Comparing a Tree-walk Interpreter with JIT compilation and embedding via Go-plugins

Evaluating the trade-offs of using the Go-plugin API for JIT compilation while comparing the approach with a Tree-walk interpreter

9525469

June 1, 2024

Contents

List of Listings

List of Figures

List of Tables

The goal of this paper is to evaluate whether the usage of the Go plugin API is feasible for just-in-time compilation of a query language intended for a high performance in memory data storage. This evaluation is done based upon the criteria of the ease of usability, performance and the robustness of the resulting implementation. For the sake of comparison the query language as well as its features are introduced. A just in time compiler is implemented and benchmarked against the same expressions evaluated with the currently employed tree walk interpreter. The paper explores the different possibilities for accessing the Go compiler, working with the Go plugin API and highlights several benchmarks comparing the performance of the new JIT compiler and the previous language evaluation implementation.

Das Ziel dieser Arbeit besteht darin zu evaluieren ob sich das Einsetzen der Go plugin API für die JIT Kompilation einer Abfragesprache einer effizienten Datenbank, die ihre Daten ausschließlich im Arbeitsspeicher ablegt, rentiert. Diese Abwägung basiert auf den Kriterien der Verwendbarkeit, Effizienz und der Stabilität der angewendeten Lösung. Um den Vergleich unter realistischen Bedingungen zu testen wird vorerst die Abfragesprache sowie einige umfangreicher werdende Beispiele vorgestellt. Diese Abfragen werden dann mit Hilfe des JIT compiliert. Das Ergebnis wird dann mit der derzeitige Implementierung die das Evaluieren mit einem Tree-walk Interpreter umsetzt, verglichen. Auch werden unterschiedliche Art und Weisen des Aufrufes des Go compiler, die Verwendung des Go plugin paketes und Tests auf die Effizienz des neuen JIT und des alten Tree-walk Interpreter vorgestellt und diskutiert.

1 Introduction

The query language is the singular interface for accessing, reading, creating and removing data in a database. This requires the query language to provide a high degree of performance in the sense of performing processing intensive queries in a fast enough time for real-time responsiveness, especially for an in memory data store with the aspiration for high performance.

Optimisations for database query languages are common, such as in the embeddable Database *SQLite* with the *SQLite Query Optimizer*[1] and the *Next-Generation Query Planner*[1] both supporting a variety of optimisations after compiling SQL expressions to byte-code instead of walking the AST[1][2]. *PostgreSQL* is an other example of a database optimising its SQL queries, using a JIT compiler[2][4].

There are several optimisations applicable for any programming language and query languages in particular [5, 3.3 Optimisations]. Some were already applied to the query language [6]. Implementing a JIT compiler can significantly improve the performance of a long running highly processing intensive program and allows it to outperform optimisations applied to the AST before walking it or compiling to bytecode instructions and executing them in a dedicated virtual machine [3, 4. Results].

The implementation of a JIT compiler introduces complexity into the codebase for the generation of optimized assembly is inherently complex and time consuming [7, 1. Introduction] while also opening the door for potential attacks [8, 2. Challenges Securing JavaScript JIT] [8, Table 1.]. Another issue is the platform dependence, a JIT compiler has to generate native code for all platforms and operating systems the database and therefore the query language support - increasing complexity and maintenance [9, Abstract]. Furthermore a JIT compiler significantly increases the memory usage [10, Fig. 1.] of the interpreter as well as requiring a not to be disregarded startup time made up of the time to start the JIT compiler and the time to compile the language constructs to machine code. [9, 4.2.7 Breakdown of compilation times].

To evaluate whether the aforementioned positive aspects outweigh the negative points and a JIT therefore does improve the query languages performance, determines the scope of this paper. The examined performance is measured on a fork of the original language project [11], this fork was necessary to pave the way for just in time compilation [12].

[1] Abstract Syntax Tree: tree of syntax nodes

[2] Just in time compiler: compiling methods on demand while a program is running [3, 1. Introduction]

2 Query Language

The query language is generic by design and by usage of the newly introduced generic proposal [13]. This allows for rudimentary and complex language variations depending on the data type the desired language is created with. Such as complexity ranging from boolean algebra (see Listing 2.1 and its corresponding language definition Listing 2.2), over arithmetic operations to complex queries on lists of objects, such as filtering for values, mapping and mutating elements of the list and iterating over the entries of the list.

```
true | false
a & !b
let c=true;
if c then
    a&b
else
    a|b
```

Listing 2.1: Boolean algebra

```
var boolParser = funcGen.New[bool]()
    .AddConstant("false", false)
    .AddConstant("true", true)
    .AddSimpleOp("^", true,
            func(a, b bool) (bool, error) { return a != b, nil })
    .AddSimpleOp("=", true,
            func(a, b bool) (bool, error) { return a == b, nil })
    .AddSimpleOp("|", true,
            func(a, b bool) (bool, error) { return a || b, nil })
    .AddSimpleOp("&", true,
            func(a, b bool) (bool, error) { return a && b, nil })
    .AddUnary("!", func(a bool) (bool, error) { return !a, nil })
    .SetToBool(func(c bool) (bool, bool) { return c, true })
```

Listing 2.2: Generating boolean algebra

2.1 Feature set

To limit the scope of performance evaluation, the feature set is determined as the language definition and dialect[1], specified in the `value` package of the `parser2` [12, `value` package] project. This dialect allows for aggregation, filtering and mutating large lists of data sets consisting of objects with many key value pairs.

```
persons
    .map(p -> p.Name)
    .reduce((a,b) -> a+", "+b)
```

Listing 2.3: Reducing and formatting objects in lists - source [12]

```
persons
  .accept(p -> p.PlaceOfBirth="New York" & p.Age>21)
  .map(e -> e.Name+": "+e.Age)
  .reduce((a,b) -> a+", "+b)
```

Listing 2.4: Applying filter, mapping, join entries with comma - source [12]

Listing 2.3 and Listing 2.4 access the global `person` constant using the dialect specific facilities for creating a list of objects of type `Person` as global state (see Listing 8.1).

```
func namedFunction(argument) argument+2;
let m = (argument) -> argument+2;
let add = (a,b) -> a+b;
```

Listing 2.5: Named Functions and unnamed/anonyoums functions - source [12]

The just in time compiler targets named functions and anonyoums functions (see Listing 2.5) and will attempt to compile their contents when conditions for their compilation are hit.

2.2 Evaluation

The runtime currently employs the visitor pattern to walk the AST the parser generated in the previous step of the stages necessary to transform a given character stream to an executable data structure. To evaluate a given input, the runtime generates a function once for each tree node it visits. This improves the performance rapidly compared to a naive tree-walk-interpreter. Furthermore the AST is optimized before being walked by the function generating stage of the runtime.

This evaluation strategy requires the runtime to hold a substantial amount of data structures in memory compared to a bytecode compiler and its corresponding virtual machine.

[1] Refers to the usage of [12] to define and generate a language for a specific data type, with static global functions and constant values

3 Compiler Invocation

Instead of manually generating optimized assembly for each function to be compiled on the fly, the go compiler toolchain is invoked and receives the previously generated Go source code that was computed from the query languages AST. This omits the complexity of implementing and maintaining several platform specific machine code generator compiler backends while allowing the JIT compiler to support all platforms supported by the go toolchain.

