1
Introduction

Managed runtime environments have become popular targets for compilers of high-level programming languages. Reasons for adoption of these runtimes include a safe execution environment for foreign code, easier interoperability, and their existing libraries. These managed runtimes provide a high-level type system with enforced runtime safety, as well as facilities such as garbage collection, possibly sandboxed access to services of the underlying platform, multithreading, and a rich library of data structures and algorithms. Examples of managed runtimes include Microsoft's Common Language Runtime [Microsoft, 2005], the Java Virtual Machine [Lindholm and Yellin, 1999], and more recently the JavaScript runtimes present in web browsers [ECMA, 1999, Manolescu et al., 2008].

As these runtimes are higher level than the usual compiler targets such as machine languages and intermediate languages close to the hardware, they inevitably lead to an impedance mismatch between the semantics of the language that is being compiled and the semantics of the target runtime, which translates to inefficiency in the generated code, changes in the language, or both. The lack of a clear performance model for these runtimes, which can have great variation even among different implementations of a particular runtime, also hinder attempts at optimizing the generated code of any language that does not have a direct mapping to the semantics of the runtime. Writing an optimizing compiler for these runtimes has to involve guesswork and experimentation.

The problem of efficient compilation is worse when compiling dynamically-typed languages to statically typed runtimes, such as the CLR and JVM. In this case, all of the operations of the language have to be compiled using runtime type checks or using the virtual dispatch mechanism of the runtime. If the language is object-oriented then it cannot use the native method dispatch mechanism of the runtime, and has to implement its own. Implementing arithmetic operations is particularly troublesome, as the runtimes usually do not have tagged unions value types, so numbers have to be boxed inside heap-allocated objects and treated as references.

Compiling to a dynamically-typed runtime is also problematic unless the semantics of the types and operations of the source language exactly match the semantics of corresponding types and operations of the target. This kind of semantic match is very rare among high-level languages. In practice, some form of wrapping and runtime checking, or even more radical program transformations, such as trampolines for tail call optimization, are still necessary.

Compiling dynamically-typed languages to machine language or low-level languages also needs runtime type checks and dynamic dispatch, but the low-level nature of the target language means these operations are more efficient than their equivalent on managed runtimes and their performance characteristics are better understood. When compiling to a machine language you have a performance model, the performance model of the target processor, that is missing in the intermediate languages of the managed runtimes.

Nevertheless, we assert that it is possible to generate efficient code from a dynamically-typed source language to a managed runtime. By efficient we mean at least as fast as the same code executed by a good native interpreter for the source language and, in a modern managed runtime with a good optimizing JIT compiler, matching or exceeding the performance of code generated by a good optimizing compiler for the source language. We support our assertion by implementing an optimizing compiler for the Lua programming language that targets the Microsoft Common Language Runtime and benchmarking this compiler against the Lua interpreter and an optimizing Lua JIT compiler.

Lua is a dynamically-typed language that has relatively simple semantics and a very efficient[1] interpreter implemented in C [Ierusalimschy, 2006]. Lua has some advanced features such as extensible semantics, anonymous functions with full lexical scoping (similar to the lexical scoping present in the Scheme language), tail call optimization, and full asymmetric coroutines [Ierusalimschy et al., 2007, de Moura et al., 2004]. It has a simple type system: nil, floating point numbers, immutable strings, booleans, functions, and tables (associative arrays), the latter having syntactical and semantic support for use as numeric arrays, as user-defined data types, and as a way to implement object orientation using prototypes or classes. Section 1.1 is a brief primer on the language.

Our approach for creating a compiler for the Lua programming language was to start with a very simple compiler, and a suite of small to medium length benchmarks that include Lua idioms used by actual programs. We then built variations of this first compiler, validating optimizations against this benchmark suite. This is a continuation of previous work we published

[1]Compared to other interpreters for dynamically-typed languages.

in Mascarenhas and Ierusalimschy [2008].

