Efficient Compilation of Lua for the CLR

Fabio Mascarenhas
Department of Computer Science, PUC-Rio
Rio de Janeiro, Brazil
mascarenhas@acm.org

Roberto Ierusalimschy
Department of Computer Science, PUC-Rio
Rio de Janeiro, Brazil
roberto@inf.puc-rio.br

ABSTRACT

Microsoft's Common Language Runtime offers a target environment for compiler writers that provides a managed execution environment and type system, garbage collection, access to OS services, multithreading, and a Just-In-Time compiler. But the CLR uses a statically typed intermediate language, which is a problem for efficient compilation of dynamically typed languages in general, and the Lua language in particular.

This paper presents a way to implement a Lua compiler for the CLR that generates efficient code. The code this compiler generates outperforms the same code executed by the Lua interpreter and similar code generated by Microsoft's IronPython compiler. It approaches the performance of Lua code compiled to native code by LuaJIT.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

clr, lua, compilers, dynamic languages

1. INTRODUCTION

Microsoft's Common Language Runtime offers a target environment for compiler writers that provides a managed execution environment and type system, garbage collection, access to OS services, multithreading, and a Just-In-Time compiler [10]. This environment also lets the languages targeting it to interface without the resource management problems common when interfacing different runtimes, as all the languages share a single garbage-collected heap.

The CLR has an object-oriented type system with both value types (structures), stored in the stack and reference types (classes, interfaces, and delegates), stored in the heap. It has polymorphism by inheritance with virtual functions, and a form of parametric polymorphism with generic types. The CLR also has a high-level intermediate language that other languages have to compile to. This language is statically typed.

The static typing of CIL is a problem for the compilation of dynamically typed languages on the CLR; either the compiled code has to resort to runtime type checks and casting, or all types in the language have to be mapped to CLR types with a single ancestor, and all operations implemented as virtual methods in these types. If the dynamically typed language is also object oriented it cannot use the CLR's method dispatch for its dynamic types without using the CLR's slow reflection API. Dynamically typed languages are not a good fit to the CLR's type system and CIL, and this translates to runtime overhead and more work if an efficient implementation is desired.

In this paper we focus on efficient implementation of a particular dynamically typed language, the Lua language, for the CLR. Lua is a scripting language designed to be simple, small, portable, fast, and easily embedded into applications. Since its creation it has seen wide use in all kinds of industrial applications and is one of the leading scripting languages in game development [6].

Despite its simplicity, Lua possesses advanced features such as extensible semantics, anonymous functions with full lexical scoping (Lua's lexical scoping is similar to Scheme's in the presence of side effects), tail call optimization, and full asymmetric coroutines [1]. It has six types: floating point numbers, immutable strings, booleans, functions, tables (associative arrays), and nil. Lua's extensible semantics lets a program redefine all operations of a table (and several operations on the other types, but affecting all instances of the type). Extensible semantics, tables, first-class functions, plus some syntactic sugar allows users to implement either class-based or prototype-based OO on top of Lua. A complete description of Lua can be found at its reference manual [7], and a brief overview is in the HOPL III paper [6].

A Lua compiler for the CLR needs to map Lua's type system onto the CLR's, implement operations on those types using CIL code, and add any extra needed runtime support. Designing a compiler that does this while generating efficient code is not a trivial task. Our previous effort [9] presented a compiler that used Lua virtual machine bytecodes as input, mapping Lua types to a parallel hierarchy of CLR types with a common ancestor and using a mixture of inline CIL code and calls to a Lua runtime to implement operations. The compiled code does not use the CLR stack for local variables and parameter passing, keeping those in a private stack. Code generated by this compiler performs similarly to the same code when ran by the Lua interpreter.

Our goal was to produce a new Lua compiler that clearly outperforms the Lua interpreter and this previous effort, now consuming Lua source code and using the CLR stack as much as possible. We wrote a first compiler that used the same type mapping as the previous effort, but with the new approach for implementing the operations. We then did an optimization of function calls with less than two return values. Our next step was to replace the original type mapping with a new one that gave a significant speedup, and finally we optimized some frequently-done operations.

We used a series of micro-benchmarks to evaluate our compilers, each stressing a different part of the implementation, and compared the compilers among themselves and against the latest version of the Lua interpreter and a Lua Just-In-Time compiler. Our last compiler beats the Lua interpreter by a good margin, and comes close the the JIT compiler in two of the benchmarks. We believe further improvement can only come through much more complex optimization.

