Computer Science
COC251
F016319

WebAssembly Target Implementation for Gleam

by

Danielle O. Maywood

Supervisor: Dr J. Day

Department of Computer Science
Loughborough University

May/June 2024

Table of Contents

1. Abstract

Gleam is a new type-safe functional programming language with a purposefully minimal set of features whilst being maximally expressive. WebAssembly is a new binary-code format that aims to be a portable compilation target to enable high-performance execution on web and non-web targets. Gleam currently supports compiling to both JavaScript and Erlang, however these targets are both dynamically typed. This means compiled Gleam loses important type information that could be used to optimise the runtime performance. In this project we aim to extend the Gleam compiler to be able to compile Gleam code to WebAssembly. I used Rust to implement this feature, as the Gleam compiler is written in Rust. By breaking down Gleam into a list of its features (such as custom types and BitArrays), and then determining which of these were most depended upon, the features could then be sorted by complexity. This allowed implementing each feature at a time, and testing that they were implemented correctly. To ensure previously implemented features did not regress in functionality, a test-suite was created and regularly ran to ensure features were not broken. Unfortunately, due to time constraints, the BitArray feature did not get implemented. When comparing the runtime performance of the generated WebAssembly, we see that it can compete with the runtime performance of the generated Erlang and JavaScript for the same programs. Future work could aim at improving how generic code is generated as a few bugs exist with the current implementation.

2. Keywords

***AST* – Abstract Syntax Tree**: A representation of the syntax of a programming language in a tree data structure.

***TCO* – Tail Call Optimisation**: An optimisation technique used to optimise tail recursive calls. Rather than create a new stack frame for the tail recursive call, it replaces the existing one. This technique is used to allow tail recursive functions to not exhaust the stack.

***WASI* – WebAssembly System Interface**: A set of API's to provide a standardised way of allowing WebAssembly modules to interact with their host environment.

***Wasm* – WebAssembly**.

3. Introduction

3.1. Gleam

Gleam [1] is a statically-typed functional programming language that compiles to both Erlang [2] and JavaScript [3], [4].

3.1.1. Type system

Gleam uses the Hindley-Milner [5], [6] type system.

The Hindley-Milner type system can perform type inference without requiring annotating variables with an explicit type annotation. An example of this type inference can be seen in Listing 1, where a function is defined without specifying the parameter's types or the function's return type. The type inference engine is able to infer that the parameters `lhs` and `rhs` must be of type `Int` as the `+` operator takes two operands of type `Int`. The engine is also able to infer that the return type of the function must be `Int` as this is the type produced by the `+` operator.

Listing 1: Add function without type annotations

```gleam
fn add(lhs, rhs) {
  lhs + rhs
}
```

The type inference algorithm uses the first instance where a variable's type can be inferred to decide what type it is. In Listing 2 the type inference algorithm reports a type mismatch as `lhs + rhs` expects `lhs` and `rhs` to be of type `Int`, however, they have been inferred to be of type `Float` due to the previous expression.

Listing 2: Add function type inference error

```gleam
fn add(lhs, rhs) {
  let _ = lhs +. rhs
  let _ = lhs + rhs
}
```

3.1.2. Data types

3.1.2.1. Nil

`Nil` is Gleam's unit type, it is intended to be used by functions that do not return anything. An example of this is Gleam's standard library's `io.println` function, it has a return type of `Nil` as there is nothing to return.

3.1.2.2. Integer

The behaviour of integers in Gleam is dependent on whether it is compiled to Erlang or JavaScript [7]. When targetting Erlang, integers have no minimum or maximum size [8]. When targetting JavaScript, however, integers are represented using the double-precision IEEE 754 format [9], [10], giving them a range of $-2^{53} + 1$ to $2^{53} - 1$ before losing precision.

3.1.2.3. Float

The behaviour of floats in Gleam is also dependent on whether it is compiled to Erlang or JavaScript [11]. Both targets represent floats using the double-precision IEEE 754 format [12], [10], however Erlang raises an error if an operation results in `NaN`, `+Inf` or `-Inf`.

3.1.2.4. Bool

Gleam's `Bool` type is represented as a custom type with the definition given in Listing 3.

Listing 3: Gleam's Bool data type

```gleam
type Bool {
  True
  False
}
```

3.1.2.5. String

The behaviour of Gleam's `String` type is not runtime dependent, Gleam's `String` type uses Erlang's `BitString` type and JavaScript's `String` type.

3.1.2.6. BitArray

Gleam's `BitArray` type represents an array of bits, meaning an array of values that are either 0 or 1. They are fully supported on the Erlang target, however have limited support on the JavaScript target. The limitation on the JavaScript target is not a technical limitation, and it is intended to be fully supported in the future.

3.1.2.7. Lists

Gleam's `List` data type uses an immutable singly linked list representation [13]. On the Erlang target, it uses the Erlang's existing list data type [14] as this is an immutable singly linked list. JavaScript's `Array` type has no specified representation [15], which means a custom list implementation had to be made [16] to have similar performance characteristics across both targets.

3.1.2.8. Tuples

Gleam's tuple data type uses Erlang's tuple data type [17], [18], and JavaScript's `Array` data type [19], [20]. They are constructed with the syntax shown in Listing 4.

Listing 4: Gleam tuple syntax

```gleam
// A tuple containing 3 integers
#(1, 2, 3)

// A tuple containing an integer, float, string and bool.
#(1, 2.5, "4", False)
```

3.1.2.9. Custom types

Gleam's custom types are created with Erlang's `record` construct [21], [22], and JavaScript's `class` construct [23], [24].

A custom type can be opaque, meaning that it cannot be constructed from outside the type's module. It is also not possible to access the fields of an opaque type from outside of the type's module.

Listing 5: An opaque type

```gleam
opaque type MyOpaqueType {
  MyOpaqueType(a: Int, b: Int)
}
```

Custom types support having multiple variants as shown in Listing 6. Accessing fields from types is only possible on fields that share the same name, type and position on all variants. To access a field that is not shared, you must use pattern matching to figure out which variant the type is.

Listing 6: Example of custom type with multiple variants

```gleam
type MyTypeWithMultipleVariants {
  VariantA(a: Int)
  VariantB(a: Int, b: Int)
  VariantC(a: Int, b: Int, c: Int)
}
```

3.1.3. Expressions

Gleam is an expression-based language, meaning that everything is an expression and returns a value. An example of this is the `let` expression, it returns the value bound to the given name.

Listing 7: Example of let binding being an expression

```gleam
let x = { let y = 5 }
// `x` has the value `5`.
```

3.1.3.1. Assignments

In Gleam, you can bind a value to a name with the `let` keyword to allow referring to a value at a later point, as shown in Listing 8.

Listing 8: Let binding

```gleam
let x = 5
let y = x * x
```

Gleam does not support mutability but you can bind a value to a name currently in scope as shown in Listing 9.

Listing 9: Let binding

```gleam
let x = 5
let x = x * x
```

3.1.3.2. Numeric operations

Gleam has two sets of binary operators for numerical values. One set of these operators works on integer operands, with the other set operating on floating-point operands.

Operation	**Int**	**Float**
Addition	`+`	`+.`
Subtraction	`-`	`-.`
Division	`/`	`/.`
Multiplication	`*`	`*.`
Greater than	`>`	`>.`
Greater than or equal to	`>=`	`>=.`
Less than	`<`	`<.`
Less than or equal to	`<`	`<=.`

3.1.3.3. Pipelines

The operator `|>` is called the pipe operator. It takes the left hand side operand and passes it as the first argument to the function on the right hand side. Listing 10 demonstrates example usage of this feature.

Listing 10: Pipeline example

```gleam
fn add(lhs: Int, rhs: Int) -> Int { lhs + rhs }

5 |> add(4) |> add(3)

// The code above is equivalent to below
add(add(5, 4), 3)
```

3.1.3.4. Anonymous functions

Gleam allows the creation of anonymous functions. Anonymous functions are able to capture variables that are in scope at their creation. This means that they are unable to call themselves as there is no reference to the function available at the time it is created. The captured variables also outlive the lexical scope they are defined in, as shown in Listing 11.

Listing 11: Anonymous function capturing example

```gleam
fn make_adder(lhs: Int) -> fn(Int) -> Int {
  fn(rhs) { lhs + rhs }
}

fn main() {
  let add_5 = make_adder(5)
  let result = add_5(3)

  // 'result' has the value '3'
}
```

3.1.3.5. Function captures

Gleam has a syntactic sugar for creating an anonymous function with a single argument. This feature is often used in combination with the pipeline feature in order to pipe into a different argument than the first one.

Listing 12: Function capture example

```gleam
// The following lines are equivalent
let x = fn(rhs) { add(5, rhs) }
let x = add(5, _)

// Example use in pipeline
let z =
  5
  |> add(4)
  |> sub(10, _)
```

3.1.3.6. Case expressions

Gleam's `case` expression is for expressing conditional logic. It performs pattern matching to decide which branch should be executed, as well as performing an exhaustiveness check to ensure all possible patterns are covered. Given the example case expression in Listing 13, the exhaustiveness checker detects that the `case` expression has no pattern that matches the `Pig` variant of the `Animal` type.

Listing 13: Case expression exhaustiveness

```gleam
type Animal {
  Cat
  Dog
  Pig
}

fn get_animal_name(animal: Animal) -> String {
  case animal {
    Cat -> "Cat"
    Dog -> "Dog"
  }
}
```

The `case` expression allows adding guards to a branch as shown in Listing 14. Due to limitations with what operations are supported by Erlang in guards, Gleam only allows a limited subset of expressions to be included in a guard.

