Typed Lua: An Optional Type System for Lua

André Murbach Maidl
PUC-Rio
Rio de Janeiro, Brazil
amaidl@inf.puc-rio.br

Fabio Mascarenhas
UFRJ
Rio de Janeiro, Brazil
mascarenhas@ufrj.br

Roberto Ierusalimschy
PUC-Rio
Rio de Janeiro, Brazil
roberto@inf.puc-rio.br

ABSTRACT

Dynamically typed languages trade flexibility and ease of use for safety, while statically typed languages prioritize the early detection of bugs, and provide a better framework for structure large programs. The idea of optional typing is to combine the two approaches in the same language: the programmer can begin development with dynamic types, and migrate to static types as the program matures. The challenge is designing a type system that feels natural to the programmer that is used to programming in a dynamic language.

This paper presents the initial design of Typed Lua, an optionally-typed extension of the Lua scripting language. Lua is an imperative scripting language with first class functions and lightweight metaprogramming mechanisms. The design of Typed Lua's type system has a novel combination of features that preserves some of the idioms that Lua programmers are used to, while bringing static type safety to them. We show how the major features of the type system type these idioms with some examples, and discuss some of the design issues we faced.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type structure*

Keywords

Lua programming language, type systems, optional typing, gradual typing

1. INTRODUCTION

Dynamically typed languages such as Lua avoid static types in favor of runtime *type tags* that classify the values they compute, and their implementations use these tags to perform runtime (or dynamic) checking and guarantee that only valid operations are performed [21].

The absence of static types means that programmers do not need to bother about abstracting types that might require a complex type system and type checker to validate, leading to simpler and more flexible languages and implementations. But this absence 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 large programs.

The early error detection and better program structure afforded by static type checking can lead programmers to migrate their code from a dynamically typed to a statically typed language, once their simple scripts become complex programs [28]. As this migration involves languages with different syntax and semantics, it 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* [7] and *gradual typing* [22] are two approaches that offer programmers the option to use type annotations where static typing is needed, incrementally migrating a system from dynamic to static types. The difference between these two approaches is the way they treat runtime semantics: while optional type systems do not affect the runtime semantics, gradual typing uses runtime checks to ensure that dynamically typed code does not violate the invariants of statically typed code.

This paper presents the initial design of Typed Lua: an optional type system for Lua that is complex enough to preserve some of the idioms that Lua programmers are already used to, while adding new constructs that help programmers structure Lua programs.

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

The primary use of Lua has always been as an embedded language for configuration and extension of other applications. Lua prefers to provide mechanisms instead of fixed

Dyla '14, June 09-11 2014, Edinburgh, United Kingdom

ACM 978-1-4503-2916-3/14/06 ...$15.00
http://dx.doi.org/10.1145/2617548.2617553.

policies for structuring programs, and even features such as a module system and object orientation are a matter of convention instead of built into the language. The result is a fragmented ecosystem of libraries, and different ideas among Lua programmers on how they should use the language features and how they should structure programs.

The lack of standard policies is a challenge for the design of a static type system for the Lua language. The design of Typed Lua is informed by a (mostly automated) survey of Lua idioms used in a large corpus of Lua libraries, instead of relying just on the semantics of the language.

Typed Lua allows statically typed Lua code to coexist and interact with dynamically typed code. The Typed Lua compiler warns the programmer about type errors, but always generates Lua code that runs in unmodified Lua implementations. The programmer can enjoy some of the benefits of static types even without converting existing Lua modules to Typed Lua: a dynamically typed module can export a statically typed interface, and statically typed users of the module will have their use of the module checked by the compiler.

Unlike gradual type systems, Typed Lua does not insert runtime checks between dynamically and statically typed parts of the program. Unlike some optional type systems, the statically typed subset of Typed Lua is sound by design, so a later version of the Typed Lua compiler can swicth to gradual instead of just optional typing.

We cover the main parts of the design, along with how they relate to Lua, in Sections 2 through 6. Section 7 reviews related work on mixing static and dynamic typing in the same language. Finally, Section 8 gives some concluding remarks and future extensions for Typed Lua.

2. ATOMIC TYPES AND FUNCTIONS

Lua values can have one of eight *tags*: *nil*, *boolean*, *number*, *string*, *function*, *table*, *userdata*, and *thread*. In this section, we will see how Typed Lua assigns types to values of the first five.

Figure 1 gives the abstract syntax of Typed Lua types. Only *first-class types* correspond to actual Lua values; *second-class types* correspond to expression lists, and Typed Lua uses them to type multiple assignment and function application.

Types are ordered by a subtype relationship, where any first-class type is a subtype of **value**. The rest of the subtype relationship is standard: union types are supertypes of their parts, **number**, **boolean**, and **string** are supertypes of their respective literal types, function types are related by contravariance on the input part and covariance in the output part, table types have width subtyping, with depth subtyping on **const** fields, tuple and vararg types are covariant.