We also evaluated our compilers against IronPython, a Python compiler for the CLR that uses Microsoft's Dynamic Language Runtime, a support layer that Microsoft is building for efficient compilation of dynamic languages on the CLR [5]. Our last compiler performed better than IronPython on our benchmarks. This result, and some mismatches between the DLR's semantics and Lua's led us to believe that targeting the CLR directly instead of the DLR is the best option for efficient compilation of the Lua language for the CLR at this time.

In the remainder of the paper, we first summarize the first compiler we implemented and give our benchmarks and results for it (section 2), then present the changes we made to build the other three compilers and show how they performed (section 3). We then present other work on compiling Lua and similar languages for the CLR (section 4), and finally offer our conclusions and give the directions where out future effort is going (section 5).

2. THE BASIC COMPILER

Our first compiler used a simple mapping from the Lua 5.1 type system to the CLR type system. We used a `Lua.Value` structure to represent all Lua values. This structure had two fields: one of them stored a number and the other a reference to values of other Lua types (or `null` if it was just a number). The other Lua types mapped to CLR classes with a common `Lua.Reference` superclass that defined virtual methods for each operation the Lua types supported. The mapping was essentially the same mapping described on previous work about executing Lua 5.0 code on the CLR [9].

Mapping to a structure and a type hierarchy with a common supertype avoided the need for casting or CLR type checks on the code the compiler generated. Each operation only needed a check to see if the value was a number (testing if the `Lua.Reference` field was `null`), and either doing the operation inline, in case of numbers, or calling a virtual method on `Lua.Reference`, for other types.

The drawback of using a parallel hierarchy for Lua types was to make it harder for other CLR languages to interface with Lua code, as CLR values needed to be wrapped in a `Lua.Value` instance before Lua code could use them, and then unwrapped if Lua code called CLR code. This wrapping/unwrapping could be made completely transparent to Lua code, though, using an approach similar to the one used to interface the Lua interpreter with the CLR [8].

Our compiler mapped each function in the Lua source to a CLR class derived from `Lua.Reference`. All functions in the actual running code were instances of their corresponding class. This class had an `Invoke` method that received the same number of parameters as the function, and returned an array of Lua values; this method held the compiled code for the function.

The base `Lua.Reference` class declared ten overloaded and virtual versions of `Invoke`, nine of them receiving a number of Lua values from zero to eight, and the last one receiving an array of Lua values. When generating code for a function our Lua compiler generated code for each of these versions that adjusted the number of arguments (or unpacked the array) and then called the version with the correct number of parameters. Calling a function with the wrong number of arguments is not an error in the Lua language; Lua silently adjusts the argument list to the correct size, either dropping extra arguments or filling missing arguments with the singleton type *nil*. Our compiler just did the same. The number of arguments received and the number needed are known at compile-time, so generating the code to adjust the argument lists was a straightforward task.

Lua functions are first-class values with lexical scoping, and the Lua interpreter implements them as true closures: distinct functions that access the same variable share the same variable slot, instead of each receiving a distinct copy of the variable, in a manner similar to closures in the Scheme language. Each Lua closure stores references to external variables (called *upvalues*) instead of their actual value.

We wanted to model Lua's closure semantics on the CLR, while still using the CLR stack to store local variables and arguments. We did this by treating upvalues differently from normal variables: upvalues actually lived in the CLR heap, and the stack frame where the upvalue was defined, as well as any closures that used it, stored a reference to the value instead of the actual value. We implemented these references as an one-element array of `Lua.Value`. Functions that reference the upvalue received this array on their constructors and stored it in a field. We needed to store upvalues in the heap because the CLR does not allow references to local variables outside the current stack frame (if it did allow these references the CLR would also need to provide some way to copy them somewhere else when the stack unwinds while keeping the references valid).

Code generation inlined operations on numbers, and delegated operations on other types to virtual methods of `Lua.Reference`. Our compiler did no other kinds of code optimization. It fully conformed with Lua 5.1's syntax and semantics[1].

The compiler presented in this section is the baseline for our benchmarks. To evaluate it and the compilers presented in the next section, we used a set of three micro-benchmarks stressing different parts of the implementation, all of them built around the Fibonacci sequence. Each set of columns in Figures 1-5 is a benchmark.