The method of invoking the compiler tool-chain has significant effects on the startup performance, the robustness and the complexity of the JIT compiler. This chapter highlights two possible approaches for invoking the compiler tool-chain.

3.1 Including the Compiler Source Code

The first idea of invoking the compiler tool-chain, is to include the source code of the compiler as a library and simply start it while passing in the generated go code. This does not require the compiler tool-chain to exist on the target system and omits the overhead of starting the compiler process. However this approach can not be used for the source code since the go compiler is not stable nor accessible outside of the go compiler tool-chain [14, *(gcToolchain).gc*] due to the usage of `internal` packages [15].

3.2 Invoking the local Go Compiler

The remaining method is to start the locally available compiler tool-chain via the `exec.Cmd` interface [16, Overview]. This enables requesting the operating system to invoke the compiler. Approaching the problem with this method has the downside of requiring the compiler to exist on the target system, the overhead of tasking the operating system with starting the compiler, writing the generated code to a temporary file and compiling this temporary file instead of doing all of the aforementioned inside of the JIT by including the compiler as a library as introduced before. Listing 3.1 shows a simplified implementation of a function invoking the go compiler.

```
func invokeCompiler(code string) {
    f, _ := os.CreateTemp(".", "jit_*.go")
    defer os.Remove(f)
    f.WriteString(code)
    exec.Command("go", "build", f.Name()).Run()
}
```

Listing 3.1: Tool-chain invocation

4 Plugin API

The `plugin` package enables the loading of shared objects compiled with the `-buildmode=plugin` compiler flag and the resolution of symbols contained in the plugin [17, Overview]. This allows the compilation and loading of go code while running a program and therefore mirrors the way a traditional just in time compiler would work.

4.1 Compiling Go Source Code to Go plugins

As introduced above the compiler tool-chain accepts different build modes via the `-buildmode` command line argument [18]. The build mode for compiling a given source file to the go plugin format is named `plugin` [18] [17, Overview].

Listing 4.1 contains a modified version of Listing 3.1, adding the compiler flags for compiling the generated source code passed via the `code` function parameter, to a go plugin. Instead of producing an executable for the target architecture and operating system the compiler now generates a shared object in the format the `plugin` package requires.

```
func invokeCompiler(code string) {
    f, _ := os.CreateTemp(".", "jit_*.go")
    defer os.Remove(f)
    f.WriteString(code)
    pre := strings.TrimSuffix(f.Name(), ".go")
    c := exec.Command(
        "go", "build", "-buildmode=plugin", "-o", pre, f.Name())
    c.Run()
}
```

Listing 4.1: Tool-chain invocation with plugin compilation

4.2 Embedding Go plugins

The loading of plugins and the resolution of plugins uses the API exposed by the previously introduced `plugin` package.

Listing 4.2 modifies Listing 4.1 for opening the previously compiled plugin. Once opened the `*plugin.Plugin` structure can be used for resolving exported functions and variables included

in the plugin. After resolving a symbol[1] its type is `any`, therefore the function `Main`[2] has to be cast to `func()` before the go type system allows a function call. Upon type casting the function is called and the generation, compilation and calling workflow of the JIT compiler is concluded.

```
func invokeCompiler(code string) {
    f, _ := os.CreateTemp(".", "jit_*.go")
    defer os.Remove(f)
    f.WriteString(code)
    pre := strings.TrimSuffix(f.Name(), ".go")
    c := exec.Command(
        "go", "build", "-buildmode=plugin", "-o", pre, f.Name())
    c.Run()

    plug, _ := plugin.Open(pre)
    // assumes generated code lives in func Main()
    symbol, _ := plug.Lookup("Main")
    Main, _ := symbol.(func())
    Main()
}
```

Listing 4.2: Plugin compilation, plugin opening and function resolution

4.3 Trade-offs, Issues and Considerations

The `plugin` package provides the program with the unique ability to allow for high performance on the fly code compilation and execution. It therefore fits the use case of a query language implementation well.

However the `plugin` package bears several downsides [17, Warnings], primarily the missing portability due to the package only supporting Linux, FreeBSD and MacOS. Another disadvantage is the strict requirement of both the host application and all plugins needing to be compiled with the same tool-chain version and build-tags - this is particularly difficult in the case of this JIT, due to the requirement of the existence of the local compiler that will most certainly not be of the exact same version as the compiler used for compiling the host application. Is the previously mentioned not strictly ensured runtime errors can occur. A further drawback is the increased difficulty of reasoning about program and plugin initialisation for the special `func init()` function is called upon opening a plugin [17, Overview], possibly opening the program up to race conditions and similar critical bugs due to global state initialisation [19, The init function].

[1] a symbol refers to a function, constant or variable

[2] for the sake of this explanation the generated code in the function parameter code is assumed to be contained in the `Main` function

5 Just in Time Compilation

Just in time compilation refers to the process of determining whether a segregated chunk of code is considered "hot"[1] and compiling this code segment into operating system and architecture specific machine code ad hoc. This machine code is then loaded into the memory of the interpreters runtime and executed instead of interpreting the code chunk [21]. The details of just in time compilation, meta tracing, categorizing code segments as "hot", improving the performance of the just in time compiler and error handling are explored in this chapter. This chapter also features the concurrent nature of the compilation pipeline while introducing type conversion, the connection and interaction of runtime and jit, and the error managment that is responsible for gracefully handling errors occuring in compiled functions.

Contrary to the previously introduced definition of a just in time compiler in the context of programming language interpreters, go does not support dynamically loading machine code into memory and executing said memory chunks. The mitigation for this is to make use of the previously introduced and explained plugin package, see chapter 4.

5.1 Meta-tracing & `JIT_CONSTANT`

```
// Function represents a function in the interpreter runtime
type Function[V any] struct {
    // ...

    // Counter stores the amount of calls made to the function
    Counter int
}
```

Listing 5.1: `Function[V any] struct` type with meta-tracing

Meta-tracing refers to the process of tracking the actions of the programming language interpreter [22, 4.1 Meta-tracing]. The interpreter uses this functionality to determine the amount of invocations of a function and updates the `Function.Counter` field accordingly, see Listing 5.1. Once this counter reaches the threshold defined in the `JIT_CONSTANT` (see Listing 5.2) the `type Function[V any] struct` instance is passed to the just in time compiler compilation queue, in which it will be compiled with other functions waiting to be compiled. Upon the Function being compiled the interpreter executes the output of the just in time compiler for each function invocation instead of walking the abstract syntax tree and thus is no longer interpreting the function, but instead uses the compiled representation.

