A study on scripting language APIs

Hisham H. Muhammad

Advisor: Roberto Ierusalimschy

Pontifícia Universidade Católica do Rio de Janeiro
Centro Técnico Científico
Departamento de Informática

November 9, 2006

Abstract

Applications written in two programming languages, in order to optimize parts where performance is critical or to obtain extensibility through user-written scripts, are commonplace nowadays. There are several ways to obtain this kind of interoperability; ideally, however, a language should provide a foreign language interface (FLI), allowing programmers to send and receive both data and function calls to code written in another language.

This work discusses the main issues involving the design of APIs for integration of language environments within C applications. We present the main problems faced in the interaction between code executed in an environment with inherently dynamic characteristics such as a scripting language and C code. We compare approaches employed by five languages when handling communication between the data spaces of C and embedded runtime environments and the consequences of these approaches in memory management, as well as sharing of code between the C application and that from the scripting language.

We illustrate the differences of the APIs of those languages and their impact in the resulting code of a C application through a case study. Different scripting languages were embedded as plugins for a library, which on its turn exposes to client applications a generic scripting API. This way, the code of each plugin allows us to observe in a clear and isolated way the procedures adopted by each language for function calls, registration of C functions and conversion of data between the environments.

Contents

1 Introduction **5**
1.1 Objectives . . . 6
1.2 Text structure . . . 6

2 Interaction between programming languages **9**
2.1 Code translation . . . 9
2.2 Sharing virtual machines . . . 10
2.3 Language-independent object models . . . 11
2.4 C as an intermediate language . . . 12
2.5 Interfaces to C . . . 12
2.6 Scripting languages . . . 14

3 Scripting language APIs **17**
3.1 Data transfer . . . 17
3.1.1 Python . . . 19
3.1.2 Ruby . . . 23
3.1.3 Java . . . 27
3.1.4 Lua . . . 30
3.1.5 Perl . . . 32
3.1.6 Comparison . . . 35
3.2 Garbage collection . . . 36
3.2.1 Python . . . 37
3.2.2 Ruby . . . 40
3.2.3 Java . . . 41
3.2.4 Lua . . . 43
3.2.5 Perl . . . 45
3.2.6 Comparison . . . 45
3.3 Calling functions from C . . . 46
3.3.1 Python . . . 47
3.3.2 Ruby . . . 48
3.3.3 Java . . . 50
3.3.4 Lua . . . 52
3.3.5 Perl . . . 53

3.3.6 Comparison 54
3.4 Registering C functions 55
3.4.1 Python 55
3.4.2 Ruby 57
3.4.3 Java 58
3.4.4 Lua 60
3.4.5 Perl 61
3.4.6 Comparison 63

4 Case study: LibScript 65
4.1 LibScript 65
4.1.1 Architecture of LibScript 65
4.1.2 Main library API 69
4.1.3 Plugins API 72
4.2 Implementation of plugins 73
4.2.1 Representation of states 74
4.2.2 Termination of states 76
4.2.3 Passing arguments 77
4.2.4 Function calls 85
4.2.5 Capturing errors 87
4.3 Conclusions 88

5 Conclusions 91

A The LibScript API 99
A.1 Startup and termination 99
A.2 Function registration 99
A.3 Arguments buffer 99
A.4 Running code 100
A.5 API exported by plugins 101

Chapter 1

Introduction

There are many situations in which it is necessary or interesting to have interaction between programs written in different languages. A typical case is the use of external libraries, such as graphic toolkits, APIs for database access, or even operating system calls. Another scenario involves applications developed using more than one programming language, in order to optimize parts where performance is critical or to allow extensibility through scripts written by end-users.

Regardless of purpose, communication between programs written in different languages brings up a number of design issues, not only in the development of the applications, but of the languages themselves. There are many ways to obtain this kind of interoperability, from translation of code of a language to another to the use of a common virtual machine. Ideally, however, a language should provide a foreign language interface (FLI) that allows programmers to send and receive both calls and data to another language [11]. Among the factors that should be taken into account when developing such an interface are the differences between type systems, memory management issues (such as garbage collection and direct access to pointers) and concurrency models. Beyond dealing with semantic differences, the design of an interface between languages involves pragmatic issues such as the balance between safe isolation of the runtime environments, performance and simplicity of the resulting API.

We can observe in existing implementations of FLIs a number of approaches to these problems. Indeed, FLIs for different languages (or even different revisions of a single language) tend to be very different from each other. Still, it is possible to trace parallels among the various techniques employed, since the fundamental problems that they address are the same.

Because of the popularity of the C language and the support it enjoys in most popular operating systems, a considerable number of implementations of foreign language interfaces are, in practice, C APIs. Besides, an interaction model for programming languages that has become especially relevant nowadays is that between statically typed compiled languages, such as C, and dynamically typed interpreted languages, as proposed by Ousterhout [32]. These two classes of languages have fundamentally different goals. Statically typed languages are usually implemented with high performance in mind and focus on lower-level

programming. In contrast, scripting languages tend to be implemented as interpreters or virtual machines, and make extensive use of high-level constructs, such as lists and hashes, as basic types. These complementary features have made the two-language programming model popular, in which a lower-level language is used for development of components, which are then connected through a higher-level language.

1.1 Objectives

This work discusses the main issues involving the design of APIs for integration of run-time environments of scripting languages in C applications. We present the main problems faced in the interaction between code executed in an environment with inherently dynamic characteristics such as that from a scripting language with C code. Besides being currently the most popular class of languages for multi-language development, typical features of scripting languages such as garbage collection and dynamic typing illustrate well the problems that arise in the communication between different programming environment, since these features are absent in C. Languages with static typing may present similar needs for type conversion, but this problem tends to be simplified by the definition of equivalent types in the API and compile-time inference (as can be observed in C APIs for Ada and Fortran). Functional languages have additional concerns related to side effects in C code, but this is equivalent to the paradigm break problem caused by handling of I/O commonly faced by those languages.

This study consists of two parts. In the first part, we performed an in-depth analysis of a set of C APIs provided by four scripting languages – namely, Python [45], Perl [47], Ruby [40], Lua [15] – as well as the API provided by the Java language [13]. Unlike the others, Java uses static typing, but like them it is based on a virtual machine model, features automatic memory management and allows dynamic loading of code. This allows us to observe how typing affects the design of the API.

In the second part, we illustrate the differences between the APIs of those languages and the impact of those in the resulting code of a C application through a case study. We performed a comparison between scripting language APIs through a concrete example, in order to present implementations in each of the studied languages side by side. The example consists of a generic scripting library, called LibScript, and a series of plugins that interface to the different languages. This way, the code of each plugin allows us to observe in a clear and isolated way the procedures used in each language for function calls, registration of C functions and data conversion between environments.

1.2 Text structure

This work is structured as follows. In Chapter 2, we discuss the various approaches for interaction between code written in different programming languages. Starting from an overview, the focus will then concentrate on the most commonly used foreign language in-

terface: interfaces with the C language. We will discuss the problems commonly presented in the communication with C code and the programming models that appeared with its popularization in the integration with scripting languages. In Chapter 3, we present in detail C APIs for a set of scripting languages. When discussing these interfaces, the different solutions employed for the main problems involving interaction between C and dynamic environments are brought up. Chapter 4 exercises these APIs through a case study: a plugin-based library that offers a simplified, uniform interface for scripting languages. By examining the implementation of each plugin, we can compare the APIs for each language performing equivalent operations. Finally, in Chapter 5, conclusions reached through this work are presented, as well as possible directions for future work.

Chapter 2

Interaction between programming languages

The approaches applied to the interaction of different programming languages vary considerably, but it is possible to identify some of the more typical techniques: language translation, from one language to the other or of both to a third; communication through an intermediate protocol or language; sharing a common execution environment, be it a virtual machine or through call conventions; and foreign language interfaces.

2.1 Code translation

Allowing the use of two languages in a program through the translation of the code of one of them to the other minimizes the problem of communication between the parts of program written in different languages, since the final program will use a single data space. On the other hand, by having to describe a language in terms of the other, the semantic differences of their constructs may become a problem. If the target language lacks constructs offered by the source language, simulating them may be costly.

A typical example of this problem is the complexity added by the simulation of higher-order functions and tail recursion when translating code from functional languages to one that does not have those features. Tarditi et al. [39] describe the development of a translator of Standard ML to ANSI C. Their measurements have exposed the cost of adapting the features of ML to C, resulting in code that is in average 2 times slower than the that generated by the native ML compiler. In [42], similar challenges are discussed in the translation of ML to Ada: in the adopted approach, the process has an intermediate step where higher-order constructs are “flattened” to first-order constructs using records, so that they could be represented in Ada.

Besides problems such as this, differences in the representation of data is also something to be handled when translating one language to another. In the particular case of C, its lower-level facilities for memory manipulation allow the description of data structures for higher-level languages without too much trouble. This makes C a frequent candidate for

use as a portable low-level representation of code. The Glasgow Haskell Compiler offers, as an alternative to the generation of native code, generation of C code for use with GCC [19]. One of the advantages of this feature is to allow the bootstrapping of the compiler in new architecture, given that GHC itself is written in Haskell. In fact, the ubiquity of C compilers has prompted the use of this language as a *lingua franca* between different languages, as we will see in Section 2.4.

2.2 Sharing virtual machines

Another approach for the interaction between languages involves the use of a common execution environment, such as a virtual machine. The code of different languages is compiled to produce compatible representations, according to the data types provided by the execution environment. Many implementations use the Java Virtual Machine [22] for this end. Jython [14] is an implementation of the Python language that produces Java bytecodes. SMLj [2] is a Standard ML compiler that generates Java bytecodes and provides access to Java classes and methods to ML structures and vice versa. The fact that the Java Virtual Machine was not designed to support different programming languages, however, shows in the limitations presented by these projects. SMLj defines extensions to the ML language to allow access to constructs that are specific to Java; Jython poses limitations to the interface between Python and the Java APIs for reflection and dynamic class loading. Besides, the instruction set of the virtual machine focuses on operations that match Java's semantics, which makes, for example, the implementation of arrays with different semantics less efficient.

The .NET Framework [3] is a runtime environment based on virtual machine that is being presented by Microsoft as their programming platform of choice in Windows system. Although the C# language [17] has been introduced specifically for it, this environment has as one of its goals multi-language support – evidenced by the very name of its Common Language Runtime (CLR) – contrasting with the limitations imposed by the Java environment to those who try to use it with other languages. However, adaptations to languages remain necessary in the .NET environment: the .NET version of Visual Basic includes changes to the language to make its semantics match those from C#; a new dialect of C++, C++/CLI, was introduced adapting its memory management model to that of the CLR [9]; similarly, a new dialect of ML called F# was developed to, among other reasons, provide better integration with .NET components written in other languages [38].

Another implementation of a virtual machine for multiple languages is being pursued by the Parrot project [34]. The scope of this project is narrower, aiming to serve as a common back-end for dynamic languages such as Perl and Python. The focus of the project, however, is currently on the implementation of Perl 6.

A kind of communication that can also be considered the use of a common runtime environment is the communication between executables and native libraries through call conventions: rules for passing parameters in the runtime stack, use of registers and name mangling. This can be considered the lowest-level method method for interaction between

code in different languages. Calling conventions, however, are a limited form of communication, as they assume data types with identical memory representation in both languages. Such compatibility is hardly the case, unless one of the languages explicitly considers this kind of interaction in its definition: the Ada standard, for example, requires its implementations to be compatible with the calling conventions of C, COBOL and Fortran [16]. Likewise, C++ allows to specify functions with C-compatible linkage, through the `extern "C"` directive.

2.3 Language-independent object models

Adopting a language-independent type model is another way to handle the issues of data interoperability between languages. This way, in the definition of the data for an application, their interfaces are described in a neutral way, typically using some language designed specifically for this end (an Interface Description Language, IDL), while the implementations are made using the specific languages. The CORBA (Common Object Request Broker Architecture) architecture [28] is one of the main examples of this model. The central motivation for the development of CORBA was to allow the development of distributed applications in heterogeneous environments; language heterogeneity was one of the aspects taken into consideration.

The challenges existing when designing a "language independent" model for data or objects, however, are not unlike those in the design of an interface between any two languages, since this model too describes a type system. When implementing bindings for any of those object models, it is necessary to define a correspondence between the types defined by the model and those offered by the target language, and provide an API for interaction with the runtime environment – in the case of CORBA, with the ORB (Object Request Broker).

If on one hand the task may be easier since the model has been designed with language interaction in mind (unlike, for example, the C type system), on the other one would usually expect a higher level of transparency in the representation of data. For example, while in an application integrating C++ and Python the distinction between C++ objects and Python objects is clear and the Python API defines the limits between these two universes, in an application developed using CORBA one would expect, in both languages, the manipulation of objects to be the same whether they were implemented in C++ or in Python. For that, the common solution is to use *stubs*, objects that give a uniform native appearance to data, regardless of the language in which they were implemented, and in the case of distributed models such as CORBA, of the location of the objects in the network. The correspondence between the life cycles of the stubs and that of the objects they represent is another factor that should be taken into account. In the Java bindings, for instance, this is done with the help of the language's own garbage collector. In languages such as C++ the control of references is explicit.

Other higher-level approaches have been proposed for the integration of applications developed in multiple languages. Coordination languages such as Linda [12] and Opus [4]

define mechanisms for message passing and a restricted set of constructs to indicate the flow of those between agents implemented in other languages.

2.4 C as an intermediate language

The wish for a universal intermediate language is an old one in the world of computing. Several proposals have surfaced through the years, from the UNCOL project [6] to the languages with extensible syntax of the 70s [26] to the most recent virtual machine environments such as .NET. In practice, the needs that these projects aimed to fulfill are being handled through the years in a more pragmatic, if less than ideal, way by using C. Two reasons make C a common choice as an intermediate language. First, its "medium-level" nature, by providing at the same time hardware independence and direct manipulation of memory. Second, the large availability of C compilers, leveraged by the proliferation of Unix systems in the most varied architectures. So, as time went by, to offer an interface for interoperability with other languages gradually became synonymous with offering an interface for communication with C code. This is especially true for dynamic languages that offer features for application extensibility. Not surprisingly, these languages are typically implemented in C.

The availability of C APIs provided by different languages also causes C to be widely used as a "bridge". The integration between Python and Fortran takes place through a Python module written in C that accesses a Fortran library, which on its turn exposes functions using a call convention compatible with C [33]. LunaticPython [27] offers bridges from Lua to Python and from Python to Lua, implemented through a pair of extension modules for each source language written in C.

However, generic intermediate languages continue to be proposed as alternatives to C. C-- [18] is a project that attempts to overcome the limitations of C as an intermediate language making the memory representation of data types more explicit and adding support to constructs that are not easily represented in C, such as tail recursion. Recent versions of the GCC compiler suite have standardized an intermediate language for communication between its various back-ends and front-ends [8].

2.5 Interfaces to C

The C language has, nowadays, a special role in the world of programming languages. Besides being widely used in the implementation of compilers, interpreters and virtual machines (the main implementations of Perl, Python, Ruby and Lua are just some examples), it is also used in compilers as an output format in the generation of portable code (two notable examples are the GHC and SmartEiffel [5] compilers, which generate C from Haskell and Eiffel, respectively). This prevalence of C makes the C API a convenient format for a foreign language interface.

In the vast majority of cases, the internal representation of code produced by compilers

for other languages is not compatible with C, be it because of differences in call or name conventions, or because they produce code for execution in virtual machines. This way, to allow a C program to access this code, the language has to expose a library of C functions that will perform the necessary translations. In virtual machine environments, this library is normally generic, offering facilities for communication with the virtual machine itself. For static languages, it is usually necessary to create a specific library to perform the conversion of calls, as it happens in interfaces that expose C++ libraries to C. An example of this is QtC [20], a library of C bindings to the Qt graphic toolkit, which is implemented in C++.

For non-imperative languages, there is still the problem of C code potentially generating side effects. Some feature for isolating calls has to be offered. In GHC, the construction for C calls, `_ccall_`, is defined in the IO monad; in the addendum for the Haskell 98 standard, the `ccall` directive was integrated, but the use of the monad is optional, requiring the programmer to ensure that the functions that use it are not pure[1].

Another possible source of incompatibility between languages that has to be handled when they interact is the difference between concurrency models. C, in particular, does not define any concurrency constructs; they are implemented through libraries. At the same time that it brings great flexibility to the language, this also imposes portability problems for languages that depend on the availability of concurrency mechanisms in C that are compatible with the models they use.

For example, APIs between C and Java must take into account the preemptive multi-threading model adopted by Java. The JNI (Java Native Interface) [21] defines functions to control mutual exclusion between data shared between the two languages. The programmer must take care to strike a balance between time spent blocking the virtual machine accessing shared data and time spent copying data between the environments to reduce sharing. Another situation in which the concurrency model of the language demands special care when integrating with C happens in the use of co-routines in Lua. The combination of two features of Lua, cooperative multitasking with multiple execution stacks and the ability to alternate between calls to C and Lua functions in a single stack, brings a limitation: a co-routine cannot execute a *yield* operation in case there is a C function in its stack, as there is no portable way to alternate between multiple stacks in C [7].

One of the most frequent motivations in the integration with C code is the use of external libraries. Exposing a C library through the FLI for access by another language may incur in the registration of hundreds of functions. It is also usual to define data types that give to structures defined by the library a more native appearance, such as, for example, converting C functions that register callbacks into Ruby methods that accept code blocks as a parameter. These initializations and adaptations are usually defined as a bindings library, that serves as a bridge between the language and the C library encapsulating the interaction with the FLI.

The patterns that arise when producing bindings are so common that they motivated

[1] A number of additional convention calls are defined (`stdcall`, `cplusplus`, `jvm`, `dotnet`), but `ccall` is the only one declared mandatory by the document.

the development of programs that attempt to automate the process. These bindings generators tend to work using some representation prepared for their use, since analyzing raw C headers may show itself to be insufficient: for example, often the program wouldn't be able to interpret the intention of a construct such as `int**`. SWIG [1] is a popular multi-language tool for generation of bindings for C and C++ libraries which defines its own format for description of interfaces. FLIs may as well use stubs generators to save the programmer from having to write repetitive or non-portable C code. Java features a generator for C headers containing prototypes for native methods to be implemented. Pyrex [10] is a generator for C modules for Python from a syntax based on the Python language itself. Another example is toLua++ [24], a tool for integrating C and C++ code to Lua, which generates stubs from C headers prepared for use by the program, which may contain special annotation to help in the conversion process.

2.6 Scripting languages

A model for interaction between languages that has shown to be especially relevant nowadays is that between statically typed compiled languages, such as C and C++, and dynamically typed interpreted languages, such as Perl and Python. In [32], Ousterhout categorizes these two groups as *systems programming languages* and *scripting languages*.

These two categories of languages have fundamentally different goals. Systems programming languages emerged as an alternative to assembly in the development of applications, having as main features static typing, which eases the understanding of data structures in large systems, and the implementation as compilers, due to concerns with performance. In contrast, scripting languages are dynamically typed and are implemented as interpreters or virtual machines. Dynamic typing and the extensive use of higher-level constructs as basic types, such as lists and hashes, brings greater flexibility in the interaction between components; in static languages, the type system imposes restrictions to those interactions, often requiring the programmer to write adaptation interfaces, which makes the reuse of components harder.

Ousterhout points out that, in a model integrating these two kinds of languages, the tendency is that systems programming languages will no longer be used to write whole applications, but will instead be used in the implementation of components, which are then connected through code written with scripting languages. The convenience offered by high-level interpreted languages allows rapid prototyping and encourages the reuse of components.

The integration of programs developed in systems programming languages coordinated through scripting languages has been common practice for a long time now. Shell scripting in Unix systems is probably the most notable example, where constructs such as pipes (which connect the output of a process to the input of another one) allow one to perform tasks combining a series of programs implemented in different languages, or even other scripts. With the introduction of Tcl [31], this kind of coordination of components through scripting languages started to take place *within* applications. In this model, the scripting

language is implemented as a library and is embedded in an application written in a lower-level language, such as C. Data structures of the application are exposed to the scripting environment as objects; conversely, the application can launch functions in the scripting language and access its data. Programmable applications have existed long before that, typically using little languages created specifically for each application, but the concept introduced by Tcl of implementing scripting languages as C libraries has propelled strongly the development of extensible applications.

The development model based in two languages does not limit itself to applications that provide customization through scripts written by the end-user. In many scenarios, there is a clear distinction between a lower-level layer where performance is a critical factor and a higher-level layer comprised by coordination operations between elements of the lower layer. Typical examples are graphic applications where the interface is described by scripting languages controlling components implemented in C and games where the logic is described in scripts and the rendering engine is implemented in lower-level languages. This greater prominence of scripting languages, where they stop being just an application extension mechanism and start having a more central role in the coordination of the execution of the program, has also promoted an inverted model of interaction between languages, where the application itself is written using the scripting language and libraries written in lower-level languages are loaded as extension modules.

Chapter 3

Scripting language APIs

Interfaces provided by scripting languages are usually understood as "extension APIs": they extend the virtual machine with features not originally offered by it, or alternatively, they extend an external application with the features offered by the runtime environment of the language, embedding it to the application. The first scenario is the one used in the programming model where the high-level coordination is made by an interpreted language and modules written in languages such as C and C++ are used to access external libraries or to implement performance-critical parts. The second scenario, in general, will also encompass the first one, when exposing to the embedded virtual machine extensions that will allow it to talk to the host application.

Both scenarios involve the same general problems: data transfer between the two languages, including how to allow the scripting language to manipulate structures declared in C and vice versa; handling the difference between the memory management models, more specifically the interaction between garbage collection in the virtual machine and explicit deallocation in C; calling functions declared by the scripting language from C; and the registration of C functions so that they can be invoked by scripts. The following sections discuss the main issues involved in the communication between C code and scripting languages, and present the approaches employed by the Python, Ruby, Java, Lua and Perl APIs when handling these issues. Each section concludes with a comparison where the different designs presented in the exposition of each language are reviewed side by side and thus put into perspective.

3.1 Data transfer

The main complexity in the interaction between programming languages is not the differences in syntax or semantics from their control flow structures, but in their data representations. In the communication between code written in two different languages, data flow in various forms: as parameters, object attributes, elements in data structures, etc.

Often, the format how these data are represented differ. In those cases, there are three

alternatives to perform data transfer between the languages. The simplest is to expose the data to the target language as an opaque entity. The target language receives only some kind of handle that allows it to identify the datum uniquely in operations requested later. This approach is useful, for example, if a language is just storing data for the other one, in order to make use of higher-level data structures offered by the language.

Another approach involves perform some conversion to the data from the type system of one language to that of the other. The duplication that takes place in this conversion limits the applicability of this method, restricting its use typically to numeric types and, in minor scale, strings. Finally, the source language may explicitly offer facilities in the target language to manipulate these data, that is, one language would offer an API for the other. The difference between this approach and the first one is that, while in the former the contents of the data remain opaque, here the API defines some means to manipulate their contents.

Because of its focus on the manipulation of pointers and structures, C provides a small set of basic types. Besides, C is very liberal with regard to the internal representation of its structured types, with each different platform having to define its own application binary interface (ABI). So, even in cases where it is possible to link C code directly using compatible basic types and appropriate calling conventions (such as in Free Pascal or several Fortran compilers), bindings libraries are still usually needed to make the manipulation of complex types more convenient.

Even in the fundamental numeric types, there are several precautions that must be taken. Some languages, like Smalltalk and Ruby, perform automatic conversion of integers to "big integers" (*bignums*). In Ruby, in particular, primitive integers have 1 bit less of precision than the machine's word size. There may also be the need to handle conversion of endianness and format of floating point numbers.

For types such as strings, the size of values brings also concerns with performance. In many cases the internal representation used for strings is the same as used in C, so an option is to simply pass to the C code a pointer to the address where the string is stored, which avoids copying of data, under risk of allowing the C code to modify the contents of the string. Exposing to C code pointers to memory areas within the runtime environment of the other language may also bring concurrency problems, in case the environment uses multiple threads.

When exposing to C data of structured types, the conversion to a native C type, in many cases, is not an option. Besides the issue of quantity of data to be converted, structured types in C are defined statically, therefore not serving to represent conveniently data of dynamic structures, such as objects that may gain or lose attributes or even change class during runtime. Even in languages with static types, like Java, the copy of objects is not usually an interesting option due to the volume of data. Copying of structured objects tends to be restricted to specific operations such as manipulation of arrays of primitive types.

The alternative to allowing C code to operate over structured data, thus, is to provide to it an API that exposes as functions the operations defined over those types. This also avoids the need to control the consistency between two copies of a given structure.

Consistency problems, however, may occur if the API allows the C code to store pointers to objects from the language – this makes it necessary for the programmer to manage explicitly the synchronicity between pointers and the life cycles of objects that may be subject to garbage collection. Section 3.2 discusses this issue in greater detail.

3.1.1 Python

All values in the Python virtual machine are represented as objects, mapped to the C API as the `PyObject` structure [44]. More specific types such as `PyStringObject`, `PyBooleanObject` and `PyListObject` are `PyObjects` by structural equivalence, that is, they can be converted through a C cast. Reflecting the dynamic typing model of Python, the API functions use `PyObject*` as a type every time they refer to Python objects, even when they are designed to act on Python values of more specific types, such as for example the `PyString_Size`, that returns the length of a string. Each specific type has a check function in the API, such as `PyNumber_Check` and `PyDict_Check`.