The first benchmark (**rec**) is a straightforward recursive implementation that stresses function calls and returns; the second benchmark (**memo**) is a memoized implementation that stresses array access and closures, and the last one (**iter**) is an iterative implementation that stresses loops and simple arithmetic. We did these micro-benchmarks to have some measure of the impact of the changes we further did on the compiler; we had no intention of simulating a real workload.

For comparison purposes we also ran the same benchmarks under the Lua 5.1.3 interpreter and under LuaJIT 1.1.3 [11], a Just-In-Time compiler of Lua to native code. We ran all benchmarks on the same platform, a computer with a 2.16Ghz Intel Core Duo processor and 2GB RAM running Windows Vista and the latest version of the CLR.

The results of the first benchmark are on Figure 1, where *Lua VM* is the Lua interpreter and `LuaCLR` is our Lua compiler. These

[1]With the exception of the treatment of weak references, where it was still restricted by CLR's weak reference semantics. See our previous work [9] for more information.

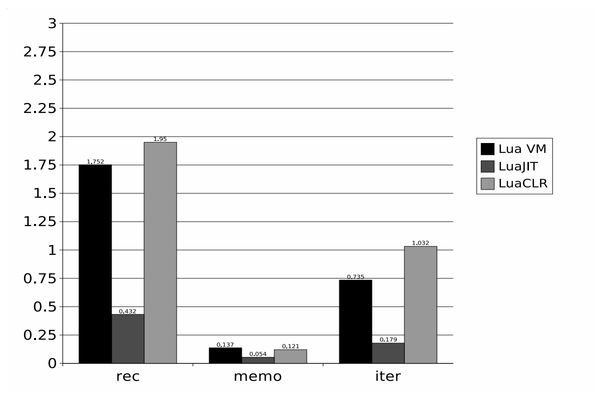

Figure 1: Benchmark results for the first LuaCLR compiler.

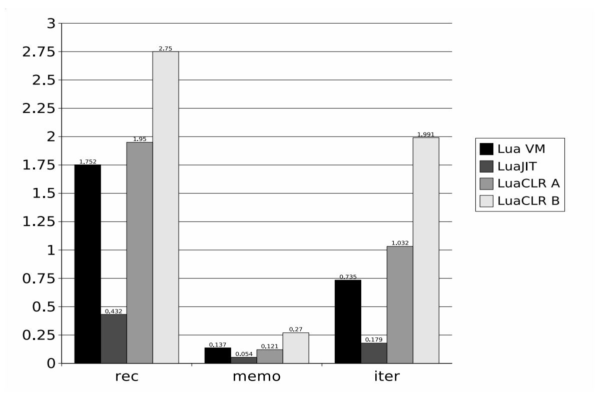

Figure 2: Benchmark results after single return optimization.

first benchmarks showed that the code our compiler generated was slower than the same code executed by the interpreter for function calls and arithmetic, even though it is actually JIT-compiled to native code by the CLR. This means that the overhead of fitting Lua code in the CLR's type and runtime system is bigger than the overhead of interpretation of the same code by an efficient VM. In the next section we will show how we improved these results with a few optimizations.

3. COMPILER OPTIMIZATIONS

In the previous section we described a simple Lua compiler for the CLR, and showed that its performance was worse than the Lua interpreter's even though CLR code is JIT-compiled to native code. Now we will show our optimizations to the compiler and the impact they had on performance.

First we optimized function returns. Lua functions can return multiple values (and the language supports multiple assignment). In our first compiler all functions returned an array of values, even if we were only interested in the first one (or we just discarded all of them). This meant that every function call that returned values needed to allocate and fill an array of `Lua.Value` instances.

The optimization we did was to treat single returns as a special case, by generating new overloaded versions of `Invoke` that returned a single `Lua.Value` instance instead of an array of them. Code that discards return values, or just needs one of them (a statically verifiable property of the code), could call one of these versions, while all other code could still call one of the versions that returned arrays.

We then had ten new versions of `Invoke`, one for each number of parameters from zero to eight and one for an arbitrary number of parameters. When a function was compiled, the version that matched the number of parameters of the function held the function's compiled code, while the other versions delegated to that one. The optimization approximately doubled the size of the compiled code, as we previously had ten versions of `Invoke` and now had twenty with similar code, but we were not concerned in optimizing for code size.

We implemented this optimization and ran our benchmarks on the second compiler, and the results are on Figure 2. *LuaCLR A* is the simple compiler and *LuaCLR B* is the second compiler, the one with the optimization for single returns. The results showed that the performance of the single return code was worse than the performance of the regular code, so this was not an optimization at all.

