2
Blending static and dynamic typing

We begin this chapter presenting a little bit of the history behind combining static and dynamic typing in the same language. Then, we introduce optional type systems and gradual typing. After that, we discuss why optional type systems and two other approaches are often called gradual typing. We end this chapter presenting some statistics about the usage of some Lua features and idioms that helped us identify how we should combine static and dynamic typing in Lua.

2.1 A little bit of history

Common LISP [Ste82] introduced optional type annotations in the early eighties, but not for static type checking. Instead, programmers could choose to declare types of variables as optimization hints to the compiler, that is, type declarations are just one way to help the compiler to optimize code. These annotations are unsafe because they can crash the program when they are wrong.

Abadi et al. [ACPP89] extended the simply typed lambda calculus with the `Dynamic` type and the `dynamic` and `typecase` constructs, with the aim to safely integrate dynamic code in statically typed languages. The `Dynamic` type is a pair `(v,T)` where `v` is a value and `T` is the tag that represents the type of `v`. The constructs `dynamic` and `typecase` are explicit injection and projection operations, respectively. That is, `dynamic` builds values of type `Dynamic` and `typecase` safely inspects the type of a `Dynamic` value. Thus, migrating code between dynamic and static type checking requires changing type annotations and adding or removing `dynamic` and `typecase` constructs throughout the code.

The *quasi-static* type system proposed by Thatte [Tha90] performs implicit coercions and run-time checks to replace the `dynamic` and `typecase` constructs that were proposed by Abadi et al. [ACPP89]. To do that, quasi-static typing relies on subtyping with a top type Ω that represents the dynamic type, and splits type checking into two phases. The first phase inserts implicit

coercions from the dynamic type to the expected type, while the second phase performs what Thatte calls *plausibility checking*, that is, it rewrites the program to guarantee that sequences of upcasts and downcasts always have a common subtype.

Soft typing [CF91] is another approach to combine static and dynamic typing in the same language. The main goal of soft typing is to add static type checking to dynamically typed languages without compromising their flexibility. To do that, soft typing relies on type inference for translating dynamically typed code to statically typed code. The type checker inserts run-time checks around inconsistent code and warns the programmer about the insertion of these run-time checks, as they indicate the existence of potential type errors. However, the programmer is free to choose between inspecting the run-time checks or simply running the code. This means that type inference and static type checking do not prevent the programmer from running inconsistent code. One advantage of soft typing is the fact that the compiler for softly typed languages can use the translated code to generate more efficient code, as the translated code statically type checks. One disadvantage of soft typing is that it can be cumbersome when the inferred types are meaningless large types that just confuse the programmer.

Dynamic typing [Hen94] is an approach that optimizes code from dynamically typed languages by eliminating unnecessary checks of tags. Henglein describes how to translate dynamically typed code into statically typed code that uses a `Dynamic` type. The translation is done through a coercion calculus that uses type inference to insert the operations that are necessary to type check the `Dynamic` type during run-time. Although soft typing and dynamic typing may seem similar, they are not. Soft typing targets statically type checking of dynamically typed languages for detecting programming errors, while dynamic typing targets the optimization of dynamically typed code through the elimination of unnecessary run-time checks. In other words, soft typing sees code optimization as a side effect that comes with static type checking.

Findler and Felleisen [FF02] proposed contracts for higher-order functions and blame annotations for run-time checks. Contracts perform dynamic type checking instead of static type checking, but deferring all verifications to run-time can lead to defects that are difficult to fix, because run-time errors can show a stack trace where it is not clear to programmers if the cause of a certain run-time error is in application code or library code. Even if programmers identify that the source of a certain run-time error is in library code, they still may have problems to identify if this run-time error is due to a violation of library's contract or due to a bug, when the library is poorly documented.

In this approach, programmers can insert assertions in the form of contracts that check the input and output of higher-order functions; and the compiler adds blame annotations in the generated code to track assertion failures back to the source of the error.

