C APIs in extension and extensible languages

Hisham Muhammad
Roberto Ierusalimschy

[1]Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro (PUC-RIO)
Rio de Janeiro, RJ – Brazil

Abstract. *Scripting languages are used in conjuction with C code in two ways: as extension languages, where the interpreter is embedded as a library into an application; or as extensible languages, where the interpreter loads C code as add-on modules. These two scenarios share many similarities, as in both of them two-way communication of code and data needs to take place. However, the differences between them impose design tradeoffs that affect the C API that bridges the two languages, often making a scripting language more suitable for extending than embedding, or vice-versa. This paper discusses how these tradeoffs are handled in the APIs of popular scripting languages, and the impact on their use as embedded or extensible languages.*

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, but ideally, a language should provide a foreign language interface that allows programmers to send and receive both calls and data to another language [Finne et al. 1998].

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 [Ousterhout 1998], 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 being implemented 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, bring 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.

Scripting languages have the distinction that, by design, they are developed having interaction with code written in other languages in mind. 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.

Scripting languages are used in conjuction with C code in two ways: extending a C application, where the interpreter is embedded as a library; or by having C code extend the language, through add-on modules written as C libraries. These two scenarios share many similarities, as in both of them two-way communication of code and data needs to take place. However, the differences between them impose tradeoffs that affect the design of the resulting C API.

This paper discusses how the design of a language's C API affects its suitability for different application scenarios. In Section 2, we discuss the different roles of scripting languages. In Section 3, the main issues involving interaction of C code with scripting language runtime environments are presented, followed by a discussion in Section 4 on how popular scripting languages address those issues and the effect of their designs in their applicability as extension and extensible languages. Finally, Section 5 concludes the paper.

2. Extension and extensible languages

Scripting languages are designed to be used in two-language scenarios. Originally, they had an auxiliary role, in which user scripts allow for customization of applications. With the increased popularity of scripting languages, a different usage model has also risen to prominence, in which the scripting language performs a more central role. Typical examples are graphical applications where the interface is described by scripts controlling components implemented in C and games where the logic is described in scripts and the runtime engine is implemented in lower-level languages.

In these scenarios, there is a clear distinction between a lower-level layer where performance is a critical factor and another, higher-level layer that coordinates operations on elements of the lower layer. Scripting languages cease to be just an extension mechanism: the application itself is written using the scripting language and libraries written in lower-level languages are loaded as extension modules.

It makes sense, then, when discussing language interaction, to make a distinction between *extensible languages* and *extension languages*. Extensible languages are those that can be extended through external modules implemented in other languages. Extension languages are those which runtime environment can be embedded in an application, allowing to use them to extend the application. Typically, scripting languages can be used, with variable degrees of convenience, as either extensible or extension languages.

Another interesting observation is that, while in one model the scripting language serves as an extension language for the lower-level language in which the application is written, in the other model the opposite happens: we can look at add-on modules written using the language's C API as a way to extend the scripting language using C; in this

perspective, C becomes the extension language.

This way, the set of features provided by an API between C and a scripting language tends to be symmetric in case it is desired to provide language extensibility as well as promote its use as an extension language. In both situations, code and data manipulation features need to be provided in both directions. A few common issues arise when implementing interaction between C and scripting languages; they are discussed in the following section.

3. Interaction between C and scripting languages

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.

3.1. Data transfer

The main complexity in the interaction between programming languages is not the difference 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.

Since the format how these data are represented often differs, the alternatives to perform data transfer between languages involve either converting the data or manipulating it opaquely through some kind of handle. The duplication that takes place when converting data limits the applicability of this method, restricting its use typically to numeric types and, in minor scale, strings. When exposing handles, the source language may explicitly offer facilities in the target language to manipulate these data, that is, the data remains opaque, but the language can access its contents through an API.

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). There may also be the need to handle conversion of endianness and format of floating point numbers. So, even in cases where it is possible to link C code directly, bindings libraries are still usually needed to make the manipulation of complex types more convenient.

For types such as strings, the size of values also brings performance concerns. 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 data of structured types to C, the conversion to a native C type, in many cases, is not an option. 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 typing, like Java, copying 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 an API that exposes the operations defined over those types as C functions. 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.

3.2. Garbage collection

From the moment when C code gains access to handles to data from the storage space of another language, the programmer must take into consideration the differences between the memory management models involved. 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.

It is necessary, then, to indicate in C 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 on the design of the C API, but also on the garbage collection mode employed by the implementation of the language.

