# 5
# Evaluation

We performed some case studies on existing Lua libraries to evaluate the design of our type system. For each library, we used Typed Lua to either annotate its modules or to write statically typed interfaces to its modules through Typed Lua's description files. In this chapter we present our evaluation results and discuss some interesting cases.

The Lua Standard Library [IdFC11] was our first case study. We started to think about how we would type its modules at the same time that we started to design our type system, as it could give us some hints on our type system. And it did: optional parameters and overloading on the return type are two Lua features that our type system should handle to allow us typing some of the functions that the standard library implements.

The second case study that we chose was the MD5 library [dF14], because we wanted a simple case study to introduce Typed Lua's description files and `userdata` declarations. These Typed Lua's mechanisms allow programmers to give statically typed interfaces to Lua libraries.

LuaSocket [Neh07] and LuaFileSystem [Kep04] were the third and fourth case studies that we used to evaluate Typed Lua. We chose them because they are the most popular Lua libraries. We wrote a script that builds the dependency graph of Lua libraries that are in the LuaRocks repository, and uses this dependency graph to identify the most popular Lua libraries.

We also randomly selected three case studies from the LuaRocks repository, they are: HTTP Digest [Cha14], Typical [Hoe12], and Mod 11 [Sch14]. The first provides client side HTTP digest authentication for Lua. The second is an extension to the primitive function `type`. The third is a generator and checker of modulo 11 numbers. We randomly selected three case studies because we wanted to evaluate Typed Lua for annotating existing libraries that are written in Lua, as the previous case studies are mostly libraries that are written in C.

The Typed Lua compiler is the last case study that we evaluated. We chose it as a case study because it is a large application. Besides, it is a case study that evaluates the evolution of a script to a program.

We used these case studies to evaluate two aspects of Typed Lua:

1. how precisely it can describe the type of the interface of a module;
2. whether it provides guarantees that the code matches the interface.

| | percentage of members | | | | |
|---|---|---|---|---|---|
| **Case study** | **easy** | **poly** | **over** | **hard** | **# members** |
| Lua Standard Library | 64% | 5% | 8% | 23% | 129 |
| MD5 | 100% | 0% | 0% | 0% | 13 |
| LuaSocket | 89% | 1% | 2% | 8% | 123 |
| LuaFileSystem | 89% | 0% | 11% | 0% | 19 |
| HTTP Digest | 0% | 0% | 100% | 0% | 1 |
| Typical | 100% | 0% | 0% | 0% | 1 |
| Modulo 11 | 78% | 0% | 0% | 22% | 9 |
| Typed Lua Compiler | 93% | 0% | 1% | 6% | 154 |

Table 5.1: Evaluation results for each case study



Table 5.1 summarizes our evaluation results for each of the case studies that we used Typed Lua for typing their members. An exported member is any Lua value that a module might export. We split the members of each case study into four categories: *easy*, *poly*, *over*, and *hard*. In the next four paragraphs we explain each category in more detail. The last column of the table shows the total number of members of each case study.

The *easy* category shows the percentage of members that we could give a precise static type. For instance, the function `string.len` from the Lua standard library is in this category because we could use Typed Lua to describe its type: `(string) -> (integer)`. This function returns the length of a given string. Note that the results that we obtained for this category give a lower bound on how much static type safety we could add to each one of our case studies.

The *poly* category shows the percentage of members that we made minimal use of the dynamic type, as a replacement for the lack of type parameters. For instance, the function `table.sort` from the Lua standard library is in this category because it is a generic function. It sorts a given list of elements, which is a generic list. However, we had to assign to this function the type `({any}, nil|(any, any) -> (boolean)) -> ()` because Typed Lua does not support parametric polymorphism. A better type for it would be `({<T>}, nil|(<T>, <T>) -> (boolean)) -> ()`.

The *over* category shows the percentage of members that require intersection types to describe their precise static types, as they are overloaded

functions. For instance, the function `math.abs` from the Lua standard library is in this category because it has two types: `(integer) -> (integer)` and `(number) -> (number)`. This function returns the absolute value of a given number, which can be either integer or float. Even though we gave this function the more general type `(number) -> (number)`, it is not precise enough because the return type is always `number` independently of the argument type. In other words, the return type should be `integer` when the argument type is also `integer`, and the return type should be `number` when the argument type is also `number`. However, we cannot give such a precise type to this function because Typed Lua does not support overloaded functions.

The *hard* category shows the percentage of members that do not fit in one of the previous categories, as they are difficult to type. For instance, the function `string.format` from the Lua standard library is in this category because it relies on format strings, which are difficult to type. Still, we gave this function the type `(string, value*) -> (string)`.