The *dynamic type* **any** allows dynamically typed code to interoperate with statically typed code; it is a subtype of **value**, but neither a supertype nor a subtype of any other type. We relate **any** to other types with the *consistency* and *consistent- subtype* relationships used by gradual type systems [22, 23]. In practice, we can pass a value of the dynamic type anytime we want a value of some other type, and can pass any value where a value of the dynamic type is expected, but these operations are tracked by the type system, and the programmer can choose to be warned about them.

Type Language

First-class types

$T ::=$	L	*literal types*
	$\mid B$	*base types*
	$\mid$ **value**	*top type*
	$\mid$ **any**	*dynamic type*
	$\mid$ **self**	*self type*
	$\mid T \cup T$	*disjoint union types*
	$\mid S \to S$	*function types*
	$\mid \{F, ..., F\}$	*table types*
	$\mid X$	*type variables*
	$\mid \mu X.T$	*recursive types*
	$\mid X_i$	*projection types*
$L ::=$	**false** $\mid$ **true** $\mid$ $<$*number*$>$ $\mid$ $<$*string*$>$	
$B ::=$	**nil** $\mid$ **boolean** $\mid$ **number** $\mid$ **string**	
$F ::=$	$T : T \mid$ **const** $T : T$	*field types*

Second-class types

$S ::=$	**void** $\mid V \mid V \cup S$	
$V ::=$	T	
	$\mid T*$	*vararg types*
	$\mid T \times V$	*tuple types*

Figure 1: Abstract syntax of Typed Lua types

Typed Lua allows optional type annotations in variable function declarations. It assigns the dynamic type to any parameter that does not have a type annotation, but assigns more precise types to unannotated variables, based on the type of the expression that gives the initial value of the variable.

In the following example, we use type annotations in a function declaration but do not use type annotations in the declaration of a local variable:

```
local function factorial(n: number): number
  if n == 0 then
    return 1
  else
    return n * factorial(n - 1)
  end
end
local x = 5
print(factorial(x))
```

The compiler assigns the type **number** to the local variable `x`, and this example compiles without warnings. Typed Lua allows programmers to combine annotated code with unannotated code, as we show in the following example:

```
local function abs(n: number)
  if n < 0 then
    return -n
  else
    return n
  end
end
local function distance(x, y)
  return abs(x - y)
end
```

The compiler assigns the dynamic type **any** to the input

parameters of `distance` because they do not have type annotations. Subtracting a value of type **any** from another also yields a value of type **any** (Lua has operator overloading, so the minus operation is not guaranteed to return a number in this case), but consistent subtyping lets us pass a value of type **any** to a function that expects a **number**.

Even though the return types of both `abs` and `distance` are not given, the compiler is able to infer a return type of **number** to both functions, as they are local and not recursive.

Lua has first-class functions, but they have some peculiarities. First, the number of arguments passed to a function does not need to match the function's arity; Lua silently drops extra arguments after evaluating them, or passes `nil` in place of any missing arguments. Second, functions can return any number of values, and the number of values returned may not be statically known. Third, Lua also has multiple assignment, and the semantics of argument passing match those of multiple assignment (or vice-versa); calling a function is like doing a multiple assignment where the left side is the parameter list and the right side is the argument list.

Typed Lua uses *second-class types* to encode the peculiarities of argument passing, multiple returns, and multiple assignment. We call them second-class because these types do not correspond to actual values and cannot be assigned to variables or parameters: they are an artifact of the interaction between the type system and the semantics of Lua.

As we can see in Figure 1, a second-class type in Typed Lua can be a tuple of first-class types optionally ending in a variadic type, or a union of these tuples. A variadic type $T*$ is a generator for a sequence of values of type $T \cup \mathbf{nil}$. Unions of tuples play an important part in functions that are overloaded on the return type, together with *projection types*. Both are explained in the next section.

In its default mode of operation, Typed Lua always adds a variadic tail to the parts of a function type if none is specified, to match the semantics of Lua function calls. In the examples above, the types of `factorial` and `abs` are actually $\mathbf{number} \times \mathbf{value}* \to \mathbf{number} \times \mathbf{nil}*$, and the type of `distance` is $\mathbf{any} \times \mathbf{any} \times \mathbf{value}* \to \mathbf{number} \times \mathbf{nil}*$.

If we call `abs` with extra arguments, Typed Lua silently ignores them, as the type signature lets `abs` receive any number of extra arguments. If we call `abs` in the right side of an assignment to more than one lvalue, Typed Lua checks if the first lvalue has a type consistent with **number**, and any other lvalues need to have a type consistent with **nil**.

There is an optional stricter mode of operation where Typed Lua does not give variadic tails to the parts of a function type unless the programmer explictly declares it, and so will also check all function calls for arity mismatch.

