Facultade de Informática
Universidade da Coruña
Departamento de Computación

Proyecto de Fin de Carrera
Ingeniería Informática

Automatic bridging of native code to Lua using existing debugging information

Student: Adrián Pérez de Castro
Director: Laura Milagros Castro Souto
Date: August 25, 2015

Todo list

Do it, or don't, but don't try.

Acknowledgements

To my wife, who supported unconditionally me during the long hours I have devoted to this project, and helped to proof-read the final iterations of the present document.

To my parents, whom have not thought that I would ever get this piece of work done.

Also, I would like to thank my Finnish "adoptive" family, who have kindly accepted me as one more of them, and that have been of invaluable support. Their appreciation of knowledge is something I am willing to pass down to upcoming generations.

Summary

The objective of this project is to implement an automated mechanism that, using the DWARF debugging information from ELF shared objects, allows the Lua virtual machine to call native functions from shared objects implemented in the C programming language. The process is automatic, in the sense that the user does not need to write code to convert values passed between Lua and the invoked C functions, and the C functions behave essentially like Lua from the user point of view. The ultimate goal is to allow transparent usage of existing C libraries from Lua.

Lua has been chosen because it provides a clean C interface to its Virtual Machine (VM), which has been designed from the ground up to be embedded in larger projects. The implementation is also compact (under 16.000 lines of code), which makes it feasible to gain in-depth knowledge of its innerworkings in a relatively short time. Lua has also grown in popularity in the last years as its adoption has skyrocketed in the game industry.

The reason to focus on the combination of debugging information in DWARF format contained in ELF shared objects is that they are a widespread, standard configuration used by the majority of contemporary Unix-like operating systems. The target system during development has been a GNU/Linux system running on the Intel x86_64 architecture, which also uses the aforementioned configuration, though provisions are to be included in the design to ease future porting efforts for other platforms.

In order to validate the correctness of the implementation, an automated test suite was also developed. Unit tests were used also as regression tests, to ensure that modifications to the system did not introduce programming errors in the implementation.

Keywords

- Automatic binding generation.
- ELF.
- DWARF.
- Debugging information.
- Lua programming language.
- Virtual machines.
- FFI.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Description & Motivation

Most programming languages provide some mechanism to use libraries —sometimes called *modules*— implemented in some other language. Most of the time, this other language belongs to the family of the C language, which can be compiled into *native object code*. The reasons are twofold: on one hand it allows to reuse functionality provided by the system that otherwise would not be available, and in the other hand it opens the door to implementing performance–critical pieces of a system using native code.

Despite the advantages, using native code from a different host programming language requires creating a layer of software often called *bridge*, or *binding* from now on, which wraps the native library to provide an interface compatible with the run-time environment of the dynamic programming language. Those bindings, created either manually or with the help of code generation tools, need to be compiled before they can be used.

When building native code, compilers are capable of adding *debugging information* to their output, which can be used to gain additional insight into a program using a *symbolic debugger*. As a matter of fact, any other tool capable of understanding the format in which the compiler writes the debugging information can make use of it

for its own purposes. Among plenty other details about the source program, debugging information includes descriptions of the functions compiled as part of each compilation unit, parameters and their corresponding data types, return types, and the memory layout of the involved user-defined types; which is a superset of the information needed to invoke those functions. In other words, the debugging information contains all the details needed to make library bindings automatically, potentially allowing dynamic programming languages to invoke native code directly without any kind of human intervention.

1.2 Project Goals

The main goal of this project to develop an automatic binding system for the Lua programming language which allows seamless usage of libraries written in C at runtime. To achieve this, it will use the debugging information generated by the C compiler. Additionally:

- Modifications to the Lua virtual machine, or its core libraries are to be avoided, if possible. The fewer the changes, the lower the maintenance cost of the system when Lua is updated. An implementation which does not modify Lua itself would be usable with Lua packages provided by the operating system, thus easing the setup process.
- The implementation will load Executable and Linkable Format (ELF) shared objects into the Lua virtual machine, and use the debugging information in Debugging With Attributed Record Formats (DWARF) format present in them.
- Values of C types, including user defined ones, will be readable and modifiable from Lua. It will also be possible to create new values of C types from Lua.
- Invocation of functions from loaded shared objects will be supported for functions of arbitrary return types, and any number of parameters of any supported type. Lua values passed to functions will be automatically converted to C types whenever possible. Values of C types created from Lua will also be accepted as valid function parameters.

- The implementation will target the GNU/Linux operating system running on the x86_64 architecture.
- The design of the system will be extensible, allowing to add support for more shared object formats, debugging information formats, operating systems, and architectures.

1.3 Planning & Methodologies

During the planification phase, the following tasks and subtasks have been identified:

1. Initial study, including:
   a) Understanding how different kinds of data are stored in ELF object files.
   b) Identifying the parts of the DWARF specification which apply to the scope of the project.
   c) Investigating existing tools which share similar goals.
2. Analysis, including:
   a) Understanding the relevant parts of the DWARF debugging information format.
   b) Getting acquainted with Lua and the implementation of its VM.
3. Development, including:
   a) Designing the automatic binding system.
   b) Implementing the automatic binding mechanism.
   c) Testing the system, including:
      - Designing a set of unit and regressions tests.

#	Task	Estimation (days)
1.	Initial study	10
2.	Analysis	15
3.	Development	50
4.	Validation	10
5.	Documentation	30
	Total	115

Table 1.1: Effort estimation

- Implementing unit and regression tests.

4. Validation, including:

   a) Developing example Lua programs which demonstrate the capabilities of the system.

   b) Rewriting at least one previously existing program to validate usage of the system in a real–world scenario.

5. Documentation, including writing of the final report.

For each one of the top-level tasks in the list above, Table 1.1 provides an estimation of the time needed for the completion, using an effort of eight hours per person, per day (8h/p/d).

Even though there is only one resource executing the tasks, some techniques from agile development methodologies are used. Namely:

- From Scrum, the concepts of *iteration* and *sprints*, with their respective planning and review seasons. Daily stand-up meetings are not used, and there is no *scrum master*: none of those would make make sense provided that there is only one person in the team.

- The *Kanban* methodology is used in order to keep an always up to date dashboard with the status of the tasks.

The Kanban method was invented by Toyota to keep the status of production lines. This methodology keeps a board (physical, in the original incarnation of the method) where each element is a task, and elements are distributed in columns depending on their status. For example, applied to software development, the columns could be "Pending", "In Progress", "Testing", and "Finished". All the tasks are always visible in the board, so this allows to know the overall status of a project intuitively by glancing at the board.

Chapter 2

Contextualization

Computing would not be understood without the accompanying tools which enable IT professionals to actually *do something* with computers: from simple switches and lights in the early times, to the sophisticated programming languages and tools of the present times, all of them enable humans to *instruct machines to do things*.

2.1 Dynamic Programming Languages

Dynamic languages are a family of high-level programming languages which, at runtime, execute programming behaviors that other programming languages perform during compilation. Runtime behaviors could include extension of the program, by adding new code, by extending objects and definitions, or by modifying the type system. Support for these operations is provided directly by the language. Most dynamic languages are also dynamically typed, and are frequently called "scripting languages".

Add bibliographic reference

Popular contemporary scripting languages include Perl, Python, PHP, the ubiquitous JavaScript, and of course Lua (described in subsection 2.1.3). Many of their features were first implemented as native features of the Lisp programming language, which itself is dynamic.

Dynamic programming languages have grown in popularity in the last decades. Programmers like them for their expressiveness, which allows for higher productivity, and the ever increasing computing power of computers makes it feasible to use them

for performance-critical tasks traditionally left for statically compiled languages. The development of novel compilation techniques aimed at this family of languages, and the relentless improvement of their virtual machines (see subsection 2.1.1), has contributed as well to increase widespread adoption.

2.1.1 Virtual Machines as Runtime Environments

A VM is an emulation of a particular computer system based on its architecture. *System virtual machines* provide a complete substitute for the real machine, and targeted towards the execution of complete operating systems and software stacks. On the other hand, *process virtual machines* execute a single computer program by providing an execution environment suitable for a particular programming language, which can be platform-independent.

Process VMs provide a so called *runtime environment*. They run as a normal application inside the host operating system. This approach has several advantages:

- Provides a platform-independent programming environment, abstracting away details of the underlying hardware or operating system, potentially allowing a program to execute in the same way on any platform.
- The level of abstraction provided by the VM is higher than that of a low-level Instruction Set Architecture (ISA), which allows it to provide services rarely available in real machines (e.g. automatic memory management).
- There is an additional level of isolation between the operating system and the execution environment. This allows discretionary control over the executed code, which is confined to the bounds allowed by the VM.

The main downside of virtual machines is the lower performance compared to the execution of native code. Performance levels comparable to compiled languages can be obtained using a combination of Just-In-Time (JIT) compilation (see subsection 2.1.2), and compiler-based optimization techniques specifically designed for them.

In some regards, the additional level of isolation provided by a VM could be a nuisance due to the fact that it could prevent programs from accessing resources (either

hardware-based, or provided by the host system) needed for its operation. For this reason, many virtual machines provide a Foreign Function Interface (FFI): a mechanism to call-out native code and re-gain access to the whole system, while still being under the control and supervision of the VM.