[1] hot in the context of just in time compilation refers to a code path or a segment of code that is executed massive amount of times [20], [21]

```
// JIT_CONSTANT sets the threshold the function invocation meta tracing counter
// has to pass for the function to be considered hot and thus compilable
var JIT_CONSTANT int = 1_000
```

Listing 5.2: `JIT_CONSTANT` definition

This constant threshold varies from compiler to compiler. The value depends on the performance needs and the hit the runtime performance takes upon invoking the jit compiler. Specifics are discussed in Section 6.4.

```
type MetaDataParameter struct {
        Name string
        Type string
}
type MetaData struct {
        Parameters []MetaDataParameter
}
// Function represents a function in the interpreter runtime
type Function[V any] struct {
    // ...

    // ArgumentNames contains the list of parameter names of the function
    ArgumentNames []string
    // Name holds the name of the function
    Name string
    // MetaData holds the necessary data for the jit to compile valid functions
    MetaData *MetaData
}
```

Listing 5.3: `Function[V any] struct` type with meta data

The JIT-compiler requires some information about a function before it's being able to start the code generation step. Not only does it require the name of the function[2], but the names of its arguments and the types the JIT can use to compile the given function. The necessary fields are stored in the previously introduced `type Function[V any] struct`, specifically the `type MetaData struct` and `type MetaDataParameter struct` structures (see Listing 5.3).

The runtime fills these values upon invoking the compilation of the current function using the constructs introduced in Section 5.5, see Listing 5.4.

[2]Unnamed/anonymous functions or closures are named by prefixing a closure counter with c, the first encountered closure will therefore be compiled as `func c0()`

```
func (f *Function[V]) Eval(st Stack[V], a V) (V, error) {
    // ...
    if !f.wasJit && f.Counter >= JIT_CONSTANT && f.JitCompiler != nil &&
    ↪  f.Ast != nil {
        f.wasJit = true
        f.MetaData = &MetaData{
            Parameters: []MetaDataParameter{
                {Name: f.ArgumentNames[0], Type:
                ↪  f.JitCompiler.TypeToString(a)},
            },
        }
    }
    // ...
}
```

Listing 5.4: Computing meta data

5.2 Connecting the JIT to the Runtime

To stay consistent with the builder pattern the current runtime employs, the jit consists of a struct (see Listing 5.5). The jit can be enabled by calling the `FunctionGenerator.SetJit()` (see Section 5.3) function to create an instance of `Jit[V any] struct` and attaching to it `FunctionGenerator`.

```
type Jit[V any] struct {}
```

Listing 5.5: `Jit[V any] struct` type representing the just in time compiler

After parsing the input, the runtime attempts to walk the generated abstract syntax tree, by doing so it simultaneously generates internal representations of the corresponding nodes for interpretation.To connect the JIT to the runtime, the previously attached `type Jit[V any] struct` reference is passed to the internal representation of a function while generating interpretable structures (see Listing 5.7), previously introduced in Listing 5.1 and Listing 5.3.

```
type FunctionGenerator[V any] struct {
    // ...
    jit             *Jit[V]
    // ...
}
```

Listing 5.6: `FunctionGenerator[V any] struct` holding a reference to the just in time compiler

Listing 5.7 not only displays passing the JIT reference to the runtimes representation of a function, but also includes the initialisation of the previously mentioned meta data necessary for the compilation of a function (see Listing 5.3).

```
func (g *FunctionGenerator[V]) GenerateFunc(ast parser2.AST, gc
↪  GeneratorContext) (ParserFunc[V], error) {
        // ...
        closureFunc, err := g.GenerateFunc(a.Func, GeneratorContext{am:
        ↪  funcArgs})
        if err != nil {
            return nil, err
        }
        return func(st Stack[V], cs []V) (V, error) {
            return g.closureHandler.FromClosure(Function[V]{
                Name:          a.Name,
                ArgumentNames: args,
                JitCompiler:   g.jit,
                Counter:       0,
            }), nil
        }, nil
    //...
}
```

Listing 5.7: Passing the JIT reference to `type Function[V any] struct`

5.3 Concurrent Compilation

Once a function is called more than the amount specified in the `JIT_CONSTANT`, see Section 5.1, the jit compiler attempts to compile said function. If done procedurally, this would stall the execution of the currently compiling function by at least the amount of time the jit takes to walk the ast of the function, generate the corresponding go code for each tree node, invoke the go compiler on the generated code to compile a shared object, load the go plugin and execute the compiled function. To minimise this performance impact on the runtime imposed by the startup of the jit compiler, as well as the compilation of functions, the compilation is moved to a go routine.

To enable concurrent compilation the jit holds a buffered channel of type `chan *Function[V]`, this enables the non blocking compilation of functions and therefore falls in line with the goal of minimizing the performance impact of the compilation and the invocation itself. Furthermore the jit makes use of the `context` package to enable the cancelation of any compilations the jit currently performs by the runtime upon it exiting the interpretation of the current program. These steps are implemented in the before mentioned function `FunctionGenerator.SetJit`, see Listing 5.8.

```
func (g *FunctionGenerator[V]) SetJit() *FunctionGenerator[V] {
    ctx, cancel := context.WithCancel(context.Background())
    g.jit = &Jit[V]{
        Queue:  make(chan *Function[V], 16),
        Ctx:    ctx,
        Cancel: cancel,
    }
    go func() {
        for {
        select {
            case f := <-g.jit.Queue:
                if err := g.jit.Compile(f); err != nil {
                    log.Println("[JIT] compilation failed, skipping this
                    ↪  function", err)
                }
            case <-g.jit.Ctx.Done():
                return
            }
        }
    }()
    return g
}
```

Listing 5.8: Invoking the JIT and its concurrent compilation

In Listing 5.5 the above explained constructs necessary for the concurrent compilation of functions were omitted, Listing 5.9 therefore extends Listing 5.5 to include said fields.

```
type Jit[V any] struct {
        Queue  chan *Function[V]
        Ctx    context.Context
        Cancel context.CancelFunc
}
```

Listing 5.9: `Jit[V any] struct` type with concurrency constructs

The consumer of the queue channel the jit compiler holds, see Listing 5.9, can be seen in Listing 5.8. The producer is defined in the `*Function[V].Eval` function. This function is invoked by the runtime for every invocation of said function. The runtime uses this function to handle meta tracing, such as the invocation counter and adding a function to the compilation queue, see Listing 5.10.

```
func (f *Function[V]) Eval(st Stack[V], a V) (V, error) {
    if f.wasJit && f.JitFunc != nil {
        // compiled function gets invoked here
    }

    if !f.wasJit && f.Counter >= JIT_CONSTANT && f.JitCompiler != nil &&
    ↪  f.Ast != nil {
        f.wasJit = true
        f.JitCompiler.Queue <- f
    }

    f.Counter++
    st.Push(a)
    return f.Func(st.CreateFrame(1), nil)
}
```

Listing 5.10: `Function[V].Eval` and queuing functions for compilation

5.4 Function Parameters and erasing Type Information

To enable the runtime calling the just in time compiled function, the signatures generated and compiled into the go plugin by the just in time compiler and the go compiler tool chain have to match. If not, the go runtime will invoke a panic upon failing to cast the function. The process of removing type information from parameters, variables and constants is often referred to as *type erasure* [23, A.2 Type erasure]. The jit performs this for both input and output values from the compiled function, the resulting type the symbol contained in the go plugin is `func(...any) (any, error)`. The function accepts variadic parameters and thus allows the runtime to pass zero or more parameters into it, therefore supporting zero, one and multiple arguments to a compiled function. To specifically access the passed in arguments the just in time compiler code generation step has to insert index accesses into the parameter list. Argument name lookup is implemented by using the value of the `Function[V].ArgumentNames` field of Listing 5.3 at the current index. Just in time compiling a function with two arguments, such as Listing 5.11, therefore results in the generated go code shown in Listing 5.12.

```
func twoArguments(a, b)
    a+b;
```

Listing 5.11: Exemplary function with multiple arguments

```
func JIT_twoArgs(args ...any) (any, error) {
    a := args[0]
    b := args[1]
    return a+b, nil
}
```

Listing 5.12: Go code generated for exemplary function with multiple arguments

However the function in Listing 5.12 will not be be accepted by the go compiler upon invocation for the variables `a` and `b` are both of type `any`, thus not supporting arithmetic operations, such as the shown addition. Typecasting both `a` and `b` to either `float64`, `float32` or any `int` like type is necessary to enable compilation. To determine the correct type for `a` and `b` the before introduced meta tracing is especially useful. This meta tracing requires the translation of the languages type system into the go type system.