A variadic type can only appear in the tail position of a tuple, because Lua takes only the first value of any expression that appears in an expression list that is not in tail position. The following example shows the interaction between multiple returns and expression lists:

```
local function multiple()
  return 2, "foo"
end
local function sum(x: number, y: number)
  return x + y
end
local x, y, z = multiple(), multiple()
print(sum(multiple(), multiple())
```

Function `multiple` is $\mathbf{value}* \to \mathbf{number} \times \mathbf{string} \times \mathbf{nil}*$, and `sum` is $\mathbf{number} \times \mathbf{number} \times \mathbf{value}* \to \mathbf{number} \times \mathbf{nil}*$. In the right side of the multiple assignment, only the first value produced by the first call to multiple gets used, so the type of the right side is $\mathbf{number} \times \mathbf{number} \times \mathbf{string} \times \mathbf{nil}*$, and the types assigned to `x`, `y`, and `z` are respectively **number**, **number**, and **string**. This also means that the call to `sum` compiles without errors, as the first two components of the tuple are consistent with the types of the parameters, and the other components are consistent with **value**.

3. UNIONS

Typed Lua uses union types to encode some common Lua idioms: optional values, overloading based on the tags of input parameters, and overloading on the return type of the functions.

Optional values are unions of some type and **nil**, and are so common that Typed Lua uses the `t?` syntax for these unions. They appear any time a function has optional parameters, and any time the program reads a value from an array or map.

```
local function message(name: string,
                       greeting: string?)
  local greeting = greeting or "Hello "
  return greeting .. name
end

print(message("Lua"))
print(message("Lua", "Hi"))
```

In this example, the second parameter is optional but, in the first line of the function, we declare a new variable that is guaranteed to have type **string** instead of $\mathbf{string} \cup \mathbf{nil}$. In Lua, any value except `nil` and `false` are "truthy", so the short-circuiting `or` operator is a common way of giving a default value to an optional parameter. Typed Lua encodes this idiom with a typing rule: if the left side of `or` has type $T \cup \mathbf{nil}$ and the right side has type T then the `or` expression has type T.

Declaring a new `greeting` variable that shadows the parameter is not necessary:

```
local function message(name: string,
                       greeting: string?)
  greeting = greeting or "Hello "
  return greeting .. name
end
```

Typed Lua lets the assignment $x = x$ **or** e change the type of x from $t \cup \mathbf{nil}$ to t, as long as the type of e is a subtype of t, x is local to the current function, and it is not assigned in another function. The change only affects the type of x in the remainder of the current scope. In the case of `greeting`, the assignment on line three changes its type to **string**.

Overloaded functions use the `type` function to inspect the tag of their parameters, and perform different actions depending on what those tags are. The simplest case overloads on just a single parameter:

```
local function overload(s1: string,
```

```
                          s2: string|number)
  if type(s2) == "string" then
    return s1 .. s2
  else
    -- string.rep: (string, number) -> string
    return string.rep(s1, s2)
  end
end
```

Typed Lua has a small set of type predicates that, when used over a local variable in a condition, constrain the type of that variable. The function above uses the `type(x) == "string"` predicate which constrains the type of x from $T \cup$ **string** to **string** when the predicate is true and T otherwise. This is a simplified form of *flow typing* [13, 30]. As with `or`, the variable must be local to the function, and cannot be assigned to in another function.

The type predicates can only discriminate based on tags, so they are limited on the kinds of unions that they can discriminate. It is possible to discriminate a union that combines a table type with a base type, or a table type with a function type, or a two base types, but it is not possible to discriminate between two different function types.

Functions that overload their return types to signal the occurrence of errors are another common Lua idiom. In this idiom, a function returns its normal set of return values in case of success but, if anything fails, returns `nil` as the first value, followed by an error message or other data describing the error, as in the following example:

```
local function idiv(d1: number, d2: number):
      (number, number)|(nil, string)
  if d2 == 0 then
    return nil, "division by zero"
  else
    local r = d1 % d2
    local q = (d1 - r)/d2
    return q, r
  end
end
```

There is also special syntax for this idiom: we could annotate the return type of `idiv` with `(number, number)?` to denote the same union[1].

The full type of `idiv` is **number** $\times$ **number** $\times$ **value**$* \rightarrow$ (**number** $\times$ **number** $\times$ **nil**$*$) $\cup$ (**nil** $\times$ **string** $\times$ **nil**$*$). A typical client of this function would use it as follows:

```
local q, r = idiv(n1, n2)
-- q is number|nil, r is number|string
if q then
  -- q and r are numbers
else
  -- r is a string
end
```

When Typed Lua encounters a union of tuples in the right side of an declaration, it stores the the union in a special type environment with a fresh name and assigns *projection types* to the variables in the left side of the declaration. If the type variable is X, variable `q` gets type X_1 and variable `r` gets type X_2.

[1]The parentheses are always necessary here: `number?` is `number|nil`, while `(number)?` is `(number)|(nil, string)`.

If we need to check a projection type against some other type, we take the union of the corresponding component in each tuple. But if a variable with a projection type appears in a type predicate, the predicate discriminates against all tuples in the union. In the example above, X is (**number** $\times$ **number** $\times$ **nil**$*$) $\cup$ (**nil** $\times$ **string** $\times$ **nil**$*$) outside of the `if` statement, but **number** $\times$ **number** $\times$ **nil**$*$ in the `then` block and **nil** $\times$ **string** $\times$ **nil**$*$ in the `else` block.