3.3. Function calls and registration

When bridging C and a scripting language, it is necessary to provide a form of invoking, from C, functions to be executed by the scripting language, and vice-versa. This combines the issues of data transfer, for passing arguments and receiving results between these two “spaces”, and the implications that this brings about the objects’ lifetime, affecting garbage collection. The tasks involved are always the same – perform conversion of input data, pass parameters to the other language, specify which function to call, obtain return values, convert them back to the other language – but approaches employed in scripting

language APIs vary widely. In the next section we will discuss how some APIs implement these tasks and the impact of their design on their usability as extension and extensible languages.

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 to define an API of functions for performing calls to the scripting language. Conversely, 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, 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.

4. Scripting language API designs

A pioneering example of an embedded, extension language is Tcl [Ousterhout 1994]. Four main goals were set in its original design [Ousterhout 1990]: focus as a command language (designed to write short programs); extensibility; simplicity in its implementation; simple interface with C applications. We observe in those goals principles that are now understood as fundamental features of extensible and extension languages: extensibility was listed as a goal explicitly; the last two goals point out its focus as an extension language.

Aiming to simplify the interaction with C code, Tcl uses strings as its single data type. This minimalism, which has shown to be an advantage for Tcl as an extension language, makes it seem limited compared to languages like Python, which provide a more complete feature set as an extensible language. Scripting languages have grown beyond Tcl's focus as a command language, and thus, Tcl gradually lost space in the scripting world. Its historical importance, however, is undeniable: it was the concept introduced by Tcl of implementing scripting languages as C libraries that pushed strongly the development of extensible applications.

In this section, we discuss the design of the C APIs of four popular scripting languages, Python [van Rossum 2006b], Perl [Wall et al. 2000], Ruby [Thomas and Hunt 2004] and Lua [Ierusalimschy 2006], in terms of the interaction issues outlined in the previous section, while also contrasting them with the C API of Java [Gosling et al. 2000]. Unlike the others, Java uses static typing – which allows us to observe how typing affects the design of an API – but like them it is based on a virtual machine model, features automatic memory management and allows dynamic loading of code, and most importantly, it can both be embedded as an extension language and be extended with native C code.

4.1. Data transfer

The basic set of functions for manipulating data in scripting language APIs is usually the same: they provide functions for converting values from the language to basic C types and vice-versa. A central design issue lies in how to represent a value between languages. All values in the Python virtual machine are represented as objects, mapped to

the C API as the `PyObject` structure [van Rossum 2006a]. More specific types such as `PyStringObject`, `PyBooleanObject` and `PyListObject` are `PyObject`s by structural equivalence, that is, they can be converted through a C cast. Similarly, in Ruby, the 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. Perl also provides handles to its data in C, but these C values are better understood as containers to Perl values: types of Perl variables are mapped to C structs `SV` for scalars, `AV` for arrays, `HV` for hashes. 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.

Lua, in contrast, employs a different approach for manipulating data in C: no pointers or handles to Lua objects are ever exposed to C code, and instead, 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. While this results in the simplest and most orthogonal data manipulation API among the ones mentioned, code in which values are associated to stack indices tends to be less natural-looking than code using C variables – the manipulation of, say, a Ruby `VALUE` is syntactically similar to that of other C types: an assignment to a `VALUE` is done in C with an assignment.

All of these languages also offer API functions for manipulating their fundamental structured types (tables in Lua, 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. In Java, static typing reduces greatly the need for explicit data conversion in C code. The Java Native Interface (JNI) [Sun 2003] defines C types equivalent to each of Java's primitive types (`jint` for `int`, `jfloat` for `float`, and so on). While to return an integer to Python from C one would have to use a command such as `return PyInteger_New(42)`, when interfacing Java they could simply write `return 42`. Reference types, such as classes and objects, are exposed to C as opaque references, instances of `jobject`. On the other hand, treatment of multi-threading complicates the access of types such as strings and arrays.

An important task when bridging C code to a scripting language is the creation of data in the scripting language environment containing C structures. Perl, Ruby and Lua provide simple mechanisms for this task. Ruby offers the `Data_Wrap_Struct` macro which receives a C structure and returns a Ruby `VALUE`. Lua defines a basic type in the language especially for this end, called *userdata*, which contains a memory block managed by the Lua VM that is accessible to C code but is an opaque object when accessed from Lua. In Perl, one can create `SV`s containing arbitrary memory blocks for use in C. In Python, 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, which are closely tied to the implementation of the Python VM. The complexity of code that interacts with C data types using the Python API tends to be less problematic in an isolated piece of code such an extension module

(which is typically centered around the declaration of these types and their methods) than when inserted in a larger body of code, as it happens with an embedded interpreter. Using the JNI, it is not possible to create new Java types from C; one can only load precompiled classes.