2.1.2 JIT Compilers

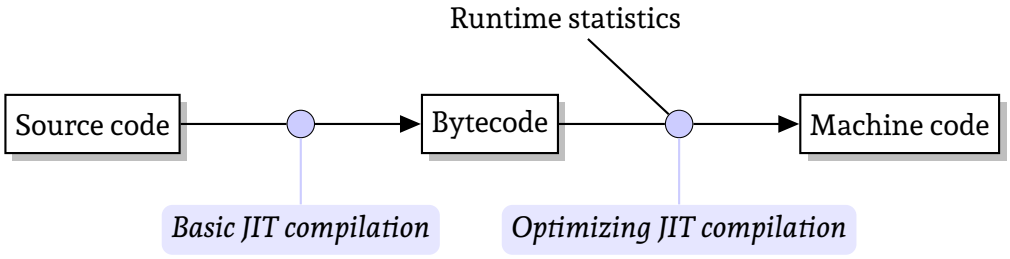

Figure 2.1: Two-tier JIT compilation

Just-In-Time compilation is a term which refers to any technique which performs compilation of program code during the execution of the program —at runtime—, instead of doing it before execution. This is also known as *dynamic translation*. Typically a VM which uses JIT compilation contains two compilers (e.g. as implemented in LuaJIT, Figure 2.1):

1. The first compiler —known as *tier 1*— performs a translation of the input source code into an intermediate binary representation known as *bytecode*, which is faster to execute than interpreting the source code in its original form. In order to start executing the program as quickly as possible, this first compiler performs only basic optimizations—if any. Additionally, the compiler adds instrumentation code in the generated bytecode, which gather statistics about program execution.

2. The second compiler —known as *tier 2*— generates either bytecode which is better optimized, or machine code, at the cost of slower compilation speed, compared to the tier 1 compiler. Therefore, the statistics gathered by the instrumentation code inserted by the tier 1 compiler are used to determine which parts of

the program code are used more often, and only those are recompiled. Depending on the implementation, compilation is done per source file, per function, or even for arbitrary code fragments. The bytecode produced by the tier 1 compiler can be used as input to avoid having to re-parse the program source code in its textual form.

Some VMs do not provide a bytecode interpreter, and must always generate machine code. In this scenario, the tier 1 compiler also generates machine code, but unoptimized to avoid having long delays during the startup process of the VM when the tier 1 compiler is used.

Code generated by JIT compilers offers better performance than interpreters, and in some cases it can perform better than statically compiled code because a JIT compiler can use optimizations which are only feasible at runtime:

- Generated code can be optimized for the exact processor and operating system where the application runs. This can be done with traditional compilers, but it requires compiling the code once for each combination of target processor and operating system.
- The VM can collect statistics about how the program is running, and rearrange and recompile the code according to them.
- The compiler can make optimistic assumptions about the program, generate machine code that works optimally in most of the situations, plus additional checks to know whether the assumptions hold, falling back to using the original implementation (or a different re-compilation) otherwise. For example, this strategy is used to inline method calls assuming that dynamic dispatch will not be needed, and performing the dynamic dispatch when the types of the involved objects to not match the ones assumed.
- Code can be rearranged after observing how it uses the cache memory to make a better usage of it.

The introduction of JIT compilation in VMs has allowed existing dynamic languages to achieve levels of performance comparable to those of languages compiled to machine code [23]. The JavaScript programming language is a paradigmatic example: it

has existed since 1995, yet it has gained much more widespread usage after the introduction in 2008 of JavaScript engines capable of JIT compilation (Mozilla's TraceMonkey was the first, with Google's V8 and WebKit's JavaScriptCode following right after [21]), making it feasible to use JavaScript as a general purpose language outside of web browsers.

2.1.3 The Lua programming language

The Lua language is a "powerful, fast, lightweight, embeddable scripting language" [7]. It was initially created as a data description language at Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), to be used for in-house software development, and has since evolved into a general purpose programming language. It has been used in proffessional applications (e.g. Adobe Lightroom) and it has seen widespread usage in the video games industry (e.g. World Of Warcraft).

Lua's main programming paradigm is imperative, but the language supports functions as first–class values and closures, making it possible to easily write programs in a functional programming style. Like in Pascal, English words (`function`, `then`, `end`) are used as delimiters for language constructs. Another defining characteristic of the language is that, by design, it only provides one compound data structure, the *table*, which is the basis for all user–defined types. Tables can be used as arrays (listing 2), structures, and objects (listing 1).

A unique and powerful feature of Lua is its support for *metatables*: values may have an associated table (the so-called *metatable*) which allows to extend the semantics of language constructs, allowing to define how tables behave when arithmetic and relational operators are applied to tables, or how table fields are accessed. Listing 2 on page 13 demostrates how customizing table access can be used to define a seemingly–infinite array which contains the n^{th} Fibonacci number at index n. A common use for metatables is enabling support for object–oriented programming, using them to define object inheritance chains (listing 3) — as Lua itself does not have the notion of classes, prototypes are used instead, as in the Self or JavaScript languages.

Lua, starting in version 5.0 [9], uses a register-based virtual machine. This allows for improved performance by avoiding excessive copying of values on stack *pop* and *push*

```lua
-- Create a table, with one key, "age" and 7 as value
animal = { age = 7 }

-- Associate a string value to the "kind" table key
animal["kind"] = "cat"

-- Keys which are valid identifiers can be accessed with "."
animal.name = "Doraemon"

-- The dot "." syntax workds for adding functions to tables
function animal.describe(self)
  print(self.name .. " is a " .. tostring(self.age) ..
        "-year old " .. self.kind)
end

-- This is equivalent to: animal.describe(animal)
animal:describe()  --> Doraemon is a 7-year old cat

-- Adding a function with colon ":" adds an implicit "self"
function animal:furryness()
  return self.kind == "cat" and "high" or "unknown"
end

animal:furryness()  --> high
```

Listing 1: Lua tables being used as objects

operations. Traditionally, most virtual machines intended for execution of languages are stack based, including heavyweight, enterprise-proven systems like the Java™ JVM, and Microsoft's .NET environment.

2.2 Binding Native Code to Lua

2.2.1 Lua C API

The Lua VM exposes a C Application Programming Interface (API) which, among other things, allows to register C functions to be called by Lua code. These communicate with the VM using a well-defined protocol (see listing 4 for an example).

```lua
fib = { 1, 1 }
setmetatable(fib, {
  __index = function (values, n)
    -- Calculate and memoize the Fibonacci(n)
    values[n] = values[n - 1] + values[n - 2]
    return values[n]
  end
})
print(fib[10])  --> 55
```

Listing 2: Memoization and dynamic programming using a Lua metatable

Despite Lua providing a register-based VM, the C API uses a *virtual stack* to exchange values with C code. When a C function is called, it gets a new stack, initially containing the arguments passed to the function, and it must adhere to the following protocol:

- C functions must be declared as returning an `int`, and accept a single `lua_State*` parameter. This is, their function pointer type is compatible with `lua_CFunction`, which Lua defines as:

  ```c
  typedef int (*lua_Cfunction) (lua_State*);
  ```

- The C function receives arguments from Lua in its call stack in direct order (the first argument is pushed first by the VM). The size of the stack —at this point the number of arguments— can be queried using `lua_gettop()`, and values can be obtained using the `lua_to*()` functions.

- To return values back to Lua, the C function pushes them onto the stack in direct order (the first result is pushed first) using the `lua_push*()` functions.

- The C function passes control to the VM returning the number of results available at the top of the stack. Any values left in the stack below the results are discarded.

The C API provided by Lua is comprehensive, but using it tends to produce verbose programs, with varying amounts of repeated code. Lua itself acknowledges this issue by officially including an *auxiliar library* as part of the package, which provides a collection of utility functions implemented using the base API. Third-party wrappers over the official API exist, which can either provide a more convenient interface

```lua
-- Base object describing an unnamed living creature
animal = {
  name = "Unnamed",
  kind = "living creature",
  describe = function (self)
    print(self.name .. " is a " .. self.kind)
  end,
}

-- When indexing the table passed as first argument, fields will
-- be looked up from the "animal" table associated to the "__index"
-- key of the metatable. The function returns the first argument.
cat = setmetatable({ kind = "cat", name = "Doraemon" },
                   { __index = animal })
dog = setmetatable({ kind = "dog", name = "Snowy", },
                   { __index = animal })

-- The :describe() method is searched in "animal"
cat:describe()  --> Doraemon is a cat
dog:describe()  --> Snowy is a dog

-- Chained key lookup can be used to make the values from the
-- base object the default ones
tom = setmetatable({ name = "Tom" }, { __index = animal  })
tom:describe()  --> Tom is a living creature
```

Listing 3: Lua metatables used for object inheritance

to Lua (like LuaAutoC[1], and `luapi`[2]), or allow languages other than C to use Lua (more than 20 at the time of writing, including support for many popular languages like C++, Objective-C, Go, Java, or Fortran).

