instr
ifElseInstr =
  byte 0x04 *> (
  If
    <$> resType
    <*> vecTill instr (lookAhead $ byte 0x05 <|> end)
    <*> ((byte 0x05 *> vecTill instr end) <|> (end *> pure V.empty))
  )

paramInstr :: Parser Instr
paramInstr =
  Drop <$ byte 0x1A <|>
  Select <$ byte 0x1B

varInstr :: Parser Instr
```

```
varInstr =
  byte 0x20 *> (LocalGet <$> idx) <|>
  byte 0x21 *> (LocalSet <$> idx) <|>
  byte 0x22 *> (LocalTee <$> idx) <|>
  byte 0x23 *> (GlobalGet <$> idx) <|>
  byte 0x24 *> (GlobalSet <$> idx)

--------------------------------------------------------------------------------
-- * Memory instructions
--------------------------------------------------------------------------------

memArg :: Parser MemArg
memArg =
  flip MemArg <$> u32 <*> u32
  <?> "memarg"

memInstr :: Parser Instr
memInstr =
  byte 0x28 *> (Load (NumInt, W32) <$> memArg) <|>
  byte 0x29 *> (Load (NumInt, W64) <$> memArg) <|>
  byte 0x2A *> (Load (NumFlt, W32) <$> memArg) <|>
  byte 0x2B *> (Load (NumFlt, W64) <$> memArg) <|>
  byte 0x2C *> (IntLoad8  W32 SN   <$> memArg) <|>
  byte 0x2D *> (IntLoad8  W32 US <$> memArg) <|>
  byte 0x2E *> (IntLoad16 W32 SN   <$> memArg) <|>
  byte 0x2F *> (IntLoad16 W32 US <$> memArg) <|>
  byte 0x30 *> (IntLoad8  W64 SN   <$> memArg) <|>
  byte 0x31 *> (IntLoad8  W64 US <$> memArg) <|>
  byte 0x32 *> (IntLoad16 W64 SN   <$> memArg) <|>
  byte 0x33 *> (IntLoad16 W64 US <$> memArg) <|>
  byte 0x34 *> (IntLoad32 SN <$> memArg) <|>
  byte 0x35 *> (IntLoad32 US <$> memArg) <|>
  byte 0x36 *> (Store (NumInt, W32) <$> memArg) <|>
  byte 0x37 *> (Store (NumInt, W64) <$> memArg) <|>
  byte 0x38 *> (Store (NumFlt, W32) <$> memArg) <|>
  byte 0x39 *> (Store (NumFlt, W64) <$> memArg) <|>
  byte 0x3A *> (IntStore8  W32 <$> memArg) <|>
  byte 0x3B *> (IntStore16 W32 <$> memArg) <|>
  byte 0x3C *> (IntStore8  W64 <$> memArg) <|>
  byte 0x3D *> (IntStore16 W64 <$> memArg) <|>
  byte 0x3E *> (IntStore32 <$> memArg) <|>
  byte 0x3F *> byte 0x3F *> pure MemorySize <|>
  byte 0x40 *> byte 0x00 *> pure MemoryGrow

--------------------------------------------------------------------------------
-- * Numerical instructions
--------------------------------------------------------------------------------

constInstr :: Parser Instr
constInstr =
  byte 0x41 *> (Constant . I32Val <$> u32) <|>
  byte 0x42 *> (Constant . I64Val <$> u64) <|>
  byte 0x43 *> (Constant . F32Val <$> f32) <|>
  byte 0x44 *> (Constant . F64Val <$> f64)

numInstr :: Parser Instr
numInstr =
  anySingle >>= \ b ->
  lookupInstr b & maybe empty pure

--------------------------------------------------------------------------------
-- * Expressions
--------------------------------------------------------------------------------

expr :: Parser Expr
expr = vecTill instr end <?> "expr"

--------------------------------------------------------------------------------
-- * Constructs
--------------------------------------------------------------------------------
importP :: Parser Import
importP =
  Import <$> name <*> name <*> importDescP
```