Another common need when interacting with C is to store pointers in the data space of the scripting language. Python, Lua and Perl offer features to do this directly. In Python, a `PyCObject` is a predefined type that holds a void pointer accessible from C. Lua offers a built-in type for this end, *light userdata*, which differs from *userdata* in that the memory block it points to is not managed by the virtual machine. In Perl, the same can be achieved storing a pointer in the data area of an `SV`. In Ruby and Java, there is no direct way to store pointers. 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)`.

4.2. Garbage collection

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 discussed in this work, the only one 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[1]. Indeed, the various available implementations of the JVM use different algorithms for garbage collection. We observe, then, that while most languages abstract the specifics of the garbage collector, details of the garbage collection algorithm tend to show up in the APIs. Since in pragmatic terms the API compatibility of an implementation is as important as language compatibility, this means that, due to the API, language implementations end up tied to specific garbage collection algorithms because of their API even if they are transparent to the language itself.

Another issue that arises in the communication between C and scripting languages is the management of references. For manipulating data through the API, Lua and Ruby demand the least concerns from the programmer about managing references. Lua avoids

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

the problem altogether, by keeping its objects in the virtual stack and not returning references to C code; accessing data from Lua, thus, always involves function calls for getting the data into the stack.

In Ruby, only objects stored in C globals and not referenced from Ruby need to be notified, using the `rb_global_variable` function; objects in the local scope of a C function 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 `#ifdefs` 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.

Both Perl and Java handle the issue of references stored in local variables in a similar way, by distinguishing references as either global or local (local references are called "mortal variables" in Perl). Local references allow for mostly implicit management. API functions in Java return local references by default, which can be converted to global ones with the API call `NewGlobalRef`. In Perl, the opposite happens, and global references can be converted to local ones with the `sv_2mortal` function. Java's approach is more interesting, as normally more locally-scoped than globally-scoped variables are used.

4.3. Function calls

In Python, Lua and Perl, functions can be accessed as language objects and invoked. Python allows any `PyObject` to be called as a function, as long as they implement the `__call__` method, which can be written in either Python or C (as a function registered in the object's `PyTypeObject` struct). Like in data manipulation, Python offers an extensive API, with several convenience functions allowing parameters to be passed as Python tuples, as Python objects given as varargs, as C values to be converted by the invocation function, etc. In Lua, there is a built-in primitive type, *function*, which represents both Lua functions and C functions registered in the Lua VM. Perl also allows functions to be manipulated as first-class objects using its C API, returning SV structs representing them.

In Ruby as well as Java, methods are not first-class objects, and therefore their APIs define specific C types used to reference them – `jmethodID` in Java and `ID` in Ruby[2]. 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 spec-

[2] An ID is merely a reference to the symbol table entry corresponding to the method's name, and not a unique identifier for the method itself.

ify how their conversion should be made. Ruby also offers some variants of invocation functions.

Lua separates the function call routine from argument passing, which is done in a previous step by setting up the contents of 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, in which arguments to the function call are written in C as arguments to the C API call. Perl also features function calls using a stack model, but its use is exceedingly complex, demanding a macro protocol to be followed which exposes the internal workings of the interpreter [Marquess 2006]. 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: 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.

4.4. Registration of C functions

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, as arguments are passed through the stack and not as arguments to the C function.

In Java, function signatures are created through the `javah` tool [Liang 1999] – 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.

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 [Roehrich 2006]. Instead of isolating the access to Perl's internals through a public API, the proposed approach is to encapsulate the process of generating wrapper code using interfaces written in `.xs` files. These files contain C code along with annotation that simplifies the handling of input and output parameters. In fact, Perl does not expose a documented API for registering functions [Okamoto and Roehrich 2006]. Because of that, it is not practical for an application to embed a Perl interpreter and expose it to a set of C functions using C code only. The alternative is to write a Perl extension using XS and import the resulting package in the embedded Perl interpreter.