In the following sections we discuss each case study in more detail. For each case study, we split the evaluation results according to the modules that each one of them include. We use these split results to discuss the contributions and limitations of our type system.



## 5.1 Lua Standard Library

All of the modules in the standard library are implemented in C, so we used Typed Lua to type just the interface of each module. The `debug` module is the only one that we did not include in our evaluation results, because it provides several functions that violate basic assumptions about Lua code [IdFC11]. For instance, we can use the function `debug.setlocal` to change the value of a local variable that is not visible in the current scope. Table 5.2 summarizes the evaluation results for the Lua Standard Library (version 5.3).

The `base` module was very difficult to type because it includes several functions that rely on reflection, as the *hard* category shows. For instance, the functions `pairs` and `getmetatable` are in this category. While `pairs` traverses all keys and values that are stored in a given table, `getmetatable` returns the metatable of a given table.

There are some functions in the `base` module that we could not give a precise static type because our type system does not have parametric polymorphism, as the *poly* category shows. This is the case of `ipairs`.

The `base` module also includes some overloaded functions, as the *over* category shows. We could not type these functions because our type sys-

| | percentage of members | | | | |
|---|---|---|---|---|---|
| **Module** | **easy** | **poly** | **over** | **hard** | **# members** |
| base | 35% | 4% | 8% | 53% | 26 |
| coroutine | 14% | 0% | 0% | 86% | 7 |
| package | 62% | 0% | 0% | 38% | 8 |
| string | 75% | 0% | 0% | 25% | 16 |
| utf8 | 100% | 0% | 0% | 0% | 6 |
| table | 14% | 72% | 14% | 0% | 7 |
| math | 81% | 0% | 19% | 0% | 27 |
| io | 81% | 0% | 0% | 19% | 21 |
| os | 82% | 0% | 18% | 0% | 11 |

Table 5.2: Evaluation results for Lua Standard Library

tem does not include intersection types. This is the case of `tonumber` and `collectgarbage`.

In the case of `tonumber`, it has two different types: `(value) -> (number)` and `(string, integer) -> (number)`. This means that the type of the first parameter depends on the type of the second parameter. For instance, we can call `tonumber(1)`, but we cannot call `tonumber(1,2)`. Note that the first argument of `tonumber` can be a value of any type if it is the only argument, but it must be a string if there is a second argument, which must be an integer.

In the case of `collectgarbage`, the return type changes according to an input literal string. For instance, calling `collectgarbage("collect")` returns an integer, calling `collectgarbage("count")` returns a floating point, and calling `collectgarbage("isrunning")` returns a boolean.

The `coroutine` module was also very difficult to type because our type system cannot describe the computational effects of a program. The *hard* category shows the amount of functions that we could not give a precise static type for this reason. Lua has one-shot delimited continuations [JS11] in the form of *coroutines* [MI09], and effect systems [NN99] are an approach that we could use to describe control transfers with continuations. However, for now, coroutines are out of scope of our type system, and we use an empty `userdata` declaration to represent the type `thread`.

Still, we could give a precise static type to one function in the `coroutine` module, as the *easy* category shows. The function `coroutine.isyieldable` has no input parameters, and it simply returns a boolean that indicates whether the running coroutine is yieldable.

We could give precise static types to the constants and functions of the `package` module, but we could not give precise static types to the

following tables that it exports: `package.loaded`, `package.preload`, and `package.loaders`. The first stores loaded modules, while the others store module *loaders*. They are difficult to type because their types rely on reflection, that is, their types depend on the modules a program loads. For this reason, they are in the *hard* category.

We could give precise static types to most of the functions of the `string` module, but we could not give precise static types to the functions that rely on format strings. For instance, the type of the arguments that we pass to `string.format` must match the format string we are using. It is fine to call `string.format("%d", 1)`, but `string.format("%d", true)` raises a run-time error. These functions that rely on format strings are in the *hard* category.

The `utf8` module was straightforward to type, as its members are only operations over strings.

The `table` module was specially difficult to type because most of its functions require parametric polymorphism, as the *poly* category shows. These functions either receive or return a list of elements, and parametric polymorphism would help us to describe them with a generic type.

However, the lack of parametric polymorphism did not prevent us from giving a precise type to one function of the `table` module, as the *easy* category shows. We could give a precise static type to `table.concat`, as it operates over lists where all elements are strings or numbers.