Python is a strongly typed language: each object is tied to a type. Types are represented by `PyTypeObject` structures, which are also structurally equivalent to `PyObject`. Each Python type has a predefined `PyTypeObject` in the API, such as `PyString_Type`, `PyBoolean_Type` and `PyList_Type`. `PyObject_TypeCheck` compares the type of a `PyObject` to a `PyTypeObject` passed as an argument.

For the conversion of data from C to Python, the language offers a series of functions that receive values of primitive C types as an argument, such as `PyString_FromString-AndSize` and `PyFloat_FromDouble`. Each of those functions returns to the C code a pointer to a new `PyObject`. Strings passed are copied by Python. The following example illustrates the creation of a Python object through the conversion of a C value:

```
PyObject* s = PyString_FromString("hello");
```

The two examples below are equivalent, and illustrate type checking through the API, first through a convenience function, and then explicitly, comparing the type of a string Python with `PyString_Type`:

```
if (PyString_Check(s)) printf("Yes.\n");
if (PyObject_TypeCheck(s, PyString_Type)) printf("Yes.\n");
```

For returning data from Python to C, a complementary set of functions is offered, mapping the basic types of Python back to C types. Some examples of those functions that take a `PyObject` pointer as an argument and return the correspondent C datum are `PyLong_AsUnsignedLong` and `PyString_AsStringAndSize`. Differently from the input functions, in these output functions no string copying takes place: the strings returned are pointers to memory stored internally by Python. The documentation recommends not to modify the content of the string except if the memory area was returned by a call to `PyString_FromStringAndSize(NULL,` *size*`)` [46]. This way, it is possible to allocate a string for storage in Python and fill its contents later through C code, as in the following example:

```
/* allocating an uninitialized string in Python */
PyObject* obj = PyString_FromStringAndSize(NULL, 51);
/* obtaining the pointer to the string memory area */
char* s = PyString_AsString(obj);
/* Now, we can fill the string in C. An example: */
for (int i = 0; i < 5; i++, s+=10)
   snprintf(s, 11, "[%-8d]", random());
```

For some of its basic types that do not have direct correspondence in ANSI C 89, Python defines equivalent C types: `Py_UNICODE` and `Py_complex`. These types were added in order to expose the internal representation of data used by Python in numeric manipulation and Unicode text modules implemented in C, avoiding frequent conversions to and from `PyObject`.

Python also offers some versions of its C conversion functions as macros without type checking, assuming that the given `PyObject` will be compatible, offering better performance in expense of safety. These functions can be identified by their uppercase names. Among the conversion macros provided are `PyString_AS_STRING`, `PyInt_AS_LONG` and `PyUnicode_AS_UNICODE`.

Besides functions for type conversions between Python and C, the Python API also offers some conversion functions between Python types. These functions receive a `PyObject` as an argument and return a new `PyObject` with the result of the conversion, and are equivalent to Python functions that perform these conversions (actually calls to `PyTypeObject` types that answer to the `__call__` method). For example, the `PyObject_Str` function is equivalent to the Python function `str`.

In Python, objects are stored in modules, which are namespaces declared globally, or as attributes of objects. Variables are stored in an *environment*, represented as a dictionary. Functions such as `PyRun_File` receive, among their parameters, a dictionary of global variables and another of local variables. The set of global variables and functions is represented as the dictionary of the `__main__` module. Built-in objects are accessible through the `__builtin__` module. For example, to obtain the `str` object, we will initially obtain a reference to the `__builtin__` module using the `PyImport_AddModule` function and then the module's dictionary with the `PyModule_GetDict` function.

```
PyObject* builtins_module = PyImport_AddModule("__builtin__");
PyObject* builtins = PyModule_GetDict(builtins_module);
PyObject* str = PyDict_GetItemString(builtins, "str");
```

In Python, `str` is a callable object, which acts as the string conversion function. So, once we obtained a reference to the PyObject equivalent to `str`, the following call is the same as calling `PyObject_Str` on a given Python object `obj`:

```
/* This is a vararg function that receives as additional arguments a
   NULL-terminated list with PyObjects to be passed to the Python function
   given in the first argument */
PyObject* result = PyObject_CallFunctionObjArgs(str, obj, NULL);
```

The storage of C data in the Python object space can be done in two ways. One way is to create an object of the `CObject` type encapsulating a C pointer, building this way a value that will be opaque to Python. The allocation functions for objects of this type allow to associate to the datum a C function to be called when the `CObject` is deallocated. According to the Python documentation, `CObjects` have as their main purpose passing C data between extension modules [46].

The other way is to declare new Python types through C structures. In C, a Python type is described in two parts: a struct type, from which instances of the type will be produced, and an instance of the `PyTypeObject` struct, which will describe the type to Python. The following example illustrates the creating of a new Python type in C. Initially, we have `point`, which will be the C type of object instances:

```
typedef struct {
   PyObject_HEAD
   int x, y;
} point;
```

The `PyObject_HEAD` macro ensures structural equivalence with `PyObject`. When functions return the object to C code as a `PyObject*`, this will be able to be converted back to `point` through a cast, giving then access to the `x` and `y` attributes. We will also define a function that operates on objects of this type:

```
PyObject* point_distance(point* p) {
    return PyFloat_FromDouble( sqrt(p->x*p->x + p->y*p->y) );
}
```

This function was defined with a `PyObject`* return type so that it can be registered in the Python virtual machine. To associate the function to the Python type, we will initially store it in an array of `PyMethodDef` structures, which will list the type's methods:

```
static PyMethodDef point_methods[] = {
   { "distance", (PyCFunction) point_distance, METH_NOARGS },
   { NULL }
};
```

To make the attributes of the type are visible from Python, we will have to implement an access routine, that receives the object and the name of the accessed attribute. Its implementation is given below:

```
PyObject* point_getattr(PyObject* self, char* name) {
   if (strcmp(name, "x") == 0)
      return PyInt_FromLong(((point*)self)->x);
   else if (strcmp(name, "y") == 0)
      return PyInt_FromLong(((point*)self)->y);
   else
      return Py_FindMethod(point_methods, self, name);
}
```

Once it is registered in the type description, this function will be responsible for returning the type's attributes and methods. This way, we can expose to the Python environment attributes stored in the C struct. The `Py_FindMethod` function locates a function in the array given as its parameter and returns it as a method[1].

Finally, we will define `point_type`, which will be the `PyTypeObject` that will describe the Python type relative to `point`[2]:

```
static PyTypeObject point_type = {
   PyObject_HEAD_INIT(NULL)
   .tp_name = "point",             /* The name of the class */
   .tp_basicsize = sizeof(point),  /* The size of the memory area to be allocated */
   .tp_getattr = point_getattr,    /* The attribute access function */
   .tp_flags = Py_TPFLAGS_DEFAULT  /* This class does not require special treatment */
};
```

Again, a macro was used at the top of the definition to ensure structural equivalence. `PyTypeObject` has many other fields, but we will keep them `NULL` so that they will be filled with default values during the construction of the type at runtime. The `PyTypeObject` type contains a number of fields that allow to describe the behavior of the declared type. In the `tp_getattr` field of `point_type`, we specified that the C function to be used to handle access to attributes will be `point_getattr`. We specified `Py_TPFLAGS_DEFAULT` in the flags field to indicate that this is a class with a default behavior, without the need for special treatment such as cycle checking during garbage collection.

While the in-memory representation of `Python` objects of the user-defined type are instances of `point`, to create a new object it is not enough to allocate an instance of `point` and use it as a `PyObject` through casting. It is necessary to initialize the object so that it is registered in the garbage collection mechanism and it has the fields of its `PyObject` header properly initialized. The allocation in C of new objects of a user-defined type must be done through the `PyObject_New` macro, which receives as arguments the type of the struct to be allocated and the `PyTypeObject` that corresponds to the type. The documentation recommends assigning the default construction function, `PyType_GenericNew`, during runtime for portability reasons [44]. Finally, any undefined fields of the struct are filled by the `PyType_Ready` function.

```
point_type.tp_new = PyType_GenericNew;
if (PyType_Ready(&point_type) < 0) return;
```

From there on, instances can be created with `PyObject_New`, as in the example below:

```
/* Creates an instance */
point* a_point = PyObject_New(point, &point_type);
a_point->x = 100; a_point->y = 200;
/* Stores the instance in the Python global 'P',
   assuming the globals dictionary was stored in 'globals' */
PyDict_SetItemString(globals, "P", (PyObject*) a_point);
```

[1] Registration of Python functions will be discussed in detail in Section 3.4.1.

[2] For brevity, we will present the example using the C99 syntax for structs, saving us from listing the elements that will be initialized with `NULL`, as the `PyTypeObject` struct has 54 fields in total.

Once declared in C, this value can be used by Python code:

```
print 'P.x = ' + str(P.x)
print 'P.y = ' + str(P.y)
print 'd   = ' + str(P.distance())
```

The Python API has a large number of functions for manipulation of predefined types in the language. *Tuples* deserve a special mention with regard to data transfer between Python and C, as they are used in several contexts: when passing arguments to Python functions from C, when receiving input arguments in C functions and also when passing and receiving multiple return values, as we will see in Sections 3.3.1 and 3.4.1.

As tuples are frequently used as a "bridge" between Python and C, the API offers a convenience function, `PyArg_ParseTuple`, that saves the programmer from having to perform access and type checking of the tuple elements one by one. This is a vararg C function that receives as arguments the tuple, a string indicating the types of expected arguments and the addresses where the values, converted to C types, should be stored. The function defines a syntax for expected type identifiers in the given string and the correspondent C types. For example: `"s#"` indicates that the tuple should contain a Python object of the `string` or `Unicode` type and that two parameters should be passed to the C function, with types `const char**` and `int*`, that will return the string pointer and its size, respectively. In a more elaborate example, `"iiO!|(dd)"` indicates that the function expects two integer addresses (`"ii"`), followed by the address of a `PyObject` pointer (`"O"`) and a PyTypeObject to be used when type checking the received object (`"!"`) and optionally (`"|"`), two addresses of `double` values given to Python through another tuple (`"(dd)"`).

In a similar fashion, the Python API has the `Py_BuildValue`, which allows the construction of structured objects, such as tuples, lists and dictionaries, in a single call. This function is frequently used both in the construction of the argument tuple when calling functions and in the construction of return values. The syntax of the parameter string resembles that of `PyArg_ParseTuple`, but it features a different set of type indicators, and allows to describe lists and dictionaries. For example, the following call creates a list containing an integer, a floating point number and a dictionary containing an element with a string key and an integer value:

```
PyObject* list = Py_BuildValue("[id{si}]", 123, 12.30, "foo", 1234);
```

This is equivalent to the following Python construct:

```
list = [123, 12.30, {"foo": 1234}]
```

3.1.2 Ruby

For the communication of data between Ruby and C, the Ruby API defines a C data type called `VALUE`, which represents a Ruby object. `VALUE` may represent both a reference to

an object (that is, a pointer to the Ruby heap) as well as an immediate value. In particular, the constants `Qtrue`, `Qfalse` and `Qnil` are defined as immediate values, allowing them to be compared in C using the == operator.

For type checking, Ruby provides the `Check_Type` and `TYPE` macros. `Check_Type` allows one to compare the type of values to constants that describe basic types of Ruby such as `T_OBJECT` and `T_STRING`. `TYPE` returns the constant relative to the type of a given value. To check the class of an object, one should use `rb_class_of`.

When transferring numeric values, the conversion between C and Ruby is made through macros such as `INT2NUM` and functions such as `rb_float_new`, which receive or return `VALUE`s.

For passing strings to Ruby from C, the API provides the `rb_str_new` function, which receives a pointer and a numeric size argument, allowing the use of strings containing null characters, and the `rb_str_new2` function, which assumes a standard C string, with the null character as a terminator. These functions make a copy of the C string to the data space of Ruby. `VALUE`s that point to Ruby strings allow C code to access and modify their contents through the `RSTRING(a_string)->ptr` cast. However, the API recommends the use of the `StringValue` macro, which returns the `VALUE` itself in case it is a string, or a new `VALUE` of the `String` class produced through the `to_s` conversion method applied to the given object (or raises a `TypeError` exception in case the conversion was not possible).

```
void show_value(VALUE obj) {
   const char* s;
   if (TYPE(obj) == T_STRING) {
      /* This would make an illegal access if TYPE(obj) != T_STRING */
      s = RSTRING(obj)->ptr;
   } else {
      /* Works for any type that accepts obj.to_s,
         otherwise, raises an exception */
      s = StringValue(obj);
   }
   printf("Value: %s\n", s);
}
```

Under the justification of increasing performance on access, some other Ruby types such as `Array`, `Hash` and `File` allow low-level access to the members of structures used in the implementation of their objects. For example, with `RARRAY(an_array)->len` one can read the size of an array directly. The recommendation of the API is to use this kind of access for reads only, since the modification of these values can easily make the internal state of objects inconsistent.

For storing C data in the Ruby object space, the API provides a macro, `Data_Wrap_Struct`, which receives a C pointer and creates a Ruby object which encapsulates this pointer. The pointer can be accessed from C code using `Data_Get_Struct`, bet not from Ruby. A C function to be executed when the object is collected is also passed to `Data_Wrap_Struct`. For example, we will create a `Point` class, similar to the Python type defined in the previous section. We will initially define a C type:

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

Allocation and deallocation functions for the `Point` class (`point_alloc` and `point_free`) follow:

```
void point_free(void* p) {
   free(p);
}

VALUE point_alloc(VALUE point_class) {
   point* p = malloc(sizeof(point));
   /* The 2nd argument is the mark function for garbage collection
      (NULL here as the type doesn't store VALUES), see Sec. 3.2.2 */
   return Data_Wrap_Struct(point_class, NULL, point_free, p);
}
```

Notice that `Data_Wrap_Struct` makes use of a `VALUE` that represents the `Point` class in Ruby. Classes are created in C with the `rb_define_class` function. This function gets a C string with the name of the new class and a VALUE to be used as a superclass (such as for example the `rb_cObject` constant, which represents the `Object` Ruby class) and returns a `VALUE` representing the new class. For classes such as `Point`, whose instances will contain C data, it is possible to register a C function that will be responsible for allocating memory of instances using the `rb_define_alloc_func` function. So, the creation of the class and the registration of the allocation function are done as follows:

```
VALUE point_class = rb_define_class("Point", rb_cObject);
rb_define_alloc_func(point_class, point_alloc);
```

Like in Ruby code, the declaration of object attributes is done in the `initialize` method, which can be implemented in C:

```
VALUE point_initialize(VALUE self, VALUE x, VALUE y) {
   point* p;
   Data_Get_Struct(self, point, p);
   p->x = NUM2INT(x);
   p->y = NUM2INT(y);
   return self;
}
```

The method is registered in the class at runtime with the `rb_define_method` function (the registration of C functions in Ruby will be discussed in detail in Section 3.4.2).

```
rb_define_method(point_class, "initialize", point_initialize, 2);
```

To ensure that the copy of objects through Ruby's `dup` and `clone` methods will handle correctly the data stored through C, it is necessary to register the `initialize_copy` method. A possible implementation in C is given below:

```
VALUE point_initialize_copy(VALUE copy, VALUE orig) {
   point* p_copy;
   point* p_orig;
   /* Ruby may call this function with the same object in both args;
      in this case, ignore the call and return the object */
   if (copy == orig) return copy;
   /* Obtain the pointers stored in the objects */
   Data_Get_Struct(orig, point, p_orig);
   Data_Get_Struct(copy, point, p_copy);
   /* Copy of the "C part" of the object */
   p_copy->x = p_orig->x;
   p_copy->y = p_orig->y;
   /* Returns the copy */
   return copy;
}
```

We will complete the example with a C function implementing the `distance` method, like it was done in the previous section for Python:

```
VALUE point_distance(VALUE self) {
   point* p;
   Data_Get_Struct(self, point, p);
   return rb_float_new( sqrt(p->x*p->x + p->y*p->y)) );
}
```

These functions are also registered as methods of `Point`:

```
rb_define_method(point_class, "initialize_copy", point_initialize_copy, 1);
rb_define_method(point_class, "distance", point_distance, 0);
```

The `rb_class_new_instance` function produces new Ruby objects that are instances of the class, receiving a C array of `VALUE`s to be passed during object initialization and the class `VALUE`.

Access of Ruby values is done through the `rb_*_get` family of functions, which return `VALUE`s relative to attributes of objects or classes, global variables and constants. For each of those there is an analogous `rb_*_set` function[3]. The `rb_iv_get` and `rb_ivar_get` functions, for example, obtain object attributes (*instance variables*). The first form uses C strings as names, the latter uses `ID`s, identifiers that replace interned strings in Ruby's symbol table, that can be obtained using the `rb_intern` function. In fact, `ID`s correspond to the *symbol* Ruby type, which in practice are immutable strings. The following example obtains the value of a global variable `g` and sets it to the field `f` of an object, and then sets the value of the global variable to zero:

```
/* Obtains the global variable */
VALUE g = rb_gv_get("g");
```

[3]Constants can be created with the `Qundef` value and have their value defined later *once* with `rb_const_set`.

```
/* Sets the field f of object obj */
VALUE obj = rb_gv_get("obj");
rb_iv_set(obj, "f", g); /* Same as: rb_ivar_set(obj, rb_intern("f"), g); */
/* Zeroes the global variable */
rb_gv_set("g", INT2NUM(0));
```

`IDs` are never collected: we observed that the symbol table is not cleaned up even after `ruby_finalize`. So, a C application that offers a scripting interface creating supposedly isolated environments, surrounding each script execution with `ruby_init` and `ruby_finalize`, may have its memory consumption increased indefinitely as scripts create symbols.

3.1.3 Java

The JNI defines in the `jni.h` header C types equivalent to each of Java's primitive types (`jint` for `int`, `jfloat` for `float`, and so on). The "reference types", such as classes and objects, are exposed to C as opaque references, instances of `jobject`. Strings and arrays are also objects in Java and are thus exposed as instances of `jobject`. However, the JNI defines as a convenience some C types that act as "subtypes" of jobject: `jclass`, `jstring`, `jthrowable`, `jarray`, `jobjectArray`, and an array type for each primitive type (`jbooleanArray`, `jbyteArray`, etc.). The `jvalue` type is a union of primitive and reference types. The `NULL` C value corresponds to Java's `null`.

Different methods are employed for reading primitive types, strings, arrays and other objects. Reading the contents of a `jstring` in C requires the conversion from the internal format used by Java, UTF-16. The API offers a utility function that allocates a string containing the representation of the text in UTF-8 (which is an ASCII-compatible format), `GetStringUTFChars`. This string must be later deallocated with `ReleaseStringUTFChars`. The `GetStringChars` function provides direct access to the string in UTF-16 format; it has an output argument that indicates if the returned string is the JVM's own internal buffer or if it is a copy. At the same time that this saves the C code from duplicating the string in cases when one wants to modify it and the JVM has returned a copy, this parameter exposes to the API low-level issues of the JVM string management. Alternatively, the `GetStringRegion` and `GetStringUTFRegion` functions perform a copy of the string to a pre-allocated buffer provided by the programmer. `GetStringCritical` returns a pointer to the JVM internal buffer, but this involves special care with regard to garbage collection, which will be discussed in Section 3.2.3.

Arrays of primitive elements are handled in a similar way to strings, differently from object arrays[4]. There are functions for performing array copies (`Get`/`Set`<*type*>`ArrayRegion`), functions akin to `GetStringChars` that return pointers to the array that may or may not perform copies (`Get`/`Release`<*type*>`ArrayElements`) and functions that can access the JVM internal buffer, like in `GetStringCritical` (`Get`/`ReleasePrimitiveArrayCritical`). For object arrays, it is not possible to obtain a pointer to an array's internal buffer. Access

[4] Multi-dimensional arrays are considered as "arrays of arrays" and, as such, are also object arrays.

```
public class ExampleJNI {
   private String[] elements = { "Earth", "Air", "Fire", "Water" };

   /* Declaration of the externally implemented method */
   private native void secondElement();

   public static void main(String[] args) {
      /* Creates an instance and invokes the native method */
      new ExampleJNI().secondElement();
   }

   static {
      /* Loads in the JVM the external code that will implement
         the secondElement method */
      System.loadLibrary("ExampleJNI");
   }
}
```

Figure 3.1: Java class containing an externally implemented method

to elements is performed one at a time, through `jobject` references, with `Get/SetObject-ArrayElement`.

The retrieval of values and attributes is done through functions such as `GetObjectField` and `GetStaticField`, which return reference of the `jobject` type. For each primitive type there is an equivalent type, such as `GetIntField` and `GetStaticIntField`. Like in Ruby, the Java API defines a specific C type to avoid the frequent use of C strings in the description of fields. However, while ruby uses `ID`s which are merely interned strings, in Java field identifiers, of the `jfieldID` type, contain type information and are specific for a field of a given class. These values are obtained with the `GetFieldID` call, which receives among its arguments a string called the "JNI field descriptor" with a special syntax. For example, the Java type `int[][]` is described with `"[[I"` and the `java.lang.String` type as `"Ljava/lang/String;"`[5]. It is also possible to obtain a `jfieldID` from a `java.lang.reflect.Field` object using the `FromReflectedField` function.

JNI calls are done in C with `(*J)->`*function*`(J, ...)`: JNI functions are accessed through function pointers stored in a table pointed by a `JNIEnv` structure, which is then propagated in calls. The goal of these two levels of indirection is to decouple the linkage of calls in C code from the library that implements the JNI, allowing to link the code at runtime to different implementations of the JVM [36].

The access to Java attributes in C code is illustrated through the following example. Initially, in Figure 3.1, we implement a Java class that has a private attribute, the elements `array`, and defines a function, `secondElement`, to be implemented in C[6].

[5]This is another place where implementation details leak through the API. Not coincidentally, this syntax is the same used in the internal representation of types in JVM bytecodes.

[6]Details about the declaration and registration of functions implemented in C will be discussed in Section 3.4.3.

```
#include <jni.h>
#include <stdio.h>
#include "ExampleJNI.h"

JNIEXPORT void JNICALL
Java_ExampleJNI_secondElement(JNIEnv* J, jobject this) {
   /* Get the class of 'this': ExampleJNI */
   jclass klass = (*J)->GetObjectClass(J, this);
   /* Get ExampleJNI.elements, a String[] */
   jfieldID elemsID = (*J)->GetFieldID(J, klass,
      "elements", "[Ljava/lang/String;");
   /* Get the contents of ExampleJNI.elements */
   jarray elems = (*J)->GetObjectField(J, this, elemsID);
   /* elems_1 = elements[1] */
   jstring elems_1 = (*J)->GetObjectArrayElement(J, elems, 1);
   /* Get the representation of elems_1 as a C string */
   const char* elems_1_c = (*J)->GetStringUTFChars(J, elems_1, NULL);
   /* Show the string */
   printf("%s\n", elems_1_c);
   /* Free the memory of the string */
   (*J)->ReleaseStringUTFChars(J, elems_1, elems_1_c);
}
```

Figure 3.2: C code implementing a Java method

The implementation of `secondElement` is presented in Figure 3.2, showing the sequence of calls needed to obtain in C the element of the Java array. To access the `elements` attribute, the function has to obtain the field identifier. For that, we need first to obtain a reference of the current class with `GetObjectClass` from the object reference (`this`) passed a parameter to the function. Once we have the class reference (`klass`), we obtain the field identifier with `GetFieldID`. The content of the field is then obtained with `GetObjectField`: a reference to the array. Using it, the element of the array is obtained with `GetObjectArrayElement`. A copy of the element, converted to a UTF-8-encoded C string, is returned with `GetStringUTFChars`. As previously discussed, after its use, the string must be freed with `ReleaseStringUTFChars`.

The manipulation of objects of the `Class` type is also done through specific functions. It is not possible to create Java classes through the C API, but it is possible to load classes at runtime using the `DefineClass` function, which receives a buffer containing the representation of a pre-compiled Java class. References of the `jclass` type can be obtained from the class name using `FindClass`, which uses a syntax for descriptors similar to that used by `GetFieldID`[7].

For assigning C values that can be converted to Java primitive types, the JNI provides functions such as `SetIntField` and `SetFloatArrayRegion`. For other types, there are no

[7] In both field and class descriptors, `"[Ljava/lang/String;"` represents `String[]`. For the `String` type, however, `"Ljava/lang/String;"` is the field descriptor and `"java/lang/String"` the class descriptor.

specific provisions for storing C data in the Java object space. In those cases, the documentation suggests the storage of pointers in numeric types [21], in spite of the portability limitations brought by this approach.

3.1.4 Lua

The Lua API defines a different approach for manipulating data in C: no pointers or handles to Lua objects are ever exposed to C code. Operations are defined in terms of indices of a virtual stack. So, data transfer from C to Lua takes place through functions that receive C types, convert them to Lua values and stack them, such as `lua_pushboolean`, `lua_pushinteger` and `lua_pushlstring`. Several operations of the API operate on the value at the top of the stack, such as, for example, `lua_setglobal`[8]:

```
lua_pushinteger(L, 123);        /* Inserts the number 123 in the stack */
lua_setglobal(L, "foo");        /* Sets the number 123 to global foo */
```

Most lookup functions, however, allow one to specific any stack index (with positive values for indexing from the bottom and negative values for indexing from the top).

Conversion of data from Lua to C is made through functions such as `lua_tonumber` and `lua_tolstring`, which receive a stack index, convert the value at the given index to the specified Lua type if necessary, and return the value converted to the equivalent C type. Numbers have the `lua_Number` C type, which corresponds to `double` by default but is a compile-time parameter for the Lua virtual machine. Strings, in particular, are immutable objects and are interned: any two identical strings share the same internal representation.