2.2.2 Binding Generators

Binding generators are tools that can be used to create a binding to a library in an automated way. Often they fall into the category of transpilers: they take as input the source code of the code to generate a binding for, and generate a new set of source

[1] `https://github.com/orangeduck/LuaAutoC`
[2] `http://lua-users.org/files/wiki_insecure/users/luapi/luapi5-1.txt`

```c
int sum_and_average (lua_State *L) {
  lua_Number result = 0.0;
  int nargs = lua_gettop (L); /* number or arguments */
  for (int i = 1; i <= n; i++) {
    if (!lua_isnumber (L, i)) {
      lua_pushliteral (L, "argument is not a number");
      lua_error (L);
    }
    result += lua_tonumber (L, i);
  }
  lua_pushnumber (L, result);        /* first result */
  lua_pushnumber (L, result / n);   /* second result */
  return 2;                        /* number of results */
}
```

Listing 4: C function callable from Lua

files which contain the code of the binding. This set of source files are themselves compiled into a loadable module for the target programming language or virtual machine, making it a *build-time* solution.

More than often, binding generators do *not* include a full parser for the programming language of origin, and they require to be fed a simplified version of the code being wrapped. This can be a nuisance for code bases which use complex language constructs unsupported by the binding generator.

A popular, general-purpose binding generator is SWIG[3] (Simple Wrapper and Interface Generator), which supports creating bindings for multiple programming languages, with Lua being just one more of the supported targets. SWIG uses its own C/C++ parser, which, while being complete [19], still fails on certain inputs.

There are also Lua-specific binding generators, of which the Lua-Users Wiki provides a comprehensive list [20]. The oldest is ToLua[4], maintained by staff from PUC-Rio like Lua itself. Being Lua the only target language, bindings generated by ToLua —and derivatives like ToLua++— are more idiomatic that those generated by e.g. SWIG. As a downside, ToLua's parser is quite limited, and only recognizes a subset of C and C++, to

[3] http://swig.org
[4] http://www.tecgraf.puc-rio.br/~celes/tolua/

the point that it is advised to provide a *cleaned header file* containing only declarations of data types, functions, and C++ classes recognizable by its parser.

2.2.3 Foreign Function Interfaces

In its most generic meaning, a FFI is any mechanism which allows a program written in a programming language to call code or use services written in another. In the Lua community, a FFI refers specifically to such a mechanism which works at run time. That is, it *does not* require using the Lua C API (subsection 2.2.1), it is not a binding generator used at build time (subsection 2.2.2), nor is it a wrapper over the Lua C API.

FFIs can be separated in two categories, depending on how they obtain the type information (functions names, arguments, return values; and the data types of all the values involved):

- FFIs which require the programmer to supply type information.
- FFIs which obtain type information in an automated fashion.

The canonical example of a FFI which requires the programmer to enter type information is the LuaJIT FFI module[5]. Its functionality is available as a regular Lua module which includes a number of support functions, including the ability to parse C-style declarations to obtain type information. Under the hood, the module generates the machine code for wrapper functions —of type `lua_CFunction` (c.f. subsection 2.2.1)— using the LuaJIT code generator, which are directly callable (example in listing 5). A standalone `luaffi`[6] module for the standard Lua VM exists, which has the same interface as the LuaJIT FFI module, and even reuses its code generator.

Where the LuaJIT FFI requires the programmer to enter C-style declarations of functions, LGI[7] uses the type information supplied by GObject Introspection [22], which provides access to most of the libraries included as part of the GNOME[8] desktop environment. The type information supplied by GObject Introspection is extracted from

[5] `http://luajit.org/ext_ffi.html`
[6] `https://github.com/jmckaskill/luaffi`
[7] `https://github.com/pavouk/lgi/`
[8] `http://gnome.org`

```
local ffi = require("ffi")
ffi.cdef("int printf(const char *fmt, ...);")
ffi.C.printf("Hello %s!\n", "world")
```

Listing 5: Using a C function with the LuaJIT FFI

the source code of the GNOME destop components at compile-time, and stored on disk in its own file format. The included `libgirepository` library is then used by LGI to allow the enumeration of the available modules and their contents, as well as the transparent invocation of functions from Lua. The metadata recorded by GObject Introspection when scanning source code includes additional information in specially formatted comments, which allows LGI to create idiomatic object oriented interfaces (c.f. listing 6).

```
local lgi = require("lgi")
local Gtk = lgi.Gtk

local w = Gtk.Window {
        title = "LGI Example",
        child = Gtk.Button { label = "Close",
                 on_clicked = Gtk.main_quit },
}
w:show_all()
Gtk.main()
```

Listing 6: Using the GTK+ user interface toolkit via LGI and GObject Introspection

2.3 Executable Formats

2.3.1 ELF

The "Executable and Linkable Format" (ELF, formerly called "Extensible Linking Format") is a common standard file format for executable programs, object code, shared libraries, and even core dumps. Since its publication as part of the System V Release 4 (SVR4) Application Binary Interface (ABI) specification [1, c. 4] it has been adopted by

many Unix-like (Solaris, most of the BSD variants, GNU/Linux), and non-Unix operating systems (most notably, OpenVMS, BeOS, and its successor Haiku).

2.3.2 DWARF

The DWARF Specification [2]

Chapter 3

Analysis & Design

Consider moving this to previous chapter

3.1 Analyzing DWARF

The DWARF format was first developed by the Bell Labs for the `sdb` debugger of System V Unix. Nowadays the formal specification is available under the GNU Free Documentation License (FDL), and its new versions have been discussed using public channels of communication, following a community-oriented process. Downloadable copies of all the version of the specification are available at `http://dwarfstd.org`.

In DWARF, all the data provided by debugging information is stored in a hierarchical tree-like structure, and each node of the tree is called a Debugging Information Entry (DIE). Each DIE consists of an identifying tag, and a series of attributes. The tag specifies the class to which an entry belongs, and the attributes define the characteristics of the entry. An entry, or a group of entries together, provide a description of an entity in the corresponding source code. The entries are contained in the `.debug_info` and `.debug_types` sections of an object file.

The attributes can reference another DIE, as shown in Figure 3.1, making it possible to create arbitrary links between nodes of the tree. Those explicit links are used extensively to group related entries together (for example the `DW_TAG_formal_parameter`

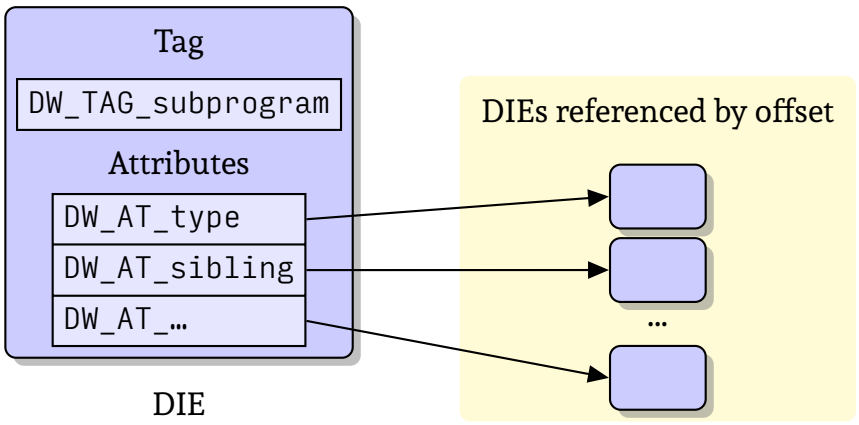

Figure 3.1: DIE attribute references.

entries, which describe the parameters of a function, are chained using `DW_AT_sibling` attributes), and to perform data deduplication of the debug information (for example, instead of making a new entry for the `int` type, only one is created and then referenced from other entries).

3.1.1 Debug Information Structure

This section provides simplified overview of the tree structure formed by the debug information entries of a program, which is enough for the goals of this project. For complete, detailed information, we refer to the DWARFv4 Specification [**?**].

Location of the Debug Information

The contents of ELF objects are described using *segments*, and *sections*. Segments contain information used at runtime for the execution of the code, and sections contain data about the code itself. Sections contain data used *offline* (i.e. not at runtime). Sections have arbitrary names and contents. Debugging information is not strictly needed at runtime for the execution of the program: it is considered ancillary, and is stored in sections.

Section	Contents
`.debug_types`	Stores DIEs for types.
`.debug_info`	Stores DIEs for executable program code (functions, mostly)
`.debug_line`	Maps of object code positions to source code line numbers
`.debug_pubnames`	Public function names and entry offsets in `.debug_info`
`.debug_pubtypes`	Public type names and entry offsets in `.debug_types`
`.debug_str`	String data (e.g. type, variable and function names)

Table 3.1: Main ELF sections used to store DWARF debugging information

As per the specification, the DWARF debugging information must be stored in the ELF objects in sections with the `.debug_` prefix. Each one of those sections stores a particular kind of information, as seen on Table 3.1.

Types

A type is represented as a DIE with one of the following tags:

- `DW_TAG_base_type`
- `DW_TAG_pointer_type`
- `DW_TAG_typedef`
- `DW_TAG_const_type`
- `DW_TAG_array_type`
- `DW_TAG_structure_type`
- `DW_TAG_union_type`
- `DW_TAG_enumeration_type`

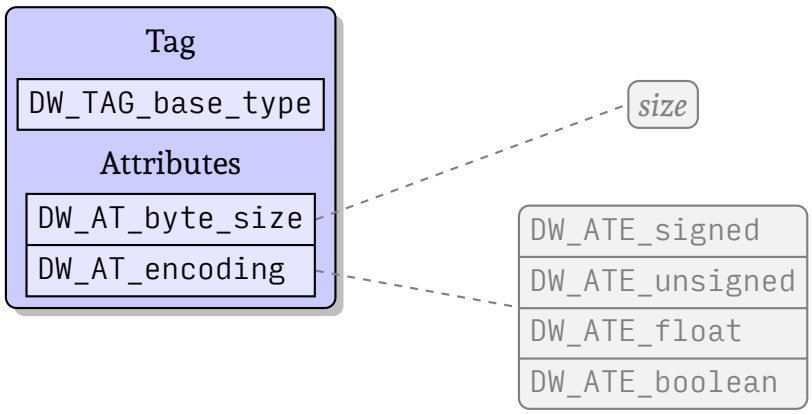

Figure 3.2: DIE describing a base type.