5.5 Type System Clashes

To enable the tracing of types and therefore casting function parameters to their correct type. The jit compiler employs three functions that are attached to the `type Jit[V any] struct` type, see Listing 5.13.

```
type Jit[V any] struct {
        TypeToString func(V) string
        ValueToUnderlying func(V) any
        UnderlyingToValue func(any) V
}
```

Listing 5.13: `Jit[V any] struct` type with type system conversion helpers

All of these functions are defined when creating the jit compiler. `TypeToString` is used to solve the afore introduced issue for casting function parameters to their respective types. Listing 5.11 is therefore no longer compiled to Listing 5.12, but instead includes parameter type casts, see Listing 5.14. Listing 5.15 shows the implementation used for type casts in the generated go code with the `value` language dialect introduced in Section 2.1.

```
func JIT_twoArgs(args ...any) (any, error) {
    a := args[0].(float64)
    b := args[1].(float64)
    return a+b, nil
}
```

Listing 5.14: Go code generated for exemplary function with parameter type casts

The JIT compiler uses the go type system because it generates go code. The compiled code has no knowledge of the type system employed in the abstract language the JIT computes its

output from. To allow for passing abstract types into just in time compiled functions the jit holds the `ValueToUnderlying` function. This function converts the aforementioned custom defined types to types the go type system and compiler accept. Furthermore it's used to convert all incoming function parameters of a JIT compiled function. `UnderlyingToValue` is the counterpart to `ValueToUnderlying`, it converts all values returned from just in time compiled functions from the go type to their abstract language object system type.

```
jit := &Jit{
    // ...
    TypeToString: func(v value.Value) string {
        switch v.(type) {
        case value.String:
            return "string"
        case value.Float, value.Int:
            return "float64"
        case value.Bool:
            return "bool"
        case value.Map:
            return "map[string]any"
        default:
            return "any"
        }
    }
}
```

Listing 5.15: `TypeToString` examplary implementation

Considering Listing 5.11, the JIT uses meta tracing, as introduced in Section 5.1, to determine the types of the variables `a` and `b`. The runtime uses type aliases, called `type Float float64` for floating point integers, `type Int int` for integers and `type String string`. Each abstracting upon the go type and thus making it incompatible with the standard `float64`, `int` and `string` type. The JIT is implemented in a generic way and does therefore not have any knowledge of this, as introduced before.

`ValueToUnderlying` (λ) is used to convert the values of the function arguments to their values, before passing the converted type information to the function (f). After the jit compiled function returns a value, `UnderlyingToValue` (γ) is used to convert the value to the abstract object system the language is using (T), Figure 5.1 formalises this type conversion process, while Listing 8.2 and Listing 8.3 show excerpts of the function implementations of the `ValueToUnderlying` and `UnderlyingToValue` for the runtime dialect introduced in Section 2.1.

$$T \to \lambda(T) \to f(\lambda(T)) \to \gamma(f(\lambda(T))) \to T \tag{5.1}$$

Figure 5.1: Formalising type conversion for function parameter and return types

5.6 Invoking a compiled Function

```
func (f *Function[V]) Eval(st Stack[V], a V) (V, error) {
    if f.wasJit && f.JitFunc != nil {
        out, _ := f.JitFunc(f.JitCompiler.ValueToUnderlying(a))
        // ...
        return out, nil
    }
    // ...
}
```

Listing 5.16: Invoking a compiled function

Listing 5.16 displays the invocation of a previously compiled function. The `ValueToUnderlying` function introduced in Section 5.5 is used to convert the parameter passed into the function. While the output of the compiled function is converted to the language object system when compiling the function and looking up the symbol in the plugin, as introduced in Section 4.2 and further explained in Section 5.8.

5.7 Bailing out to the Interpreter

The JIT can be understood as a separate stage of the evaluation of a given input. An interpreter at first the input is split into tokens, this process is referred to as lexical analysis. The following step is composed of parsing the tokens computed by the previous step and forming an abstract syntax tree, in which each node represents an executable entity, such as a constant, control flow or function calls. The last stage is to evaluate the previously built abstract syntax tree by walking each node and executing it [24]. A just in time compiler differs from these stages by being invoked parallel to the runtime executing the input and being able to stop the execution of a compiled function and exiting out of the compiled function back into the runtime evaluation of the function, while discarding the compiled function. This process of discarding compiled code and passing the task of executing a function back to the runtime upon encountering an error, is commonly described as a *bailout* [25]. This process is implemented in the previously introduced `(*Function).Eval()` function.

`type Function[V any] struct` is the internal representation of a function in the runtime and the abstract syntax tree and holds a reference to the compiled form of the function. Listing 5.17 shows the `Eval()` function. The compiled version of the function is discarded of, once the execution of the compiled function yields an error. Furthermore the counter of the meta tracing logic for marking a function as compilable is reset to 0 and the flag for marking a function as compiled by the jit is set to `false`. The last step taken, is to jump to the `bailout` label and thus restart the execution of the current function, but this time in the runtime, not in the compiled form of the function.

```
func (f *Function[V]) Eval(st Stack[V], a V) (V, error) {
    if f.wasJit && f.JitFunc != nil {
        out, err := f.JitFunc(f.JitCompiler.ValueToUnderlying(a))
        if err != nil {
            f.JitFunc = nil
            f.wasJit = false
            f.Counter = 0
            goto bailout
        }
        return out, nil
    }

    // ...

bailout:
    f.Counter++
    st.Push(a)
    return f.Func(st.CreateFrame(1), nil)
}
```

Listing 5.17: Bailing out of the jit context to the runtime upon encountering an error

5.8 Compiler pipeline

As previously introduced, the runtime invokes the just in time compiler for a given function once the compilation criteria are met, in this case the compilation threshold. This invocation happens in the form of adding the function to the just in time compilation queue.

