LeanWasm: An Intrinsically-Typed Interpreter for WebAssembly

MSc Thesis

Alex Ionescu
a.a.ionescu@uu.nl
Student number 0792918

July 10th, 2024

1st supervisor: Dr. Marco Vassena
2nd supervisor: Dr. Wouter Swierstra

Department of Computer Science
Faculty of Science

Universiteit Utrecht

LeanWasm: An Intrinsically-Typed Interpreter for WebAssembly

Alex Ionescu
Utrecht University
a.a.ionescu@uu.nl

Abstract

WebAssembly is a new low-level programming language and portable bytecode format designed with the goal of increasing interoperability and security across the software ecosystem.

In order to ensure correctness and adherence to the official specification, some WebAssembly implementations use formally verified interpreters as testing oracles. This thesis explores a novel approach to defining verified interpreters, by using an intrinsically-typed representation of the WebAssembly abstract syntax.

The main contributions of this thesis include an intrinsically-typed WebAssembly interpreter that closely follows the official reference interpreter, and an optimized interpreter based on an improved representation of control-flow. The interpreters are implemented in the Lean 4 theorem prover, leveraging its support for dependent typing and functional-but-in-place programming.

Contents

Appendix

1. Introduction

WebAssembly (or Wasm for short) [1] is a new low-level programming language and portable bytecode format designed with the goal of increasing interoperability and security across the software ecosystem. While initially intended for the Web, its usage has since expanded into a wide range of domains, including IoT, containerization, and even blockchains.

WebAssembly is primarily intended as a compilation target for higher-level programming languages (such as C++, Rust, and Haskell), serving as a common binary format that can be executed on multiple platforms. Its official specification [2] is published by the World Wide Web Consortium, together with a reference interpreter, and there are a large number of independent third-party implementations, ranging from Web browser engines such as V8 and SpiderMonkey to standalone execution engines such as Wasmtime [3], Wasmer [4], and Wizard [5].

An important aspect when designing WebAssembly implementations is ensuring and verifying their correctness. Concretely, this means verifying that the implementation (be it an interpreter, just-in-time compiler or ahead-of-time compiler) follows the official WebAssembly specification. While most implementations rely on Wasm's official test suite [6] for this purpose, implementations that want stronger guarantees make use of formal verification techniques to ensure adherence to the specification. An example of this is Wasmtime [3], a widely-used WebAssembly engine that uses the formally-verified WasmRef [7] interpreter as a fuzzing oracle.

However, an approach to verification that has not been attempted by prior WebAssembly implementations is that of an *intrinsically-typed* interpreter [8], that is, an interpreter that uses the host language's dependent typing facilities to encode the type system of WebAssembly in its definition of the abstract syntax. This approach makes it impossible to construct the syntax trees of ill-typed Wasm programs, thus alleviating the need for separate type-safety proofs, and allows for the definition of an interpreter that is correct-by-construction.

In this thesis, we implement two intrinsically-typed WebAssembly interpreters: One that closely follows the official reference interpreter, and then an optimized version of it, based on an improved representation of control-flow. We provide both *relational* and *functional* implementations for each interpreter, and (partial) soundness proofs attesting their equivalence. We also discuss a soundness proof for attesting the correctness of the control-flow optimizations.

The interpreters and proofs are implemented in the Lean 4 theorem prover [9], leveraging its support for dependent typing and functional-but-in-place programming.

The full source code for the LeanWasm project is available on GitHub [10], and the organization of the source code is described in Appendix A.

2. Background

2.1. WebAssembly

WebAssembly has some distinct properties that are not present in other similar binary formats and low-level languages:

- Type-safety: WebAssembly is a stack-based language, and all instructions are assigned a type that encodes their usage of the stack. Before execution, Wasm programs go through a *validation phase* that rejects ill-typed programs.
- Memory-safety: Wasm programs cannot access arbitrary memory addresses, they can only access buffers that are explicitly shared by their execution host (operating system, browser etc.). Moreover, all memory operations in Wasm employ dynamic in-bounds checks, preventing corruption of the host's memory.
- Structured control-flow: WebAssembly does not include an arbitrary jump or `goto` instruction. Instead, programs in WebAssembly are structured into functions with type-checked signatures, and lexically-scoped *block* and *loop* instructions are used for local control-flow within each function.

This section describes the parts of the WebAssembly specification that are most relevant to this thesis, based on the official specification [2] and on the work of Haas et al. [1].

2.1.1. Type system

The types that WebAssembly values and instructions are allowed to take are defined by its *type system*. This section gives an overview of Wasm's type system, highlighting the aspects that are most relevant to this thesis.

Types

Value types describe the types of individual values that WebAssembly programs compute. *Result types* describe the results of executing instructions and functions, and are represented as sequences of value types. *Function types* classify the signatures of Wasm instructions and functions, consisting of a pair of result types, representing the inputs and outputs of the function or instruction. The grammar for value and function types is given in Figure 1 below.

$$
\begin{array}{rcl}
\mathit{valtype} & ::= & \mathsf{i32} \\
& | & \mathsf{i64} \\
& | & \mathsf{f32} \\
& | & \mathsf{f64} \\
\mathit{resulttype} & ::= & \mathit{valtype}^* \\
\mathit{functype} & ::= & \mathit{resulttype} \to \mathit{resulttype}
\end{array}
$$

Figure 1: Grammar of WebAssembly types.

Instructions

Instructions are typed with *function types*, which encode each instruction's *inputs* (what types it expects to find on the stack before executing) as well as its *outputs* (what types it leaves on the stack after executing). Typing judgements for instructions include a context C that stores type information about various Wasm components such as local and global variables, labels, and functions. Figure 2 provides a simplified definition of contexts, and Figure 3 shows the typing rules for a few illustrative Wasm instructions.

$$
C ::= \{ \mathsf{locals}\ \mathit{valtype}^*, \mathsf{labels}\ \mathit{resulttype}^*, \mathsf{funcs}\ \mathit{functype}^*, \dots \}
$$

Figure 2: Grammar for typing contexts.

$$\frac{}{C \vdash t.\mathbf{add} : t\ t \to t}\ \textit{T-add}$$

$$\frac{}{C \vdash t.\mathbf{const}\ c : \varepsilon \to t}\ \textit{T-const}$$

$$\frac{}{C \vdash \mathbf{drop} : t \to \varepsilon}\ \textit{T-drop}$$

$$\frac{C.\mathsf{locals}(i) = t}{C \vdash \mathbf{local.get}\ i : \varepsilon \to t}\ \textit{T-get}$$

$$\frac{C.\mathsf{locals}(i) = t}{C \vdash \mathbf{local.set}\ i : t \to \varepsilon}\ \textit{T-set}$$

Figure 3: Typing rules for instructions (illustrative).

In the typing rules above, t is a variable that stands in for a value type, and can be instantiated to form instructions such as `i32.add` or `f64.add`.

Instruction sequences

The typing of instruction sequences is defined recursively, as shown in Figure 4.

$$\frac{}{C \vdash \varepsilon : \varepsilon \to \varepsilon}\ \textit{T-empty}$$

$$\frac{C \vdash e_1 : t_1^* \to t_2^* \qquad C \vdash e_2^* : t_2^* \to t_3^*}{C \vdash e_1 e_2^* : t_1^* \to t_3^*}\ \textit{T-seq}$$

Figure 4: Typing rules for instruction sequences.

However, this rule is not sufficient to properly type instruction sequences. To see why, consider the instruction sequence `i32.add i32.add`. It's reasonable to think that its type should be i32 i32 i32 → i32: the first `i32.add` adds the first 2 values on the stack, summing them into a single value, and the second `i32.add` sums the remaining 2 values into one. But the typing rule for instruction sequences will reject this, as it expects the output of the first instruction to *exactly* match the input of the second, which does not hold in this case.

To allow the construction of such instruction sequences, the WebAssembly type system includes a *weakening* rule, which allows instructions to be given "larger" types, if both the input and output stack are extended with the same type sequence t^*, as shown in Figure 5.

$$\frac{C \vdash e^* : t_1^* \to t_2^*}{C \vdash e^* : t^*\ t_1^* \to t^*\ t_2^*}\ \textit{T-weaken}$$

Figure 5: The weakening rule.

Control instructions

As previously discussed, one of WebAssembly's distinguishing features is its support for *structured control-flow*. Concretely, there are two structured control instructions, `block` and `loop`, which encapsulate a nested sequence of instructions. Both of these instructions implicitly introduce a *label* in the typing contexts of the nested instructions. Labels are used as targets for the branching instructions `br` and `br_if`. The typing rules for control instructions are given below, in Figure 6.

$$\frac{tf = t_1^n \to t_2^m \qquad C, \mathsf{label}(t_2^m) \vdash e^* : tf}{C \vdash \textbf{block}\ tf\ e^*\ \textbf{end} : tf}\ \textit{T-block}$$

$$\frac{tf = t_1^n \to t_2^m \qquad C, \mathsf{label}(t_1^n) \vdash e^* : tf}{C \vdash \textbf{loop}\ tf\ e^*\ \textbf{end} : tf}\ \textit{T-loop}$$

$$\frac{C.\mathsf{labels}(i) = t^*}{C \vdash \textbf{br}\ i : t_1^*\ t^* \to t_2^*}\ \textit{T-br}$$

$$\frac{C.\mathsf{labels}(i) = t^*}{C \vdash \textbf{br\_if}\ i : t^*\ \mathsf{i32} \to t^*}\ \textit{T-br\_if}$$

Figure 6: Typing rules for control-flow instructions.

The labels introduced by the `block` and `loop` instructions have the following semantics:

- Branching to a `block`'s label jumps to the *end* of that `block` instruction.
- Branching to a `loop`'s label jumps back to the *beginning* of that `loop` instruction. This jump isn't performed automatically though: a `loop` with no `br` instructions jumping back to it will only execute once.

The *result type* associated to each label represents the types that must be on the stack when a jump to that label is performed. The label introduced by `block` is typed with the block's *output*, since a jump to that label will continue execution immediately following its end, so the stack must match its output. Similarly, the label introduced by `loop` is typed with the loop's *input* stack type.

The typing rule for `br` ensures both that the label i is in scope, and that the values at the top of the stack match the label's type. An important detail is the additional type list t_1^*, which is meant to capture the tail (or base) of the stack. This typing rule encodes the fact that when a branch is performed, all values on the stack beyond those that are included in the label's type are implicitly discarded. Once a branch is taken, none of the instructions following it are executed, so its output type is a completely unconstrained variable t_2^*.

The `br_if` instruction differs from `br` in that it branches *conditionally*, only if the topmost value on the stack is non-zero. Because it's possible that a `br_if` does nothing, its type is more restrictive than `br`'s: It doesn't arbitrarily discard values, and its output type exactly matches its input type.

Labels in Wasm are identified by de Bruijn indices [11], thus the label introduced by the innermost `block` or `loop` has index 0, and labels introduced by successive outer blocks are identified by ascending indices.

2.1.2. Semantics

The WebAssembly specification also includes a small-step operational semantics describing the execution of Wasm programs. This section describes the structure of the Wasm abstract machine, and shows the reduction rules for a few illustrative instructions.

Runtime structure

Store

The *store* represents the global state that WebAssembly programs operate on. It contains the *instances* (runtime representations) of functions, global variables, memories, and other elements. The grammar for stores is presented below, in Figure 7.

$$store ::= \{\, \mathsf{funcs}\ funcinst^*, \mathsf{globals}\ globalinst^*, \mathsf{mems}\ meminst^*, \ldots \,\}$$

Figure 7: Definition of the Wasm store.

Call frames

Call frames, or *activation frames*, store information about a particular function call, such as the values of its local variables (including arguments), its return arity n, and a reference to the function's defining module instance. Their definition is provided in Figure 8 below.

$$\begin{array}{rcl} frame & ::= & \mathsf{frame}_n\{framestate\} \\ framestate & ::= & \{\, \mathsf{locals}\ val^*, \mathsf{module}\ moduleinst \,\} \end{array}$$

Figure 8: Definition of call frames.

Modules

A *module* is the unit of encapsulation and distribution for WebAssembly programs. A *module instance* is the runtime representation of a module, and contains runtime representations of all entities that are imported, exported, and defined by the module.

Memory

Each WebAssembly module has access to a *linear memory*, a contiguous buffer of memory that is allocated and initialized by the host platform.

The specification simply defines a *memory instance* (the runtime representation of the memory) to be a list of bytes, as shown in Figure 9 below.

$$meminst ::= b^*$$

Figure 9: Definition of linear memory instances.

All accesses to the linear memory are bounds-checked, and an out-of-bound access causes execution to *trap* (enter an error state). The size of the memory can also be queried and resized by the Wasm program during its execution.

Values

WebAssembly programs create and manipulate values of the *value types* defined in Figure 1. Values are represented with an abstract syntax that mimics the notation of the `const` instruction, as shown in Figure 10.

$$\begin{array}{rcl} val & ::= & \mathsf{i32.const}\ i32 \\ & | & \mathsf{i64.const}\ i64 \\ & | & \mathsf{f32.const}\ f32 \\ & | & \mathsf{f64.const}\ f64 \end{array}$$

Figure 10: Grammar of WebAssembly values.