Notice that we could also discriminate `r` using `type(r) == "number"` as our predicate, with the same result. The first form is more succinct, and more idiomatic. We can also use projection types to write overloaded functions where the type of a parameter depends on the type of another parameter.

Assigning to a variable with a projection type is forbidden, unless the union has been discriminated down to a single tuple, Unrestricted assignment to these variables would be unsound, as it could break the dependency relation between the types in each tuple that is part of the union.

Currently, a limitation of our overloading mechanisms is that the return type cannot depend on the input types; we cannot write a function that is guaranteed to return a number if passed a number and guaranteed to return a string if passed a string, for example. While intersection types provide a way to express the type of such a function as **number** $\rightarrow$ **number** $\cap$ **string** $\rightarrow$ **string**, more sophisticated flow typing is needed to actually check that a function has this type, and we are still working on this problem.

4. TABLES AND INTERFACES

Tables are the main mechanism that Lua has to build data structures. They are associative arrays where any value (except `nil`) can be a key, but with language support for efficiently using tables as tuples, arrays (dense or sparse), records, modules, and objects. In this section, we show how Typed Lua encodes tables as arrays, records, tuples, and plain maps in its type system.

Typed Lua uses the same framework to represent the different uses that a Lua table has: *table types*. A table type $\{t_1 : u_1, \ldots, t_n : u_n\}$ represents a map from values of type t_i to values of type u_i.

The concrete syntax of Typed Lua has syntax for common table types. One syntax defines table types for maps: it is written `{ t: u }`, and maps to the table type $\{t : u\}$. This table type represents a map that maps values of type t to values of type u. Another syntax defines table types for arrays: it is written `{ t }`, and is equivalent to the table type $\{$**number** $: t\}$. A third syntax defines table types for records: it is written `{ s1: t1, ..., sn: tn }`, where each s_i is a literal number, string, or boolean, and maps to the table type $\{s_1 : t_1, ..., s_n : t_n\}$, where each s_i is the corresponding literal type.

The example below shows how we can define a map from strings to numbers; the dot syntax for field access in Lua is actually syntactic sugar for indexing a table with a string literal:

```
local t: { string: number } = { foo = 1 }
local x: number = t.foo       -- x gets 1
local y: number = t["bar"]    -- runtime error
```

When accessing a map, there is always the possibility that the key is not there. In Lua, accessing a non-existing key

returns `nil`. Typed Lua is stricter, and raises a runtime error in this case. To get a map with the behavior of standard Lua tables, the programmer can use an union:

```
local t: { string: number? } = { foo = 1 }
local x: number = t.foo       -- compile error
local y: number = t.bar or 0  -- y gets 0
local z: number? = t["bar"]   -- z gets nil
```

Now the Typed Lua compiler will complain about the assignment on line two. The following example shows how we can declare an array:

```
local days: { string } = { "Sunday", "Monday",
  "Tuesday", "Wednesday", "Thursday",
  "Friday", "Saturday" }
local x = days[1]          -- x gets "Sunday"
local y = days[8]          -- runtime error
```

Notice that we have the same strictness with missing elements, unless the type of the elements has `nil` as a possible value.

While this runtime check is an instance where the semantics of a Typed Lua program deviates from the semantics of plain Lua, the alternatives would be to make all arrays and maps have an element type that includes `nil`, and either make the programmer narrow the type with `or` or an `if` statement, making using the elements more incovenient, or make `nil` a subtype of every type, reducing the amount of type safety in the system. Notice that this does not change the semantics of dynamically typed programs, as the runtime checks are only added when the table has a strict static type.

If we want to declare a tuple, we can leave the variable declaration unannotated and let Typed Lua assign a more specific table type to the variable. If we remove the annotation in the previous example, the compiler assigns the following table type to `days`:

$$\{1 : \textbf{string},\ 2 : \textbf{string},\ 3 : \textbf{string},\ 4 : \textbf{string},\ 5 : \textbf{string},\ 6 : \textbf{string},\ 7 : \textbf{string}\}$$

This type is not a subtype of $\{\textbf{number} : \textbf{string}\}$, nor is $\{\textbf{number} : \textbf{string}\}$ a subtype of $\{\textbf{number} : \textbf{string} \cup \textbf{nil}\}$, because in both cases the subtype relationship would be unsound. In the first case, a table type with the same fields as the type above, plus $8 : \textbf{number}$, is a subtype of the table type above, so would also be a subtype of $\{\textbf{number} : \textbf{string}\}$, which is clearly unsound. In the second case, covariance in the type of mutable fields is also unsound, for the same reason as the unsoundness of array covariance.

While the record type above, $\{\textbf{number} : \textbf{string}\}$, and $\{\textbf{number} : \textbf{string} \cup \textbf{nil}\}$ are disjoint, all three types are valid for the table constructed in the example. Typed Lua actually assigns different types to a table constructor expression depending on the context where it is used.