So, unlike languages such as Python and Ruby, it is not possible to modify the contents of a Lua string from C through its memory representation as a `char*`. To make the incremental construction of Lua strings from C more efficient, the Lua auxiliary library defines a C type called `luaL_Buffer` and functions such as `luaL_addstring` and `luaL_addvalue`, which allow the construction of a string in stages until it can be finally converted to a Lua string with `luaL_pushresult`. This way, one avoids consecutive string concatenation operations through the Lua API.

Lua defines two specific data types for storing C data, *full userdata* and *light userdata*. *Full userdata* describe memory blocks managed by Lua and used by C code. They exist in Lua as opaque objects, and are created by `lua_newuserdata`, which inserts the new object in the Lua stack and returns to C a pointer to the memory area of the requested size. Objects of the *light userdata* type, created through `lua_pushlightuserdata`, allow storing C pointers in Lua; allocation and management of the memory block are to be handled by C code. The following example illustrates the use of userdata, assuming the same `point` struct defined on page 25. The userdata object is created this way:

```
/* Creates a full userdata, inserts it in the stack and returns
   the pointer to C. The memory is allocated by Lua */
```

[8] API functions that operate on a Lua execution state receive an initial argument (in our examples, called `L`), indicating the state they refer to. This will be discussed later, in Section 3.4.4.

```
point* full_p = (point*) lua_newuserdata(L, sizeof(point));
/* We then use the pointer in C... */
full_p->x = 100; full_p->y = 200;
/* Assigns the object to the global variable Point */
lua_setglobal(L, "Point");
```

Now, accessing it:

```
/* Pushes the global Point */
lua_getglobal(L, "Point");
/* Gets the C pointer from the userdata at the top of the stack (-1) */
point* p = (point*) lua_touserdata(L, -1);
printf("(%d,%d)\n", p->x, p->y);
/* Restores the stack to its original position, removing the item.
   It won't be collected, as it is tied to the global variable */
lua_pop(L, 1);
```

Assuming that the global `Point` is the only reference to this block, to free it all we have to do is overwrite `Point` with, for example, `nil`; the memory of the full userdata will then be eligible for retrieval by the garbage collector, like that of any Lua value with no references.

```
lua_pushnil(L);
lua_setglobal(L, "Point");
```

The stack storage area does not adjust itself dynamically and the API functions do not perform overflow checks. So, the programmer is responsible for controlling the stack size, through the `lua_checkstack` function. In practice, the stack size will only grow in loops pushing elements, since typical sequences of operations tend to push values and pop them later.

Tables are the only type available for construction of data structures in Lua. Lua offers a complete API for manipulation of tables from C. Tables can be created with `lua_newtable` or `lua_createtable`; the second form allows one to pre-allocate memory for table elements. The `lua_gettable` and `lua_settable` implement the semantics of reading and writing fields to a Lua table, including eventual calls to metamethods; for calls without metamethod invocation there are `lua_rawget` e `lua_rawset`, which are equivalent to `rawget` e `rawset` in Lua (besides two convenience variants, `lua_rawgeti` and `lua_rawseti`). There is also the `lua_next` function, equivalent to the Lua function `next`, which is designed for traversing elements of a table. An example of table manipulation is given below:

```
/* tbl["key"] = 12345, in C: */
lua_getglobal(L, "tbl");
lua_pushstring(L, "key");
lua_pushinteger(L, 12345);
/* lua_settable inserts the item at the top of the stack
   to the table given as a parameter,
   using as a key the item right below the top: */
lua_settable(L, -3);
```

Many Lua concepts are represented through tables – the global environment, metatables, registry – are are thus handled in C using the API functions for table manipulation. The global environment table of the currently executing thread can be accessed through a special index of the virtual stack, `LUA_GLOBALSINDEX`. One can also define a function environment table, indexed with `LUA_ENVIRONINDEX`, to isolate data to be shared internally by functions in modules written in C. For example, the global environment can be manipulated as a table this way:

```
lua_pushstring(L, "Point");
lua_gettable(L, LUA_GLOBALSINDEX);
```

This is equivalent to:

```
lua_getglobal(L, "Point");
```

3.1.5 Perl

The procedures for extending and embedding Perl are very different from each other. For extensions, Perl provides an interface description language called XS. Instead of isolating the access to Perl's internal structures through a public API, the proposed approach is to encapsulate the process of generating wrapper code for the communication of functions written in C to the internal structures of Perl using interfaces written in XS. Files of the `.xs` type contain C code along with annotation that simplifies the handling of input and output parameters. These are fed to the `xsubpp` pre-processor, which generate then code using the API provided by the Perl library. This library offers low-level access to the inner workings of the interpreter, allowing one, for example, to manipulate its internal stack pointer. The goal of XS is to hide these details from the extension developer.

To embed the Perl interpreter to an application, the library that implements it offers some functions that allow one to launch an interpreter. In the higher-level API, one can build an array of arguments to be passed to the interpreter in the same way as options are given to the Perl command-line interpreter, even using the `"-e"` flag to execute pieces of code.

The types of Perl variables are mapped to C structs: `SV` for scalars, `AV` for arrays, `HV` for hashes. These C values are better understood as *containers* to Perl values: a scalar variable in Perl has an `SV` associated to itself; however, one can create in C an `SV` that is not associated to any Perl variable name.

Primitive types of Perl are represented as C values through typedefs: `IV`, `UV`, `NV` and `PV` correspond, respectively, to signed and unsigned integers, floating-point values and strings. These values can be copied to `SV`s. Perl references are represented as `RV`, and are also a kind of `SV`. There is also the `GV` type, which represents any type representable through a Perl variable.

Variables from the Perl data space are accessed with `get_sv`, `get_av` and `get_hv`. These functions receive a C string with the variable name (possibly qualified through the `"package::variable"` syntax). The content of scalar values are converted back to C types

with the Sv* macros: SvIV returns an integer, SvPV returns a char* and the string length in the second argument, etc. The following C code prints the contents of the Perl variable $a, assuming it contains an integer value:

```
printf("a = %d\n", SvIV(get_sv("a", FALSE)));
```

The flag given as a second argument for get_sv/av/hv indicates whether the variable should be created if the given name does not correspond to an existing variable. Passing an unexisting name and using TRUE as the second argument is a convenient way to create a new variable accessible from the C space and at the same making its binding in the Perl space.

```
/* Creates a variable of the array type,
   accessible in Perl as the global @arr and in C as the AV* arr */
AV* arr = get_av("arr", TRUE);
```

An SV can be created in C with the newSV* family of functions: newSViv generates a new SV storing a signed integer; newSVpv stores a string, and so on. The newSV function creates a SV with an uninitialized memory area, accessible through the SvPVX function, providing a way to create scalars with arbitrary values generated by C code. Using the same example of the point struct from previous sections, we can store a C object in a Perl value in the following way:

```
/* Allocates an uninitialized SV, the size of a point */
SV* v = newSV(sizeof(point));
/* Gets the pointer to the SV's memory area */
point* p = (point*) SvPVX(v);
/* Manipulates the point in C. When v is returned to Perl,
it will be an opaque variable (its contents won't be accessible) */
p->x = 100; p->y = 200;
```

Values are assigned to SVs using the sv_set* family of functions: sv_setiv, sv_setpv, etc. Functions for string handling have variants such as newSVpvn and sv_setpvf, which receive the string length or perform sprintf-style formatting. For strings, there are also the sv_cat* functions, which act like sv_set* but concatenate the value given to the current content of the string instead of replacing it. The sv_setsv function copies the value of an SV to another. The SV created in the previous example can be assigned to a global variable like this:

```
/* Gets the SV from the global $point */
SV* pnt = get_sv("point", TRUE);
/* Assigns the value of v to $point */
sv_setsv(pnt, v);
```

The type of data stored in SVs is checked with the SvIOK macro for integers, SvNOK for floating-point values, and SvPOK for strings. These functions return success if the scalar is

convertible to the specified type – the variants SvIOKp, SvNOKp, SvPOKp verify if the value stored in the SV is actually of that type.

Arrays and hashes are created with newAV and newHV. Arrays can be populated with a C array of SV pointers through av_make. Operations such as av_fetch, av_pop, hv_fetch and hv_exists operate on elements of these structures. In av_fetch and hv_fetch, the return type is SV**, to differentiate between returning an existing element which points to NULL from returning "element not found". In the following example, we create a Perl array containing the 10 first elements of the Fibonacci series:

```
/* Create a new array */
AV* a = newAV();
/* Store two values, 0 and 1, at its first positions */
av_push(a, newSViv(0));
av_push(a, newSViv(1));
for (int i = 2; i < 10; i++) {
   /* Obviously, it would be more efficient to store these values in
      temporary values in C, but we'll obtain the last two values
      back from the Perl array for illustration purposes: */
   SV** next_to_last_sv = av_fetch(a, i-2, FALSE);
   SV** last_sv = av_fetch(a, i-1, FALSE);
   /* Obtain the integers stored in those SVs */
   int next_to_last = SvIV(*next_to_last_sv);
   int last = SvIV(*last_sv);
   /* Create a new SV and insert it in the end of the array */
   av_push(a, newSViv( next_to_last + last ) );
}
```

Once this AV is created, however, there is no way to associate it to a Perl variable. Its contents must be copied item by item. To make it accessible from Perl, we should have created it with get_av, and not newAV. AVs that are not associated to variables are useful, however, when passing arguments in function calls and as return values.

Some functions for hash manipulation expose the key/value pairs as HE pointers. The HeSVKEY and HeVAL macros extract the key and value from an HE. The following C function prints the elements of a Perl hash:

```
void print_hash(HV* hash) {
   HE* item;
   /* Each HV keeps its own internal iteration control */
   hv_iterinit(hash);
   /* Get the next key/value pair in the iteration */
   while ( (item = hv_iternext(hash)) ) {
      /* Get the string representation of scalars
         representing key and value of the item */
      char* key = SvPV_nolen(HeSVKEY(item));
      char* value = SvPV_nolen(HeVAL(item));
      printf("%s => %s\n", key, value);
   }
}
```

Special care should be taken when using the values `undef`, `true` and `false` in arrays and hashes, even though Perl exposes these constants in its C API (`PL_sv_undef`, `PL_sv_true`, `PL_sv_false`). The constant `PL_sv_undef` is used internally in the implementation of `AV`s and `HV`s, and the update of values in `HV`s happens in-place, which generates problems when updating elements containing these constants. The documentation recommends generating copies of these values when using them in `AV` and `HV` structures [29].

Perl references are created with `newRV_inc` and `newRV_noinc`, which receive an `SV`, `AV` or `HV` pointer as a parameter (these two functions differ in how they handle reference counted, which will be discussed in Section 3.2.5). The value pointed by a reference is obtained with `SvRV`. The return value of this macro must be converted through a cast to the appropriate type (`IV`, `PV`, `AV`, etc.), which can be checked with `SvTYPE`.

Many functions of the API have argument or return types declared as `SV` when in fact they accept `AV`s or `HV`s; this is analogous to the concept of Perl *contexts*, in which the same value can be treated as a list (array or hash) or scalar depending on the expression where it is inserted [25]. In Perl code, the context where a function is executed can be inspected with `wantarray`. In C, the context can be checked with the `GIMME_V` macro, which returns `G_VOID`, `G_SCALAR` or `G_ARRAY`.

3.1.6 Comparison

The basic set of functions for manipulating data in the five languages presented here is similar: all of them provide functions for converting values from the language to basic C types and vice versa. All of them also offer functions for manipulating their fundamental structured types (tables in Lua, arrays in Java, arrays and hashes in Ruby and Perl, lists and dictionaries in Python). Python, in particular, defines an extensive function API for operations on its built-in classes; most of these functions could be performed using the generic API for method invocation, but they are offered directly in C as a convenience.

Lua stands out for having, with its stack model, the simplest and most orthogonal data manipulation API among the ones studied. However, the resulting code often loses in readability when the stack indices are not obvious. It is commonplace to see C code using the Lua API commented line by line, to save the programmer from having to simulate mentally the stack operations while reading the program.

In Java, static typing reduces greatly the need of explicit data conversion in C code. On the other hand, treatment of multi-threading complicates the access of types such as strings and arrays.

A negative point in the Ruby API is the exposure of implementation details of the struct fields that describe its fundamental `VALUE` type. This restricts the flexibility of the language implementation and fosters unsafe programming practices. Perl also exposes a good deal of its internal structures; not as directly as Ruby, but through macros. These macros, however, assume the adherence to usage protocols so strict that in practice they also largely limit the possibilities of changes in the implementation (an example of this is the sequence for function calls, which will be presented in Section 3.3.5).

The creation of data containing C structures stored by the scripting language is an easy

task in Perl, Ruby and Lua: Perl allows one to create `SV`s containing arbitrary memory blocks for use in C; Ruby offers the `Data_Wrap_Struct` macro which generates a Ruby object which encapsulates a C structure; Lua defines a basic type in the language especially for this end. In Python, on its turn, the process is not as straightforward. Creating a Python class from C involves declaring parts of it statically and other parts dynamically, being usually necessary to define three different C structures. In Java, it is not possible to create new types from C, one can only load classes.

Another common task when interacting with C is the need to store pointers in the data space of the scripting language. Python, Lua and Perl offer features to do this directly: creating a `PyCObject` in Python; a light userdata in Lua; or storing a pointer in the data area of an `SV` in Perl. In Java and Ruby, the alternative is to convert pointers and store them as numbers. In fact, this happens internally in the implementation of Ruby, and the portability limitations of this approach are made evident by the fact that the compilation of Ruby fails if `sizeof(void*) != sizeof(long)`.

Finally, an aspect that deserves being mentioned is the concern on not polluting the C namespace. Python, Java and Lua define all its functions and C types with prefixes that aim to avoid conflicts with names defined by the application. Perl and Ruby define names in a disorganized fashion, which occasionally causes problems[9]. Perl has options to disable a series of macros and force a common prefix in its functions, but this feature is incomplete and using it hampers the functionality of its headers[10].

3.2 Garbage collection

From the moment when C code gains access to references to data stored in the storage space of another language, be them pointers or identifiers, the programmer must take into consideration the differences between the memory management models involved, since code executed in the other language may free the data. For example, the C program may deallocate an object referenced by data in the scripting language, or the scripting language may remove an element from a structure causing it to be collected. In principle, this task of maintaining consistency between these two environments is no different from manual memory management normally taken care by C programmers. However, the interaction with some languages adds an important factor of complexity: the garbage collection mechanisms perform deallocation of data from memory in an implicit manner. The fundamental principle of garbage collection dictates that an object is not collected in case there is some element (variable, data structure) pointing to it. However, the same is not valid for the C environment: the presence of a pointer pointing to an object does not guarantee that it will not be collected, given that the garbage collector does not manage pointers from C

[9] For example, conflicts of this kind happened in the Ruby bindings of the Subversion revision control system in Win32 platforms (`http://svn.haxx.se/dev/archive-2005-04/1789.shtml`).

[10] In the case study presented in Chapter 4, when using the Perl API we tried to restrict ourselves to the `Perl_`-prefixed versions of the API functions, but many necessary macros are only available in versions with no prefix.

code.

It is necessary, then, to indicate from C code that the data remain accessible from it and must not be collected. In a complementary way, when transferring the control of C objects to the domain of the other language – for example, when storing them in a data structure of the other language – it is necessary to indicate to the language how to deallocate the memory of the structure when the garbage collector detects that it is no longer in use. The way how the API will provide this functionality depends not only of the design of the C API, but also of the garbage collection mode employed by the implementation of the language.

3.2.1 Python

The Python virtual machine features a garbage collector based on reference counting. As the Python API returns to C code pointers to `PyObjects`, the programmer must have the care of ensuring that they will remain valid. For such, it is necessary to increment and decrement the pointed object's reference count according to how one wants to control the validity of the pointers in C code.

In general, once the C code needs to retain a `PyObject*`, it should use the `Py_INCREF` macro to increment its reference count and so prevent it from being collected. Once the value is no longer needed, the reference count should be accordingly decremented with `Py_DECREF`. Python works with the concept of *"reference ownership"* to define when the programmer needs to increment or decrement the counter of references returned by API functions. Most API functions that return pointers to `PyObjects` transfer references to the caller; the reference becomes the caller's responsibility – it can either pass it on or it will have to decrement it with `Py_DECREF` when it is not needed anymore (C code can store owned references in its data structures; they will remain valid even after the return of the function, until they are explicitly decremented). Other functions *lend* references; the code that borrows the reference does not need to decrement it after using it, but the validity of the object is attached to the validity of the reference in the object that returned it to C. For example, `PyList_GetItem` lends a reference to an element of the list. The pointer returned will remain valid while the the item remains stored in the list. It is possible to obtain "ownership" of a borrowed reference by incrementing the object count with `Py_INCREF`: the validity of the pointer becomes independent from the container object that returned it, but the C code becomes responsible for decrementing the reference later with `Py_DECREF`.

For object references passed from C back to Python, there are two cases in the API where functions *"steal"* references, that is, in which the reference no longer belongs to the calling C function: `PyList_SetItem` and `PyTuple_SetItem`. The given reference, which belonged to the caller, becomes owned by the list or tuple. In the context of the caller, it is now a borrowed reference, which does not have to be decremented anymore. The point of this is to allow nested function calls where, for example, the argument for `PyList_SetItem` is a call that generates a new object to be stored in the list. So, this avoids having to store a pointer to the object only to decrement its reference later.

```
void bug(PyObject* list) {
   PyObject* item = PyList_GetItem(list, 0);
   if (!item) return;
   PyList_SetItem(list, 1, PyInt_FromLong(0L));
   PyObject_Print(item, stdout, 0); /* BUG! */
}
```

(a) Possibly invalid access in `PyObject_Print`

```
void no_bug(PyObject* list) {
   PyObject* item = PyList_GetItem(list, 0);
   if (!item) return;
   Py_INCREF(item);
   PyList_SetItem(list, 1, PyInt_FromLong(0L));
   PyObject_Print(item, stdout, 0);
   Py_DECREF(item);
}
```

(b) `item` is definitely valid in `PyObject_Print`

Figure 3.3: Possibly invalid access in a reference to a `PyObject` in C code

The interaction with the reference counter can be very subtle. The example in Figure 3.3, extracted from the Python documentation, demonstrates that a reference can be invalidated by apparently unrelated code[11]. At first sight, the insertion of an element in `list[1]` seems not to affect the `item` reference, which corresponds to `list[0]`. However, the insertion of `list[1]` may have removed from the list an element that was in this position. In case the list was the last valid reference to the element, it might be collected. The collection of the object can invoke its finalizer method `__del__`, that can run arbitrary Python code. If this code removes the element from position 0 of `list` and this triggers its collection, `item` becomes invalid, because `PyList_GetItem` returns a borrowed reference.

When implementing C functions that return references to `PyObject`s, the same care of defining the lifetime policy of the reference should be taken. To return a new reference to be owned by the caller, it may be necessary to increment the object's count. This manifests itself, for example, in the correct way for a C function to return the value None, which involves calling `Py_INCREF(Py_none)`[12]. Even Python objects representing numbers need to have their reference count controlled by the C programmer.

It is possible to define a deallocation function in the `tp_dealloc` field of the `PyTypeObject` structure so that C code can perform finalization operations over data stored in a Python type defined in C. This function is normally responsible for freeing resources allocated through C code (open files, pointers to memory areas inaccessible from Python, etc.) and

[11] In fact, the documentation informs that older versions of Python contained variants of this bug in some of its modules.

[12] This pattern is so common that the sequence `Py_INCREF(Py_none); return Py_none;` was encapsulated in the `Py_RETURN_NONE` macro.

decrementing references to other Python objects maintained by the object.

When deallocating data structures such as lists it is possible to trigger an arbitrarily large chain of deallocations, as each element causes the deallocation of the next element of the structure. This launches the deallocation function recursively and could easily cause a stack overflow in C. To work around this problem, Python includes a pair of macros, `Py_TRASHCAN_SAFE_BEGIN` and `Py_TRASHCAN_SAFE_END`, that control the accepted number of recursion levels. At each execution of `Py_TRASHCAN_SAFE_BEGIN` an internal counter is incremented. While this counter does not reach the limit value defined in the `PyTrash_UNWIND_LEVEL` constant (50 by default), the function runs normally. When the limit is reached, `Py_TRASHCAN_SAFE_BEGIN` stores the object in an internal list and jumps straight to `Py_TRASHCAN_SAFE_END`, avoiding the deallocation the object and another recursion. At the end of each level of the recursion, `Py_TRASHCAN_SAFE_END` decrements the counter. When the counter reaches zero, `Py_TRASHCAN_SAFE_END` launches again `tp_dealloc` on the elements stored in the internal list, restarting then the recursion on the structure. So, a chain of n deallocations is broken into n/`PyTrash_UNWIND_LEVEL` chains, none of them exceeding `PyTrash_UNWIND_LEVEL` levels of recursion in the C stack. The implementation of the main structured types of Python, such as lists, tuples and dictionaries, make use of this mechanism.

Garbage collection using reference counts brings with it concerns about circular references: a chain of objects maintaining references to each other keeps the count of each of its elements greater than zero, even if they are not reachable from any other object. Python includes a cycle detector, but special measures must be taken to ensure that types implemented in C behave correctly if they can generate cycles. One must implement a function to traverse references contained in the object and a function to decrement their reference counts. These functions must be registered in the `tp_traverse` and `tp_clear` fields of the `PyTypeObject` structure. The `tp_clear` function has to take the precaution of clearing the value of its `PyObject*` fields to `NULL` before decrementing each reference, since the decrement operation may start the deallocation of the object and launch a call to `tp_traverse` which, due to the cycle, may return to the previous object. The type must be, then, identified with the `Py_TPFLAGS_HAVE_GC` flag in the `tp_flags` field of `PyTypeObject`.

Besides, the implementation of Python objects that support cyclic collection in C implies in yet more care. Objects must be allocated with `PyObject_GC_New` or `PyObject_-GC_NewVar` instead of the usual functions `PyObject_New` and `PyObject_NewVar`. During the construction of the object, after the fields to be visited by `tp_traverse` are filled, it is still necessary to call a notification function, `PyObject_GC_Track`, and during deallocation, before invalidating the object's fields, to call `PyObject_GC_UnTrack`. For objects that need the "trashcan" mechanism to avoid stack overflow, it is also necessary to take the precaution of unmarking the object with `PyObject_GC_UnTrack` before entering the `Py_TRASHCAN_SAFE_BEGIN`/`END` block.

In spite of offering a cycle detection mechanism, Python is unable to collect cycles whose objects contain finalizers implemented in Python itself (`__del__` methods); the only way to access those objects is then through the `garbage` list in the `gc` module. This module (accessible to C through Python function calls using the C API) offers an interface to

the garbage collector, including `enable` and `disable` functions, to activate and deactivate the garbage collector; `collect`, to run a collection; `get_objects`, which returns a list containing all objects controlled by the collector (except the list itself); `get_referrers` and `get_referents`, which return the list of object that refer or are referred by a given object – these lists are obtained using the `tp_traverse` function, which may not point to all objects actually reachable, or may still return objects in an invalid state (such as objects in cycles that were not yet collected or objects still not fully constructed) and therefore should be used only for debugging purposes.

3.2.2 Ruby

Ruby uses a mark-and-sweep garbage collector [48]. This technique avoids the problem of cyclic references faced by Python; having valid objects correctly indicated as reachable is sufficient.

Objects that are reachable from the Ruby data space – assigned to a Ruby global variable or inserted in some data structure reachable in Ruby – will not be subjected to garbage collection. In addition, we have objects returned by Ruby to the C space, since many API functions return `VALUE`s. The documentation warns that, to store Ruby objects in C, either in global variables or in data structures, it is necessary to notify the virtual machine that the `VALUE` must not be collected using the `rb_global_variable` function [40] (although not mentioned in the documentation, it is possible to unmark a global value with `rb_gc_unregister_address`).

Objects in the local scope of a C function, however, do not need to be notified. The way how Ruby ensures the validity of local `VALUE`s is remarkably peculiar: when performing the mark phase, the garbage collector scans the C stack looking for values that look like `VALUE` addresses, that is, numeric sequences that correspond to valid `VALUE` addresses. These addresses can be identified because objects are always allocated within heaps maintained by the Ruby interpreter. Each `VALUE` found in the stack is then marked. This ensures that any `VALUE` locally accessible by C code becomes invalidated, but may generate "false positives" stopping data that could be collected from being so.

In spite of programmer convenience, such approach is extremely non-portable. The implementation of the garbage collector in Ruby 1.8.2 has `#ifdef`s for IA-64, DJGPP, FreeBSD, Win32, Cygwin, GCC, Atari ST, AIX, MS-DOS, Human68k, Windows CE, SPARC and Motorola 68000. Besides, the collector forces the discharge of registers to the stack using `setjmp`, to prevent variables of the `VALUE` type that may have been optimized into registers by the compiler from being missed.

As we have seen in Section 3.1.2, Ruby objects created with `Data_Wrap_Struct` contain C structs, and those may contain references to Ruby `VALUE`s. The encapsulated struct, however, is opaque to the Ruby universe. So, to ensure that these `VALUE`s are marked as reachable during garbage collection this has to be done through C code. `Data_Wrap_Struct` accepts, beside the struct to be wrapped, pointers to a mark function and to a deallocation function. When the garbage collector visits the object in the mark phase, it invokes the registered function, which must call `rb_gc_mark` on each `VALUE` stored in the object's

struct, informing thus that these objects are reachable. When an object wrapped with `Data_*_Struct` is considered unreachable, its deallocation function is called. On structures that do not store other `VALUE`s, the mark function can be set to `NULL` and the deallocation function to `free`.