```
  <?> "import"

importDescP :: Parser ImportDesc
importDescP =
  byte 0x00 *> (ImportFunc <$> idx) <|>
  byte 0x01 *> (ImportTable <$> idx) <|>
  byte 0x02 *> (ImportMem <$> idx) <|>
  byte 0x03 *> (ImportGlobal <$> idx)
  <?> "importdesc"

global :: Parser Global
global =
  Global <$> globalTypeP <*> expr
  <?> "global"

export :: Parser Export
export =
  Export <$> name <*> exportDesc
  <?> "export"

exportDesc :: Parser ExportDesc
exportDesc =
  byte 0x00 *> (ExportFunc <$> idx) <|>
  byte 0x01 *> (ExportTable <$> idx) <|>
  byte 0x02 *> (ExportMem <$> idx) <|>
  byte 0x03 *> (ExportGlobal <$> idx)
  <?> "exportdesc"

elemP :: Parser Elem
elemP =
  Elem <$> idx <*> expr <*> vec idx
  <?> "elem"

bodyP :: Parser (Vec ValType, Expr)
bodyP =
  u32 *> ((,) <$> localsP <*> expr)

localsP :: Parser (Vec ValType)
localsP =
  join <$> vec (
      V.replicate <$> (fromIntegral <$> u32) <*> valType
  )

makeFunc :: Vec FuncType -> Idx -> (Vec ValType, Expr) -> Func
makeFunc funcTypes funcIdx (valTypes, funcBody) =
  let funcType = funcTypes V.! (fromIntegral funcIdx) in
  Func funcType valTypes funcBody

dataP :: Parser Data
dataP =
  Data <$> idx <*> expr <*> vec anySingle
  <?> "data"

--------------------------------------------------------------------------------
-- * Modules
--------------------------------------------------------------------------------

customSection :: Parser ()
customSection =
  byte 0x00 *>
  (u32 >>= takeP (Just "custom section") . fromIntegral) *>
  pure ()
  <?> "custom section"

section :: Byte -> a -> Parser a -> Parser a
section nID fallback p =
  (byte nID *> u32 *> p <|> pure fallback) <* skipMany customSection

vecSection :: Byte -> Parser a -> Parser (Vec a)
vecSection nID p = section nID V.empty (vec p)

moduleP :: Parser Module
moduleP =
```

```
  do
    _ <- chunk (B.pack [0x00, 0x61, 0x73, 0x6D, 0x01, 0x00, 0x00, 0x00])
    skipMany customSection
    mTypes <- vecSection 1 funcTypeP
    mImports <- vecSection 2 importP
    mFuncIdxs <- vecSection 3 idx
    mTables <- vecSection 4 tableType
    mMemory <- vecSection 5 memType
    mGlobals <- vecSection 6 global
    mExports <- vecSection 7 export
    mStart <- section 8 Nothing (optional idx)
    mElems <- vecSection 9 elemP
    mCode <- vecSection 10 bodyP
    mDatas <- vecSection 11 dataP
    eof

    return $ Module {
      types = mTypes,
      funcs = V.zipWith (makeFunc mTypes) mFuncIdxs mCode,
      tables = mTables,
      mems = mMemory,
      globals = mGlobals,
      elems = mElems,
      dat = mDatas,
      start = mStart,
      imports = mImports,
      exports = mExports
    }

parseWASM :: String -> B.ByteString -> Either String Module
parseWASM filename text =
  parse moduleP filename text & first errorBundlePretty
```

A.8 ParserInstr.hs

```
module ParserInstr where

import Data.Word (Word8)
import AST

-- Mapping of bytes to their respective non-parametric instructions

lookupInstr :: Word8 -> Maybe Instr
lookupInstr b =
  case b of
    0x45 -> Just $ IUnary W32 IEQZ
    0x46 -> Just $ ICompare W32 IEQ
    0x47 -> Just $ ICompare W32 INE
    0x48 -> Just $ ICompare W32 (ILT SN)
    0x49 -> Just $ ICompare W32 (ILT US)
    0x4A -> Just $ ICompare W32 (IGT SN)
    0x4B -> Just $ ICompare W32 (IGT US)
    0x4C -> Just $ ICompare W32 (ILE SN)
    0x4D -> Just $ ICompare W32 (ILE US)
    0x4E -> Just $ ICompare W32 (IGE SN)
    0x4F -> Just $ ICompare W32 (IGE US)