Base Types

Base types(Figure 3.2) are represented by DIEs with a `DW_TAG_base_type` tag. This covers all the C/C++ numeric types: signed and unsigned integers, floating point types (`float`, `double`), the `char` type, and the new integer-based types of C99 (`_Bool`, `int32_t`, etc.)

The size of the values is given using a `DW_AT_byte_size` attribute, in bytes. Sizes are the same reported by the C `sizeof` operator. A `DW_AT_encoding` attribute specifies how the values are used in the program. For example, `DW_ATE_boolean` corresponds with the `bool` type (or `_Bool`) in the source program. Table 3.2 summarizes how C types are represented by a base type DIE for the x86_64 architecture (sizes may vary in other architectures).

The rest of C/C++ base types are defined as aliases of these using `typedef`.

Pointer Types

Pointer types (Figure 3.3) are represented by DIEs with a `DW_TAG_pointer_type` tag, and a lone `DW_AT_type` attribute points to the DIE of the pointed-to type. This allows to represent pointers of/to any type. Multiple levels of indirection are represented by chains of `DW_TAG_pointer_type` DIEs. The example in Figure 3.3 exemplifies this situation, in what could be the representation of the `int**` when the rightmost "Type DIE" contains the information for the `int` type.

C Type	DW_AT_byte_size	DW_AT_encoding
bool	1	DW_ATE_boolean
char	1	DW_ATE_signed_char
unsigned char	1	DW_ATE_unsigned_char
short int	2	DW_ATE_signed
unsigned short int	2	DW_ATE_unsigned1
int	4	DW_ATE_signed
unsigned int	4	DW_ATE_unsigned
long int	8	DW_ATE_signed
unsigned long int	8	DW_ATE_unsigned
float	4	DW_ATE_float
double	8	DW_ATE_float
long double	16	DW_ATE_float

Table 3.2: DWARF representation of C base types for the x86_64 architecture

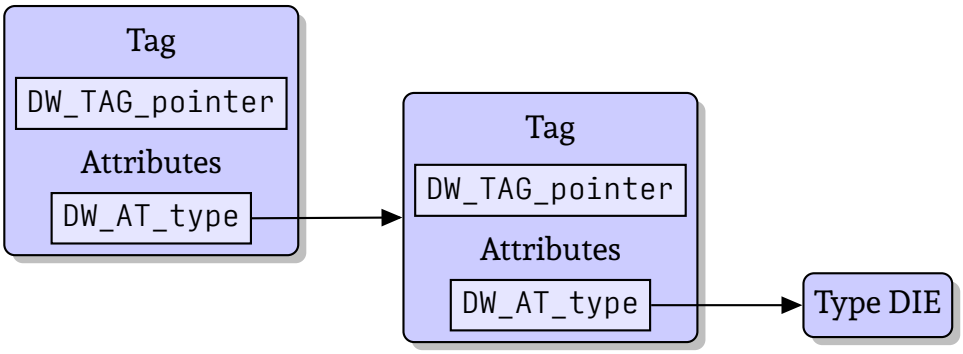

Figure 3.3: DIEs describing a Pointer-to-pointer-to type.

Constant Types

Flagging values of a type as constant is done in the same way as representing a pointer type: a DIE with a DW_TAG_const_type contains a DW_AT_type attribute with reference pointing to a type DIE. This way, the pointed type is marked as immutable.

Array Types

Array types (Figure 3.4) are represented by a DIE with a DW_TAG_array_type tag. A DW_AT_type attribute contains a reference to the type of the elements of the array. The space reserved for attribute values can be insufficient to store the number of elements

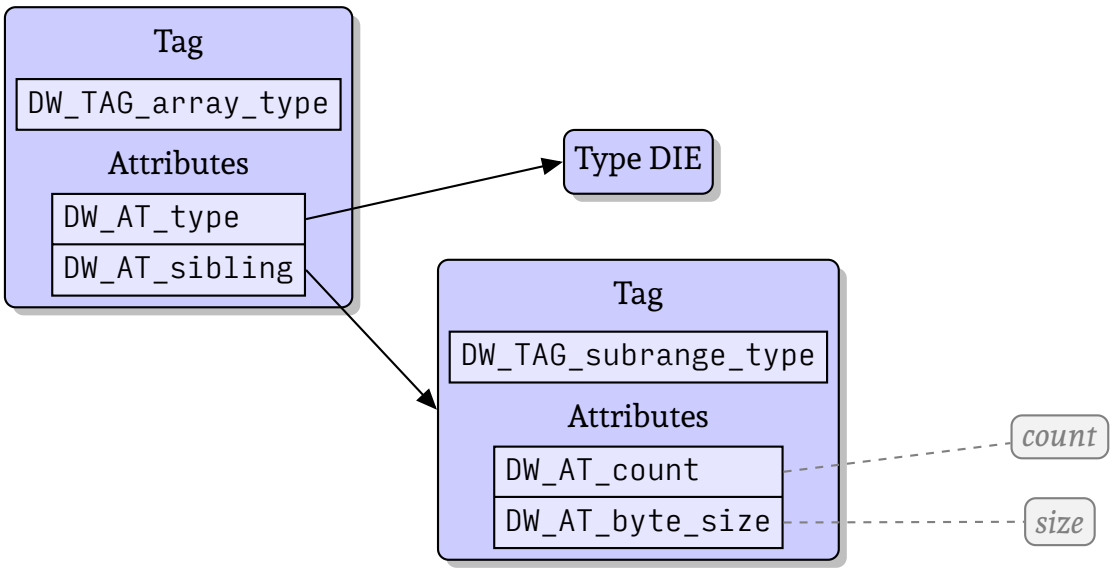

Figure 3.4: DIEs describing an array type.

in the array, so instead a chain of related DIEs is referenced using a `DW_AT_sibling` attribute, and one of the elements in the chain is `DW_TAG_subrange_type` DIE. The latter specifies the *size* —using a `DW_AT_byte_size` attribute— of the type needed to store the *count* of items, and the location of the value itself using a `DW_AT_count` attribute.

Type Aliases

As a convenience, programming languages usually allow defining new user-defined names for types. This is particularly handy to avoid repeating complex type declarations in the code of a program. In C type aliases are introduced with the `typedef` keyword:

```
/* "compare_function_t" is an alias to a function pointer
 ↪  type */
typedef int (*compare_function_t) (const void* a, const
 ↪  void*b);
```

A type alias (Figure 3.5) is represented by a DIE with tag `DW_TAG_typedef`[1] tag. The aliased type is referenced using a `DW_AT_type` attribute, and a `DW_AT_name` attribute

[1]The name of the tag hints that the DWARF format was designed with the C language in mind.

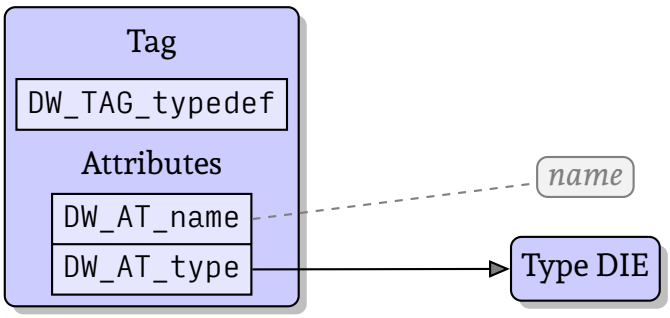

Figure 3.5: DIE describing a type alias.

provides the *name* of the aliased type.

Structured Types

Structured types, also known as *record types* or *compound data types* depending on the programming languages, are used-defined data structures which contain a collection of related elements, which can be accessed using a numeric index or (most commonly) by names given to them. Each element may also be called *field* or *member*. In C records are defined using the `struct` keyword:

```
struct Point {
  int x;  /* First member  */
  int y;  /* Second member */
};
```

A record type (Figure 3.6) is described using a set of related DIEs. At the top level, a DIE with tag `DW_TAG_structure_type` provides the size of the record type in bytes, by means of a `DW_AT_byte_size` attribute, and the name of the record type using a `DW_AT_name` attribute.

In order to describe the members, a DIE with tag `DW_TAG_member` is used for each of them. These DIEs contain a `DW_AT_type` attribute which references the DIE of the member type, a `DW_AT_name` attribute with name of the member, an attribute of type `DW_AT_data_member_location` specifies the *offset* of the member in memory, counted from the starting address of the record. The member DIEs are linked from the top level DIE using a chain of `DW_AT_sibling` links.