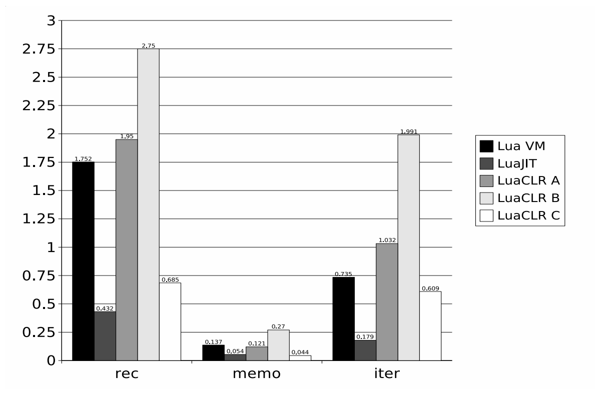

Figure 3: Benchmark results after new type mapping.

The results puzzled us at first, but further testing indicated that its behavior happened because of the CLR optimizations on native code (and the lack thereof). Our code made heavy use of value types (the `Lua.Value` structures), and the single return code made even heavier use of them; while functions in the first compiler returned a reference, single return functions returned a structure. The CLR did not optimize this passing around of structures, but it did optimize the allocation and passing of arrays. This is not a strange engineering decision, given that most of the CLR code involves allocating and using reference types, such as instantiating objects and passing them as arguments and return values.

This behavior of the CLR pointed to another, more radical optimization of our compiler: we dropped the `Lua.Value` type in favor or dealing directly with numbers and `Lua.Reference` instances. Instead of dealing with `Lua.Value` instances we dealt with `object` instances which are either boxed numbers or `Lua.Reference` instances. Prior to each operation we did a type test and then unboxed or downcast depending on the result. By doing this we could drop the use of structures in our code, and could enjoy the results of the CLR optimizations for allocating and using reference types.

We ran the benchmarks on our third compiler, and the results are on Figure 3, where *LuaCLR C* is the third compiler, the one that does not use structures (but still treats single returns specially). The results showed a great improvement over the previous two compilers, and for the first time Lua code running under the CLR consistently beat the same code running under the Lua interpreter, and

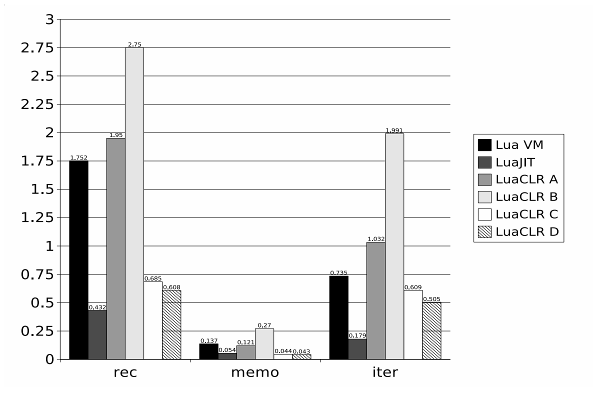

Figure 4: Final benchmark results.

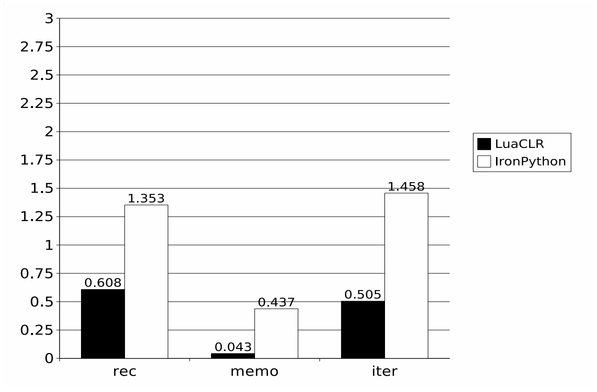

Figure 5: LuaCLR vs IronPython benchmark.

even approached code directly compiled to native code by LuaJIT. Yet the compiler was still simple in its implementation.

Our final optimizations were more incremental, building on the third compiler. First we changed the way we were doing the type tests to a way that was marginally more efficient (although it used three CIL opcodes instead of one). Then we reimplemented upvalues as instances of a `Lua.UpValue` class (with the value stored in a field) instead of a single-element array. This let the CLR JIT drop bounds-checking on upvalue accesses.