    0x50 -> Just $ IUnary W64 IEQZ
    0x51 -> Just $ ICompare W64 IEQ
    0x52 -> Just $ ICompare W64 INE
    0x53 -> Just $ ICompare W64 (ILT SN)
    0x54 -> Just $ ICompare W64 (ILT US)
    0x55 -> Just $ ICompare W64 (IGT SN)
    0x56 -> Just $ ICompare W64 (IGT US)
    0x57 -> Just $ ICompare W64 (ILE SN)
    0x58 -> Just $ ICompare W64 (ILE US)
    0x59 -> Just $ ICompare W64 (IGE SN)
    0x5A -> Just $ ICompare W64 (IGE US)

    0x5B -> Just $ FCompare W32 FEQ
    0x5C -> Just $ FCompare W32 FNE
    0x5D -> Just $ FCompare W32 FLT
    0x5E -> Just $ FCompare W32 FGT
```

```
    0x5F -> Just $ FCompare W32 FLE
    0x60 -> Just $ FCompare W32 FGE

    0x61 -> Just $ FCompare W64 FEQ
    0x62 -> Just $ FCompare W64 FNE
    0x63 -> Just $ FCompare W64 FLT
    0x64 -> Just $ FCompare W64 FGT
    0x65 -> Just $ FCompare W64 FLE
    0x66 -> Just $ FCompare W64 FGE

    0x67 -> Just $ IUnary W32 IClz
    0x68 -> Just $ IUnary W32 ICtz
    0x69 -> Just $ IUnary W32 IPopCnt

    0x6A -> Just $ IBinary W32 IAdd
    0x6B -> Just $ IBinary W32 ISub
    0x6C -> Just $ IBinary W32 IMul
    0x6D -> Just $ IBinary W32 (IDiv SN)
    0x6E -> Just $ IBinary W32 (IDiv US)
    0x6F -> Just $ IBinary W32 (IRem SN)
    0x70 -> Just $ IBinary W32 (IRem US)

    0x71 -> Just $ IBinary W32 IAnd
    0x72 -> Just $ IBinary W32 IOr
    0x73 -> Just $ IBinary W32 IXOr
    0x74 -> Just $ IBinary W32 IShiftL
    0x75 -> Just $ IBinary W32 (IShiftR SN)
    0x76 -> Just $ IBinary W32 (IShiftR US)
    0x77 -> Just $ IBinary W32 IRotL
    0x78 -> Just $ IBinary W32 IRotR

    0x79 -> Just $ IUnary W64 IClz
    0x7A -> Just $ IUnary W64 ICtz
    0x7B -> Just $ IUnary W64 IPopCnt

    0x7C -> Just $ IBinary W64 IAdd
    0x7D -> Just $ IBinary W64 ISub
    0x7E -> Just $ IBinary W64 IMul
    0x7F -> Just $ IBinary W64 (IDiv SN)
    0x80 -> Just $ IBinary W64 (IDiv US)
    0x81 -> Just $ IBinary W64 (IRem SN)
    0x82 -> Just $ IBinary W64 (IRem US)

    0x83 -> Just $ IBinary W64 IAnd
    0x84 -> Just $ IBinary W64 IOr
    0x85 -> Just $ IBinary W64 IXOr
    0x86 -> Just $ IBinary W64 IShiftL
    0x87 -> Just $ IBinary W64 (IShiftR SN)
    0x88 -> Just $ IBinary W64 (IShiftR US)
    0x89 -> Just $ IBinary W64 IRotL
    0x8A -> Just $ IBinary W64 IRotR

    0x8B -> Just $ FUnary W32 FAbs
    0x8C -> Just $ FUnary W32 FNeg
    0x8D -> Just $ FUnary W32 FCeil
    0x8E -> Just $ FUnary W32 FFloor
    0x8F -> Just $ FUnary W32 FTrunc
    0x90 -> Just $ FUnary W32 FNearest
    0x91 -> Just $ FUnary W32 FSqrt