BabyJ [AD03] is an object-oriented language without inheritance that allows programmers to incrementally annotate the code with more specific types. Programmers can choose between using the dynamically typed version of BabyJ when they do not need types at all, and the statically typed version of BabyJ when they need to annotate the code. In statically typed BabyJ, programmers can use the *permissive type* $*$ to annotate the parts of the code that still do not have a specific type or the parts of the code that should have dynamic behavior. The type system of BabyJ is nominal, so types are either class names or the permissive type $*$. However, the type system does not use type equality or subtyping, but the relation $\approx$ between two types. The relation $\approx$ holds when both types have the same name or any of them is the permissive type $*$. Even though the permissive type $*$ is similar to the dynamic type from previous approaches, BabyJ does not provide any way to add implicit or explicit run-time checks.

Ou et al. [OTMW04] specified a language that combines static types with dependent types. To ensure safety, the compiler automatically inserts coercions between dependent code and static code. The coercions are run-time checks that ensure static code does not crash dependent code during run-time.

2.2 Optional Type Systems

Optional type systems [Bra04] are an approach for plugging static typing in dynamically typed languages. They use optional type annotations to perform compile-time type checking, though they do not affect the original run-time semantics of the language. This means that the run-time semantics should still catch type errors independently of the static type checking. For instance, we can view the typed lambda calculus as an optional type system for the untyped lambda calculus, because both have the same semantic rules and the type system serves only for discarding programs that may have undesired behaviors [Bra04].

Strongtalk [BG93, Bra96] is a version of Smalltalk that comes with an optional type system. It has a polymorphic type system that programmers can use to annotate Smalltalk code or leave type annotations out. Strongtalk assigns a dynamic type to unannotated expressions and allows programmers to cast unannotated expressions to any static type. This means that the interaction of the dynamic type with the rest of the type system is unsound, so

Strongtalk uses the original run-time semantics of Smalltalk when executing programs, even if programs are statically typed.

Pluggable type systems [Bra04] generalize the idea of optional type systems that Strongtalk put in practice. The idea is to have different optional type systems that can be layered on top of a dynamically typed language without affecting its original run-time semantics. Although these systems can be unsound in their interaction with the dynamically typed part of the language or even by design, their unsoundness does not affect run-time safety, as the language run-time semantics still catches any run-time errors caused by an unsound type system.

Dart [Goo11] and TypeScript [Mic12] are new languages that are designed with an optional type system. Both use JavaScript as their code generation target because their main purpose is Web development. In fact, Dart is a new class-based object-oriented language with optional type annotations and semantics that resembles the semantics of Smalltalk, while TypeScript is a strict superset of JavaScript that provides optional type annotations and class-based object-oriented programming. Dart has a nominal type system, while TypeScript has a structural one, but both are unsound by design. For instance, Dart has covariant arrays, while TypeScript has covariant parameter types in function signatures, besides the interaction between statically and dynamically typed code that is also unsound.

There is no common formalization for optional type systems, and each language ends up implementing its optional type system in its own way. Strongtalk, Dart, and TypeScript provide an informal description of their optional type systems rather than a formal one. In the next section we will show that we can use some features of gradual typing [ST06, ST07] to formalize optional type systems.

2.3 Gradual Typing

The main goal of gradual typing [ST06] is to allow programmers to choose between static and dynamic typing in the same language. To do that, Siek and Taha [ST06] extended the simply typed lambda calculus with the dynamic type ?, as we can see in Figure 2.1. In gradual typing, type annotations are optional, and an untyped variable is syntactic sugar for a variable whose declared type is the dynamic type ?, that is, $\lambda x.e$ is equivalent to $\lambda x{:}?.e$. Under these circumstances, we can view gradual typing as a way to add a dynamic type to statically typed languages.