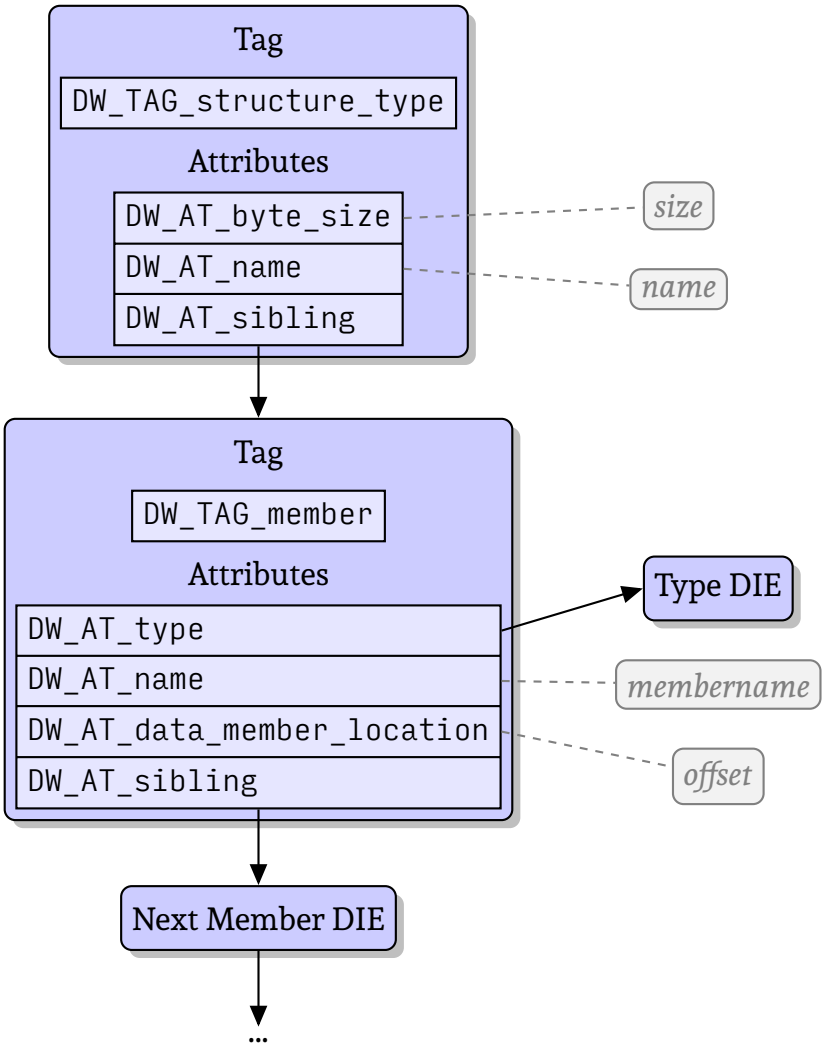

Figure 3.6: DIEs describing a structured type

Some programming languages allow defining *opaque* record types. The layout and fields of the type are only visible in the program module where the type is implemented. Such type is defined in C by omitting the specification of the `struct` fields in `.h` a header, and writing its complete description in the `.c` implementation:

```
/* File: opaque.h */
typedef struct _Opaque Opaque;
...

/* File: opaque.c */
struct _Opaque {
  uint8_t data_blob[100];
  ...
};
...
```

In this case, an additional DIE with tag `DW_TAG_structure_type` is present, *without* specifying the size of the data type with a `DW_AT_byte_size` attribute. Instead, a `DW_AT_declaration` attribute is added, flagging the type as declared, but not neccessarily defined. Optionally, if the complete definition is known by the compiler, it might add `DW_AT_specification` attribute pointing to the top level DIE which describes the type in full.

Union Types

Union types use the same structure of DIEs as record types, with a couple of small differeces: the tag for the top-level DIE is `DW_TAG_union_type`, and a `DW_AT_byte_size` attribute which indicates the size of the *largest member type*.

Enumeration Types

The representation of enumerations (Figure 3.7) shares a number of similarities with the representation of record and enumeration types: a top level DIE, in this case with tag `DW_TAG_enumeration_type`, describes the size of the integral type needed for values of the enumerated type using a `DW_AT_byte_size` attribute, and a `DW_AT_name` attribute contains the name of the type.

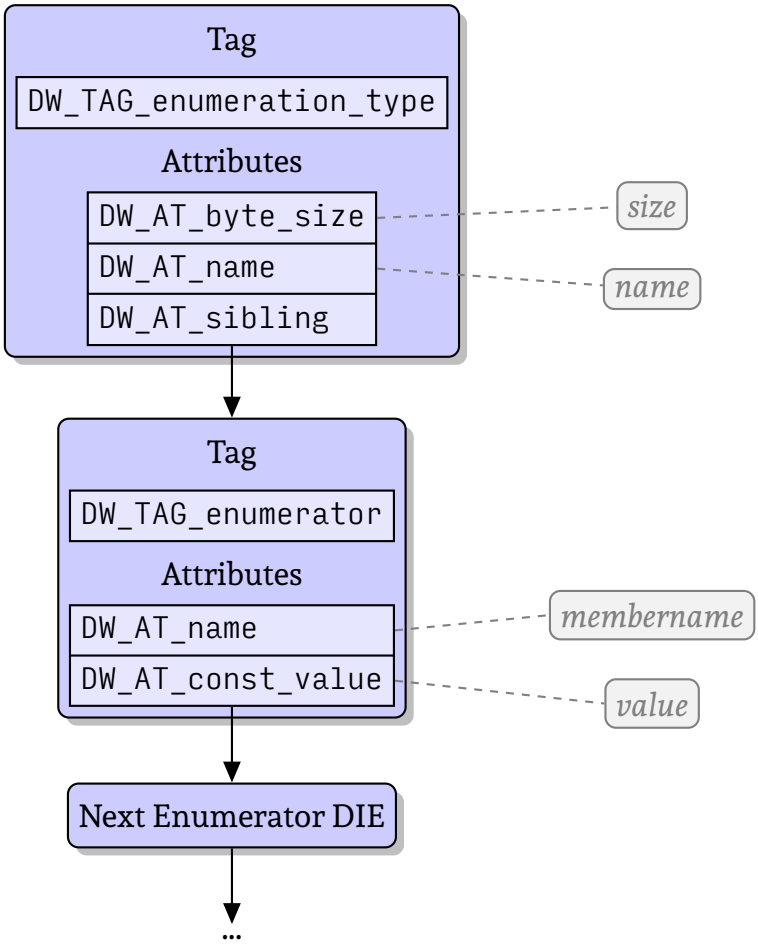

Figure 3.7: DIEs describing an enumeration type

The set of values is described by a DIE each, with tag `DW_TAG_enumerator`. Enumerators contain a `DW_AT_name` attribute with the name of the enumerator, and its associated value in a `DW_AT_constvalue` attribute.

Functions

The representation functions using DIEs uses the same structure as record and enumeration types: the top level entry contains general information about the function itself, using attributes, plus a link to a chain of DIEs. Function entries come in different flavors, depending on their tag:

- `DW_TAG_entry_point`

Describes a piece of executable that can be invoked in the same way as a function, which does not have entity as a function in the source program. For example, this can be used by a Pascal compiler to represent the code inside the `program` block.

- `DW_TAG_inlined_subroutine`
  Describes a function for which the code has been copied by the compiler inside another part of the program (i.e. *inlined*) as a performance optimization.
- `DW_TAG_subroutine`
  Describes normal functions. In object oriented languages is also used to describe methods, with the type the method applies to linked as type DIE of the first parameter.

The basic structure is shared among the three types, with the following attributes in the top level DIE:

- A `DW_AT_name` attribute provides the name of the function.
- A `DW_AT_type` attribute references the return type of the function. This is only present if the function returns a value.
- A `DW_AT_external` attribute, which contains a flag indicating whether the function is available to be used from compilation units other than the one where its definition lives. This flag is optional, and only required to be present (and have a non-zero value) for functions which are referenced from the `.debug_pubnames` ELF section.

From the top-level DIE, a `DW_AT_sibling` attribute references the first entry of the chain, which provides additional information about the function and its source code. Among all the entries in the chain, the ones with the `DW_TAG_formal_parameter` describe the parameters of the function. It is guaranteed that the entries describing the parameters will be in the chain in the same order as they appear in the source program, but they may not be consecutive. Each parameter entry contains the following attributes:

- A `DW_AT_name` attribute provides the name of the parameter.
- A `DW_AT_type` attribute references the DIE for the type of the parameter.

