Abstract

Jsonnet is an open-source programming language that helps to automatically create software configuration. Interfacing with other languages could allow *Jsonnet* programmers use popular existing libraries, and in some cases improve performance as well. Although currently there are ways to achieve this in *Jsonnet*, they require deeper knowledge about the interpreter, and are not consistent with the core concepts of the language. This thesis describes the process of designing and implementing a mechanism for calling functions written in other programming languages from *Jsonnet*, using *WebAssembly* as the intermediate layer between the interpreter and the library. The reference implementation is then tested using diagnostic tools to check for bugs, verify the assumptions made earlier in the process, assess the memory and computation time overhead on short programs, and to confirm that for expensive operations a speed-up can indeed be measured. The final product is portable, fast, and tailored to the needs of *Jsonnet*. The results of the development done as part of this research could be merged into the *Jsonnet* project or serve as a proof of concept to inform decisions about a foreign function interface in the future.

Keywords

Jsonnet, foreign function interface, WebAssembly, Rust, C

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

Software and its engineering
Software notations and tools
Software configuration management and version control systems

Tytuł pracy w języku polskim

Dodanie w języku Jsonnet wsparcia dla wywoływania funkcji napisanych w innych językach

Contents

1. Introduction . . . 7
- 1.1. Foreign function interface . . . 7
- 1.2. *Jsonnet* . . . 7
  - 1.2.1. The *go-jsonnet* interpreter . . . 7
  - 1.2.2. Motivations for a foreign function interface in *Jsonnet* . . . 7
  - 1.2.3. Requirements for a foreign function interface in *Jsonnet* . . . 8
  - 1.2.4. Native functions . . . 8
- 1.3. *WebAssembly* . . . 9
  - 1.3.1. WASI . . . 9
- 1.4. *WebAssembly* runtime . . . 9
- 1.5. Libary languages – *Rust* and *C* . . . 10
- 1.6. BSON . . . 10
- 1.7. Diagnostic tools . . . 10
  - 1.7.1. *Valgrind* . . . 10
  - 1.7.2. The *strace* program . . . 11
  - 1.7.3. The *Heaptrack* program . . . 11

2. Solution . . . 13
- 2.1. *WASM* library format . . . 13
  - 2.1.1. Design decisions . . . 13
  - 2.1.2. Suggested format . . . 13
  - 2.1.3. Rationale . . . 14
- 2.2. Interpreter implementation . . . 15
  - 2.2.1. Syntax . . . 15
  - 2.2.2. Lifetime of an exported function . . . 16
  - 2.2.3. Caching . . . 16
- 2.3. Rust support . . . 17
  - 2.3.1. *Rust* attribute macro usage . . . 18
- 2.4. *C* support . . . 19
  - 2.4.1. *C* macros and functions usage . . . 20
- 2.5. Preventing memory leaks . . . 20
  - 2.5.1. Testing leaks in *Rust* . . . 21
  - 2.5.2. Testing leaks in *C* . . . 22
- 2.6. Requirements . . . 23
  - 2.6.1. Pure functions . . . 23

3. Efficiency . . . 25
3.1. Speed improvement . . . 25
3.2. Library initialisation overhead . . . 26
3.3. Function execution overhead . . . 28
3.3.1. Language switch overhead . . . 28
3.3.2. Argument serialisation and result deserialisation overhead . . . 30
3.4. Memory overhead . . . 30
3.4.1. Peak memory usage . . . 31
3.5. *Wasmtime* vs *Wasmer* . . . 33

4. Conclusion . . . 35
4.1. Areas for improvement . . . 35
4.1.1. Laziness, functions as values . . . 35
4.1.2. Side effects . . . 36
4.1.3. Language support . . . 36
4.2. Merge status . . . 36

A. Examples illustrating the proposed format of a *WebAssembly* library exported to *Jsonnet* . . . 37
A.1. *Rust* file . . . 37
A.2. *WebAssembly* module . . . 37
A.3. Intermediate *BSON* documents . . . 38
A.3.1. Result *BSON* of the `__jsonnet_internal_meta_add` function . . . 38
A.3.2. Argument *BSON* of the `__jsonnet_export_add` function . . . 38
A.3.3. Result *BSON* of the `__jsonnet_export_add` function . . . 38

Bibliography . . . 39

List of Figures

2.1. Lifetime of an exported function 17

3.1. *SHA-1* hash calculation time comparison 26
3.2. Library initialisation overhead 27
3.3. Function execution overhead measured with *strace* 29
3.4. Argument evaluation and serialisation time depending on number of parameters in a function 31
3.5. *Heaptrack – Jsonnet FFI* memory consumption over time 32
3.6. *Heaptrack – Jsonnet FFI* flame graph of the allocated memory during peak memory usage 32
3.7. Import time of a basic library in *Jsonnet* depending on the runtime and the *WASM* compiler 34

Chapter 1

Introduction

1.1. Foreign function interface

A foreign function interface is mechanism thanks to which a programming language can call functions implemented in a different programming language. Notable examples include the *Java Native Interface (JNI)* [1] and *cffi* [2], which enable running *C/C++* libraries from *Java* and *Python* respectively. Using an *FFI* can be dictated by the fact that there is an existing library which would be inconvenient or impractical to rewrite, or by performance requirements (for example, some part of a *Python* program needs to be written in *C*, because otherwise it would be too slow).

1.2. *Jsonnet*

Jsonnet [3] is a functional data-templating language used to manage system configuration. It is a superset of *JSON* [4], which means that existing *JSON* files can be incrementally simplified by incorporating various *Jsonnet* language constructs. Its features include variables, pure functions (functions without side effects like I/O), conditional expressions, and imports. That way a single big *JSON* object can be split into multiple smaller, more readable *Jsonnet* files, where repetition is avoided by extracting values into functions and variables. When writing a complicated *JSON* file one approach is to do it entirely manually. A different approach would be to prepare a script in a language like *Python. Jsonnet* lies somewhere in between. On the one hand, you can immediately easily see the structure of the object, as you would with a plain *JSON* file, on the other, you can perform computation, as in a regular programming language.

Jsonnet features in projects like the popular analytics web application Grafana [5], as well as the configuration management tool Kapitan (formerly developed by DeepMind [6]).

1.2.1. The *go-jsonnet* interpreter

There are two official *Jsonnet* interpreters, an older one written in *C++* and a newer one in *Go* (*go-jsonnet* [7]). The goal of this thesis was to add a foreign function interface to the latter one. It is more actively maintained and may become the only supported interpreter in the future [8].

1.2.2. Motivations for a foreign function interface in *Jsonnet*

There are several reasons why *Jsonnet* would benefit from having a foreign function interface.

First, there is a lot of useful existing code written in mainstream programming languages. The *Jsonnet* standard library does not address all the needs a programmer might encounter while writing configuration files. Things like cryptographic functions (for example *SHA-2*), conversion to or from non-standard (meaning not supported by *Jsonnet*) file formats, or advanced mathematical utilities are not directly included in *Jsonnet*, but are implemented and well tested in *Rust* or *C*.

Second, running code close to native speed could reduce the time needed to generate configuration files. In particular, some parts of the standard library could be reimplemented in a language that is closer to the bare metal. Although configurations usually do not require to be generated very frequently and performance is usually not a problem, in some cases, where the system is very big, the compile-times may become impractical. The issue of the inefficiency of the standard library, among other things, was raised by the *Databricks* company in their blog [9].

Third, sometimes it may be convenient to share common functions between the core of an application and its configuration files. When *Jsonnet* is used as a part of a large system, some logic may already be implemented in another area of the codebase. Being able to import it can avoid code duplication.

1.2.3. Requirements for a foreign function interface in *Jsonnet*

In order to incorporate a foreign function interface in *Jsonnet* in a way that is useful to its user and consistent with the design principles of the language, the solution should have the following characteristics:

1. It should be easy to add support for a new target language.
2. It should be easy to implement support for the foreign function interface in an existing *Jsonnet* interpreter (other than *go-jsonnet*) or in a new one.
3. The format of the compiled library should not depend on the variant of the interpreter and on the platform it is executed on.
4. The performance of the code written in the target language should be comparable to its performance running outside *Jsonnet*.
5. Foreign functions should not have access to I/O operations. One core idea of *Jsonnet* is that functions should be pure and foreign functions should conform to the same constraint.

1.2.4. Native functions

There is already a way to execute functions written in other languages built into *Jsonnet*. However, it does not meet the criteria outlined in the previous paragraph. *Native functions* [10] are a mechanism where you embed a function directly within the *Jsonnet* interpreter. For example, in case of the *go-jsonnet*, you can write a callback in *Go*, put it inside an instance of the `NativeFunction` interface provided by the interpreter, and register the structure as a *native function* [11].

While this functionality does enable running foreign functions in *Jsonnet*, it has several disadvantages. First, it is tightly coupled to the implementation of *Jsonnet* you are using. If you have a *native function* in *Go*, you will not be able to easily embed it in the *C++* interpreter. Second, you have to build *Jsonnet* from source yourself, and each time you change your code, you have to rebuild the interpreter as well. Third, inside a *native function* you have unrestricted access to I/O, which means you can, deliberately or not, introduce side effects and violate the assumption that *Jsonnet* programs are pure. Fourth, the maintainers of the language reserve the right to change the mechanism, so some future release could break existing *native functions.* [11]

1.3. *WebAssembly*

WebAssembly is a portable binary instruction format. You can use it as a target when compiling many popular programming languages like *C, C++, Rust*, or *C#.* The main goals of the project are portability, security, and performance [12]. Programs compiled to *WebAssembly* run inside a sandboxed environment and should achieve close to native speed.

WebAssembly was initially designed to run inside a web browser. One core use case is to easily use existing libraries on a web page [13]. However, it recently became apparent, that the technology has potential outside client-side web development. The objectives behind the project make it a viable solution for writing a piece of code once and running it efficiently across many environments on the server-side as well [14]. A module compiled to a *WebAssembly* binary can be executed on many architectures. Because of these features of *WebAssembly*, it was chosen as the compilation target for the foreign language library imported by the *Jsonnet FFI.*

1.3.1. WASI

WASI, short for *The WebAssembly System Interface*, standardises the way *WebAssembly* modules interact with resources such as files, sockets, or the clock, and serves as an intermediate layer between the sandboxed code and the operating system. Before it was introduced, running server-side *WebAssembly* often required runtimes to implement functions tied to a specific toolchain, like *Emscripten* [14].

Since *WASI* is becoming the standard way of running server-side *WebAssembly*, it is used as well to implement the foreign function interface described in this thesis. This means that, for example, the *Rust* modules are compiled to the `wasm32-wasi` target instead of `wasm32-unknown-unknown`.

After the announcement of *WASI*, Solomon Hykes, the founder of *Docker* wrote (spelling original):

> *If WASM+WASI existed in 2008, we wouldn't have needed to created Docker. That's how important it is. WebAssembly on the server is the future of computing. A standardised system interface was the missing link. Let's hope WASI is up to the task! (Solomon Hykes, [15])*

This shows, that the tech community sees *WebAssembly* and *WASI* as a potential future standard for writing secure, portable code.