Finally, the next example shows how we can declare a record:

```
local person: { "firstname": string,
                "lastname": string } =
  { firstname = "Lou", lastname = "Reed" }
```

We could leave the type annotation out, and Typed Lua would assign the same type to `person`.

As records get bigger, and types of record fields get more complicated, writing table types can be unwieldy, so Typed Lua has *interfaces* as syntactic sugar for record types:

```
local interface Person
  firstname: string
  lastname: string
end
```

The declaration above declares `Person` as an alias to the record type {"firstname": **string**, "lastname": **string**} in the remainder of the current scope. We can now use `Person` in type declarations:

```
local function greet(person: Person)
  return "Hello, " .. person.firstname ..
         " " .. person.lastname
end
local user1 = { firstname = "Lewis",
                middlename = "Allan",
                lastname = "Reed" }
local user2 = { firstname = "Lou" }
local user3 = { lastname = "Reed",
                firstname = "Lou" }
local user4 = { "Lou", "Reed" }
print(greeter(user1)) -- Hello, Lewis Reed
print(greeter(user2)) -- Error
print(greeter(user3)) -- Hello, Lou Reed
print(greeter(user4)) -- Error
```

If our record type has fields that can be `nil`, we need to use an explicit type declaration when declaring a variable of this record type, as the following example shows:

```
local interface Person
  firstname: string
  middlename: string?
  lastname: string
end
local user1: Person = { firstname = "Lewis",
                        middlename = "Allan",
                        lastname = "Reed" }
local user2: Person = { lastname = "Reed",
                        firstname = "Lou" }
```

We need an explicit type declaration because neither of the types that the Typed Lua compiler assigns to the table constructors above is a subtype of {"firstname": **string**, "middlename": **string** $\cup$ **nil**, "lastname": **string**}, the type that `Person` describes.

We can also use interfaces to define recursive types:

```
local interface Element
  info: number
  next: Element?
end
```

It is common in Lua programs to build a record incrementally, starting with an empty table, as in the following example:

```
local person = {}
person.firstname = "Lou"
person.lastname = "Reed"
```

Ideally, we want the type of `person` to change as the table gets built, from {} to {"firstname": **string**} and finally to {"firstname": **string**, "lastname": **string**}. This is tricker than the type change introduced by assignment that we saw in Section 2.2, as what is changing is not just the type of the variable `person` but the type of the value that `person` points to. This is safe in the example above, but not in the example below:

```
local bogus = { firstname = 1 }
local person: {} = bogus
person.firstname = "Lou"
person.lastname = "Reed"
```

The assignment on line two is perfectly legal, as the type of `bogus` is a subtype of {}. But changing the type of `person` would be unsound: `person.firstname` is now a **string**, but `bogus.firstname` is still typed as a **number**. We do not even need to declare a type for aliasing to be a problem:

```
local person = {}
local bogus = person
bogus.firstname = 1
person.firstname = "Lou"
person.lastname = "Reed"
```

Taken individually, the changes to the type of the two variables look fine, but aliasing makes one of them unsound. The location of the change also matters, as the next example shows:

```
local person = {}
local bogus = { firstname = 1 }
do
  person.firstname = 1
  bogus = person
end
do
  person.firstname = "Lou"
end
-- bogus.firstname is now "Lou"
```

The initial type of `person` in all of these examples is {}, but the *origin* of this type judgment matters on whether it is sound to allow a change to the type of `person` or not, even if the change is always towards a subtype of the current type.

Typed Lua tags a variable with a table type as either *open* or *closed*. If a variable gets its type from a table constructor then it is open, otherwise it is always closed. The type of an open variable may change by field assignment, subject to three restrictions: the variable must be local to the current block, the new type must be a subtype of the old type, and the variable cannot have been assigned to in another function.

An variable with an open table type may be aliased, but these aliases are not open. Any mutation on these aliases is not a problem, as the type of the original reference can only change towards a subtype. For mutable fields this means that the type of the field cannot change once it is added to the type of the table.

Using a variable with an open type can also trigger a type change, if the type of the missing fields has `nil` as a possible value. This lets the programmer incrementally create an instance of an interface with an optional type:

```
local interface Person
  firstname: string
  middlename: string?
  lastname: string
end
local user = {}
user.firstname = "Lou"
user.lastname = "Reed"
local person: Person = user
```

Table types are the foundation for modules and objects in Typed Lua. Type changes triggered by field assignment are also an important part of Typed Lua's support for the idiomatic definition of Lua modules, which are the subject of the next section.

5. MODULES

Lua's module system, like other parts of the language, is a matter of convention. When Lua first needs to load a module, it executes the module's source file as a function; the value that this function returns is the module, and Lua caches it for future loads. While a module can be any Lua value, most modules are tables where the fields of the table are functions and other values that the module exports.

The modules that we surveyed build this table using three distinct styles. In the first style, the module's source file ends with a `return` statement that uses a table constructor with the exported members. In the second style, the module declares an empty table in the beginning of its source file, adds exported members to this table throughout the module, and returns this table at the end.

These two styles are straightforward for Typed Lua, which can just take the type of the first value that the module returns and use it as the type of the module.