The central idea of gradual typing is the *consistency* relation, written $T_1 \sim T_2$. The consistency relation allows implicit conversions to and from the

$T ::=$		TYPES:
	number	*base type number*
	$\mid$ **string**	*base type string*
	$\mid ?$	*dynamic type*
	$\mid T \to T$	*function types*
$e ::=$		EXPRESSIONS:
	l	*literals*
	$\mid x$	*variables*
	$\mid \lambda x{:}T.e$	*abstractions*
	$\mid e_1 e_2$	*application*

Figure 2.1: Syntax of the gradually-typed lambda calculus

dynamic type, and disallows conversions between inconsistent types [ST06]. For instance, **number** $\sim ?$, $? \sim$ **number**, **string** $\sim ?$, and $? \sim$ **string**, but **number** $\not\sim$ **string**, and **string** $\not\sim$ **number**. The consistency relation is both reflexive and symmetric, but it is not transitive.

$$T \sim T \text{ (C-REFL)} \quad T \sim ? \text{ (C-DYNR)} \quad ? \sim T \text{ (C-DYNL)}$$

$$\frac{T_3 \sim T_1 \quad T_2 \sim T_4}{T_1 \to T_2 \sim T_3 \to T_4} \text{ (C-FUNC)}$$

Figure 2.2: The consistency relation

Figure 2.2 defines the consistency relation. The rule C-REFL is the reflexive rule. Rules C-DYNR and C-DYNL are the rules that allow conversions to and from the dynamic type ?. The rule C-FUNC resembles subtyping between function types, because it is contravariant on the argument type and covariant on the return type.

Figure 2.3 uses the consistency relation in the typing rules of the gradual type system of the simply typed lambda calculus extended with the dynamic type ?. The environment Γ is a function from variables to types, and the directive *type* is a function from literal values to types. The rule T-VAR uses the environment function Γ to get the type of a variable x. The rule T-LIT uses the directive *type* to get the type of a literal l. The rule T-ABS evaluates the expression e with an environment Γ that binds the variable x to the type T_1, and the resulting type is the the function type $T_1 \to T_2$. The rule T-APP1 handles function calls where the type of a function is dynamically typed; in this case, the argument type may have any type and the resulting type has the dynamic type. The rule T-APP2 handles function calls where the type

of a function is statically typed; in this case, the argument type should be consistent with the argument type of the function's signature.

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \text{(T-VAR)} \quad \frac{\mathit{type}(l) = T}{\Gamma \vdash l : T}\ \text{(T-LIT)}$$

$$\frac{\Gamma[x \mapsto T_1] \vdash e : T_2}{\Gamma \vdash \lambda x : T_1.e : T_1 \to T_2}\ \text{(T-ABS)} \quad \frac{\Gamma \vdash e_1 : ? \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 e_2 : ?}\ \text{(T-APP1)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \quad \Gamma \vdash e_2 : T_3 \quad T_3 \sim T_1}{\Gamma \vdash e_1 e_2 : T_2}\ \text{(T-APP2)}$$

Figure 2.3: Gradual type system gradually-typed lambda calculus

Gradual typing [ST06] is similar to two previous approaches [ACPP89, Tha90], because they also include a dynamic type in a statically typed language. However, these three approaches differ in the way they handle the dynamic type. While Siek and Taha [ST06] rely on the consistency relation, Abadi et al. [ACPP89] rely on type equality with explicit projections plus injections, and Thatte [Tha90] relies on subtyping.

The subtyping relation $<:$ is actually a pitfall on Thatte's quasi-static typing, because it sets the dynamic type as the top and the bottom of the subtying relation: $T <: ?$ and $? <: T$. Subtyping is transitive, so we know that

$$\frac{\textbf{number} <: ? \quad ? <: \textbf{string}}{\textbf{number} <: \textbf{string}}$$

Therefore, downcasts combined with the transitivity of subtyping accepts programs that should be rejected.

Later, Siek and Taha [ST07] reported that the consistency relation is orthogonal to the subtyping relation, so we can combine them to achieve the *consistent-subtyping* relation, written $T_1 \lesssim T_2$. This relation is essential for designing gradual type systems for object-oriented languages. Like the consistency relation, and unlike the subtyping relation, the consistent-subtyping relation is not transitive. Therefore, $\textbf{number} \lesssim ?$, $? \lesssim \textbf{number}$, $\textbf{string} \lesssim ?$, and $? \lesssim \textbf{string}$, but $\textbf{number} \not\lesssim \textbf{string}$, and $\textbf{string} \not\lesssim \textbf{number}$.

Now, we will show how we can combine consistency and subtyping to compose a consistent-subtyping relation for the simply typed lambda calculus extended with the dynamic type ?.

