7
Conclusions

In this work we presented Typed Lua, an optional type system for Lua. We implemented Typed Lua as a Lua extension that allows programmers to combine static and dynamic typing in Lua code, making easier the evolution of simple scripts into large programs.

Our main contribution is the formalization of a complete optional type system that introduces several novel type system features to statically type check Lua programs. Even though Lua shares several features with other dynamically typed languages such as JavaScript, Lua also has several unusual features. These unusual features include tables (or associative arrays) as the sole mechanism for structured data, besides functions with multiple return values and flexible arity that interact with multiple assignment. We highlight the following novel features of our type system:

- type refinement allows the incremental evolution of record and object types, playing an important role in statically type checking the idiomatic way in which Lua programmers use tables to define modules and objects;
- projection types handle functions that are overloaded on the number and types of return values, allowing programmers to narrow the types of a set of variables by narrowing the type of a single component of this set;
- union types and variadic types help our type system handle functions with flexible arity, that is, union types are helpful in describing optional parameters while variadic types are helpful in describing the type of the vararg expression and the type of functions that can receive or return any number of values.

A key feature in optional type systems is usability. This means that optional type systems should not change the idioms that programmers are already familiar with. Instead, optional type systems should fit existing idioms to statically type check them. Designing a too simple type system can overload programmers by forcing them to change the way they program in the language to fit the type system, while designing a too complex type system can overload

programmers with types and error messages that are hard to understand, even if type inference removes the necessity of annotating the program with these complex types. The most challenging aspect of designing optional type systems is to find the right amount of complexity for a type system that feels natural to the programmers.

Usability has been a concern in the design of Typed Lua since the beginning. We realized that we should not rely on the semantics of Lua only, as this could lead to a cumbersome type system that would not support several Lua idioms. For this reason, we performed a mostly automated survey of Lua idioms and features to inform our design choices.

After designing and implementing Typed Lua, we performed several case studies to evaluate how successful we were in our goal of providing an usable type system. We evaluated 29 modules from 8 different case studies, and we could give precise static types to 83% of the 449 members that these modules export. For half of the modules, we could give precise static types to at least 89% of the members from each module. Our evaluation results showed that our type system can statically type check several Lua idioms and features, though the evaluation results also exposed several limitations of our type system. We found that the three main limitations of our type system are the lack of intersection types, parametric polymorphism, and operator overloading. Overcoming these limitations is our major target for future work, as it will allow us to statically type check more programs.

Unlike other optional type systems, we designed Typed Lua without deliberate unsound parts. However, we still do not have proofs that the novel features of our type system are sound. We see a soundness proof as another major future work, as it is necessary to use static types for code optimization.

Finally, we believe that Typed Lua is a major contribution to the Lua community, because it offers a framework that programmers can use to document, test, and better structure their applications. For libraries where a full conversion to static type checking should prove unfeasible or too much work, the community can use Typed Lua just to document the external interfaces of the libraries, giving the benefits of static type checking to the users of these libraries. In fact, we already have user feedback from Lua programmers that are using Typed Lua in their projects. For instance, ZeroBrane Studio is an IDE for Lua development that is evaluating the use of Typed Lua to perform static analysis in Lua code.