In the third style, which has been deprecated in the current version of Lua, a module begins with a call to the `module` function. This function installs a fresh table as the global environment for the rest of the module, so any assignments to global variables are field assignments to this table. The `module` function also sets an `_M` field in this table as a circular reference to the table itself, so the module can end with `return _M`, but this explicit return is not necessary.

While this style has been deprecated, our survey indicated that around a third of Lua modules in a popular module repository still use this style, so Typed Lua also supports this style: it treats accesses to global variables as field accesses to an open table in the top-level scope.

6. OBJECTS AND CLASSES

Lua's built-in support for object-oriented programming is minimal. The basic mechanism is the `:` syntactic sugar for method calls and method declarations. The Lua compiler translates `obj:method(args)` to an operation that evaluates `obj`, looks for a field named "method" in the result, then calls it with the result of evaluating `obj` as the first argument, followed by the result of evaluating the argument list in the original expression.

We can use table types and the *self type* to represent objects, and Typed Lua has syntactic sugar to make defining these types easier:

```
interface Shape
  x, y: number
```

```
  const move: (dx: number, dy: number) => ()
end
```

The double arrow in the type of the two methods is syntactic sugar for having a first parameter named `self` with type **self**. The `const` qualifier is necessary for covariance in the types of the methods, and to make subtyping among object types work.

While `:` is syntactic sugar in plain Lua, Typed Lua uses it to type-check method calls, to bind any ocurrence of the **self** type in the type of the method to the receiver. Indexing a method but not immediately calling it with the correct receiver is a compile-time error.

There is still the matter of how to construct a value with the object type above. The following example shows one way:

```
local shape = { x = 0, y = 0 }
const function shape:move(dx: number,
                          dy: number)
  self.x = self.x + dx
  self.y = self.y + dy
end
```

The `:` syntactic sugar that the example uses also comes from Lua, and assigns a function to the field with a first parameter named `self`, plus any other parameters. Typed Lua adds the `const` annotation, and gives type **self** to the implicit parameter `self`.

Lua has a mechanism for Self-like (or JavaScript-like) delegation of missing fields in a table. After `setmetatable(t1, { __index = t2 })`, Lua looks up in `t2` any missing fields of `t1`. Lua programmers often use this mechanism to simulate classes, as in the following example:

```
local Shape = { x = 0, y = 0 }
const function Shape:new(x: number, y: number)
  local s = setmetatable({},
                         { __index = self })
  s.x = x
  s.y = y
  return s
end
const function Shape:move(dx: number,
                          dy: number)
  self.x = self.x + dx
  self.y = self.y + dy
end
local shape1 = Shape:new(0, 5)
local shape2: Shape = Shape:new(10, 10)
```

In the last line of the example, notice how we can refer to `Shape` in the type annotation, as a shortcut to the table type that Typed Lua has assigned to this variable.

Typed Lua assigns the type $\mathbf{self} \times \mathbf{number} \times \mathbf{number} \times \mathbf{value}* \to \mathbf{self} \times \mathbf{nil}*$ to `new`. In a `setmetatable` expression, if the type t_1 of the first operand is a supertype of the type t_2 of the second operand's __index field, it changes the type of the first operand to t_2, so the local variable `s` has the same type as `self`.

We can also use `setmetatable` to simulate single inheritance, as `setmetatable` on a table constructor assigned to a fresh local variable gives this variable an open type, which lets us add new methods and override existing ones:

```
local Circle = setmetatable({},
                            { __index = Shape })
Circle.radius = 0
const function Circle:new(x: number,
                          y: number,
                          radius: value)
  local c = setmetatable(Shape:new(x, y),
                         { __index = self })
  c.radius = tonumber(radius)
  return c
end
const function Circle:area()
  return math.pi * self.radius * self.radius
end
```

In the first line of the redefinition of `new`, notice how we can call `Shape`'s constructor inside the overridden constructor. A limitation of this class system is that the overridden constructor must be a subtype of the original constructor, so the type of `radius` has to be very permissive.

If we erase all type and `const` annotations, the two examples above are valid Lua code, with the same semantics as the Typed Lua code.

The current version of Typed Lua does not have a polymorphic type system, so programmers currently cannot hide the calls to `setmetatable` behind nicer abstractions, as some Lua libraries do. A few Lua programs also use other features of `setmetatable` which are currently not typeable, such as operator overloading.

7. RELATED WORK

Common LISP introduced optional type annotations in the early eighties [25], but they were optimization hints to the compiler instead of types for static checking. These annotations were unsafe, and could crash the program when wrong.

Abadi et al. [1] used tagged pairs and explicit injection and projection operations (coercions) to embed dynamic typing in the simply-typed lambda calculus. Dynamically-typed values had a `Dynamic` static type. Thatte [27] removes the necessity of explicit coercions with a system that automatically inserts coercions and checks them for correctness.