    0x92 -> Just $ FBinary W32 FAdd
    0x93 -> Just $ FBinary W32 FSub
    0x94 -> Just $ FBinary W32 FMul
    0x95 -> Just $ FBinary W32 FDiv
    0x96 -> Just $ FBinary W32 FMin
    0x97 -> Just $ FBinary W32 FMax
    0x98 -> Just $ FBinary W32 FCopySign

    0x99 -> Just $ FUnary W64 FAbs
    0x9A -> Just $ FUnary W64 FNeg
    0x9B -> Just $ FUnary W64 FCeil
    0x9C -> Just $ FUnary W64 FFloor
    0x9D -> Just $ FUnary W64 FTrunc
```

```
    0x9E -> Just $ FUnary W64 FNearest
    0x9F -> Just $ FUnary W64 FSqrt

    0xA0 -> Just $ FBinary W64 FAdd
    0xA1 -> Just $ FBinary W64 FSub
    0xA2 -> Just $ FBinary W64 FMul
    0xA3 -> Just $ FBinary W64 FDiv
    0xA4 -> Just $ FBinary W64 FMin
    0xA5 -> Just $ FBinary W64 FMax
    0xA6 -> Just $ FBinary W64 FCopySign

    0xA7 -> Just $ WrapI64
    0xA8 -> Just $ IntTruncFlt W32 W32 SN
    0xA9 -> Just $ IntTruncFlt W32 W32 US
    0xAA -> Just $ IntTruncFlt W32 W64 SN
    0xAB -> Just $ IntTruncFlt W32 W64 US
    0xAC -> Just $ ExtendI32 SN
    0xAD -> Just $ ExtendI32 US
    0xAE -> Just $ IntTruncFlt W64 W32 SN
    0xAF -> Just $ IntTruncFlt W64 W32 US
    0xB0 -> Just $ IntTruncFlt W64 W64 SN
    0xB1 -> Just $ IntTruncFlt W64 W64 US
    0xB2 -> Just $ Flt2Int W32 W32 SN
    0xB3 -> Just $ Flt2Int W32 W32 US
    0xB4 -> Just $ Flt2Int W32 W64 SN
    0xB5 -> Just $ Flt2Int W32 W64 US
    0xB6 -> Just $ FltDemote
    0xB7 -> Just $ Flt2Int W64 W32 SN
    0xB8 -> Just $ Flt2Int W64 W32 US
    0xB9 -> Just $ Flt2Int W64 W64 SN
    0xBA -> Just $ Flt2Int W64 W64 US
    0xBB -> Just $ FltPromote
    0xBC -> Just $ Reinterpret (NumInt, W32) (NumFlt, W32)
    0xBD -> Just $ Reinterpret (NumInt, W64) (NumFlt, W64)
    0xBE -> Just $ Reinterpret (NumFlt, W32) (NumFlt, W32)
    0xBF -> Just $ Reinterpret (NumFlt, W64) (NumFlt, W64)
    _ -> Nothing
```

A.9 Assertions.hs

```
module Assertions where

import Data.SBV hiding (Predicate, label)

import Control.Arrow ((>>>))
import Data.Functor ((<&>))
import Data.Function ((&))
import Control.Applicative (liftA2)

import qualified Data.ByteString as B
import Data.SBV.Tools.Overflow
import qualified Data.Vector as V

import Machine
import SValue
import AST

parseAsserts :: [String] -> Predicate
parseAsserts = map assertLookup >>> foldl1 combinePredicates

assertLookup :: String -> Predicate
assertLookup "none" = assertTrue
assertLookup "unreachable" = assertUnreachable
assertLookup "intDivZero" = assertIntDivZero
assertLookup str = error ("unknown assertion: " ++ str)

combinePredicates :: Predicate -> Predicate -> Predicate
combinePredicates =
  liftA2 $ \ (b1, sb1) (b2, sb2) ->
  (b1 || b2, (literal b1 .&& sb1) .|| (literal b2 .&& sb2))