1.4. *WebAssembly* runtime

In order to execute *WebAssembly* outside a browser, a runtime is needed. This is the program that will run the *WebAssembly* binary. As indicated above, the runtime should also support *WASI.* Two such popular runtimes are *Wasmer* [16] and *Wasmtime* [17]. *Wasmtime* was chosen to solve the problem described in this thesis, but an implementation with *Wasmer* was also written and used for comparison in section 3.5. *WebAssembly*. *Wasmtime* enables running *WebAssembly* as a standalone application from the command line, as well as embedding it within an existing application in another language. Among many supported languages are *Go* and *C++*, which means, that it was possible to embed it in the *go-jsonnet* interpreter, as well as make sure, that extending support to the other official interpreter would not be a problem in the future.

1.5. Libary languages – *Rust* and *C*

When adding a foreign function interface, one of the design decisions that had to be made was what foreign languages to choose. Because of implementation details described later in the thesis, *WebAssembly* files have to conform to a specified interface to be accepted by *Jsonnet FFI*. While technically any *WebAssembly* module can be modified to follow these rules, one of the goals was to facilitate writing such libraries in a selected subset of languages. To that end, necessary glue code needed to be written in each of them.

Rust and *C* were chosen. The main factors were the size and quality of the codebases. Both of these languages are fast. Both are widely used by the industry in various critical systems as well. One of the examples is the *Linux* kernel. It is implemented in *C*. However, recently the community began work on adding *Rust* as a second supported language [21].

An additional factor was the availability of a mature toolchain. *Rust* and *C* are some of the languages with the best support for *WebAssembly*. For *Rust*, there is a `wasm32-wasi` target in `rustc`, which makes compiling projects in `cargo` (the *Rust* package manager [18]) to *WASI* effortless. For *C*, on the other hand, there is a separate project called *Emscripten* [19], which contains not only a compiler (*emcc*), but also tools like *emmake* and *emcmake*, to facilitate building existing projects which use *GNU Make* and *CMake* as their build methods respectively. An interesting fact is that *Emscripten* was used to compile the *C++ Jsonnet* interpreter and embed it on the project website to run example code snippets inside a browser [20].

1.6. BSON

BSON (short for *Binary JSON*) [22] is a binary data format first introduced by *MongoDB*, a document-oriented *NoSQL* database, for their underlying *JSON* user data [23]. It can store *JSON* objects more efficiently than their simple *UTF-8* representation. This means, for example, that numbers are mapped to raw bytes instead of sequences of text characters. Improvements such as this make *BSON* more lightweight and convenient to use for low-level exchange of *JSON* data. Therefore, it is a good fit to act as an interface between *Jsonnet*, which operates on *JSON* objects, and a foreign language such as *Rust*, which in such a foreign function interface has to serialize and deserialize its data structures into *JSON*.

1.7. Diagnostic tools

To test the computation time and memory efficiency of the solution, several diagnostic programs were utilised.

1.7.1. *Valgrind*

Valgrind [24] is a suite of tools for program analysis, which, among other things can be used to check for memory leaks and invalid memory accesses, as well as report total memory usage.

It was used to look for potential memory management bugs in the implementation of the foreign function interface in the interpreter and the glue code in *C* and *Rust*. Memory errors in crucial parts of code responsible for calling foreign libraries could have negative effects that would make the whole mechanism unusable. To read more about how this was tested, see section 2.5.

1.7.2. The *strace* program

The *strace* program is a *Linux* diagnostic tool which, among other things, can trace system calls and provide microsecond timestamps of their start and time deltas of their duration. It was used to assess how long it takes to pass control from the interpreter to a foreign function, as well as compare that with a similar scenario in *Go* (see section 3.3.1).

1.7.3. The *Heaptrack* program

The *Heaptrack* program is developed by the *KDE* community. It can track memory allocations and deallocations together with the stack traces of the functions that initiated them, summarise the memory usage of a binary over time, as well as show the various collected metrics on graphs.

Chapter 2

Solution

2.1. *WASM* library format

2.1.1. Design decisions

The implementation of a foreign function interface in *Jsonnet* with *WebAssembly* (*WASI*) as the target architecture can be split in two main parts – support in the interpreter and glue code in the foreign language. The most important design decision is the specification of an interface between those two areas. To decide that, several questions need to be answered.

First, how will *Jsonnet* know which functions from the *WebAssembly* module should be accessible to the library user? Apart from the actual exports written by the foreign library author, the toolchain may expose some internal functionality, like stack management, which could be useful in a different scenario. It is not the case with *Rust*, but it is with *Emscripten*. Libraries in *Jsonnet* are regular objects, which means that one is able to execute methods like `std.objectFields` (that lists all the fields of an object) on them. The exported functions that are specific to *Emscripten* should not be seen as part of the library object and should not be callable.

Second, how will *Jsonnet* know the names of method arguments? If a function in *Rust* is called from *Jsonnet* with an insufficient number of arguments, the interpreter must produce a runtime error message with the name of the parameter which was omitted. In *WebAssembly* parameters are referenced by an integer index [25], so it is impossible to retrieve such information just from the signature.

Third, how will the arguments be passed to the actual function, and how will the return value be received? Because *WebAssembly* is low level, its types include little more than the basic 32-bit and 64-bit integers and floating point numbers. This means, that it is hard to express objects like strings of characters, let alone more complex structures. A closely related problem is where will the memory for the arguments be allocated and deallocated if they are to be passed by a pointer. It is important to remember here as well one of the requirements from section 1.2.3, which stated that the format of the compiled library should not depend on the platform it is executed on.

Answering these questions should in turn imply a specific format of the *WebAssembly* module.

2.1.2. Suggested format

The proposed library format is as follows:

1. The module should export two functions for internal use by the interpreter. One called `__jsonnet_internal_allocate`, for memory allocation, and one called `__jsonnet_internal_deallocate`, for memory deallocation.

2. *BSON* should be used as the exchange format. All data sent from *Jsonnet* to the *WebAssembly* module and back should be formatted as raw *BSON* bytes and passed via a pointer. The first 32 bits of a *BSON* document represent its size, so it is not necessary to pass any additional information, even if, for example, the receiver needs to deallocate that memory.

3. To be accepted by the interpreter, the *BSON* values returned by the library need to be one of the following: 64-bit floating point numbers, 32-bit or 64-bit signed integers, booleans, strings, `null` values, documents, or arrays. *BSON* supports more types, but they do not have immediate equivalents in *JSON* or *Jsonnet*. (Complex objects like documents and arrays can also recursively contain only types mentioned in this point.)

4. The names of the functions which are to be accessible from *Jsonnet* have to start with `__jsonnet_export_`.

5. If an exported function has parameters, they should be passed by a pointer to a single *BSON* object in the resulting *WebAssembly*. For each argument, the document should contain the parameter name as a key, with the argument as the corresponding value.

6. Function arguments, if any, are allocated by the caller using `__jsonnet_internal_allocate` and deallocated by the callee.

7. The return value of a function is a pointer to a *BSON* object where the result value has an empty string as a key. It can be any supported *BSON* type (see point 3). Note that while in *JSON* the top-level item can be any value (object, array, string, etc.), *BSON* restricts it to be an object. Therefore, it is necessary to wrap the return value in a document and an empty string as a key minimises the overall memory usage.

8. The return value is allocated internally by the callee and deallocated by the caller using `__jsonnet_internal_deallocate`.

9. For each function exported by the user, the module should export as well a metadata function with a corresponding name, but starting with `__jsonnet_internal_meta_`. This function should return an array of parameter names.

For an example illustrating the proposed format, see appendix A.

2.1.3. Rationale

One of the most important choices was the data exchange format. The simplest solution would be *JSON* encoded in *UTF-8*. However, it would be wasteful for such low-level operations as function calls. It would, for example, mean having to serialize and deserialize each number as well as store them as sequences of characters in their decimal representation. *BSON* was chosen to solve these issues. It provides a compact and well-defined schema for function arguments and results.

Alternative solutions with a similar approach to *JSON* serialisation include: *BJData* [26], *CBOR* [27], *MessagePack* [28], and *UBJSON* [29]. *BSON*, however, backed by the authors of the *MongoDB* database, has the most widespread adoption in the industry and the best library support across *Go*, *Rust*, and *C*. Different serialisation methods, like *Apache Avro* [30] and *Protocol Buffers* [31], were considered as well, but seemed to require the exchanged data to conform to a more rigid schema.

Having chosen the format and keeping in mind the fact that more complex data structures in *WebAssembly* have to be passed via a pointer, the next major step was to decide who should be responsible for allocating and deallocating memory for arguments and results. Because the library running inside the *Wasmer* instance only has access to its own block of memory, the best way to pass arguments to it is by exporting an allocation function and copying the arguments to the location returned by that function. The deallocation of the arguments, on the other hand, could be done either by the interpreter or the *WASM* function. The second approach was chosen, because this way in *Rust* functions can own their arguments, and do not have to borrow them. If they were deallocated by the interpreter, they would either have to be passed by reference or copied. In *C* the choice of where to deallocate arguments does not make such a difference. It is possible to make callee-side deallocation inside the glue code, as shown in section 2.4. Since only the *Jsonnet* interpreter can know when a return value becomes unused, it is natural that it is responsible for deallocating that memory.

Finally, instead of having a separate metadata function for each function exported by the user, it would be possible to pack all information about parameter names for the whole module into one function. This approach could reduce the time necessary for *Jsonnet* to initialise a library object in some scenarios. One call would be sufficient, rather than a number of calls which is equal to the number of library methods. However, in this case the initialisation of a large module could require considerably more memory in order to store all the parameter names in a single *BSON* object. Moreover, with this approach it could be significantly harder to provide elegant metaprogramming glue code, especially in *Rust*, where an elegant solution was found in the shape of an attribute macro (see section 2.3), but also in *C*, and potentially other programming languages in the future.

2.2. Interpreter implementation

2.2.1. Syntax

Part of the work preparing this thesis was to implement the foreign function interface described in the previous sections inside the *go-jsonnet* [7] interpreter. The syntax of *Jsonnet* was only marginally changed by introducing an `importwasm` keyword, analogous to the already present `import` keyword used for *Jsonnet* files. The goal was to make the new mechanism behave as close as possible to the existing one. This meant, that it should return a regular *Jsonnet* object with the exported functions visible as its methods. One notable difference between `import` and `importwasm` is that normal imports can include fields which are any valid *Jsonnet* value, while *WebAssembly* imports support only functions. Listing 2.1 shows the syntax and the ease of use of the proposed foreign function interface.

Listing 2.1: New syntax

```
{
  local wasmLib = importwasm "lib.wasm",
  local jsonnetLib = import "lib.jsonnet",
  wasmLibResult: wasmLib.add(1, 2),
  jsonnetLibResult: jsonnetLib.add(1, 2),
}
```

2.2.2. Lifetime of an exported function

To be able to execute *WebAssembly*, the interpreter needs to read a module file and then create a *Wasmer* runtime instance with it. Because *Jsonnet* is a lazy programming language, none of that work is done if an imported library is not used. This means, that the foreign function interface does not incur any memory or computational costs whatsoever, unless it is necessary to call a foreign function in order to calculate the result of the program. Consequently, existing code will not be affected in any way by the introduction of new features. If, however, the execution of a program requires a method from *WASM* to be called, the interpreter takes the following steps (an illustration of this process can be seen on figure 2.1).

