5
Conclusions

Writing an optimizing compiler for a managed runtime involves guesswork and experimentation. Instead of targeting low-level machine code with a clear performance model we are targeting a high-level language with its own type system, runtime library, and optimizing Just-In-Time compiler. Not only it is difficult to predict how a particular approach will perform, but the performance can vary among different implementations of the managed runtime, or even different versions of the same implementation.

We have shown the difficulty in compiling to a managed runtime by building a series of compilers for the Lua programming language that targets the Common Language Runtime. We built several compilers with different ways to represent Lua types in the CLR type system and different ways to compile Lua operations, and then benchmarked these compilers on different implementations of the CLR. Our benchmarks show how the best approach for compiling Lua to the CLR depends on what implementation of the CLR we are targeting.

The choice of implementation approach and implementation target influences not only the performance of our Lua implementation but also its semantics, as tail call optimization does not work for some combinations of implementations of our compiler and implementations of the CLR, even though it should have worked by the CLR standard. There are other corners of Lua's semantics that are problematic to implement in the CLR: weak tables, finalizers, and coroutines, but we already covered these in Mascarenhas and Ierusalimschy [2005], so we focused on the efficient implementation of Lua's core semantics for the Lua compilers of this dissertation.

The influence of the implementation target on the semantics of Lua has parallels to the work on definitional interpreters in Reynolds [1998], where the closer the semantics of the language that you want to interpret is to the semantics of the language you are using to write the interpreter the simpler the interpreter can be. Where the semantics differ you have to implement the correct semantics in terms of the underlying language. With compilers for high-level targets such as managed runtime environments, the closer the

semantics of the language you are compiling to the semantics of the runtime the more direct the compilation, and where the semantics differ you need extra scaffolding and support to implement the correct semantics.

Lua is a dynamically typed language, and the CLR is a statically typed runtime, so in most of our compilers all of Lua's operations had to be compiled using runtime type checks and the virtual dispatch mechanism of the CLR. We also had to use a unified representation for Lua values that either wasted memory and interacted in a less than optimal way with the JIT of one of the CLR implementations we tested, or had to store all numbers in boxes in the heap instead of using the efficient native representation for floating-point numbers that the CLR has.

We specified a type system and type inference algorithm for Lua that can statically assign more precise types to several kinds of Lua operations. We implemented this algorithm in one of our compilers, and used its output to generate more efficient representations for Lua values and better performing code. Analysis of the output of the type inference algorithm and the performance gains showed that the type inference algorithm correctly infers precise types for most variables and operations in our benchmarks.

Compared to other Lua implementations, our best combination of Lua compiler without type inference and CLR implementation has the level of performance of version 5.1.4 of the Lua interpreter, being worse by a factor of less than two in benchmarks that are heavily dependent on floating point computation, and faster by a little over a factor of two in benchmarks that are heavily dependent on recursion.

With type inference, our best combination of Lua compiler and CLR implementation outperforms the Lua interpreter and performs better than version 1.1.5 of the LuaJIT compiler for most benchmarks. Our results show that it is possible to get good performance out of a dynamic language in a managed runtime if the managed runtime has a good implementation.

We also benchmarked our Lua compilers against IronPython 2.0, a Python compiler for the CLR that uses a different implementation approach based on runtime generation of specialized code, in contrast to our simpler approach that only uses offline compilation. Without type inference our best compiler performs equal or better than IronPython in almost all benchmarks; with type inference our best compiler outperforms IronPython by a large margin in all benchmarks. We believe our approach is the best one for compiling Lua on the CLR given the current state of the Dynamic Language Runtime that IronPython is built on.

Future implementations of the CLR may change the impact of some of

our implementation decisions, so the specific results may change, but this only restates our general thesis that the optimal implementation approach depends on the specific implementations that are the targets.

Type inference was the key to the optimizations with the most impact on performance, and this suggests directions for future research. Our type inference algorithm works just as badly across module boundaries as the local type propagation of Section 2.2.4 works across function call boundaries. Parameters of exported functions in our type system always have the dynamic type, and imported functions also always return values of this type.

Recent work on *gradual typing* [Siek and Taha, 2006, 2007] and *Typed Scheme* [Tobin-Hochstadt and Felleisen, 2008] may lead to an approach combining type annotations in module boundaries with type inference used for intra-module optimization. Gradual typing is a type system where parts of a program may be annotated with precise types, and parts not annotated have a dynamic type. The type system guarantees that type errors only occur in the dynamically-typed portions of the program. Typed Scheme is a gradual typing system for Scheme that lets the programmer mix statically and dynamically typed Scheme code. The Typed Scheme runtime is still the same runtime as regular Scheme; the type annotations provide static type safety, not increased performance through removal of type checks or better representations.

Gradual typing may be orthogonal to the type system we use for our type inference algorithm. Siek and Vachharajani [2008] has already show that gradual typing is compatible with Hindley-Milner type inference, and Herman et al. [2007] use techniques taken from Heinglein's work on dynamic typing that we reviewed in Section 3.3.1 [Henglein, 1992a]. Combing gradual typing and our type inference should make it possible to have inference working across modules with minimal type annotations, as well as increasing the static type safety of Lua programs.