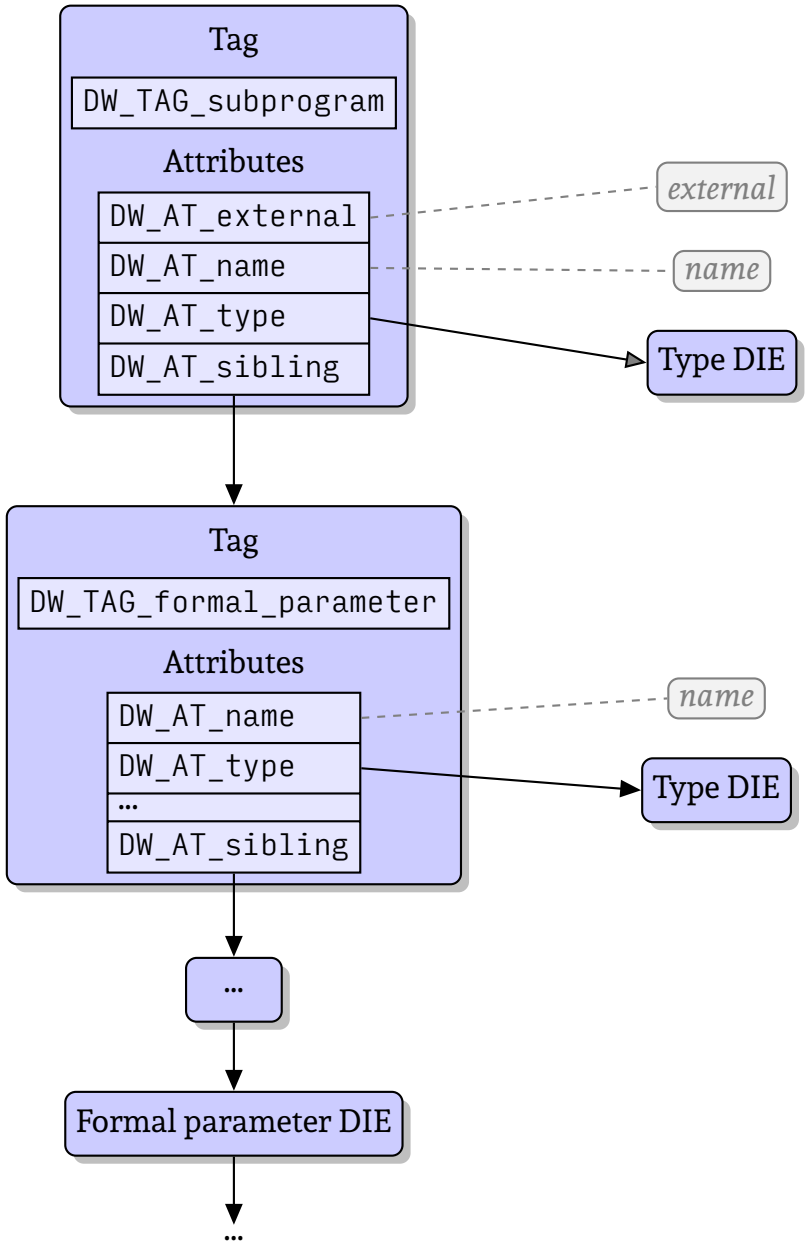

Figure 3.8: DIEs describing a function type

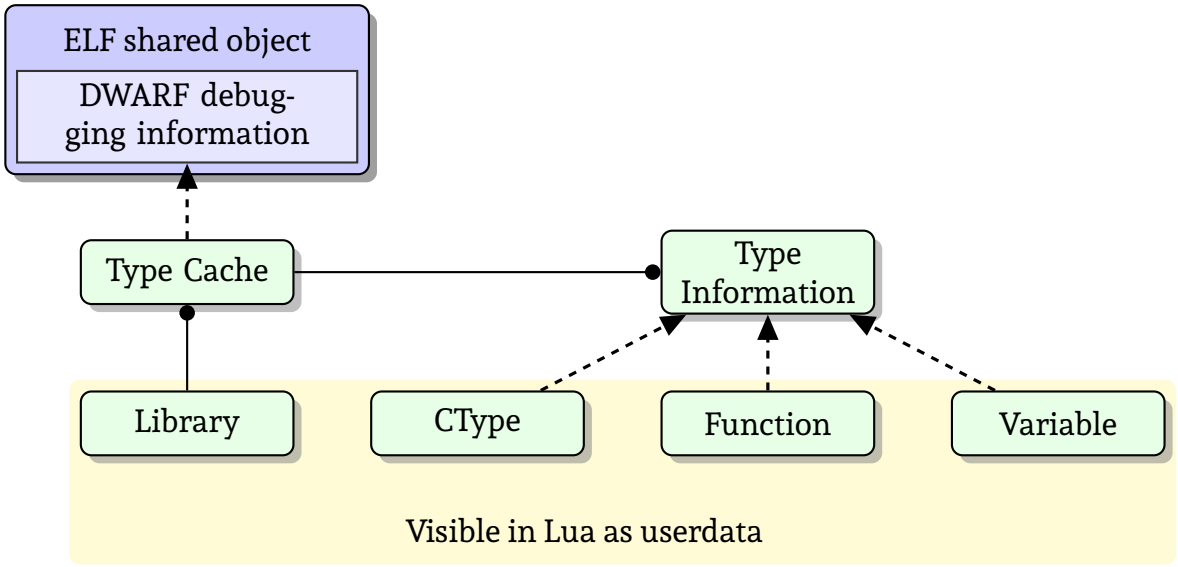

Figure 3.9: Architecture of EÖL.

3.2 Design

3.2.1 Naming

In the Lua community there is a certain tradition of naming projects after celestial bodies, or terms related to them —after all, Lua means *moon* in Portuguese—, but unfortunately the name initially chosen for the project was Eris —a dwarf planet, neither a planet nor a moon— was already being used by another Lua-related project[2]. A closer inspection showed that other dwarf planet names were already in use for software projects, so in the end it was needed to draw inspiration from a different area.

Eöl, also known as "The Dark Elf", is a fictional character in J. R. R. Tolkien's Middle-earth legendarium, who is said to be the elf with closest relationships with dwarves, and one of the first able to speak their language. EÖL can also be an backronym for "ELF Object Loader", which describes well the purpose of the developed solution.

[2]The Eris persistence system, `https://github.com/fnuecke/eris`

Chapter 4

Implementation

4.1 Project Source Structure

The EöL source code

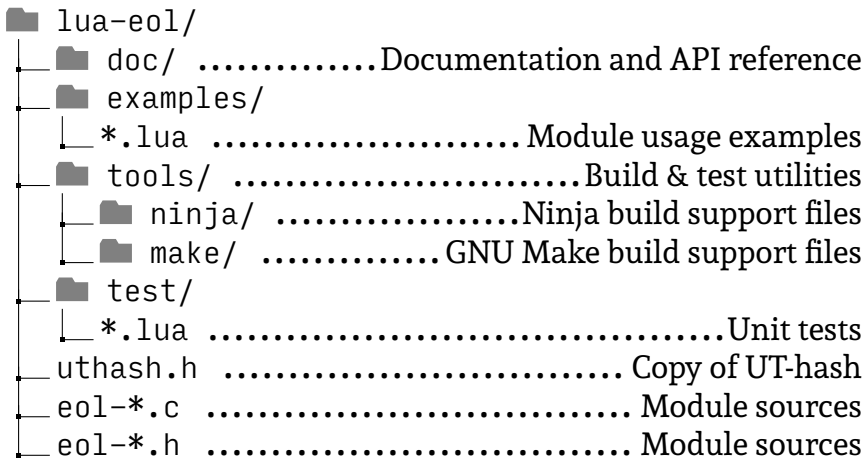

Figure 4.1: Source tree structure.

4.2 Type Representation

Converting values from C to Lua, and vice versa, is one of the most important tasks performed by EöL: C values need to be made accessible from Lua. Therefore, this information needs to be read from the DWARF debugging information (see Debug Information Structure), and kept around in a suitable data structure. This structure must be:

- Exhaustive, to hold all the needed information.
- Compact, to minimize memory usage.

Describing base types is possible using just an enumerated type: there is a fixed amount of them, and the characteristics (size, name, etc) are well known. The challenging part is representing user defined types (`struct`, `enum`, `union`), and derived types (pointers, arrays).

The data structure for describing types is `EolTypeInfo` (listing 7). It is a tagged `struct`, with the tag indicatingthe type kind (`EOL_TYPE_S32` for 32-bit signed integers, `EOL_TYPE_STRUCT` for a `struct`, etc; the complete list of values can be seen in listing 8 on page 41). The contained data will vary depending on the value of the *kind* tag. The members for all possible values are grouped in an `union` in order to make them share the same memory space.

```
struct _EolTypeInfo {
  EolType type;
  union {
    struct TI_base     ti_base;
    struct TI_pointer  ti_pointer;
    struct TI_typedef  ti_typedef;
    struct TI_const    ti_const;
    struct TI_array    ti_array;
    struct TI_compound ti_compound;
  };
};
typedef struct _EolTypeInfo EolTypeInfo;
```

Listing 7: `EolTypeInfo`.

The following sections describe in detail the members of `EolTypeInfo`.

4.2.1 Base Type Representation

```
struct TI_base {
  char               *name;
  uint32_t            size;
};
```

Even though it is sufficient to provide type kind codes for all the base types as discussed before, providing the possibility of querying their `name` and `size` is a convenient feature, at a very small cost: the `EolTypeInfo` value for each one of the base types is a singleton, defined as follows:

```
/* File: eol-typing.h */
extern const EolTypeInfo* eol_typeinfo_u32;

/* File: eol-typing.c */
const EolTypeInfo* eol_typeinfo_u32 = &((EolTypeInfo) {
  .kind         = EOL_TYPE_U32,
  .ti_base.name = "uint32_t",
  .ti_base.size = sizeof (uint32_t),
});
```

In practice, to avoid writing the definitions of all the base types, the C preprocessor and a couple generator macros are used (see subsection 4.4.1).

4.2.2 Pointer Representation

```
struct TI_pointer {
  const EolTypeInfo *typeinfo;
};
```

Pointers are represented by referencing the `EolTypeInfo` of the pointed-to type. Thus, it is the only member in `struct TI_pointer`. The size of a pointer value is platform dependent, but well known and constant for each platform, and is the value of the C expression `sizeof(void*)`.