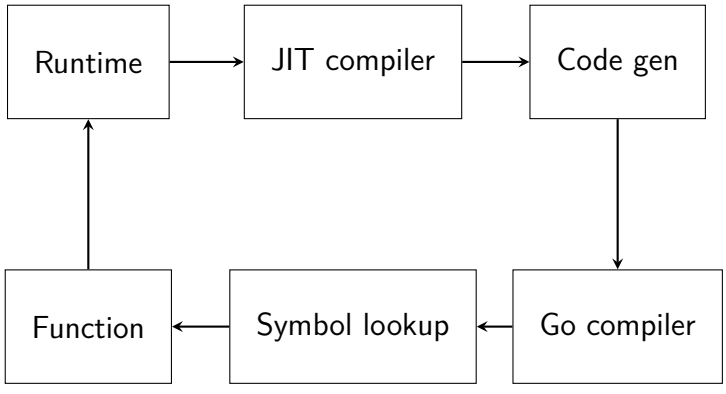

Figure 5.2: Compilation pipeline

The just in time compiler proceeds to pass the accepted function and the meta data for its returning types along with the passed in arguments to the code generation step and converts the abstract syntax tree nodes for the function into their corresponding go representation. After the code generation step of the compilation pipeline, the go toolchain, including the compiler, is invoked and thus creates a shared object, generally named a go plugin [17]. Once

this plugin is compiled, the symbol lookup stage denotes the process of loading the plugin and performing a lookup for the compiled function as a non typed symbol. The last step of this lookup stage is the type conversion for the return value of the compiled function as well as type casting the whole function to a function accepting and returing values which types are erased. After this the function is regarded as compiled and ready to be attached to the runtime and executed once the runtime invokes the function again.

Figure 5.2 visualizes the above while highlighting the dependence of the steps in the JIT compilation pipeline. The compiler pipeline is seperate from the runtime and therefore does not interfer with the execution of a function in the runtime of the programming language. This also enables the parallelisation of the compilation processes, such as moving the code generation, the go compiler tool chain invokation and the symbol lookup into a go routine.

6 Benchmarks

The following sections will measure the impact the jit compilation has on selected workloads.

Benchmark results may be subject to various influences, including the workload on the system conducting the benchmarks, insufficiently sized data sets for comprehensive and accurate testing, as well as comparing inherently dissimilar benchmarks. Most of the aforementioned can be mitigated by using the `testing` package included in the go programming languages standard library [26].

To avoid the influences of the current workload of the system the tests are performed many times, therefore accounting for statistical outliers and external system influences on the test, which furthermore accounts for the possibility of choosing insufficiently sized data sets.

By benchmarking the execution of the same given input with the JIT-compiler enabled and with the JIT-compiler disabled an inherently comparable data set is created due to the shared purpose of evaluating the runtime performance under differing runtime configurations. Therefore, conducting benchmarks under the two previously mentioned configurations and comparing the results is a valid evaluation of the two benchmarks.

The benchmarks simulate a hot path by executing a given operations for a given iteration count and running the benchmark itself multiple times using the command line benchmarking tool *hyperfine*.

6.1 Arithmetics

```
func b(a)
    a*a/25*a-12+a/a*a*a*a/25*a-12+a/a*a*a*a/25*a
    -12+a/a*a*a*a/25*a-12+a/a*a*a*a/25*a-12+a/a*a
    *a*a/25*a-12+a/a*a*a*a/25*a-12+a/a*a*a*a/25
    *a-12+a/a*a*a*a/25*a-12+a/a*a*a*a/25*a-12+a/a
    *a*a*a/25*a-12+a/a*a*a*a/25*a-12+a/a*a;
let s = list(100000).map(b).sum();
s
```

Listing 6.1: Heavy load arithmetic operations

Benchmarking the performance of arithmetic operations allows for a first execution efficiency evaluation of the language runtime. The benchmark using Listing 6.1 results of up to 14.51x

improvement (0.94s instead of 13.64s). While the original runtime scales linearly and proportionally to the input, the runtime supported by the JIT-compiler scales at an almost constant rate, see Figure 6.1 and Table 6.1.

Iterations	Mean Execution Time	Mean Execution Time (JIT)	Δ	Improvement
100k	00.28s	0.20s	00.08s	01.40x
500k	01.37s	0.26s	01.11s	05.27x
1mio	02.73s	0.34s	02.39s	08.03x
5mio	13.64s	0.94s	12.70s	14.51x

Table 6.1: Arithmetic operations benchmark results

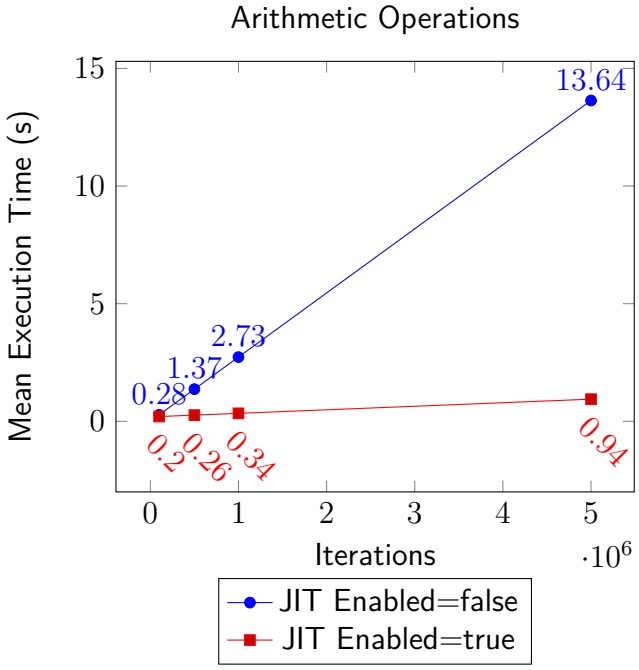

Figure 6.1: Benchmark: Arithmetic operations

6.2 String operations

```
func b(a)
    a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a+a;
let s = list(100000).map(e->e.string()).map(b).size();
s
```

Listing 6.2: Heavy load string concatenating

Merging strings is an often used language feature, therefore making the need for high efficiency evident. This benchmark aims to simulate a real world use with heavy load, similar to the benchmark performed before. Both Table 6.2 and its visualisation Figure 6.2 show the

performance improvement of at least 1.43x and at most 4.59x - resulting in a mean delta, comparing the current runtime and the runtime enhanced with the JIT, of at most 11.08s and at least 0.10s.

Iterations	Mean Execution Time	Mean Execution Time (JIT)	Δ	Improvement
100k	00.33s	0.23s	00.10s	1.43x
500k	01.57s	0.48s	01.09s	3.27x
1mio	02.99s	0.77s	02.22s	3.89x
5mio	14.17s	3.09s	11.08s	4.59x

Table 6.2: String concatenation benchmark results

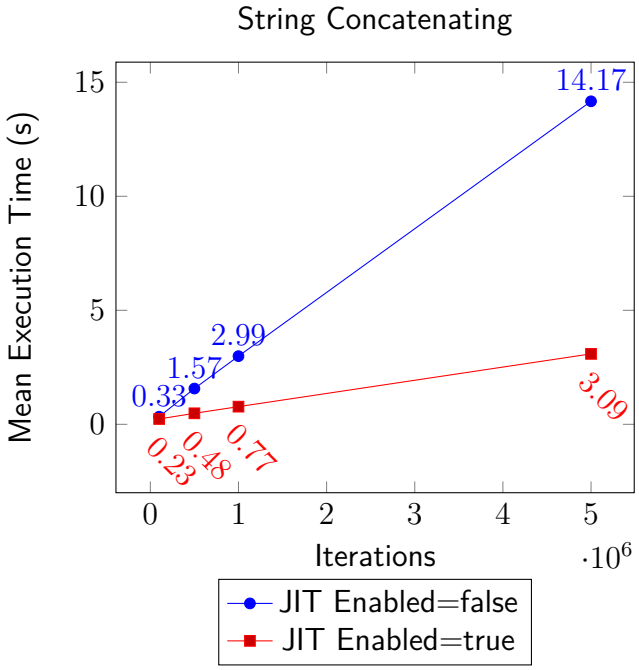

Figure 6.2: Benchmark: String concatenating

6.3 Real world workloads

Contrary to the before examined language features, this benchmark focusses on common operations performed on large datasets, such as aggregating values, reading and overwriting data, chaining list operations and working with lists of objects. Due to the mixture of operations featured in this benchmark the resulting numbers are a combination of different factors.