Ruby has a `GC` module featuring functions to turn the collector on and off (`GC.enable` and `GC.disable`), as well as to launch a collection immediately (`GC.start`). There are equivalent functions in the C API: `rb_gc_enable`, `rb_gc_disable` e `rb_gc_start`. The C API includes also a function that inserts an object immediately in the list of objects to be recycled by Ruby's memory allocator, `rb_gc_force_recycle`. This function should be used with caution, since if there are still any references pointing to the recycled object, they will point to the new object when the memory area is reclaimed by the Ruby allocator.

Ruby also offers as a convenience to the C programmer some wrappers to the malloc and realloc functions that interact with the garbage collector, forcing its execution during large allocations[13] or in low-memory situations.

3.2.3 Java

Like in Python and Ruby, the Java API returns references to objects from the virtual machine that can be stored in C variables. The JNI defines three types of references, *local*, *global* and *weak global* references, to aid in controlling their lifetime and their interaction with the garbage collector.

Most functions of the JNI return local references, which are valid until the return of the C function that has obtained them. It is not necessary to deallocate a local reference explicitly: during the execution of a C function, the JVM maintains a list of local references passed to the function and frees them all when control returns to the virtual machine. This way, in general, the programmer does not need to worry about garbage collection when manipulating values returned during a function. On the other hand, in code that may use a large number of local references it is more efficient to free local references explicitly, using `DeleteLocalRef`. In Java version 1.2, functions were added to manage local references in blocks. `PushLocalFrame` and `PopLocalFrame` allow one to create nested scopes of local references, which are freed all at once. `PushLocalFrame` receives also an argument indicating a number of slots to be pre-allocated, as an optimization. This value can also be configured with `EnsureLocalCapacity`.

Global references are generated from local references using `NewGlobalRef`. References of this kind remain valid until they are explicitly deallocated with `DeleteGlobalRef`. A global reference stops the object from being collected; it can therefore be used to store Java objects in C space beyond the duration of a function, for example, in global or static variables.

Figure 3.4 shows an example of the kind of reference management that is needed when a loop creates temporary references for an arbitrary number of objects.

[13] The definition of "large" is adjusted dynamically, based on the execution of the collector and previously performed allocations.

In the example, the Java_Example_concatArray function (equivalent, therefore, to the concatArray method from class Example) converts the elements of an array to strings using Object.toString and concatenates them using String.concat. Notice that, as the number of iterations of the loop depends on the size of the given array, one should prevent the number of references from increasing on each iteration. For that, the options would be either to use Push/PopLocalFrame, or to destroy references one by one with DeleteLocalRef. If we used Push/PopLocalFrame in the example, we would have to keep the temporary string holding the concatenation in a global reference. Further, this reference would have to be destroyed and recreated on each iteration, since strings are immutable in Java. As the number of locals is small, it is more convenient in this case to control them explicitly with DeleteLocalRef than resorting to global references.

PopLocalFrame allows, through an additional argument, transferring a local reference from the set that is being popped to the external scope of local reference, creating this way a new reference. For the example of Figure 3.4, this would still not avoid the need of freeing references explicitly on each iteration of the loop, since each PopLocalFrame would create a new local reference.

Since Java 1.2, the JNI includes weak global references, with the goal of offering a simplified version of Java's weak references (java.lang.ref) – an object pointed only by weak global references can be collected. Originally, the API defined the IsSameObject function as a way to check the validity of a weak reference, but evidently this method is insufficient: since Java is multi-threaded, the garbage collector may invalidate the reference between the test and the following instruction in C code. The revised documentation warns about this limitation and recommends the use of global references, as well as alerting on undefined behaviors in the relationship between weak global references in C and Java's own weak reference types [37].

More issues arise from the combination of Java's multi-threaded model with the exposure of references of virtual machine objects to C code. To reduce the volume of data copying between Java and C, the JNI offers some functions that return and release pointers to the internal representation of strings and arrays of primitive types: Get/ReleaseStringCritical and Get/ReleasePrimitiveArrayCritical. The use of these functions, however, has important restrictions. The API specifies that, once a pointer is obtained through these functions, the C code must not call other JNI functions or perform calls that may block the current thread and make it wait for another Java thread, under risk of a deadlock. It is recommended that memory blocks held using these functions are not retained for a long time, since one of the possible techniques fro implementing this "critical section" consists in disabling the garbage collector. It is also important to note that local references and the pointer to the JNI environment passed to native functions are valid only in the thread where they were created; global references can be shared among threads.

Besides the weak reference mechanism provided by the java.lang.ref package, the only way provided by Java to interact in a more direct way with the garbage collector is through the System.gc() call, which asks the virtual machine to launch the collection thread as soon as possible so that it deallocate unreachable objects. There is no equivalent

```
static jmethodID concat = NULL, toString = NULL;
/* Caching jmethodIDs in C code is a common technique.
   It's worth mentioning that jmethodIDs are not Java objects,
   and are therefore not subjected to garbage collection. */
void cache_ids(JNIEnv* J) {
  jclass cls = (*J)->FindClass(J, "java/lang/String");
  concat = (*J)->GetMethodID(J, cls, "concat",
     "(Ljava/lang/String;)Ljava/lang/String;");
  cls = (*J)->FindClass(J, "java/lang/Object");
  toString = (*J)->GetMethodID(J, cls, "toString",
     "()Ljava/lang/String;");
}

JNIEXPORT jstring JNICALL
Java_Example_concatArray(JNIEnv* J, jobject this, jobjectArray a) {
   if (!concat) cache_ids(J);
   jstring s = (*J)->NewString(J, NULL, 0);     /* s = "" */
   int len = (*J)->GetArrayLength(J, a);        /* len = a.length */
   for (int i = 0; i < len; i++) {
      jobject o = (*J)->GetObjectArrayElement(J, a, i);      /* o = a[i] */
      jstring os = (*J)->CallObjectMethod(J, o, toString);   /* os = o.toString() */
      jstring s2 = (*J)->CallObjectMethod(J, s, concat, os); /* s2 = s.concat(os) */
      (*J)->DeleteLocalRef(J, s);
      (*J)->DeleteLocalRef(J, o);
      (*J)->DeleteLocalRef(J, os);
      s = s2;
   }
   return s;
}
```

Figure 3.4: Routine for concatenating elements of an array represented as strings.

C function in the JNI, but this method can be invoked from C with `CallStaticVoidMethod`.

3.2.4 Lua

The interaction of native C code with the Lua garbage collector is greatly simplified by the fact that the Lua API does not return explicit references to Lua objects to the C space. Operations on Lua objects are always specified through indices of the virtual stack. This way, the virtual machine retains all control over which objects are accessible from C at any given moment.

Although pointers to objects are not manipulated in the API, some functions return pointers to structures managed by Lua: `lua_newuserdata`, `lua_to*string` and `lua_touserdata`. The validity of pointers returned by these functions is dependent on the lifetime of the object they correspond to; for strings in particular, a returned pointer is only decidedly valid as long as the string is in the stack. Lua offers still the `lua_topointer` function, which returns pointers to some kinds of objects (userdata, tables, threads and

functions), but only with the intention of providing debugging information, as it is not possible to convert such pointers back into Lua values.

The virtual stack is emptied when the C function returns control to the Lua virtual machine. This way, it is not possible to retain pointers returned by Lua for later use in global variables or C structures. Alternatively, the API offers a mechanism for storing Lua values in a location that is known to C code and that cannot be altered by Lua code: the registry. The registry is a table made available through the Lua API for the storage of Lua values from C; this table is not normally accessible from Lua. Since the table that implements the registry is part of the root set of the garbage collector, the inclusion of an object in this table prevents it from being collected, keeping it in the registry until it is explicitly removed through C code.

Using the registry, a possible way to describe data from the Lua space in C data structures is to store data in the registry and store the used indices in the C structure. Lua's auxiliary library encapsulates such idiom through two functions, `luaL_ref` and `luaL_unref`. The `luaL_ref` function associates the given Lua value to an integer numeric key in the registry, and returns this number. This value can then be seen as a high-level handle to the object: C code can store it in variables and structures and use it to refer to the object through its registry field. The `luaL_unref` function removes the Lua value from the registry and frees the index for reuse. To ensure that this mechanism works properly, integer keys should not be used directly be the programmer to store data in the registry.

The API allows associating a deallocation function, `__gc`, to the metatable of full userdata objects. When present, this function will be typically implemented in C, performing resource finalization. For example, the `__gc` metamethod of objects returned by the `io.open` Lua function is a C function that closes the corresponding file descriptor using the `fclose` function.

In principle, the fact that its possible to obtain and modify the metatable of userdata through Lua code may seem problematic, as one could replace its finalizer in `__gc`. However, collection functions implemented in C typically validate received userdata checking its "type", identified through its metatable. So, even if Lua code manipulates the table, a collection function implemented in C which uses `luaL_checkudata` will not be made to operate on userdata of incorrect type. To stop Lua code from modifying the collection function of a userdata object, one can assign some value, such as false, to the `__metatable` field of the metatable; this will be returned in place of the metatable, making the metatable itself unaccessible.

Another resource related to memory management provided by Lua is the possibility of configuring, at runtime, the allocation function to be used by the virtual machine. In the creation of a new Lua state, an allocation function is passed as a its first argument. This function must offer functionality like that from the C functions `free` and `realloc`, depending if the given block size is equal or greater than zero.

Lua offers an interface to its garbage collector through two functions: `lua_gc` in C and `collectgarbage` in Lua. The collector implements incremental mark-and-sweep and allows the programmer to configure parameters related to collection intervals, as well as enable, disable, launch full cycles and executing collection steps.

3.2.5 Perl

Like Python, Perl performs garbage collection based on reference counting. The API provides functions for explicit control of reference counts: `SvREFCNT_inc` e `SvREFCNT_dec` and a getter, `SvREFCNT`. Another way to modify the reference count of a value is to assign it to a Perl reference with `newRV_inc`. The count of the referenced value will be incremented, keeping it valid – unless it has its count altered explicitly – as long as it is referenced by the `RV`. It is important to note, however, that API functions that create values, such as `newSViv`, initialize their reference counts with 1. This has the effect that if a value is created in a C function, stored in an `RV` with `newRV_inc` and this reference is returned to Perl, the value will never be collected, because its counter will not reach 0 when the reference is destroyed. The correct form, then, is to use `newRV_noinc` for `RV`s containing newly-created values and `newRV_inc` when an `RV` needs to retain an already existing value.

Initializing reference counts with 1 ensures that values created will remain valid during the execution of a C function without storing the value in Perl space. These values can also be stored in C global variables and data structures and will remain valid until their reference count is decremented. For values with a lifetime restricted to a single function, the Perl API defines the concept of "mortal" variables as a way of deallocating all temporary values of a function at once. An `SV`, `AV` or `HV` can be created with `sv_newmortal` or, more commonly, converted to a mortal with `sv_2mortal`. In practice, marking a value as mortal corresponds to indicating that it should have its reference count decremented by the `FREETMPS` macro by the end of a function, as we will see in Section 3.3.5. Some functions of the API return mortal values: for example, `hv_delete` removes an element of a hash and, unless the `G_DISCARD` is passed, returns the removed element as a mortal `SV`.

The Perl API does not provide facilities for interfacing with the garbage collector, but features some debugging support functions that report information about the state of garbage collection. The `sv_report_used` function displays the contents of every `SV` stored in the interpreter. The `Devel::Peek` module allows examining from Perl the content of values (reference counts, flags, etc.) – from C, this information is directly available, since their structures are not opaque.

3.2.6 Comparison

Garbage collection aims to isolate, as much as possible, the programmer from memory management. This way, ideally an API should also be as independent as possible from the garbage collection algorithm used in the implementation of the virtual machine. Perl and Python perform garbage collection based on reference counting, and this shows through in the reference increment and decrement operations frequently needed during the use of their APIs.

Ruby uses a mark-and-sweep garbage collector. Its API manages to abstract this fact well for manipulation of native Ruby objects, but the implementation of the collector is evident in the creation of Ruby types in C, where we need to declare a mark function when there are C structures that store reference to Ruby objects. The Lua API goes further

when isolating itself from the implementation of the garbage collector: the only point of the API where the use of an incremental garbage collection is apparent is in the routine for direct interaction with the collector, `lua_gc`, where its parameters can be configured.

Of the five languages studied, the only whose API abstracts entirely the implementation of the garbage collector is Java. The only interfacing operation provided by the language, `System.gc()`, does not receive any arguments and does not specify how or when the collection should be done[14]. Indeed, the various available implementations of the JVM use different algorithms for garbage collection.

For manipulating data through the API, Lua and Ruby are the languages that demand the least concerns from the programmer about managing references. Ruby keeps control of references returned to C functions scanning the C stack during garbage collection, detecting the presence of references stored in local variables. Lua avoids the problem altogether, by keeping its objects in the virtual stack and not returning references to C code.

The issue of references stored in local variables is handled by Perl and Java in a similar way, by defining two types of references, global and local (local references are called "mortal variables" in Perl). Local references have implicit management (save a few cases, as discussed in Section 3.2.3). API functions in Java return local references by default, which can be converted to global ones with `NewGlobalRef`. In Perl, the opposite happens, and global references can be converted to local ones with `sv_2mortal`. Java's approach is more interesting, as normally more locally-scoped than globally-scoped variables are used. Values stored globally always need to have some for of explicit management to them, even in Ruby and Lua, through `rb_global_variable` and `luaL_ref`/`luaL_unref`.

3.3 Calling functions from C

The API must provide a form of invoking from C functions to be executed by the scripting language. This involves passing data between these two "spaces", as seen in Section 3.1 and the implications that this brings about the objects' lifetime, discussed in Section 3.2. Because of the static typing of C, it is not possible to use a transparent syntax for calling functions registered at runtime. It is therefore necessary for the API to define functions for performing calls to the scripting language.

In this section, we will discuss the facilities provided by each API for invoking functions to be executed by its virtual machine. The main issues involved are how to reference the function to be called, how to pass arguments to it and how to obtain the return value, including forms of notification in case of errors. For illustration purposes, for each language we will present an example of a simple function call. Assume that in the space of each scripting language a `test` function was defined, which receives an integer and a string as arguments and returns an integer as a result. For brevity, error handling will be omitted in the examples.

[14]The documentation is purposely vague, stating only that this method "suggests that the Java Virtual Machine expend effort toward recycling unused objects" .

3.3.1 Python

When calling a Python function from C, one should initially obtain a pointer to the `PyObject` corresponding to the function, as seen in Section 3.1.1. Besides functions implemented in Python and C functions registered through the Python API, any data type that implements the `__call__` method (or declares a function in the `tp_call` field of its `PyTypeObject` structure) can be called as a function.

The Python API offers several functions for performing calls from C. The most general function, `PyObject_Call`, receives as arguments the object to be called, a Python tuple containing the arguments to be passed and optionally a dictionary of keyword arguments. As a convenience, other functions allow passing arguments in other ways. For example, `PyObject_CallFunction` encapsulates the call to `Py_BuildValue` (seen in Section 3.1.1), accepting directly the format string and the value to be converted. `PyObject_CallFunctionObjArgs` is a vararg function that accepts a sequence of pointers to `PyObjects`.

There are also many convenience functions for method invocation. The `PyObject_CallMethod` is a variant of `PyObject_CallFunction` that receives as arguments a `PyObject` and a C string containing the method name. So, for example, both forms below are equivalent to the Python statement `ret = some_string.split(" ")`:

```
/* "s" indicates that the next parameter is a string */
PyObject* ret = PyObject_CallMethod(some_string, "split", "s", " ");

PyObject* split = PyObject_GetAttrString(some_string, "split");
PyObject* ret = PyObject_CallFunction(split, "s", " ");
```

It is interesting to note that when a method is called as a function, the `self` argument is not passed explicitly.

The return value in all invocation functions is a `PyObject` pointer. As it happens in Python code, when Python functions return multiple values, they are encapsulated in a tuple. For functions that do not return a value, C functions must return `Py_None`. In case of errors in the call, these functions return `NULL`. The occurrence of exceptions can then be verified with the `PyErr_Occurred` function.

A typical way of calling a Python function called `test`, including retrieval of the function and conversion of input and output values between Python and C, is shown below:

```
PyObject* globals = PyModule_GetDict(PyImport_AddModule("__main__"));
PyObject* test = PyDict_GetItemString(globals, "test");
/* "si" indicates string and integer arguments */
PyObject* obj_result = PyObject_CallFunction(test, "si", "foo", 2);
/* Converts the value to C */
long result = PyInt_AsLong(obj_result);
/* Frees the temporary PyObject that was returned */
Py_DECREF(obj_result);
```

A global function is obtain through the dictionary in the `__main__` module. The conversion of input data from C to Python is made through a format string received by

PyObject_CallFunction. This call is equivalent to obj_result = test("foo", 2) in Python. The output value is returned as a new reference to a Python object and, as such, needs to have its reference count decremented after its use. The PyImport_AddModule, PyModule_GetDict and PyDict_GetItemString return borrowed references, therefore the reference count of PyObjects returned by them do not need to be decremented after their use. However, after calling the Python function, there is no guarantee that the globals and test pointers still point to valid objects – we would have to have incremented their reference counts if we wanted to use them again.

3.3.2 Ruby

Since methods are not first-class values in Ruby, they are not represented as VALUEs in their C API. For calling Ruby methods in C, the API offers the rb_funcall function and some variations. In common, all of them receive as an argument the VALUE indicating to the object the method refers to, an ID referring to the interned string containing the method name and an integer informing the number of arguments.

Like in Python, the API functions for method invocation differ in how arguments are passed. For example, rb_funcall receives arguments as VALUEs passed as C varargs; rb_funcall2 receives a C array of VALUEs; rb_apply receives a VALUE that must be a Ruby array containing arguments. All of them return a VALUE as an argument. Like in Ruby code, multiple return values are represented as a Ruby array.

All function call routines from the API refer to methods, expecting thus an object on which the method should be applied. Global functions are defined in Ruby as methods of the Kernel module, which is included by the class Object and are, therefore, accessible from every object, including nil. This way, one can invoke global functions passing the C constant Qnil as the method's target object.

Below, we present the typical way a Ruby global function test is called from C, again including conversion of input and output values between C and the interpreter.

```
ID test = rb_intern("test");
VALUE val_result = rb_funcall(Qnil, test, 2, rb_str_new2("foo"), INT2NUM(2));
long result = NUM2LONG(val_result);
```

Unlike it happens in Python, it is not necessary to obtain a reference to the function, sufficing to pass its name as an ID and the object it refers to (in this case, Qnil, indicating a global function). Conversion of input data from C to Ruby is done through the rb_str_new2 function and the INT2NUM macro, which return VALUEs.

As discussed in Section 3.2.2, the control of validity of VALUEs is done implicitly. So, we can call functions that create VALUEs directly when passing parameters to rb_funcall. Actually, all three lines above could have been condensed, passing rb_funcall as a parameter to NUM2LONG and rb_intern as the second argument of rb_funcall; they were separated here for greater readability.

A data type that is treated in a quite irregular way in Ruby is that from code blocks. Ruby features special syntax for declaring blocks: they can only be defined as the last

argument of a method call. This way, they are not first-class values and cannot be, for example, declared in a variable assignment. They can, however, be promoted to first-class values, as objects of the `Proc` class. This can be done in two ways: explicitly, passing a block to the `Proc.new` method, or implicitly, when a block is passed to a method that declares a final formal argument preceded by `&`. This variable will contain the block converted to a `Proc`. When calling functions that expect blocks, `&` converts a `Proc` to a block. `Proc` objects can be manipulated through the C API as any other Ruby object, but there is no match in the C API to the functionality of the `&` operator in function calls.

The special status of code blocks complicates their use from C code, and in particular the invocation of methods that expect them as a parameter. Say we want to invoke the following Ruby method from C:

```
def a_ruby_function()
   print("a_ruby_function will invoke the block.\n")
   yield
   print("a_ruby_function is done.\n")
   return 42
end
```

This function expects a code block to be passed to it, so it can be invoked by the `yield` command. Since we will invoke the function from C, we also want to pass C code as a block, represented by the following function:

```
VALUE a_C_block() {
   fprintf(stderr, "a_C_block is running.\n");
}
```

The conversion of `Proc` objects to blocks performed by the `&` operator in Ruby has no equivalent in the C API. Therefore, `rb_funcall` is unable to pass `Proc`s to functions expecting blocks. The intuitive way of doing Ruby function calls from C, in this case then, does not work:

```
ID a_ruby_function = rb_intern("a_ruby_function");
/* The second argument is an additional argument to be
   optionally passed when invoking a Proc */
VALUE a_proc = rb_proc_new(a_C_block, Qnil);
/* Doesn't work! A Proc is not a code block */
VALUE result = rb_funcall(Qnil, a_ruby_function, 1, a_proc);
```

The only ways for invoking a Ruby method passing a code block are through `rb_eval_-string` and `rb_iterate`. The first approach, besides the performance penalty caused by parsing a string of code, has the inconvenience of requiring temporary variables so that one can obtain return values back to the C data space. In the model using `rb_eval_string`, the C function that will act as a block must be declared in the Ruby space. There are two alternatives on how to do this: registering the method in Ruby and invoking it in a wrapper block declared in the string of Ruby code:

```
/* Declares a global function with 0 input parameters */
rb_define_global_function("a_C_block", a_C_block, 0);
rb_eval_string("$result = a_ruby_function { a_C_block() }");
VALUE result = rb_gv_get("$result");
```

Or encapsulating the function in a `Proc` object from C with `rb_proc_new` and then using the `&` notation in the string of Ruby code:

```
VALUE a_proc = rb_proc_new(a_C_block, Qnil);
rb_gv_set("$a_proc", a_proc);
rb_eval_string("$result = a_ruby_function(&$a_proc)");
VALUE result = rb_gv_get("$result");
```

The second approach makes use of the fact that the only API function that is capable of producing code blocks directly is `rb_iterate`. This function receives two function pointers, one to the function to be invoked and another to the function that will act as a block; calls to `yield` withing the first function will invoke the second one. The block may break the flow of execution with `rb_iter_break`. By passing as a "iteration function" to rb_iterate a wrapper function that simply calls the desired Ruby method with `rb_funcall`, it is possible to simulate a call to `rb_funcall` that receives a C function as a code block.

```
VALUE call_a_ruby_function() {
   ID a_ruby_function = rb_intern("a_ruby_function");
   return rb_funcall(Qnil, a_ruby_function, 0);
}
...
/* The Qnil arguments indicate that there are no parameters
   to be passed to either function */
VALUE result = rb_iterate(call_a_ruby_function, Qnil, a_C_block, Qnil);
```

Notice that no arguments are passed to `rb_funcall` – `rb_iterate` defines `a_C_block` as the "current code block" and this definition is inherited implicitly by `rb_funcall`.

For the common case of performing iteration on the `each` method of collections, Ruby offers a wrapper function, `rb_each`. This function was designed to be passed as a first argument to `rb_iterate`. C functions executing as a code block can break the control flow with `rb_iter_break`. The yielding mechanism, for both C code and native Ruby calls, is implemented using the `setjmp` and `longjmp` C functions.

For correct error handling, C functions that perform calls to Ruby functions must be encapsulated by a `rb_protect` call or one of its variants, `rb_ensure` and `rb_rescue`. If a program does not use `rb_protect`, Ruby exceptions will result in fatal errors.

3.3.3 Java

Like in attribute access, when calling Java methods from C one must initially obtain a method identifier, of the `jmethodID` type. These identifiers are typically obtained with the `GetMethodID` function, which receives as arguments the class (an instance of jclass)

and two strings, one with the method name and the other with the method signature. The syntax for describing method signatures is similar to that of field descriptors discussed in Section 3.1.3. Arguments are listed in parentheses, followed by the return type. For example, "([Ljava/lang/String;II)V" describes a function with String[], int, int arguments and void return. Alternatively to GetMethodID, since Java 1.2 it is possible to obtain a jmethodID corresponding to a method by applying the FromReflectedMethod function on a Java object of the Method class – that is, a method reified through Java's reflection API.

Once a jmethodID was obtained, a method can be invoked through some of the 90 functions of the Call*Method* family. Function names follow this format:

Call<*type*><*return*>Method<*arguments*>

Here, <*type*> may be Static for static functions, passing a jclass as an argument in calls; Nonvirtual when invoking implementations of a method from a specific class on a given object, passing a jclass and a jobject as parameters; or omitted for instance methods, passing the jobject on which the method will be applied. Return type is indicated in <*return*>: Void, Object, Int, etc.

Method arguments can be passed in three ways: as varargs, as a C array of jvalues, or propagating a received va_list. For example, in its simplest form, an instance method without input or output values is invoked with CallVoidMethod. CallStaticIntMethodA, in contrast, invokes a static method which returns a jint and has its argument list passed in an array of jvalues. Since Java is a statically typed language, it is not necessary to specify the number of type of arguments passed in functions for method invocation. This information is already specified in jmethodIDs.

It is important to point out that, when obtaining method and field identifiers resolving them based on the jobject obtained in the this variable and the method or field name, with GetObjectClass and GetFieldID, we are effectively resolving names through dynamic scoping. This implies that, for example, if a method called Parent.method implemented in C accesses a private attribute anAttribute and a Child subclass also defines a private attribute with the same name, a call to this method in an instance c of Child would end up accessing Child.anAttribute and not Parent.anAttribute. This behavior differs from what would happen if Parent.method was implemented in Java, where binding of private members is defined lexically. To ensure to the C implementation of Parent.method that the anAttribute it is accessing is really Parent.anAttribute, one must store in C space the field identifier from Parent's jclass – obtained, for example, in a static native function.