First, if the library has not already been initialised, a *Wasmtime* instance is created, which will be responsible for running it. The contents of the module are passed on to the instance at this point in time. This means that all unique invocations of `importwasm` receive a separate runtime (thanks to caching, duplicated imports run on a single *Wasmtime* instance – see 2.2.3). Afterwards, all metadata functions (with a `__jsonnet_internal_meta_` prefix) are called for corresponding exported functions (with a `__jsonnet_export_` prefix). They do not take any arguments and return pointers to *BSON* objects with arrays of parameter names (see A.3.1). The result is then stored internally in a structure from which it can be retrieved if a runtime error requires that information.

Second, once the library has been initialised (or fetched from memory if it has been initialised before), the function arguments are prepared. They are serialised to *BSON*. A chunk of memory of an appropriate size is allocated in the internal memory of the *WASM* runtime using the `__jsonnet_internal_allocate` function. The binary buffer containing the arguments is then copied to that memory. It is important to observe, that this forces eager evaluation. Although lazy argument passing from *Jsonnet* to *WASM* might be an interesting research topic, it is outside the scope of this thesis.

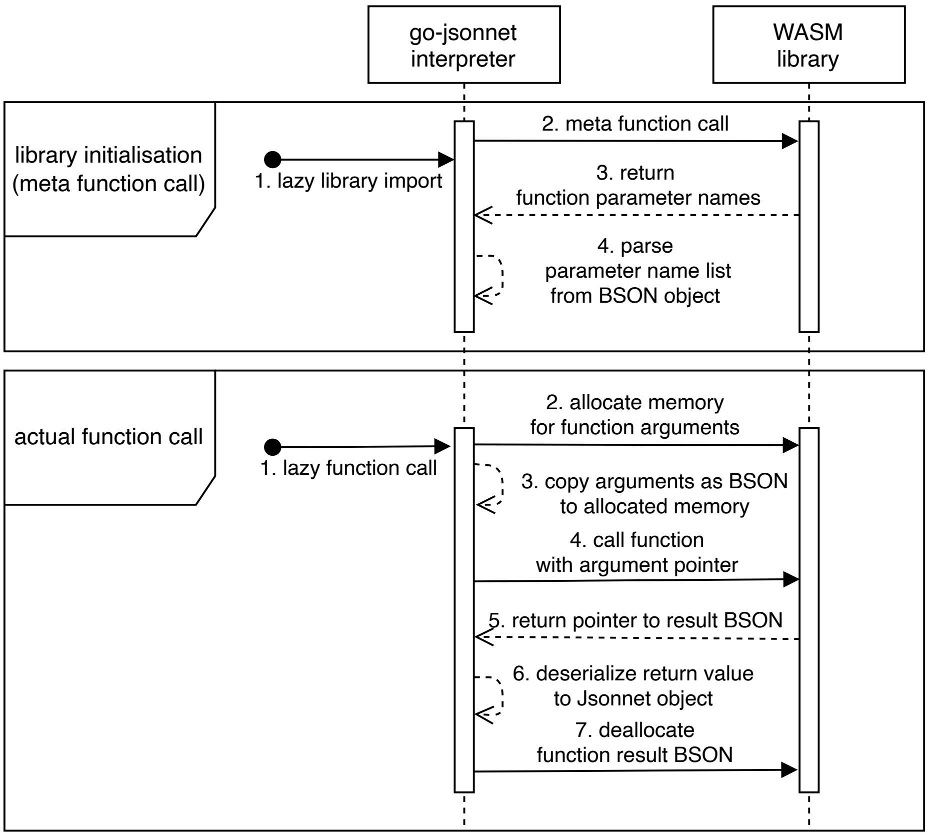
Third, with the input prepared, the function that does the actual computation is executed. Arguments are passed via a pointer to the *BSON* object (see A.3.2). They are consumed and deallocated by the callee. A pointer to a *BSON* result is then returned to the caller, deserialized, and parsed as a *Jsonnet* value. The memory that was used to hold the binary buffer for the result is then freed.

In total, three *WebAssembly* calls are performed for each *Jsonnet* foreign function call – one for computation and two for allocation purposes. In addition, there is one call for each imported method during library initialisation.

2.2.3. Caching

Like everything in *Jsonnet*, imports are expressions. They can appear anywhere in the source code. In particular, a single file can be imported multiple times. The semantics of the language mean, that merging all those objects into one will produce the same result. Libraries and methods inside them cannot contain any internal state, because that would violate the purity principle. Owing to that, it is possible to cache a library and reuse it if encountered later. Apart from an improvement in efficiency, an additional benefit of this approach is that reading the same file twice could produce different results if the file was written to between those two moments, which would be contrary to the principles of *Jsonnet*. The existing imports, that is the `import` keyword mentioned in section 2.2.1 and the simpler `importstr` used to include raw text files as strings, employ caching. Therefore, `importwasm` was implemented to behave analogously. As a consequence, only one *WASM* runtime instance is created for each imported physical file. The metadata functions are only called a single time as well. Most

Figure 2.1: Lifetime of an exported function


go-jsonnet interpreter
WASM library
library initialisation (meta function call)
1. lazy library import
2. meta function call
3. return function parameter names
4. parse parameter name list from BSON object
actual function call
1. lazy function call
2. allocate memory for function arguments
3. copy arguments as BSON to allocated memory
4. call function with argument pointer
5. return pointer to result BSON
6. deserialize return value to Jsonnet object
7. deallocate function result BSON

often, however, it is only sensible to import a library once, so this optimisation will rarely matter. But, in these scenarios, it helps reduce the overhead of the foreign function interface, as well as keep the overall behaviour of the language consistent.

2.3. Rust support

The first language to receive glue code for the foreign function interface was *Rust*. It has excellent *WebAssembly* and *WASI* support built into the official toolchain in the form of the `wasm32-wasi` target.

The goal was to make exporting existing programs as easy as possible. An ideal way to do that would be to provide the user with some kind of annotations, which would transform each function into two: an equivalent computation function that takes a *BSON* pointer as an argument and returns a pointer to the *BSON* result, as well as a metadata function, which returns the function argument names. Solving that requires metaprogramming.

There are two main metaprogramming mechanisms in *Rust* – declarative macros and procedural macros. Declarative macros, or "macros by example", are defined using pattern matching rules and invoked similarly to regular functions, but with an exclamation mark at the end. The `println!` macro, used for standard output, is an example of that. Even if it were possible to write such a macro for the transformation described in the previous paragraph, it would mean that the definition of the exported function would have to either be fully contained in a macro call, or require a separate macro call. Neither of these solutions feels intuitive. Fortunately, procedural macros in *Rust* help to address this issue. There are three kinds of procedural macros: function-like macros, derive macros, and attribute macros. They need to be compiled in a separate `proc-macro` type crate. All of them are defined by functions that take `TokenStreams` (which represent the raw abstract syntax tree of the item they are transforming) as input and produce a `TokenStream` as output. Function-like macros are executed similarly to declarative macros, which does not solve the problem described above. Derive macros, on the other hand, can be invoked only on `structs`, `enums`, and `unions`. Attribute macros, however, can be attached to a function declaration and fit all the other requirements.

A `#[jsonnet_export]` attribute macro is introduced. Each function annotated with that macro will be exported in a format readable by *Jsonnet*. First, the macro parses the input `TokenStream` using the `syn` crate. Then, it extracts parameter names from the signature. A metadata function is declared with an array of the parameter names as a result. The input function is then copied, but with a different name, for internal computation. It is referenced by the third, main function, with the `__jsonnet_export_` prefix. In *Rust*, function names can be mangled, so all functions created by that macro, that are to be visible from *Jsonnet*, have the `#[no_mangle]` attribute. Attribute macros can take parameters, but in this use case they are not needed.

The `bson` crate in *Rust* supplies functions for easy *BSON* serialisation and deserialisation. They work well with the popular `serde` crate. Because of that, in order to be accepted as a parameter in an exported function, the type only needs to implement the `DeserializeOwned` trait. Symmetrically, return values need to implement the `Serialize` trait. Most of the basic types already meet those requirements. In structures defined by the user, they can be easily implemented by adding the `#[derive(Serialize, Deserialize)]` attribute.

2.3.1. *Rust* attribute macro usage

Exporting the following snippet as a library:

```
#[jsonnet_export]
pub fn add(x: f64, y: f64) -> f64 {
  x + y
}
```

leads to this code (after macro expansion, previewed using `cargo expand`):

```
#[no_mangle]
pub extern fn __jsonnet_internal_meta_add() -> *mut u8 { ... }

pub fn __jsonnet_internal_add(x: f64, y: f64) -> f64 {
  x + y
}

#[no_mangle]
pub unsafe extern fn __jsonnet_export_add(input_bson: *mut u8)
  -> *mut u8 { ... }

#[no_mangle]
pub extern fn __jsonnet_internal_allocate(size: usize)
  -> *mut c_void { ... }
```

```
#[no_mangle]
pub extern fn __jsonnet_internal_deallocate(pointer: *mut c_void,
                                            capacity: usize)
  { ... }
```

The original `add` function is split into three: `__jsonnet_internal_meta_add`, `__jsonnet_export_add`, and `__jsonnet_internal_add`. The first two are consistent with the format proposed in section 2.1.2. The last one, not mentioned there, is equal to the input `add` function, but has a different name. It is called by `__jsonnet_export_add` to calculate the result and separates safe code provided by the user from unsafe memory management and pointer passing done in the `extern` functions. The `add` (and consequently the `__jsonnet_internal_add` function) takes and returns values of type `f64`, which has the `DeserializeOwned` and `Serialize` traits. These two functions own their arguments. This means that when the arguments go out of scope after the function returns, the underlying memory is released (the difference could be more apparent if the passed type was a `struct`, like `bson::Document`, which could be used to pass a *Jsonnet* object). If argument deallocation was to be done on the side of the interpreter (caller), the arguments would either have to be copied at some point or passed by reference.

2.4. *C* support

The second language chosen for the foreign function interface was *C*. In this ecosystem the build tools are tied less closely to the compiler. The *Clang* compiler can generate *WebAssembly* with *WASI*. Additionally, the libraries necessary to link *C* programs (like `libc`), are provided by *wasi-sdk* [32]. However, building existing *GNU Make* and *CMake* projects using only those tools could become complicated and involve individually modifying their build setup. Fortunately, this process is made easy by *Emscripten* with the help of the *emmake* and *emcmake* commands.

Unlike *Rust*, *C* does not have attribute macros. Macros in *C* could be compared to *Rust*'s declarative or function-like macros. Therefore, the approach here had to be different. A similarly elegant solution seemed unlikely. Instead of a single macro, two macros and some additional helper functions are provided.

First, there is a `JSONNET_ATTRIBUTES` macro to insert compiler attributes necessary to modify and export function names to *WebAssembly* (the actual *C* function name does not have an effect). The second macro, `JSONNET_REGISTER_ARGS`, is used to define the `__jsonnet_internal_meta_` function. It takes the function name and the parameter names as arguments. Third, there is a `jsonnet_parse_args` function, which takes a pointer to the raw argument buffer and parses it as *BSON*. Fourth, there is a collection of `jsonnet_get_*` functions to retrieve the previously parsed arguments by name. The asterisk stands for any of the supported *BSON* data types (`bool`, `double`, `int32`, `int64`, `null`, `string`, `array`, or `document`). Finally, there is an analogous collection of `jsonnet_return_*` functions for returning a value. An additional function, `jsonnet_get_arg_type`, is provided to help check the type of an argument, which in *Jsonnet* can differ between calls. (Since this is possible, in some sense the *C* glue code is more flexible than the glue code in *Rust*, because in *Rust* the parameter types are fixed in the signature.) Declaring the *Jsonnet* allocation and deallocation functions was easy, because in *C* they can be implemented by using the popular `malloc` and `free`.