Figure 2.4 presents the subtyping relation for the simply typed lambda calculus extended with the dynamic type ?. Even though we could have used

$$\textbf{number} <: \textbf{number}\ \text{(S-NUM)} \quad \textbf{string} <: \textbf{string}\ \text{(S-STR)}$$

$$? <: ?\ \text{(S-ANY)} \quad \frac{T_3 <: T_1 \quad T_2 <: T_4}{T_1 \to T_2 <: T_3 \to T_4}\ \text{(S-FUN)}$$

Figure 2.4: The subtyping relation

the reflexive rule $T <: T$ to express the rules S-NUM, S-STR, and S-ANY, we did not combine them into a single rule to make explicit the neutrality of the dynamic type ? to the subtyping rules. The dynamic type ? must be neutral to subtyping to avoid the pitfall from Thatte's quasi-static typing. The rule S-FUN defines the subtyping relation for function types, which are contravariant on the argument type and covariant on the return type.

$$\textbf{number} \lesssim \textbf{number}\ \text{(C-NUM)} \quad \textbf{string} \lesssim \textbf{string}\ \text{(C-STR)}$$

$$T \lesssim ?\ \text{(C-ANY1)} \quad ? \lesssim T\ \text{(C-ANY2)}$$

$$\frac{T_3 \lesssim T_1 \quad T_2 \lesssim T_4}{T_1 \to T_2 \lesssim T_3 \to T_4}\ \text{(C-FUN)}$$

Figure 2.5: The consistent-subtyping relation

Figure 2.5 combines the consistency and subtyping relations to compose the consistent-subtyping relation for the simply typed lambda calculus extended with the dynamic type ?. When we combine consistency and subtyping, we are making subtyping handle which casts are safe among static types, and we are making consistency handle the casts that involve the dynamic type ?. The consistent-subtyping relation is not transitive, and thus the dynamic type ? is not neutral to this relation.

So far, gradual typing looks like a mere formalization to optional type systems, as a gradual type system uses the consistency or consistent-subtyping relation to statically check the interaction between statically and dynamically typed code, without affecting the run-time semantics.

However, another important feature of gradual typing is the theoretic foundation that it provides for inserting run-time checks that prove dynamically typed code does not violate the invariants of statically typed code, thus preserving type safety. To do that, Siek and Taha [ST06, ST07] defined the run-time semantics of gradual typing as a translation to an intermediate language with explicit casts at the frontiers between statically and dynamically

typed code. The semantics of these casts is based on the higher-order contracts proposed by Findler and Felleisen [FF02].

Herman et al. [HTF07] showed that there is an efficiency concern regarding the run-time checks, because there are two ways that casts can lead to unbounded space consumption. The first affects tail recursion while the second appears when first-class functions or objects cross the border between static code and dynamic code, that is, some programs can apply repeated casts to the same function or object. Herman et al. [HTF07] use the coercion calculus outlined by Henglein [Hen94] to express casts as coercions and solve the problem of space efficiency. Their approach normalizes an arbitrary sequence of coercions to a coercion of bounded size.

Another concern about casts is how to improve debugging support, because a cast application can be delayed and the error related to that cast application can appear considerable distance from the real error. Wadler and Findler [WF09] developed *blame calculus* as a way to handle this issue, and Ahmed et al. [AFSW11] extended blame calculus with polymorphism. Blame calculus is an intermediate language to integrate static and dynamic typing along with the blame tracking approach proposed by Findler and Felleisen [FF02].

On the one hand, blame calculus solves the issue regarding error reporting; on the other hand, it has the space efficiency problem reported by Herman et al. [HTF07]. Thus, Siek et al. [SGT09] extended the coercion calculus outlined by Herman et al. [HTF07] with blame tracking to achieve an implementation of the blame calculus that is space efficient. After that, Siek and Wadler [SW10] proposed a new solution that also handles both problems. This new solution is based on a concept called *threesome*, which is a way to split a cast between two parties into two casts among three parties. A cast has a source and a target type (a *twosome*), so we can split any cast into a downcast from the source to an intermediate type that is followed by an upcast from the intermediate type to the target type (a *threesome*).