Even if our type system had parametric polymorphism, there is still one function of the `table` module that we could not give a precise static type because it is an overloaded function, as the *over* category shows. This function is `table.insert`, and its type depends on the calling arity. That is, calling `table.insert(l, v)` inserts the element `v` at the end of the list `l`, while `table.insert(l, p, v)` inserts the element `v` at the position `p` of the list `l`, and generates a run-time error when `p` is out of bounds. This function also does not follow the semantics of Lua on discarding extra arguments, and generates a run-time error whenever we pass more than three arguments, even if the first three arguments match its signature.

Even though the `math` module looks straightforward to type, the *over* category shows that it includes several overloaded functions. For instance, the function `math.random` is in this category because it has two different types: `() -> (number)` and `(integer, integer?) -> (integer)`. This means that the type of `math.random` depends on the calling arity. Calling `math.random()` returns a random floating point between `0` and `1`. Calling `math.random(10)` is equivalent to `math.random(1,10)`, and returns an integer between this interval. Like `table.insert`, this function also does not follow the semantics

of Lua on discarding extra arguments, and generates a run-time error whenever we pass more than two arguments.

The `io` module provides operations for manipulating files, and these operations can use implicit or explicit file descriptors. The implicit operations are functions in the `io` table, while the explicit operations are methods of a file descriptor. We used an `userdata` declaration to introduce the type `file` for representing the type of a file descriptor and its methods. The evaluation results include both implicit and explicit operations.

We could give precise static types to most of the members of the `io` module, but the *hard* category shows that it includes some members that we could not give a precise static type. The functions `io.read` and `io.lines` are in the *hard* category along with the methods `file:read` and `file:lines`.

We could not precisely type `io.read` because its return type relies on format strings. For instance, calling `io.read("l")` returns a string or `nil`, `io.read("n")` returns a number or `nil`, and `io.read("l", "n")` returns a string or `nil` and a number or `nil`. The function `io.lines`, and the methods `file:read` and `file:lines` have the same issue.

There are two functions in the `os` module that we could not give a precise static type because they are overloaded functions. The functions that are in the *over* category are `os.date` and `os.execute`.

The evaluation results show that our type system should include intersection types, parametric polymorphism, and effect types, as these features would help us increase the static typing of the Lua Standard Library. Intersection types would allow us to define overloaded function types. Parametric polymorphism would allow us to define generic function and table types. Effect types would allow us to type coroutines.



## 5.2 MD5

The MD5 library is an OpenSSL based message digest library for Lua. It contains just the `md5` module that is written in C, so we used Typed Lua's description file to type it. Table 5.3 summarizes the evaluation results for MD5.

| | percentage of members | | | | |
|---|---|---|---|---|---|
| **Module** | **easy** | **poly** | **over** | **hard** | **# members** |
| md5 | 100% | 0% | 0% | 0% | 13 |

Table 5.3: Evaluation results for MD5

Even tough it was straightforward to type the MD5 library, we found a little difference between its documentation and its behavior. The document-

ation suggests that the type of `md5.update` is `(md5_context, string) -> (md5_context)`, though there is a call to this function in the test script that passes an extra string argument. Reading the source code, we found that its actual type is `(md5_context, string*) -> (md5_context)`, that is, we can pass zero or more strings to `md5.update`.

This case study shows that type annotations help programmers maintain the documentation updated, as the type checker always validates them.

## 5.3 LuaSocket

LuaSocket is a library that adds network support to Lua, and it is split into two parts: a core that is written in C and a set of Lua modules. The C core provides TCP and UDP support, while the Lua modules provide support for SMTP, HTTP, and FTP client protocols, MIME encoding, URL manipulation, and LTN12 filters [Neh08]. We used Typed Lua's description files to type both parts, as we also wanted to use LuaSocket to test description files to statically type the interface of modules that are written in Lua. Table 5.4 summarizes the evaluation results for LuaSocket.

| | percentage of members | | | | |
|---|---|---|---|---|---|
| **Module** | **easy** | **poly** | **over** | **hard** | **# members** |
| socket | 83% | 0% | 0% | 17% | 60 |
| ftp | 83% | 0% | 17% | 0% | 6 |
| http | 80% | 0% | 20% | 0% | 5 |
| smtp | 100% | 0% | 0% | 0% | 7 |
| mime | 100% | 0% | 0% | 0% | 17 |
| ltn12 | 95% | 5% | 0% | 0% | 20 |
| url | 100% | 0% | 0% | 0% | 8 |

Table 5.4: Evaluation results for LuaSocket

We could give precise static types to most of the members in the `socket` module, which is the C core. However, this module includes some functions that we could not give a precise static type because they rely on reflection, as the *hard* category shows. For instance, `socket.skip` is a function that is in this category. We can use this function to choose the number of values that we want to return. As an example, calling `socket.skip(1, nil, "hello")` returns only the string `"hello"`, because `1` indicates that we do not want to return the first value. Passing a negative number to `socket.skip` can be dangerous, as it returns anything that might be in the stack. As an example, calling `socket.skip(-1, nil, "hello")` returns the tuple `(-1, nil, "hello")`, because `-1` makes `socket.skip` not skip any values. As another example, the

code `f = socket.skip(-2)` assigns `socket.skip` to `f`, as `-2` gets `socket.skip` from the stack. Our type system cannot handle the type of negative numbers, as this requires more complex types such as the refinement types from hybrid type checking [Fla06].