C code may verify the occurrence of exceptions through ExceptionCheck and choose to handle it, obtaining a local reference of the exception with ExceptionOccurred and later clearing it with ExceptionClear; or keep it active so that it propagates to Java code.

For the example of the test function, since Java does not have global functions, we will assume that test is a static method of a class called Example and that we are running the following C code in a context where we have a reference to a Java runtime environment called J (this pointer, of JNIEnv type, will be discussed in Section 3.4.3).

```
jclass example = (*J)->FindClass(J, "Example");
jmethodID test = (*J)->GetStaticMethodID(J, example,
                        "test", "(Ljava/lang/String;I)I");
jstring foo = (*J)->NewStringUTF(J, "foo");
long result = (*J)->CallStaticIntMethod(J, example, test, foo, (jint)2);
```

Initially, we get a reference to the `Example` class, from which we obtain the identifier of the desired method, based on its name and signature. Like in Ruby, the string passed as a parameter must be converted to a virtual machine type, but for the second argument and for the return value we explore the fact that `jint`, corresponding to the Java type `int` (32-bit integer) is compatible with the C type `long` (defined as an integer of at least 32 bits). All these API functions return local references, which will be freed automatically by the end of the C function where the API calls were made.

3.3.4 Lua

Both in C functions launched by Lua and in Lua function calls performed from C code, input arguments and return values are passed through the virtual stack presented in Section 3.1.4.

To call a Lua function from C, we must initially push to the stack the Lua object referring to it: for global functions, obtaining it with `lua_getglobal`, for functions stored in tables, with `lua_gettable`. Afterwards, we push its arguments and then invoke `lua_call` or `lua_pcall`, indicating how many stack values are to be passed as a parameter. The difference between these two functions is in error handling: `lua_call` propagates any signalled errors, using `longjmp`; `lua_pcall` captures errors, returning a status code and the error message in the stack.

When no errors occur, the stack will contain any values returned by the called function. The number of return values can be explicitly requested with lua_call or lua_pcall, or be defined at runtime, requesting the special value `LUA_MULTRET`. If a specific number of return values is requested and these are not passed by the called function, the number of values will be adjusted by adding `nil` elements or discarding excessive values. For calls with `LUA_MULTRET`, all values are pushed. In this case, the only way to find out how many values were returned is comparing the size of the stack before and after the call.

The `lua_cpcall` function allows calling C functions performing error capture like that which takes place on `lua_pcall` without having to register them as Lua values. This functionality is similar to that offered by `rb_protect` in Ruby. Ruby, however, does not offer any function analogous to `lua_pcall`, being sometimes necessary to wrap Ruby function calls in C functions that follow the signature expected by `rb_protect`.

Lua does not have a distinction between functions and methods, but features syntactic sugar that allows one to invoke functions stored in tables with a method-call-style syntax: `t:m(x)` means `t.m(t,x)`. Still, there is no specific call in the C API to replicate this abbreviation. For functions stored in tables, the function must be obtained with `lua_gettable` and the table has to be pushed explicitly alongside the other parameters.

The example of the test function call demonstrates the stack discipline adopted by the Lua API. Similarly to the Java example of Section 3.3.3, we will assume the existence of a lua_State pointer called L, which will be explained later on in Section 3.4.4.

```
lua_getglobal(L, "test");             /* Pushes the function test */
lua_pushstring(L, "foo");             /* Pushes the string "foo" */
lua_pushinteger(L, 2);                /* Pushes the number 2 */
lua_call(L, 2, 1);                    /* Calls the function with 2 arguments,
                                         expect 1 as the result */
long result = lua_tointeger(L, -1);   /* Get the result at the top (-1) */
lua_pop(L, 1);                        /* Remove it off the stack */
```

With lua_getglobal, we push the global function test. Then, two input arguments are pushed. The function is invoked with lua_call, indicating two input arguments and requesting one output value. The return value, at the top of the stack (index -1) is converted to C with lua_tointeger. This last function does not pop the value off the stack: to return it to its initial state, we need to remove it explicitly with lua_pop. As the API never returns pointers to Lua objects, there are no concerns related to garbage collection.

3.3.5 Perl

Calling Perl functions from C is done through a stack discipline, like in Lua. Input parameters are specified through push operations and return values are obtained from the stack after the function call. The call functions call_sv, call_pv and call_method vary only in the way how the function to be called is specified: through an SV, a C string, or a C string describing the name of a method in some object or class previously inserted in the stack. The call_argv function, as a convenience, receives as an additional argument a C array containing C strings representing arguments to be pushed. All of them return the number of return values available in the stack.

All call_* functions receive an argument with flags to be passed that indicate how the function should be called and how to handle input arguments and return values. G_VOID, G_SCALAR and G_ARRAY indicate the context how the function should be called. In scalar contexts, for example, only one scalar is returned in the stack; if the called function returns a list, only its last element will be available on the stack. G_DISCARD specifies that return values should be automatically discarded; G_NOARGS indicates that the default array of parameters, @_, should not be constructed[15].

The procedure for error checking depends on the context and given flags, which affect how error situations are reported to the return value of call_* functions and to values returned on the stack. The G_EVAL flag encapsulates the call in an eval block, capturing errors. So, the occurrence of errors can be checked through the ERRSV macro, which returns an SV containing the error message. By adding the G_KEEPERR flag, error messages do not

[15] This has the side effect that the called function inherits the value of @_ from its caller.

overwrite the special variable $@, but concatenate to it, accumulating sequences of errors in different call levels.

A series of macros describe a protocol for calling functions and manipulating input and output parameters. The main ones will be explained below, as we present the Perl version of the test function call:

```
dSP;
ENTER;
SAVETMPS;
PUSHMARK(SP);
XPUSHs(sv_2mortal(newSVpv("foo", 0)));
XPUSHs(sv_2mortal(newSViv(2)));
PUTBACK;
call_pv("test", G_SCALAR);
SPAGAIN;
long result = POPl;
PUTBACK;
FREETMPS;
LEAVE;
```

First, dSP declares a local copy of Perl's stack pointer. Then, ENTER and SAVETMPS create a scope for mortal values. PUSHMARK starts the count of parameters to be passed to the function. These parameters are then pushed with XPUSHs. Values created with newSVpv and newSViv are converted to mortal values with sv_2mortal, so that they do not need to have their reference counts decremented explicitly after the function call. PUTBACK finishes counting parameters. Then, we call the global Perl function test, in scalar context, with call_pv.

After this function returns, memory in Perl's stack may have been reallocated, changing the address of the stack pointer obtained with dSP. To make sure its value is correct, one must call SPAGAIN after call_* functions. The POPl function pops a value and converts it to long (there are similar functions for other types, such as POPs for SVs and POPpx for strings). These operations pop values updating the local copy of the stack pointer. Thus, PUTBACK must be called again to update the global pointer. Finally, FREETMPS and LEAVE decrement the reference count of mortal values.

3.3.6 Comparison

In Python, Lua and Perl, functions can be accessed as language objects and invoked. In Ruby and Java, the API defines special types used to reference methods. Like in data manipulation, Python offers an extensive API, with several convenience functions allowing parameters to be passed as Python tuples, Python objects given as varargs, C values to be converted by the invocation function, etc. Java also offers a large number of method invocation functions and, due to static typing, input parameters can be passed as varargs in a direct way, without having to specify how their conversion should be made. Ruby also offers some variants for call functions.

Lua, in contrast, separates the function call routine from argument passing, which is done previously through the stack. This is a very simple solution, but the resulting code is less clear than the equivalent calls in languages such as Ruby and Python. Perl also features function calls using a stack model, but unlike Lua its use is exceedingly complex, demanding a macro protocol to be followed which exposes the internal workings of the interpreter. Another complicating factor is the handling of return values, for these vary according to the Perl context in which the function is called.

In Lua and Python, the occurrence of errors can be checked through the function's return value. In a similar way, Perl allows detecting errors in the most recent call checking a special variable; in Java, this is done calling an API function. In Ruby, error handling is more convoluted: strangely, the API offers a function for invoking C functions in protected mode, but lacks an equivalent for calling Ruby functions. It is necessary to write a wrapper function in those cases, which will be illustrated in Section 4.2.5.

3.4 Registering C functions

To allow the invocation of C functions from code written in a scripting language, its API must provide a way to register these functions in the execution environment. In statically typed languages, such as Java, to make it possible to call external functions using the same syntax as native calls, the set of external functions must be declared *a priori* in some way. On the other hand, in dynamically typed languages, as it is the case with Python, Lua, Ruby and Perl, functions can be used directly; defining them at some point in time before their call is sufficient. This way, one can declare external functions at runtime through C code using the scripting language API.

Again in this section, the presentation of each language will conclude with an example. A C function, which like in the previous section's examples, receives an integer and a string and returns an integer, will be registered. We will present as well, for each language, how to register this function as a global value[16] `test` so that it can be directly invoked from the language or through the API.

3.4.1 Python

Python does not have a proper "function" type declarable from C. Class methods, however, are objects and have a specific type, which can be verified with the `PyMethod_Check` function. Typically, methods are created passing an array of `PyMethodDef` structures. These structures are composed by the name of the function, the pointer to the C function, a flags vector and a documentation string. These flags are used to indicate the convention adopted for input parameters in the C function. The most common flags are: `METH_NOARGS`, used for Python functions which receive no arguments, indicating that the C function will receive a sole `PyObject` pointer which will contain the method's `self`; `METH_VARARGS`, for

[16] Or in the case of Java, a static method.

functions that receive as a second parameters a Python tuple containing a variable number of parameters passed from Python to C; and `METH_KEYWORDS`, to indicate that the C function receives as a third parameter a dictionary containing keyword arguments passed to the function.

With this information in hand, API functions which operate on `PyMethodDef` arrays can create and associate method objects to Python's space. `Py_InitModule`, for example, initializes a module with functions from a `PyMethodDef` array. Likewise, methods of a class implemented in C can be given in the tp_methods field of the `PyTypeObject` structure relative to the class.

C functions registered in Python must return a pointer to PyObject, or NULL in case of error (optionally declaring an exception with `PyErr_SetString` or `PyErr_SetObject`). Functions that do not return values must return the pre-defined object `Py_None`, always keeping in mind reference counting issues for returned values as discussed in Section 3.2.1.

Although methods are usually created in C using `PyMethodDef` structures, it is also possible to create a method object explicitly from C with the `PyMethod_New` function, passing as a parameter any callable Python object and the object or class is should refer to. As seen in Section 3.3.1, Python object can be made callable by implementing a `__call__` method in Python or associating a C function to the `tp_call` field of their corresponding `PyTypeObject`.

A simple implementation of a C function that can be registered in Python as the `test` global function is given below:

```
PyObject* test_py(PyObject* self, PyObject* args) {
   char* foo; long n;
   /* In case of argument errors, PyArg_ParseTuple
      raises the appropriate exception automatically */
   if (!PyArg_ParseTuple(args, "sl", &foo, &n))
      return NULL;
   printf("Received: %s and %ld \n", foo, n);
   return PyInt_FromLong(42);
}
```

Since arguments were received as a tuple in the second parameter, the function signature corresponds to the `METH_VARARGS` flag. Input arguments are converted to C and checked with `PyArg_ParseTuple`. The return value is converted from the native C type to a `PyObject` with `PyInt_FromLong`, generating a new reference.

The Python API is designed primarily to the development of extension modules for the language. Though several functions exist for registering methods in classes and initializing modules with function lists, there is no direct way for registering global functions in the virtual machine. A possible way is using the utility routing for method lookup, `Py_FindMethod`, and inserting the returned method in the dictionary of the global module `__main__`:

```
PyObject* globals = PyModule_GetDict(PyImport_AddModule("__main__"));
static PyMethodDef test_def[] = {
```

```
    { "test", (PyCFunction) test_py, METH_VARARGS, "a test" },
    { NULL }
};
PyObject* test_obj = Py_FindMethod(test_def, NULL, "test");
PyDict_SetItemString(globals, "test", test_obj);
```

Notice that `NULL` was passed to `Py_FindMethod`, indicating that there is no object the method will be part of. Because of that, the `self` argument received by the C function `test_py` will also be `NULL` and can be ignored. The `test_def` array was declared `static` to ensure that the `PyMethodDef` will remain valid as long as the global function is registered, as in the creation of `test_obj` a pointer to it is stored internally in the newly created object.

3.4.2 Ruby

For C functions to be callable from Ruby, they must be declared as methods of some class or module, or as a global function. For that, a C function pointer and the number of arguments expected by the function is passed to one of the appropriate functions of the Ruby API: `rb_define_method`, `rb_define_module_function`, `rb_define_global_function` or `rb_define_singleton_method`. The number of arguments passed indicates the expected signature for the C function. Ruby supports explicitly C functions with up to 15 arguments; as an alternative, the special values -1 and -2 indicate, respectively, that the C function will receive its arguments as a C array of `VALUE`s or as a `VALUE` corresponding to a Ruby array.

Resembling the `PyArg_ParseTuple` function discussed in Section 3.1.1, Ruby features a function designed to simplify the processing of input values in C functions: `rb_scan_args`. This function can be used when input arguments are received in a Ruby array. Like `PyArg_ParseTuple`, it is a vararg function which receives a format string indicating the number of arguments to be collected. Unlike its Python counterpart, though, it does not perform type checking in its arguments. The format string allows indicating the minimum and maximum number of parameters that will be accepted and if exceeding parameters should be collected into a Ruby array.

Once declared in Ruby's object space, a C function can be called like any other method. The C function can check if Ruby code has passed it a code block through the `rb_block_given_p` function. The block can then be invoked with `rb_yield`, which receives a `VALUE` as an argument. To pass multiple arguments to `rb_yield`, one must pass a Ruby array. To obtain a `VALUE` of the Proc type produced from a received code block it is necessary to use `rb_scan_args`, which provides functionality similar to that of the `&` operator in Ruby function declarations.

C functions implementing Ruby methods must always return a `VALUE` (`Qnil` when there is no result). Functions that return multiple values must do so through Ruby arrays.

Proceeding with the series of examples, the Ruby global function `test` can be implemented in C as follows:

```
VALUE test_rb(VALUE self, VALUE val_foo, VALUE val_n) {
   char* foo = StringValuePtr(val_foo);
   long n = NUM2INT(val_n);
   printf("Received: %s and %ld \n", foo, n);
   return INT2NUM(42);
}
```

Conversion of input `VALUE`s is done with the `StringValuePtr` and `NUM2INT` macros. There is no explicit code for error handling during these conversions because these macros trigger exceptions that escape the function using `longjmp` in case the conversion was not possible. For output, a `VALUE` is produced with the `INT2NUM` macro. The first input argument is necessary because of the convention of function signatures adopted by the API, but for global functions it should be ignored.

As Ruby offers an API function for defining global functions, the registration of `test` is very simple:

```
rb_define_global_function("test", test_rb, 2);
```

We indicate the Ruby name of the function, the corresponding C function and the number of arguments it expect (not including `self`).

3.4.3 Java

Methods declared in Java that are not implemented in the language itself must be declared through a prototype including the `native` modifier. Thus, `native` does not refer to a native implementation in Java, but to the fact that the method's code will be compiler with native code of its runtime environment, as opposed to virtual machine bytecodes. The implementation of the method, usually wrapped in a C dynamic library, must be loaded before its execution using the `System.loadLibrary` call in Java, usually in a static block of the corresponding class. For each `native` method, a matching C function must be declared in the loaded library.

The `javah` utility generates C header files from Java classes, with prototypes for C functions following the format specified by the JNI. This format specifies not only the signature of input parameters and return types, but also the name of the function, so that the loader can link the C function to the proper Java method in the virtual machine. Functions must be called Java_ <*class name*>_ <*method name*>. In case of function overloading, a suffix is added indicating the type of input parameters (for example, `Java_SomeClass_method__DI` for the version of `SomeClass.method` which accepts a `double` and an `int` as arguments).

The function input arguments are a pointer to `JNIEnv`, which represents a thread in the JVM, a `jobject` representing the object on which the method was applied (or a `jclass` for static methods) and the remaining arguments of the Java method in their C representations, discussed in Section 3.1.3. As the types of given arguments are defined statically both in Java and C, it is not necessary to perform type checking of received arguments in C code. The signatures of functions which implement methods, specified in header files generated by the `javah` tool, already declare correct types.

The return value corresponds the equivalent C type to the return type declared in the Java method. Values represented as reference types can be returned either as local or global references. Besides handling or propagating errors as discussed in Section 3.3.3, C functions can also generate exceptions with `Throw` and `ThrowNew` and return immediately. The return value will be ignored when the exception is captured in Java code.

Since the way for exposing to the virtual machine Java functions implemented in C is different than that used in languages presented earlier, we will start by declaring the function to the Java space, and present the C implementation of `test` afterwards. In the Java class, we declare a `native` method:

```
public class Example {
   static native int test(String foo, int n);
   // ...other class members
   static {
      System.loadLibrary("Example");
   }
}
```

After compiling this class we can pass it to the `javah` command, which will generate a C header. This file will contain the name and signature of the C function that the JVM will lookup in the library that will be loaded by `System.loadLibrary`[17]. This library will implement functions relative to methods declared as `native`.

Below, we present a C implementation, using the header generated by `javah`, for the `test` method:

```
#include <jni.h>
#include <stdio.h>
/* Header generated by javah */
#include "Example.h"

JNIEXPORT jint JNICALL
Java_Example_test(JNIEnv* J, jclass c, jstring obj_foo, jint n) {
   const char* foo = (*J)->GetStringUTFChars(J, obj_foo, NULL);
   printf("Received %s and %ld \n", foo, n);
   (*J)->ReleaseStringUTFChars(J, obj_foo, foo);
   return 42;
}
```

`JNIEXPORT` and `JNICALL` are macros defined in jni.h to provide greater portability to the resulting C code. Since the method was declared `static` in Java, a class reference is received as an argument to the function. The remaining arguments correspond to the arguments of the Java method, and are given in the equivalent types defined by the JNI. As discussed in Section 3.1.3, the JNI handles reference types and immediate types differently. Because of that, only `obj_foo` needs to be converted to C; both `n` and the return value are used directly as basic C data types.

[17] The argument given to it in Java code is used as a basis when constructing a platform-dependent name. In Unix systems, for example, `System.loadLibrary("Example")` loads the file `libExample.so`.

The string obtained with GetStringUTFChars is converted to UTF-8 from its internal Unicode representation in Java. The same pointer can be returned by the JVM to different threads that request the same string. This way, C code must explicitly notify its release with ReleaseStringUTFChars.

3.4.4 Lua

C functions exposed to Lua must match the lua_CFunction type, receiving as a single argument a pointer to a variable of the lua_State type and returning an int. A lua_State encapsulates the entire state of a Lua virtual machine; multiple Lua states can be maintained in parallel. Every function of the core API receives a lua_State as a first argument, except for lua_newstate, which creates a new lua_State.

At the beginning of the C function, arguments given to it are available in the virtual stack. Like in Lua functions, there is no checking on the number of arguments given to a C function invoked from Lua or through the API. C code can check the number of received arguments inspecting the size of the stack.

The auxiliary library also provides functions for checking in a more convenient way types of passed arguments. Functions of the luaL_check* family (luaL_checkint, luaL_checkstring, etc.) check the type of a stack elements and return them, signalling error if the element type is not as requested. The luaL_opt* functions behave in a similar way, allowing also to indicate a default value if the requested element is absent or nil.

Return values are also passed by the C function back to Lua through the virtual stack. The integer value returned by the C function indicates how many elements of the stack should be returned to the caller function. Any remaining values are discarded.

A C function of the lua_CFunction type can be passed to Lua through the lua_pushcfunction call. Lua has also some convenience functions for registering a set of C functions at once. Like when using PyMethodDef arrays in Python, the luaL_register function registers a list of functions, receiving an array of luaL_Reg structures containing names and function pointers.

A C function implemented the example function test is given below:

```
int test_lua(lua_State* L) {
   const char* foo = luaL_checkstring(L, 1);  /* Get the first argument */
   long n = luaL_checkinteger(L, 2);         /* Get the second argument */
   printf("Received %s and %ld \n", foo, n);
   lua_pushinteger(L, 42);                     /* Push the return value */
   return 1;               /* Return one value, off the top of the stack*/
}
```

The function's signature matches the definition of lua_CFunction. Input parameters are obtained from stack positions 1 and 2 and their types are checked using the auxiliary library functions luaL_checkstring and luaL_checkinteger. These functions signal error in case of conversion failure, causing a longjmp like in Ruby.

The type of the obtained string is const char*, as it points to a memory block managed by the virtual machine. In Lua, however, it is not necessary to notify explicitly the release

of the string, because it remains valid as long as the value is on the stack. As the functions luaL_check* do not pop the parameters and the stack is emptied implicitly by the end of the C function, the obtained C string will remain valid during the whole function.

At the end of test_lua, the return value passed to Lua is pushed using lua_push-integer. The return value of the function in C, 1, indicates to the virtual machine that there is a single output value to be fetched from the stack and used as a function result in Lua.

The function is registered in Lua creating a Lua object of the function type from the C function and storing this object in a global variable. This could be done with lua_pushcclosure and lua_setglobal, but Lua's header file has a macro that wraps these two calls. Therefor, hte function can be registered simply with:

```
lua_register(L, "test", test_lua);
```

Using lua_pushcclosure, it is possible to associate to a C function Lua values that will be accessible to the function every time it is called, akin to local static variables in C. This feature provides functionality that is similar to Lua closures, but in a more limited form: these values are private to C functions, while in Lua two closures defined in a single scope will access the same variables, that is, changes to values in one function will affect the other. This restricted form, however, is often enough for implementing in C "stateful functions" such as iterators and generators. Once registered in Lua, C functions are seen as values of the function type, no differently than Lua functions. In fact, lua_pushcfunction is a particular case of lua_pushcclosure in which no Lua values are associated to the function.

3.4.5 Perl

As discussed in Section 3.1.5, the interface between Perl and C was designed having in mind that the connection between C functions and the Perl interpreter is made through generated code from a description given in a higher-level language, XS. XS code consists of function signature declarations using a special syntax, indicating conversion rules for input and output parameters, and C code describing the implementation of these functions. XS was designed for the development of Perl extensions including functions implemented in C: the end result of the compilation of code generated by the XS tools (h2xs, xsubpp) are C and Perl code that combined describe a Perl package (a set of variables and functions stored under a common namespace).

There is a public API for manipulating Perl data in C code, but this consists basically of the interpreter's internal structures exposed for use by the XS pre-processor, extended with some macros for programmer convenience. In fact, Perl does not expose a documented API for registering functions [30]. Because of that, it is not practical for an application to embed a Perl interpreter and expose to it a set of C functions using C code only. The alternative is to create a Perl extension using XS which exposes functions from the application and import the resulting package into the embedded interpreter. We observed the use of this

approach in Perl scripting plugins of several applications[18].

The `h2xs` utility generates a directory containing the skeleton of a Perl module: a Makefile generator script, `.xs` and `.pm` files to be filled by the programmer with XS and Perl code, as well as auxiliary files. Resuming the example of the `test` function, this is how it would be declared in XS:

```
long test(foo, n)
        char* foo
        int n
        CODE:
                printf("Received %s and %ld \n", foo, n);
                RETVAL = 42;
        OUTPUT:
                RETVAL
```

The .xs file is converted to `.c` with `xsubpp`. C code for converting input and output parameters is generated automatically. In some cases, however, we need to manipulate values from the Perl stack explicitly, as described in Section 3.1.5. In vararg functions, for example, additional arguments must be accessed directly from the stack. Code for registering module functions is also generated automatically.

XS creates variable called `RETVAL` automatically for storing the return value in C code. The contents of this variable are then converted to a Perl value by generated C code. To make sure that functions returning arrays will operate correctly in scalar contexts, their code should verify the context the function was called with `GIMME_V` and then return an `SV` or `AV` accordingly. In those cases, a function must be declared with `SV*` as the return type, and as such, C values have to be converted to Perl `SV`s explicitly. The documentation alerts that, for the case of `AV`s, one must declare the return value as a mortal variable[19].

Once a extension is compiled using Makefiles generated by `h2xs`, it can be loaded an used from Perl:

```
use Example;
$ret = Example::test("foo", 2);
print $ret . "\n";
```

To expose functions from a C application to an embedded Perl interpreter, we have to create an extension that wraps these functions using XS, link the extension to the application and load it. The loading is performed passing to the interpreter during its initialization

[18] Vim (`http://www.vim.org`), Gimp (`http://search.cpan.org/search?mode=dist&query=gimp`) and Gaim (`http://gaim.sourceforge.net`) are some applications that implement Perl plugins through XS extensions. In the Perl plugin for Xchat (`http://www.xchat.org`), there are no `.xs` files, but `.c` sources include functions declared with undocumented APIs and Perl code equivalent to the `.pm` file generated by xsubpp is declared as a C string evaluated with `eval_pv`, leading us to assume that the plugin was implemented as an XS extension and later converted to a single C source file.

[19] This behavior is described in the documentation as “an unfixable bug (fixing it would break lots of existing CPAN modules)” [35].

a C function containing `newXS` calls. The `ExtUtils::Embed` Perl module has a utility routine called `xsinit` which generates C code for this function. In practice, generating code with `xsinit` is the best approach, since the initialization protocol depends on undocumented routines (the example initialization function included in Perl's documentation [23] is out-of-date).

3.4.6 Comparison

Python and Ruby offer to the programmer various options for C function signatures that are recognized by the API, which is practical, given that this way one can choose different C representations for the input parameters (collected in an array, obtained one by one, etc.) according to their use in the function. Lua offers only one possible signature for C functions to be registered in its virtual machine, but this is appropriate for the stack model adopted by its API.