We ran our benchmarks on the last compiler, and the final results are on Figure 4, on column *LuaCLR D*. The fourth compiler's optimizations gave a modest improvement over our third compiler, and we believe this is as far as we can go without employing dynamic type feedback and type inference and thus making the compiler much more complex.

In the next section we present other work on running dynamic languages on the CLR, and how these other efforts relate to our work.

4. RELATED WORK

From 1999 to 2002 Microsoft sponsored the development of several compilers for the CLR under the banner of Project 7. Among these was a compiler for the Python scripting language by Mark Hammond and Greg Stein. The compiler supported most of the Python language and allowed Python programs to interface with CLR code, but the authors judged the performance to be "so low as to render the current implementation useless for anything beyond demonstration purposes", applying to "both the compiler itself, and the code generated by the compiler" [2]. The effort was abandoned sometime in 2002.

Python for .NET used a type mapping similar to the one we used on our first compiler, with a CLR structure representing Python values, but the operations were inefficient, and noted that "simple arithmetic expressions take hundreds or thousands of Intermediate Language instructions". The Python for .NET runtime was responsible for all operations, the compiler did not inline any cases.

Lua2IL was a previous attempt at running Lua code inside the CLR, and worked by translating Lua 5.0 bytecodes to CIL instructions, with the help of a support runtime [9]. Our first compiler used the same mapping from Lua types to CLR types as Lua2IL's, with a structure holding either a Lua number of a reference to other Lua types, and the other types represented by CLR classes with a common subclass. Lua2IL also inlined operations on numbers.

Our compilers differ from Lua2IL on the treatment of local variables, function arguments and upvalues. We used the CLR stack as much as possible, while Lua2IL kept a parallel Lua stack in the CLR heap and threaded this stack through the compiled Lua code. Code generated by Lua2IL had similar performance to the same code when executed by the Lua 5.0.2 interpreter.

The research prototype of IronPython, another Python compiler for the CLR, showed that Python can have good performance in the CLR if the compiler is designed with careful consideration to performance [4]. IronPython boxes numbers and casts other types to `object`, like our third compiler, and also inlines common operations and uses the CLR stack when possible.

Recent work on IronPython has focused on generalizing its runtime to other dynamic languages, building a "Dynamic Language Runtime" on top of the CLR [5]. The DLR uses Self-like dynamic type feedback to implement operations [3], and the DLR runtime system generates CIL code that makes heavy use of method calls, relying on the CLR JIT for inlining and optimization. The DLR developers call this use of type feedback "dynamic sites", and they optimize from simple arithmetics to method calls of external CLR objects. The resulting code and runtime system is much more complex than our Lua compilers, though.

Currently the DLR supports Python, Ruby and JavaScript, and the set of the DLR features is biased towards these languages. In particular there is no native support for multiple return values from functions, as these three languages fake these with tuples or arrays, as well as no tail call optimization. Lua's semantics require the latter, and efficient implementation of Lua function calls requires the former. Another issue is the DLR's implementation of lexical scoping and varargs, which revert to building stack frames in the heap instead of using the CLR stack.

We compared the results of our last compiler's benchmark with similar Python code (using a straightforward translation) compiled by IronPython 2.0 Alpha 4, the latest version of IronPython that uses the DLR as of the writing of this paper. The results are on Figure 5. They show that a simple compiler, if tailored and optimized to the semantics of a single language, can perform better than one using more complex and general optimizations.

The DLR may be the an answer to Microsoft's problem of developing compilers to three dynamic languages at the same time, but we are not convinced that it's the best answer to the general problem of developing efficient compilers for dynamic languages on the CLR.

5. CONCLUSIONS

This paper presented a series of compilers that compile Lua programs to the Common Language Runtime. Lua is a dynamic language, so there is a mismatch between its type system and execution model and the CLR's statically typed type system. Our goal was to enable efficient execution of Lua code despite this mismatch.

We presented four different compilers with different type mappings and optimizations, but all using the CLR stack for parameter passing and local variables, and inlining common operations. The first compiler did a simple mapping from Lua types to a parallel hierarchy of CLR types, and was slower than the Lua interpreter by a small margin. The second compiler adds a specialized code path for function calls that don't need to return more than one value, but this actually made the resulting code slower.