2.4.1. *C* macros and functions usage

Exporting the following snippet as a library:

```
void *add(uint8_t *arguments_ptr) JSONNET_ATTRIBUTES(add) {
  jsonnet_args *arguments = get_jsonnet_args(arguments_ptr);
  int32_t x = jsonnet_get_int32_arg(arguments, "x");
  int32_t y = jsonnet_get_int32_arg(arguments, "y");
  return jsonnet_return_int32(x + y, arguments_ptr, arguments);
}

JSONNET_REGISTER_ARGS(add, "x", "y");
```

leads to this code (after macro expansion, previewed using the `--save-temps` flag in *Clang*):

```
uint8_t *__jsonnet_internal_meta_add()
  __attribute__(( export_name("__jsonnet_internal_meta_""add")))
  { ... }

uint8_t *add(uint8_t *arguments_ptr)
  __attribute__((export_name("__jsonnet_export_""add"))) {
  jsonnet_args *arguments = get_jsonnet_args(arguments_ptr);
  int32_t x = jsonnet_get_int32_arg(arguments, "x");
  int32_t y = jsonnet_get_int32_arg(arguments, "y");
  return jsonnet_return_int32(x + y, arguments_ptr, arguments);
}

uint8_t *__jsonnet_internal_allocate(size_t size)
    __attribute__((export_name("__jsonnet_internal_allocate"))) {
  return malloc(size);
}

void __jsonnet_internal_deallocate(void *pointer, size_t capacity)
    __attribute__((export_name("__jsonnet_internal_deallocate"))) {
  free(pointer);
}
```

It is important to note, that the `jsonnet_return_*` group of functions take, apart from the return value, two additional parameters, `arguments_ptr` and `arguments`. They are used to deallocate the arguments, as requested by point 6 of the format suggested in section 2.1.2

2.5. Preventing memory leaks

The *go-jsonnet* interpreter, being implemented in *Go*, which has a garbage collector, is not exposed to memory safety problems encountered in languages where memory is managed by the programmer. This, however, does not mean, that a *Go* program cannot have memory management problems. They can appear, for example, when parts of the memory are not managed by the garbage collector, like when *Go* is interfacing with a different language. Therefore, although it would not be useful to test the whole modified interpreter with the foreign function interface with a tool like *Valgrind*, it is possible to some extent to test the general pattern of allocation and deallocation of arguments and return values, as well as the glue code in *Rust* and *C*, which could prevent some runtime errors in the future. One of such errors could be an out of memory error, when the memory pool allocated by *Wasmer* for the *WASM* module is exhausted. To test the memory allocation and deallocation, a simple mock in *C* of the part of the interpreter responsible for the foreign function calls was created. It is possible to execute *Rust* functions from *C*, so this mock was used to test the glue code in both languages, once by linking a *Rust* library and once by linking a *C* library to the exact same test executable. The *Rust* and *C* libraries were compiled to native code instead of *WebAssembly*, in order to be traced by *Valgrind*.

2.5.1. Testing leaks in *Rust*

Rust has strong safety guarantees. One of the core concepts of *Rust* is that it should be very hard, if not impossible, to write code that leaks memory. However, reasons like lack of performance or expressive power (e.g. while talking to the operating system) mean that sometimes it is useful to bypass the constraints enforced by the compiler. This was the reason behind introducing *unsafe Rust*. Using the `unsafe` keyword in *Rust*, one can perform risky operations, like dereferencing raw pointers, which could result in accessing invalid memory addresses and would otherwise be blocked during compilation.

In order to pass data between the *Jsonnet* interpreter and a library written in *Rust*, it is necessary to use the `unsafe` parts of the language. Function arguments and the function result are stored as *BSON* data in raw vectors, which are allocated and deallocated inside the library. This memory is contained in a *WASM* runtime instance embedded within the interpreter. Incorrect management of it, although probably not dangerous outside the runtime, could cause the foreign library to quickly exhaust its memory, and therefore should be prevented. Of course, it is impossible to eliminate all possible memory leaks in such a foreign library in *Rust*, because apart from the *unsafe* keyword being used in the glue code supplied for the developer, the actual code written by the developer can be unsafe as well.

Two foreign functions were prepared and tested using *Valgrind* – one doing simple addition (like the one in appendix A.1) and one not taking any arguments and just returning a constant value. (Functions that take arguments are handled in a different branch in the glue code than functions that do not take arguments.) Unexpectedly, running the foreign functions compiled to native code in *Valgrind* returns three leaks, all similar to each other (only the first one is shown):

Listing 2.2: *Valgrind* output for *Rust* glue code test (some details omitted for the sake of readability)

```
HEAP SUMMARY:
    in use at exit: 88 bytes in 3 blocks
  total heap usage: 41 allocs, 38 frees, 3,504 bytes allocated

8 bytes in 1 blocks are still reachable in loss record 1 of 3
   malloc (in /usr/libexec/valgrind/vgpreload_memcheck-arm64-linux.so)
   alloc::alloc::Global::alloc_impl (alloc.rs:181)
   alloc::alloc::exchange_malloc (alloc.rs:330)
   ahash::random_state::RandomState::get_src::{{closure}} (boxed.rs:218)
   once_cell::race::once_box::OnceBox<T>::get_or_init::{{closure}}
                                                  (race.rs:243)
   once_cell::race::once_box::OnceBox<T>::get_or_try_init (race.rs:263)
   once_cell::race::once_box::OnceBox<T>::get_or_init (race.rs:243)
   ahash::random_state::RandomState::get_src (random_state.rs:184)
   ahash::random_state::RandomState::new (random_state.rs:197)
   <ahash::random_state::RandomState as core::default::Default>::default
                                            (random_state.rs:257)
   <indexmap::map::IndexMap<K,V,S> as core::default::Default>::default
                                                  (map.rs:1466)
   bson::document::Document::new (document.rs:188)

...

LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 88 bytes in 3 blocks
        suppressed: 0 bytes in 0 blocks
```

Although this looks dangerous, upon closer examination two facts make these leaks acceptable.

First, the number of leaks seems to be constant and not to depend on the number of function calls. The leaked memory is allocated while a `bson::document::Document` is created for the return value of the first executed function. However, no new memory is leaked for subsequent calls.

Second, similar leaks appear even in the simplest possible use case of the `Document` structure. For the following program, *Valgrind* prints an analogous error summary, listing 88 bytes in 3 blocks as leaked but still reachable (`doc` is a macro used to simplify the creation of `Document` objects):

Listing 2.3: Simplest example of a *Rust* program using the `bson::document::Document` structure, which causes memory leaks

```
use bson::doc;

fn main() {
    let _ = doc! {};
}
```

This suggests, that the issue is on the side of the `bson` crate, or rather one of its dependencies. Somewhere in the hash map initialisation function a global value of type `RandomState` is created and apparently used for all following hash maps as well. As a consequence, it would be impossible to eliminate this problem without substituting the `bson` crate, which is not feasible.

2.5.2. Testing leaks in *C*

In contrast, in *C* it is the programmer who is responsible for memory allocation and deallocation. Here, there is no distinction between *safe* and *unsafe* code as well, as there is in *Rust*. Therefore, running an analogous test with *Valgrind* in *C* is even more useful. Fortunately, this time no errors were detected.