In Java, function signatures are created through the `javah` tool – due to its static type system, types of input parameters passed by Java are converted automatically by the JNI, which is very convenient as it avoids explicit operations for conversion and type checking in the function. Because of their dynamic type systems, the other languages offer specific API functions for performing these checks. Perl function signatures are created only through the XS tool, but differently from Java they are not exposed to the programmer. This brings the inconvenience that the programmer needs to pre-process C code as an XS extension even when they are embedding Perl in an application.

Registration of functions in Ruby and Lua is simple. In Lua, in particular, it is an assignment, not different from any other object. In Python, there are features for batch registering, using `PyMethodDef` arrays (Lua offers a similar feature with `luaL_register`), but there is no simple way to register a single function. Both in Java and Perl, function registration is done implicitly, and there are no API functions for registering new C functions at runtime in either of them.

Chapter 4

Case study: LibScript

In previous chapters, we discussed the main issues involving language interfaces for C and the way these issues are handled by the languages covered by this study. In this chapter, we will make a comparison between their APIs through a concrete example, in order to put implementations on each of those languages side by side. The example consists of a generic scripting library called LibScript, and a series of plugins that interface different scripting languages.

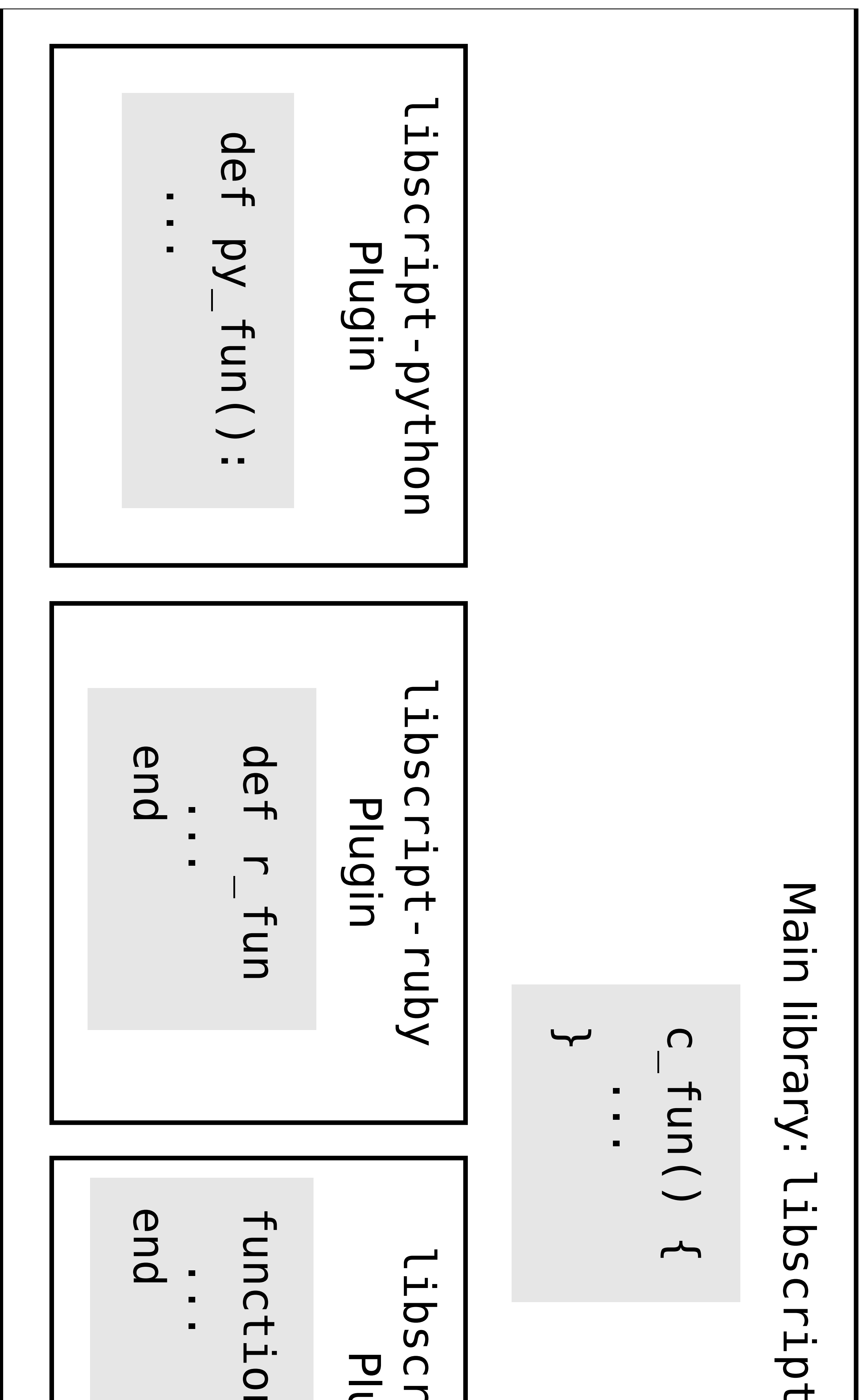
4.1 LibScript

LibScript is a library designed to provide extensibility to applications through scripting in a language-independent way. It is based on a plugin architecture, in order to decouple the application from runtime environments provided by the various languages. The main library provides a language-independent scripting API, allowing an application to register its functions and launch scripts that use these functions. This library then invokes a plugin for the appropriate language to run the script (for example, LibScript-Python for Python code). This way, application developers allow their users to employ different scripting languages without adding all of them as program dependencies.

The main library provides features for registering C functions from the application and for calling these functions from the plugins (allowing scripts to access these functions), besides functions for transferring data between the application and plugins. It is also possible to invoke functions implemented in virtual machines embedded in plugins, enabling scripts written in different languages to interact with each other.

4.1.1 Architecture of LibScript

LibScript is composed of a main dynamic library, `libscript`, and plugins for different languages (Figure 4.1). The main library is linked to an application, and exposes to it a language-independent scripting API which allows running files, strings of code and invoking functions. This library is a thin layers which forwards these operations to plugins, which


libscript-python
Plugin
def py_fun():
...
libscript-ruby
Plugin
def r_fun
...
end
...
end
c_fun() {
...
}

are auxiliary dynamic libraries, loaded at runtime by the main library. These plugins embed the scripting languages' runtime environments.

The application can register C functions in the main library (illustrated by the `c_fun` function in the figure) and ask it to run scripts which register functions in different languages. However, the application does not interact directly with plugins. When the main library receives code to be executed in a given language, it loads the appropriate plugin (in case it was not already loaded) and forwards the code. The plugin will run the script in its virtual machine, which may register in it new functions (illustrated by functions `py_fun`, `r_fun`, `l_fun` and `pl_fun` in the figure).

The main library decides which plugin to load through an identifier which specifies which is the language of the code to be executed. This identifier can be obtained from the filename extension of a script, through the "`#!`" identification line at the beginning of the script[1] or even passed explicitly by the application.

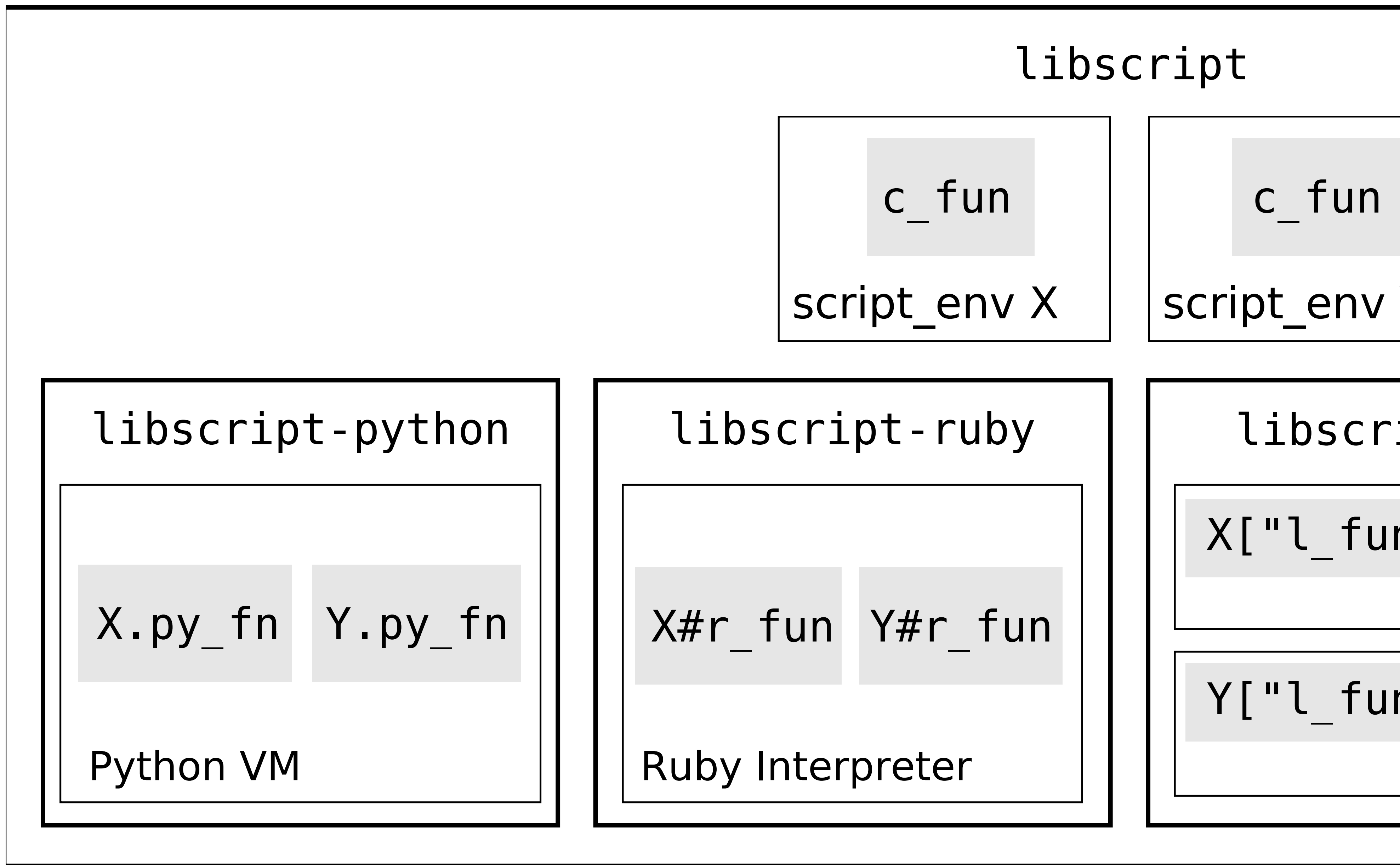
Functions are registered in LibScript in a virtual environment. An application can create one or more environments in the main library, identifying them through a name. A virtual environment has, in each plugin, a language-specific data structure (class, module, etc.) which will represent it. In the example in Figure 4.2 we have two virtual environments created by the application in the main library, `X` and `Y`. In each of these environments, the application registered a C function with the name `c_fun` (which may or may not correspond to the same C function). Scripts were executed in these environments, which prompted the loading of plugins. In the example, these scripts registered some functions (`X.py_fn`, `Y.py_fn`, `X#r_fun`, etc.).

Apart from the function for creating a virtual environment, all functions in the LibScript API receive as an argument a virtual environment they should operate on. This indicates in which C structure should be stored error messages and return values. For languages that allow multiple independent runtime states, like Lua and Perl, this also indicates in which state the script should run.

When a script declares a function in a virtual environment, this function becomes accessible through the LibScript API. For example, in the Lua plugin, virtual environments are represented as tables named after the environment; once a Ruby method `r` is declared in class `X`, this function becomes callable by C (using the LibScript API) or by other plugins. So, for example, even though the Lua table that implements virtual environment `X` contains only function `l_fun`, Lua scripts can invoke other functions through the virtual environment, like `X.c_fun` and `X.r_fun`. These calls will be handled by the main library and resolved by itself, in the case of C functions such as `X.c_fun`, or forwarded to the appropriate plugin, as in the case of `X.r_fun`, performing the call in the Ruby plugin and forwarding return values back to the Lua plugin. The main library finds the function do be executed consulting each plugin, as it will be explained in Section 4.1.3.

When implementing plugins, we used features offered by these languages to handle

[1]The "`#!`" line is used only to detect which language the script is written in. For example, a line with `#!/usr/bin/perl -w` will indicate that `libscript-perl` should be loaded, but the Perl interpreter in `/usr/bin` is not used and the `-w` flag is not considered.


libscript
c_fun
script_env X
c_fun
script_env
libscript-python
X.py_fn
Y.py_fn
Python VM
libscript-ruby
X#r_fun
Y#r_fun
Ruby Interpreter

accesses to missing elements in their structures, capturing these accesses and forwarding them to the main library. These features will be discussed in Section 4.2.4.

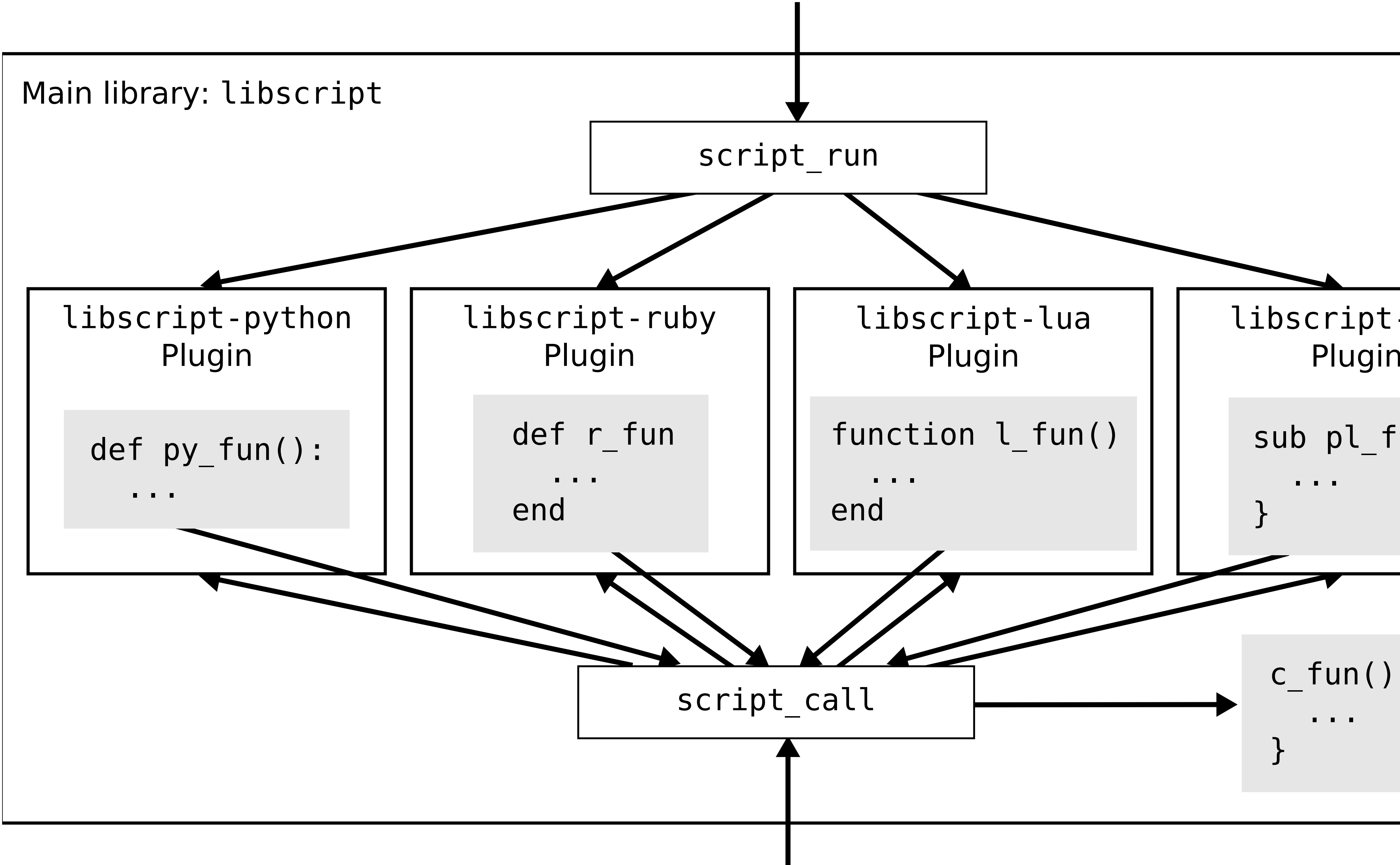
4.1.2 Main library API

The API provided by LibScript isolates the application from the different APIs provided by scripting languages. It is not only a matter of adding a layer of indirection between calls, which would be appropriate only for features that are common to all of them, such as initialization and function calls. The main issue here are the various features that are particular to each language. An unpractical approach would be to define the API as the union of the feature sets of every supported language (such as offering features for sequence handling to map this Python feature, table handling features for Lua, and so on). This path would bring several problems: the API would be complex and would probably have to be extended as each new language is introduced; even for mappings that could apparently be reused (for example, mapping Python hashes and Lua tables to a single API of associative arrays) there is the problem of subtle semantic variations between the implementations of those features in the various languages. Besides, application bindings could offer functionality that is available for a single language, going against the model of language independence proposed by LibScript.

Another approach is to, instead of exposing the language API to the application, expose only a function API of the application to the language and keep its data structures and features restricted to the domain where it is invoked. The application interacts with the virtual machine sending strings of code to be executed and obtains results back when the script passes parameters when calling application functions. This approach is proposed in [41] and uses what, for example, Python calls a "very high level layer" [44, 45]. Primitives for running strings of code are a basic feature in scripting languages – `luaL_loadstring` in Lua, `PyRun_SimpleString` in Python, `rb_eval_string` in Ruby , `perl_eval_sv` in Perl [23].

LibScript adopts this minimalist approach for its API: no specific operations for data structure manipulation are offered, only for *executing strings* – `script_run` (and the convenience function `script_run_file`, which reads a file and sends it to script_run) – and *function calls* with basic types (numbers and strings) – `script_call`. Operations on more complex data of language-specific types, when needed, can be encapsulated in functions implemented in the scripting language. One can also reference language objects from C storing it in structures of the scripting language and returning to C numerical indices of these structures, acting as high-level handles for these objects.

Figure 4.3 depicts the interaction between the application, the main library and plugins with regard to these two fundamental operations, represented by functions `script_run` and `script_call`. For executing strings, the main library receives input from the application and forwards code to be executed to the appropriate plugin. When using `script_run`, two strings are passed, one identifying the language and another containing code; for


Main library: libscript
script_run
libscript-python
Plugin
def py_fun():
...
libscript-ruby
Plugin
def r_fun
...
end
libscript-lua
Plugin
function l_fun()
...
end
libscript-
Plugin
sub pl_f
...
}
script_call
c_fun()
...
}

script_run_file, a filename[2]. The following example declares a virtual environment, registers a C function called hello and invokes it from Lua code:

```
script_env* env = script_init("example");
script_new_function(env, hello, "hello");
script_run(env, "lua", "example.hello()");
```

The virtual environment is declared with the script_init function. It receives the name that will identify the environment and returns an identifier of the script_env type, which is an opaque pointer that represents a virtual environment. The C function is registered using script_new_function, which receives as arguments the environment, a function to be registered and the name that the function will have in the virtual environment. In Lua code, the function is accessed as element hello (the registered name of the function) of global table example (name of the virtual environment).

For function calls, the application should pass input parameters (how this is done will be discussed later on), and call script_call, indicating the name of a function registered in the virtual environment. This same function script_call is used by plugins when they wish to invoke functions from the virtual environment registered in C or implemented by other plugins.

For this reason, we adopted a generic API for data transfer, to be used both in input and output of data, both in the communication between the application and the main library and between the main library and plugins. We chose an approach similar to those employed by Lua (Section 3.3.4) and Perl (Section 3.3.5) for sending data when passing parameters and obtaining return values, using an internal buffer as a transfer area. Differently from those languages, however, we pass indices to parameters explicitly instead of implementing a stack discipline. Functions script_{get,put}_{string,int,double,bool} are used for input and output of values. Functions script_put_* store values in the internal buffer and script_get_* remove them. A call to a function called test passing a string and an integer as parameters and obtaining an integer as a result is performed like this:

```
script_put_string(env, 0, "foo");   /* index 0: "foo" */
script_put_int(env, 1, 2);          /* index 1: 2 */
script_call(env, "test");
result = script_get_int(env, 0);    /* return index 0 */
```

Function calls are offered as a primitive operation because they allow a minimum degree of language-independent interoperability. Two goals are met this way. The first one is that this way C programs embedding LibScript can access the functionality of loaded scripts without having to include in their source code strings of code written in some specific scripting language, for example, inserting in their code an invocation to a callback function to be defined through a script. Notice that in the above example, the language in which the test function is implemented is not specified. If the call was made by running

[2] For code executed with script_run_file, the language is automatically detected as discussed in the previous section.

a string of code, this would tie the application to at least one scripting language. Using script_run_file and script_call, one can implement an extensible application without specifying explicitly the scripting language to be used with it. The second goal is allowing the plugins themselves to invoke functions defined in other plugins. We would have to provide to plugins an invocation function anyway, to allow them to invoke C functions registered into LibScript. Making this invocation function generic enough so that it can also invoke functions implemented in the plugins themselves does not make, thus, the main library API any more complex.

The LibScript buffer was designed to be used only as a temporary transfer area between the main library and plugins, and not as a general facility for data storage and manipulation. Therefore, its API is focused on sequential insertion and removal of elements. For example, the insertion of an element at position 0 automatically empties the buffer, avoiding in most cases the need to use the `script_reset_buffer` function, which performs this operation explicitly.

C functions registered with `script_new_function` must receive the virtual environment as a parameter and return an error code. Functions `script_get_*` and `script_put_*` are used to receive arguments and return values when implementing functions that can be called through LibScript, the same way they are used to pass arguments and obtain return values and perform calls with `script_call`.

```
script_err test_lua(script_env* env) {
   char* foo = script_get_string(env, 0);  /* Input, index 0: string */
   int n = script_get_int(env, 1);         /* Input, index 1: integer */
   /* Escape the function with an error if any script_get* failed */
   SCRIPT_CHECK_INPUTS(env);
   printf("Received %s and %ld \n", foo, n);
   free(foo);
   script_put_int(env, 0, 42);             /* Output, index 0: integer */
   return SCRIPT_OK;
}
```

In LibScript, strings returned by script_get_string belong to the caller, being their responsibility to deallocate its memory, unlike what happens in similar function of the APIs of languages discussed in this work. Such decision was made due to the temporary nature of the LibScript buffer: returning to the caller a pointer to a string whose validity would be ensured only until the next API call would be counter-intuitive, and in practice would force the programmer to copy strings frequently.

4.1.3 Plugins API

A plugin embedding a scripting language must implement four operations: `init`, `run`, `call` and `done`. The main library expects that the dynamic library implementing a plugin for a language will expose four functions, named as `script_plugin_`*operation*`_`*language*.

The `script_plugin_init_`*language* function is responsible for initializing a plugin, and is called by the `script_init` function of the main library. When initializing a plugin,

the main library passes to `script_plugin_init_`*language* a `script_env` pointer and receives a `script_plugin_state`, which is an opaque type which is always passed back to the plugin in subsequent calls. Each plugin defines its internal representation for `script_plugin_state`. Typical the virtual machine state and the LibScript virtual environment should be stored so that they can be later accessible through this handle. In Section 4.2.1 we will discuss how each plugin represents the environment and its internal state in `script_plugin_state`.

The `script_plugin_run_`*language* function is invoked by `script_run`. It receives a string containing code in the scripting language, executes this code in the virtual machine and returns a status code indicating success or the occurrence of compile or runtime errors. In case of errors, plugins should capture exceptions raised by the virtual machine and return the constant `SCRIPT_ERRLANGRUN`. If it is possible to obtain from the language an error message, it can be propagated using the `script_set_error_message` function from the main library. The message can be later consulted by the application using the `script_error_message` function.

The function `script_plugin_call_`*language* is used by `script_call`, and is responsible for performing calls to functions implemented in the language embedded by the plugin. If the function was defined in the plugin, that is, if a function with the given name was registered in the data structure that describes the environment within the virtual machine, it will be executed, and success or failure will be reported like it happens with `script_plugin_run_`*language*. If the requested function was not defined in this virtual machine, `script_plugin_call_`*language* must return the constant `SCRIPT_ERRFNUNDEF`. Input arguments and return values are passed through the argument buffer, using the same functions from the main library that are used to transfer data between the application and the main library, `script_get_*` and `script_put_*`.

The implementation of `script_call` in the main library makes use of this behavior of plugins for invoking functions in a language-independent way. Initially, it tries to find a requested function in the list of registered C functions. If there is no C function in the virtual environment with that name, `script_call` tries to locate the function in loaded plugins, calling `script_plugin_call_`*language* on each plugin and trying the next one each time it receives `SCRIPT_ERRFNUNDEF`.

Finally, the `script_plugin_done_`*language* function is called by `script_done` when a virtual environment is terminated. Depending on the internal representation used in the plugin, the termination of a state may or may not imply in the termination of the virtual machine. Preferably, this function should remove the data structure that describe the virtual environment, but as we will see in Section 4.2.2, this is not always possible.

4.2 Implementation of plugins

In this section we will discuss the main aspects involved in the implementation of the plugins developed in this case study. We implemented plugins for Python, Ruby, Lua and Perl. We will present here how the representation of virtual states is made in each

plugin (Section 4.2.1), issues involving state termination (Section 4.2.2), passing parameters between the main library and plugins (Section 4.2.3), how function calls from scripts are handled by plugins (Section 4.2.4) and error capturing (Section 4.2.5).