The improvements for smaller iterations, such as 100 thousand and 500 thousand, can be considered negligible. For the larger iterations the JIT improves the performance by at least 0.3s (1.24x) and at most 2.42s (1.431x).

```
list(100000)
    .filter(e -> e >= 1)
    .map(e -> ((1024*(1024-e))/(e+1024))+(e*e+e*e-e*e+e*e))
    .map(e -> {Key: "Number", Value: e})
    .map(e -> e.Value)
    .map(e -> e.string().len().string())
    .map(e -> e+e+e+e+e+e+e)
    .last()
```

Listing 6.3: Real world heavy load benchmark

The reasoning for the smaller performance improvements compared to the previous benchmarks is the invocation of methods defined on types by the specific language dialect used, such as `(int).string().len().string()`. The just in time compiler does not posses any knowledge of these methods defined on type aliases in the language type system and therefore stops the code generation for closures and functions containing method calls.

Iterations	Mean Execution Time	Mean Execution Time (JIT)	Δ	Improvement
100k	0.18s	0.18s	0.00s	0.000x
500k	0.87s	0.82s	0.05s	1.061x
1mio	1.68s	1.36s	0.32s	1.240x
5mio	8.04s	5.62s	2.42s	1.431x

Table 6.3: Real world heavy load benchmark results

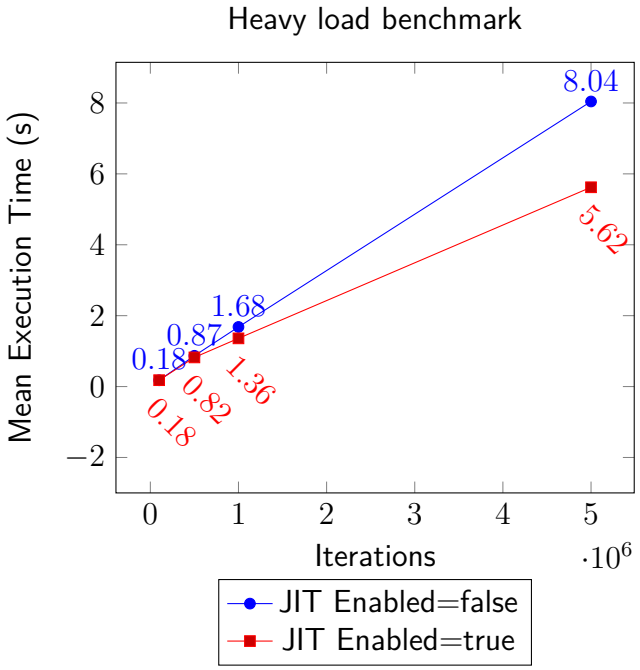

Figure 6.3: Benchmark: Real world heavy load benchmark

6.4 Determining the JIT_CONSTANT

The JIT_CONSTANT, introduced in Section 5.1, is used to establish a optimal relation between the possible performance improvements and the negative performance impact of invokation of the compilation of a function. Therefore the threshold acts as the criteria for categorizing a given function as a compilation target for the jit compiler. To find the optimal relation between compiling a function too early which could result in degraded overall performance or compiling a function too late, which limits the JIT from achiving its full potential.

The benchmark to determine the aforementioned value is conducted by executing the benchmark representing the real world workload, shown in Listing 6.3, with differing values ranging from a thousand to $500,000$ and recording the delta between the runtime with and without the JIT enabled for each JIT_CONSTANT value. Section 6.4 displays the results of the measurements.

Constant	Iterations	Mean Execution Time	Δ
Off	100k	0.18s	0.00s
	500k	0.87s	0.00s
	1mio	1.68s	0.00s
	5mio	8.04s	0.00s
1,000	100k	0.18s	0.00s
	500k	0.82s	0.05s
	1mio	1.36s	0.32s
	5mio	5.62s	2.42s
10,000	100k	0.18s	0.00s
	500k	0.83s	0.04s
	1mio	1.36s	0.32s
	5mio	5.57s	2.47s
100,000	100k	0.18s	0.00s
	500k	0.85s	0.02s
	1mio	1.42s	0.26s
	5mio	5.64s	2.40s
250,000	100k	0.18s	0.00s
	500k	0.86s	0.01s
	1mio	1.51s	0.17s
	5mio	5.79s	2.25s
500,000	100k	0.18s	0.00s
	500k	0.86s	0.01s
	1mio	1.63s	0.05s
	5mio	5.93s	2.11s

Table 6.4: JIT_CONSTANT benchmark result

Figure 6.4 notates the importance of letting the runtime execute functions that aren't executed often. This is underlined by the results of $100,000$ function invocations. In this case the runtime is as fast as the JIT, therefore the startup cost is not justifiable.

The first differences in total runtime performance can be observed for $500,000$ function calls. The runtime starts to slow down in comparison to the JIT results. Smaller JIT_CONSTANT

values result in approximately the same values. However, Most of these values are faster than the runtime.

Signification speed-ups can be identified once the invocation count exceeds the one million mark. The results for the `JIT_CONSTANT` values 1000 and $10,000$ both enable execution time savings of around 0.32 seconds or 320ms. All larger values for the `JIT_CONSTANT` result in less substantial improvements.