Benchmarking was an essential part of our approach; the CLR intermediate code that our compilers generate passes through the CLR optimizers on conversion to native code, and we could not know beforehand how a particular piece of CLR code would perform. This makes it very difficult to accurately assess the impact of even simple changes in the compiler. In the worst case, what we can think is an optimization may in fact make real programs run slower. Benchmarks helped assess the impact of our changes, and anomalies found in the results of some of the benchmarks are evidence of the unpredictability introduced by the lack of a performance model. A benchmark suite is also useful for programmers, being a source of tips on how to write efficient code for the compiler.

Our simplest compilers did not use static program analysis, so the scope of optimizations they implemented was limited to what was possible with extremely local information. The compilers had different mappings of Lua types to CLR types, such as using value types versus using boxing and reference types and interning strings. The compilers also had different ways to implement the return of multiple values from a function call.

Our more advanced compilers used type inference to extract more static information out of Lua programs. Having more information let us generate better code. For example, if we can statically determine the runtime types of each variable and parameter then we can avoid boxing and type checking for every variable that we are certain can only hold numbers. Sufficiently precise type information let the compiler synthesize CLR classes for Lua tables, transforming what was a hash lookup in the simpler compilers to a simple address lookup.

We did both local and interprocedural type inference. The local inference does not cross function boundaries and is much simpler to implement, but the information it obtains is very imprecise. Languages that use local type inference use explicit type annotation of function arguments to get more precise results. Interprocedural type inference was harder to specify and implement, but yielded much better results in our benchmarks. The basic problem of an interprocedural type inference of Lua programs is the same as of other languages with first-class functions: which functions are callable at each call site is initially unknown, and has to be found during inference. To know which function is callable at a call site you need the type of the variable referencing the function, or the expression that yields it.

Type inference is a complex algorithm that can subtly alter the behavior of a program if specified and implemented incorrectly. We did a formal specification of our typing rules and inference algorithm with regards to an operational semantics of a subset of Lua to make the specification of our type inference more precise and easier to understand. The actual inference algorithm works on the full Lua language, but leaves parts of a program outside of this subset dynamically typed.

Related work on implementing compilers for dynamic typed languages that target managed runtimes uses approaches that are based on runtime optimization using the code generation and dynamic code loading facilities of the managed runtimes. These approaches adapt the concept of polymorphic inline caches [Deutsch and Schiffman, 1984, Hölzle and Ungar, 1994], while our approach uses compile-time optimization via static analysis. Examples of the former include Microsoft's Dynamic Language Runtime for the CLR [Chiles and Turner, 2009] and the invokedynamic opcode for the JVM [Sun Microsystems, 2008]. The implementation of approaches such as the DLR and invokedynamic is very complex, and their performance characteristics are even more opaque than the performance characteristics of the underlying runtime, due to the extra level of indirection in the compilation.

The following sections are a small primer on the Lua language and the CLR. The rest of the dissertation is organized as follows: Chapter 2 presents our basic Lua compiler and the variations that do not depend on interprocedural type inference. Chapter 3 is a presentation of our type inference algorithm for the Lua language, and how we used it in our Lua compiler; Chapters 2 and 3 also discuss related work. Chapter 4 presents our benchmark suite, our benchmark results and the analysis of these results. Finally, Chapter 5 states our conclusions and outlines possible future work.

1.1 A Lua Primer

Lua [Ierusalimschy, 2006] is a scripting language designed to be easily embedded in applications, and used as a configuration and extension language. Lua has a simple syntax, and combines common characteristics of imperative languages, such as loops, assignment, and mutable data structures, with features such as associative arrays, first-class functions, lexical scoping, and coroutines.