Listing 2.4: *Valgrind* output for *C* glue code test (some details omitted for the sake of readability)

```
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 30 allocs, 30 frees, 6,497 bytes allocated

All heap blocks were freed -- no leaks are possible

For lists of detected and suppressed errors, rerun with: -s
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

2.6. Requirements

Now, it is important to compare the proposed solution with the requirements outlined in section 1.2.3. First, it is easy to add support for a new target language. If it is possible to compile a language to *WebAssembly* with *WASI*, it should not be a problem to include it in the foreign function interface. Potential candidates for future development are the languages listed on the official webpage of the *WebAssembly* project [33]. Second, it is easy to implement a similar mechanism in other *Jsonnet* interpreters, like the official *C++* one. Both *Wasmtime* and *Wasmer* have *C/C++* bindings. Third, the format of the library does not depend on the platform it is executed on. *WebAssembly* is portable, so it should be possible to compile on an *x86* processor and run on an *ARM* processor, or in any other configuration. Fourth, the performance is close to native [12]. This was tested, and the findings are presented in the following chapter. Finally, it was possible to block I/O operations. This means, that accessing standard input and standard output, as well as any other files, is forbidden. However, having fully pure foreign functions was not achieved.

2.6.1. Pure functions

WASI, the proposed standard way to run *WebAssembly* outside a browser, exposes functions called "system calls" for a module to execute when there is a need to interface with the outside world. Examples include `fd_read` and `fd_write`, for reading and writing to file descriptors, or `environ_get` for getting environment variables [34]. By default, though, no descriptors and no environment variables are exposed by the runtime. They can be provided while the module is initialised. Therefore, while it is possible to implement an impure function, most of them are automatically pure. Out of all the "system calls", the ones that seem to cause most trouble are `clock_res_get`, `clock_time_get`, and `random_get`. With them, a *WebAssembly* function can be written that returns different values when passed the same arguments. An example would be simply returning the current time or generating a random number in *Rust*.

No solution was found to this problem. It would be possible to modify a *WebAssembly* runtime to disable these features or change their behaviour, but maintaining such a fork would be troublesome. *Wasmtime* provides two interesting flags with its command line interface: `--wasi-modules` and `--wasm-features`, which enable and disable some *WebAssembly* functionality. However, the control is not fine-grained enough and blocking too much could break some useful programs. In general, compiling with stricter focus on purity could break more existing libraries. The *WASI* standard is changing, and new proposals are constantly being made to improve it. Maybe future versions will make it easier to implement pure functions.

A second, more fundamentally hard to eliminate way to write non-pure functions, is by using global variables. Global variables are a part of the *WebAssembly* standard [35]. While rejecting all modules that use them would fix the problem, it might not be practical. Both *rustc* and *emscripten* use global variables for internal purposes. Many other compilers with *WASM* support probably do the same.

An alternative solution, which to some degree would address both issues mentioned in this section, would be caching function results for given *BSON* function arguments. Once a function was called with arguments, which serialise to a certain *BSON* document, the return value would be stored in a hash map with that *BSON* as a key for future use. This, however, would come with memory overhead and in some cases computation time overhead. Additionally, there would still be ways to exploit this mechanism. Although generating a random number would return the same answer within a single execution of a *Jsonnet* program, successive executions would differ. Similarly, it would be harder to maintain internal state using global variables, but any function could be modified to work the same way by taking an additional integer parameter, which would be incremented with every call.

Chapter 3

Efficiency

3.1. Speed improvement

Having implemented the foreign function interface, it was important to confirm, that one of the main requirements, that the execution time of a function should be comparable to its native execution time, was satisfied (point 4 from section 1.2.3). In particular, foreign functions in efficient languages like *C* or *Rust* should be significantly more performant than analogous ones in *Jsonnet*, which is relatively inefficient. To test this, a CPU intensive function was selected, which was not yet available in *Jsonnet*, but could be useful for *Jsonnet* programmers. It was then implemented three times. First, in pure *Jsonnet* without using any external libraries. Second, in *Rust*, using an existing library. Finally, in *Jsonnet*, with a foreign function using the same *Rust* library, but compiled to *WebAssembly*.

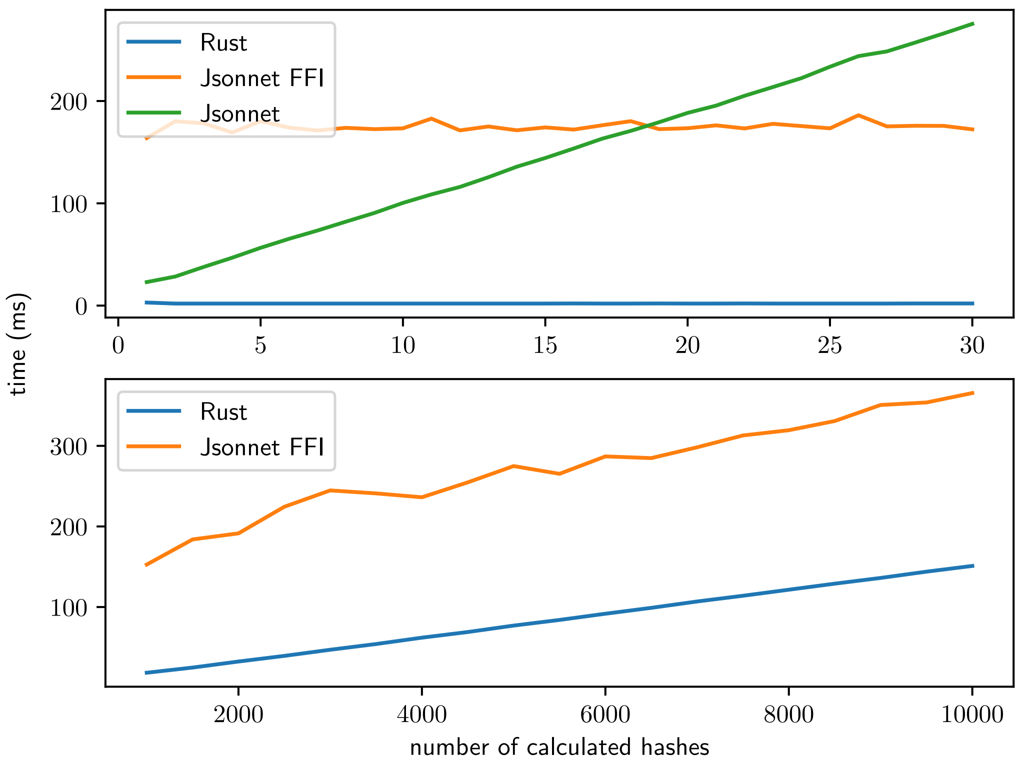
While comparing a custom implementation written by the author of this thesis to a well established and tested one in a faster language may sound unfair, it reflects a real world scenario. When writing *Jsonnet* code, one might need to use some existing functionality. In the absence of a foreign function interface, it would be necessary to implement it manually.

It is important to note here, that in order for the *Jsonnet* implementation not to hit stack size limits in the following tests, it was necessary to increase them using the `-s` command line flag. There are ways to address that issue without increasing the stack size, like the poorly documented `tailstrict` keyword [38] which forces strict evaluation of arguments. However, implementing complex algorithms is hard in itself, so having to know *Jsonnet* optimisations would be an additional burden on the programmer.

The algorithm chosen to highlight the performance differences was *SHA-1*. Although not cryptographically safe, it is still used in different important pieces of software, like the *Git* [36] version control system, where, among other things, it stores commit hashes [37]. The compared programs calculated multiple times the hash of a simple message – `"helloworld"`. This process involves loops (in *Rust*) or folds (in *Jsonnet*, owing to its functional nature), binary arithmetic, as well as array manipulations. All tests were run multiple times and the lines shown are an average of these calculations. The time measured was the complete execution time of the binaries. Results of the comparison are presented on figure 3.1. All the tests apart from section 3.5 were done on an M1 MacBook Air.

There are two graphs. On the first one, all three programs are present. It shows, that there indeed is a significant advantage to importing *Rust* compiled to *WebAssembly*, as opposed to running naïve *Jsonnet*. On the other hand, this is only apparent after the number of calculated hashes increases past a certain point. This is due to the fact, that in short programs, the overhead of initialising a library (which includes creating a *WebAssembly* runtime and

Figure 3.1: *SHA-1* hash calculation time comparison



executing metadata functions) makes pure *Jsonnet* more efficient. This overhead is also visible when comparing *Rust* to the *Jsonnet FFI*. On this graph it is not yet clear how much slower the latter one would be if the overhead were to be ignored.

The difference is visible on the second graph. The implementation in pure *Jsonnet* was omitted, because for numbers of calculated hashes thousands of times bigger than on the previous graph, it would be orders of magnitude slower and cause the other two lines to merge into one. Apart from the overhead, expected to be constant in this example, the slowdown is less than twofold, which is consistent with figures mentioned on the internet [39]. This suggests that the goal of achieving near native speed was met.

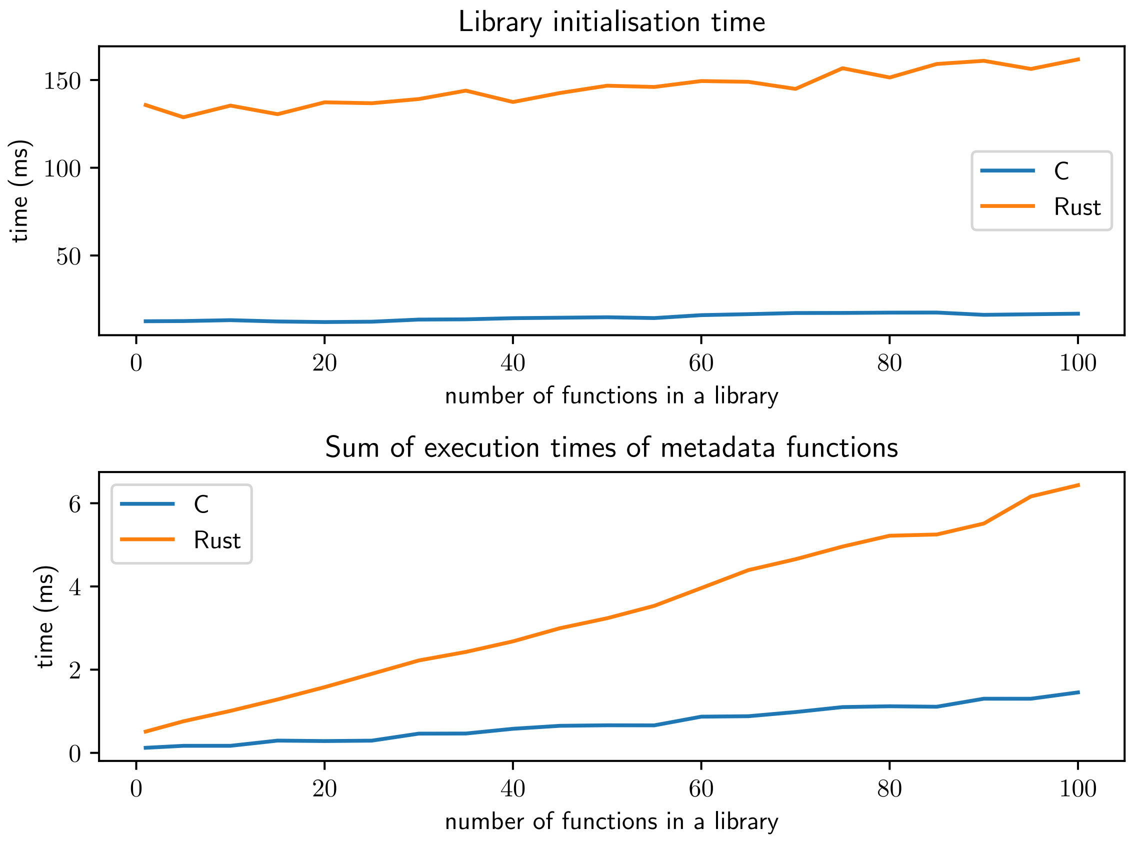
3.2. Library initialisation overhead

There are two places where the foreign function interface can cause computation time overhead – library initialisation and the actual execution of the exported function. It is important to reiterate here, that all the costs described in the following sections are only present when a programmer is actively using the foreign function interface. Otherwise, the interpreter runs normally, as it would without this feature being present. The lazy nature of *Jsonnet* means, that even if the `importwasm` keyword is present in a program, no additional computation is done unless it is actually required to calculate the result.

The things done during library initialisation are: reading, parsing, and compiling the *WebAssembly* module, extracting function names from it, filtering functions starting with `__jsonnet_export_`, and running the corresponding metadata functions to extract parameter names (for a description of the process, see section 2.2.2). Many factors could affect the time these tasks take. Some of them include: the number of functions in the module, the size of the binary (dictated by the size of the dependencies which are included in the binary, as well as the size of the functions defined by the user), and physical limitations like disk speed and CPU performance. This section explores the effects of the first of these factors.

To test how much time is required for library initialisation depending on the function count, multiple *Rust* and *C* modules were prepared, each with a different number of addition functions. The functions were equivalent to the one in appendix A.1 and differed from each other only in their name. Results are presented on figure 3.2.

Figure 3.2: Library initialisation overhead



The top graph presents library initialisation time. This is the time of execution of the `makeRuntimeInstance` function in the interpreter. It is responsible for all the *WASM* runtime-specific parts like reading and compiling the module. The bottom graph shows the total time taken by the metadata functions. It includes steps like *BSON* deserialisation of the argument names returned by those functions as well. For example, for a library with 100 exported functions, this is the combined time of calling all 100 corresponding metadata functions, and then parsing the results as string slices (dynamically-sized arrays in *Go*) of parameter names for future use by the interpreter. All values were measured inside the *go-jsonnet* interpreter, using the `time` package. The following conclusions come to mind while observing the graphs.

First, the metadata functions have a comparably insignificant effect on the overall performance. In *Rust* The 6 ms required for 100 metadata function calls are dwarfed by the 150 ms of preparing the runtime. The proportions are similar in *C*. As a consequence, trying to optimise this area of the solution (as proposed in the last paragraph of section 2.1.3) is not necessary.

Second, a lot of time is spent during library initialisation. A closer inspection reveals, that on average 98% of this step is consumed compiling the module (inside `wasmtime.NewModule`). Comments in the source code of *Wasmtime* acknowledge, that this is an expensive operation [40]. Reducing this overhead could be hard. *Wasmtime* command-line interface has a `wasmtime compile` command, which can be used for ahead-of-time compilation. However, it would mean that additional files, in form of the precompiled code, would be stored in the directory where *Jsonnet* is run. Moreover, this functionality is not yet supported in the *Wasmtime Go* bindings, which were used to implement the foreign function interface.

Third, it takes significantly longer to initialise a basic *Rust* library, than a comparable *C* library. This difference could be caused by module size – the *C* modules are 40-70 KiB, while the *Rust* modules are around 10 MiB.

3.3. Function execution overhead

The second place where there could be computation time overhead connected to the foreign function interface is the actual function execution. Two potentially expensive actions here are argument serialisation and the switch of control from code running in the interpreter to code running in the foreign language.

3.3.1. Language switch overhead

The method chosen to test how much time is required for passing control from *go-jsonnet* to a *Rust* function and back again, was to measure using *strace* the time between pairs of artificially injected system calls. To this end, a custom *WebAssembly* foreign function written in *Rust* was created, which executes two system calls – one just at the beginning and one just before it returns. Analogous system calls are executed in *Go* on the side of the interpreter – one just before the *WebAssembly* function is called and one just after it finishes.

Testing with *strace* might not be the most precise way to measure time durations. Moreover, there is no certainty, that between two system calls, one in an outer function and another in an inner function, some additional code is not executed. This is the reason why `syscall.Syscall` was used instead of a function like `fmt.Println`, which also executes a system call, but which could have additional operations after that. Nevertheless, this should give an upper bound on how expensive the switch from *Go* to a *WebAssembly* function is. It could be especially useful if compared with a similar test in pure *Go*.

The following snippet presents a reference setup in *Go* of the pattern that was implemented:

Listing 3.1: *Go* program to measure time spent entering and exiting a function using *strace*

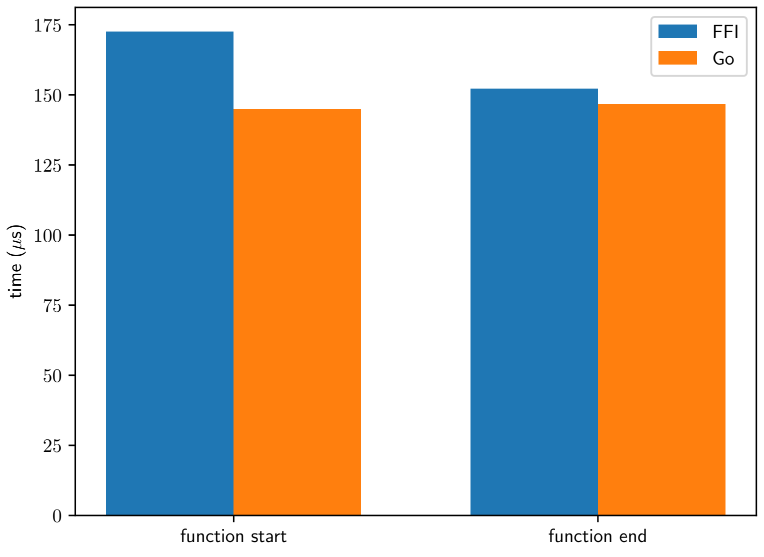
```
func inner() {
  syscall.Syscall(syscall.SYS_CLOCK_GETRES, 0, 0, 0) // 2
  ...
  syscall.Syscall(syscall.SYS_CLOCK_GETRES, 0, 0, 0) // 3
}