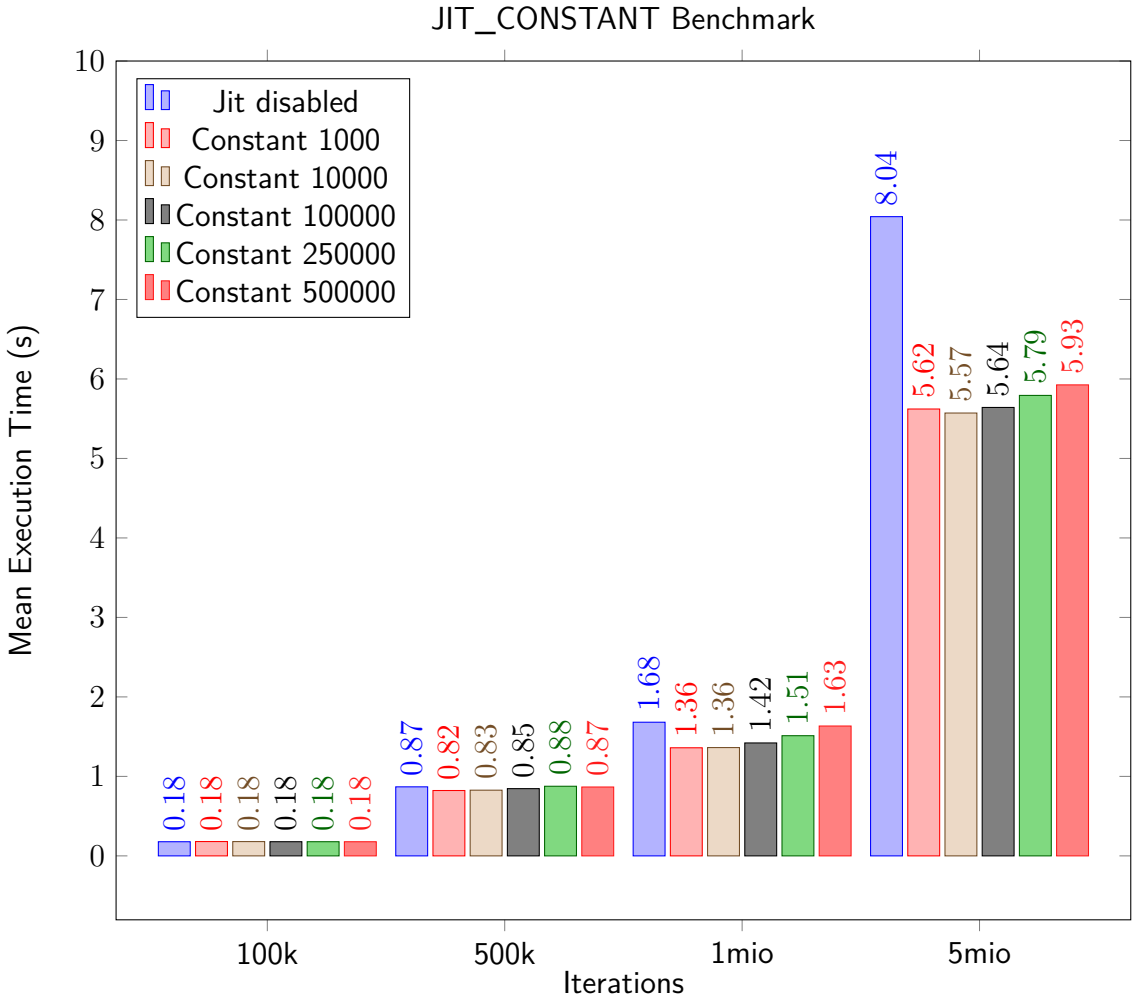

Figure 6.4: Benchmark: `JIT_CONSTANT` benchmark

Due to the nature of a JIT compiler, it targets code chunks that are executed more often. Therefore justifying the start up of the just in time compilation and the compilation cost of a function. To test this intent, the last measured iteration count was chosen to be five million iterations. This substantiates the previous observations of lower values for the `JIT_CONSTANT` resulting in larger performance wins. For five million operations, all JIT compiled tests resulted in at least a saving of 2.11s or 2110ms. The largest improvements are surveyed for the `JIT_CONSTANT` value of $10,000$. Compared to the runtime this value enabled the improvement of 2.47s or 2470ms, resulting in a runtime reduction of 30.72% or reducing the time taken to execute the function five million times 1.443 times.

The conclusion of these benchmark is to change the `JIT_CONSTANT` from its previously set value of $1,000$ to the determined optimal value of $10,000$.

7 Conclusion

To evaluate wheter the go plugin api is feasible for just in time compilation one must consider the usability and robustness of the resulting implementation. Furthermore it is crusial to take the resulting performance impact of the solution into account, while not neglecting the complexity of the implementation. The following section includes in depth examination of the above criteria to conclude on the aforementioned consideration and introduces other approaches to improve the performance of the runtime.

7.1 Usability and Robustness

The largest limiting usability factor is the requirement to have a local copy of the go toolchain available. This makes user interaction in the way of installing the go compiler necessary.

The just in time compiler is engineered to be robust and not interfere with the runtime when encoutering issues while generating code, invoking the tool-chain or collecting meta data. Therefore the runtime does not halt execution upon failing to call a just in time compiled function or if the just in time compilation of a function fails. The runtime and just in time compiler interaction allow for a seamless transition between compiled funciton and interpreted function if necessary, such as after compiling a function or bailing out of a compiled funciton. This is possible due to the inherently concurrent nature of the just in time compilation pipeline.

7.2 Performance

Based on performed benchmarks, their results and visual representation (see chapter 6), the performance impact of the JIT-Compiler is substantial for most workloads, but especially for larger iterations, such as in hot paths. The Go compiler optimises compiled plugins as if it were to compile a Go application, such as reducing memory allocations[27, Escape Analysis], inlining functions[27, Inlining] and not allocating empty structures [27, Interface Values]. This allows the interpreter to leverage the optimized performance the Go compiler applies to binaries it compiles, thus enabling the observed large performance improvements for string manipulation, arithmetic operations and the less substantial improvements shown for the real world benchmarks.

However the overhead of walking the tree to generate go code from each node, writing this code to a file, invoking the local go compiler via the `os/exec` package, loading the go plugin and looking the compiled function up, can not be neglected. Due to said influences Jit compilation is inherently unsuitable for small workloads and fast executing inputs, as seen on the workloads, inputs and iteration counts choosen for the benchmarks.

7.3 Implementation Complexity

Due to the architecture of the runtime, the abstract syntax tree for a given node is available while evaluating said node. The just-in-time compiler can therefore use this tree to generate go code for every encountered node. Furthermore the just-in-time compiler uses the existing runtime facilities for performing meta tracing and asserting types. While the runtime architecture is advantageous for the above, it made designing and implementing the just-in-time compiler itself complex. There are several reasons for this, foremost the package architecture, the just-in-time compiler has to depend on multiple packages and use type assertions for custom types defined in subpackages, thus it can not be extracted into a `jit` package. Furthermore the runtime was designed to be used with a generic language dialect and using the Go generics feature. This made it especially complicated to implement type assertions and type conversion for the generated go code, this is the reason for the introduction of the functions presented in Section 5.5 and the need for including the implementation of these functions when defining a language dialect for the runtime.

7.4 Differing Approaches & Future Work

While a just in time compiler can substantially improve the performance of a runtime for long running applications and hot paths, there are a lot of different approaches to improve a runtime. These approaches often do not introduce the large technical debt of having to maintain a code generation backend for multiple architectures or the need for a code generation step as a whole, contrary to a just in time compilation pipeline. These possible performance improvement strategies also often do not take a large toll on the runtime itself to collect meta data of its execution or start a thread responsible of compiling functions.

An interpreter that walks the abstract syntax tree generated by the parsing of the input is commonly denoted as the naive implementation of an interpreter. Most interpreters implement a bytecode compiler and a virtual machine executing the compiled bytecode. This reduces the amount of data to keep in memory while switching the evaluation model from traversal to an iteration of a list of bytes. This solution is often combined with a JIT compiler generating machine specific instructions comparable to the byte code operators. Furthermore a byte code interpreter allows for fast variable storage in virtual registers or in a stack based storage solution [28], [29, 1. Introduction] and [30, Introduction].