Listing 14: Guard in case expression

```gleam
fn get_generation(year_born: Int) -> Option(String) {
  case year_born {
    _ if year_born >= 2013 -> Some("Generation Alpha")
    _ if year_born >= 1997 -> Some("Generation Z")
    _ if year_born >= 1981 -> Some("Millennials")
    _ if year_born >= 1965 -> Some("Generation X")
    _ if year_born >= 1946 -> Some("Baby Boomers")
    _ if year_born >= 1928 -> Some("Silent Generation")
    _ if year_born >= 1901 -> Some("Greatest Generation")
    _ if year_born >= 1883 -> Some("Lost Generation")
    _ -> None
  }
}
```

3.1.3.7. Use expressions

Gleam's `use` expression allows reducing how indented callback-oriented code is. It is syntax sugar for passing a callback to the last parameter of the called function as shown in Listing 15.

Listing 15: Example use expression

```gleam
import gleam/result.{try}
import iox/stdin

pub fn get_with_use() -> Result(User, Nil) {
  use username <- result.try(stdin.read_line())
  use password <- result.map(stdin.read_line())

  User(username, password)
}

pub fn get_without_use() -> Result(User, Nil) {
  result.try(readline.read(), fn(username) {
    result.map(readline.read(), fn(password) {
      User(username, password)
    })
  })
}
```

3.2. Rust

As the Gleam compiler is written in Rust, we will need to use it to write the Wasm (*WebAssembly*) target implementation. Rust is a language with an aim at allowing developers to write high performance and memory efficient code whilst guaranteeing memory-safety.

In the following section we will briefly cover the aspects of Rust that will be used in this report.

3.2.1. Enums

Enums in Rust are types that can contain zero or more variants. The variants can contain data, although do not need to. Listing 16 demonstrates all the different kinds of variants.

Listing 16: Example enum in Rust

```rust
enum Example {
  WithoutData,
  WithSimple(i64),
  WithMultiple(String, i64),
  WithFields {
    foo: String,
    bar: i64,
  }
}
```

3.2.2. Generics

We can have generic functions and data-types in Rust. For both functions and types we can specify the list of generics directly after the name as shown in Listing 17.

Listing 17: Example enum in Rust

```rust
enum SomeType<A, B, C> {}

fn do_something<A, B, C>() {}
```

3.2.3. The `Result<T, E>` type

In Rust if we want to indicate to a consumer that our function can fail, we can use the `Result<T, E>` type, where `T` is the type of the value for if the function succeeds and `E` is the error type for if the function fails.

The `Result<T, E>` type is defined as shown in Listing 18. As we can see it is an enum with two variants: the `Ok(T)` variant and the `Err(E)` variant. Unlike a typical enum, the `Ok` and `Err` constructors can be called without having to write `Result::Ok` or `Result::Err`.

Listing 18: `Result<T, E> definition`

```rust
enum Result<T, E> {
  Ok(T),
  Err(E),
}
```

Rust

3.2.4. Lists

You can define lists in Rust by surrounding a comma-separated list of expressions with square brackets, with each expression having the same type. An example of this can be seen in Listing 19.

Listing 19: Example list in Rust

```rust
[1, 2, 3, 4, 5]
```

Rust

3.2.5. Tuples

You can define a tuple in Rust by surrounding a comma-separated list of expressions with parenthesis, where each element of the tuple can be a different type. An example of this can be seen in Listing 20.

Listing 20: Example tuple in Rust

```rust
("Foo", 123, 4.56, true)
```

Rust

3.3. WebAssembly

Wasm is a binary instruction format intended to be executed by a stack-based virtual machine in both web and non-web targets. One of the main goals of Wasm is to enable the use of high-performance applications on the web for situations where JavaScript is insufficient.

Example use cases for Wasm are as follows:
- Running code in languages other than JavaScript on Cloudflare's Workers. [25]
- Writing web applications in Rust with Leptos. [26]
- Running the chess engine Stockfish in the browser in Lichess. [27]
- Running games written in Unity in the browser. [28]
- Writing plugins in any language for Zellij. [29]

3.3.1. Advantages

When running CPU intensive tasks in the browser, Wasm can offer better runtime performance than JavaScript. According to a study into the energy efficiency and runtime performance of JavaScript and WebAssembly, it was seen that when running the same application in both JavaScript and Wasm, the Wasm version had better runtime performance and energy consumption [30].

Wasm has been designed with a focus on security. The Wasm runtime runs Wasm inside of a sandboxed execution environment, and all attempts to interact with the host system pass through the Wasm runtime. This design allows limiting the access software running in Wasm has to cause harm to the host environment.

3.3.2. Disadvantages

A disadvantage of Wasm for web applications is that it currently does not have access to directly manipulate the contents of a web page. The current approach is to call JavaScript code from Wasm to handle this use case, however this has a performance overhead that can exceed the runtime performance benefits from using Wasm.

Another disadvantage of Wasm for web applications is the file size of the Wasm binaries. When compiling languages that have garbage collectors into Wasm, they have to package their garbage collector into the binary. This can result in binaries that are larger than the equivalent JavaScript code, which can result in web pages taking more time to load.

3.3.3. Text Format

Wasm also supports a text-based format [31] to assist developers when testing, learning and experimenting with Wasm. The text format is a representation of a Wasm module in an S-expr format.

3.3.3.1. A minimal module

A minimal module in WebAssembly contains no types, data, or functions.

Listing 21: A minimal WebAssembly module

WebAssembly
```wasm
(module)
```

3.3.3.2. Defining a function

Defining a function that takes two integers and returns their sum can be defined as shown in Listing 22. We define a function that can be referenced with the identifier `$add`. It takes two parameters of type `i32` (a 32-bit integer) with identifiers `$x` and `$y`, and returns a single value of type `i32`. The instruction `local.get` takes an index of a local, and pushes the local's value onto the stack. The text format allows us to also pass an identifier instead of an integer index. The instruction `i32.add` pops the top two values off the stack, adds them together, and pushes the resulting value back onto the stack. This final value on the stack is treated as the return value.

Listing 22: Add function defined in WebAssembly

```
WebAssembly
(module
  (func $add (param $x i32) (param $y i32) (result i32)
    (i32.add (local.get $x) (local.get $y))
  )
)
```

3.3.3.3. If expressions

Wasm provides the ability to perform conditional execution with the `if` control instruction. The instruction takes an optional block type that specifies what value will be left on the stack after it has been executed. The instruction also takes a 'then' branch with an optional 'else' branch.

When the `if` expression executes it follows the following steps:

1. Ensure a value of type `i32` is on the top of the stack
2. Pop the value *c* from the top of the stack
3. If *c* is a non-zero value:
   1. Execute the `then` branch
4. Otherwise:
   1. Execute the `else` branch

Listing 23: The structure of an if/them/else expression

```
WebAssembly
(; condition ;)
(if (; block type ;)
  (then (; then branch ;))
  (else (; else branch ;))
)
```

Using this we can create a function that computes the absolute value of a number as shown in Listing 24.

Listing 24: Function to calculate the absolute value of a number

```wasm
(module
  (func $abs (param $x i64) (result i64)
    (i64.lt_s (local.get $x) (i64.const 0))
    (if (result i64)
      (then (i64.sub (i64.const 0) (local.get $x)))
      (else (local.get $x))
    )
  )
)
```

3.3.4. Proposals

The implementation discussed in Section 4 makes use of the Typed Function References [32] and Garbage Collection [33] proposals. These proposals are both in Phase 4 [34], [35] of the standardisation process. This means two or more Web VMs have correctly implemented the proposal, at least once toolchain has implemented it, and that consensus has been reached both in support of the proposal and that the specification has been complete.

3.3.4.1. Typed Function References

This proposal adds the ability to create a typed reference to a function. This allows generating type-safe Wasm that can pass around a reference to a function. The instructions that will be useful for our implementation from this proposal are `ref.func` and `call_ref`.

The `ref.func` instruction takes a function index and pushes onto the stack the reference to that function. It is important to note that the function index is a value provided when the Wasm binary is created, not a runtime value from the stack.

The `cell_ref` instruction takes a function type index that corresponds with the function ref being called. At the top of the stack it expects the function reference (from `ref.func`), as well as a list of arguments.

3.3.4.2. Garbage collection

This proposal adds a garbage collector to the WebAssembly virtual machine. The proposal is intended to reduce the binary size of wasm modules as languages can leverage the builtin garbage collector instead of bundling their own garbage collector in the wasm module. To make use of the garbage collector, it allows the creation of heap allocated struct and array types with the `struct.new` and `array.new` instructions. Accessing values from these types is possible with the `struct.get` and `array.get` instructions.

4. Implementation

4.1. Gleam compiler structure

Figure 1 shows the current structure of the Gleam compiler. The compiler reads the entire source file to memory, and constructs a lexer that references this source. A parser is then constructed with a reference to the lexer. The parser requests tokens from the lexer as it progresses, rather than lexing the entire file at once. The output produced by the parser is an Untyped AST (*Abstract Syntax Tree*), this data structure is a representation of the structure of the Gleam code. This Untyped AST is then type-checked to produce an AST with type information added. At this stage, the appropriate code generator is chosen to produce either Erlang or JavaScript source. To add the Wasm target, the only phase requiring a change is the final step as Figure 2 shows.

Figure 1: Gleam compiler's current structure.