4.2.3 Array Representation

```
struct TI_array {
  const EolTypeInfo *typeinfo;
  uint64_t           n_items;
};
```

Arrays are represented by referencing the `EolTypeInfo` of the array items, plus the number of items (`n_items`) present in the array. The size of an array value can be calculated multiplying the size of the item type by the number of items in the array.

4.2.4 User Defined Type Representation

```
struct TI_compound {
  char             *name;
  uint32_t          size;
  uint32_t          n_members;
  EolTypeInfoMember members[];
};
```

This record type represents all user defined types: enumerated types (`enum`), record types (`struct`), and union types (`union`):

`name`
: User defined types are usually given a name, but it is optional and in this case the value will be `NULL`.

`size`
: Contains the size of the type, in bytes.

`n_members` / `members`
: Count of members (or enumerators, for `EOL_TYPE_ENUM`) in the type, and an array contaning their descriptions. Using a flexible array member, allows usage of a single chunk of memory for the `EolTypeInfo` itself and the items in the array.

The auxiliar `EolTypeInfoMember` type is defined as follows:

```
typedef struct {
  const char            *name;
  union {
    int64_t              value; /* enum */
    struct {                    /* union, struct */
      uint32_t           offset;
      const EolTypeInfo *typeinfo;
    };
  };
} EolTypeInfoMember;
```

This always contains the (optional) `name` of the types, and usage of the remaining fields varies with the type being described:

- For `EOL_TYPE_STRUCT`, the `offset` of the member (in bytes, from the beginning of the record) and a pointer to its type information (`typeinfo`) are used.
- For `EOL_TYPE_UNION`, the `offset` is ignored, and only the type information of the member (`typeinfo`) is used.
- For `EOL_TYPE_ENUM`, only the `value` associated with the enumerator is used.

An `union` is used to make fields share the same memory space.

4.2.5 Type Alias Representation

```
struct TI_typedef {
  char              *name;
  const EolTypeInfo *typeinfo;
};
```

Type aliases assign a name to an arbitrary type. They are represented by the `name` and a pointer to the `EolTypeInfo` of the type.

4.2.6 Read-only Type Representation

```
struct TI_const {
  const EolTypeInfo *typeinfo;
};
```

Flagging a type as read-only (i.e. using the `const` type qualifier in C) is represented in the same way as pointers (subsection 4.2.2): by keeping a pointer to the `EolTypeInfo` that is read-only.

4.3 Memory owners and life-cycle

Problem: functions that allocate memory

```
struct point {
        int x;
        int y;
};
```

In Lua, we could create an instance like this:

```
local Geometry = eol.wrap("libgeometry.so")
-- Creates heavy userdata, the Lua GC handles freeing memory
local point = Geometry.point { x = 1, y = -1 }
```

Complete this section

4.4 Miscellanea

This section describes assorted techniques used in the implementation of EOL which do not fall under a common umbrella, yet they deserve to be mentioned.

4.4.1 Preprocessor "Generator Macros"

This is a programming pattern used thorough the code of EOL: the C preprocessor is used in a convoluted way as a rudimentary code generator using lists of related elements. First, a macro of related elements is defined (*enumerator macro*, from now on), and it must accept the identifier for another macro (the *generator macro*) as a parameter. Each element in the enumerator macro is an expansion of the generator, passing the parameters needed by the generator.

In order to better understand how generator macros work, let us walk through a complete example adapted from the EOL source code. The following macro expands into a function which checks the type of an `EolTypeInfo` — it is the *generator*:

```
#define MAKE_TYPEINFO_IS_TYPE(suffix, name, ctype)            \
    bool eol_typeinfo_is_ ## name (const EolTypeInfo *înfo) \
    { return info->type == EOL_TYPE_ ## suffix; }
```

In generator macros like this, the concatenation operator (##) of the preprocessor is used extensively to build pieces of valid C code. The example shows how the `name` parameter is concatenated to create the name of the generated function, and the `suffix` parameter is concatenated to create a valid `EolType` (listing 8 on page 41) value. A valid expansion of the above macro is:

```
MAKE_TYPEINFO_IS_TYPE (S32, s32, int32_t)
```

which generates the following valid C function:

```
bool eol_typeinfo_is_s32 (const EolTypeInfo *info)
{ return info->type == EOL_TYPE_S32; }
```

The *enumerator macro* is made by grouping a set of macro expansions like the one above. The key is using a generic name for the generator macro, which will be passed as a parameter. The next listing defines an enumerator which expands a generator `F` for each signed integer type:

```
#define INTEGER_S_TYPES(F) \
    F (S8,  s8,  int8_t  ) \
    F (S16, s16, int16_t ) \
    F (S32, s32, int32_t ) \
    F (S64, s64, int64_t )
```

Using the above definition, an expansion of the *enumerator macro* causes multiple expansions at once of the *generator macro* passed as `F`, which in turn creates as many functions as elements in the enumerator macro. In this example, using generator macros reduces the amount of code that the programmer must write manually close to one fourth of the original.

Another use case for generator macros is creating the code for cases in a `switch` statement. Instead of constructing the code for an entire function at a time, only a single `case` label and its associated statements are generated. This is done in the following example:

```
#define MAKE_SIGNED_TYPE_CASE(suffix, name, ctype) \
    case EOL_TYPE_ ## suffix: return true;

bool eol_type_is_signed (EolType type) {
  switch (type) {
    INTEGER_S_TYPES (MAKE_SIGNED_TYPE_CASE)
    default: return false;
  }
}
```

```
typedef enum {
  EOL_TYPE_VOID,    /* void        */
  EOL_TYPE_BOOL,    /* _Bool       */
  EOL_TYPE_S8,      /* int8_t      */
  EOL_TYPE_U8,      /* uint8_t     */
  EOL_TYPE_S16,     /* int16_t     */
  EOL_TYPE_U16,     /* uint16_t    */
  EOL_TYPE_S32,     /* int32_t     */
  EOL_TYPE_U32,     /* uint32_t    */
  EOL_TYPE_S64,     /* int64_t     */
  EOL_TYPE_U64,     /* uint64_t    */
  EOL_TYPE_FLOAT,   /* float       */
  EOL_TYPE_DOUBLE,  /* double      */
  EOL_TYPE_TYPEDEF, /* typedef … T */
  EOL_TYPE_CONST,   /* const T     */
  EOL_TYPE_POINTER, /* T*          */
  EOL_TYPE_ARRAY,   /* T …[n]      */
  EOL_TYPE_STRUCT,  /* struct …    */
  EOL_TYPE_UNION,   /* union …     */
  EOL_TYPE_ENUM,    /* enum …      */
} EolType;
```

Listing 8: `EolType` enumeration.

C Construct	DWARF DIE	EÖL Type
`void`	ø	`EOL_TYPE_VOID`
`bool`	`DW_TAG_base_type`	`EOL_TYPE_BOOL`
`int8_t`	`DW_TAG_base_type`	`EOL_TYPE_S8`
`uint8_t`	`DW_TAG_base_type`	`EOL_TYPE_U8`
`int16_t`	`DW_TAG_base_type`	`EOL_TYPE_S16`
`uint16_t`	`DW_TAG_base_type`	`EOL_TYPE_U16`
`int32_t`	`DW_TAG_base_type`	`EOL_TYPE_S32`
`uint32_t`	`DW_TAG_base_type`	`EOL_TYPE_U32`
`int64_t`	`DW_TAG_base_type`	`EOL_TYPE_S64`
`uint64_t`	`DW_TAG_base_type`	`EOL_TYPE_U64`
`float`	`DW_TAG_base_type`	`EOL_TYPE_FLOAT`
`double`	`DW_TAG_base_type`	`EOL_TYPE_DOUBLE`
`typedef`…	`DW_TAG_typedef`	`EOL_TYPE_TYPEDEF`
`const`…	`DW_TAG_const_type`	`EOL_TYPE_CONST`
…*	`DW_TAG_pointer_type`	`EOL_TYPE_POINTER`
…[n]	`DW_TAG_array_type`	`EOL_TYPE_ARRAY`
`struct`…	`DW_TAG_structure_type`	`EOL_TYPE_STRUCT`
`union`…	`DW_TAG_union_type`	`EOL_TYPE_UNION`
`enum`…	`DW_TAG_enumration_type`	`EOL_TYPE_ENUM`

Table 4.1: Mapping of C types, DWARF DIEs and `EolType`.

Chapter 5

Conclusions

Installation

1 Prerequisites

Instead of providing its own implementation for certain functionality, EÖL uses existing, proved software components.

Component	Version	Required	Optional	Bundled
Lua	5.3	⋆		⋆
LuaBitOp	1.0.2		⋆	⋆
`libdwarf`	20150507	⋆		⋆
`libelf`	0.152	⋆		
`readline`	5.0		⋆	
`libffi`	3.1		⋆	

Table 1: Dependencies

Table 1 shows the dependencies expected to be installed in the system. The items marked (⋆) as *bundled* are not included in the source repository, but the build system includes support for downloading tarballs with the source code and doing a local build. When enabled, bundled dependencies will be automatically downloaded, built, and used instead instead of the versions provided by the system. In the case of using `libdwarf` bundled, it will be statically linked. See subsection 2.1 for instructions to enable bundled libraries. This is particularly useful for systems which do not provide Lua 5.3 packages (for example, the case Debian and Ubuntu at the time of writing).