4.2.1 Representation of states

The design of LibScript allows plugins to have multiple independent states of execution. Ideally these states would be totally isolated from each other, like for example with different virtual machine instances. However, languages offer different degrees of isolation between independent states. Lua and Perl allow multiple isolated instances of their runtime environments in a simple way, since their API calls include a state identifier[3]. Language that keep state in a static manner, like Python and Ruby, don't allow working with multiple isolated states easily[4]. For languages that do not allow multiple virtual machine instances, we can only define separate namespaces for LibScript virtual environments, which share a single global state of execution within the plugin. We term the representation of a state of execution relative to a LibScript virtual environment within a plugin a *virtual state*, which may or may not correspond to an isolated state of execution.

As discussed in the previous section, the `script_plugin_init_`*language* function returns a `script_plugin_state` to the main library, which is the opaque representation of its virtual state. The contents of this representation vary from language to language, but the basic principle is that two data should be available from this value: a reference to the LibScript virtual environment, received as an argument by `script_plugin_init_`*language*, so that the plugin can make calls to the main library, and an identifier that allows the plugin to access the data structure that represents for the language the namespace of LibScript-accessible functions. In the Lua plugin, this structure is a table; in Python, a module; in Ruby, a class; in Perl, a package.

In LibScript-Lua, states are implemented as `lua_State`s (Section 3.4.4). This way, scripts executed in an environment are fully isolated from other environments. For example, the modification of the value of a global variable in an environment does not affect the other ones. In fact, `script_plugin_state` as returned by the Lua plugin is simply a `lua_State` converted with a cast. The pointer to the LibScript environment is stored in Lua in the registry, as follows:

```
lua_pushstring(L, "LibScript.env");     /* Pushes the index */
lua_pushlightuserdata(L, env);          /* Pushes the LibScript environment */
lua_settable(L, LUA_REGISTRYINDEX);     /* registry["LibScript.env"] = env */
```

[3] The feature of multiple independent states is optional in Perl, and can be selected during compilation of the interpreter's library.

[4] Python's threading model offers a way for alternating between virtual machine states obtaining `PyThreadState` objects through the `Py_NewInterpreter` call, but this can cause problems when extension modules written in C use global static variables or when modules manipulate their own dictionary, which is shared between states. The documentation states, since 1999, that "This is a hard-to-fix bug that will be addressed in a future release." [43, 46]

The plugin creates in this lua_State a table representing the virtual environment for Lua scripts. This table is stored in the lua_State as a global variable with the name of the virtual environment.

In LibScript-Perl states are isolated like in Lua. Each state created initializes a new instance of PerlInterpreter. In this interpreter, a package is created, which will be the visible representation of the environment in Perl code. The script_plugin_state type is then a typedef for PerlInterpreter*.

As discussed in Section 3.4.5, the implementation of C functions exported to a Perl interpreter is made writing an extension module using the XS pre-processor, and the way to obtain communication in the Perl→C direction in an embedded virtual machine is linking an extension module to the virtual machine. Thus, part of LibScript-Perl is implemented as an XS module, exposed to the embedded virtual machine as the LibScript Perl package. During the initialization of a virtual state, the pointer to the LibScript virtual environment is stored in this package, in the $LibScript::env variable. The package that represents the virtual environment is created by the script_plugin_init_perl function, executing the string of code "package <*environment*>;" using the function Perl_eval_pv.

As Python does not feature facilities for launching multiple fully isolated virtual machines, the Python plugin implements virtual states only as separate modules, sharing a single global state. During the initialization of a state, a Python module with the name of the environment is created. The following excerpt of the script_plugin_init_python function shows the sequence where the module is created and imported:

```
/* Get the name of the environment */
char* namespace = script_namespace(env);
/* Creates the module. The first argument is the module name,
   the second one the module's method list, which will start empty */
PyObject* module = Py_InitModule3(namespace, NULL);
/* Get the globals dictionary */
PyObject* globals = PyModule_GetDict(PyImport_AddModule("__builtin__"));
/* Assigns the module to the global with its name */
PyDict_SetItemString(globals, namespace, module);
```

The script_plugin_state type is a typedef for PyObject*. The object returned by the initialization function is the elements dictionary of the module, obtained with PyModule_GetDict(module). In this dictionary, we store the pointer to the virtual environment as the private attribute __env.

Similarly, in Ruby virtual states are implemented as classes that share a single global state, since Ruby does not allow multiple isolated runtime environments either. In the initialization function, script_plugin_init_ruby, a class with the name of the virtual environment is created using the rb_define_class function. The pointer to the virtual environment is stored in a class constant as a number. The VALUE corresponding to the new class is returned as the script_plugin_state.

```
VALUE state;
/* ... (initialization of the interpreter omitted) ... */
```

```
/* class_name is the name of the virtual environment,
   with its initial converted to uppercase,
   respecting Ruby's class naming convention */
state = rb_define_class(class_name, rb_cObject);
/* This assumes void* fits in a long */
rb_const_set(state, rb_intern("@@LibScriptEnv"), INT2NUM((long)env));
/* ... */
return (script_plugin_state) state;
```

4.2.2 Termination of states

Because Lua and Perl represent states in an independent way, state termination in these plugins is simple: the language structure which wraps the complete runtime environment is terminated. The implementation of the finalization function in the Lua plugin is the following:

```
void script_plugin_done_lua(script_plugin_state state) {
    /* In Lua, a state is a lua_State */
    lua_State* L = (lua_State*) state;
    /* Terminates the state. Does not affect other environments */
    lua_close(L);
}
```

In Perl, the process, although a tad more elaborate, is essentially the same:

```
void script_plugin_done_perl(script_perl_state* state) {
    /* Some macros assume the interpreter pointer is called my_perl */
    PerlInterpreter* my_perl = (PerlInterpreter*) state;
    /* Some operations act on the "current state",
       so the PERL_SET_CONTEXT macro must be used to
       switch the active interpreter */
    PERL_SET_CONTEXT(my_perl);
    /* This flag must be activated so that the cleaning
       of an environment is complete, which is needed
       when there may be more than one active interpreter */
    PL_perl_destruct_level = 1;
    /* Terminating the interpreter */
    perl_destruct(my_perl);
    perl_free(my_perl);
}
```

In Python and Ruby, plugins need to keep track of the number of active states to deallocate the virtual machine only when it reaches zero. Besides, both in Ruby and Python there are no features in their APIs (or in the languages, actually) to remove, respectively, classes and modules. In Ruby, we could assign `nil` to the constant that represents the class which describes the virtual environment, but after that it is not possible to define a new class in its place: both `rb_define_class` through C and `class` <*Name*> through Ruby indicating that the value was already defined with another type. Since Ruby features

open classes, a `class` *<Name>* construct for an already existing *<Name>* is understood as a resumption of the class description, and not as the redefinition of *<Name>*. Python, on its turn, does not provide API features for unloading modules, but allows assigning `None` to the global referring the module. The module can then be imported again, but the same instance, stored internally by Python, will be returned. The following interactive command-line session allows us to observe this behavior, which happens both directly in Python as well as through the C API:

```
>>> import sys
>>> sys.foo = "hello"
>>> sys.foo
'hello'
>>> sys = None
>>> import sys
>>> sys.foo
'hello'
```

Thus, data structures referring to LibScript states are not terminated in LibScript-Python and LibScript-Ruby. This is the implementation of the termination routine in the Ruby plugin:

```
void script_plugin_done_ruby(script_ruby_state state) {
   /* Decrements the state counter, a static global variable */
   script_ruby_state_count--;
   /* Terminates the interpreter if this is the last state */
   if (script_ruby_state_count == 0)
      ruby_finalize();
}
```

The Python implementation is basically the same:

```
void script_plugin_done_python(script_python_state state) {
   script_python_state_count--;
   if (script_python_state_count == 0)
      Py_Finalize();
}
```

4.2.3 Passing arguments

Data transfer between the main library and plugins is concentrated in two operations: one to pass the contents of the LibScript argument buffer to the virtual machine data space and another to perform the inverse operation. The first one is used for passing input parameters when scripting language functions are called from C and for obtaining return values when the scripting language makes calls that are handled by C. The second operation, complementarily, is used for filling return values when C calls the scripting language and for input parameters when a scripting language call is handled by C code.

In the implementation of LibScript-Lua, the `script_lua_stack_to_buffer` function converts the contents of the Lua stack to the LibScript argument buffer. The plugin

function responsible for invoking Lua functions from C, `script_plugin_call_lua`, uses `script_lua_stack_to_buffer` to store in the LibScript buffer return values of the invoked Lua function, since these are returned in Lua's virtual stack. When Lua code calls functions implemented in C or some other plugin, `script_lua_stack_to_buffer` is used to convert the function's input parameters, also retrieved through the virtual stack. Below, we see the implementation of this function:

```
static void script_lua_stack_to_buffer(script_env* env, lua_State *L) {
   int nargs; int i;
   nargs = lua_gettop(L);    /* Number of elements in the Lua stack */
   script_reset_buffer(env); /* Empties the LibScript buffer */
   for (i = 1; i <= nargs; i++) {
      /* Checks the Lua type of element in stack position i */
      /* and for each type, convert the element and store in the buffer */
      switch(lua_type(L, i)) {
      case LUA_TNUMBER:
         script_put_double(env, i-1, lua_tonumber(L, i)); break;
      case LUA_TSTRING:
         script_put_string(env, i-1, lua_tostring(L, i)); break;
      case LUA_TBOOLEAN:
         script_put_bool(env, i-1, lua_toboolean(L, i)); break;
      default:
         /* Unhandled types are replaced by zero */
         script_put_double(env, i-1, 0);
      }
   }
}
```

We assume in LibScript C-format strings: the `script_put_string` function copies the given string up to the first null character. Thus, when obtaining strings from languages that allow arbitrary contents, these will be truncated in case they contain nulls. Because of that, in the Lua plugin we use directly the `lua_tostring` function, and not the more general `lua_tolstring` (which also returns the buffer size). This design decision coincides with the goal explained earlier of restricting the main library API to features available in all languages.

Values of unknown types are replaced by zero, which keeps the position of the other elements in the arguments list unaltered. We chose not to signal error is this situation to avoid raising exceptions here, which would complicate our presentation. Capture and propagation of errors will be seen in Section 4.2.5.

The second data transfer function from LibScript-Lua, `script_lua_buffer_to_stack`, obtains values from the LibScript buffer and inserts them into Lua's virtual stack. This function is used to pass input arguments to Lua in `script_plugin_call_lua` and to pass to Lua return values from the `script_call` function, which is internally invoked by the plugin when Lua invokes a C function.

```
static int script_lua_buffer_to_stack(script_env* env, lua_State *L) {
   int i; char* s;
```

```
   int len = script_buffer_len(env); /* Number of elements in the buffer */
   for (i = 0; i < len; i++) {
      /* Checks the type of element in buffer position i */
      /* and for each type, obtains it and inserts in in Lua's stack */
      type = script_get_type(env, i);
      switch (type) {
      case SCRIPT_DOUBLE:
         lua_pushnumber(L, script_get_double(env, i)); break;
      case SCRIPT_STRING:
         s = script_get_string(env, i); /* The string belongs to the caller */
         lua_pushstring(L, s);
         free(s); /* Frees the string, since Lua stores its own copy */
         break;
      case SCRIPT_BOOL:
         lua_pushboolean(L, script_get_bool(env, i)); break;
      }
   }
   return len;
}
```

In LibScript-Python, it was not possible to concentrate data transfer operations in two functions only. Each operation had to be split in two parts. The conversion of data sent from Python to the LibScript buffer was split into `script_python_put_object` and `script_python_tuple_to_buffer`. The first function converts a single value Python and inserts it in the requested buffer position:

```
static void script_python_put_object(script_env* env, int i, PyObject* o) {
   if (PyString_Check(o))
      script_put_string(env, i, PyString_AS_STRING(o));
   else if (PyInt_Check(o))
      script_put_int(env, i, PyInt_AS_LONG(o));
   else if (PyLong_Check(o))
      script_put_double(env, i, PyLong_AsDouble(o));
   else if (PyFloat_Check(o))
      script_put_double(env, i, PyFloat_AS_DOUBLE(o));
   else if (PyBool_Check(o))
      script_put_bool(env, i, o == Py_True ? 1 : 0);
   else
      script_put_int(env, i, 0);
}
```

It is important to note that Python types `PyInt` and `PyLong` do not correspond to C types `int` and `long`: `PyInt` is the integer type corresponding to the machine word size (analogous to `int`), but a `PyLong` is an arbitrary-precision integer. In LibScript, we represent `PyLong`s as `double`s. The LibScript API offers the `script_put_int` function as a convenience, but internally, as it happens for example in Lua, all numbers are stored as `double`s.

The second function, `script_python_tuple_to_buffer`, inserts elements of a tuple in the buffer:

```
static void script_python_tuple_to_buffer(script_env* env, PyObject* tuple) {
   int i;
   int len = PyTuple_GET_SIZE(tuple); /* Number of elements in the tuple */
   script_reset_buffer(env);          /* Empties the LibScript buffer */
   for (i = 0; i < len; i++) {
      PyObject* o = PyTuple_GET_ITEM(tuple, i); /* Get a tuple element */
      script_python_put_object(env, i, o);      /* Insert it in the buffer */
   }
}
```

The inverse operation, for transferring data from the LibScript buffer to Python, is also implemented in two functions, one handling objects individually and another handling tuples. The `script_get_object` function converts a buffer element to an equivalent `PyObject`:

```
static PyObject* script_python_get_object(script_env* env, int i) {
   PyObject* ret; char* s;
   switch (script_get_type(env, i)) {
   case SCRIPT_DOUBLE:
      return PyFloat_FromDouble(script_get_double(env, i));
   case SCRIPT_STRING:
      s = script_get_string(env, i);
      PyObject* ret = PyString_FromString(s);
      free(s);
      return ret;
   case SCRIPT_BOOL:
      return PyBool_FromLong(script_get_bool(env, i));
   }
}
```

The `script_python_buffer_to_tuple` function generates a tuple containing every element of the LibScript buffer:

```
static PyObject* script_python_buffer_to_tuple(script_env* env) {
   int i;
   int len = script_buffer_len(env);
   PyObject* ret = PyTuple_New(len);
   for(i = 0; i < len; i++) {
      PyObject* o = script_python_get_object(env, i);
      PyTuple_SetItem(ret, i, o);
   }
   return ret;
}
```

This way, these two pairs of functions perform functions equivalent to `script_lua_stack_to_buffer` and `script_lua_buffer_to_stack` do in the Lua script. They were separated in two parts because of the model for return values in Python functions: for the case of multiple return values, they are returned as a tuple; for single values, they are passed directly. This is made evident in the following excerpt of the `script_plugin_call_python` function:

```
PyObject *ret, *args;
/* ... */
args = script_python_buffer_to_tuple(env);  /* Get input parameters */
ret = PyEval_CallObject(func, args);        /* Call a Python function */
/* ... */
if (ret == Py_None)                         /* If the function returned no values */
   script_reset_buffer(env);                /* Just empty the LibScript buffer */
else if (PyTuple_Check(ret))                /* If a tuple was returned */
   script_python_tuple_to_buffer(env, ret); /* Insert its elements in the buffer */
else                                        /* If another type was returned */
   script_python_put_object(env, 0, ret);   /* Insert it as the only element */
```

In the plugin handler for calls to external functions, communication in the opposite direction follows a similar logic:

```
script_python_tuple_to_buffer(env, args);    /* Get input parameters */
err = script_call(env, fn_name);             /* Call a function through LibScript */
/* ... */
switch(script_buffer_len(env)) {
case 0:                                       /* If the function returned no values */
   Py_RETURN_NONE;                            /* Return the Python value 'None' */
case 1:                                       /* If a single value was returned */
   return script_python_get_object(env, 0);   /* Convert it and return it */
default:                                      /* If more than one value was returned */
   return script_python_buffer_to_tuple(env); /* Return them in a tuple */
}
```

Like in Python, Ruby functions return multiple values by wrapping them in an aggregate type. This way, data transfer operations in LibScript-Ruby are also split into pairs of functions, one converting a buffer value and another operating on a Ruby array. The analogous function to `script_python_put_object` is `script_ruby_put_value`:

```
static void script_ruby_put_value(script_env* env, int i, VALUE arg) {
   switch (TYPE(arg)) {
   case T_FLOAT:
   case T_FIXNUM:
   case T_BIGNUM:
      script_put_double(env, i, NUM2DBL(arg)); break;
   case T_STRING:
      script_put_string(env, i, StringValuePtr(arg)); break;
   case T_TRUE:
      script_put_bool(env, i, 1); break;
   case T_FALSE:
      script_put_bool(env, i, 0); break;
   default:
      script_put_int(env, i, 0);
   }
}
```

Here, some problems of the Ruby API become apparent. Besides the inconsistency in the naming of object conversion function names, the meaning of the value returned by

the TYPE macro can only be understood through the internal representation of VALUEs in Ruby's implementation, and not through the type hierarchy of language objects. Classes that has special handling in the internal structure of VALUEs have constants associated to them, such as T_FLOAT and T_STRING; the rest are identified only as T_OBJECTs. The use of T_TRUE and T_FALSE may lead to think that some specific values also return special results for TYPE. Indeed, these values are defined as VALUEs that do not correspond to Ruby heap indices and are handled especially by the implementation. From the point of view of Ruby code, however, this classification of values true and false as separate types in the C API is justified by defining them as singletons of classes TrueClass and FalseClass, an approach probably influenced by Smalltalk. Unlike Smalltalk, though, where True and False are subclasses of Boolean, in Ruby TrueClass and FalseClass are direct subclasses of Object. This brings the inconvenience that checking if a type matches a boolean value two tests are needed.

Like LibScript-Python has a function to store in the buffer elements from a tuple, LibScript-Ruby has a function to store elements of an array:

```
static void script_ruby_array_to_buffer(script_env* env, VALUE array) {
   int i;
   int len = RARRAY(array)->len;
   script_reset_buffer(env);
   for (i = 0; i < len; i++) {
      VALUE o = rb_ary_entry(array, i);
      script_ruby_put_value(env, i, o);
   }
}
```

Ruby does not have a function in its C API to return the size of an array; instead, the internal structure of VALUE is exposed through the RARRAY macro (which just wraps a cast).

Operations for converting values from the LibScript buffer to Ruby are also similar to those implemented in the Python plugin. Again, where in Python there is a function for manipulating tuples, we have in Ruby a function that operates on arrays:

```
static VALUE script_ruby_get_value(script_env* env, int i) {
   VALUE ret; char* s;
   switch (script_get_type(env, i)) {
   case SCRIPT_DOUBLE:
      return rb_float_new(script_get_double(env, i));
   case SCRIPT_STRING:
      s = script_get_string(env, i);
      ret = rb_str_new2(s);
      free(s);
      return ret;
   case SCRIPT_BOOL:
      return script_get_bool(env, i) ? Qtrue : Qfalse;
   }
}
```

```
static VALUE script_ruby_buffer_to_array(script_env* env) {
   int i;
   int len = script_buffer_len(env);
   VALUE ret = rb_ary_new2(len);
   for (i = 0; i < len; i++) {
      VALUE o = script_ruby_get_value(env, i);
      rb_ary_store(ret, i, o);
   }
   return ret;
}
```

In a similar way to the Python plugin, the implementation of Ruby function calls from LibScript uses the `script_ruby_buffer_to_array` to convert input parameters and functions `script_ruby_put_value` or `script_ruby_array_to_buffer` to convert the return value, depending on the number of values returned (or, more precisely, if the function has returned an array or not). In calls to LibScript functions from Ruby, input parameters are converted with `script_ruby_array_to_buffer` and return values are converted with `script_ruby_get_value` or `script_ruby_buffer_to_array`.

In the Perl plugin, we have three functions: data transfer from the stack to the LibScript buffer was implemented in a single function like in Lua, but transfer in the opposite direction had to be split in two functions, like in Python and Ruby. This asymmetry comes from the fact that handling of return values is wrapped by the XS pre-processor through the `RETVAL` variable; so, in this situation we cannot manipulate the stack directly, but only pass `SV`s as output values.

Transferring data from the Perl stack to the LibScript buffer is reasonably simple:

```
void script_perl_stack_to_buffer(pTHX_ int ax, script_env* env,
                                 int count, int offset) {
   int i;
   script_reset_buffer(env);
   for (i = 0; i < count; i++) {
      /* Obtain a pointer to a SV from Perl's stack */
      SV* o = ST(offset+i);
      if (SvIOK(o))
         script_put_int(env, i, SvIV(o));
      else if (SvNOK(o))
         script_put_double(env, i, SvNV(o));
      else if (SvPOK(o))
         script_put_string(env, i, SvPV_nolen(o));
      else
         script_put_int(env, i, 0);
   }
}
```

This functions' input parameters deserve mention. Initially, we have the `pTHX_` macro. This macro was added to the API when Perl started allowing multiple simultaneous interpreters per process: API functions were transformed into macros that wrap this first argument. For example, `eval_sv` can be called as `Perl_eval_sv`, passing the `aTHX_` macro

as an initial parameter. In general the use of these macros remains implicit, but when writing functions that use the Perl API it becomes necessary to use the pTHX_ macro in declarations[5], to propagate interpreter state information through function calls, and `aTHX_` in calls.

Another symptom that the Perl API was designed more for internal use of the XS preprocessor than for direct manipulation shows in the second argument, `ax`. Some macros assume the existence of this value, which is not propagated by `pTHX_`, but is implicitly declared when functions are wrapped with XS. The API seems to assume that an XS function will not invoke another C function which also uses the API. We had to propagate the variable ourselves (which is mentioned in the documentation, but only as "the 'ax' variable" [30], with no explanations of its purpose).

The other two parameters, count and offset, are needed due to the different ways that the information they represent are obtained in the two contexts where this function is used. In other plugins, we can obtain the number of input elements in a uniform way (checking the number of elements in a Python tuple, for example). In Perl, in the two situations where the function is called, the number of elements to be read from the stack should be obtained in different ways, and because of that we pass it as the `count` parameter. In the routine that calls LibScript functions, implemented in the XS file, the size of the stack is obtained through a special variable, `items`. When calling Perl functions, the value of `count` is obtained as the result of the invocation function, `Perl_call_pv`.

The start position from where to obtain elements (`offset`) also varies. Inside the XS function, input parameters start from position 2, because LibScript passes the environment pointer and function name in the first two arguments. In calls to Perl functions, the value of offset is zero because, as seen in the protocol for invocation of Perl functions discussed in Section 3.3.5, the stack bottom is adjusted after the function call by the `SPAGAIN` macro.

Conversion of values from the LibScript buffer to the Perl stack is implemented in two functions, one that generates a single `SV` and another that pushes all elements:

```
SV* script_perl_get_sv(pTHX_ script_env* env, int i) {
   switch (script_get_type(env, i)) {
   case SCRIPT_DOUBLE: return newSVnv(script_get_double(env, i));
   /* 0 indicates that the size of the string should be computed by Perl */
   case SCRIPT_STRING: return newSVpv(script_get_string(env, i), 0);
   case SCRIPT_BOOL: return newSViv(script_get_bool(env, i));
   }
}

SV** script_perl_buffer_to_stack(pTHX_ SV** sp, script_env* env) {
   int i;
   int len = script_buffer_len(env);
   for (i = 0; i < len; i++) {
      XPUSHs(sv_2mortal(script_perl_get_sv(aTHX_ env, i)));
   }
```

[5]The `pTHX_` is used without a comma separating it from the following argument. When it is the only argument, one should use `pTHX`.

```
    return sp;
}
```

Again, a variable created internally by Perl had to be propagated explicitly: `sp`, the stack pointer. This variable is referenced within the `XPUSHs` macro. Besides, as `XPUSHs` can resize the stack, we need to return the updated value of `sp` back to the caller. Apart from that, generation of `SV`s, their registration as mortal variables and their insertion in the stack happens as usual, as presented in Section 3.3.5.

Like in the other plugins, the transfer of input parameters in LibScript-Perl, both for Perl functions and for functions called through LibScript, is made calling the conversion function that acts on the buffer as a whole: when calling Perl functions we use `script_perl_buffer_to_stack` and for functions called using LibScript, `script_perl_stack_to_buffer`. For handling return values for Perl functions, we were able to use directly the `script_perl_stack_to_buffer` function, not unlike it was done in LibScript-Lua. For the return value of functions called using LibScript, however, we need to deal with the special XS variable `RETVAL` and with the different call contexts of Perl. The excerpt below illustrates the handling of return values in this case:

```
err = script_call(env, function_name);
/* ... (error handling omitted) ... */
switch (GIMME_V) {
case G_SCALAR:
   /* Return the first item of the buffer */
   RETVAL = script_perl_get_sv(aTHX_ env, 0);
   break;
case G_ARRAY:
   len = script_buffer_len(env);
   /* Create an array */
   RETVAL = (SV*)newAV();
   /* Returned arrays have to be marked as mortal */
   sv_2mortal((SV*)RETVAL);
   /* Insert the contents of the buffer in the array */
   for (i = 0; i < len; i++)
      av_push((AV*)RETVAL, script_perl_get_sv(aTHX_ env, i));
   break;
case G_VOID:
   /* As the return value is discarded in void contexts, */
   /* we return the Perl constant undef */
   RETVAL = &PL_sv_undef;
   break;
}
```

4.2.4 Function calls

In LibScript plugins, functions implemented externally (in C or other plugins) are located only in the moment when they are called. The goal here, besides optimizing initialization time and memory consumption in the scripting language runtime environment

(avoiding the declaration of functions that will not be used), is to allow the location of functions declared after the environment's initialization. To allow function resolution in a dynamic way, it is necessary to capture the access to missing elements in the structure which describes the virtual environment to the plugin and forward the call to the main library using `script_call`. When comparing approaches employed in each plugin to get this behavior, we can evaluate some meta-programming features provided by each language and their availability through their APIs.

