1
Introduction

Dynamically typed languages such as Lua avoid static types in favor of simplicity and flexibility, because the absence of static types means that programmers do not need to bother with abstracting types that should be validated by a type checker. Instead, dynamically typed languages use run-time *type tags* to classify the values they compute, so their implementation can use these tags to perform run-time (or dynamic) type checking [Pie02].

This simplicity and flexibility allows programmers to write code that might require a complex type system to statically type check, though it may also hide bugs that will be caught only after deployment if programmers do not properly test their code. In contrast, static type checking helps programmers detect many bugs during the development phase. Static types also provide a conceptual framework that helps programmers define modules and interfaces that can be combined to structure the development of programs.

Thus, early error detection and better program structure are two advantages of static type checking that can lead programmers to migrate their code from a dynamically typed to a statically typed language, when their simple scripts evolve into complex programs [THF06]. Dynamically typed languages certainly help programmers during the beginning of a project, because their simplicity and flexibility allows quick development and makes it easier to change code according to changing requirements. However, programmers tend to migrate from dynamically typed to statically typed code as soon as the project has consolidated its requirements, because the robustness of static types helps programmers link requirements to abstractions. This migration usually involves different languages that have different syntaxes and semantics, which usually requires a complete rewrite of existing programs instead of incremental evolution from dynamic to static types.

Ideally, programming languages should offer programmers the option to choose between static and dynamic typing: *optional type systems* [Bra04] and *gradual typing* [ST06] are two similar approaches for blending static and dynamic typing in the same language. The aim of both approaches is to offer programmers the option to use type annotations where static typing is

needed, allowing the incremental migration from dynamic to static typing. The difference between these two approaches is the way they treat run-time semantics. While optional type systems do not affect run-time semantics, gradual typing uses run-time checks to ensure that dynamically typed code does not violate the invariants of statically typed code.

Programmers and researchers sometimes use the term *gradual typing* to mean the incremental evolution of dynamically typed code into statically typed code. For this reason, gradual typing may also refer to optional type systems and other approaches that blend static and dynamic typing to help programmers incrementally migrate from dynamic to static typing without having to switch to a different language, though all these approaches differ in the way they handle static and dynamic typing together. We use the term *gradual typing* to refer to the work of Siek and Taha [ST06].

In this work we present the design and evaluation of Typed Lua: an optional type system for Lua that is rich enough to preserve some of the Lua idioms that programmers are already familiar with, but that also includes new constructs that help programmers structure Lua programs.

Lua is a small imperative language with first-class functions (with proper lexical scoping) where the only data structure mechanism is the *table* – an associative array that can represent arrays, records, maps, modules, objects, etc. Tables also have syntactic sugar and metaprogramming support through operator overloading built into the language. Unlike other scripting languages, Lua has very limited coercion among different data types.

Lua prefers to provide mechanisms instead of fixed policies due to its primary use as an embedded language for configuration and extension of other applications. This means that even features such as a module system and object orientation are a matter of convention instead of default language constructs. The result is a fragmented ecosystem of libraries, and different ideas among Lua programmers on how they should use the language features, or how they should structure programs.

The lack of standard policies is a challenge for the design of an optional type system for Lua. For this reason, we are not relying entirely on the semantics of the language to design our type system. We also run a mostly automated survey of Lua idioms used in a large corpus of Lua libraries, which also has helped in the design of Typed Lua.

So far, Typed Lua is a Lua extension that allows statically typed code to coexist and interact with dynamically typed code through optional type annotations. In addition, it adds default constructs that programmers can use to better structure Lua programs. The Typed Lua compiler warns

programmers about type errors, but always generates Lua code that runs in unmodified Lua implementations. Programmers can enjoy some of the benefits of static types even without converting existing Lua modules to Typed Lua – they can export a statically typed interface to a dynamically typed module, and statically typed users of the module can use the Typed Lua compiler to check their use of the module. Thus, implementing an optional type system for Lua offers Lua programmers one way to obtain most of the advantages of static typing without compromising the simplicity and flexibility of dynamic typing. We have an implementation of the Typed Lua compiler that is available online[1].

Typed Lua's intended use is as an application language, and we believe that policies for organizing a program in modules and writing object-oriented programs should be part of the language and checked by its optional type system. An application language is a programming language that helps programmers develop applications from scratch until these applications evolve into complex systems rather than just scripts. We will show that Typed Lua introduces the refinement of tables to support the common idioms that Lua programmers use to encode both modules and objects.

We also believe that Typed Lua helps programmers give more formal documentation to already existing Lua code, as static types are also a useful source of documentation in languages that provide type annotations, because type annotations are always validated by the type checker and therefore never get outdated. Thus, programmers can use Typed Lua to define axioms about the interfaces and types of dynamically typed modules. We enforce this point by using Typed Lua to statically type the interface of the Lua standard library and other commonly used Lua libraries, so our compiler can check Typed Lua code that uses these libraries.

Typed Lua performs a very limited form of local type inference [PT00], as static typing does not necessarily mean that programmers need to insert type annotations in the code. Several statically typed languages such as Haskell provide some amount of type inference that automatically deduces the types of expressions. Still, Typed Lua only requires a small amount of type annotations due to the nature of its optional type system.

Typed Lua does not deal with code optimization, although another important advantage of static types is that they help the compiler perform optimizations and generate more efficient code. However, we believe that the formalization of our optional type system is precise enough to aid optimization in some Lua implementations.

[1] https://github.com/andremm/typedlua

We use some of the ideas of gradual typing to formalize Typed Lua. Even though Typed Lua is an optional type system and thus does not include run-time checks between dynamic and static regions of the code, we believe that using the foundations of gradual typing to formalize our optional type system will allow us to include run-time checks in the future.

Finally, we believe that designing an optional type system for Lua may shed some light on optional type systems for scripting languages in general, as Lua is a small scripting language that shares some features with other scripting languages such as JavaScript.

This work is split into seven chapters. In Chapter 2 we review the literature about blending static and dynamic typing in the same language, we discuss the differences between optional type systems and gradual typing, and we also present the results of our survey on Lua idioms. In Chapter 3 we use code examples to present the design of Typed Lua. In Chapter 4 we present our type system. In Chapter 5 we discuss the evaluation results that we obtained while using Typed Lua to type existing Lua code. In Chapter 6 we present some related work. In Chapter 7 we outline our contributions.