Table 2 shows required packages as provided by popular GNU/Linux distributions. Some versions of Debian (and derivatives like Ubuntu) include only a static version of `libdwarf` in the packages, most likely not built as Position-Independent Code (PIC), which is a requirement.

Distribution	Installation Command	Packages
Debian, Ubuntu	`apt-get install`	`libdwarf-dev ninja-build`
Arch Linux	`pacman -S`	`libdwarf ninja lua`

Table 2: Dependency packages in popular GNU/Linux distributions.

2 Building

The build process follows the convention pioneered by GNU Autotools, in which an autoconfiguration script (`configure`) is run first to inspect the system, determine which optional components are to be enabled at build time, and generate the needed build files. In short, building EÖL is done by executing the following commands from the top level source directory:

```
./configure
make
```

or, using Ninja:

```
./configure
ninja
```

2.1 Autoconfiguration

The `configure` script accepts a number of command line paramers, which determine how the systm is built. In most cases the script will figure out automatically whether the required prerequisites (section 1) are available, and whether the bundled versions should be used. In case the detection failed, the following are some of the command line options can be passed to `configure`:

`--enable-bundled-libdwarf`
: Uses the bundled `libdwarf` instead of trying to use the one provided by the system.

`--enable-bundled-lua`
: Uses the bundled Lua distribution instead of trying to use the one provided by the system.

`--enable-ffi`
: Always use `libffi` to perform function calls, and do not build support their JIT compilation.

`--jit-arch=ARCH`
: Skip detecting the operating system and processor, and use JIT compilation for the supplied architecture (`ARCH`). It is possible to obtain a list of the supported architectures by running `./configure --jit-arch=help`.

In order to obtain a complete list of all the command line options that the script accepts, use:

```
./configure --help
```

Glossary

B

backronym

A *backward acronym* is an acronym constructed in reverse, by creating a new phrase to fit an existing word, name, or acronym. 31

C

closure

Technique for implementing lexically scoped name binding in languages with first–class functions.. 11

D

data deduplication

Any technique which eliminates duplicate copies of repeating data in order to improve storage utilization. 20

dynamic dispatch

Process of selecting a concrete implementation of a polymorphic method (or function) at runtime. It is typically used in object oriented languages when different classes contain different implementations of the same method due to inheritance . 10

dynamic programming

Problem solving method —and programming technique— which solves a complicated problem by breaking it up in smaller problems in a recursive manner . xvii, 13

E

emulation

Piece of hardware or software that enables one computer system (called the *host*) to behave like another computer system (called the *guest*) which enables the host system to run software or use peripheral devices designed for the guest system. 8

F

Fibonacci number

Number from the sequence $1, 1, 2, 3, 5, 8, 13, ...$, given by the recurrence relation $F_n = F_{n-1} + F_{n-2}$, with $F_1 = 1$, and $F_2 = 1$, as defined by the Italian mathematician Leonardo Fibonacci. 11

first–class value

In programming language design, an entity which supports all the operations generally available to other entities of a laguage, typically: being passed as a parameter, returned from a function, and assigned to a variable. 11

flexible array member

Feature introduced in the C99 standard of the C programming language which allows the last member of a `struct` to be an array of an unspecified dimension. Space needed by the array does not contribute to the size of the `struct` type, and must be manually accounted for when allocating the `struct` from the heap. 36

I

Instruction Set Architecture

Part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification of the machine language, and the native commands implemented by a particular processor.. 8, 54

L

LuaJIT

Just-In-Time compiler (JIT) for the Lua programming language. It is a third-party, independent implementation of the Lua VM created and maintained by Mike Pall, who allegedly was born in planet Krypton. Available at `http://luajit.org`.. 16

Lua-Users Wiki

Community operated *wiki* site which contains resources for Lua development, written by users of the programming language themselves. URL address: `http://lua-users.org/wiki`. 15

M

memoization

Optimization technique which stores the results of expensive function calls, and returns the previously calculated value when the same inputs occur again. xvii, 13

O

object oriented

Programming paradigm based on the concept of *objects*, which are data structures that encapsulate both data, and its behavior. 29

P

Pascal

Procedural programming language designed in the late 60s by Niklaus Wirth to encourage good programming practices. 11, 29

T

Toyota

Japanese car manufacturer. 5

transpiler

Type of compiler that takes source code of a programming language as its input, and produces a different source code as output, usually in a different programming language. Also known as *source-to-source compiler*, or *transcompiler*. 14

Type Unit Entry

A particular kind of DWARF DIE that contains information about a data type. 54

Acronyms

A

API

Application Programming Interface. 12

D

DIE

Debugging Information Entry. 19

DWARF

Debugging With Attributed Record Formats. 2, 3, 19

E

ELF

Executable and Linkable Format. 2, 3, 20

F

FDL

Free Documentation License. 19

FFI

Foreign Function Interface. 9, 16

I

ISA

Instruction Set Architecture. 8, *Glossary:* Instruction Set Architecture

J

JIT

Just-In-Time. 8, 9, 51

P

PIC

Position-Independent Code. 45

PUC-Rio

Pontifícia Universidade Católica do Rio de Janeiro. 11, 15

T

TUE

Type Unit Entry. *Glossary:* Type Unit Entry

V

VM

Virtual Machine. vii, 3, 8, 9, 12, 16

Bibliography

[1] System V Application Binary Interface Edition 4.1
The Santa Cruz Operation, *AT&T*, and *The 88open Consortium*
http://www.sco.com/developers/devspecs/gabi41.pdf
Accessed: May 3rd, 2015.

[2] DWARF Debugging Information Format Committee
DWARF Debugging Information Format Version 4
http://dwarfstd.org/doc/DWARF4.pdf
Accessed: May 5th, 2015.

[3] Introduction to the DWARF Debugging Format
Michael J. Eager
http://dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf
Accessed: May 5th, 2015.

[4] How Debuggers Work: Part 3 - Debugging Information
Eli Bendersky
http://eli.thegreenplace.net/2011/02/07/how-debuggers-work-part-3-debugging-information/
Accessed: May 5th, 2015.

[5] Programming in Lua
Roberto Ierusalimschy
Lua.org, Third Edition (January 3rd, 2013), ISBN 859037985X.

[6] Programming in Lua
Roberto Ierusalimschy
Lua.org, First Edition (December 2003), ISBN 8590379817.
http://www.lua.org/pil/contents.html

[7] Lua: About
`http://www.lua.org/about.html`
Checked April 20th, 2015.

[8] Lua 5.3 Reference Manual
Roberto Ierusalimschy, *Luiz Henrique de Figueiredo*, and *Waldemar Celes*
`http://www.lua.org/manual/5.3/manual.html`
Accessed: April 29th, 2015.

[9] The Implementatin of Lua 5.0
Roberto Ierusalimschy, *Luiz Henrique de Figueiredo*, and *Waldemar Celes*
Journal of Universal Computer Science 11 #7 (2005)
`http://www.jucs.org/jucs_11_7/the_implementation_of_lua`

[10] ffi.* API Functions
Mike Pall
`http://luajit.org/ext_ffi_api.html`
Checked May 11th, 2015.

[11] FFI Semantics
Mike Pall
`http://luajit.org/ext_ffi_semantics.html`
Accessed: May 11th, 2015.

[12] Getting Started with Scrumban
`http://www.aboutscrumban.com/how-to-start-using-scrumban/`
Accessed: May 5th, 2015.

[13] TAP Specification
Michael G. Schwern, and *Andy Lester*
`http://testanything.org/tap-specification.html`
Accessed: May 4th, 2015.

[14] A Consumer Library Interface to DWARF
David Anderson
`https://github.com/Distrotech/libdwarf/blob/distrotech-libdwarf/libdwarf/libdwarf2.1.pdf`
Accessed: May 11th, 2015.

[15] A Producer Library Interface to DWARF
David Anderson
`https://github.com/Distrotech/libdwarf/blob/distrotech-libdwarf/libdwarf/libdwarf2p.1.pdf`
Accessed: May 11th, 2015.

[16] Ninja documentation
Evan Martin
`http://martine.github.io/ninja/manual.html`
Accessed: May 11th, 2015.

[17] GNU Make: A Program for Directing Recompilation
Free Software Foundation, ISBN 1-882114-83-3.
`https://www.gnu.org/software/make/manual/make.pdf`
Accessed: May 13th, 2015.

[18] `uthash` User Guide
Troy D. Hanson
`https://troydhanson.github.io/uthash/userguide.html`
Accessed: May 15th, 2015.

[19] SWIG 3.0 Documentation
`http://swig.org/Doc3.0/SWIGDocumentation.html`
Accessed: June 25th, 2015.

[20] Lua-Users Wiki: Binding Code To Lua
`http://lua-users.org/wiki/BindingCodeToLua`
Accessed: June 25th, 2015.

[21] The race for speed part 1: The JavaScript engine family tree
John Dalziel, CreativeJS
`http://creativejs.com/2013/06/the-race-for-speed-part-1-the-javascript-engine-`
Accessed: August 16th, 2015.

[22] GNOME Wiki: GObject Introspection
`https://wiki.gnome.org/Projects/GObjectIntrospection`
Accessed: August 15th, 2015.

[23] `lua-l` mailing list, August 2009: LuaJIT performance
Mike Pall
`http://lua-users.org/lists/lua-l/2009-08/msg00151.html`
Accessed: August 22nd, 2015