Stacks

Even though the operand stack is an important component of WebAssembly execution, it is not represented as an explicit structure in the Wasm abstract machine. Instead, the stack simply consists of a sequence of values (`const` instructions) embedded in the instruction stream.

Execution

The execution of WebAssembly instructions is modeled as a small-step operational semantics. Every *reduction rule* specifies one step of execution, and has the following general form:

$$configuration \hookrightarrow configuration$$

A *configuration* is a syntactic description of an abstract machine state. It is typically a 3-tuple $(S; F; instr^*)$ consisting of the current store S, the call frame F of the current function, and a sequence

of unexecuted instructions. The store S and call frame F are omitted from reduction rules that do not access or modify them.

Basic instructions

Figure 11 shows the reduction rules for a few illustrative WebAssembly instructions.

$$val\ \textbf{drop} \hookrightarrow \varepsilon$$

$$(\mathsf{i32}.\textbf{const}\ c_1)(\mathsf{i32}.\textbf{const}\ c_2)\ \mathsf{i32}.\textbf{add} \hookrightarrow (\mathsf{i32}.\textbf{const}\ (c_1 + c_2))$$

$$F; (\textbf{local}.\textbf{get}\ x) \hookrightarrow F; val\ (\mathsf{if}\ F.\mathsf{locals}(x) = val)$$

$$F; val\ (\textbf{local}.\textbf{set}\ x) \hookrightarrow F'; \varepsilon\ (\mathsf{if}\ F' = F\ \mathsf{with}\ \mathsf{locals}(x) = val)$$

Figure 11: Reduction rules for simple Wasm instructions.

In the above reduction rules it can be seen that the Wasm stack is simply embedded in the instruction stream as a sequence of `const` instructions. The `drop` instruction simply discards the value preceding it, while `i32.add` sums up its 2 preceding `i32` values, producing a single value. The `local.get` and `local.set` instructions refer to the current function's list of local variables, which is accessed through the call frame F.

Administrative instructions

Control instructions such as `block` and `loop` aren't directly amenable to small-step evaluation. As such, in order to express their reduction, the specification extends the syntax of instructions with a series of *administrative instructions*. These additional instructions are only created as a program is being executed, and cannot appear in Wasm source code.

The relevant administrative instruction for reducing blocks and loops is `label`, described by the following reduction rules:

$$val^m\ \textbf{block}\ bt\ instr^*\ \textbf{end} \hookrightarrow \textbf{label}_n\{\varepsilon\}\ val^m\ instr^*\ \textbf{end}\quad (\mathsf{if}\ bt = t_1^m \to t_2^n)$$

$$val^m\ \textbf{loop}\ bt\ instr^*\ \textbf{end} \hookrightarrow \textbf{label}_m\{\textbf{loop}\ bt\ instr^*\ \textbf{end}\}\ val^m\ instr^*\ \textbf{end}\quad (\mathsf{if}\ bt = t_1^m \to t_2^n)$$

Figure 12: Reduction rules for blocks and loops.

Blocks and loops immediately reduce to `labels`. The instruction sequence surrounded by braces {} represents the label's *continuation*, and it is executed a branch targeting that label is taken. When reducing a `block`, the continuation of the resulting `label` is empty, as branching to a block label simply jumps past the block. On the other hand, in the case of `loops`, the continuation stores a copy of the loop itself, so branching to a loop label will re-execute its body.

The values described as val^m represent the input values of the `block` or `loop`, and are captured inside the generated `label` instruction.

Label contexts

In order to describe the reduction of branches, the specification defines the following syntax of *label contexts*:

$$\begin{array}{rcl} B^0 & ::= & val^*\ [\_]\ instr^* \\ B^{k+1} & ::= & val^*\ \textbf{label}_n\{instr^*\}\ \textbf{end}\ instr^* \end{array}$$

Figure 13: The syntax of label contexts.

A label context B^k represents a nesting of k labels surrounding a *hole* $[\_]$ that marks the location where the next step of computation is taking place.

This definition of label contexts allows for the very reduction of branches to be defined very easily:

$$\textbf{label}_n\{instr^*\}\ B^l[val^n(\textbf{br}\ l)]\ \textbf{end} \hookrightarrow val^n\ instr^*$$

Figure 14: Reduction rule for `br`, making use of label contexts.

The reduction rule shown above uses label contexts to very concisely match a branch to the l-th label (where l is a de Bruijn index) with a context of l nested labels, and reduces the entire set of nested labels to just the continuation of the l-th label. The branch's output values, val^n, are preserved by this reduction rule, while all the other values inside the nested label context are discarded.

2.1.3. Reference interpreter

Alongside the Wasm specification, the W3C also publishes a *reference interpreter* [12] for WebAssembly, written in OCaml [13].

The goal of the reference interpreter is to approximate an "executable specification" [1, section 7], thus it is defined in a way that closely matches the semantics presented above. However, its modeling of Wasm execution differs from the specification in two ways:

- The operand stack is stored as a separate data structure from the instruction stream.
- The reduction of branching instructions is implemented in terms of a new administrative instruction, called `breaking`, rather than in terms of nested label contexts.

These differences arise from the fact that the semantics, as defined in the specification, is not directly executable, due to the interleaving of values and instructions, and the arbitrary matching and discarding of nested label contexts.

LeanWasm's unoptimized interpreter closely follows the reference interpreter, and thus differs from the specification in the same ways.

2.2. Dependently-typed programming

A large number of programming languages use static types to verify the (partial) correctness of programs before they are compiled or executed. Most type systems can prevent simple errors like adding a number to a string (e.g. the expression `2 + "a"`), while high-level languages such as Haskell employ advanced type systems that allow programmers to specify more fine-grained invariants in their programs' types.

Dependently-typed languages like Agda [14], Coq [15], and Lean [9] take this one step further, allowing a the types within a program to depend on *values*. This allows programmers to specify fine-grained invariants in their programs' types, thus eliminating large classes of errors. A common example of such an invariant is encoding the length of a vector in its type, and using this information to restrict the domains of partial functions such as `head` and `tail`, as illustrated by the following Lean example:

```
inductive Vector (a : Type) : Nat -> Type where
  | nil : Vector a 0
  | cons : a -> Vector a n -> Vector a (n+1)

def head (v : Vector a (n+1)) : a :=
  match v with
  | cons h _ => h

def tail (v : Vector a (n+1)) : Vector a n :=
  match v with
  | cons _ t => t
```

2.2.1. Theorem proving

Using dependently-typed programming languages, we can define types that represent mathematical or logical propositions such that values of those types correspond to *proofs* of the respective propositions. For example, assuming a definition of natural numbers based on the Peano axioms (the `Nat` type), we can define the type `Lt m n` for any natural numbers `m` and `n` (using the notational shorthand `0` for `zero` and `n+1` for `succ n`):

```
inductive Nat : Type where
  | zero : Nat
  | succ : Nat -> Nat

inductive Lt : Nat -> Nat -> Type where
  | lt_zero : Lt 0 (n+1)
  | lt_succ : Lt m n -> Lt (m+1) (n+1)
```

The type `Lt m n` encodes the proposition $m < n$, and values of this type (proofs of the proposition) can be constructed in one of two ways:

- `lt_zero` asserts that `0` is less than the successor of any natural number (`n+1`)
- `lt_succ` asserts that for any natural numbers `m` and `n`, given a proof that $m < n$, we can conclude that $m + 1 < n + 1$.

Using the above definition, we can provide, for example, a proof that $2 < 3$, by constructing a value of type `Lt 2 3`, such as `lt_succ (lt_succ lt_zero)`. On the other hand, it is impossible to construct values of the type `Lt 4 3`, which would correspond to a proof of the (invalid) proposition $4 < 3$. In other words, the type `Lt 4 3` is *uninhabited*.

This association between logical propositions and types is called the Curry-Howard correspondence, and it covers equivalences between logical introduction forms and type constructors, as summarized in Table 1:

Logic	Programming
Universal quantification ($\forall$)	Dependent function type ($\rightarrow$)
Existential quantification ($\exists$)	Dependent sum type (Σ)
Implication ($\Rightarrow$)	Function type ($\rightarrow$)
Conjunction ($\wedge$)	Product type ($\times$)
Disjunction ($\vee$)	Sum type ($\uplus$)
True formula ($\top$)	Unit type ((), $\top$)
False formula ($\bot$)	Empty type (`Void`, $\bot$)

Table 1: Correspondence between logical formulas and types.

Programming languages that support the definition of propositions and proofs are generally referred to as *proof assistants*.

2.2.2. Lean 4

Since the aim of this thesis is to implement an intrinsically-typed interpreter, which requires a powerful type system (as described further in Section 2.3.2), we have chosen Lean 4 as the implementation language, due to its support for dependent types.

2.3. Interpreters for programming languages

Interpretation is a common technique for implementing programming languages. This section describes the design space of interpreters, focusing on the distinction between untyped and well-typed interpreters.

For this comparison, we will consider different interpreters for a simple expression-based language consisting of numeric and boolean constants, addition, less-than comparison, and choice (if-then-else expressions). The grammar of the language's types and expressions is shown in Figure 15.

Type	t	::= Nat	*natural numbers*
		| Bool	*booleans*
Expr	e	::= n	*numeric constant*
		| b	*boolean constant*
		| $e + e$	*addition*
		| $e < e$	*comparison*
		| if e then e else e	*choice*

Figure 15: Grammar of the simple arithmetic language

2.3.1. Untyped interpreters

This section describes a simple, untyped interpreter for the language described above.

Abstract syntax tree

When designing interpreters, operating directly on the interpreted program's source text, or *concrete syntax*, is inefficient, as the concrete syntax includes information that is only relevant for parsing. As such, it is common practice to convert the source text into a structure that's easier to process, called the *abstract syntax tree* (or AST for short).

To illustrate this, consider the following Lean 4 definition of the type of abstract syntax trees for the language described in Figure 15:

```
inductive Expr where
  num : Nat -> Expr
  bool : Bool -> Expr
  add : Expr -> Expr -> Expr
  lt : Expr -> Expr -> Expr
  ifThenElse : Expr -> Expr -> Expr -> Expr
```

Using the above definition, the program $2 + 3$ can be represented by the abstract syntax tree `add (num 2) (num 3)`. ASTs such as the one above are easier and more efficient for interpreters to process and traverse, as they only encode the abstract syntactic structure of programs, while ignoring irrelevant information such as whitespace.

Evaluation function

The core of an interpreter is its *evaluation function*. This function is responsible for traversing the abstract syntax tree of the program and executing the computations it represents.

The output of the evaluation function is a *value*, an element of the domain of abstract values that expressions in the interpreted language reduce to. In the case of the simple arithmetic language described so far, there are 2 different kinds of values to consider: natural numbers, and booleans. These are encoded as the following *tagged union*, or inductive type, in Lean:

```
inductive Value where
  | vNat : Nat -> Value
  | vBool : Bool -> Value
```

Using this definition, we can define the evaluation function as a (partial) mapping from abstract syntax trees to values:

```
def eval (e : Expr) : Option Value :=
  match e with
  | num n => some (vNum n)
  | bool b => some (vBool b)

  | add a b => do
      let a' <- eval a
      let b' <- eval b
      match a', b' with
      | vNum va, vNum vb => just (vNum (va + vb))
      | _, _ => none

  | lt a b => do
      let a' <- eval a
      let b' <- eval b
      match a', b' with
      | vNum va, vNum vb => just (vBool (va < vb))
      | _, _ => none

  | ifThenElse c t f => do
      let c' <- eval c
      match c' with
      | vBool true => eval t
      | vBool false => eval f
      | _ => none
```

The evaluation function `eval` takes an expression (a value of `Expr` type) as input, and produces a `Value` as output. To account for the possibility of receiving an invalid program as input (such as the expression `add (num 2) (bool true)` that tries to add a number to a boolean), the output must be wrapped in the `Option` type.

The `eval` function performs a pattern-match on its input expression `e`, thus defining an evaluation rule for each constructor of the `Expr` type. In the simple cases, `num` and `bool`, which denote constant expressions, the evaluator simply produces a `Value` using the appropriate constructor.

The `add` case, however, is more involved. First, it needs to evaluate the left and right sub-expressions, and short-circuit evaluation of the whole expression if any of the operands fail to evaluate. To make the code more readable, we are exploiting the fact that the `Option` type is a `Monad`, and making use of Lean's `do` notation. Once both sub-expressions have been evaluated, `eval` pattern-matches on the resulting values. If both are numbers (i.e. constructed with the `vNum` constructor), then evaluation succeeds with a new value representing the sum of the numbers. However, if either of the values is *not* a number, evaluation fails by returning `none`.

The case for `if`-expressions is similar to `add`: It first evaluates the condition expression, and if its evaluation succeeds and produces a boolean value, it conditionally evaluates one of the sub-expressions.