func main() {
  syscall.Syscall(syscall.SYS_CLOCK_GETRES, 0, 0, 0) // 1
  inner()
  syscall.Syscall(syscall.SYS_CLOCK_GETRES, 0, 0, 0) // 4
}
```

An analogous sequence of system calls was injected in the interpreter and a foreign function. The aforementioned *strace* tool then registered the system calls made by these two programs (having passed the `-ttt` and `-T` flags, because by default *strace* does not print timestamps).

```
// start_timestamp             syscall                     duration
1667148793.977069 clock_getres(CLOCK_REALTIME, NULL) = 0 <0.000127> // 1
1667148793.977351 clock_getres(CLOCK_REALTIME, NULL) = 0 <0.000089> // 2
... // function execution
1667148793.977900 clock_getres(CLOCK_REALTIME, NULL) = 0 <0.000137> // 3
1667148793.978201 clock_getres(CLOCK_REALTIME, NULL) = 0 <0.000132> // 4
```

The overhead of entering a function was then calculated by subtracting the timestamp of the end of the first system call (obtained by adding the duration to the start timestamp) from the timestamp of the beginning of the second system call ($\texttt{start\_timestamp\_2} - (\texttt{start\_timestamp\_1} + \texttt{duration\_1})$). The overhead of exiting a function was determined similarly. The results of these two experiments are presented on figure 3.3 (the values are averages of over 50 executions).

Figure 3.3: Function execution overhead measured with *strace*



No significant difference is visible between the foreign function interface and pure *Go*. There is a slight increase in the time it takes to enter and exit a function. However, the order of magnitude is the same. This is consistent with the description of how *Wasmtime* runs *WebAssembly*. *WASM* modules are compiled to native code [41] (this is the cause of the library initialisation overhead mentioned in section 3.2). As a consequence, switching to this native code from *Go* should not have a big impact.

3.3.2. Argument serialisation and result deserialisation overhead

Apart from the time it takes to pass control from the interpreter to a foreign function, the second potentially expensive operation is *BSON* serialisation and deserialisation. To put into perspective the cost related to that, functions with various numbers of integer and string parameters were prepared. The time to serialise those arguments was then compared to the time spent evaluating them (they are first evaluated, and then serialised). Although argument evaluation can be very slow, depending on what computation it requires, in this experiment only literals (integers and short strings) were passed as arguments. This should therefore give an idea of how long preparing and reading the *BSON*s takes in comparison to other steps in the process of calling a function. The results are shown on figure 3.4.

The graphs show, that argument serialisation takes comparable time to evaluation of the literals that are serialised. Even though there are three other places (argument deserialisation, result serialisation, result deserialisation) where a similar task is performed, all of them combined should not be unreasonably expensive.

3.4. Memory overhead

While the previous sections were concerned with the computation time of the proposed *Jsonnet* foreign function interface, it is also crucial to look at a different important aspect – the memory usage of the solution. Especially interesting is the inevitable memory overhead of executing foreign functions in a *WebAssembly* runtime. To this end, three basic *Jsonnet* files were prepared, each one importing and executing a different implementation of a simple addition function – one in *Rust* (compiled to *WebAssembly*), one in *C* (compiled to *WebAssembly* as well), and an analogous one in *Jsonnet*. These three programs were then compared using *Valgrind*. Apart from tracking memory management problems, *Valgrind* can also report total memory usage. The results should give a picture of the minimal amount of memory required by the foreign function interface

For a program which calls only *Jsonnet* code, *Valgrind* reports around 1.5 KiB allocated. In contrast, calling a *Rust WebAssembly* function results in allocating nearly 1 GiB, more than 500 000 000 as much as in the previous case. The *C* function lies somewhere in the middle, with 30 MiB.

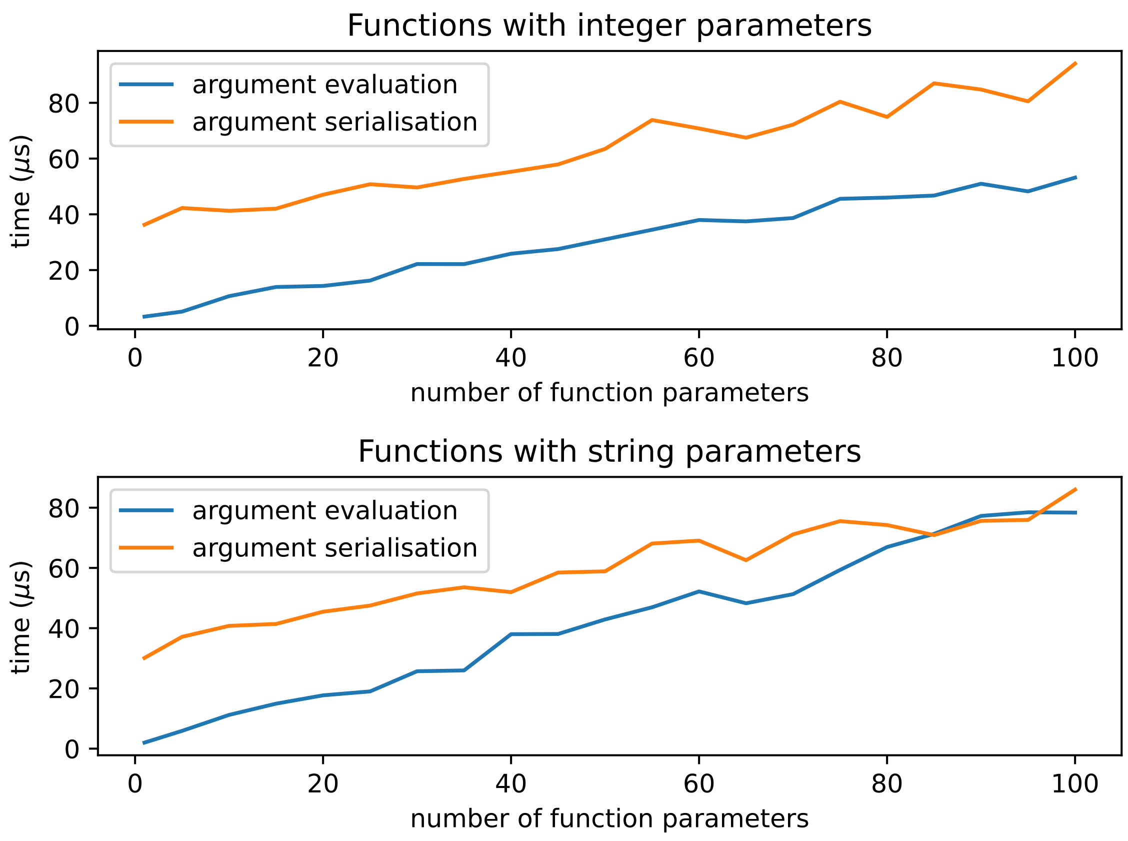
Listing 3.2: Memory usage of *go-jsonnet* in *Valgrind* when importing a *Jsonnet* file

```
total heap usage: 11 allocs, 6 frees, 1,624 bytes allocated
```

Listing 3.3: Memory usage of *go-jsonnet* in *Valgrind* when importing a *Rust* module compiled to *WebAsssembly*

```
total heap usage: 2,147,560 allocs, 2,146,451 frees,
                  955,870,417 bytes allocated
```

Figure 3.4: Argument evaluation and serialisation time depending on number of parameters in a function



Listing 3.4: Memory usage of *go-jsonnet* in *Valgrind* when importing a *C* module compiled to *WebAsssembly*