The use of value types by the two compilers was impacting performance. The second compiler used value types even more than the first, so was slower despite being more specialized. The third compiler changed the type mapping to use only reference types (and boxing numbers) instead of CLR value types, trusting that the CLR JIT compiler would be able to apply more optimizations to the code. The result is approximately a twofold speedup compared to the performance of the first compiler, and beating the Lua interpreter. Finally, we did some tuning on the code generated by a very common operations, and had a little increase on performance over the third compiler, putting the performance closer to that achieved by Lua code compiled to native code by LuaJIT.

Our results show that optimizing in the presence of another existing optimizer (in our case the CLR JIT) is hard, specially in the absence of a clear and detailed performance model. Any optimization needs to be benchmarked, and all assumptions reevaluated as the platform evolves.

We also compared the performance of our compilers with the performance of the latest version of IronPython, the version that uses Microsoft's new Dynamic Language Runtime, a layer on top of the CLR that Microsoft is using to build their new Python, Ruby and JavaScript compilers. Our Lua compiler generated faster code than the similar Python code compiled by IronPython, as we generate simpler code that is tailored to Lua's semantics. We also found that the DLR's semantics are generic enough to emulate Python, Ruby and JavaScript semantics, but still not enough to efficiently emulate Lua's semantics.

Our current priority in this work is to bring our Lua compiler to production quality, with a full reimplementation of the Lua standard library (which is implemented in C on the Lua interpreter). Further performance improvements will need more radical optimizations; we want to investigate how type inference and dynamic type feedback can help in this matter, the first reducing the need for type checks and the second adding more specialized code paths to the ones we already use.

Another open issue is how to best implement Lua's coroutines. Our current implementation uses threads and synchronization using semaphores to implement coroutines, but this limits the number of coroutines a program can use (as CLR threads are OS-level threads), and require an OS context switch each time a coroutine suspends or resumes.

We can use the way of implementing first-class continuations on the CLR presented by Pettyjohn et al [12] to implement coroutines without using threads, but the cost is a full stack traversal each time a coroutine suspends (as that point's continuation has to be captured and stored). We need to evaluate how this compares to a context switch.

Another approach we are considering to implement coroutines is to compile a CPS-transformed version of each Lua function, and suspending a coroutine then is a simple return. The CPS code wouldn't use the CLR stack, storing variables and control information in the heap. Code running inside a coroutine runs slower in this approach, though, and a coroutine can't have calls to CLR methods in the call stack when it suspends.

6. ACKNOWLEDGMENTS

The authors would like to thank Microsoft Research for supporting this work through the Phoenix and SSCLI 2005 Request for Proposals.

7. REFERENCES

[1] A. L. de Moura, N. Rodriguez, and R. Ierusalimschy. Coroutines in lua. *Journal of Universal Computer Science*, 10(7):910–925, 2004.

[2] M. Hammond. Python for .NET: Lessons Learned, 2000. Technical report by ActiveState, not available online at the time of this paper's writing.

[3] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 326–336, New York, NY, USA, 1994. ACM Press.

[4] J. Hugunin. Ironpython: A fast python implementation for .net and mono. In *Proceedings of PyCon DC 2004*, 2004. Available at `http://www.python.org/pycon/dc2004/papers/9`.

[5] J. Hugunin. A Dynamic Language Runtime (DLR), 2007. Available at `http://blogs.msdn.com/hugunin/archive/2007/04/30/a-dynamic-language-runtime-dlr.aspx`.

[6] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of lua. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1–2–26, New York, NY, USA, 2007. ACM Press.

[7] R. Ierusalimschy, L. H. Figueiredo, and W. Celes. Lua 5.1 Reference Manual. Technical Report 14/03, PUC-Rio, 2006. Available at `http://www.lua.org/manual/5.1`.

[8] F. Mascarenhas and R. Ierusalimschy. Luainterface: Scripting the .net clr with lua. *Journal of Universal Computer Science*, 10(7):892–909, 2004.

[9] F. Mascarenhas and R. Ierusalimschy. Running lua scripts on the clr through bytecode translation. *Journal of Universal Computer Science*, 11(7):1275–1290, 2005.

[10] Microsoft. ECMA C# and Common Language Infrastructure Standards, 2005. Available at `http://msdn.microsoft.com/net/ecma/`.

[11] M. Pall. The LuaJIT project, 2007. Available at `http://luajit.org/`.

[12] G. Pettyjohn, J. Clements, J. Marshall, S. Krishnamurthi, and M. Felleisen. Continuations from generalized stack inspection. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 216–227, New York, NY, USA, 2005. ACM Press.