-- | This term is never satisfied
assertTrue :: Predicate
```

```
assertTrue = const (False, sFalse)

-- TODO: Add addition, subtraction, division, etc.
overflowTerm :: Predicate
overflowTerm =
  (,) <$> (
    frames >>> head >>> contexts >>> head >>> code >>> V.head >>> \ instr ->
    instr == IBinary W32 IMul
  ) <*> (
    frames >>> head >>> contexts >>> head >>> stack >>> \ (op2:op1:_) ->
    case (op1, op2) of
      (SW32 x, SW32 y) -> snd $ bvMulO x y
      _ -> sFalse
  )

-- | This term is satisfied if an `unreachable` instruction is encountered
assertUnreachable :: Predicate
assertUnreachable =
  frames >>> head >>> contexts >>> head >>> code >>> V.head >>> \ instr ->
  (instr == Unreachable, sTrue)

-- | This term is satisfied if a division or modulus is performed where the
-- denominator is zero
assertIntDivZero :: Predicate
assertIntDivZero =
  (,) <$> (
    frames >>> head >>> contexts >>> head >>> code >>> V.head >>> \ instr ->
    case instr of
      IBinary _ (IDiv _) -> True
      IBinary _ (IRem _) -> True
      _ -> False
  ) <*> (
    frames >>> head >>> contexts >>> head >>> stack >>> \ (v:_) ->
    case v of
      SW32 x -> x .== 0
      SW64 x -> x .== 0
      _ -> sFalse
  )
```

B Appendix - test modules

B.1 math.wat

```
;; Testing conditionals mainly through loopy mathematic calculations
(module
  ;; Is the output of the two fibonacci functions equal?
  (export "test_fibo_equiv" (func $test_fibo_equiv))
  (func $test_fibo_equiv (param $n i32)
    local.get $n
    call $fibo_rec
    local.get $n
    call $fibo_loop

    i32.eq
    (if (then unreachable))
  )

  ;; Calculates the nth fibonacci number using a loop method
  (export "fibo_loop" (func $fibo_loop))
  (func $fibo_loop (param $n i32)
                   (result i32)
                   (local $acc i32)
                   (local $acc_old i32)
    ;; Initialize both accumulators to 1
    i32.const 1
    local.tee $acc
    local.set $acc_old

    (loop $add_loop
      ;; Retrieve both accumulators, add them together. Write the new
      ;; accumulator to the old one, write the result to the new accumulator
```

```
      local.get $acc_old
      local.get $acc
      local.tee $acc_old
      i32.add
      local.set $acc

      ;; Repeat n times
      local.get $n
      i32.const 1
      i32.sub
      local.tee $n
      br_if $add_loop
    )
    local.get $acc
  )

  ;; Calculates the nth fibonacci number using a recursive method
  (export "fibo_rec" (func $fibo_rec))
  (func $fibo_rec (param $n i32) (result i32)
    local.get $n
    i32.const 1
    i32.lt_u
    (if (result i32) (then
      i32.const 1
    ) (else
      local.get $n
      i32.const 1
      i32.sub
      call $fibo_rec

      local.get $n
      i32.const 2
      i32.sub
      call $fibo_rec

      i32.add
    ))
  )

  ;; Calculates n factorial using a loop method
  (export "factorial" (func $factorial))
  (func $factorial (param $n i32) (result i32) (local $acc i32)
    ;; Initialize the accumulator to one
    i32.const 1
    local.set $acc

    (loop
      ;; Multiply n into the accumulator
      local.get $acc
      local.get $n
      i32.mul
      local.set $acc

      ;; Subtract 1 from n
      local.get $n
      i32.const 1
      i32.sub
      local.tee $n

      ;; Repeat of n is non-zero
      br_if 0
    )
    local.get $acc
  )

  ;; Integer power function for non-negative exponents
  (export "power" (func $power))
  (func $power (param $val i32) (param $power i32) (result i32)
                  (local $acc i32)
    ;; Initialize the accumulator to 1
    i32.const 1
    local.set $acc

```

```
    (loop $mul_loop
      (block $mul_block
        ;; Subtract 1 from power
        local.get $power
        i32.const 1
        i32.sub
        local.tee $power