Downsides of the untyped representation

- Error-handling: As seen above, the definition of the evaluation function is complicated by the need to perform error-handling. This is, in turn, required because the abstract syntax only encodes the *syntactic* structure of expressions, but allows the construction of *semantically* invalid programs.

- Code maintenance: Most interpreters include a validation or *type-checking* phase before evaluation in order to reject incorrect programs before they are executed. However, this doesn't completely eliminate the issue, as invalid programs may still end up being passed to `eval`, either due to bugs in the type-checker, or due to the type-checker and AST definition going out of sync as the interpreter's codebase evolves.
- Value tagging: Another issue imposed by the lack of semantic information in the abstract syntax tree is that of *value tagging*. Because the type of the expression to evaluate is not known ahead of time, the evaluation function has to produce a generic `Value` as output, even though this doesn't accurately represent the expression's domain: an `add` or `num` expression should never produce a `vBool` value.

2.3.2. Intrinsically-typed interpreters

In order to alleviate the issues described in the previous section, interpreters can be designed in an *intrinsically-typed* way [16]. This consists of encoding semantic information about programs (usually their *types*) as part of the abstract syntax tree, thus completely eliminating the possibility of constructing invalid syntax trees.

Abstract syntax tree

In order to augment the interpreter's abstract syntax tree with semantic information, we *index* the type of expressions by their resulting types:

```
inductive Expr : Type -> Type where
  num : Nat -> Expr Nat
  bool : Bool -> Expr Bool
  add : Expr Nat -> Expr Nat -> Expr Nat
  lt : Expr Nat -> Expr Nat -> Expr Bool
  ifThenElse : Expr Bool -> Expr a -> Expr a -> Expr a
```

In this updated definition of the syntax tree, each constructor refines the index of its output type to match the semantics of the language that is being described: `num` and `add` construct expressions that reduce to natural numbers, while `bool` constructs expressions that reduce to boolean values. Thus, it is impossible to construct the syntax tree for an expression such as `2 + true`: it will lead to a Lean compilation error.

Evaluation function

Using the type-indexed abstract syntax tree, the `eval` function can be rewritten to a much more concise form:

```
def eval (e : Expr t) : t :=
  match e with
  | num n => n
  | bool b => b

  | add a b => eval a + eval b
  | lt a b => eval a < eval b

  | ifThenElse c t f =>
      match eval c with
      | true => eval t
      | false => eval f
```

This definition of the evaluation function is much simpler than the previous implementation. Firstly, there's no need to tag the resulting values: The extra information encoded in the AST guarantees that a program of type `Expr t` evaluates to a value of type `t`. Secondly, the type indexes also eliminate

the need for `Option`-wrapping, and allow us to define `eval` in a direct functional style, rather than an imperative monadic stye.

Restricting the type domain

In the previous definition, the `Expr` type is indexed by `Type`, the type of all Lean types. This gives us maximal flexibility: We could define constructors that make use of the more complex types available in Lean, such as lists, products, or even dependent types. However, often times the type domain of the object language (the language that is being interpreted) is much more restricted than the entire universe of Lean types, and so it is preferred to explicitly restrict the types that are available to expressions. This is where Lean's dependent type system shines, allowing us to express this by using a technique known as "universes à la Tarski" [17], which consists of defining an inductive type that contains only the types that are part of the object language:

```
inductive Ty : Type where
  tNat : Ty
  tBool : Ty
```

The `Ty` inductive type (named as such to avoid confusion with Lean's `Type`) only allows the construction of two types: natural numbers (through the `tNat` constructor), and booleans (through `tBool`). Using it, we can refine our definition of `Expr`, replacing the occurrences of `Type` with `Ty`, and the concrete Lean types `Nat` and `Bool` with the `Ty` constructors `tNat` and `tBool`:

```
inductive Expr : Ty -> Type where
  num : Nat -> Expr tNat
  bool : Bool -> Expr tBool
  add : Expr tNat -> Expr tNat -> Expr tNat
  lt : Expr tNat -> Expr tNat -> Expr tBool
  ifThenElse : Expr tBool -> Expr a -> Expr a -> Expr a
```

However, trying to apply the same transformation to the evaluation function results in a compilation error:

```
def eval (e : Expr t) : t := -- Error: type expected, got (t : Ty)
  ...
```

The issue is that the function's return type, `t`, is now no longer a Lean `Type`, but a piece of *data*, built using one of `Ty`'s constructors.

To solve this, we need to introduce a translation function that maps the types from our restricted domain into their corresponding Lean types:

```
def Ty.type (t : Ty) : Type :=
  match t with
  | tNat => Nat
  | tBool => Bool
```

The `Ty.type` notation defines the `type` function as part of the `Ty` namespace, which allows it to be used with "field notation", providing a convenient shorthand: The Lean expression `tNat.type` is equivalent to `Ty.type tNat`. With this definition, it is now possible to give a type signature for `eval` that Lean accepts:

```
def eval (e : Expr t) : t.type :=
  ...
```

Furthermore, we can use Lean's support for type-class based coercions [18] to define an *implicit* conversion from a value of type `Ty` to its `Type` equivalent, by creating an instance of the `CoeSort` class for `Ty`:

```
instance : CoeSort Ty where
  coe := Ty.type
```

With this instance in scope, the original type of the `eval` function is accepted by Lean, allowing us to restrict the types it accepts without modifying its signature:

```
def eval (e : Expr t) : t := -- t is implicitly coerced to t.type
  ...
```

The body of the function remains otherwise unchanged from its previous definition: We have only *restricted* the types it can operate on.

3. An intrinsically-typed interpreter for WebAssembly

The main contribution of this thesis is the construction of an intrinsically-typed interpreter for WebAssembly, using Lean 4 as the implementation language.

This section describes the design and implementation of the interpreter, including its representation of Wasm's syntax and type system, the translation of its reduction rules, as well as the optimizations that the interpreter employs.

3.1. Syntax representation

This section details how the WebAssembly syntax is represented in the intrinsically-typed interpreter.

3.1.1. Types and stacks

The first step to defining an intrinsically-typed interpreter is to define the domain of WebAssembly types. We do so by using the Lean pattern of "universes à la Tarski" (similarly to the arithmetic language in Section 2.3.2.3):

```
inductive Ty where
  | i32
  | i64
  | f64

def Ty.type (ty : Ty) : Type :=
  match ty with
  | i32 => BitVec 32
  | i64 => BitVec 64
  | f64 => Float

instance : CoeSort Ty where
  coe := Ty.type
```

The `Ty` inductive above represents the available WebAssembly types, and the `Ty.type` function provides a mapping from WebAssembly types to Lean types. The `CoeSort` instance declaration allows us to use `Ty` values in contexts where Lean expects `Type`s, using `Ty.type` for the implicit conversion.

The WebAssembly integer types are mapped to Lean's native bit-vector types. This is because some instructions have both *signed* and *unsigned* variants (e.g. division), and the `BitVec` type allows us to interpret integer values as either signed or unsigned on a per-operation basis.

While they are part of the WebAssembly specification, single-precision floating-point numbers (`f32`) are not supported in our interpreter, because Lean does not natively support them.

The types of WebAssembly stacks (or *stack types* for short) are simply represented as lists of value types (`List Ty`), using Lean's native list data structure. Lean's dependent types allow us to easily use data structures such as lists in types.

3.1.2. Instructions

To represent Wasm instructions in a type-safe way, it is necessary to encode a large part of the Wasm type system into the instruction type's indices, including the stack, local variables, labels, and other elements.

In this section, we will describe the encoding of Wasm instructions in Lean, starting with a simple definition that is sufficient to represent simple instructions, and extending it as we add support for more features and instructions.

First and foremost, the most important information to encode are the types of an instruction's input and output stacks. To achieve this, we use an inductive type `Instr` that is indexed by 2 stack types:

```
inductive Instr : List Ty -> List Ty -> Type where
  | drop : Instr [a] []
  | i32_const : i32.type -> Instr [] [i32]
  | i32_add : Instr [i32, i32] [i32]
  | ... -- more instructions, described later
```

Each constructor (corresponding to one WebAssembly opcode) constrains the types of the stacks to match the instruction's inputs and outputs. The type of each instruction closely matches its type in the official specification. For example, the `i32_add` constructor has type `Instr [i32, i32] [i32]`, which matches its equivalent instruction: i32.add : i32 i32 → i32.

Similar to their definition in the specification, instruction sequences can be represented by a recursive type (here called `Instrs`, plural for `Instr`):

```
inductive Instrs : List Ty -> List Ty -> Type where
  | empty : Instrs i i
  | seq : Instr i o -> Instrs o o' -> Instrs i o'
```

The `empty` constructor produces an empty instruction sequence, which leaves its stack unchanged. The `seq` constructor appends a single instruction at the head of an existing sequence, composing their types such that the output stack of the first instruction matches the input stack of the rest of the sequence.

However, our current encoding of instruction types is too restrictive. Trying to construct the abstract syntax tree for the simple instruction sequence `i32.add i32.add` leads to a compilation error:

```
-- The #check directive instructs Lean to type-check the following expression.
#check seq i32_add (seq i32_add empty)

-- Error:
-- application type mismatch
--   seq i32_add
-- argument
--   i32_add
-- has type
--   Instr [i32, i32] [i32] : Type
-- but is expected to have type
--   Instr [i32] ?m
```

This error is caused by the same issue described in Section 2.1.1.3: the output of the first instruction (`[i32]`) must *exactly* match the input to the second (`[i32, i32]`).

The type system described in the official specification relies on the *weakening rule* (shown in Figure 5) to type such instruction sequences:

$$\dfrac{\dfrac{\dfrac{}{C \vdash \texttt{i32.add} : \texttt{i32 i32} \to \texttt{i32}}\,\textit{T-add}}{C \vdash \texttt{i32.add} : \texttt{i32 i32 i32} \to \texttt{i32 i32}}\,\textit{T-weaken} \quad \dfrac{}{C \vdash \texttt{i32.add} : \texttt{i32 i32} \to \texttt{i32}}\,\textit{T-add}}{C \vdash \texttt{i32.add i32.add} : \texttt{i32 i32 i32} \to \texttt{i32 i32}}\,\textit{T-seq}$$

Figure 16: Typing derivation for the sequence `i32.add i32.add`.

To allow our abstract syntax tree definition to represent non-trivial instruction sequences, we need to add support for weakening. One possibility would be to add a `weaken` constructor to the `Instr` type, mirroring the weakening rule in the type system. However, this gives rise to another issue, in that constructing a valid AST would necessitate the introduction of `weaken`s in non-obvious locations. The solution employed in LeanWasm is to embed weakening in the types of the instruction constructors, by making them *polymorphic* in the base of the stack:

```
inductive Instr : List Ty -> List Ty -> Type where
  | drop : Instr (a :: i) i
  | i32_const : i32.type -> Instr i (i32 :: i)
  | i32_add : Instr (i32 :: i32 :: i) (i32 :: i)
  | ... -- more instructions, described later
```

In this definition, the type of each constructor is (implicitly) parameterized over an input stack i. The type of `i32_add` specifies that its input stack should contain 2 integers (`i32`) followed by i, and its output is just one integer followed by i.

The expression `seq i32_add (seq i32_add empty)` is now accepted by Lean, as the compiler can infer the appropriate instantiations for i for each occurrence of `i32_add`.

Local variables

Next, let's consider extending `Instr` data type with support for local variables and their associated instructions (namely `local.get`, `local.set`, and `local.tee`).

In WebAssembly, the local variables of each function are declared up-front at the start of the function's body, and they are identified with numeric indices rather than alphanumeric identifiers. To track their types in our encoding, we add a new type parameter to `Instr`, of type `List Ty` (each element of the list corresponding to one local variable):

```
--                \/ new \/
inductive Instr : List Ty -> List Ty -> List Ty -> Type where
  -- Existing instructions become polymorphic in the new parameter
  | drop : Instr locs (a :: i) i
  | ...
```

The instructions that make use of local variables refer to them by their index. In order to prevent the construction of an instruction that refers to an invalid local variable (either an inexistent variable, or one of the wrong type), the indices are encoded using *well-typed list indices*:

```
inductive Ix {t : Type} (a : t) : List t -> Type where
  | hit : Ix a (a :: as)
  | miss : Ix a as -> Ix a (b :: as)
```

The inductive type `Ix` represents an index into a list, together with the type of the element at that index. The `hit` constructor creates an index that points to the head of the list, while the `miss` constructor inductively constructs an index for a larger list, if given an index into its tail.

Using well-typed list indices, we can define new constructors corresponding to the `local.*` family of instructions:

```
inductive Instr : List Ty -> List Ty -> List Ty -> Type where
  | ...
  | local_get : Ix t locs -> Instr locs i (t :: i)
  | local_set : Ix t locs -> Instr locs (t :: i) i
  | local_tee : Ix t locs -> Instr locs (t :: i) (t :: i)
```

`local_get` reads the value of a local variable and pushes it onto the stack. `local_set` pops a value off the stack and writes it to the variable. `local_tee` also writes the value on top of the stack to a variable, but doesn't pop it. These instructions are also defined to be stack-polymorphic, to allow them to be easily composed with `seq`.