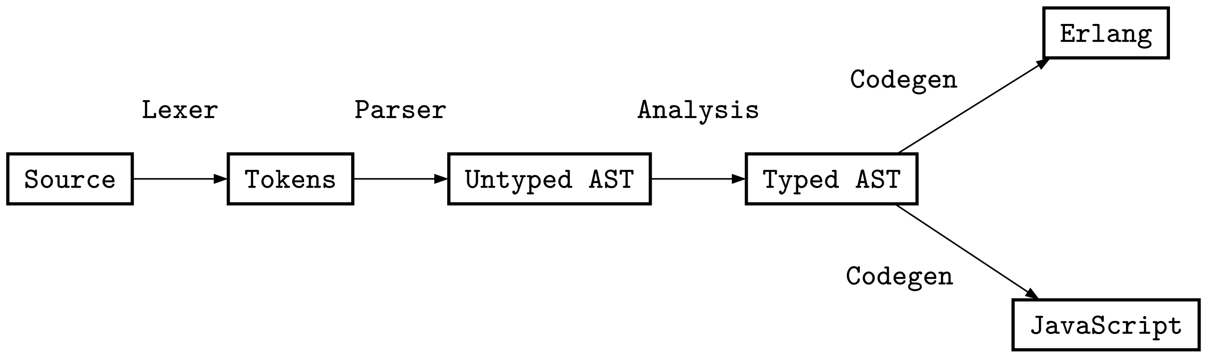
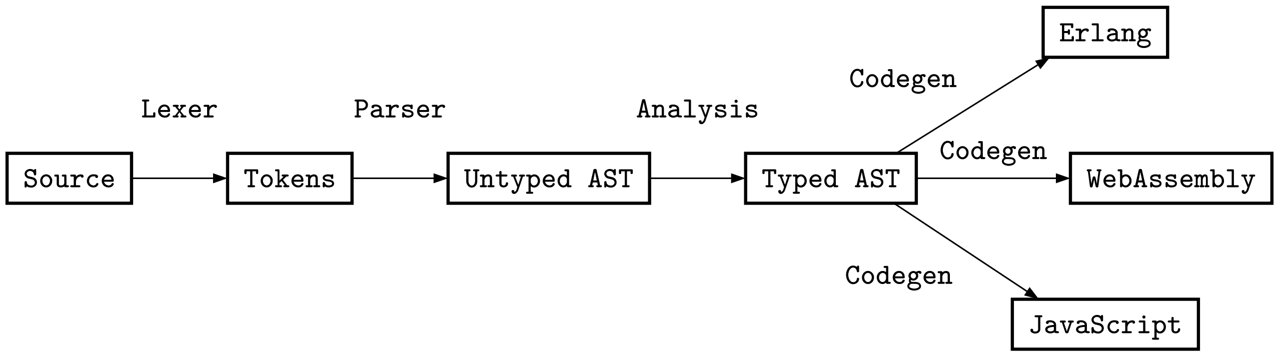


Figure 2: Gleam compiler's current structure with WebAssembly target added.



4.1.1. The lexing phase

The first phase of the Gleam compiler is the lexer. In this phase, the compiler reads the program's source code and converts it into a stream of tokens for use by the parser. Each token contains the token's position in the source file and what the token represents. The purpose of the keeping the token's position in the source file is for use in reporting compile errors to developers writing Gleam programs. By keeping track of where tokens are in the source file, it allows reporting where the source code that caused the error has occurred.

Listing 25: Example add function

```gleam
pub fn add(lhs, rhs) {
  lhs + rhs
}
```

Listing 26 shows the output from the lexer for the example Gleam code written in Listing 25, where each token is represented by a tuple containing: an index in the source file where the start of the token is; the kind of token it is; and an index in the source file immediately after the token.

Listing 26: Lexer output for Listing 25

```rust
[
  Ok((0, Token::Pub, 3)),
  Ok((4, Token::Fn, 6)),
  Ok((7, Token::Name { name: "add" }, 10)),
  Ok((10, Token::LeftParen, 11)),
  Ok((11, Token::Name { name: "lhs" }, 14)),
  Ok((14, Token::Comma, 15)),
  Ok((16, Token::Name { name: "rhs" }, 19)),
  Ok((19, Token::RightParen, 20)),
  Ok((21, Token::LeftBrace, 22)),
  Ok((25, Token::Name { name: "lhs" }, 28)),
  Ok((29, Token::Plus, 30)),
  Ok((31, Token::Name { name: "rhs" }, 34)),
  Ok((35, Token::RightBrace, 36)),
]
```

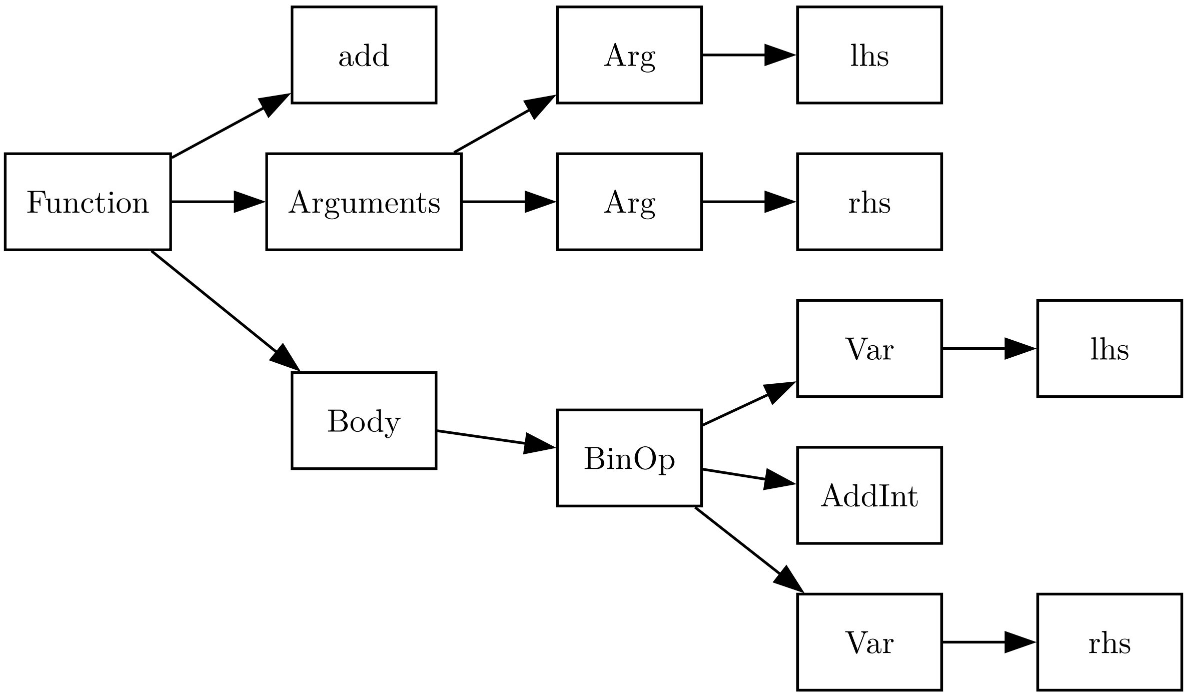
4.1.2. The parsing phase

The second phase of the Gleam compiler is the parser. In this phase, the compiler takes the stream of tokens produced by the lexer and converts it into an untyped AST. The untyped AST produced is a representation of the structure of the program, prior to any analysis as to whether the program is valid or not.

Listing 27: Untyped AST produced when parsing Listing 26

```rust
Function {
  location: SrcSpan { start: 0, end: 20 },
  name: "add",
  arguments: [
    Arg {
      names: Named { name: "lhs" },
      location: SrcSpan { start: 11, end: 14 },
      type_: (),
    },
    Arg {
      names: Named { name: "rhs" },
      location: SrcSpan { start: 16, end: 19 },
      type_: (),
      },
    ],
    body: [Expression(BinOp {
      location: SrcSpan { start: 25, end: 34 },
      name: AddInt,
      left: Var {
        location: SrcSpan { start: 25, end: 28 },
        name: "lhs",
      },
      right: Var {
        location: SrcSpan { start: 31, end: 34 },
        name: "rhs",
      },
    })],
    return_type: (),
  }
```

Figure 3: Listing 27 visualised as a graph



4.1.3. The analysis phase

The third phase of the Gleam compiler is the analysis phase. In this phase, the compiler performs multiple analysis tasks to ensure that the source code is a valid Gleam program and then produces a typed AST. This typed AST contains the types of each definition and expression, for use in the code generation phase.

This stage performs the following analysis:

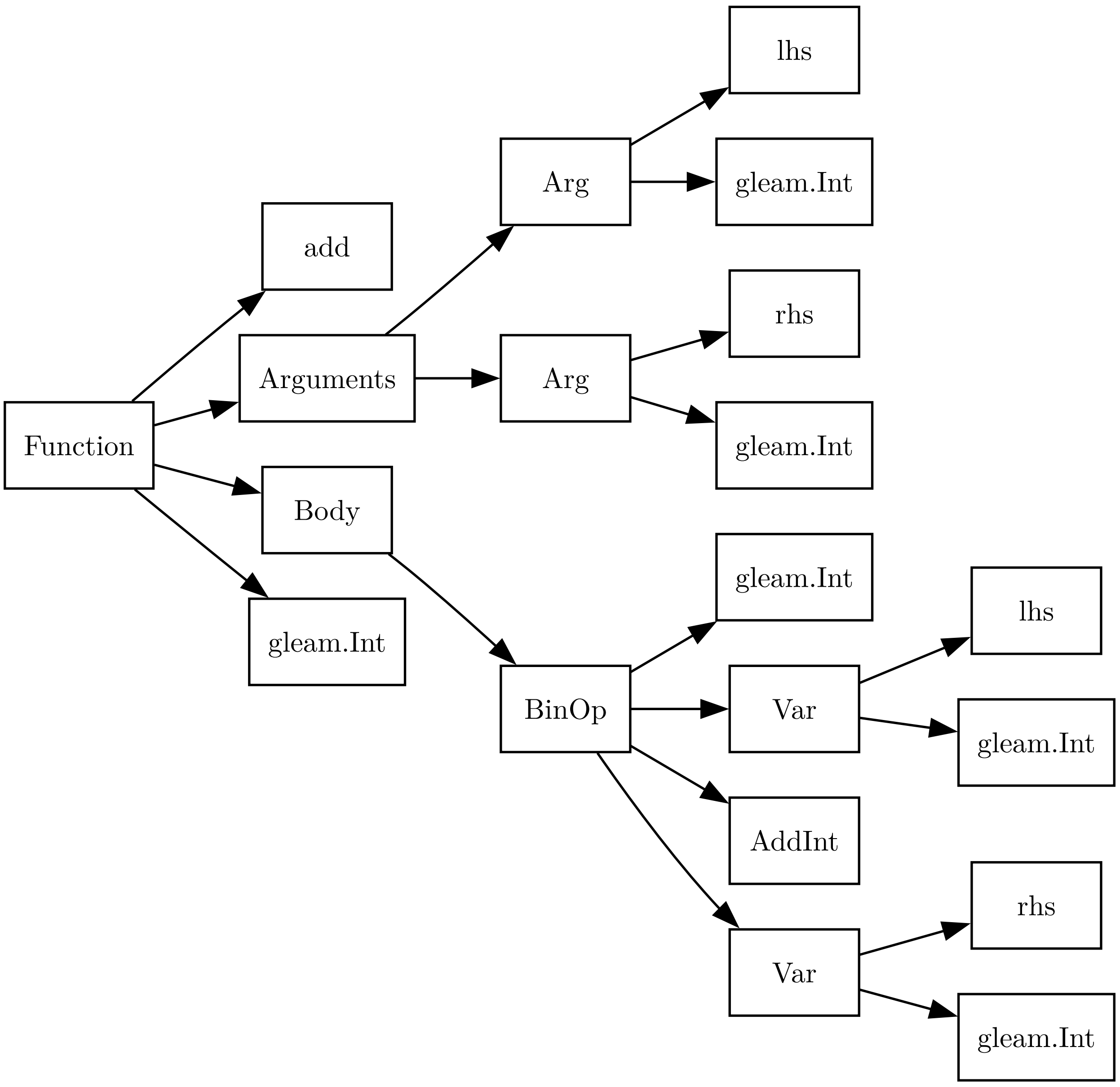
- Performs type-checking.
- Generates warnings for unused variables.
- Ensures exported values do not expose private types in their type signature.