        ;; If power is less than 0, break out of loop
        i32.const 0
        i32.lt_u
        br_if $mul_block

        ;; Multiply the value into the accumulator
        local.get $acc
        local.get $val
        i32.mul
        local.set $acc

        ;; Repeat
        br $mul_loop
      )
    )
    local.get $acc
  )

  ;; Is the output of the two triangular functions equal?
  (export "test_triangular_equiv" (func $test_triangular_equiv))
  (func $test_triangular_equiv (param $n i32)
    local.get $n
    call $triangular_expr
    local.get $n
    call $triangular_loop

    i32.eq
    (if (then unreachable))
  )

  ;; Triangular sum, using loop method
  (export "triangular_loop" (func $triangular_loop))
  (func $triangular_loop (param $n i32) (result i32) (local $acc i32)
    ;; Initialize accumulator to zero
    i32.const 0
    local.set $acc

    (loop
      ;; Add n to accumulator
      local.get $acc
      local.get $n
      i32.add
      local.set $acc

      ;; Subtract 1 from n
      local.get $n
      i32.const 1
      i32.sub
      local.tee $n

      ;; Repeat if n is non-zero
      br_if 0
    )

    local.get $acc
  )

  ;; Triangular sum, using constant expression
  (export "triangular_expr" (func $triangular_expr))
  (func $triangular_expr (param $n i32) (result i32)
    local.get $n
    local.get $n
    i32.const 1
    i32.sub
    i32.mul
```

```
    i32.const 2
    i32.div_u
  )

  ;; Is the output of the two gcd functions equal?
  (export "test_gcd_equiv" (func $test_gcd_equiv))
  (func $test_gcd_equiv (param $a i32) (param $b i32)
    local.get $a
    local.get $b
    call $gcd_rec

    local.get $a
    local.get $b
    call $gcd_loop

    i32.eq
    (if (then) (else
      unreachable
    ))
  )

  ;; Calculate the greatest common divisor using a loop
  (export "gcd_loop" (func $gcd_loop))
  (func $gcd_loop (param $a i32) (param $b i32) (result i32)
    (loop $l
      ;; If b is zero, we're done and can fallthrough to exit the loop
      local.get $b
      (if (then
        ;; Calculate a mod b
        local.get $a
        local.get $b
        i32.rem_u

        ;; Write b to a, write (a mod b) to b
        local.get $b
        local.set $a
        local.set $b
        br $l
      ))
    )
    local.get $a
  )

  ;; Calculate the greatest common divisor using recursion
  (export "gcd_rec" (func $test_gcd_equiv))
  (func $gcd_rec (param $a i32) (param $b i32) (result i32)
    local.get $b
    (if (result i32) (then
      local.get $b
      local.get $a
      local.get $b
      i32.rem_u
      call $gcd_rec
    ) (else
      local.get $a
    ))
  )

  ;; Collatz conjecture
  (export "collatz_loop" (func $collatz_loop))
  (func $collatz_loop (param $n i32)
    (loop $l
      (block $b
        ;; Quit if the number equals 1
        local.get $n
        i32.const 1
        i32.eq
        br_if $b

        ;; Test whether n is even
        local.get $n
        i32.const 2
        i32.rem_u
```

```
        (if (result i32) (then
          ;; 3n + 1
          local.get $n
          i32.const 3
          i32.mul
          i32.const 1
          i32.add
        ) (else
          ;; n / 2
          local.get $n
          i32.const 2
          i32.div_u
        ))
        local.set $n
        br $l
      )
    )
  )
)
```

B.2 examples.wat

```
;; Various tests. The postfix indicates whether the function can reach an
;; undesirable state - functions labeled with "good" cannot

(module
  ;; Contains an unreachable instruction which is indeed unreachable
  (export "unreach_good" (func $unreach_good))
  (func $unreach_good (param $a i32) (param $b i32) (param $c i32)
                         (param $d i32)
                         (local $r1 i32) (local $r2 i32) (local $r3 i32)
                         (local $r4 i32)