While the `Ix` type enables us to safely encode variable indices, its constructors are unwieldy to use. For example, a `local.get` instruction targeting the third variable needs to be written as `local_get (miss (miss hit))`.

Thankfully, Lean supports fine-grained overloading of numeric literals through the `OfNat` typeclass. Using it, we can define an implicit coercion from natural numbers to well-typed indices, which allows us to write the above as `local_get 2` instead:

```
instance : OfNat (Ix a (a :: as)) 0 where
  ofNat := hit

instance [o : OfNat (Ix a as) n] : OfNat (Ix a (b :: as)) (n+1) where
  ofNat := miss o.ofNat
```

The base instance for `OfNat 0` maps the literal `0` to the constructor `hit`, while the recursively-defined instance for `OfNat (n+1)` defines mappings from the rest of the naturals into `miss` constructors. The square brackets denote an instance constraint.

Control-flow

To extend the `Instr` type with control instructions such as `block`, `loop`, and `br`, we need to track the list of in-scope labels and their types. This looks similar to the representation of local variables, but requires a list of *stack types* (as each label is represented by a stack type):

```
--                              \/ new \/
inductive Instr : List Ty -> List (List Ty) -> List Ty -> List Ty -> Type where
  | ...
```

The `block` and `loop` instructions make use of this additional parameter to introduce new labels:

```
inductive Instr : List Ty -> List (List Ty) -> List Ty -> List Ty -> Type where
  | ...
  | block : Instrs locs (o :: lbls) i o -> Instr locs lbls i o
  | loop  : Instrs locs (i :: lbls) i o -> Instr locs lbls i o
```

The `block` instruction wraps a sequence of instructions, and introduces a new label typed with its output stack into the context of the wrapped instructions. Similarly, `loop` introduces a label typed with its *input* stack.

The above definitions closely match the typing rules for `block` and `loop`. However, much like the naive translation of `i32_add`, they don't take weakening into account. A more accurate encoding witnesses the fact that there may be more values on the stack than the block's type specifies:

```
inductive Instr : List Ty -> List (List Ty) -> List Ty -> List Ty -> Type where
  | ...
  | block : Instrs locs (o :: lbls) i o -> Instr locs lbls (i ++ b) (o ++ b)
  | loop  : Instrs locs (i :: lbls) i o -> Instr locs lbls (i ++ b) (o ++ b)
```

Because the stack types i and o are themselves lists, weakening is represented by using Lean's built-in list concatenation operator `(++)`.

Branching

The `br` and `br_if` instructions reuse the `Ix` type to refer to label indices in a type-safe manner:

```
inductive Instr : List Ty -> List (List Ty) -> List Ty -> List Ty -> Type where
  | ...
  | br : Ix i lbls -> Instr locs lbls (i ++ b) o
  | br_if : Ix i lbls -> Instr locs lbls (i32 :: i ++ b) (i ++ b)
```

Because the output type of `br` is unconstrained (as executing a branch discards the instructions following it), it does not need to refer to the polymorphic stack base b.

Administrative instructions

Since LeanWasm mirrors the reference interpreter's handling of control-flow, it needs to support the same administrative instructions. Their types are given below:

```
inductive Instr : List Ty -> List (List Ty) -> List Ty -> List Ty -> Type where
  | ...
  | label :
      (∀ b', Instr locs lbls (i ++ b') (o ++ b')) ->
      Stack i' -> Instrs locs (i :: lbls) i' o ->
      Instr locs lbls b (o ++ b)
  | breaking : Ix i' lbls -> Stack (i' ++ b) -> Instr locs lbls i o
```

These administrative instructions are more general than their source-level counterparts, which is reflected in their more complex types. Notably, the type of a label's *continuation* needs to be stack-base-polymorphic (`∀` is Lean's notation for a dependent function type), because it's not known ahead of time in what context the continuation will be executed.

The `breaking` instruction has a similar type to `br`, with the addition of also capturing the current stack when it's created. This instruction is described in more detail in Section 3.2.2.

Instruction kinds

Administrative instructions only arise during execution, they are not part of WebAssembly's surface syntax. Currently, the `Instr` type does not make this distinction, allowing for the creation of programs containing administrative instructions. To prevent this without duplicating the entire type, we extend it with yet another type parameter, representing the instruction's *kind*:

```
inductive InstrKind where
  | src
  | spec

inductive Instr : InstrKind -> ... -> Type where
  | drop : Instr knd locs lbls i o
  | ...
  | label : ... -> Instr spec locs lbls b (o ++ b)
```

The `InstrKind` type distinguishes between source instructions (`src`) and administrative instructions (`spec`).

Source-level instructions are polymorphic in their kind: They can be instantiated as either `src` or `spec` depending on the context. Administrative instructions, on the other hand, have their kind fixed as `spec`. With this distinction, we can represent source programs as `Instr src locs lbls i o`, and programs that are mid-evaluation as `Instr spec locs lbls i o`.

Arithmetic instructions

So far, the only arithmetic instruction we considered was `i32_add`. This was a simplification, as the WebAssembly specification defines arithmetic instructions in a generic way:

$$\frac{}{C \vdash t.binop : t\ t \to t}\ \textit{T-binop}$$

$$\frac{}{C \vdash t.relop : t\ t \to \mathsf{i32}}\ \textit{T-relop}$$

Figure 17: Typing rules for arithmetic and comparison instructions.

$$\begin{array}{rcl} binop & ::= & \textbf{add} \\ & | & \textbf{sub} \\ & | & \textbf{mul} \\ & | & \ldots \\ relop & ::= & \textbf{eq} \\ & | & \textbf{ne} \\ & | & \textbf{lt} \\ & | & \textbf{gt} \\ & | & \ldots \end{array}$$

Figure 18: Syntax of arithmetic operators.

The above figures show the (slightly simplified) syntax of arithmetic and comparison operators, and the typing rules of the generic arithmetic and comparison instructions. Binary operators take as inputs two values of type t and produce another value of type t, while relational operators take two inputs of type t and produce an `i32` that encodes a boolean value as either 0 or 1.

In LeanWasm, the definitions for `binop` and `relop` are split between integer operators and floating-point operators:

```
inductive Signedness where
  | s
  | u

inductive IBinOp where
  | add
  | mul
  | div : Signedness -> IBinOp
  | and
  | xor
  | ...

inductive IRelOp where
  | eq
  | ne
  | lt : Signedness -> IRelOp
  | gt : Signedness -> IRelOp
  | ...

inductive FBinOp where
  | f_add
  | f_mul
  | ...

inductive FRelOp where
  | f_eq
```

```
  | f_ne
  | ...
```

In the case of integers, operators such as division and comparison are parametric over the `Signedness` of the operation: they can interpret their inputs as either signed or unsigned. Integers also support bitwise operators, while floats do not.

In order to avoid duplication when representing the integer-specific operations, we generalize the `i32` and `i64` constructors of the `Ty` type:

```
inductive Ty where
  | i : Size -> Ty
  | f64 : Ty

inductive Size where
  | _32
  | _64

instance : OfNat Size 32 where
  ofNat := _32

instance : OfNat Size 64 where
  ofNat := _64

instance : Coe Size Nat where
  toNat := ... -- inverse of OfNat conversion

@[match_pattern]
abbrev Ty.i32 : Ty := Ty.i 32

@[match_pattern]
abbrev Ty.i64 : Ty := Ty.i 64
```

The new definition of `Ty` has only one constructor for integer types, which is parameterized over a `Size`, a simple inductive type with cases for 32-bit and 64-bit integers. To make working with `Sizes` easier, we also define implicit coercions between `Sizes` and their corresponding `Nat` literals. The pattern synonyms `Ty.i32` and `Ty.i64` allow the previous definitions to keep using the old names for the types.

Using the new `Ty.i` constructor, we can define the arithmetic instructions generically:

```
inductive Instr knd locs lbls i o where
  | ...
  | i_binop : IBinOp -> Instr knd locs lbls (Ty.i s :: Ty.i s :: i) (Ty.i s :: i)
  | i_relop : IBinOp -> Instr knd locs lbls (Ty.i s :: Ty.i s :: i) (i32 :: i)

  | f_binop : FBinOp -> Instr knd locs lbls (f64 :: f64 :: i) (f64 :: i)
  | f_relop : FRelOp -> Instr knd locs lbls (f64 :: f64 :: i) (i32 :: i)
```

Memory

Since Wasm's linear memory is untyped, memory-related instructions are simple to encode without extending the intrinsically-typed machinery:

```
inductive Instr knd locs lbls i o where
  | ...
  | memory_size : Instr knd locs lbls i (i32 :: i)
  | memory_grow : Instr knd locs lbls (i32 :: i) (i32 :: i)
```

```
  | i_load : Instr knd locs lbls (i32 :: i) (Ty.i s :: i)
  | i_store : Instr knd locs lbls (Ty.i s :: i32 :: i) i
```

`memory_size` queries the current size of the memory buffer, and `memory_grow` allows programs to extend it, returning the new size as output. The address space of the memory is limited to 32 bits, to increase portability.

`i_load` and `i_store` allow programs to read from, and respectively write to the memory buffer at a specified address. While they are defined for all value types in the Wasm specification, in LeanWasm they can only be used with integral types (due to Lean's limited floating-point support).

Trapping

The ability to access memory at arbitrary addresses brings along the possibility of *trapping* if an address is out of bounds. To represent trapping states, LeanWasm, like the reference interpreter, includes a new administrative instruction:

```
inductive Instr knd locs lbls i o where
  | ...
  | trapping : Instr spec locs lbls i o
```

The new instruction's kind is fixed to `spec`, like the previous administrative instructions. Its input and output type are completely unconstrained, because it can appear in any context, and once a `trapping` instruction is encountered, the entire instruction sequence reduces to a single `trapping`.

3.2. Execution

This section describes an interpreter for the well-typed syntax tree introduced in the previous section, modeled after the official reference interpreter [12].

3.2.1. Contexts and configurations

The previously-described `Instr` type has gained a large number of type parameters as it was extended to support more and more Wasm features. The `Context` type neatly packs them into a record (called a `structure` in Lean):

```
structure Context where
  locs : List Ty
  lbls : List (List Ty)
  i : List Ty
  o : List Ty
```

Using this context type, we can define the type of *configurations*, which encapsulate the state of the Wasm abstract machine.

```
abbrev Stack ts := HList Ty.type ts

inductive SpecConfig (c : Context) where
  mem : Memory
  ls : Stack c.locs
  vs : Stack c.i
  es : Instrs spec c.locs c.lbls c.i c.o
```

A configuration has 4 components:

- mem: The linear memory buffer.
- ls: The list of values of the local variables.
- vs: The value stack that instructions operate on.
- es: The instruction sequence to execute.

The stack and the list of local variables are stored as *heterogenous lists* parameterized by the types present in the context. This ensures that the values of the stack and locals always match their expected types. Because the instruction stream es is parameterized in the same types, this guarantees that configurations are well-typed: it's impossible to construct a configuration where the values on the stack don't match the types expected by the instructions.

Notably, the configuration's instruction stream is of kind `spec`, because configurations represent the state of Wasm programs mid-execution, so they can include administrative instructions. When constructing an initial configuration, a sequence of source instructions (of kind `src`) can be trivially converted to the `spec` kind. This coercion is provided by a helper function `Instrs.toSpec`.

Memory representation

One aspect in which LeanWasm differs from the reference interpreter is its representation of the linear memory. The reference interpreter stores the memory as an *immutable* linked lists of bytes. This is inefficient because accessing elements in a linked list has linear time complexity, so reading data at a memory address takes time proportional to the size of the memory. Furthermore, due to immutability, when *writing* data to memory a new linked list needs to be created, reallocating all the bytes preceding the written-to address, which is also a linear time operation.

The reference interpreter represents the linear memory in this way due to its goal of keeping a very close correspondence to the official specification.

In LeanWasm, the memory is represented as a `ByteArray`, Lean's native type for packed byte arrays:

```
abbrev Memory := ByteArray
```

The `ByteArray` type exposes an immutable interface, similar to a linked list, where modifying a value in the array returns a new array as output. However, Lean's novel reference-counting garbage-collector [19] enables *functional-but-in-place* programming: If the reference to the `ByteArray` is unique, the underlying array is mutated in-place, rather than allocating and returning a new array.

Because the memory buffer is used *linearly* in LeanWasm, this allows to write the interpreter in a purely functional way, while generating efficient code that mutates the memory in-place without reallocating it.

3.2.2. Relational interpreter

In order to keep a close correspondence with the with the official semantics, LeanWasm includes a *relational* definition of the interpreter. This means that the small-step reduction relation is expressed as an inductive datatype parameterized in its input and output configurations.