Soft typing [9] starts with a dynamically-typed language, and layers a static type system with a sophisticated global type inference algorithm on top, to try to find errors in programs without needing to rewrite them. In cases where an error may or may not be present, it warns the programmer and inserts a runtime check. One problem with the soft typing approach was the complexity of the inferred types, leading to errors that were difficult to understand and fix.

While the goal of soft typing is catching errors, *dynamic typing* [14] is another approach for optimizing dynamically-typed programs. First the program is translated to a program that uses a `Dynamic` type and explicit coercions and runtime checks, then a static analysis removes some of these coercions and checks.

Instead of trying to add static checking to a dynamic language, Findler and Felleisen [10] enhances the dynamic checks with the possibility of *contracts* that give assertions about the input and output of (possibly higher-order) functions. In case of higher-order functions, the actual failing check can be far away from the actual source of the error, so contracts can also add *blame annotations* to values as a

way to trace failures back to the source.

Strongtalk [8, 6] is an optionally-typed version of Smalltalk. It has a polymorphic structural type system that programmers can use to annotate Smalltalk programs, but type annotations can be left out; unannotated terms are dynamically typed, and can be cast to any static type. The interaction of the dynamic type with the rest of the type system is unsound, so Strongtalk uses the dynamically-checked semantics of Smalltalk when executing programs, even if the programs are statically typed.

Pluggable type systems [7] generalize the idea of Strongtalk, to have type systems that can be layered on top of a dynamic language without influencing its runtime semantics. These systems can be unsound in themselves, or in their interaction with the dynamically typed part of the language, without sacrificing runtime safety, as the semantics of the language catch any runtime errors caused by an unsound type system.

Dart [12] and TypeScript [18] are two recent examples of languages with optional type systems in this style. Dart is an object-oriented language with a semantics that is similar to Smalltalk's, while TypeScript is an object-oriented extension of JavaScript. Dart has a nominal type system, while TypeScript has a structural one, but both type systems are designed on purpose with unsound parts (such as covariant arrays in case of Dart, and covariant function return types in case of TypeScript) to increase programmer convenience. The interaction of statically and dynamically typed code is also unsound.

Tobin-Hochstadt and Felleisen [28] shows how programs in the untyped lambda calculus can be incrementally translated to the simply typed lambda calculus, using contracts to guarantee that the untyped part cannot cause errors in the typed part, as a model on how scripts in a dynamically typed language can be incrementally translated to a statically typed language. This approach has been realized in the Typed Scheme (later Typed Racket) language [29, 30].

Gradual typing [22] combines the optional type annotations of optional and pluggable type systems with higher-order contracts. The gradual type system is the simply typed lambda calculus enriched with a *dynamic type* ?, where a value with the dynamic type can assume any type, and vice-versa. A dynamic value that assumes a static type generates a runtime check for a first-order value, or a wrapper for a higher-order value. A static value that assumes the dynamic type is tagged. The whole system is sound: any runtime errors in a well-typed program must happen in the dynamic parts.

While both gradual typing and the approach of [28] have the goal of having a sound interaction of dynamically and statically typed code, they differ in the granularity, with the gradual typing providing a finer-grained transition from dynamically typed to statically typed code.

Gradual typing has been combined with subtyping in a simple object calculus Siek and Taha [23], with a nominal, polymorphic type system Ina and Igarashi [16], with a row polymorphism Takikawa et al. [26], and with the polymorphic lambda calculus Ahmed et al. [2]. Gradual typing also adopted blame tracking from higher-order contracts Siek and Wadler [24**?**], Wadler and Findler [31].

While Typed Racket is the first fully-featured programming language with a sound mixture of dynamic and static typing, Gradualtalk [3] is the first fully-featured language with a fine-grained gradual type system. In Gradualtalk's case, the extra runtime checks needed by gradual typing impose a big runtime cost [4], and the programmer has the option of turning off these checks, and downgrading Gradualtalk to an optional type system.

Grace Black et al. [5] is an object-oriented language with optional typing. Grace is not an dynamically typed language that has been extended with an optional type system, but a language that has been designed from the ground up to have both static and dynamic typing. Homer et al. [15] explores some useful patterns that derive from Grace's use of objects as modules and its brand of optional structural typing, which can also be expressed with Typed Lua's modules as tables.

The main feature of Typed Racket's type system is *occurrence typing* [30], where the type of a variable can be refined through tests using predicates that check the runtime type of the variable. As these types of checks are common in other languages, but ocurrence typing is not sound in the presence of mutation, related systems have appeared [13, 32, 20].

Tidal Lock [11] is a prototype of another optional type system for Lua. Compared with Typed Lua, Tidal Lock has richer record types, and a more robust system of incremental evolution of these record types, but it lacks unions and support for expressing objects and classes.

8. CONCLUSION

This paper presented the initial design of Typed Lua, an optional type system for the Lua scripting language. While Lua shares several characteristics with other dynamic languages such as JavaScript, so Typed Lua's type system has several parts in common with optional and gradual type systems for dynamic languages, Lua also has some uncommon features.