Listing 28: Typed AST produced after analysis of Listing 27

Rust

```
Function {
  name: "add",
  arguments: [
    Arg {
      names: Named { name: "lhs" },
      type_: Var { type_: RefCell { value: Link {
        type_: Named { module: "gleam", name: "Int" }
      } } },
    },
    Arg {
      names: Named { name: "rhs" },
      type_: Var { type_: RefCell { value: Link {
        type_: Named { module: "gleam", name: "Int" }
      } } },
    },
  ],
  body: [Expression(BinOp {
    typ: Named { module: "gleam", name: "Int" },
    name: AddInt,
    left: Var {
      constructor: ValueConstructor {
        variant: LocalVariable { },
        type_: Var { type_: RefCell { value: Link {
          type_: Named { module: "gleam", name: "Int" }
        } } },
      },
      name: "lhs",
    },
    right: Var {
      constructor: ValueConstructor {
        variant: LocalVariable { },
        type_: Var { type_: RefCell { value: Link {
          type_: Named { module: "gleam", name: "Int" }
        } } },
      },
      name: "rhs",
    },
  })],
  return_type: Named { module: "gleam", name: "Int" },
}
```

Figure 4: Listing 28 visualised as a graph



4.2. Adding the WebAssembly compile target flag

The Gleam compiler supports compiling to both Erlang and JavaScript. When building a project with Gleam, you can choose which target it compiles to by passing the flag `--target <target>` flag to the compiler, where `<target>` is either erlang or javascript. Adding an option for compiling to Wasm first requires support for passing Wasm asn an option to the `--target` flag.

The `--target` flag is generated from the Rust struct `gleam_core::build::Target` shown in Listing 29. Adding a Wasm target option for the Gleam compiler requires adding an extra variant to this enum, as shown in Listing 30.

Listing 29: `gleam_core::build::Target` definition [36]

```rust
pub enum Target {
  #[strum(serialize = "erlang", serialize = "erl")]
  #[serde(rename = "erlang", alias = "erl")]
  Erlang,
  #[strum(serialize = "javascript", serialize = "js")]
  #[serde(rename = "javascript", alias = "js")]
  JavaScript,
}
```

Listing 30: `gleam_core::build::Target` updated definition

```rust
pub enum Target {
  #[strum(serialize = "erlang", serialize = "erl")]
  #[serde(rename = "erlang", alias = "erl")]
  Erlang,
  #[strum(serialize = "javascript", serialize = "js")]
  #[serde(rename = "javascript", alias = "js")]
  JavaScript,
  #[strum(serialize = "webassembly", serialize = "wasm")]
  #[serde(rename = "webassembly", alias = "wasm")]
  WebAssembly,
}
```

4.3. WebAssembly code generation

4.3.1. Entrypoint to code generation in the compiler

To begin generating Wasm, the logic for dictating which compile target is used needs to be updated to handle generating the Wasm target. The change required has been highlighted in Listing 31. The `modules` argument is an array of each `.gleam` file's Typed AST.

Listing 31: Function that delegates code generation for each target [37]

```rust
fn perform_codegen(
  &mut self,
  modules: &[Module]
) -> Result<()> {
  match self.target {
    TargetCodegenConfiguration::JavaScript =>
      self.perform_javascript_codegen(modules),
    TargetCodegenConfiguration::Erlang =>
      self.perform_erlang_codegen(modules),
    TargetCodegenConfiguration::WebAssembly =>
      self.perform_wasm_codegen(modules),
  }
}
```

4.3.2. Generating WebAssembly

Wasm has both a text format and a binary format. For the purpose of readability, examples of Wasm will be written in the text format with the implementation using the binary format. We will use the binary format as it is more compact than the text format, allowing for smaller file size. The benefit of having a smaller file size is that it reduces the download size, which can be beneficial for user experience when being deployed as a web application.

We will use the rust library, wasm-encoder [38], to assist encoding a Rust representation of Wasm instructions into their binary format. The decision to use this library is due to it being the most popular and well supported.

4.3.3. Gleam types in WebAssembly

4.3.3.1. Variants

Gleam supports giving a type multiple constructors. This feature allows you to define a type with multiple variants, similar to enums in other languages. As Gleam allows you to pattern match on a type to run code depending on which constructor was used to create the type, there must be a runtime representation of which variant a type is.

Given the example code in Listing 32, the representation would require a `tag` so that the variant can be checked at runtime. Listing 32 demonstrates how this type can be defined in Wasm. To construct an Animal with the Cat constructor, we would set the value of the tag to be 0 and for the Dog constructor, we would set the tag to be 1. At runtime checking if an instance of Animal is a Cat would be performed by checking if the first field has the value 0.

Listing 32: Example type with multiple constructors

```gleam
type Animal {
  Cat
  Dog
}
```

Listing 33: Wasm representation

```wasm
(type (struct
  (field i32)
))
```

Using this approach we can implement Gleam's Bool type.

Listing 34: Gleam's `Bool` type

```rust
type Bool {
  False
  True
}
```

Constructors can also have different fields as Listing 4.10 shows. To support this, we will use Wasm's subtyping feature as demonstrated in Listing 4.11. We first define the type `$FooBar` with a single field to act as the 'tag', then create a type for each variant that extends this type with their fields added. This allows passing values of type `$FooBar.Foo` and `$FooBar.Bar` in locations that specify their type as `$FooBar`.

Listing 35: Example type with multiple constructors

```gleam
type FooBar {
  Foo(Int)
  Bar(Float)
}
```

Listing 36: Wasm representation of type in Listing 35

```wasm
; FooBar type with tag
(type $FooBar (struct (field i32)))

; Foo variant
(type $FooBar.Foo (sub $FooBar (struct
  (field i32)
  (field (ref $gleam.Int))
)))

; Bar variant
(type $FooBar.Bar (sub $FooBar (struct
  (field i32)
  (field (ref $gleam.Float))
))
```

4.3.3.2. Generic types

Gleam supports parametrising custom types with other types, however Wasm GC does not support this feature. There are two approaches for supporting this feature, one is performing monomorphization and the other is using Wasm's anyref type and casting at runtime. Monomorphization is the process of turning a single generic type and replacing it with many non-generic types for each unique usage of that type. Gleam does not currently perform monomorphization during the analysis phase, so this approach would require a substantial change to the compiler. This means we will instead choose the anyref approach as it will be easier to implement, although there are some drawbacks.

When a field type is `anyref`, it requires that the value's type is a heap type. Wasm's `i64` and `f64` types are not a heap types, meaning if we were to use these types as the representation of `gleam.Int` and `gleam.Float` then integers and floats could not be used in this context. Wasm does have a `i31ref` type that can be used for `gleam.Int`, however this would limit integers size to $\approx \pm 1$ billion. There is no equivalent type for floating point values, so another approach is required.

To resolve this, we can define `gleam.Int` as described in Listing 37 and `gleam.Float` as described in Listing 38. This approach defines a heap allocated type with one field, which will allow `gleam.Int` and `gleam.Float` to be used when a field's type is `anyref`.

Listing 37: Definition of `gleam.Int`

```wasm
(type (struct
  (field i64)
))
```

Listing 38: Definition of `gleam.Float`

```wasm
(type (struct
  (field f64)
))
```

4.3.4. Generating functions

4.3.4.1. Top-level named functions

Gleam functions always return a single value, and can take zero or more parameters. Wasm also has the concept of a function, although it additionally supports returning zero or more values from a function instead of just one. Gleam functions that have a return type of `Nil` return the value `Nil` instead of not returning anything.

When defining a function in Wasm, you must first define its type before defining the function. Given the example Gleam function in Listing 39, we would generate the Wasm in Listing 40.

Listing 39: Example add function definition in Gleam

```gleam
fn add(lhs: Int, rhs: Int) -> Int {
  // function body
}
```

Listing 40: Wasm code generated for Listing 39

```wasm
(type $add.type (func
  (param (ref $gleam.Int))
  (param (ref $gleam.Int))
  (result (ref $gleam.Int))
))

(func $add (type $add.type)
  ;; function body
)
```

4.3.4.2. Anonymous functions

Defining Gleam's anonymous functions in Wasm requires extra work than top-level functions. This is due to the fact that they are defined within an existing function and can capture state from that function, as shown in Listing 41. Wasm has no concept of anonymous functions so we must create them the same way we create top-level functions.