We could give precise static types to most of the members of the modules `ftp` and `http`, but we could not precisely type two overloaded functions: `ftp.get` and `http.request`.

The function `ftp.get` downloads data from a given URL, which can be either a string or a table. More precisely, `ftp.get(url)` returns the tuple `(string) | (nil, string)` if `url` is a string, and it returns the tuple `(number) | (nil, string)` if `url` is a table.

The function `http.request` downloads data from a given URL, which can be either a string or a table. More precisely, `http.request(url, body)` returns the tuple type `(string, number, {string:string}, number) | (nil, string)` if `url` is a string and `body` is another string or `nil`, but it returns the tuple type `(number, number, {string:string}, number) | (nil, string)` if `url` is a table and `body` is `nil`.

The modules `mime` and `ltn12` have a strong connection. The `mime` module offers low-level and high-level filters that apply and remove some text encodings. The low-level filters are written in C, while the high-level filters use the function `ltn12.filter.cycle` along with the low-level filters to create standard filters.

Even though we could type all the members of the `mime` module, the function `ltn12.filter.cycle` is the only member of the `ltn12` module that we could not give a precise type. This function is difficult to type because it is polymorphic.

The modules `smtp` and `url` were straightforward to type. The `smtp` module provides functions that send e-mails. The `url` module provides functions that manipulate URLs.

## 5.4 LuaFileSystem

LuaFileSystem is a library that extends the set of functions for manipulating file systems in Lua. It contains just the `lfs` module that is written in C, so we used Typed Lua's description files to type it. Table 5.5 summarizes the evaluation results for LuaFileSystem.

Even though we could precisely type most of the functions exported by the `lfs` module, we could not type two overloaded functions due to the lack of intersection types in our type system.

| | percentage of members | | | | |
|---|---|---|---|---|---|
| **Module** | **easy** | **poly** | **over** | **hard** | **# members** |
| lfs | 89% | 0% | 11% | 0% | 19 |

Table 5.5: Evaluation results for LuaFileSystem

## 5.5 HTTP Digest

The HTTP Digest library implements client side HTTP digest authentication for Lua. Table 5.6 summarizes the evaluation results for HTTP Digest.

| | percentage of members | | | | |
|---|---|---|---|---|---|
| **Module** | **easy** | **poly** | **over** | **hard** | **# members** |
| http-digest | 0% | 0% | 100% | 0% | 1 |

Table 5.6: Evaluation results for HTTP Digest

It is difficult to type the interface of the `http-digest` module because it is an extension to the `http` module from LuaSocket. The `http-digest` module only exports the function `http-digest.request`, which extends the function `http.request` with MD5 authentication. Like `http.request`, `http-digest.request` is also an overloaded function.

Even though we could not precisely type the interface that `http-digest` exports, we could use only static types to annotate this module, and they pointed a bug in the code. The problem was related to the way the library was loading the MD5 library that should be used. This part of the code checks the existence of three different MD5 libraries in the system, and uses the first one that is available, or generates an error when none is available. The code that loads the first option was fine, but the code that loads the second and third options were trying to access an undefined global variable.

## 5.6 Typical

Typical is a library that extends the behavior of the function `type`. Table 5.7 summarizes the evaluation results for Typical.

| | percentage of members | | | | |
|---|---|---|---|---|---|
| **Module** | **easy** | **poly** | **over** | **hard** | **# members** |
| typical | 100% | 0% | 0% | 0% | 1 |

Table 5.7: Evaluation results for Typical

The interface of the `typical` module is straightforward to type, as it contains only the function `typical.type`, which has the same type of the function `type`: `(value) -> (string)`.

However, we hit some limitations of our type system while annotating this module.

First, it uses the `getmetatable` to get a table and checks whether this table has the field `__type`. We could not give a precise type to `getmetatable`, so we used the dynamic type `any` as its return type, and this generates a warning.

Second, it returns a metatable that extends `__call` with `typical.type`, that is, we can use the module itself as a function, though it is a table. Our type system still does not support metatables, so we did not extend our version of the `typical` module to support `__call`.