There are some projects that incorporate gradual typing into some programming languages. Reticulated Python [Vit13, VKSB14] is a research project that evaluates the costs of gradual typing in Python. Gradualtalk [ACF+13] is a gradually-typed Smalltalk that introduces a new cast insertion strategy for gradually-typed objects [AFT13]. Grace [BBHN12, BBH+13] is a new object-oriented, gradually-typed, educational language. In Grace, modules are gradually-typed objects, that is, modules may have types with methods as attributes, and they can also have a state [HBNB13]. ActionScript [Moo07] is one the first languages that incorporated gradual typing to its implementation

and Perl 6 [Tan07] is also being designed with gradual typing, though there is few documentation about the gradual type systems of these languages.

2.4 Approaches that are often called Gradual Typing

Gradual typing is similar to optional type systems in that type annotations are optional, and unannotated code is dynamically typed, but unlike optional type systems, gradual typing changes the run-time semantics to preserve type safety. More precisely, programming languages that include a gradual type system can implement the semantics of statically typed languages, so the gradual type system inserts casts in the translated code to guarantee that types are consistent before execution, while programming languages that include an optional type system still need implement the semantics of dynamically typed languages, so all the type checking also belongs to the semantics of each operation.

Still, we can view gradual typing as a way to formalize an optional type system when the gradual type system does not insert run-time checks. BabyJ [AD03] and Alore [LG11] are two examples of object-oriented languages that have an optional type system with a formalization that relates to gradual typing, though the optional type systems of both BabyJ and Alore are nominal. BabyJ uses the relation $\approx$ that is similar to the consistency relation while Alore combines subtyping along with the consistency relation to define a *consistent-or-subtype* relation. The consistent-or-subtype relation is different from the consistent-subtyping relation of Siek and Taha [ST07], but it is also written $T_1 \lesssim T_2$. The consistent-or-subtype relation holds when $T_1 \sim T_2$ or $T_1 <: T_2$, where $<:$ is transitive and $\sim$ is not. Alore also extends its optional type system to include optional monitoring of run-time type errors in the gradual typing style.

Hence, optional type annotations for software evolution are likely the reason why optional type systems are commonly called gradual type systems. Typed Clojure [BS12] is an optional type system for Clojure that is now adopting the gradual typing slogan.

Flanagan [Fla06] introduced *hybrid type checking*, an approach that combines static types and *refinement* types. For instance, programmers can specify the refinement type $\{x : Int \mid x \geq 0\}$ when they need a type for natural numbers. The programmer can also choose between explicit or implicit casts. When casts are not explicit, the type checker uses a theorem prover to insert casts. In our example of natural numbers, a cast would be inserted to check

whether an integer is greater than or equal to zero.

Sage [GKT+06] is a programming language that extends hybrid type checking with a dynamic type to support dynamic and static typing in the same language. Sage also offers optional type annotations in the gradual typing style, that is, unannotated code is syntactic sugar for code whose declared type is the dynamic type.

Thus, the inclusion of a dynamic type in hybrid type checking along with optional type annotations, and the insertion of run-time checks are likely the reason why hybrid type checking is also viewed as a form of gradual typing.

Tobin-Hochstadt and Felleisen [THF06] proposed another approach for gradually migrating from dynamically typed to statically typed code, and they coined the term *from scripts to programs* for referring to this kind of interlanguage migration. In their approach, the migration from dynamically typed to statically typed code happens module-by-module, so they designed and implemented Typed Racket [THF08] for this purpose. Typed Racket is a statically typed version of Racket (a Scheme dialect) that allows the programmer to write typed modules, so Typed Racket modules can coexist with Racket modules, which are untyped.

The approach used by Tobin-Hochstadt and Felleisen [THF08] to design and implement Typed Racket is probably also called gradual typing because it allows the programmer to gradually migrate from untyped scripts to typed programs. However, Typed Racket is a statically typed language, and what makes it gradual is a type system with a dynamic type that handles the interaction between Racket and Typed Racket modules.