Listing 41: Example closure capturing state

```gleam
pub make_adder(x: Int) -> fn(Int) -> Int {
  fn(y: Int) { x + y }
}

pub fn main() {
  let add_5 = make_adder(5)
  let out = add_5(10)
  // 'out' = '15'
}
```

To represent the value of a closure at runtime, we will need to have a struct type containing a reference to the closure and a struct containing the captured state. When calling the closure, this state will be passed as an extra parameter to the closure so that they can be accessed by the closure.

This approach creates an issue with regular functions as we would need to be able to differentiate at runtime whether a value contains a closure or a top-level function so they can be treated differently. Unfortunately, the Wasm GC proposal does not have a type that can contain either a reference to a function or a reference to a struct. This means we have to have a uniform way to call both top-level functions and closures. To achieve this we will add an extra, but unused, parameter to all top-level functions and have them represented by the same runtime structure as discussed earlier but with the state parameter being empty.

4.3.5. Generating unary and binary expressions

4.3.5.1. Numeric operators

Due to our implementation of `gleam.Int` and `gleam.Float`, we cannot pass them to `i64.<operation>` and `f64.<operation>` instructions. This is because these instructions expect `i64` and `f64` values to be passed, but `gleam.Int` and `gleam.Float` are both Wasm heap types.

We cannot use `i64.<operation>` and `f64.<operation>` instructions directly on our `gleam.Int` and `gleam.Float` types. For `i64` operations this is because they expect values of type `i64` on the stack, but `gleam.Int` is of type `(struct (field i64))`. The same is true of `f64` operations, except they expect `f64` on the stack and `gleam.Float` is of type `(struct (field f64))`.

This can be resolved by using the struct.get instruction to pull the value out of the heap allocated type as demonstrated in Listing 4.16.

Listing 42: Wasm generated for expression `4 + 5`.

```wasm
(struct.new $gleam.Int
  (i64.add
    (struct.get $gleam.Int 0
      (struct.new $gleam.Int (i64.const 4))
    )
    (struct.get $gleam.Int 0
      (struct.new $gleam.Int (i64.const 5))
    )
  )
)
```

4.3.6. Generating local immutable bindings

Given the following Gleam program in Listing 43, we expect that the phrase `"get_x called"` will get printed to the terminal once. This requires storing the value assigned to `x` instead of recomputing the value in each instance it is used as that would cause `"get_x called"` to be printed twice.

Listing 43: Example Gleam program to demonstrate let bindings

```gleam
import gleam/io

fn get_x() {
  io.println("get_x called")
  10
}

fn main() {
  let x = get_x()
  let y = x * x
}
```

Whilst Wasm is a stack based language, it does support creating local variables. A function has to declare how many local variables it requires in its definition, meaning we have to know how many local variables a function has before generating its definition. This is achieved by creating a representation of the function before generating the Wasm code for it.

Listing 44: Example Wasm function with local variables

```wasm
(func (result i64) (local $x i64) (local $y i64)
  (local.set $x (i64.const 5))
  (local.set $y (i64.const 4))

  (i64.add (local.get $x) (local.get $y))
)
```

4.3.7. Generating record field access expressions

Gleam's AST for a record access expression (shown in Listing 45) contains the index of the field being accessed. We can use this value to generate our `struct.get` expression to access the desired field. Directly using this value would result in generating invalid code, as the 0th index in our representation is the type's tag. This means we need to offset the index by `+1`.

Listing 45: Record access AST [39]

```rust
RecordAccess {
  location: SrcSpan,
  typ: Arc<Type>,
  label: EcoString,
  index: u64,
  record: Box<Self>,
}
```

4.3.8. Generating function calls

Given the example code in Listing 46, it would not be possible to use Wasm's call instruction. This is because the instruction takes a function index that has to be known at compile time, instead taking an index or reference to a function. The solution is to use Wasm's `call_ref` instruction in conjunction with the `ref.func` instruction. `ref.func` takes a compile time known index of the function and returns a reference to that function, allowing it to be stored and later called by a `call_ref` instruction. This allows us to generate the example in Listing 46.

Listing 46: Gleam function call example

```gleam
fn foo() { 0 }
fn bar() { 1 }

fn example(flag: Bool) {
  let perform = case flag {
    True -> foo
    False -> bar
  }

  perform()
}
```

4.3.9. Generating block expressions

Both Gleam and Wasm have the concept of a 'block'. In Gleam a block is one or more expressions surrounded by curly braces. In WebAssembly a block is a sequence of instructions that can consume input values from the stack, and produce output values onto the stack. One key difference is a block in Gleam allows the creation of new bindings that only stay in scope for the duration of the block, whereas blocks in Wasm do not allow creating locals. This means we need to hoist the Gleam block bindings up to the function level in the generated WebAssembly. For the example Gleam code in Listing 47, we need to generate the Wasm demonstrated in Listing 48. The highlighted lines in Listing 48 show that the bindings `x` and `y` are being defined outside of the block.

Listing 47: Example block expression in Gleam

```gleam
let foo = {
  let x = 5
  let y = 6

  x + y
}
```

Listing 48: Example block expression in Wasm

```webassembly
(local $foo (ref $gleam.Int))
(local $_block0.x (ref $gleam.Int))
(local $_block0.y (ref $gleam.Int))

(local.set $foo (block (result (ref $gleam.Int))
  (local.set $_block0.x (struct.new $gleam.Int (i64.const 5)))
  (local.set $_block0.y (struct.new $gleam.Int (i64.const 6)))

  (struct.new $gleam.Int (i64.add
    (struct.get $gleam.Int 0 (local.get $_block0.x))
    (struct.get $gleam.Int 0 (local.get $_block0.y))
  ))
))
```

4.3.10. Generating tuple expressions

Wasm has no concepts of tuples, so we need to use its `struct` concept instead. Given the Gleam tuple `#(1, "foo")` we would need to generate a struct with 2 fields, `gleam.Int` and `gleam.String`. Accessing fields from this struct can then be done by using the `index` field in the `TupleIndex` AST (shown in Listing 49) node as this index directly maps to the Wasm struct representation.

Listing 49: Tuple index AST [40]

```rust
TupleIndex {
  location: SrcSpan,
  typ: Arc<Type>,
  index: u64,
  tuple: Box<Self>,
}
```

4.3.11. Generating case expressions

A case expression consists of `subjects` and `clauses`, as shown by the Case AST shown in Listing 50. We need to generate Wasm that evaluates these subjects and run through each clause until a match is found.

Listing 50: `gleam_core::ast::typed::TypedExpr`

```rust
Case {
  subjects: Vec<TypedExpr>,
  clauses: Vec<Clause>,
},
```

We first need to generate Wasm that will evaluate each subject and save the resulting values into a set of Wasm locals. We store the values into locals so that they can be referred to later without having to be evaluated multiple times as that could cause expressions with side-effects to have their side-effects be ran multiple times.

Wasm has no instructions that are similar to Gleam's `case`, so we will instead generate a nested `if then else` expression that tests if a clause should be executed, if it should then it executes it, otherwise it handles the next clause with the same logic until it finally reaches the `unreachable` instruction. This `unreachable` instruction should never be executed as Gleam performs exhaustive pattern matching, meaning code that does not handle every possibility should never reach this stage.

Generating the code to test if a clause (AST Node shown in Listing 51) should be executed requires generating code to first check if it matches all patterns in the `pattern` field. If it fails to match these patterns, it then goes through each set of patterns provided in `alternative_patterns` until either a match is found, or no more patterns remain. If these all fail to match then it returns false to indicate no match was found. If there was a successful match and there is no guard, then it returns true, otherwise it tests the guard and returns the result of that test.

Listing 51: `gleam_core::ast::typed::Clause`

```rust
struct Clause {
  pattern: Vec<Pattern>,
  alternative_patterns: Vec<Vec<Pattern>>,
  guard: Option<ClauseGuard>,
  then: TypedExpr,
}
```

4.3.11.1. Getting `Pattern` matching

Testing if a subject matches a pattern depends on the type of the subject and the provided pattern. As an example, if the subject is of type `gleam.Int`, then it can only be matched by the `Int`, `Variable`, `Assign` and `Discard` patterns.

Listing 52: `gleam_core::ast::Pattern`

```rust
enum Pattern {
  Int, Float, String, Variable, VarUsage,
  Assign, Discard, List, Constructor, Tuple,
  BitArray, StringPrefix,
}
```

The `VarUsage` and `BitArray` patterns have not been implemented as they are used for Gleam's `BitArray` feature.

The `Constructor` pattern allows matching against which variant of a type the subject is. As we have a tag stored in the first index of every type, we can use this value to check if the variant described in the pattern matches the subject at runtime. If the tags do not match, we want to early return false, but if they do match we want to proceed to check if all arguments match. The code for generating the Wasm for this is shown in Listing 53.

Listing 53: Code for checking if `Constructor` pattern matches

```rust
Instruction::If {
  type_: BlockType::Result(ValType::I32),
  cond: do_tags_match,
  then: vec![do_arguments_match],
  else_: vec![Instruction::I32Const(0)],
}
```

Checking if the subject's type tag matches the pattern's variant tag is shown in Listing 54. The function `runtime::get_type_tag` creates an `Instruction` that extracts the tag from the given value. We then compare that value to `target_variant_tag`, which has been computed by looking up what the tag associated with the variant provided in the pattern should be.