Registration of functions in Ruby and Lua is simple. In Lua, in particular, it is an assignment (made through API calls), not different from any other object. In Python, there are features for batch registering, using arrays of the `PyMethodDef` struct (Lua offers a similar feature with `luaL_register` function), but there is no simple way to register a single function – again, this shows a focus on extending rather than embedding:

extension modules tend to register many functions at once, while embedded interpreters often register global functions. Both in Java and Perl, function registration is done implicitly by the generation tools, and there are no public API functions for registering new C functions at runtime in either of them.

5. Conclusion

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.

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.

Interesting observations resulted from the comparison of the Java API with that from the other four scripting languages, given that, 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.

Throughout the development of this work, we implemented as a case study a C library called LibScript[3], which provides an extensibility architecture for applications in

[3] `http://libscript.sourceforge.net`

a language-independent manner: scripting language VMs are loaded dynamically as LibScript plugins. In the implementation of these plugins, we had a chance to exercise the APIs of the different scripting languages performing similar tasks.

The disparity between languages with regard to the availability of documentation 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 the case study on the provided documentation. The documentation of Ruby relative to its C API is sparser; in [Thomas and Hunt 2004] only part of its public API is covered. One has to make use of undocumented functions for tasks as fundamental as freeing global references registered through C.

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 [Ierusalimschy 2006] 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 functions for internal use exposed in their headers[4]. This also shows that the documentation is not only 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 functions is not a good parameter for comparison with the other APIs as, because of its 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.

An aspect that is equally important when extending or embedding 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 other names, which in the case of embedding are defined by the application, and in the case of extending are defined by the library being exposed to the language. Perl and Ruby define names in a disorganized fashion, which occasionally causes problems. 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.

Another point 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.

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

The focus in extending or embedding adopted by a language's C API has as much impact in its suitability for one or other scenario as the design of the language itself. The interaction between the design of the language, its implementation and its API all affect each other in often subtle ways – APIs like those from Lua and Java, which allow multiple interpreters to run concurrently, show a design concern on embedding, while those from Perl, Python and Ruby focus on providing facilities to make it easier to write extension modules. Given that an API designed towards embedding also encompasses the needs of APIs for extension modules, and that module generation tools such as SWIG [Beazley 1996] (as well as language-specific tools such as [Ewing 2006, Niemeyer 2006, Manzur and Celes 2006]) are becoming increasingly powerful and popular, we observe that a C API aiming to support both extension and embedding should focus on the latter, as that tends to demand more from both the API and the language implementation.

References

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

Ewing, G. (2006). Pyrex - a language for writing Python extension modules. `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`.

Finne, S., Leijen, D., Meijer, E., and Jones, S. P. (1998). 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. ACM Press.

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2000). *The Java Language Specification*. Addison-Wesley, Boston, MA, USA, 2nd edition.

Ierusalimschy, R. (2006). *Programming in Lua*. Lua.org, 2nd edition.

Liang, S. (1999). *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Manzur, A. and Celes, W. (2006). toLua++ reference manual. `http://www.codenix.com/~tolua/tolua++.html`.

Marquess, P. (2006). *perlcall(1)*. Perl 5 Porters, 5.8.8 edition. `http://perldoc.perl.org/perlcall.html`.

Niemeyer, G. (2006). Lunatic Python. `http://labix.org/lunatic-python`.

Okamoto, J. and Roehrich, D. (2006). *perlapi(1)*. Perl 5 Porters, 5.8.8 edition. `http://perldoc.perl.org/perlapi.html`.

Ousterhout, J. K. (1990). Tcl: An embeddable command language. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 133–146, Berkeley, CA. USENIX Association.

Ousterhout, J. K. (1994). *Tcl and the Tk Toolkit*. Addison Wesley.

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

Roehrich, D. (2006). *perlxs(1)*. Perl 5 Porters, 5.8.8 edition. `http://perldoc.perl.org/perlxs.html`.

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

Thomas, D. and Hunt, A. (2004). *Programming Ruby: The Pragmatic Programmer's Guide*. Addison Wesley Longman, Inc., Boston, MA, USA, 2nd edition.

van Rossum, G. (2006a). *Extending and Embedding the Python Interpreter*, 2.4.3 edition. `http://docs.python.org/ext/ext.html`.

van Rossum, G. (2006b). *Python Reference Manual*. Python Software Foundation, 2.4.3 edition. `http://docs.python.org/ref/`.

Wall, L., Christiansen, T., and Orwant, J. (2000). *Programming Perl*. O'Reilly, 3rd edition.