The following sections describe this relational interpreter, showing how it encodes the reduction rules for the instructions that make up the `Instr` type.

```
inductive SpecStep :
  (c : Context) -> (c' : Context) -> SpecConfig c -> SpecConfig c' -> Type
where
  | sstep_drop :
      SpecStep ⟨locs, lbls, _ :: i, o⟩ ⟨locs, lbls, i, o⟩
               ⟨mem, ls, cons a vs, seq drop es⟩ ⟨mem, ls, vs, es⟩

  | sstep_const :
      SpecStep ⟨locs, lbls, i, o⟩ ⟨locs, lbls, t :: i, o⟩
               ⟨mem, ls, vs, seq (const a) es⟩ ⟨mem, ls, cons a vs, es⟩
  | ...
```

The above definition contains the signature of the `SpecStep` type , and an encoding of the reduction rule for the `drop` instruction. Because the values on the stack change as evaluation progresses, the *type* of the configuration also changes at each step. This is reflected by the fact that `SpecStep` is parameterized not only in the input and output configurations, but also in their context types.

The constructor for each reduction rule refines the type indices of `SpecStep` to reflect changes it performs on the configuration, as well as its context. As an example, the `drop` rule presented above defines its input stack type to be `_ :: i`, which will only match non-empty stacks, and defines its output as `i`, the tail of the input stack. Similarly, it removes the first value from the stack (`cons` is one of the constructors for heterogenous lists), and removes the instruction from the configuration once it's been executed.

Similarly, the rule for `const` pushes the instruction's immediate argument a onto the stack, so its output stack contains one more type than its input. Even though we didn't explicitly introduce the type variable t, it is constrained by the type indices of `Instr` to match the type introduced by the `const` instruction. Writing the wrong type for the context would result in a compilation error.

It might look surprising that the rules above apply their modifications the *input* stack type of the *output* context, rather than the *output* stack type. This is because the type of a configuration's current stack is described by the input stack type, so when defining the output configuration with an updated stack, it's necessary to update its *input* stack type. The output stack type of a configuration describes the shape of the stack at the end of evaluation, once the configuration reaches a final state.

Arithmetic instructions

Below are the reduction rules for arithmetic instructions (using the shorter Lean notation `.i` for `Ty.i`)

```
inductive SpecStep :
  (c : Context) -> (c' : Context) -> SpecConfig c -> SpecConfig c' -> Type
where
  | ...
  | sstep_i_binop :
      SpecStep ⟨locs, lbls, .i s :: .i s :: i, o⟩ ⟨locs, lbls, .i s :: i, o⟩
               ⟨mem, ls, cons b (cons a vs), seq (i_binop op) es⟩
               ⟨mem, ls, cons (op.eval a b) vs, es⟩
  | sstep_i_relop :
      SpecStep ⟨locs, lbls, .i s :: .i s :: i, o⟩ ⟨locs, lbls, i32 :: i, o⟩
               ⟨mem, ls, cons b (cons a vs), seq (i_relop op) es⟩
               ⟨mem, ls, cons (op.eval a b) vs, es⟩

  | sstep_f_binop :
      SpecStep ⟨locs, lbls, f64 :: f64 :: i, o⟩ ⟨locs, lbls, f64 :: i, o⟩
               ⟨mem, ls, cons b (cons a vs), seq (f_binop op) es⟩
               ⟨mem, ls, cons (op.eval a b) vs, es⟩
  | sstep_f_relop :
      SpecStep ⟨locs, lbls, f64 :: f64 :: i, o⟩ ⟨locs, lbls, i32 :: i, o⟩
               ⟨mem, ls, cons b (cons a vs), seq (f_relop op) es⟩
               ⟨mem, ls, cons (op.eval a b) vs, es⟩
```

Both the integer and floating-point versions of the arithmetic and comparison operators work in a similar way: they pop 2 values of the appropriate types from the stack, and push a single result back onto the stack. The actual computation is performed by helper functions defined on the binary operator's type, such as `IBinOp.eval` and `IRelOp.eval`, which map Wasm's arithmetic operations to Lean's native operations on bit-vectors and floating-points, taking into account signedness if the operator specifies it. Below is an excerpt from the implementation of `IBinOp.eval`:

```
def IBinOp.eval (op : IBinOp) (a b : Ty.i s) : Ty.i s :=
  match op with
  | add => a + b
  | sub => a - b
  | mul => a * b

  -- Pattern-matching on signedness and mapping to the appropriate operation
  | div s => a.sdiv b
  | div u => a.udiv b

  | and => a &&& b
  | or  => a ||| b

  | ...
```

Local variables

The instructions that manipulate local variables are reduced in a similar manner:

```
inductive SpecStep :
  (c : Context) -> (c' : Context) -> SpecConfig c -> SpecConfig c' -> Type
where
  | ...
  | sstep_local_get :
      SpecStep ⟨locs, lbls, i, o⟩ ⟨locs, lbls, _ :: i, o⟩
               ⟨mem, ls, vs, seq (local_get ix) es⟩ ⟨mem, ls, cons (ls.get ix) vs, es⟩
  | sstep_local_set :
      SpecStep ⟨locs, lbls, _ :: i, o⟩ ⟨locs, lbls, i, o⟩
               ⟨mem, ls, cons a vs, seq (local_set ix) es⟩ ⟨mem, ls.set ix a, vs, es⟩
  | sstep_local_tee :
      SpecStep
        ⟨locs, lbls, t :: i, o⟩ ⟨locs, lbls, t :: i, o⟩
        ⟨mem, ls, cons a vs, seq (local_tee ix) es⟩ ⟨mem, ls.set ix a, cons a vs, es⟩
```

Unlike the instructions seen so far, the `local_*` family of instructions refer to the list of local variables *ls*. The helper functions `HList.get` and `HList.set` are used to read and update elements in this list at the indices indicated by the well-typed `Ix` values.

Control instructions

The control instructions `block` and `loop` are immediately reduced to the more general `label` administrative instruction, as described previously:

```
inductive SpecStep :
  (c : Context) -> (c' : Context) -> SpecConfig c -> SpecConfig c' -> Type
where
  | ...
  | sstep_block :
      SpecStep
        ⟨locs, lbls, i' ++ i, o⟩ ⟨locs, lbls, i, o⟩
        ⟨mem, ls, vs' ++ vs, seq (block es') es⟩
        ⟨mem, ls, vs, seq (label (λ _ => nop) vs' es') es⟩

  | sstep_loop :
      SpecStep
        ⟨locs, lbls, i' ++ i, o⟩ ⟨locs, lbls, i, o⟩
        ⟨mem, ls, vs' ++ vs, seq (loop es') es⟩
        ⟨mem, ls, vs, seq (label (λ _ => loop es') vs' es') es⟩
```

The λ-abstractions wrapping the label continuations generalize over the base of the stack. The surrounding types provide the Lean compiler with enough context to infer this stack base, so the function argument can simply be ignored, by use of the wildcard (_) pattern.

Branching

The reduction of branch instructions follows the implementation of the reference interpreter. A `br` instruction is immediately reduced to the administrative instruction `breaking`, which also captures the current stack of values. For `br_if`, there are 2 reduction rules: One that rewrites the `br_if` to an unconditional `br` with the same target, if the topmost value on the stack is non-zero, and another that completely eliminates the `br_if`, if the topmost stack value is 0.

```
inductive SpecStep :
  (c : Context) -> (c' : Context) -> SpecConfig c -> SpecConfig c' -> Type
where
  | ...
  | sstep_br :
        SpecStep
          ⟨locs, lbls, i' ++ i, o⟩ ⟨locs, lbls, i' ++ i, o⟩
          ⟨mem, ls, vs, seq (br ix) es⟩ ⟨mem, ls, vs, seq (breaking ix vs) es⟩

  | sstep_br_if_false :
      SpecStep
        ⟨locs, lbls, i32 :: i ++ i', o⟩ ⟨locs, lbls, i ++ i', o⟩
        ⟨mem, ls, List.cons_append _ _ _ ▸ cons 0 vs, seq (br_if ix) es⟩
        ⟨mem, ls, vs, es⟩

  | sstep_br_if_true :
      SpecStep
        ⟨locs, lbls, i32 :: i ++ i', o⟩ ⟨locs, lbls, i ++ i', o⟩
        ⟨mem, ls, List.cons_append _ _ _ ▸ cons n vs, seq (br_if ix) es⟩
        ⟨mem, ls, vs, seq (br ix) es⟩
```

The `(▸)` operator used in the constructors for `br_if` is Lean's built-in type substitution. For any two types α and β, `(▸)` takes a proof that `α = β` on its left-hand side, and uses it to substitute occurrences of α with β in the type of its right-hand side expression. In this case, we are applying it with a proof that `cons` associates with list concatenation, to witness the fact that `(i32 :: i) ++ i'` is equal to `i32 :: (i ++ i')`.

Administrative instructions

The constructor encoding the reduction rule for `labels` is different from the rest, in that it depends on another `SpecStep` value:

```
inductive SpecStep :
  (c : Context) -> (c' : Context) -> SpecConfig c -> SpecConfig c' -> Type
where
  | ...
  | sstep_label_inner :
      SpecStep
        ⟨locs, _ :: lbls, i', o'⟩ ⟨locs, _ :: lbls, i'', o'⟩
        ⟨mem, ls, vs', es'⟩ ⟨mem', ls', vs'', es''⟩ ->
      SpecStep
        ⟨locs, lbls, i, o⟩ ⟨locs, lbls, i, o⟩
        ⟨mem, ls, vs, seq (label k vs' es') es⟩
        ⟨mem', ls', vs, seq (label k vs'' es'') es⟩
```

The above rule states that given a `SpecStep` encoding the reduction of the label's body, we can construct a larger `SpecStep` encoding the reduction of the `label` itself.

Labels are eliminated either when their body has been reduced to an empty instruction sequence, or when a `breaking` instruction is encountered:

```
inductive SpecStep :
  (c : Context) -> (c' : Context) -> SpecConfig c -> SpecConfig c' -> Type
where
  | ...
  | sstep_label_done :
      SpecStep
        ⟨locs, lbls, i, o⟩ ⟨locs, lbls, i' ++ i, o⟩
        ⟨mem, ls, vs, seq (label k vs' done) es⟩
        ⟨mem, ls, vs' ++ vs, es⟩

  | sstep_label_breaking_hit :
      SpecStep
        ⟨locs, lbls, i, o⟩ ⟨locs, lbls, i' ++ i, o⟩
        ⟨mem, ls, vs, seq (label k vs' (seq (breaking hit vs'') es')) es⟩
        ⟨mem, ls, vs''.take _ ++ vs, seq (k _) es⟩

  | sstep_label_breaking_miss :
      SpecStep
        ⟨locs, lbls, i, o⟩ ⟨locs, lbls, i, o⟩
        ⟨mem, ls, vs, seq (label k vs' (seq (breaking (miss ix) vs'') es')) es⟩
        ⟨mem, ls, vs, seq (breaking ix vs'') es⟩
```

When the first instruction in the body of a `label` is a `breaking` targeting that label, `sstep_label_breaking_hit` reduces the entire label to its continuation instruction k. Because the continuation is stack-base-polymorphic, it has to be instantiated with the type of the current stack. Lean is able to infer this type by itself, so we instruct it do so by passing an underscore as the function parameter.

The reduction rule `sstep_label_breaking_miss` handles the remaining case, when a `breaking` instruction targeting an outer label is encountered. Here, the label is replaced with a single `breaking` instruction, but with a decremented label index (the outermost `miss` constructor is eliminated).

Trapping

During execution, a Wasm program might *trap* if it tries to access memory with an out-of-bounds address. Trapping is represented as an additional administrative instruction, with the following reduction rule:

```
inductive SpecStep :
  (c : Context) -> (c' : Context) -> SpecConfig c -> SpecConfig c' -> Type
where
  | ...
  | sstep_trapping :
      SpecStep ⟨locs, lbls, i, o⟩ ⟨locs, lbls, i, o⟩
               ⟨mem, ls, vs, seq trapping es⟩ ⟨mem, ls, vs, seq trapping done⟩
```

If the first instruction in the configuration's instruction list is `trapping`, then the rule above reduces the entire instruction list to a single `trapping` instruction.