Lua is dynamically typed, and has eight types: `nil`, *number*, *string*, *boolean*, *table*, *function*, *userdata*, and *thread*. The *nil* type has a single value, `nil`, and represents an uninitialized or invalid reference; values of type *string* are immutable character strings; the *boolean* type has two values, `true` and `false`, but every value except `false` and `nil` has a true boolean value for the purposes of tests in the language; values of type *table* are associative arrays; the *userdata* type is the type of external values from the application that is embedding Lua, and *thread* is the type of coroutines.

Table indexes can be any value except `nil`. Lua also has syntactic sugar for using tables as records and objects. The expression `tab.field` is syntactic sugar for `tab["field"]`. Lua functions are first-class values; storing functions in tables is the base of Lua's support for object-oriented programming. The expression `tab:method(x)` is a method call and is syntactical sugar for `tab.method(tab, x)`. This syntactic sugar also works for defining methods, as in the fragment below:

```
function tab:method(x)
  -- method body
end
```

The fragment above is the same as the following one:

```
tab.method = function (self, x)
  -- method body
end
```

The behavior of Lua values can be extended using *metatables*. A metatable is a table with functions that modify the behavior of the table or the type it is attached to. Each table can have an attached metatable, but for the other types there can only be one metatable per type. A common use of metatables is to implement single inheritance for objects, as in the following fragment:

```
local obj = setmetatable({ a = 0 },
  { __index = parent })
function parent:method(x)
  self.a = self.a + x
end
obj:method(2)
print(obj.a) -- 2
```

In the fragment above, whenever the code tries to read a field from `obj` and this field does not exist, Lua looks it up in the `__index` field of the metatable, so the value of this field works as the parent object. Other metatable fields modify operations such as assignment to a field, arithmetic operations, and comparisons.

First-class functions, lexical scoping, and imperative assignment interact in the same way as they do in Scheme [Steele, 1978, Adams et al., 1998].

The following fragment creates a counter and returns two functions, one to increment and one to decrement the counter. Both functions share the same variable in the enclosing lexical scope, and each pair of functions returned by make_counter shares a `counter` variable distinct from the one of the other pairs:

```
function make_counter()
  local counter = 0
  return function ()
           counter = counter + 1
           return counter
         end,
         function ()
           counter = counter - 1
           return counter
         end
end
```

The fragment above also shows how Lua functions can return more than one value. In most function calls Lua just takes the first returned value and discards the others, but if the function call is part of a multiple assignment, as in `inc, dec = make_counter()`, or if you are composing functions, as in `use_counter(make_counter())`, then Lua will use the extra values.

A function call that has more or less arguments the the arity of the function is legal in Lua. Extra arguments are simply ignored, and missing arguments have the value `nil`.

1.2 The Common Language Runtime

The CLR [Microsoft, 2005] is a managed runtime created to be a common target platform for different programming languages, with the goal to make it easier for those languages to interoperate. It has an intermediate language and shared execution environment with resources such as a garbage-collected heap, threading, a library of common data structures, and a security system with code signing. The CLR also has an object-oriented type system augmented with parametric types with support for reflection and tagging of types with metadata [Yu et al., 2004].

CLR types can be *value types* or *reference types*. Value types are the primitive types (numbers and booleans) and structures; assignment of value types copies the value. Reference types are classes, interfaces, delegates, and arrays. Assignment of reference types copies the reference to the value, and the value is kept in the heap. The CLR has a single-inheritance object system, rooted in the `object` type, but classes can implement *interfaces* which are types that only have abstract methods. *Delegates* are typed function pointers. Each value type has a corresponding reference type, used for *boxing* the value type in the heap.

The CLR execution engine is a stack-based intermediate language with about 200 opcodes, called Common Intermediate Language, abbreviated as CIL or just IL. The basic unit of execution is the method; each method has an activation record kept in an execution stack. The activation record has the local variables and arguments of the method being executed, metadata about the method, and the evaluation stack of the execution engine. IL opcodes implement operations such as transferring values between local variables, arguments, or fields and the evaluation stack, creation of new objects, method calls, arithmetic, and branching.