Recently, Siek et al. [SVCB15] described a formal criteria on what is gradual typing: the *gradual guarantee*. Besides allowing static and dynamic typing in the same code along with type soundness, the gradual guarantee states that removing type annotations from a gradually typed program that is well typed must continue well typed. The other direction must be also valid, that is, adding correct type annotations to a gradually typed program that is well typed must continue well typed. In other words, the gradual guarantee states that any changes to the annotations does not change the static or the dynamic behavior of a program [SVCB15]. The authors prove the gradual guarantee and discuss whether some previous projects match this criteria.

2.5 Statistics about the usage of Lua

In this section we present statistics about the usage of Lua features and idioms. We collected statistics about how programmers use tables, functions, dynamic type checking, object-oriented programming, and modules. We shall

see that these statistics informed important design decisions on our optional type system.

We used the LuaRocks repository to build our statistics database; LuaRocks [MMI13] is a package manager for Lua modules. We downloaded the 3928 `.lua` files that were available in the LuaRocks repository at February 1st 2014. However, we ignored files that were not compatible with Lua 5.2, the latest version of Lua at that time. We also ignored *machine-generated* files and test files, because these files may not represent idiomatic Lua code, and might skew our statistics towards non-typical uses of Lua. This left 2598 `.lua` files from 262 different projects for our statistics database; we parsed these files and processed their abstract syntax tree to gather the statistics that we show in this section.

To verify how programmers use tables, we measured how they initialize, index, and iterate tables. We present these statistics in the next three paragraphs to discuss their influence on our type system.

The table constructor appears 23185 times. In 36% of the occurrences it is a constructor that initializes a record (e.g., `{ x = 120, y = 121 }`); in 29% of the occurrences it is a constructor that initializes a list (e.g., `{ "one", "two", "three", "four" }`); in 8% of the occurrences it is a constructor that initializes a record with a list part; and in less than 1% of the occurrences (4 times) it is a constructor that uses only the booleans `true` and `false` as indexes. At all, in 73% of the occurrences it is a constructor that uses only literal keys; in 26% of the occurrences it is the empty constructor; in 1% of the occurrences it is a constructor with non-literal keys only, that is, a constructor that uses variables and function calls to create the indexes of a table; and in less than 1% of the occurrences (19 times) it is a constructor that mixes literal keys and non-literal keys.

The indexing of tables appears 130448 times: 86% of them are for reading a table field while 14% of them are for writing into a table field. We can classify the indexing operations that are reads as follows: 89% of the reads use a literal string key, 4% of the reads use a literal number key, and less than 1% of the reads (10 times) use a literal boolean key. At all, 93% of the reads use literals to index a table while 7% of the reads use non-literal expressions to index a table. It is worth mentioning that 45% of the reads are actually function calls. More precisely, 25% of the reads use literals to call a function, 20% of the reads use literals to call a method, that is, a function call that uses the colon syntactic sugar, and less than 1% of the reads (195 times) use non-literal expressions to call a function. We can also classify the indexing operations that are writes as follows: 69% of the writes use a literal string key, 2% of the writes use a literal

number key, and less than 1% of the writes (1 time) uses a literal boolean key. At all, 71% of the writes use literals to index a table while 29% of the writes use non-literal expressions to index a table.

We also measured how many files have code that iterates over tables to observe how frequently iteration is used. We observed that 23% of the files have code that iterates over keys of any value, that is, the call to `pairs` appears at least once in these files (the median is twice per file); 21% of the files have code that iterates over integer keys, that is, the call to `ipairs` appears at least once in these files (the median is also twice per file); and 10% of the files have code that use the numeric `for` along with the length operator (the median is once per file).

The numbers about table initialization, indexing, and iteration show us that tables are mostly used to represent records, lists, and associative arrays. Therefore, Typed Lua should include a table type for handling these uses of Lua tables. Even though the statistics show that programmers initialize tables more often than they use the empty constructor to dynamically initialize tables, the statistics of the empty constructor are still expressive and indicate that Typed Lua should also include a way to handle this style of defining table types.

We found a total of 24858 function declarations in our database (the median is six per file). Next, we discuss how frequently programmers use dynamic type checking and multiple return values inside these functions.