Trapping instructions are "bubbled up" through labels, in a similar fashion to `breaking` instructions:

```
inductive SpecStep :
  (c : Context) -> (c' : Context) -> SpecConfig c -> SpecConfig c' -> Type
where
  | ...
  | sstep_label_trapping :
    SpecStep
      ⟨locs, lbls, i, o⟩ ⟨locs, lbls, i, o⟩
      ⟨mem, ls, vs, seq (label k vs' (seq trapping es')) es⟩ ⟨mem, ls, vs, seq trapping es⟩
```

Memory

The reduction rule for the `memory_size` instruction is straight-forward, pushing the size of the memory onto the stack:

```
inductive SpecStep :
  (c : Context) -> (c' : Context) -> SpecConfig c -> SpecConfig c' -> Type
where
  | ...
  | sstep_memory_size :
      SpecStep
        ⟨locs, lbls, i, o⟩ ⟨locs, lbls, i32 :: i, o⟩
        ⟨mem, ls, vs, seq memory_size es⟩ ⟨mem, ls, cons mem.size vs, es⟩
```

The reduction rules for the other memory-related instructions are more complex, as they have to account for the possibility of trapping. For example, let's consider the reduction rules for the `memory_grow` instruction, which can trap if its argument (the number of additional bytes to allocate) is negative:

```
inductive SpecStep :
  (c : Context) -> (c' : Context) -> SpecConfig c -> SpecConfig c' -> Type
where
  | ...
  | sstep_memory_grow_trap :
      mem.grow a = none ->
      SpecStep
        ⟨locs, lbls, i32 :: i, o⟩ ⟨locs, lbls, i, o⟩
        ⟨mem, ls, cons a vs, seq memory_grow es⟩
        ⟨mem, ls, vs, seq trapping es⟩

  | sstep_memory_grow_ok :
      mem.grow a = some mem' ->
      SpecStep
        ⟨locs, lbls, i32 :: i, o⟩ ⟨locs, lbls, i32 :: i, o⟩
          ⟨mem, ls, cons a vs, seq memory_grow es⟩
          ⟨mem', ls, cons mem'.size vs, es⟩
```

There are 2 rules for the `memory_grow` instruction, one that specifies the trapping case, and one that specifies the valid execution case. The logic that checks if the argument is valid is encapsulated in the helper function `Memory.grow`, which returns an `Option` type: `none` if the argument was invalid, or a `some` containing the enlarged memory buffer otherwise. The constructor for each rule requires as parameter an equality proof relating the output of `mem.grow` to the appropriate `Option` constructor. In the trapping case, the `memory_grow` instruction is replaced with a `trapping` instruction. In the case where the `grow` operation succeeds, the configuration's memory is updated and the new memory's size is pushed onto the stack.

The reduction rules for the `i_load` and `i_store` instructions (shown below) are similar: For each instruction, there is a constructor for the trapping case and one for the non-trapping case, each depending on an equality proof.

```
inductive SpecStep :
  (c : Context) -> (c' : Context) -> SpecConfig c -> SpecConfig c' -> Type
where
  | ...
  | sstep_i_load_trap :
      mem.loadILittleEndian addr = none ->
      SpecStep
        ⟨locs, lbls, i32 :: i, o⟩ ⟨locs, lbls, i, o⟩
        ⟨mem, ls, cons addr vs, seq i_load es⟩
        ⟨mem, ls, vs, seq trapping es⟩

  | sstep_i_load_ok :
      mem.loadILittleEndian addr = some a ->
      SpecStep
        ⟨locs, lbls, i32 :: i, o⟩ ⟨locs, lbls, .i s :: i, o⟩
        ⟨mem, ls, cons addr vs, seq i_load es⟩
        ⟨mem, ls, cons a vs, es⟩

  | sstep_i_store_trap :
      mem.storeILittleEndian addr a = none ->
      SpecStep
        ⟨locs, lbls, .i s :: i32 :: i, o⟩ ⟨locs, lbls, i, o⟩
        ⟨mem, ls, cons a (cons addr vs), seq i_store es⟩
        ⟨mem, ls, vs, seq trapping es⟩

  | sstep_i_store_ok :
      mem.storeILittleEndian addr a = some mem' ->
      SpecStep
        ⟨locs, lbls, .i s :: i32 :: i, o⟩ ⟨locs, lbls, i, o⟩
        ⟨mem, ls, cons a (cons addr vs), seq i_store es⟩
        ⟨mem', ls, vs, es⟩
```

The helper functions `Memory.loadILittleEndian` and `Memory.storeILittleEndian` check that all addresses are in bounds and read, respectively update the corresponding bytes in the memory.

Execution traces

So far we have shown the reduction rules for the small-step reduction relation `SpecStep`. To represent a sequence of reductions corresponding to an *execution trace*, we define the `SpecSteps` type, which just describes the reflexive, transitive closure of the `SpecStep` relation:

```
inductive SpecSteps :
  (c : Context) -> (c' : Context) -> SpecConfig c -> SpecConfig c' -> Type
where
  | ssteps_refl :
      SpecSteps ⟨locs, lbls, i, o⟩ ⟨locs, lbls, i, o⟩
                ⟨mem, ls, vs, es⟩ ⟨mem, ls, vs, es⟩

  | ssteps_trans :
      SpecStep ⟨locs, lbls, i, o⟩ ⟨locs, lbls, i', o'⟩
               ⟨mem, ls, vs, es⟩ ⟨mem', ls', vs', es'⟩ ->

      SpecSteps ⟨locs, lbls, i', o'⟩ ⟨locs, lbls, i'', o''⟩
                ⟨mem', ls', vs', es'⟩ ⟨mem'', ls'', vs'', es''⟩ ->
```

```
SpecSteps ⟨locs, lbls, i, o⟩ ⟨locs, lbls, i'', o''⟩
          ⟨mem, ls, vs, es⟩ ⟨mem'', ls'', vs'', es''⟩
```

The `ssteps_refl` constructor builds an empty execution trace, while the inductive `ssteps_trans` constructor prepends a single reduction step to an existing execution trace.

Final states

A Wasm configuration can reach a *final state*, a state in which trying to apply a further reduction results in the same configuration. There are 2 cases in which a configuration is considered final: either its instruction list is empty (execution has finished), or the head of the list is a `trapping` administrative instruction (execution resulted in a trap):

```
def SpecConfig.isFinal (cfg : SpecConfig ctx) : Bool :=
  match cfg.es with
  | done => true
  | seq trapping _ => true
  | _ => false

def SpecConfig.isTrap (cfg : SpecConfig ctx) : Bool :=
  match cfg.es with
  | seq trapping _ => true
  | _ => false
```

3.2.3. Runnable interpreter

While the relational definition of the interpreter helps to clearly express the reduction rules, it is not directly runnable. The `SpecStep` and `SpecSteps` types just form a data structure representing an execution trace of a Wasm program.

In order to provide a *runnable* interpreter, LeanWasm also defines a *functional* version of the small-step reduction rule, implemented as the function `eval` in the namespace of the type `SpecConfig`:

```
def SpecConfig.eval (cfg : SpecConfig c) : Σ c', SpecConfig c' :=
  ...
```

The above function applies one reduction rule, mapping an input configuration to an output configuration. As with the relational interpreter, the type of the output configuration may differ from the input configuration, so the function uses a dependent sum type (introduced by the Σ symbol) to return the output context type alongside the output configuration.

The implementation of `eval` unpacks the configuration structure and passes its components to a helper `eval_unpacked`, which applies the reduction rules. Then, it packs the helper's results back into a configuration, together with the type of the resulting context:

```
def SpecConfig.eval (cfg : SpecConfig c) : Σ c', SpecConfig c' :=
  match cfg with
  | ⟨mem, ls, vs, es⟩ =>
      match SpecConfig.eval_unpacked mem ls vs es with
      | ⟨i', mem', ls', vs', es'⟩ => ⟨⟨c.locs, c.lbls, i', c.o⟩, ⟨mem', ls', vs', es'⟩⟩
```

We chose to implement `SpecConfig.eval` using the unpacked helper due to the difficulty of working with tuples and structures in Lean. In many situations, the compiler doesn't normalize field accesses on structure literals in types, so it fails to unify types of the form `P ⟨x, y⟩.x ⟨x, y⟩.y` with `P x y`.

`SpecConfig.eval_unpacked` proceeds by pattern-matching on the instruction sequence to determine which reduction rule to apply. The logic implemented by this function is extremely similar to that

encoded in the relational `SpecStep` type. The listing below includes an excerpt of its implementation, covering a few illustrative reduction rules:

```
def SpecConfig.eval_unpacked
  (mem : Memory) (ls : Stack locs) (vs : Stack i) (es : Instrs spec locs lbls i o)
  : Σ i', Memory × Stack locs × Stack i' × Instrs spec locs lbls i' o
:=
  match es with
  | done => ⟨_, mem, ls, vs, done⟩ -- Corresponds to sstep_done
  | seq e es =>
      match e, vs with
      | drop, cons _ vs => ⟨_, mem, ls, vs, es⟩ -- Corresponds to sstep_drop
      | ...
      | i_binop op, cons b (cons a vs) =>
          ⟨_, mem, ls, cons (op.eval a b) vs, es⟩ -- Corresponds to sstep_i_binop
      | ...
      | memory_grow, cons a vs =>
          match mem.grow a with
          | none => ⟨_, mem, ls, vs, seq trapping es⟩ -- sstep_memory_grow_trap
          | some mem' => ⟨_, mem', ls, cons mem'.size vs, es⟩ -- sstep_memory_grow_ok
      | ...
      | label k vs' es', vs =>
          let ⟨_, mem', ls', vs'', es''⟩ := SpecConfig.eval_unpacked mem ls vs' es'
          ⟨_, mem', ls', vs, seq (label k vs'' es'') es⟩ -- sstep_label_inner
      | ...
      | trapping, vs => ⟨_, mem, ls, vs, seq trapping done⟩ -- sstep_trapping
```

The existential variable i' in the function's output witnesses the fact that the shape of the stack may differ between the input and output configurations.

Execution traces

For applying multiple steps of evaluation on an initial configuration, similar to the `SpecSteps` type, the functional interpreter includes a `SpecConfig.evals` function (defined similarly in terms of an `unpacked` helper):

```
def SpecConfig.evals (cfg : Config c) (fuel : Nat) : Σ c', Config c' :=
  match cfg with
  | ⟨mem, ls, vs, es⟩ =>
      match SpecConfig.evals_unpacked mem ls vs es fuel with
      | ⟨i', mem', ls', vs', es'⟩ => ⟨⟨c.locs, c.lbls, i', c.o⟩, ⟨mem', ls', vs', es'⟩⟩

def SpecConfig.evals_unpacked
  (mem : Memory) (ls : Stack locs) (vs : Stack i) (es : Instrs spec locs lbls i o)
  (fuel : Nat)
  : Σ i', Memory × Stack locs × Stack i' × Instrs spec locs lbls i' o
:=
  match fuel with
  | 0 => ⟨_, mem, ls, vs, es⟩
  | fuel + 1 =>
      if es.isFinal then
        ⟨_, mem, ls, vs, es⟩
      else
        let ⟨_, mem', ls', vs', es'⟩ := SpecConfig.eval_unpacked mem ls vs es
        SpecConfig.evals_unpacked mem' ls' vs' es' fuel
```

An interesting detail of this function is its usage of a *fuel* parameter. This is necessary because WebAssembly is a Turing-complete language, and execution of a Wasm program might enter an infinite

loop. To avoid this, `SpecConfig.evals` short-circuits evaluation after *fuel* steps (or if the configuration reaches a final state).

3.2.4. Proving the runnable interpreter correct

While the pattern-matching logic in `SpecConfig.evals_unpacked` is very similar to the reduction rules described by `SpecStep`, it's possible that a bug could appear during their implementation that would cause them to go out of sync.

To determine that the functional evaluator encodes the same reduction rules as the relational one, we need a *soundness* proof showing that any execution trace of the functional interpreter can be mapped to an equivalent execution trace of the relational interpreter.

Small-step soundness proof

It's easier to split the proof into two parts: First prove that the small-step reduction relation is implemented equivalently by both interpreters, and then use this fact to construct a proof relating entire execution traces.

Concretely, the soundness proof for the small-step reduction is a function that for any two configurations cfg and cfg', given a proof that `SpecConfig.eval` reduces cfg to cfg', produces a value of type `SpecStep` representing the same reduction rule:

```
def SpecConfig.eval_sound :
  (cfg : SpecConfig ctx) -> (cfg' : SpecConfig ctx') ->
  (h : cfg.eval = (ctx', cfg')) ->
  : SpecStep ctx ctx' cfg cfg'
:=
  ...
```

The implementation of the function above would proceed in a very similar manner to the implementation of `eval` itself, leading to a lot of code duplication. However, this duplication can be avoided by using a technique known as *proof-carrying code*: We can update the definition of the functional interpreter to return a pair of the output configuration *and* the equivalent `SpecStep` structure. The updated type signature of `eval` is as follows:

```
def SpecConfig.eval_proofCarrying :
  (cfg : SpecConfig ctx) ->
  Σ ctx', (cfg' : SpecConfig ctx') × SpecStep ctx ctx' cfg cfg'
```

The function now outputs a triple consisting of the output context type, output configuration, and equivalent `SpecStep` structure. The difference between using the Σ or `(×)` symbol to denote the dependent pair is purely notational, Lean interprets both as the same type.

The signature of the unpacked helper function is updated similarly:

```
def SpecConfig.eval_unpacked_proofCarrying
  (mem : Memory) (ls : Stack locs) (vs : Stack i) (es : Instrs spec locs lbls i o)
  : Σ i',
    (mem' : Memory) × (ls' : Stack locs) × (vs' : Stack i')
    × (es' : Instrs spec locs lbls i' o)
    × SpecStep ⟨locs, lbls, i, o⟩ ⟨locs, lbls, i', o⟩
               ⟨mem, ls, vs, es⟩ ⟨mem', ls', vs', es'⟩
```

The more complex type signature complicates the implementation of the function, as it is now required to show that the types of all configuration components (and other intermediate expressions) match the types required by the `SpecStep` type's constructors. This necessitates the use of *inaccessible patterns*

[20]: pattern-matches on expressions that are not essential to the computation, but are required to specialize the types.