Listing 54: Code for checking subject's tag matches pattern

```rust
let target_type_index = program
  .resolve_type_index(encoder, type_);

let target_variant_tag = program
  .resolve_type_variant_tag(target_type_index, variant);

let do_tags_match = Instruction::I32Eq {
  lhs: runtime::get_type_tag(
    subject_type_index,
    subject_local),
  rhs: Instruction::I32Const(target_variant_tag),
};
```

The Wasm for checking if all argument patterns match is a nested `if-then-else` expression, meaning if a single match fails then it does not need to continue checking. For each argument, we generate the Wasm required for pattern matching that argument, and if it returns true then we check the next argument, otherwise we return false. We generate this, as shown in Listing 55, by utilising Rust's `DoubleEndedIterator::rfold` method. This method has two parameters: an initial value; and a function that has an accumulator parameter and an element parameter. On the first pass, it calls the passed function with the initial value in the accumulator parameter and the final element in the iterator as the element parameter. The result of this function call is then used as the accumulator parameter for the next call, with the previous element from the iterator as the element parameter. Once all elements have been iterated over, the accumulator value is returned.

Listing 55: Code for checking argument patterns match

```rust
let do_arguments_match = arguments.iter()
  .enumerate()
  .rfold(
    Instruction::I32Const(1),
    |then, (argument_index, argument)| {
      let field_type = program
        .resolve_type(encoder, &arg.value.type_());
      let field_local = function
        .declare_anonymous_local(field_type);
      let field = runtime::get_field_from_type(
        variant_type_index, argument_index,
        runtime::cast_to(subject_local, variant_heap_type));

      let does_argument_match_pattern = encode_case_pattern(
        program, encoder, function, &arg.value, field_local);

      Instruction::Block {
        type_: BlockType::Result(ValType::I32),
        code: vec![
          Instruction::LocalSet {
            local: field_local, value: Box::new(field),
          },
          Instruction::If {
            type_: BlockType::Result(ValType::I32),
            cond: Box::new(does_argument_match_pattern),
            then: vec![then],
            else_: vec![encoder::Instruction::I32Const(0)],
          }
        ]
      }
    }
  );
```

4.3.11.2. Generating `ClauseGuard` guard.

Due to limitations imposed by Erlang on what it allows in its guard clause, Gleam imposes similar limitations. Rather than allowing any expression in guard clauses, a limited subset of expressions is allowed as seen in Listing 56. The generated Wasm here resembles the same as what is generated for a normal expression.

Listing 56: `gleam_core::ast::ClauseGuard`

```rust
enum ClauseGuard {
  Equals, NotEquals, GtInt, GtEqInt,
  LtInt, LtEqInt, GtFloat, GtEqFloat,
  LtFloat, LtEqFloat, Or, And,
  Not, Var, TupleIndex, FieldAccess,
  ModuleSelect, Constant,
}
```

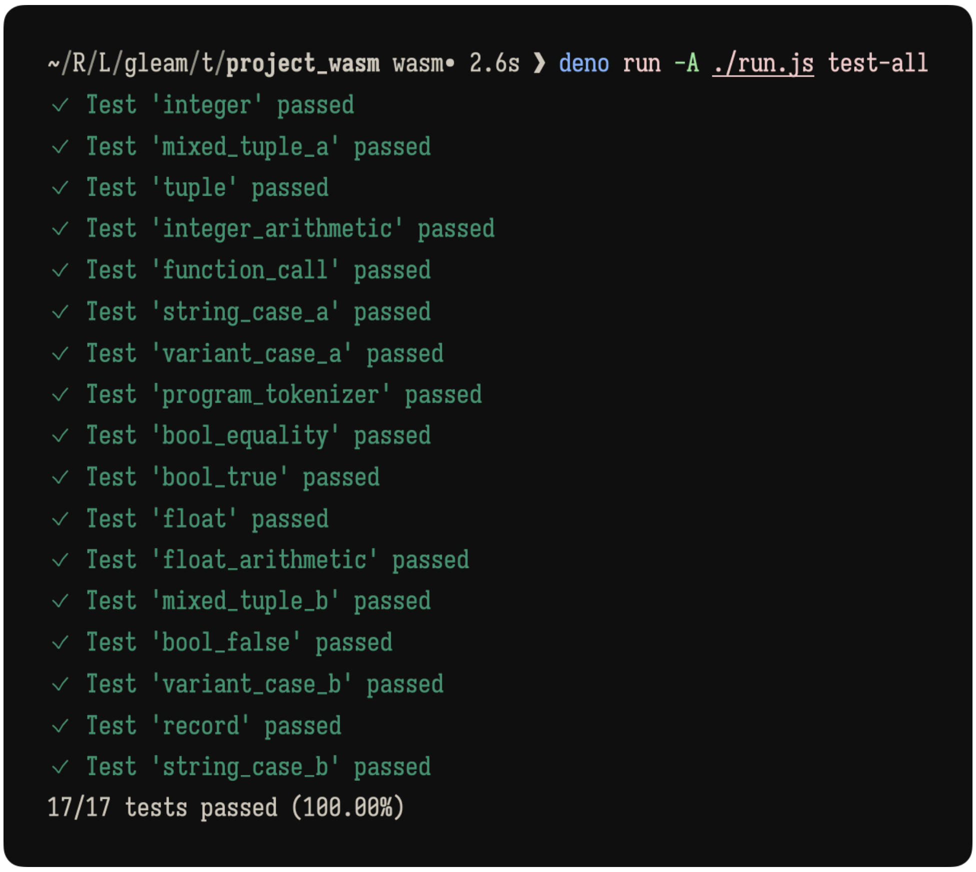
4.3.12. Generating use expressions

As use expressions are syntactic sugar, we do not have to write code specifically for them. By implementing anonymous functions, `use` code can successfully be compiled.

5. Testing

In the `test/project_wasm` directory we have a test suite to test the Wasm target. This test suite allowed checking that the behaviour did not regress whilst making changes. The test suite can be ran from the `test/project_wasm/run.js` file by running the following command `deno run -A ./run.js test-all` whilst in the `test/project_wasm` directory. An example run of the test suite can be seen in Figure 5.

Figure 5: Example run of the test suite



```
~/R/L/gleam/t/project_wasm wasm• 2.6s ❯ deno run -A ./run.js test-all
✓ Test 'integer' passed
✓ Test 'mixed_tuple_a' passed
✓ Test 'tuple' passed
✓ Test 'integer_arithmetic' passed
✓ Test 'function_call' passed
✓ Test 'string_case_a' passed
✓ Test 'variant_case_a' passed
✓ Test 'program_tokenizer' passed
✓ Test 'bool_equality' passed
✓ Test 'bool_true' passed
✓ Test 'float' passed
✓ Test 'float_arithmetic' passed
✓ Test 'mixed_tuple_b' passed
✓ Test 'bool_false' passed
✓ Test 'variant_case_b' passed
✓ Test 'record' passed
✓ Test 'string_case_b' passed
17/17 tests passed (100.00%)
```

Listing 57 shows the file structure of a single test. `gleam.toml`, `manifest.toml` and `src/test_<name>.gleam` are all required files for a typical gleam program. To specify the expected outcome of the test there is an `expected.txt` file. The contents of this file follow the format `<type>:<value>`. The test runner uses this information to figure out whether the test has succeeded or not by checking the value returned from `main` against what is specified in `expected.txt`.

Listing 57: File structure of a test

```
└── test_<name>
    ├── src
    │   └── test_<name>.gleam
    ├── expected.txt
    ├── gleam.toml
    └── manifest.toml
```

In Listing 58 we can see an example test. The criteria for this test to pass is that it returns the integer 21. If the test fails to compile, fails to run, or produces a value that is not 21 then the test is deemed to have failed as it does not have the behaviour expected.

Listing 58: An example test to test tuple behaviour

```gleam
pub fn main() {
  let a = #(1, 2, 3)
  let b = #(4, 5, 6)

  let a_sum = a.0 + a.1 + a.2
  let b_sum = b.0 + b.1 + b.2

  a_sum + b_sum
}
```

6. Evaluation

6.1. Performance comparison

In this section we will cover the performance difference between Gleam's existing Javascript and Erlang targets with the Wasm target.

All benchmarks have been ran on a 2023 M3 14″ MacBook Pro with 11 CPU cores running MacOS 14.4.1. The generated Erlang code was ran on Erlang/OTP 26. The generated JavaScript and Wasm were ran on Deno 1.42.4 with V8 12.3.219.9.

The benchmark titled "WebAssembly (optimised)" benchmarked the output from the Wasm target piped through the `wasm-opt` tool to optimise the generated Wasm. The `wasm-opt` tool is a command-line tool that analyses the Wasm bytecode and attempts to generate code that should have better runtime performance [41].

The benchmark titled "JavaScript (Bundled + Minified)" benchmarked the output from the JavaScript target piped through `esbuild` with the `--minify` and `--bundle` flags applied. `esbuild` is a tool that can minify and bundle JavaScript code. Minifying is the process of minimising the amount of bytes to represent the same JavaScript code by reducing whitespace and shortening the names of variables. Bundling is the process of creating a single `.js` file from multiple `.js` files.

6.1.1. Fibonacci

The code in Listing 59 calculates the 40th Fibonacci number using a recursive implementation. The results of benchmarking this program on each target is in Table 1. The Wasm target produces code that runs half as quickly as the JavaScript or Erlang targets.

The implementation of this Fibonacci algorithm calls itself $331,160,280$ times and performs $496,740,420$ maths operations when `n = 40`. As the implementation of `gleam.Int` in the Wasm target is heap allocated, there are $827,900,700$ `struct.new` invocations. The implementation for functions calls results in an extra $331,160,280$ `struct.new` invocations.

Table 1: Performance of different targets

Target	Time	Range
JavaScript (Bundled + Minified)	701.8ms ± 23.0ms	689.3ms … 765.5ms
Erlang	825.9ms ± 40.2ms	809.1ms … 940.1ms
JavaScript	834.3ms ± 54.0ms	810.1ms … 986.9ms
Wasm (Optimised)	1.504s ± 0.015s	1.483s … 1.523s
Wasm	1.719s ± 0.039s	1.667s … 1.775s

Listing 59: Fibonacci

```gleam
fn fibonacci(n: Int) -> Int {
  case n {
    0 | 1 -> n
    _ -> fibonacci(n - 1) + fibonacci(n - 2)
  }
}

pub fn main() {
  fibonacci(40)
}
```

6.1.2. Sieve of Eratosthenes

The code in Listing 60 computes the amount of prime numbers under 30,000 using the "Sieve of Eratosthenes" algorithm. The results of benchmarking this program on each target is in Table 2. The results show that the generated Wasm has similar performance to both the Erlang and JavaScript targets.