Bibliography

[1] "The sqlite query optimizer overview." (), [Online]. Available: https://www.sqlite.org/optoverview.html (visited on 01/20/2024).

[2] "The next-generation query planner." (2023), [Online]. Available: https://www.sqlite.org/queryplanner-ng.html (visited on 01/20/2024).

[3] A. Krall, "Efficient javavm just-in-time compilation," in *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*, IEEE, 1998, pp. 205–212. [Online]. Available: https://www.complang.tuwien.ac.at/andi/papers/pact_98.pdf.

[4] "Chapter 32. just-in-time compilation (jit)." (), [Online]. Available: https://www.postgresql.org/docs/current/jit-reason.html (visited on 01/20/2024).

[5] T. Suganuma, T. Yasue, and T. Nakatani, "A region-based compilation technique for a java just-in-time compiler," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, 2003, pp. 312–323.

[6] "Parser2.optimizer." (2023), [Online]. Available: https://github.com/hneemann/parser2/blob/361d9bd2f6daf0ff76b47a1186ed2cfcd1a96d12/parser2.go#L23 (visited on 01/20/2024).

[7] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, *et al.*, "Design, implementation, and evaluation of optimizations in a just-in-time compiler," in *Proceedings of the ACM 1999 conference on Java Grande*, 1999, pp. 119–128.

[8] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, "Jit-picking: Differential fuzzing of javascript engines," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 351–364.

[9] K. Ishizaki, M. Takeuchi, K. Kawachiya, T. Suganuma, *et al.*, "Effectiveness of cross-platform optimizations for a java just-in-time compiler," in *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, 2003, pp. 187–204.

[10] J. Aycock, "A brief history of just-in-time," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 97–113, 2003.

[11] H. Neemann, *Parser2*, 2024. [Online]. Available: https://github.com/hneemann/parser2 (visited on 04/16/2024).

[12] xnacly and dani0611, *Parser2*, 2024. [Online]. Available: https://github.com/xnacly/parser2 (visited on 04/16/2024).

[13] I. L. Taylor. "Type parameters proposal." (2021), [Online]. Available: https://go.googlesource.com/proposal/+/HEAD/design/43651-type-parameters.md (visited on 04/02/2024).

[14] "Gc.go." (2023), [Online]. Available: https://github.com/golang/go/blob/master/src/cmd/go/internal/work/gc.go#L58 (visited on 01/20/2024).

[15] "Internal directories." (2024), [Online]. Available: https://pkg.go.dev/cmd/go#hdr-Internal_Directories (visited on 01/20/2024).

[16] "Os/exec package." (2024), [Online]. Available: https://pkg.go.dev/os/exec (visited on 01/20/2024).

[17] "Plugin package." (2024), [Online]. Available: https://pkg.go.dev/plugin (visited on 01/20/2024).

[18] "Build modes." (2024), [Online]. Available: https://pkg.go.dev/cmd/go#hdr-Build_modes (visited on 01/20/2024).

[19] "Effective go - the go programming language." (2024), [Online]. Available: https://go.dev/doc/effective_go (visited on 01/20/2024).

[20] "-xjit / -xnojit - ibm documentation." (Feb. 12, 2024), [Online]. Available: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=options-xjit#optlevel (visited on 04/16/2024).

[21] A. Krall, "Efficient javavm just-in-time compilation," in *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*, IEEE, 1998, pp. 205–212.

[22] C. F. Bolz and L. Tratt, "The impact of meta-tracing on vm design and implementation," *Science of Computer Programming*, vol. 98, pp. 408–421, 2015.

[23] K. Crary, S. Weirich, and G. Morrisett, "Intensional polymorphism in type-erasure semantics," *Journal of Functional Programming*, vol. 12, no. 6, pp. 567–600, 2002. [Online]. Available: https://www.cambridge.org/core/services/aop-cambridge-core/content/view/B46353534BE6D97919C442C4BD326845/S0956796801004282a.pdf/intensional-polymorphism-in-type-erasure-semantics.pdf (visited on 05/13/2024).

[24] A. Hippisley, "Lexical analysis.," *Handbook of natural language processing*, vol. 2, pp. 31–58, 2010.

[25] J. Aycock, "A brief history of just-in-time," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 97–113, 2003.

[26] "Go testing package." Official documentation for the Go testing package. (), [Online]. Available: https://pkg.go.dev/testing (visited on 04/16/2024).

[27] "Go wiki: Compiler and runtime optimizations." (), [Online]. Available: https://go.dev/wiki/CompilerOptimizations (visited on 05/03/2024).

[28] R. McIlroy. "Firing up the ignition interpreter." (2016), [Online]. Available: https://v8.dev/blog/ignition-interpreter (visited on 06/01/2024).

[29] W. C. Roberto Ierusalimschy Luiz Henrique de Figueiredo. "Lua 5.3 reference manual." (2020), [Online]. Available: https://www.lua.org/manual/5.3/manual.html (visited on 06/01/2024).

[30] P. S. Foundation. "Python developer's guide - the bytecode interpreter (3.11)." (2024), [Online]. Available: https://devguide.python.org/internals/interpreter (visited on 06/01/2024).

8 Appendix

```
type Person struct {
        Name         string
        Surname      string
        PlaceOfBirth string
        Age          int
}

var Persons = []Person{
        {"John", "Doe", "London", 23},
        {"Jane", "Doe", "London", 25},
        {"Bob", "Smith", "New York", 21},
        {"Frank", "Muller", "New York", 22},
        {"Mary", "Green", "Seattle", 21},
        {"Jake", "Muller", "Washington", 22},
}

var PersonToMap = value.NewToMapReflection[Person]()
var persons = value.NewListOfMaps[Person](PersonToMap, Persons)
```

Listing 8.1: Passing Go values into the language context - source [12]

```
func toUnderlying(v value.Value) (o any) {
    switch t := v.(type) {
        case value.Bool:
            o, _ = t.ToBool()
        case value.Int:
            to, _ := t.ToInt()
            o = float64(to)
        case value.Float:
            o, _ = t.ToFloat()
        case value.String:
            o, _ = t.ToString(funcGen.Stack[value.Value]{})
        case value.Map:
            m := make(map[string]any, t.Size())
            t.Iter(func(key string, v value.Value) bool {
                m[key] = toUnderlying(v)
                return true
            })
            o = m
        default:
            panic(fmt.Sprintf("%T conversion to underlying type not
            ↪  supported by jit", t))
    }
    return
}
jit := &Jit{
    // ...
    ValueToUnderlying: toUnderlying,
}
```

Listing 8.2: `ValueToUnderlying` implementation

```
func toValue(v any) value.Value {
    switch t := v.(type) {
        case int:
            return value.Int(t)
        case float64:
            return value.Float(t)
        case bool:
            return value.Bool(t)
        case string:
            return value.String(t)
        case map[string]any:
            m := make(value.RealMap, len(t))
            for k, v := range t {
                m[k] = toValue(v)
            }
            return value.NewMap(m)
        default:
            panic(fmt.Sprintf("%T conversion to high level type not
             ↪  supported by jit", t))
    }
}
jit := &Jit{
    // ...
    UnderlyingToValue: toValue,
}
```

Listing 8.3: `UnderlyingToValue` implementation