    (local.set $r1 (i32.gt_u (local.get $a) (local.get $b)))
    (local.set $r2 (i32.gt_u (local.get $b) (local.get $c)))
    (local.set $r3 (i32.gt_u (local.get $c) (local.get $d)))
    (local.set $r4 (i32.gt_u (local.get $d) (local.get $a)))

    (if (local.get $r1)
    (if (local.get $r2)
    (if (local.get $r3)
    (if (local.get $r4) (unreachable)))))
  )

  ;; Contains an unreachable instruction which can be reached - should fail
  (export "unreach_bad" (func $unreach_bad))
  (func $unreach_bad (param $a i32) (param $b i32) (param $c i32)
                        (param $d i32)
                        (local $r1 i32) (local $r2 i32) (local $r3 i32)
                        (local $r4 i32)

    (local.set $r1 (i32.gt_u (local.get $a) (local.get $b)))
    (local.set $r2 (i32.gt_u (local.get $b) (local.get $c)))
    (local.set $r3 (i32.gt_u (local.get $c) (local.get $d)))
    (local.set $r4 (i32.gt_u (local.get $d) (local.get $a)))

    (if (local.get $r1)
    (if (local.get $r2)
    (if (local.get $r3)
    (if (local.get $r4) (nop) (unreachable)))))
  )

  ;; Direct division by zero test
  (export "divzero_bad" (func $divzero_bad))
  (func $divzero_bad (param $n i32) (result i32)
     local.get $n
     i32.const 0
     i32.div_u
  )

  ;; A layer of indirection. Does check that y is non-zero, but not that the
  ;; remainder from dividing x by is non-zero.
  (export "modulo_div_bad" (func $modulo_div_bad))
```

```
  (func $modulo_div_bad (param $x i32) (param $y i32) (param $z i32)
                        (result i32)
    local.get $y
    (if (result i32) (then
      local.get $z
      local.get $x
      local.get $y
      i32.rem_u
      i32.div_u
    ) (else
      i32.const -1
    ))
  )

  ;; A layer of indirection, but properly checked.
  (export "modulo_div_good" (func $modulo_div_good))
  (func $modulo_div_good (param $x i32) (param $y i32) (param $z i32)
                         (result i32)
    local.get $y
    (if (result i32) (then
      local.get $z
      local.get $x
      local.get $y
      i32.rem_u
      local.set $y
      local.get $y
      (if (result i32) (then
        local.get $y
      ) (else
        i32.const -1
      ))
      i32.div_u
    ) (else
      i32.const -1
    ))
  )

  ;; Another layer of indirection, this time demonstrating a function call.
  (export "modulo_func_bad" (func $modulo_func_bad))
  (func $modulo_func_bad (param $w i32) (param $x i32) (param $y i32)
                         (param $z i32)
    local.get $w

    local.get $x
    local.get $y
    local.get $z
    call $modulo_div_good

    i32.div_u
    drop
  )

  ;; Demonstrate that assertions can be broken inside function calls
  (export "inside_func_bad" (func $inside_func_bad))
  (func $inside_func_bad (param $x i32) (param $y i32) (param $z i32)
    local.get $x
    local.get $y
    local.get $z
    call $modulo_div_bad
    drop
  )

  ;; A function which will encounter an unreachable after running a loop for
  ;; some time
  (export "loopy_func_bad" (func $loopy_func_bad))
  (func $loopy_func_bad (param $x i32) (param $y i32)
    ;; Limit the initial value: x < 100
    local.get $x
    i32.const 100
    i32.lt_u

    ;; Limit y: y < 30
    local.get $y
```

```
    i32.const 30
    i32.lt_u

    ;; Combined constraints
    i32.and
    (if (then
      (loop $l
        ;; Add y to x
        local.get $x
        local.get $y
        i32.add
        local.set $x

        ;; If we surpass a constant, fail
        local.get $x
        i32.const 10000
        i32.gt_u
        (if (then unreachable))

        ;; Keep looping until x is larger than our constant
        local.get $x
        i32.const 10000
        i32.lt_u
        br_if $l
      )
    ))
  )
)
```