Due to the nature of the algorithm's recursive implementation, and the implementation not being written to take advantage of TCO (*Tail Call Optimisation*), the call stack overflows for both the Wasm and JavaScript targets. To get around this, we pass the following flag to Deno `--v8-flags=--stack-size=10000` to set the size of the stack to 10MB.

Table 2: Performance of different targets

Target	**Time**	**Range**
Erlang	180.0ms ± 8.5ms	175.4ms … 210.8ms
WebAssembly (Optimised)	198.6ms ± 1.3ms	195.9ms … 201.0ms
WebAssembly	223.1ms ± 2.0ms	220.3ms … 227.1ms
JavaScript (Bundled + Minified)	253.9ms ± 1.6ms	251.3ms … 257.0ms
JavaScript	267.7ms ± 1.7ms	264.9ms … 271.7ms

Listing 60: Sieve of Eratosthenes

```gleam
fn range(from: Int, to: Int) -> List(Int) {
  case from < to {
    True -> [from, ..range(from + 1, to)]
    False -> []
  }
}

fn length(list: List(Int)) -> Int {
  case list {
    [] -> 0
    [_, ..tail] -> 1 + length(tail)
  }
}
```

```

fn filter(list: List(Int), should_keep: fn(Int) -> Bool) -> List(Int) {
  case list {
    [] -> []
    [head, ..tail] ->
      case should_keep(head) {
        True -> [head, ..filter(tail, should_keep)]
        False -> filter(tail, should_keep)
      }
  }
}

fn sieve(n: Int) -> List(Int) {
  do_sieve(range(2, n))
}

fn do_sieve(list: List(Int)) -> List(Int) {
  case list {
    [] -> []
    [head, ..tail] -> {
      [head, ..do_sieve(filter(tail, fn(item) { item % head != 0 }))]
    }
  }
}

pub fn main() {
  let primes = sieve(30_000)

  length(primes)
}
```

6.2. Performance evaluation

6.2.1. Fibonacci

In Section 6.1.1, we can see that our Wasm target performs slower than the JavaScript and Erlang targets. Listing 61 demonstrates the optimal Wasm that could be generated for the code in Listing 59. Table 3 demonstrates that our target implementation generates Wasm that performs approximately 3x slower than optimal.

Table 3: Fibonacci performance comparison for WebAssembly

Target	Time	Range
WebAssembly (Handwritten)	552.2ms ± 2.2ms	549.2ms … 555.7ms
WebAssembly (Optimised)	1.509s ± 0.020s	1.484s … 1.547s
WebAssembly	1.710s ± 0.023s	1.684s … 1.740s

Listing 61: Optimal Wasm for Fibonacci

```wasm
(module
  (func $fib (param $n i64) (result i64)
      (if (result i64)
          (if (result i32)
              (i64.eq (local.get $n) (i64.const 0))
              (then (i32.const 1))
              (else (i64.eq (local.get $n) (i64.const 1))))
          (then (local.get $n))
          (else (i64.add
              (call $fib (i64.sub (local.get $n) (i64.const 1)))
              (call $fib (i64.sub (local.get $n) (i64.const 2)))
          )))
  )

  (func $main (export "main") (result i64)
    (call $fib (i64.const 40))
  )
)
```

The design rationale for representing `gleam.Int` as a heap allocated type is described in Section 4.3.3.2. If we rewrite Listing 61 but with the requirement of representing `gleam.Int` as a heap allocated type, then we can see that the performance impact of this decision doubles the time taken to calculate `fib(40)`.

If we rewrite Listing 61 with the requirement of handling function calls as described in Section 4.3.8, then we can see that this also has a performance impact, although not as great as the performance cost of `gleam.Int`'s implementation.

Table 4: Time to compute `fib(40)` with handwritten WebAssembly

Target	Time	Range
Optimal	579.1ms ± 1.7ms	575.6ms … 581.1ms
With heap allocated function	805.5ms ± 27.1ms	776.5ms … 840.7ms
With heap allocated integer	1.258s ± 0.075s	1.156s … 1.390s

6.3. Future work

6.3.1. Performance improvements

6.3.1.1. Optimising `gleam.Int` and `gleam.Float`.

To improve performance we could optimise the implementation of `gleam.Int`, as well as `gleam.Float`, as the current implementation has a significant performance cost as seen in Section 6.2.1. This would likely require performing monomorphization due to the limitations of Wasm's current type system. By performing monomorphization we could use the `i64` and `f64` types provided by Wasm and avoid the overhead of calling `struct.get` and `struct.new` when accessing and creating integers and floats.

6.3.1.2. Optimising function calls

There is also room to improve performance by optimising how function calls are handled, as seen in Section 6.2.1. This could be achieved by inspecting the AST for the cases where it is known that a function call is calling a named function, rather than an anonymous function, and generate code that doesn't have to handle both cases.

6.3.1.3. Optimising case expressions

Another area to improve performance is by optimising the logic for choosing which branch of a case expression to enter. Currently it goes through each branch and runs each pattern until a match is found, however this could result in checking the same pattern for multiple branches despite it having already computed that the pattern returns true or false. An example optimisation could be computing a decision tree and then generating the Wasm from this rather than directly from the AST.

Listing 62: Example case expression that could benefit from a decision tree

```gleam
case #(1, 2, 3), #(4, 5 ,6) {
  #(1, 2, 3), #(7, 8, 9) -> 0
  #(1, 2, 3), #(4, 5, 6) -> 1
  _ -> 2
}
```

Given the case expression in Listing 62, the current algorithm works as follows:

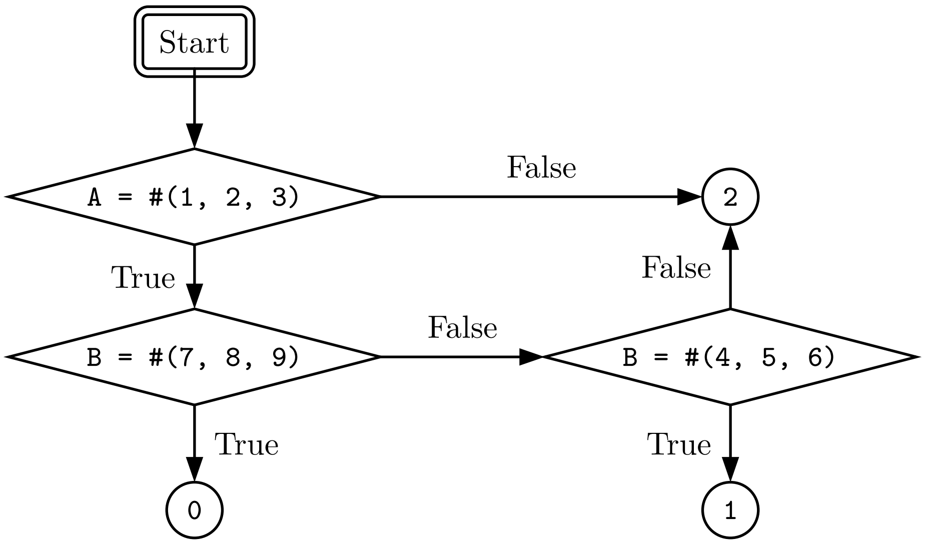
```
   input: $a$ is #(1, 2, 3) and $b$ is #(4, 5, 6)
1  if $a$ = #(1, 2, 3) then
2    if $b$ = #(7, 8, 9) then
3      return 0
4  if $a$ = #(1, 2, 3) then
5    if $b$ = #(4, 5, 6) then
6      return 1
7  return 2
```

This is suboptimal as the algorithm checks if the first subject matches the same pattern twice. If we instead created a decision tree as described in , it would work as follows:

```
   input: $a$ is #(1, 2, 3) and $b$ is #(4, 5, 6)
1  if $a$ = #(1, 2, 3) then
2    if $b$ = #(7, 8, 9) then
3      return 0
4    if $b$ = #(4, 5, 6) then
5      return 1
6  return 2
```

As we can see, this reduces the number of comparisons required for the final 2 cases. Depending on the complexity of the case expression this could achieve a significant improvement in performance.

Figure 6: Decision tree for Listing 62



6.3.2. Implementing `BitArray`

Currently the `BitArray` type, and features that depend on it, have not been implemented. Wasm does not have an equivalent feature so it would likely require a custom type with at least two fields: an array of `i8` to represent the data; and an `i64` to contain the number of bits the array contains. It is required to know the number of bits as a `BitArray` does not have to be an exact number of bytes, for example it could contain only 7 bits. To reduce copying the entire array of bytes when extracting data from the front of the array, the type could also contain an `i64` to indicate a bit offset for where in the data the current `BitArray` starts.

6.3.3. Bug in equality checking

Currently the code in Listing 63 fails to compile, this is because we do not perform monomorphization and at the time of generating the function is it unknown what the type of `a` is. This results in being unable to lookup the correct equality function to call. This could be solved if Wasm had a structural equality instruction, however it only provides a referential equality instruction. This could also be solved by generating a function that attempts to cast the types to the set of known types at runtime until it succeeds, however this will likely have a significant performance impact that will grow as the number of types in a program grow.

Listing 63: Demonstration of equality issue

```gleam
fn are_equal(lhs: a, rhs: a) -> Bool {
  lhs == rhs
}
```

6.3.4. Compiler caching

Currently the Gleam compiler saves in a cache modules that have already been compiler so that it doesn't repeat work if they have not been changed. Unfortunately, the current Wasm target implementation compiles all modules into a single `.wasm` file. This means all modules need to be compiled every time, but due to the caching behaviour they are not. To work around this problem, the build directory is removed between each compilation.