Unfortunately, we have not managed to implement this proof-carrying version of the function in its entirety. While most of the reduction rules are implemented and type-correct, there are 3 reduction rules which we failed to type properly, namely the reduction rules for `block`, `loop`, and, surprisingly, `nop`: the instruction that leaves the configuration completely unchanged. We have axiomatized the 3 missing cases using Lean's built-in `sorry` tactic, thus our soundness proof for the small-step relation is *partial*.

Soundness proof for execution traces

Assuming the soundness proof for small-step relation is correct, a soundness proof for *execution traces* can be implemented using the same technique of proof-carrying code:

```
def SpecConfig.evals_proofCarrying (cfg : SpecConfig ctx) (fuel : Nat)
  : Σ ctx', (cfg' : SpecConfig ctx') × SpecSteps ctx ctx' cfg cfg'
:=
  match cfg with
  | ⟨mem, ls, vs, es⟩ =>
      match SpecConfig.evals_unpacked_proofCarrying mem ls vs es fuel with
      | ⟨i', mem', ls', vs', es', ssteps⟩ =>
          ⟨⟨ctx.locs, ctx.lbls, i', ctx.o⟩, ⟨mem', ls', vs', es'⟩, ssteps⟩
```

The `unpacked` version follows similarly, constructing a `SpecSteps` value from the output of the first evaluation steps and the result of the recursive call:

```
def SpecConfig.evals_unpacked_proofCarrying
  (mem : Memory) (ls : Stack locs) (vs : Stack i)
  (es : Instrs spec locs lbls i o) (fuel : Nat)
  : Σ i',
      (mem' : Memory) × (ls' : Stack locs) × (vs' : Stack i')
      × (es' : Instrs spec locs lbls i' o)
      × SpecSteps ⟨locs, lbls, i, o⟩ ⟨locs, lbls, i', o⟩
                  (mem, ls, vs, es) ⟨mem', ls', vs', es'⟩
:=
  match fuel with
  | 0 => ⟨_, mem, ls, vs, es, ssteps_refl⟩
  | fuel + 1 =>
      if es.isFinal then
        ⟨_, mem, ls, vs, es, ssteps_refl⟩
      else
        let ⟨_, mem', ls', vs', es', sstep⟩ :=
          SpecConfig.eval_unpacked_proofCarrying mem ls vs es

        let ⟨_, mem'', ls'', vs'', es'', ssteps⟩ :=
          SpecConfig.evals_unpacked_proofCarrying mem' ls' vs' es' fuel

        ⟨_, mem'', ls'', vs'', es'', ssteps_trans sstep ssteps⟩
```

3.3. Optimizing control-flow

This section describes the inefficiencies of LeanWasm's (and the official interpreter's) representation of control-flow, and presents an improved approach to modeling control-flow, implemented as part of a new interpreter.

3.3.1. Inefficiencies of labels

The implementation of Wasm control-flow as nested `label` instructions, which is used by both LeanWasm and the reference interpreter, exhibits a particular inefficiency: Each step of reduction has to recursively traverse the nested `labels` to find the next reducible instruction. This leads to poor performance, especially for Wasm programs with deeply-nested labels, such as those resulting from compilation of higher-level languages with support for *irreducible* control-flow (such as `goto`) [21].

3.3.2. An improved representation of control-flow

The primary issue with the current implementation of labels is that the redex (the next reducible instruction) is deeply nested within a stack of labels, and the interpreter has to recurse through these labels at each step.

The optimization is based on a simple idea: turn this stack of labels *inside-out*, such that the innermost block of instructions is now readily accessible at the top of the stack. This technique, also known as a *zipper* [22], is prevalent in the context of functional programming.

3.3.3. The optimized interpreter

To realize this optimization, we implement a second intrinsically-typed Wasm interpreter that closely follows the previously-described `Spec` interpreter, differing only in their handling of control-flow.

Similar to the `Spec` interpreter, the optimized interpreter (nicknamed `Fast`) is comprised of the following components:

- A type for configurations: `FastConfig` (reusing the existing `Context` type).
- A relational definition of its reduction rules: `FastStep` and `FastSteps`.
- A functional, runnable definition of its reduction rules: `FastConfig.eval`, `FastConfig.evals`, with proof-carrying variants.
- A new instruction kind: `fast`.

These components are very similar to their `Spec` equivalents, as the reduction rules for most instructions (including arithmetic, locals, and memory) are identical between both interpreters. In the following sections, we will focus on the differences introduced in the `Fast` interpreter, and describe its optimized representation of control-flow.

Administrative instructions

Because of its different encoding of control-flow, the `Fast` interpreter does not require any administrative instructions. However, to make it easier to adapt the existing definitions, it includes a new instruction kind `fast`, and one new instruction of this kind, called `trap`, which serves the same purpose (and shares the same type and reduction rules) as the `trapping` administrative instruction. Its definition is provided below.

```
inductive InstrKind where
  | src
  | spec
  | fast -- new

inductive Instr knd locs lbls i o where
  | ...
  | trapping : Instr spec locs lbls i o

  | trap : Instr fast locs lbls i o
```

Configurations

The `Fast` interpreter's configuration type includes one additional field, representing a reified *label stack*:

```
inductive FastConfig (c : Context) where
  mem : Memory
  ls : Stack c.locs
  lstk : LabelStack c.locs c.lbls c.o
  vs : Stack c.i
  es : Instrs fast c.locs c.lbls c.i c.o
```

The role of the instructions es is also different: in the `FastConfig`, this field represents the innermost *currently-executing* instruction sequence, rather than the top-level list of instructions that make up the program.

Label stacks

The reified `LabelStack` stores the context surrounding the currently-executing instructions, in an inside-out fashion: The top of the label stack represents the immediate outer label context, and the bottom of the stack corresponds to the program's top-most label.

The definition of the `LabelStack` type is provided below:

```
inductive LabelStack (locs : List Ty) : List (List Ty) -> List Ty -> Type where
  | empty : LabelStack locs [] o
  | push :
      (∀ b', Instr fast locs lbls (i ++ b') (o ++ b')) ->
      Stack b -> Instrs fast locs lbls (o ++ b) o' ->
      LabelStack locs lbls o' ->
      LabelStack locs (i :: lbls) o
```

The `empty` constructor is the initial state of the label stack. When a `block` or `loop` is encountered during evaluation, the `push` constructor adds a new frame to the `LabelStack`.

The `push` constructor mirrors the definition of the `label` administrative instruction, which is reproduced below:

```
inductive Instr : InstrKind -> List Ty -> List (List Ty) -> List Ty -> List Ty -> Type
where
  | ...
  | label :
      (∀ b', Instr spec locs lbls (i ++ b') (o ++ b')) ->
      Stack i' -> Instrs spec locs (i :: lbls) i' o ->
      Instr spec locs lbls b (o ++ b)
```

To see this relation more clearly, let's first recap what each field of the `label` instruction represents:

- The *continuation*, an instruction sequence that gets executed when the label is branched to. The continuation is stack-base-polymorphic, as witnessed by the universal quantifier ∀.
- The inner stack, storing the values that the inner instruction sequence operates on. When a label is created, the stack is split according to its type, and the inner stack is stored inside the label, while the outer stack is stored in the vs field of the configuration.
- The inner instruction sequence, representing the body of the label. When a label is created, the inner instruction sequence is initialized with the body of the `block` (or `loop`). The rest of the program's instruction stream (the *outer* instructions) is sequenced after the `label` instruction, and stored in the es field of the configuration.

The fields of the `push` constructor mirror those of `labels`:

- The *continuation*, with the same meaning and type as the label's continuation.
- The *outer* stack. When a `push` frame is created, the stack is split according to the type of the newly-created label, and the *outer* stack is stored inside in the `push` frame. The *inner* stack is stored in the vs field of the configuration.

- The *outer* instruction sequence. When a `push` frame is created, the inner instruction sequence (the `block` or `loop`'s body) is stored in the *es* field of the configuration, and the *outer* instruction stream is stored inside the `push` frame.

To summarize the difference between `push` frames and `labels`, when a `label` is created, it encapsulates the *inner* state of its label, while the configuration stores the *outer* state of the program. On the other hand, when a `push` frame is created, the *inner* state of the label is stored in the configuration, while the *outer* state is encapsulated inside the `push` frame.

This is where the efficiency of `push` frames comes from: The state of the *innermost* label, which includes the currently-executing instructions, is readily-available in the configuration, while the state of the outer labels is stored away inside the `LabelStack`.

Execution

The only reduction rules that differ between the `Spec` and the `Fast` interpreter are those related to control-flow. Instead of reducing `blocks` and `loops` to `labels`, the `Fast` interpreter converts them into `push` frames:

```
inductive FastStep :
  (c : Context) -> (c' : Context) -> FastConfig c -> FastConfig c' -> Type
where
  | ...
  | fstep_block :
      FastStep
        ⟨locs, lbls, i' ++ i, o⟩ ⟨locs, o  :: lbls, i', o⟩
        ⟨mem, ls, lstk, vs' ++ vs, seq (block es') es⟩
        ⟨mem, ls, push (λ _ => nop) vs es lstk, vs', es'⟩

  | fstep_loop :
      FastStep
        ⟨locs, lbls, i' ++ i, o⟩ ⟨locs, i' :: lbls, i', o⟩
        ⟨mem, ls, lstk, vs' ++ vs, seq (loop es') es⟩
        ⟨mem, ls, push (λ _ => loop es') vs es lstk, vs', es'⟩
```

When a `br` instruction is encountered, it can immediately start "bubbling up" the label stack, without being converted to a `breaking` administrative instruction:

```
inductive FastStep :
  (c : Context) -> (c' : Context) -> FastConfig c -> FastConfig c' -> Type
where
  | ...
  | fstep_br_hit :
      FastStep
        ⟨locs, l :: lbls, l ++ i, o⟩ ⟨locs, lbls, l ++ i', o⟩
        ⟨mem, ls, push k vs' es' lstk, vs, seq (br hit) es⟩
        ⟨mem, ls, lstk, vs.take _ ++ vs', seq (k _) es'⟩

  | fstep_br_miss :
      FastStep
        ⟨locs, l :: lbls, l ++ i, o⟩ ⟨locs, lbls, l ++ i, o⟩
        ⟨mem, ls, push k vs' es' lstk, vs, seq (br (miss ix)) es⟩
        ⟨mem, ls, lstk, vs, seq (br ix) es'⟩
```

The `fstep_br_hit` rule replaces a branch targeting the current label with its continuation, which is popped from the `LabelStack`. The *inner* stack (readily available in the configuration) is appended to the *outer* stack, which was also stored in the just-popped frame.

The `fstep_br_miss` rule simply pops a frame from the `LabelStack` and decrements the index of the `br`. Because the inner stack (which ultimately needs to be sent to the target of the branch) is stored in the configuration, there's no need for a separate `breaking` instruction to capture it.

When the instruction stream es becomes empty, that means the innermost label has executed to completion. If the `LabelStack` is empty, then the configuration has reached a final state. If the `LabelStack` isn't empty, then the next frame is popped and its *outer* instruction sequence is moved into the configuration, and its outer stack is appended to the current inner stack:

```
inductive FastStep :
  (ctx : Context) -> (ctx' : Context) -> FastConfig ctx -> FastConfig ctx' -> Type
where
  | fstep_done_empty :
      FastStep
        ⟨locs, [], i, i⟩ ⟨locs, [], i, i⟩
        ⟨mem, ls, empty, vs, done⟩ ⟨mem, ls, empty, vs, done⟩

  | fstep_done_push :
      FastStep
        ⟨locs, l :: lbls, i, i⟩ ⟨locs, lbls, i ++ i', o⟩
        ⟨mem, ls, push k vs' es lstk, vs, done⟩
        ⟨mem, ls, lstk, vs ++ vs', es⟩
```

The runnable implementations of these reduction rules are implemented similarly in the `FastConfig.eval` function and its proof-carrying variant `FastConfig.eval_proofCarrying`.

3.3.4. Proving correctness of the optimization

Proving that the `LabelStack` optimization is correct involves another soundness proof, showing that for any execution trace produced by the `Fast` interpreter, an equivalent execution trace can be produced by the `Spec` interpreter.

The signature of this proof is most easily expressed using the *relational* versions of the interpreters:

```
def optimization_sound :
  (cfg : FastConfig ctx) -> (cfg' : FastConfig ctx') ->
  FastSteps ctx ctx' cfg cfg' ->
  SpecSteps ctx ctx' cfg cfg'
```

However, the above signature is not quite correct. The types of the configurations differ between the `Spec` and `Fast` interpreters, so a suitable *configuration translation* function is required to properly relate them. Furthermore, because of the different nesting of `labels` and `push` frames, the *contexts* of equivalent configurations also differ, so a context translation function is also required.