Third, the module uses `ipairs` to iterate over an array of functions, but our type system also has limited support to `ipairs`, and generates a warning when we try to use the indexed value inside the `for` body. As we mentioned in this chapter, we use the dynamic type as a replacement for the lack of type parameters. This means that we get warnings inside an `ipairs` iteration, because all iterated elements have the dynamic type. We removed this warning using the numeric `for` to perform the same loop.



## 5.7 Modulo 11

Modulo 11 is a library that generates and verifies modulo 11 numbers. Table 5.8 summarizes the evaluation results for Typical.

| | percentage of members | | | | |
|---|---|---|---|---|---|
| **Module** | **easy** | **poly** | **over** | **hard** | **# members** |
| mod11 | 78% | 0% | 0% | 22% | 9 |

Table 5.8: Evaluation results for Modulo 11

The `mod11` module was written using an object-oriented idiom that our type system does not support, and that is the reason why we could not type all the members of its interface. More precisely, the original code uses `setmetatable` to hide two attributes, which our type system cannot hide.

In addition, it returns a metatable that extends `__call` with the class constructor. This allows us to use the module itself to create new instances of a Modulo 11 number. However, our type system does not support this feature, and we need to make explicit calls to the constructor whenever we want to create a new instance.

Even though we had these two issues to annotate the `mod11` module, we could use only static types to annotate it, and we found some interesting points. The code relies on implicit conversions between strings and numbers, and some parts of the code keep on changing the type of local variables. These are two practices that may hide bugs.

## 5.8 Typed Lua Compiler

The Typed Lua compiler is the last case study that we evaluated. Table 5.9 summarizes its evaluation results.

| | percentage of members | | | | |
|---|---|---|---|---|---|
| **Module** | **easy** | **poly** | **over** | **hard** | **# members** |
| tlast | 98% | 0% | 2% | 0% | 47 |
| tltype | 100% | 0% | 0% | 0% | 65 |
| tlst | 100% | 0% | 0% | 0% | 26 |
| tllexer | 18% | 0% | 0% | 82% | 11 |
| tlparser | 100% | 0% | 0% | 0% | 1 |
| tldparser | 100% | 0% | 0% | 0% | 1 |
| tlchecker | 100% | 0% | 0% | 0% | 2 |
| tlcode | 100% | 0% | 0% | 0% | 1 |

Table 5.9: Evaluation results for Typed Lua Compiler

The `tlast` module implements the Abstract Syntax Tree for the compiler. We could not precisely type just one function, because it has an overloaded type that requires intersection types.

The `tltype` module implements the types introduced by Typed Lua. It also implements the subtyping and consistent-subtyping relations. The interface that this module exports was straightforward to type.

The `tlst` module implements the symbols table for the compiler. The interface that this module exports was also straightforward to type.

The `tllexer` module defines common lexical rules for the Typed Lua parser and the description file parser. This module is hard to type because it uses LPeg [Ier08, Ier09] patterns, and LPeg uses overloaded arithmetic operators to build LPeg patterns. Even though LPeg is the third most popular Lua module, we cannot precisely type LPeg patterns because our type system still does not support overloading arithmetic operators. In the `tllexer` module, we could only give precise static types to two error reporting functions that it exports.

The `tlparser` and `tldparser` modules implement the Typed Lua parser and the description file parser, respectively. Even though they use LPeg to

implement the grammar rules, they only export a parsing function. Both use LPeg to parse a string and return the corresponding AST.

We could type the interfaces that modules `tlchecker` and `tlcode` export. The former traverses the AST to perform type checking, while the latter traverses the AST to perform code generation.

Even though we could precisely type the interface that most of the modules export, we had issues to write mutually recursive functions. This kind of functions often appear in compilers construction to traverse the data structures that they use. However, Typed Lua still does not support mutually recursive functions. A way to overcome this limitation was to predeclare these functions with an empty body, and then redeclare them with their actual body. The first declaration specifies the function type, while the second specifies what the function actually does without changing any type definition.

Traversing the AST would also be problematic if we had not included a way to discriminate unions of table types, as we mentioned in Section 3.5. Without a way to discriminate a union of table types, any attempt to index this union of table types would generate a warning.

Bootstraping the compiler also helped revealing some bugs. We found some accesses to undeclared global variables and also to undeclared table fields. The compiler also helped pointing the places where we should narrow a nilable value before using it. In fact, this point appeared in all the case studies that we used Typed Lua to annotate Lua code. This means that Lua programmers often use possibly `nil` values before checking whether it is `nil`.