These language features demanded features in the type system that are not present in the type systems of these other languages: functions with flexible arity and their interaction with multiple assignment, functions that are overloaded on the *number* of values they return, and how to support the idiomatic way of handling their return values, and incremental evolution of record and object types.

Typed Lua is a work in progress: while the design covers most of the Lua language as it is actually used by Lua programmers, there are missing parts. A major omission of the current version is the lack of polymorphic function and table types, which will be present in the next version. The `setmetatable` primitive that we briefly saw in Section 6 has several other uses that are currently outside of the scope of the type system: operator overloading, proxies, changing the behavior of built-in types, multiple inheritance, etc. Finally, Lua has one-shot delimited continuations [17] in the form of *couroutines* [19], which are also currently ignored by the type system.

Unlike some of the other optional type systems, the type system of Typed Lua does not have deliberately unsound parts, but we still do not have proofs that the novel parts of our type system are sound.

Finally, while we our survey of Lua modules gives some confidence that the design of Typed Lua is useful for the majority of Lua programs, we have to validate the design by using Typed Lua to to type a representative sample of Lua modules.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 213–227, New York, NY, USA, 1989. ACM.

[2] A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 201–214, New York, NY, USA, 2011. ACM.

[3] E. Allende, O. Callaú, J. Fabry, É. Tanter, and M. Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, Aug. 2013.

[4] E. Allende, J. Fabry, and E. Tanter. Cast insertion strategies for gradually-typed objects. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 27–36, New York, NY, USA, 2013. ACM.

[5] A. P. Black, K. B. Bruce, M. Homer, J. Noble, A. Ruskin, and R. Yannow. Seeking Grace: A new object-oriented language for novices. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 129–134, New York, NY, USA, 2013. ACM.

[6] G. Bracha. The Strongtalk type system for Smalltalk. In *Proceedings of the Workshop on Extending the Smalltalk Language*, OOPSLA, 1996.

[7] G. Bracha. Pluggable type systems. In *Proceedings of the Workshop on Revival of Dynamic Languages*, OOPSLA, October 2004.

[8] G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 215–230, New York, NY, USA, 1993. ACM.

[9] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 278–292, New York, NY, USA, 1991. ACM.

[10] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. ACM.

[11] F. Fleutot. Tidal Lock: Gradual static typing for Lua. `https://github.com/fab13n/metalua/tree/tilo/src/tilo`, 2013. [Visited on Februrary 2014].

[12] Google. Dart. `https://www.dartlang.org/`, 2011. [Visited on February 2014].

[13] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, pages 256–275, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19717-8. URL `http://dl.acm.org/citation.cfm?id=1987211.1987225`.

[14] F. Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.

[15] M. Homer, K. B. Bruce, J. Noble, and A. P. Black. Modules as gradually-typed objects. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, DYLA '13, pages 1:1–1:8, New York, NY, USA, 2013. ACM.

[16] L. Ina and A. Igarashi. Gradual typing for generics. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 609–624, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048114. URL `http://doi.acm.org/10.1145/2048066.2048114`.

[17] R. P. James and A. Sabry. Yield: Mainstream delimited continuations. In *First International Workshop on the Theory and Practice of Delimited Continuations (TPDC 2011)*, page 20, 2011.

[18] Microsoft. TypeScript. `http://www.typescriptlang.org/`, 2012. [Visited on Februrary 2014].

[19] A. L. D. Moura and R. Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2):6:1–6:31, Feb. 2009. ISSN 0164-0925. doi: 10.1145/1462166.1462167. URL `http://doi.acm.org/10.1145/1462166.1462167`.

[20] D. J. Pearce. A calculus for constraint-based flow typing. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs*, FTfJP '13, pages 7:1–7:7, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2042-9. doi: 10.1145/2489804.2489810. URL `http://doi.acm.org/10.1145/2489804.2489810`.

[21] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

[22] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

[23] J. G. Siek and W. Taha. Gradual typing for objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, pages 2–27, Berlin, Heidelberg, 2007. Springer-Verlag.

[24] J. G. Siek and P. Wadler. Threesomes, with and without blame. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 365–376, New York, NY, USA, 2010. ACM.

[25] G. L. Steele, Jr. An overview of Common LISP. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, LFP '82, pages 98–107, New York, NY, USA, 1982. ACM.

[26] A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Gradual typing for first-class classes. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 793–810, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384674. URL `http://doi.acm.org/10.1145/2384616.2384674`.

[27] S. Thatte. Quasi-static typing. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 367–381, New York, NY, USA, 1990. ACM.

[28] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 964–974, New York, NY, USA, 2006. ACM.

[29] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 395–406, New York, NY, USA, 2008. ACM.

[30] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 117–128, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863561. URL `http://doi.acm.org/10.1145/1863543.1863561`.

[31] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 1–16, Berlin, Heidelberg, 2009. Springer-Verlag.

[32] J. Winther. Guarded type promotion: Eliminating redundant casts in java. In *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs*, FTfJP '11, pages 6:1–6:8, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0893-9. doi: 10.1145/2076674.2076680. URL `http://doi.acm.org/10.1145/2076674.2076680`.