We observed that 9% of the functions perform dynamic type checking on their input parameters, that is, these functions use `type` to inspect the tags of Lua values (the median is once per function). We randomly selected 20 functions to sample how programmers are using `type`, and we got the following data: 50% of these functions use `type` for asserting the tags of their input parameters, that is, they raise an error when the tag of a certain parameter does not match the expected tag, and 50% of these functions use `type` for overloading, that is, they execute different code according to the inspected tag.

These numbers show us that Typed Lua should include union types, because the use of the `type` idiom shows that disjoint unions would help programmers define data structures that can hold a value of several different, but fixed types. Typed Lua should also use `type` as a mechanism for decomposing unions, though it may be restricted to base types only.

We observed that 10% of the functions explicitly return multiple values. We also observed that 5% of the functions return `nil` plus something else, for signaling an unexpected behavior; and 1% of the functions return `false` plus something else, also for signaling an unexpected behavior.

Typed Lua should include function types to represent Lua functions, and tuple types to represent the signatures of Lua functions, multiple return values, and multiple assignments. Tuple types require some special attention, because Typed Lua should be able to adjust tuple types during compile-time, in a similar way to what Lua does with function calls and multiple assignments during run-time. In addition, the number of functions that return `nil` and `false` plus something else show us that overloading on the return type is also useful to the type system.

We also measured how frequently programmers use the object-oriented paradigm in Lua. We observed that 23% of the function declarations are actually method declarations. More precisely, 14% of them use the colon syntactic sugar while 9% of them use `self` as their first parameter. We also observed that 63% of the projects extend tables with metatables, that is, they call `setmetatable` at least once, and 27% of the projects access the metatable of a given table, that is, they call `getmetatable` at least once. In fact, 45% of the projects extend tables with metatables and declare methods: 13% using the colon syntactic sugar, 14% using `self`, and 18% using both.

Based on these observations, Typed Lua should include support to object-oriented programming. Even though Lua does not have standard policies for object-oriented programming, it provides mechanisms that allow programmers to abstract their code in terms of objects, and our statistics confirm that an expressive number of programmers are relying on these mechanisms to use the object-oriented paradigm in Lua. Typed Lua should include some standard way of defining interfaces and classes that the compiler can use to type check object-oriented code, but without changing the semantics of Lua.

We also measured how programmers are defining modules. We observed that 38% of the files use the current way of defining modules, that is, these files return a table that contains the exported members of the module at the end of the file; 22% of the files still use the deprecated way of defining modules, that is, these files call the function `module`; and 1% of the files use both ways. At all, 61% of the files are modules while 39% of the files are plain scripts. The number of plain scripts is high considering the origin of our database. However, we did not ignore sample scripts, which usually serve to help the users of a given module on how to use this module, and that is the reason why we have a high number of plain scripts.

Based on these observations, Typed Lua should include a way for defining table types that represent the type of modules. Typed Lua should also support the deprecated style of module definition, using global names as exported members of the module.

Typed Lua should also include some way to define the types of userdata. This feature should also allow programmers to define userdata that can be used in an object-oriented style, as this is another common idiom from modules that are written in C.

The last statistics that we collected were about variadic functions and vararg expressions. We observed that 8% of the functions are variadic, that is, their last parameter is the vararg expression. We also observed that 5% of the initialization of lists (or 2% of the occurrences of the table constructor) use solely the vararg expression. Typed Lua should include a *vararg type* to handle variadic functions and vararg expressions.

Table 2.1 summarizes the statistics that we presented in this section.

table constructor (% per static occurrences)	create a record	36%
	create a list	29%
	create an empty table	26%
	create a table with a record part and a list part	8%
	create a table with non-literal keys	1%
table access (% per static occurrences)	reading with literal keys	80%
	writing with literal keys	10%
	reading with non-literal keys	6%
	writing with non-literal keys	4%
iteration over tables (% per files)	files that iterate over a list	27%
	files that iterate over a map	23%
function declarations (% per static occurrences)	inspect the tags of their input parameters	9%
	return multiple values to signal errors	6%
	are variadic	8%
	are method declarations	23%
object-oriented programming (% per projects)	projects that use metatables and declare methods	45%
modules (% per files)	files that are modules	61%
	files that are plain scripts	39%

Table 2.1: Summary of the statistics about the usage of Lua