Context and configuration translation

Below is the definition for context translation. Similar to the other functions operating on configurations, it makes use of an `unpacked` helper:

```
def FastConfig.toSpecContext_unpacked
  (lstk : LabelStack locs lbls o) (vs : Stack i) (es : Instrs spec locs lbls i o)
  : Context
:=
  match lstk with
  | empty => ⟨locs, lbls, i, o⟩
  | push k vs' es' lstk =>
      FastConfig.toSpecContext_unpacked
        lstk vs' (seq (label (λ b => (k b).toSpec) vs es.toSpec) es'.toSpec)
```

```
def FastConfig.toSpecContext (cfg : FastConfig ctx) -> Context :=
  match cfg with
  | ⟨_, _, lstk, vs, es⟩ =>
      FastConfig.toSpecContext_unpacked lstk vs es.toSpec
```

Given a `FastConfig`, the function `toSpecContext` computes the context type of an equivalent `SpecConfig`. To achieve this, it recurs over the `LabelStack`, converting it into a series of nested `labels`. Finally, when the `LabelStack` is empty, which means all `push` frames have been transformed into `labels`, it simply returns the type of the current instruction stream.

The configuration translation function uses the same recursion pattern, but returning the translated configuration itself rather than just its type:

```
def FastConfig.toSpec_unpacked
  (mem : Memory) (ls : Stack locs) (lstk : LabelStack locs lbls o)
  (vs : Stack i) (es : Instrs spec locs lbls i o)
  : SpecConfig (FastConfig.toSpecContext_unpacked lstk vs es)
:=
  match lbls, lstk with
  | _, empty => ⟨mem, ls, vs, es⟩
  | _ :: _, push k vs' es' lstk =>
      FastConfig.toSpec_unpacked
        mem ls lstk vs' (seq (label (λ b => (k b).toSpec) vs es.toSpec) es'.toSpec)

def FastConfig.toSpec (cfg : FastConfig ctx) : SpecConfig cfg.toSpecContext :=
  match cfg with
  | ⟨mem, ls, lstk, vs, es⟩ => FastConfig.toSpec_unpacked mem ls lstk vs es.toSpec
```

Using these translation functions, we can now give a type-correct signature for the soundness proof:

```
def optimization_sound :
  (cfg : FastConfig ctx) -> (cfg' : FastConfig ctx') ->
  FastSteps ctx ctx' cfg cfg' ->
  SpecSteps cfg.toSpecContext cfg'.toSpecContext cfg.toSpec cfg'.toSpec
```

Similar to the previous soundness proofs, the optimization proof can be divided into a proof relating the small-step semantics, and a secondary proof that extends to execution traces. However, unlike the proofs showing equivalence between the relational and functional interpreters, the small-step proof needs to relate a single step of evaluation in the `Fast` interpreter to *multiple* steps in the `Spec` interpreter. This is because some instructions, notably `br`, reduce directly in the `Fast` interpreter, while the `Spec` interpreter first converts them into administrative instructions.

Unfortunately, despite multiple different attempts, due to the complexity of the types involved, we were unable to provide an implementation of the proof.

4. Related work

The official WebAssembly specification includes a reference interpreter [12] implemented in OCaml. This interpreter keeps a close correspondence to Wasm's semantics, differing only in its representation of the stack and of label contexts. However, its implementation of control-flow leads to severe performance issues, especially when executing programs compiled from languages that support irreducible control flow [21].

Watt et al. have implemented WasmCert [23], a pair of two mechanizations of the WebAssembly semantics, in the Coq and Isabelle [24] theorem provers. The mechanization includes an extrinsic mechanization of the Wasm type system. They also provide executable verified interpreters extracted

from the mechanizations. Like the reference interpreter, these mechanizations exhibit the same efficiency issues.

WasmRef-Isabelle [7] is a verified WebAssembly interpreter written in Isabelle and proven correct with respect to the WasmCert-Isabelle mechanization. WasmRef also uses an optimized representation of control-flow, and is performant enough to be used as part of the fuzzing infrastructure of Wasmtime [25], a widely-used WebAssembly implementation. However, due to Isabelle's lack of dependent types, WasmRef-Isabelle's encoding of the WebAssembly syntax is extrinsically-typed, which limits the authors to proving *partial* correctness of the interpreter [7, section 4.3.2].

Titzer has implemented Wizard [5], an *in-place* WebAssembly interpreter with a strong focus on performance. While this implementation addresses the performance issues of the reference interpreter and employs a number of innovative optimizations, it would be difficult to formally verify, as it is not implemented in a proof assistant language.

5. Limitations and future work

The current implementation of LeanWasm only supports a subset of the WebAssembly language. Some features were omitted due to the difficulty of encoding them in Lean (such as SIMD instructions and single-precision floating point numbers), while other features were omitted due to time constraints (function calls, modules, module instantiation) and could be implemented in future work.

There is currently a great amount of code duplication between the naive `Spec` interpreter and the optimized `Fast` interpreter, as most of the reduction rules are identical between them. The techniques introduced by van der Rest et al. for defining intrinsically-typed interpreters à la carte [16] could be used to separate the interpreters into composable *language fragments*, with one fragment defining the common operations, and distinct fragments defining each interpreter's handling of control-flow.

Another interesting avenue to explore is measuring the performance impact of the intrinsic typing machinery by performing benchmarks against an untyped version of the LeanWasm interpreter. While the intrinsically-typed syntax adds some overhead in the form of runtime-retained type indices, it also eliminates the need for value tagging and for validity checks during execution.

In the same vein, more optimizations could be explored, such as using more efficient heterogenous data structures [26] instead of `HLists` for value stacks and local variables.

Lastly, it would be valuable to explore intrinsically-typed encodings for extensions to the WebAssembly type system, such as SecWasm [27] (for information-flow control) or CT-wasm [28] (for constant-time cryptography).

6. Conclusion

In this thesis, we have described an intrinsically-typed encoding of the WebAssembly abstract syntax, and implemented two interpreters for this syntax. The first interpreter closely follows the official reference interpreter, which causes it to handle control-flow in an inefficient way. The second interpreter uses an optimized representation of control-flow to alleviate this issue.

Each interpreter is defined in both *relational* and *functional* styles, with soundness proofs attesting the equivalence between the different versions. We have also discussed a soundness proof verifying the correctness of the control-flow optimizations, and provided functions for translating typing contexts and runtime configurations between the two interpreters.

Bibliography

[1] A. Haas *et al.*, "Bringing the web up to speed with WebAssembly," *SIGPLAN Not.*, vol. 52, no. 6, pp. 185–200, Jun. 2017, doi: 10.1145/3140587.3062363.

[2] "WebAssembly Specification." [Online]. Available: https://webassembly.github.io/spec/core/

[3] "Wasmtime." [Online]. Available: https://wasmtime.dev/

[4] "Wasmer: The Universal WebAssembly Runtime." [Online]. Available: https://wasmer.io/

[5] B. L. Titzer, "A fast in-place interpreter for WebAssembly," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, Oct. 2022, doi: 10.1145/3563311.

[6] WebAssembly Community Group, "WebAssembly Test Suite." [Online]. Available: https://github.com/WebAssembly/testsuite

[7] C. Watt, M. Trela, P. Lammich, and F. Märkl, "WasmRef-Isabelle: A Verified Monadic Interpreter and Industrial Fuzzing Oracle for WebAssembly," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Jun. 2023, doi: 10.1145/3591224.

[8] C. Bach Poulsen, A. Rouvoet, A. Tolmach, R. Krebbers, and E. Visser, "Intrinsically-typed definitional interpreters for imperative languages," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017, doi: 10.1145/3158104.

[9] "The Lean Programming Language and Theorem Prover." [Online]. Available: https://lean-lang.org/

[10] "aionescu/lean-wasm: An intrinsically-typed interpreter for WebAssembly." [Online]. Available: https://github.com/aionescu/lean-wasm/

[11] N. de Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem," *Indagationes Mathematicae (Proceedings)*, vol. 75, no. 5, pp. 381–392, 1972, doi: https://doi.org/10.1016/1385-7258(72)90034-0.

[12] WebAssembly Community Group, "WebAssembly Reference Interpreter." [Online]. Available: https://github.com/WebAssembly/spec/tree/main/interpreter

[13] "The OCaml Programming Language." [Online]. Available: https://ocaml.org/

[14] "The Agda Wiki." [Online]. Available: https://wiki.portal.chalmers.se/agda/pmwiki.php

[15] "The Coq Proof Assistant." [Online]. Available: https://coq.inria.fr/

[16] C. van der Rest, C. B. Poulsen, A. Rouvoet, E. Visser, and P. Mosses, "Intrinsically-typed definitional interpreters à la carte," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, Oct. 2022, doi: 10.1145/3563355.

[17] David Thrane Christiansen, "The Universe Design Pattern - Functional Programming in Lean." [Online]. Available: https://lean-lang.org/functional_programming_in_lean/dependent-types/universe-pattern.html

[18] "Coercions using Type Classes - Theorem Proving in Lean 4." [Online]. Available: https://lean-lang.org/theorem_proving_in_lean4/type_classes.html#coercions-using-type-classes

[19] S. Ullrich and L. de Moura, "Counting immutable beans: reference counting optimized for purely functional programming," in *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*, in IFL '19. Singapore, Singapore: Association for Computing Machinery, 2021. doi: 10.1145/3412932.3412935.

[20] "Inaccessible Patterns - Theorem Proving in Lean 4." [Online]. Available: https://lean-lang.org/theorem_proving_in_lean4/induction_and_recursion.html#inaccessible-patterns

[21] Yuri Iozzelli, "Solving the structured control flow problem once and for all." [Online]. Available: https://medium.com/leaningtech/solving-the-structured-control-flow-problem-once-and-for-all-5123117b1ee2

[22] G. HUET, "The Zipper," *Journal of Functional Programming*, vol. 7, no. 5, pp. 549–554, 1997, doi: 10.1017/S0956796897002864.

[23] C. Watt, X. Rao, J. Pichon-Pharabod, M. Bodin, and P. Gardner, "Two Mechanisations of WebAssembly 1.0," in *Formal Methods*, M. Huisman, C. Păsăreanu, and N. Zhan, Eds., Cham: Springer International Publishing, 2021, pp. 61–79.

[24] "Isabelle." [Online]. Available: https://isabelle.in.tum.de/

[25] Bytecode Alliance, "A fast and secure runtime for WebAssembly." [Online]. Available: https://github.com/bytecodealliance/wasmtime

[26] W. Swierstra, "Heterogeneous binary random-access lists," *Journal of Functional Programming*, vol. 30, p. , 2020, doi: 10.1017/S0956796820000064.

[27] I. Bastys, M. Algehed, A. Sjösten, and A. Sabelfeld, "SecWasm: Information Flow Control for WebAssembly," in *Static Analysis: 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5–7, 2022, Proceedings*, Auckland, New Zealand: Springer-Verlag, 2022, pp. 74–103. doi: 10.1007/978-3-031-22308-2_5.

[28] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "CT-wasm: type-driven secure cryptography for the web ecosystem," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019, doi: 10.1145/3290390.

A Organization of the source code

The full source code for LeanWasm is available on GitHub [10]. The module hierarchy of the source code is organized in the following manner:

- `LeanWasm.HList`: Contains the definition of heterogenous lists (`HList`) and well-typed list indices (`Ix`), as well as related operations.
- `LeanWasm.Syntax.Ty`: Contains the definitions of WebAssembly value types (`Ty`) and stack types, and associated coercions and pattern synonyms.
- `LeanWasm.Syntax.Instr`: Contains the definitions of intrinsically-typed WebAssembly instructions (`Instr`), instruction sequences (`Instrs`), and arithmetic operators (`IBinOp`, `IRelOp` etc.).
- `LeanWasm.Eval.Common`: Defines typing contexts (`Context`) and helper functions used in the execution of Wasm programs, such as memory operations.
- `LeanWasm.Eval.Spec`: Defines the unoptimized `Spec` evaluator, including its configurations (`SpecConfig`), relational semantics (`SpecStep`, `SpecSteps`), and executable evaluation functions (`SpecConfig.eval`, `SpecConfig.evals`) along with proof-carrying variants (`SpecConfig.eval_proofCarrying`, `SpecConfig.evals_proofCarrying`).
- `LeanWasm.Eval.Fast`: Defines the unoptimized `Fast` evaluator, including its configurations (`FastConfig`), relational semantics (`FastStep`, `FastSteps`), and executable evaluation functions (`FastConfig.eval`, `FastConfig.evals`) along with proof-carrying variants (`FastConfig.eval_proofCarrying`, `FastConfig.evals_proofCarrying`).
- `LeanWasm.Examples`: Contains a few example Wasm programs (expressed in the intrinsically-typed abstract syntax tree) that were used to test the interpreters.
- `Main`: Defines a small harness that runs the example programs.