As seen in Section 4.2.1, in Lua, during plugin initialization, a table is created and stored in a global variable with the same name as the environment. Functions are inserted dynamically in this table through a metatable associated to it right after its creation in `script_plugin_init_lua`. The `__index` field of this metatable points to a C function internal to the plugin, `script_lua_make_caller`, which is then invoked always that a nonexistent element is requested in the table. The `script_lua_make_caller` function creates a C closure, which consists of another C function internal to the plugin (`script_lua_caller`) and the name of the requested function. This closure is associated to the proper entry in the environment table. This way, calls to functions implemented externally are resolved by `script_lua_caller`, which will pass them on to `script_call`.

In the Python plugin, when calling a function in the virtual environment module, the module's `__getattro` callback, defined as the internal function `script_python_get`, is called. This function searches for an entry in the module's dictionary and, if it is not found, creates an object of the `script_python_object` type, and returns it as the result of `__getattro`. This data type is declared in the plugin as a Python class, whose instances contain a pointer to the virtual environment and a C string with the name of the function they represent. These objects have their `__call` callback defined as `script_python_caller`, a function that, like `script_lua_caller`, converts its received parameters to the LibScript buffer, invokes `script_call` and converts the return values back to Python. Therefore, objects of this type are *functors*, and behave similarly to the closure defined in the Lua plugin.

Function resolution is implemented in Ruby using the `method_missing` method, which is a language-defined fallback, always called when a nonexistent method is invoked in a class. Unlike `__getattro` in Python and `__index` in Lua, which are attribute access handlers and therefore need to return an object which is called in a later step, `method_missing` handles calls directly. Hence, when invoked, `method_missing` receives the name of the requested method and the given parameters and forwards them to `script_call`.

In the Perl plugin, like in Lua and Python, there is also a C function responsible for invoking `script_call` and converting input parameters and return values. So that this function, `script_perl_caller`, can be exposed to the Perl interpreter, it is implemented in an XS module. Once the module is loaded the function is visible in Perl as `LibScript::caller`. Dynamic resolution of functions from the Perl package representing the LibScript virtual environment is done using the `AUTOLOAD` Perl function, which behaves like `method_missing` in Ruby, capturing calls to missing functions. In the plugin's initialization function, Perl code is executed to load the extension module, initialize the environment package and insert in it an `AUTOLOAD` function which will call `LibScript::caller`:

```
snprintf(code, LEN_CODE,
   "bootstrap LibScript;"   /* Initializes the extension module */
   "package %s;"            /* Declare the environment package */
   "$LibScript::env = %p;"  /* Store the environment pointer in Perl */
   "sub AUTOLOAD {"
      "our $AUTOLOAD;"
      /* Extract the method name from the qualified "package::method" name */
      "$AUTOLOAD =~ s/[^:]*:://;"
      /* Invokes caller passing the environment pointer, */
      /* the method name, and the argument array */
      "LibScript::caller(%p, $AUTOLOAD, @_);"
   "}",
   state->package, env, env);
/* Evaluate the code string:
   TRUE indicates that any errors should be reported. */
Perl_eval_pv(my_perl, code, TRUE);
```

4.2.5 Capturing errors

Plugins should capture the occurrence of runtime errors when executing strings of code and function calls. In Lua, both operations are performed using the `lua_pcall` function, which indicates errors in its return value. In case of errors, the error message is obtained from the top of Lua's virtual stack and propagated to the main library using `script_set_error_message`. In case of execution of strings of code, compilation errors are detected through the return value of `luaL_loadstring`, which loads code to be executed by `lua_pcall`.

In Python, the occurrence of errors is signalled by the return value of functions for executing strings, `PyRun_SimpleString`, and for calling functions, `PyEval_CallObject`. In case of errors, we call the `PyErr_Occurred` function which returns a Python object representing the exception. The error message is obtained converting this object to a Python string using `PyObject_Str`, and finally to a C string with `PyString_AS_STRING`.

In Perl, errors are signalled in the special variable `$@`; its contents can be checked from the C API with the `ERRSV` macro. The presence of errors is tested with `SvTRUE(ERRSV)`, and the error message can be obtained converting this variable to a C string with the `SvPV` macro.

Ruby provides a function for executing strings of code, `rb_eval_string`, and a variant that captures errors and signals them through its return value, `rb_eval_string_protect`. However, for method calls, there is no protected version of `rb_funcall`. The only function provided by the API to protect calls, `rb_protect`, does not get as a parameter a Ruby method, but a C function instead. To call Ruby methods in a protected way, we had to write a C function that wraps the call:

```
static VALUE script_ruby_pcall(VALUE args) {
   /* Extract the method name from the arguments array */
   ID fn_id = SYM2ID(rb_ary_pop(args));
   /* Extract the class from the arguments array */
```

```
    VALUE klass = rb_ary_pop(args);
    return rb_apply(klass, fn_id, args);
}
```

and then invoke it using `rb_protect`:

```
/* Insert the class in the arguments array */
rb_ary_push(args, klass);
/* Insert the method name in the arguments array */
rb_ary_push(args, ID2SYM(rb_intern(fn)));
/* Call the wrapper function */
ret = rb_protect(script_ruby_pcall, args, &error);
if (error) {
    script_reset_buffer(env);
    script_set_error_message(env, StringValuePtr(ruby_errinfo));
    ruby_errinfo = Qnil;
    return SCRIPT_ERRLANGRUN;
}
```

As `rb_protect` passes a single `VALUE` to the C function, we had to store the class, method identifier and all input parameters of the Ruby method to be invoked in a Ruby array. The occurrence of errors is signalled in a variable passed as the third parameter of rb_protect, and the error message is obtained in the global `VALUE ruby_errinfo`.

4.3 Conclusions

The case study presented here illustrated, through the implementation of plugins, the process of embedding four scripting languages interfacing a given C API. Several aspects of the interaction between C and scripting languages were covered, contemplating initialization and termination of their runtime environments, passing data and calling functions in both directions and signalling errors. From that, we can make some observations on the adequacy of those languages as embedded environments in applications.

In many applications it is important to have isolation between each script executed, such as for example, when one scripts from different customers running in a web server. As we have seen, Lua and Perl allow launching multiple runtime environments, which gives isolation guarantees. Python and Ruby, on the other hand, allow only a single state, reducing their applicability for scenarios where scripts need to run isolated from each other[6]. These two languages have yet another problem: in some cases it is not possible to bring its data space back to the original state during the execution of an application. In Python, imported modules cannot be unloaded. In Ruby, a class cannot be redefined (only extended) and `ID`s are not collected.

[6] In Python it is possible to alternate the globals table during the execution of different threads, which offers an alternative, even if less than straightforward, to obtain isolation. Even then, the global state shared by extension modules in the same.

In the implementation of the Perl plugin it became evident that its API was not designed for embedding the interpreter in applications. Besides demanding the development of an extension module so that Perl code can access C functions, we observed here that its API is incomplete with regard to its use as an embedded language. Many macros were developed assuming they would always be invoked from code written in XS files, or even by code generated by the XS pre-processor. This is confirmed by the need to pass additional undocumented parameters so that macros work, as presented in Section 4.2.3.

Lua, on its turn, has shown to be appropriate as an embedded language, not sharing the limitations described here about other languages. Besides, it features a simple API, which handles language constructs in a complete and orthogonal way, which is due both to the focus of its implementation as an embedded language, and to the minimalist design of the language itself. Even in small projects like the one presented here, which exercises only a small fraction of the APIs, we could observe that aspects where languages define special cases or have less uniformity leak through their C APIs. Both in Python and Ruby, functions returning multiple values cause implicit conversions to aggregate types (lists and arrays). In a similar way, multiple results are represented in Perl as array contexts. In their respective LibScript plugins, these features had to be handled especially. In the Lua plugin, in contrast, handling a single return value is no different from handling multiple values, like it happens in native language code.

Chapter 5

Conclusions

Choosing a scripting language depends on a series of factors, many of them relative to the language itself, others relative to its implementation. When we deal with multi-language development scenarios, an aspect that should not be neglected is the design of interfaces between languages. Be it extending the scripting language through C code, or making a C application extensible through a scripting language, the API offered by the language has a fundamental role, often influencing the design of the application.

This work discussed the main issues faced in the integration between C code and the runtime environment of a scripting language. We presented how the APIs of five languages handle these issues, indicating positive and negative points of the various approaches used. We performed a practical comparison of the use of these APIs through a case study where scripting languages were embedded to C libraries exporting one single interface. The implementation consists of a generic scripting library, called LibScript, and a series of plugins which interface different languages. We were able, this way, to observe how they handle important aspects related to embedded languages, such as passing data, function calls between two languages, error handling and isolation of runtime environments within applications.

Although the same general problems, such as data transfer, function registration and calling, are common to different usage scenarios of a scripting language API, applications embedding a virtual machine tend to demand more from the API than libraries implementing extension modules. This point is illustrated by the difficulties imposed by the Python API both in the access to global variables and registration of global functions; and, more evidently, by the complexity of Perl's API for function calls.

The fact that the Python API makes the use of global variables and functions difficult, favoring the use of modules, can be justified as a way to promote a more structured programming discipline. This is interesting when using the API for developing extension modules, given that using global variables and functions is extremely harmful in those cases, as it would pollute the namespace of Python applications. For the case where the language is embedded to provide scripting support for a C application, the absence of a convenient way to define global functions in the scripts' namespace is questionable.

The approach adopted by Perl, using a pre-processor which generates automatically

code for converting data when passing parameters and return values, has shown to be inadequate for scenarios involving embedded interpreters. Although the use of a pre-processor simplifies the simpler cases of declaration of C functions, the lack of a well-defined API for handling data transfer between the Perl interpreter and C code becomes apparent in more elaborate cases. Two of these situations happened in our case study: when receiving vararg parameters and when handling return values supporting multiple call contexts. Both demanded manipulation of low-level structures and constructs which the pre-processor aims to hide.

Interesting observations resulted from the comparison of the Java API with that from the other four scripting languages, given than, although it shares several traits with those languages, Java is not considered a scripting language. While static typing does reduce considerably the need for explicit data conversion in C code for primitive types of the language, in practice type checking for objects and the linking of fields and methods happens in a dynamic way, as these have to be performed at runtime by the JNI. Thus, regarding interaction of the virtual machine with C, advantages brought by static typing are reduced. Besides, dynamic resolution of fields and methods through C has subtle differences in behavior when compared to what occurs in native Java code, which can be a source of programmer errors.

When comparing APIs, we considered only their interfaces, making a qualitative usability analysis of each of them from the perspective of a C programmer, and not a quantitative analysis of their implementations. The performance cost added by code which performs bridging between two languages, for example, cannot be disconsidered. Many design decisions from an API are influenced by implementation requirements such as portability or performance restrictions. For example, automatic handling of scope of `VALUE`s in Ruby, scanning the C stack, brings great convenience to the programmer, but reduces the portability of its implementation.

The disparity between languages with regard to the availability of documentation also deserves mention. Java, Python and Lua feature extensive documentation, both for the languages themselves and to their C APIs. For those languages, we were able to largely base our study and the implementation of examples for the case study in the provided documentation. The documentation of Ruby relative to its C API is sparser; in [40] only part of its public API is covered. We had to make use of undocumented functions for tasks as fundamental as freeing global references registered through C. During the development of the Ruby plugin for the case study, we referred frequently to its source code to understand aspects which are not covered by the documentation about the behavior of its public functions. The documentation for Perl's C API is also incomplete, spread over several Unix manual pages included in its distribution, and in certain cases, out-of-date. To understand the various protocols involved in the practical use of the Perl API, we had to resort to the source code of applications using it.

The balance between simplicity and convenience is another recurring theme when comparing APIs. Python's extensive API, containing 656 public functions, contrasts with the 113 functions exposed by the Lua API (79 from the core API, 34 in its auxiliary API). In many situations, Python API functions abbreviate two, three ore even more calls, as in the

case of powerful functions such as `Py_BuildValue` and `PyObject_CallFunction`, resulting in short and readable C code. The approach defended by Lua is that of a minimalistic API, offering mechanisms with which more elaborate functionality can be built. In fact, in [15] a C function equivalent to `PyObject_CallFunction` is presented, using the Lua API.

Ruby exports 530 functions in its header and Perl 1209, but as only a small fraction of those is documented, it is hard to evaluate the size of their "public API" and how many of these are just function for internal use exposed in their headers[1]. This also shows that the documentation is not relevant as support material for development, but it also indicates how well-defined an API is.

The Java API is well-documented, like that from Python and Lua, but the number of exported function is not a good parameter for comparison with the other APIs as, because of statically defined types, many functions have a variant for each primitive type. Java exports its API as a structure containing function pointers; 228 functions in total are exported in this structure.

Another aspect that could be observed in this work is that the consistency of an API depends greatly on the consistency of the language it exposes. Constructions where a language lacks orthogonality, such as code blocks in Ruby or the differences when manipulating scalar and array values in Perl, end up increasing the complexity of the API and demand from the programmer specific handling in C code.

As possibilities for future work, this work can be extended through the study of other aspects of scripting language. A possible focus is the performance impact of different API designs in multi-language applications. Another is the relation between the design of a virtual machine and its respective API. Additionally, another perspective for future work lies in the continued development of the LibScript library. Possibilities include adding new plugins, review its API and exercise it by embedding the library in actual applications. LibScript and the four plugins implemented in this work are free software and are available for download at `http://libscript.sourceforge.net`.

[1] Some functions are marked as being for internal use only, but most of them have no indication whatsoever.

Bibliography

[1] D. M. Beazley. SWIG: an easy to use tool for integrating scripting languages with C and C++. In USENIX Association, editor, *4th Annual Tcl/Tk Workshop '96*, pages 129–139, Berkeley, CA, USA, July 1996. USENIX.

[2] Nick Benton and Andrew Kennedy. Interlanguage working without tears: blending SML with Java. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 126–137, New York, NY, USA, 1999. ACM Press.

[3] Don Box and Chris Sells. *Essential .NET, Volume I: The Common Language Runtime*. Addison-Wesley, Boston, MA, USA, 2002.

[4] Barbara Chapman, Matthew Haines, Piyush Mehrotra, Hans Zima, and John Van Rosendale. Opus: A coordination language for multidisciplinary applications. *Scientific Programming*, 6(4):345–362, Winter 1997.

[5] Suzanne Collin, Dominique Colnet, and Olivier Zendra. Type inference for late binding: The SmallEiffel compiler. In *Joint Modular Languages Conference, JMLC'97*, volume 1204 of *Lecture Notes in Computer Sciences*, pages 67–81. Springer Verlag, 1997.

[6] Melvin E. Conway. Proposal for an UNCOL. *Communications of the ACM*, 1(10):5–8, 1958.

[7] Ana Lucia de Moura, Noemi Rodriguez, and Roberto Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, July 2004.

[8] Zdenek Dvorak. Gimplification improvements. In *GCC Developers' Summit*, pages 47–56, Ottawa, Canada, June 2005.

[9] Ecma International. *C++/CLI Language Specification*, 2005. Standard ECMA-372.

[10] Greg Ewing. Pyrex - a language for writing Python extension modules, 2006. `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`.

[11] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. H/Direct: a binary foreign language interface for Haskell. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 153–162, New York, NY, USA, 1998. ACM Press.

[12] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison-Wesley, Boston, MA, USA, 2nd edition, 2000.

[14] Jim Hugunin. Python and Java - the best of both worlds. In *Proceedings of the 6th International Python Conference*, pages 11–20, San Jose, CA, USA, October 1997.

[15] Roberto Ierusalimschy. *Programming in Lua.* Lua.org, 2nd edition, March 2006.

[16] International Organization for Standardization. *Ada 95 Reference Manual. The Language. The Standard Libraries*, January 1995. ANSI/ISO/IEC-8652:1995.

[17] International Organization for Standardization. *C# Language Specification*, 2006. ISO/IEC 23270:2003.

[18] Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. C--: A portable assembly language that supports garbage collection. In *PPDP '99: Proceedings of the International Conference PPDP'99 on Principles and Practice of Declarative Programming*, pages 1–28, London, UK, 1999. Springer-Verlag.

[19] Simon Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell Compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, pages 249–257, Keele, Staffordshire, UK, March 1993.

[20] KDE.org. KDE developer's corner - Language bindings, October 2006. `http://developer.kde.org/language-bindings/`.

[21] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[22] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 2nd edition, April 1999.

[23] Doug MacEachern and Jon Orwant. *perlembed(1).* Perl 5 Porters, 5.8.8 edition, January 2006. `http://perldoc.perl.org/perlembed.html`.

[24] Ariel Manzur and Waldemar Celes. toLua++ reference manual, April 2006. `http://www.codenix.com/~tolua/tolua++.html`.

[25] Paul Marquess. *perlcall(1)*. Perl 5 Porters, 5.8.8 edition, January 2006. `http://perldoc.perl.org/perlcall.html`.

[26] John R. Metzner. A graded bibliography on macro systems and extensible languages. *SIGPLAN Not.*, 14(1):57–64, 1979.

[27] Gustavo Niemeyer. Lunatic Python, July 2006. `http://labix.org/lunatic-python`.

[28] Object Management Group, Inc., Framingham, MA, USA. *The Common Object Request Broker: Architecture and Specification, Version 3.0*, 2002.

[29] Jeff Okamoto. *perlguts(1)*. Perl 5 Porters, 5.8.8 edition, January 2006. `http://perldoc.perl.org/perlguts.html`.

[30] Jeff Okamoto and Dean Roehrich. *perlapi(1)*. Perl 5 Porters, 5.8.8 edition, January 2006. `http://perldoc.perl.org/perlapi.html`.

[31] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[32] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, 1998.

[33] Pearu Peterson, Joaquim R. R. A. Martins, and Juan J. Alonso. Fortran to Python interface generator with an application to aerospace engineering. In *Proceedings of the 9th International Python Conference*, Long Beach, CA, USA, March 2001.

[34] Allison Randal, Dan Sugalski, and Loepoid Toetsch. *Perl 6 and Parrot Essentials*. O'Reilly Media, Inc., 2nd edition, 2004.

[35] Dean Roehrich. *perlxs(1)*. Perl 5 Porters, 5.8.8 edition, January 2006. `http://perldoc.perl.org/perlxs.html`.

[36] Levon Stepanian, Angela Demke Brown, Allan Kielstra, Gita Koblents, and Kevin Stoodley. Inlining Java native calls at runtime. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 121–131, New York, NY, USA, 2005. ACM Press.

[37] Sun Microsystems. *Java Native Interface 5.0 Specification*, 5.0 edition, 2003. `http://java.sun.com/j2se/1.5.0/docs/guide/jni/`.

[38] Don Syme and James Margetson. Microsoft F#, 2006. `http://research.microsoft.com/fsharp/`.

[39] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: compiling standard ML to C. *ACM Lett. Program. Lang. Syst.*, 1(2):161–177, 1992.

[40] David Thomas and Andrew Hunt. *Programming Ruby: The Pragmatic Programmer's Guide*. Addison Wesley Longman, Inc., Boston, MA, USA, 2nd edition, 2004.

[41] Reuben Thomas. Lua Technical Note 4 - a thin API for interlanguage working, or Lua in four easy calls, August 2002. `http://www.lua.org/notes/ltn004.html`.

[42] Andrew P. Tolmach and Dino Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.

[43] Guido van Rossum. *Python/C API Reference Manual.* Corporation for National Research Initiatives (CNRI), Reston, VA, USA, 1.5.2 edition, April 1999. `http://www.python.org/doc/1.5.2/api/api.html`.

[44] Guido van Rossum. *Extending and Embedding the Python Interpreter*, 2.4.3 edition, March 2006. `http://docs.python.org/ext/ext.html`.

[45] Guido van Rossum. *Python Reference Manual.* Python Software Foundation, 2.4.3 edition, March 2006. `http://docs.python.org/ref/`.

[46] Guido van Rossum. *Python/C API Reference Manual.* Python Software Foundation, 2.4.3 edition, March 2006. `http://docs.python.org/api/api.html`.

[47] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl.* O'Reilly, 3rd edition, July 2000.

[48] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, pages 1–42, Saint-Malo, France, 1992. Springer-Verlag.

Appendix A

The LibScript API

A.1 Startup and termination

- `script_env* script_init(const char* namespace)`
  Initializes LibScript and returns a pointer to a virtual environment. The `namespace` argument indicates the name to be used in structures to be created in the namespace of virtual machines to represent the virtual environment.

- `void script_done(script_env* env)`
  Terminates the virtual environment.

A.2 Function registration

- `typedef script_err (*script_fn)(script_env*)`
  Type of C functions to be registered in the virtual environment. When exposing an existing API to LibScript, the function will typically be a wrapper that loads input parameters from the environment, calls a program function and sends output parameters back to the environment.

- `script_err script_new_function(script_env* env, script_fn fn, const char* name)`
  Registers a new C function in the virtual environment.

A.3 Arguments buffer

- `double script_get_double(script_env* env, int index)`
  `int script_get_int(script_env* env, int index)`
  `int script_get_bool(script_env* env, int index)`
  `const char* script_get_string(script_env* env, int index)`
  Obtain data from the buffer. These functions should be called by the beginning

of wrapper functions. For each input parameter a call should be performed. Once done, one can invoke the `SCRIPT_CHECK_INPUTS(env)` macro, which terminates the function with an error code in case any of these function have not found data of the expected type (the API does not perform automatic conversion between strings and numbers). In `script_get_string`, the returned string belongs to the caller, which becomes responsible for deallocating it.

- `script_type script_get_type(script_env* env, int index)`
  `int script_buffer_len(script_env* env)`
  These functions allow writing C functions that perform type checking and verify the number of arguments at runtime. The `script_get_type` function obtains the type of the requested buffer element and `script_buffer_len` returns the number of arguments in the buffer.

- `void script_put_double(script_env* env, int index, double value)`
  `void script_put_int(script_env* env, int index, int value)`
  `void script_put_bool(script_env* env, int index, int value)`
  `void script_put_string(script_env* env, int index, const char* value)`
  Insert data in the buffer. By the end of a function, return values should be passed with calls to these functions and a `SCRIPT_OK` error code as the return value of the C function.

- `void script_reset_buffer(script_env* env)`
  Empties the buffer.

A.4 Running code

- `script_err script_run(script_env* env, const char* language, const char* code)`
  Runs a string of code in a given language. If necessary, the appropriate plugin is loaded and initialized.

- `script_err script_run_file(script_env* env, const char* filename)`
  Convenience function; loads the contents of a file and runs it with `script_run`.

- `script_err script_call(script_env* env, const char* fn)`
  Requests the execution of a function in some of the registered plugins. Input parameters should be passed previously with calls to the `script_put_*` functions; return values can be obtained with `script_get_*`. Initially, the environment's table of C functions is consulted. If there is no function defined in C, plugins are consulted in the same order as they were implicitly initialized through script_run or `script_run_file`: functions registered in each plugin's representation of the virtual environment are available through `script_call`.

- `script_err script_error(script_env* env)`
  `const char* script_error_message(script_env* env)`
  `void script_set_error_message(script_env* env, const char* message)`
  Obtain the most recent error code and message from an environment. After calling `script_error`, the error code is reset back to `SCRIPT_OK`. The error message is not reset after being checked. The `script_set_error_message` function defines a new value for the environment's error message. This allows the plugin propagating to the application error messages from the virtual machine.

- `const char* script_get_namespace(script_env* env)`
  Returns the name of the namespace registered with `script_init`.

A.5 API exported by plugins

Calls to plugins that implement interfaces to different virtual machines are performed internally by the main library, which expects to find in plugins the following functions:

- `script_plugin_state script_plugin_init_`*language*`(script_env* env)`
  Responsible for initializing a plugin. During initialization, the virtual environment returns a `script_plugin_state` to the main library, which is the opaque representation of its virtual state. The contents of this representation vary from language to language, but the basic principle is that two data should be available from this value: a reference to the LibScript virtual environment, so that the plugin can make calls to the main library, and an identifier that allows the plugin to access the language-specific data structure that represents the namespace of LibScript-accessible functions.

- `script_err script_plugin_run_`*language*`(script_plugin_state st, char* text)`
  Sends code for execution in the virtual machine. This function is used internally by `script_run` and `script_run_file`. It should return `SCRIPT_OK` in case of success, `SCRIPT_ERRLANGCOMP` for compile errors or `SCRIPT_ERRLANGRUN` for runtime errors, preferably defining an error message with `script_set_error_message`.

- `script_err script_plugin_call_`*language*`(script_plugin_state st, char* fn)`
  Calls a function which has been defined natively in the environment namespace of the plugin's virtual machine. When calling a function from this language's copy of virtual environment, either in C through `script_call` or from code from other plugins (which would also route through `script_call`), LibScript will use this function to try to run it. If the function was not defined in the plugin, the `SCRIPT_ERRFNUNDEF` value must be returned. Otherwise, it should be executed, with input parameters obtained through `script_get_*` and return values sent with `script_put_*`, and values `SCRIPT_OK` or `SCRIPT_ERRLANGRUN` should be returned as appropriate.

- `void script_plugin_done_`*language*`(script_plugin_state st)`
  Responsible for terminating an environment.