Fixing this issue would either be to have the Gleam compiler not use the caching logic for the Wasm target, or to generate each module as their own `.wasm` file and merge them together as a final step. This will likely introduce challenges as importing types in a `.wasm` module is currently not possible, however there is a proposal to the Wasm specification that aims to provide this feature.

6.3.5. Performing I/O

Currently the Wasm target has no ability to perform any I/O operations, such as logging to the terminal, reading files or making HTTP requests. This functionality is currently possible in Wasm by using the WASI (*WebAssembly System Interface*) set of APIs. The downside of using WASI is that it does not yet provide a stable set of APIs as it is still in preview. WASI Preview 2 makes use of the Component Model proposal, which is currently in Phase 1 of the proposal process which means there is limited support for this feature by existing tools.

Another reason for not using the WASI set of APIs is that Preview 1 is incompatible with values created using the features provided by the GC proposal, and with Preview 2 being released midway through the project.

6.3.6. Mutually recursive types

Currently the code in Listing 64 fails to compile. This is due to how types are generated in the Wasm binary. By default, types in a Wasm binary can only reference types that are defined earlier in the binary. This means that you cannot have two types reference each other as one has to be defined first. The way to resolve this is by putting mutually recursive types into a `rec` block in the binary as shown in Listing 65.

Listing 64: Example of mutually recursive types

Gleam
```gleam
type Foo {
  FooWithBar(Bar)
  FooWithoutBar
}

type Bar {
  BarWithFoo(Foo)
  BarWithoutFoo
}
```

Listing 65: Group of mutually recursive types

WebAssembly
```wasm
(rec
  (type $Foo (struct (field (ref $Bar))))
  (type $Bar (struct (field (ref $Foo))))
)
```

7. References

[1] "Gleam." Accessed: Mar. 02, 2024. [Online]. Available: https://gleam.run/

[2] "Index - Erlang/OTP." Accessed: Mar. 02, 2024. [Online]. Available: https://www.erlang.org/

[3] "JavaScript | MDN." Accessed: Mar. 02, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[4] "Frequently asked questions – Gleam." Accessed: Mar. 02, 2024. [Online]. Available: https://gleam.run/frequently-asked-questions

[5] Type inferred too weakly: usage not taken into consideration · Issue #790 · gleam-lang/gleam. Accessed: Mar. 02, 2024. [Online]. Available: https://github.com/gleam-lang/gleam/issues/790#issuecomment-691573757

[6] R. Hindley, "The Principal Type-Scheme of an Object in Combinatory Logic," *Transactions of the American Mathematical Society*, vol. 146, pp. 29–60, 1969.

[7] "Ints - The Gleam Language Tour." Accessed: Mar. 02, 2024. [Online]. Available: https://tour.gleam.run/basics/ints

[8] "Erlang – Advanced." Accessed: Mar. 02, 2024. [Online]. Available: https://www.erlang.org/doc/efficiency_guide/advanced

[9] "IEEE Standard for Floating-Point Arithmetic," *Transactions of the American Mathematical Society*, pp. 1–84, 2019.

[10] "Number - JavaScript | MDN." Accessed: Mar. 02, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number#number_encoding

[11] "Floats - The Gleam Language Tour." Accessed: Mar. 02, 2024. [Online]. Available: https://tour.gleam.run/basics/floats/

[12] "Erlang – Data Types." Accessed: Mar. 02, 2024. [Online]. Available: https://www.erlang.org/doc/reference_manual/data_types.html#representation-of-floating-point-numbers

[13] "Lists - The Gleam Language Tour." Accessed: Mar. 10, 2024. [Online]. Available: https://tour.gleam.run/basics/lists/

[14] "Erlang – Data Types." Accessed: Mar. 10, 2024. [Online]. Available: https://www.erlang.org/doc/reference_manual/data_types.html#list

[15] "ECMAScript® 2025 Language Specification." Accessed: Mar. 10, 2024. [Online]. Available: https://tc39.es/ecma262/multipage/indexed-collections.html#sec-array-objects

[16] "gleam/compiler-core/templates/prelude.mjs at main · gleam-lang/gleam." Accessed: Mar. 10, 2024. [Online]. Available: https://github.com/

gleam-lang/gleam/blob/98faf3c0aa549d3c321ae75d174f8cb76a0ec232/compiler-core/templates/prelude.mjs#L13-L91

[17] “gleam/compiler-core/src/erlang.rs at main · gleam-lang/gleam.” Accessed: Mar. 10, 2024. [Online]. Available: https://github.com/gleam-lang/gleam/blob/98faf3c0aa549d3c321ae75d174f8cb76a0ec232/compiler-core/src/erlang.rs#L509-L514

[18] “Erlang – Data Types.” Accessed: Mar. 10, 2024. [Online]. Available: https://www.erlang.org/doc/reference_manual/data_types.html#tuple

[19] “gleam/compiler-core/src/javascript/expression.rs at main · gleam-lang/gleam.” Accessed: Mar. 10, 2024. [Online]. Available: https://github.com/gleam-lang/gleam/blob/98faf3c0aa549d3c321ae75d174f8cb76a0ec232/compiler-core/src/javascript/expression.rs#L660-L664

[20] “Array - JavaScript | MDN.” Accessed: Mar. 10, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

[21] “gleam/compiler-core/src/erlang.rs at main · gleam-lang/gleam.” Accessed: Mar. 10, 2024. [Online]. Available: https://github.com/gleam-lang/gleam/blob/98faf3c0aa549d3c321ae75d174f8cb76a0ec232/compiler-core/src/erlang.rs#L88-L145

[22] “Erlang – Data Types.” Accessed: Mar. 10, 2024. [Online]. Available: https://www.erlang.org/doc/reference_manual/data_types.html#record

[23] “gleam/compiler-core/src/javascript.rs at main · gleam-lang/gleam.” Accessed: Mar. 10, 2024. [Online]. Available: https://github.com/gleam-lang/gleam/blob/98faf3c0aa549d3c321ae75d174f8cb76a0ec232/compiler-core/src/javascript.rs#L215-L285

[24] “Classes - JavaScript | MDN.” Accessed: Mar. 10, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

[25] “WebAssembly on Cloudflare Workers.” Accessed: Apr. 30, 2024. [Online]. Available: https://blog.cloudflare.com/webassembly-on-cloudflare-workers

[26] “Getting Started.” Accessed: Apr. 30, 2024. [Online]. Available: https://book.leptos.dev/getting_started/index.html

[27] “Lichess's Blog • Stockfish 13 NNUE on Lichess • lichess.org.” Accessed: Apr. 30, 2024. [Online]. Available: https://lichess.org/@/lichess/blog/stockfish-13-nnue-on-lichess/YDOKRxQA

[28] “WebAssembly is here | Unity Blog.” Accessed: Apr. 30, 2024. [Online]. Available: https://blog.unity.com/engine-platform/webassembly-is-here

[29] “Plugins - Zellij User Guide.” Accessed: Apr. 30, 2024. [Online]. Available: https://zellij.dev/documentation/plugins

[30] De Macedo, João and Abreu, Rui and Pereira, Rui and Saraiva, and João, “WebAssembly versus JavaScript: Energy and Runtime Performance,” *2022 International Conference on ICT for Sustainability (ICT4S)*. pp. 24–34, 2022.

[31] “FAQ - WebAssembly.” Accessed: Mar. 24, 2024. [Online]. Available: https://webassembly.org/docs/faq/#will-webassembly-support-view-source-on-the-web

[32] “WebAssembly/function-references: Proposal for Typed Function References.” Accessed: Mar. 23, 2024. [Online]. Available: https://github.com/WebAssembly/function-references

[33] “WebAssembly/gc: Branch of the spec repo scoped to discussion of GC integration in WebAssembly.” Accessed: Mar. 23, 2024. [Online]. Available: https://github.com/WebAssembly/gc

[34] “WebAssembly/proposals: Tracking WebAssembly proposals.” Accessed: Mar. 23, 2024. [Online]. Available: https://github.com/WebAssembly/proposals

[35] “meetings/process/phases.md at main · WebAssembly/meetings.” Accessed: Mar. 23, 2024. [Online]. Available: https://github.com/WebAssembly/meetings/blob/main/process/phases.md#4-standardize-the-feature-working-group

[36] “gleam/compiler-core/src/build.rs at v1.1.0 · gleam-lang/gleam.” Accessed: Apr. 18, 2024. [Online]. Available: https://github.com/gleam-lang/gleam/blob/431b9bb7448c64e1a336ed04343190cfe1974638/compiler-core/src/build.rs#L54-L61

[37] “gleam/compiler-core/src/build/package_compiler.rs at v1.1.0 · gleam-lang/gleam.” Accessed: Apr. 18, 2024. [Online]. Available: https://github.com/gleam-lang/gleam/blob/431b9bb7448c64e1a336ed04343190cfe1974638/compiler-core/src/build/package_compiler.rs#L272-L291

[38] Bytecode Alliance, “wasm-encoder - crates.io.” Accessed: Jan. 06, 2024. [Online]. Available: https://crates.io/crates/wasm-encoder

[39] “gleam/compiler-core/src/ast/typed.rs at v1.1.0 · gleam-lang/gleam.” Accessed: Apr. 18, 2024. [Online]. Available: https://github.com/gleam-lang/gleam/blob/431b9bb7448c64e1a336ed04343190cfe1974638/compiler-core/src/ast/typed.rs#L86-L92

[40] “gleam/compiler-core/src/ast/typed.rs at v1.1.0 · gleam-lang/gleam.” Accessed: Apr. 18, 2024. [Online]. Available: https://github.com/gleam-lang/gleam/blob/431b9bb7448c64e1a336ed04343190cfe1974638/compiler-core/src/ast/typed.rs#L109-L114

[41] “WebAssembly/binaryen: Optimizer and compiler/toolchain library for WebAssembly.” Accessed: Apr. 30, 2024. [Online]. Available: https://github.com/WebAssembly/binaryen