```
total heap usage: 53,787 allocs, 53,083 frees,
                  30,490,895 bytes allocated
```

It should be noted, that at no point in the lifetime of the program were the 995 870 417 bytes (or 30 490 895 in case of *C*) all used at the same time. Apart from allocations, deallocations were also happening, which means, that the peak memory usage was much smaller.

3.4.1. Peak memory usage

According to another memory tracing tool, *Heaptrack*, the maximum memory usage in a single moment of *Jsonnet* running the *Rust* function was around 9 MiB (see figure 3.5). This is still a lot more than the 1.5 KiB when not running *WASM*. The flame graph on figure 3.6 shows which functions were responsible for the highest amount of allocated memory (towards the bottom are the outermost functions, which then call other functions, going up on the graph). The vast majority of them (as highlighted by the looking glasses) are connected either to *Cranelift*, which is the *WebAssembly* compiler embedded in *Wasmtime*, or to the `wasmtime::module::Module` structure, which is also responsible for code compilation and calls *Cranelift* internally.

Figure 3.5: *Heaptrack – Jsonnet FFI* memory consumption over time

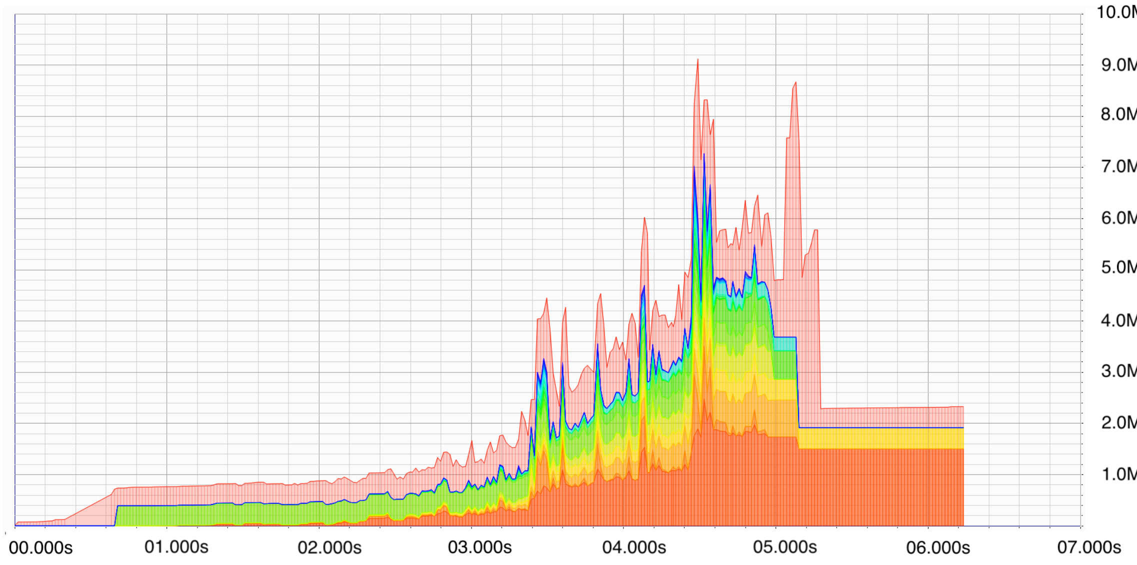
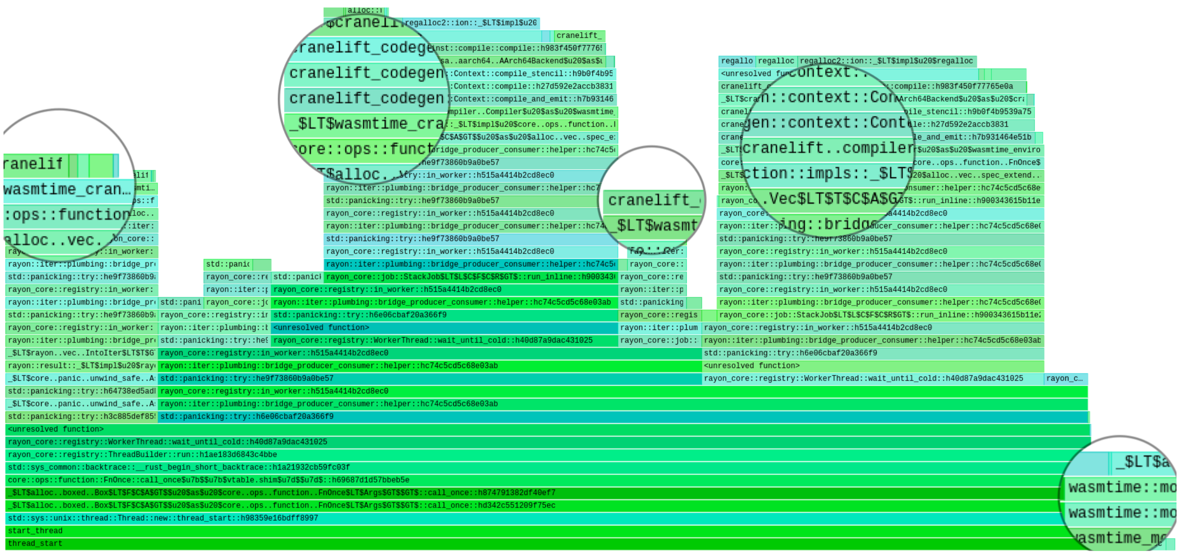


Figure 3.6: *Heaptrack – Jsonnet FFI* flame graph of the allocated memory during peak memory usage



There is at least one valid reason why so much memory should be used during the preparation and compilation of the *WebAssembly* module. According to comments in the source code, *Wasmtime* "requires that the entire binary is loaded into memory all at once" [42]. Even though the *Rust* library used in this test is very basic, its size is more than 9 MiB when compiled to *WASM*. This could be affected, among other things, by the `bson` crate requirement, which is the only external dependency in this case. Needing the whole module in memory for compilation does imply at least 9 MiB of peak usage.

On the other hand, the module written in *C* is only 42 KiB. However, *Heaptrack* still reports a peak of 4.9 MiB used during its compilation and execution. This means, that at best a similar memory consumption (single digit megabytes) should be expected regardless of the size of the library.

3.5. ***Wasmtime*** vs ***Wasmer***

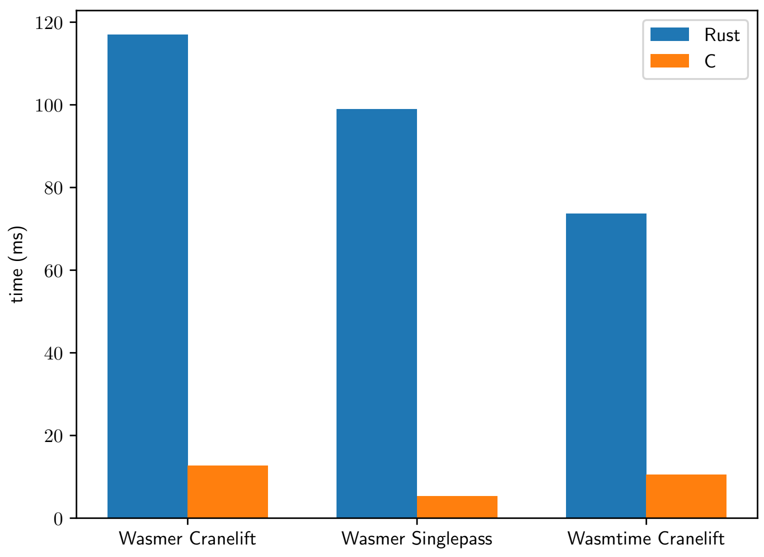
There are two popular *WebAssembly* runtimes – *Wasmtime* [17] and *Wasmer* [16]. *Wasmtime* was chosen instead of *Wasmer* for two main reasons. First it is developed by the Bytecode Alliance, which is a company backed by many recognisable corporations [44]. Second, in the version of *Go* bindings for *Wasmer* current as of the writing of this paragraph, it is possible to override the standard output and standard error streams, but there is no such mechanism provided for standard input (however, there is ongoing work on this topic [43]).

In spite of that, it could be interesting how these two runtimes compare to each other. *Wasmer* has a feature of choosing an alternative *WebAssembly* compiler as well. For this comparison, the foreign function interface was reimplemented in *Wasmer*. Both runtimes support the *Cranelift* compiler (it is the default in *Wasmer*, and the only compiler in *Wasmtime*). However, *Wasmer* also has the *Singlepass* compiler, which is supposed to generate native code faster, but result in slower execution time.

On figure 3.7 are shown the runtime initialisation times (the measured function in the interpreter was `makeRuntimeInstance`, similarly to section 3.2) of two small libraries – one in *Rust* and one in *C*. The tests in this section, due to technical reasons, were performed on a PC with an *Intel* i5-10400 processor running the *Linux* kernel. This is the cause of the discrepancy between results on this graph and on figure 3.2 (for number of functions equal to 1). As mentioned when discussing the memory overhead, the *C* library is much lighter and this results in quicker compilation and consequently library initialisation (compilation is the most expensive step).

The initialisation of the library when using *Cranelift* is faster in *Wasmtime* both in *C* and in *Rust*. It is worth noting, that *Cranelift* is currently part of the *Wasmtime* project. What is interesting, is that the initialisation of the *Rust* library is slower in *Wasmer* even when using *Singlepass*. On the other hand, *Singlepass* leads to the fastest initialisation of the *C* library. The results of this comparison suggest, that apart from the functional reason mentioned above, the performance aspect also speaks in favour of *Wasmtime* for the use case described in this thesis.

Figure 3.7: Import time of a basic library in *Jsonnet* depending on the runtime and the *WASM* compiler



Chapter 4

Conclusion

There are many decisions that need to be taken when designing and implementing a foreign function interface. This thesis described one approach to this problem, guided by the needs and peculiarities of the *Jsonnet* programming language. It described the advantages of adding such a mechanism in *Jsonnet* and showed, that *WebAssembly* is a viable tool to solve it. The proposed reference implementation satisfies most of the requirements, as outlined in section 1.2.3. This solution opens *Jsonnet* programmers to the vast amounts of libraries and resources provided by the *Rust* and *C* communities. The tests in chapter 3 confirm, that it can significantly improve the performance of some *Jsonnet* programs, consistent with claims of near-native speed found on the internet. On the other hand, they also show, that there is an overhead of up to several hundred milliseconds and around 10 mebibytes of memory, most likely as a result of compiling *WebAssembly* to binary code during the runtime of the interpreter. The effects of this could be especially apparent in short programs. However, the cost should be outweighed by the benefits when running more demanding calculations. Additionally, future improvements in *Wasmtime* and the `bson` crate or the `rustc` compiler could help alleviate this problem.

4.1. Areas for improvement

There are several directions in which further development could go. One would be to achieve better, tighter integration of *Jsonnet* with *Rust* and *C*. Another possibility would be to add support for a new language altogether.

4.1.1. Laziness, functions as values

One of the fundamental assumptions made during the design of this foreign function interface for *Jsonnet* was that, in order to pass data between the interpreter and a foreign function, all arguments need to be eagerly evaluated and serialised. In particular, functions as object fields, while perfectly normal in *Jsonnet*, were not allowed. This enabled the author to focus on more basic questions, like the feasibility of having *WebAssembly* as the middle layer and the performance achieved as a result.

One way to make the solution more in line with the basic principles of *Jsonnet* would be to somehow switch to lazy argument passing. This would most probably require additional glue code on the side of the foreign language, especially if, like in *C* or *Rust*, the arguments there are evaluated eagerly. An orthogonal problem is passing functions as arguments. Although at first it may sound harder than the previous one, it is actually possible to import functions in

WebAssembly during module initialisation. An example of this are the *WASI* "system calls" mentioned in section 2.6.1. To run a function that generates a random number, it needs to be imported by the module. Similarly, one could import a *Jsonnet* function and later use it when it is referenced during the interpretation of a *Jsonnet* file. There are at least two issues that would need to be addressed to implement this mechanism – the necessity to declare imports during runtime initialisation (when some functions used later may not yet be created) and the fact, that a function may not only be passed as a top-level argument, but also nested inside an object. The first problem could be solved by some form of a dispatch function, which would be imported by all *WebAssembly* modules, and then called whenever a *Jsonnet* function call was required, with the *Jsonnet* function name (for the dispatch function to know which *Jsonnet* function to call) in addition to all the regular arguments. The second problem would probably necessitate a more sophisticated custom data structure to be used for object arguments.

Adding the functionality described above would help bring the *Jsonnet* foreign function interface closer to full interoperability with the foreign languages.

4.1.2. Side effects

As mentioned in section 2.6.1, no satisfying solution was found to guarantee that foreign functions imported by *Jsonnet* are pure. Future developments in the *WebAssembly* and *WASI* standards might help accomplish that. On the other hand, some isolation is provided in form of lack of access to input/output operations, the filesystem, and the network.

4.1.3. Language support

Despite the fact that *C* and *Rust* already provide a substantial codebase of high quality libraries, another area of improvement could be including more programming languages. There already are many languages which can be compiled to *WebAssembly*, as mentioned on the official website [33]. However, many projects are working on adding *WASM* support. An good example is the proposed *WebAssembly* backend with *WASI* support for the *Glasgow Haskell Compiler* [45]. Combining *Jsonnet* with another lazy language like *Haskell* could provide interesting possibilities.

4.2. Merge status

The solution described in this thesis has been developed in contact with members of the *Jsonnet* community. The requirements mentioned in previous chapters are real requirements, that need to be satisfied for a foreign function interface mechanism to be useful in *Jsonnet* and to be merged into its codebase. While initially it was intended for this merge to happen during the development leading to the creation of this document, time constraints have made it impossible. Moreover, the concerns about function purity presented in section 2.6.1 could outweigh the benefits brought by this solution. Nevertheless, the findings will be shared with the *Jsonnet* community and might help, if not to include this particular *WebAssembly* foreign function interface as part of the project right away, then at least to inform decisions about similar matters in the future.

Appendix A

Examples illustrating the proposed format of a *WebAssembly* library exported to *Jsonnet*

A.1. *Rust* file

Suppose one wants to export the following function from *Rust* to *Jsonnet*:

```
pub fn add(x: f64, y: f64) -> f64 {
  x + y
}
```

A.2. *WebAssembly* module

The resulting *WebAssembly* binary should be analogous to this (ellipses were inserted where unimportant text was omitted):

```
(module
  (type (;0;) (func (param i32) (result i32)))
  (type (;4;) (func (param i32 i32)))
  (type (;14;) (func (result i32)))
  ...
  (func $__jsonnet_internal_allocate (type 0) (param i32) (result i32) ...)
  (func $__jsonnet_internal_deallocate (type 4) (param i32 i32) ...)
  (func $__jsonnet_internal_meta_add (type 14) (result i32) ...)
  (func $__jsonnet_export_add (type 0) (param i32) (result i32) ...)
  ...
  ;; allocation function, gets buffer size as an argument
  ;; and returns a pointer to the allocated memory
  (export "__jsonnet_internal_allocate"
      (func $__jsonnet_internal_allocate.command_export))
  ;; deallocation function, gets a pointer and a buffer size
  (export "__jsonnet_internal_deallocate"
      (func $__jsonnet_internal_deallocate.command_export))
  ;; metadata function, returns a pointer to a document
  ;; with the list of argument names for the current function
  (export "__jsonnet_internal_meta_add"
      (func $__jsonnet_internal_meta_add.command_export))
  ;; function which does the actual computation,
  ;; gets a pointer to a document with the arguments
  ;; and returns a pointer to the document with the result
  (export "__jsonnet_export_add"
      (func $__jsonnet_export_add.command_export)))
```

A.3. Intermediate *BSON* documents

A.3.1. Result *BSON* of the `__jsonnet_internal_meta_add` function

This function returns the parameter names that need to be passed in the *BSON* received by the `__jsonnet_export_add` function.

```
{
  "": [
    "x",
    "y"
  ]
}
```

A.3.2. Argument *BSON* of the `__jsonnet_export_add` function

In this instance, the function was called with 1 and 2 as arguments.

```
{
  "x": 1,
  "y": 2
}
```

A.3.3. Result *BSON* of the `__jsonnet_export_add` function

In this instance, the function was called with 1 and 2 as arguments.

```
{
  "": 3
}
```

Bibliography

[1] Oracle and/or its affiliates, *Java Native Interface Specification*, `https://docs.oracle.com/en/java/javase/16/docs/specs/jni/index.html`, accessed 08.05.2022 7

[2] Armin Rigo, Maciej Fijalkowski, *CFFI documentation*, `https://cffi.readthedocs.io/`, accessed 08.05.2022 7

[3] Jsonnet contributors, *Jsonnet – The Data Templating Language*, `https://jsonnet.org/`, accessed 08.05.2022 7

[4] Jsonnet contributors, *Jsonnet Tutorial*, `https://jsonnet.org/learning/tutorial.html`, accessed 08.05.2022 7

[5] Joey Bartolomeo, *How to configure Grafana as code*, `https://grafana.com/blog/2020/02/26/how-to-configure-grafana-as-code/`, accessed 03.11.2022 7

[6] kapicorp, *Kapitan: Generic templated configuration management for Kubernetes, Terraform and other things*, `https://kapitan.dev/`, accessed 03.11.2022 7

[7] Jsonnet contributors, *go-jsonnet*, `https://github.com/google/go-jsonnet`, accessed 08.05.2022 7, 15

[8] Jsonnet contributors, *Getting Started – C++ or Go?*, `https://jsonnet.org/learning/getting_started.html#cpp-or-go`, accessed 08.05.2022 7

[9] Li Haoyi, Josh Rosen, Ahir Reddy, *Writing a Faster Jsonnet Compiler*, `https://databricks.com/blog/2018/10/12/writing-a-faster-jsonnet-compiler.html`, accessed 08.05.2022 8

[10] Jsonnet contributors, *Jsonnet Specification*, `https://jsonnet.org/ref/spec.html`, accessed 07.05.2022 8

[11] Aleš Plšek, Stanisław Barzowski, Dave Cunningham, James R. Qualls, *Custom builtin functions #223*, `https://github.com/google/go-jsonnet/issues/223`, accessed 07.05.2022 8, 9

[12] World Wide Web Consortium, *WebAssembly High-Level Goals*, `https://webassembly.org/docs/high-level-goals/`, accessed 07.05.2022 9, 23

[13] Mozilla Foundation, *Compiling an Existing C Module to WebAssembly*, `https://developer.mozilla.org/en-US/docs/WebAssembly/existing_C_to_wasm`, accessed, 07.05.2022 9

[14] Lin Clark, *Standardizing WASI: A system interface to run WebAssembly outside the web*, https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/, accessed 07.05.2022 9

[15] Solomon Hykes, *tweet @solomonstre, 9:39 pm, 27 Mar 2019*, https://twitter.com/solomonstre/status/1111004913222324225, accessed 07.05.2022 9

[16] Wasmer, Inc., *Wasmer*, https://wasmer.io/, accessed 08.05.2022 9, 33

[17] Bytecode Alliance, *Wasmtime – A fast and secure runtime for WebAssembly*, https://wasmtime.dev/, accessed 08.05.2022 9, 33

[18] Rust Foundation, *The Cargo Book*, https://doc.rust-lang.org/cargo/, accessed 01.10.2022 10

[19] Emscripten contributors, *emscripten*, https://emscripten.org, accessed 01.10.2022 10

[20] Jsonnet contributors, *Getting Started – JavaScript*, https://jsonnet.org/learning/getting_started.html#javascript, accessed 01.10.2022 10

[21] Michael Larabel, *Rust Code Updated For The Linux Kernel - Networking & Async Support Started*, https://www.phoronix.com/scan.php?page=news_item&px=Rust-v6-For-Linux-Kernel, accessed 09.05.2022 10

[22] MongoDB Inc., *BSON*, https://bsonspec.org/, accessed 10.09.2022 10

[23] MongoDB Inc., *JSON and BSON*, https://www.mongodb.com/json-and-bson, accessed 10.09.2022 10

[24] Valgrind™ Developers, *Valgrind*, https://valgrind.org/, accessed 24.10.2022 10

[25] World Wide Web Consortium, *Modules*, https://webassembly.github.io/spec/core/syntax/modules.html#functions, accessed 01.10.2022 13

[26] Qianqian Fang, *Binary JData: A portable interchange format for complex binary data*, https://github.com/NeuroJSON/bjdata/blob/Draft_2/Binary_JData_Specification.md, accessed 03.10.2022 14

[27] Carsten Bormann, *CBOR RFC 8949 Concise Binary Object Representation*, http://cbor.io/, accessed 03.10.2022 14

[28] Sadayuki Furuhashi, *MessagePack*, https://msgpack.org/, accessed 03.10.2022 14

[29] UBJSON contributors, *Universal Binary JSON Specification*, https://ubjson.org/, accessed 03.10.2022 14

[30] The Apache Software Foundation, *Apache Avro™ - a data serialization system*, https://avro.apache.org/, 03.10.2022 15

[31] Google LLC, *Protocol Buffers*, https://developers.google.com/protocol-buffers, accessed 03.10.2022 15

[32] World Wide Web Consortium, *WASI SDK*, https://github.com/WebAssembly/wasi-sdk, accessed 10.10.2022 19

[33] World Wide Web Consortium, *Developer's Guide*, `https://webassembly.org/getting-started/developers-guide/`, accessed 14.10.2022 23, 36

[34] World Wide Web Consortium, *WASI docs*, `https://github.com/WebAssembly/WASI/blob/main/phases/snapshot/docs.md`, accessed 15.10.2022 23

[35] World Wide Web Consortium, *Modules*, `https://webassembly.github.io/spec/core/syntax/modules.html#syntax-global`, accessed 25.10.2022 24

[36] Members of the Git community, *Git*, `https://git-scm.com/`, accessed 25.10.2022 25

[37] Members of the Git community, *Hash function transition*, `https://git-scm.com/docs/hash-function-transition/`, accessed 25.10.2022 25

[38] Shimin Guo, Dave Cunningham, Stanisław Barzowski, *document tailstrict #343*, `https://github.com/google/jsonnet/issues/343`, accessed 26.10.2022 25

[39] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha, *Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code*, `https://www.usenix.org/conference/atc19/presentation/jangda`, accessed 26.10.2022 26

[40] Bytecode Alliance, *module.rs file in the wasmtime project, comment about compilation time*, `https://github.com/bytecodealliance/wasmtime/blob/v2.0.0/crates/wasmtime/src/module.rs#L42`, accessed 27.10.2022 28

[41] Bytecode Alliance, *Architecture of Wasmtime*, `https://docs.wasmtime.dev/contributing-architecture.html`, accessed 31.10.2022 30

[42] Bytecode Alliance, *module.rs file in the wasmtime project, comment about inmemory loading*, `https://github.com/bytecodealliance/wasmtime/blob/v2.0.0/crates/wasmtime/src/module.rs/#L136`, accessed 02.11.2022 32

[43] Paul Berg, Ivan Enderlin, Syrus Akbary, Manos Pitsidianakis, Felix Schütt, *[c-api] add more functions for stdin/stderr/stdout control*, `https://github.com/wasmerio/wasmer/issues/2334`, accessed 05.11.2022 33

[44] The Bytecode Alliance contributors, *Bytecode Alliance*, `https://bytecodealliance.org/`, accessed 05.11.2022 33

[45] Cheng Shao, *WebAssembly backend*, `https://gitlab.haskell.org/ghc/ghc/-/wikis/WebAssembly-backend`, accessed 03.11.2022 36