puc.pdf

André Murbach Maidl

Typed Lua: An Optional Type System for Lua

TESE DE DOUTORADO

Thesis presented to the Programa de Pós Graduação em Informática of the Departamento de Informática, PUC–Rio as partial fulfillment of the requirements for the degree of Doutor em Informática

Advisor : Prof. Roberto Ierusalimschy
Co–Advisor: Prof. Fabio Mascarenhas de Queiroz

Rio de Janeiro
April 2015

puc.pdf

André Murbach Maidl

Typed Lua: An Optional Type System for Lua

Thesis presented to the Programa de Pós Graduação em Informática, of the Departamento de Informática do Centro Técnico Científico da PUC–Rio, as partial fulfillment of the requirements for the degree of Doutor.

Prof. Roberto Ierusalimschy
Advisor
Departmento de Informática — PUC–Rio

Prof. Fabio Mascarenhas de Queiroz
Co–Advisor
UFRJ

Prof. Ana Lúcia de Moura
Departamento de Informática — PUC-Rio

Prof. Edward Hermann Haeusler
Departamento de Informática — PUC-Rio

Prof. Anamaria Martins Moreira
UFRJ

Prof. Roberto da Silva Bigonha
UFMG

Prof. José Eugênio Leal
Coordinator of the Centro Técnico Científico da PUC–Rio

Rio de Janeiro, April 10th, 2015

André Murbach Maidl

Aqui vai o meu resume.

Bibliographic data

Maidl, André Murbach

Typed Lua: An Optional Type System for Lua / André Murbach Maidl; advisor: Roberto Ierusalimschy; co–advisor: Fabio Mascarenhas de Queiroz. — 2015.

?? f. : il. ; 30 cm

Tese (Doutorado em Informática) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2015.

Inclui bibliografia.

1. Informática – Teses. 2. Linguagens de script. 3. Lua. 4. Sistemas de tipos. 5. Sistemas de tipos opcionais. 6. Tipagem gradual. I. Ierusalimschy, Roberto. II. Queiroz, Fabio Mascarenhas de. III. Pontifícia Universidade Católica do Rio de Janeiro. Departmento de Informática. IV. Título.

CDD: 004

Acknowledgments

Abstract

Maidl, André Murbach; Ierusalimschy, Roberto; Queiroz, Fabio Mascarenhas de. **Typed Lua: An Optional Type System for Lua**. Rio de Janeiro, 2015. ??p. DSc Thesis — Departmento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Dynamically typed languages such as Lua avoid static types in favor of simplicity and flexibility, because the absence of static types means that programmers do not need to bother with abstracting types that should be validated by a type checker. In contrast, statically typed languages provide the early detection of many bugs, and a better framework for structuring large programs. These are two advantages of static typing that may lead programmers to migrate from a dynamically typed to a statically typed language, when their simple scripts evolve into complex programs.
Optional type systems allow combining dynamic and static typing in the same language, without affecting its original semantics, making easier this code evolution from dynamic to static typing. Designing an optional type system for a dynamically typed language is challenging, as it should feel natural to programmers that are already familiar with this language.
In this work we present and formalize the design of Typed Lua, an optional type system for Lua that introduces novel features to statically type check some Lua idioms and features. Even though Lua shares several characteristics with other dynamically typed languages such as JavaScript, Lua also has several unusual features that are not present in the type system of these languages. These features include functions with flexible arity, multiple assignment, functions that are overloaded on the number of return values, and the incremental evolution of record and object types. We discuss how Typed Lua handles these features and our design decisions. Finally, we present the evaluation results that we achieved while using Typed Lua to type existing Lua code.

Keywords

Scripting languages. Lua. Type systems. Optional type systems. Gradual typing.

Resumo

Maidl, André Murbach; Ierusalimschy, Roberto; Queiroz, Fabio Mascarenhas de. **Typed Lua: um sistema de tipos opcional para Lua**. Rio de Janeiro, 2015. ??p. Tese de Doutorado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Linguagens dinamicamente tipadas, tais como Lua, não usam tipos estáticos em favor de simplicidade e flexibilidade, porque a ausência de tipos estáticos significa que programadores não precisam se preocupar em abstrair tipos que devem ser validados por um verificador de tipos. Por outro lado, linguagens estaticamente tipadas ajudam na detecção prévia de diversos *bugs* e também ajudam na estruturação de programas grandes. Tais pontos geralmente são vistos como duas vantagens que levam programadores a migrar de uma linguagem dinamicamente tipada para uma linguagem estaticamente tipada, quando os pequenos *scripts* deles evoluem para programas complexos.
Sistemas de tipos opcionais nos permitem combinar tipagem dinâmica e estática na mesma linguagem, sem afetar a semântica original da linguagem, tornando mais fácil a evolução de código tipado dinamicamente para código tipado estaticamente. Desenvolver um sistema de tipos opcional para uma linguagem dinamicamente tipada é uma tarefa desafiadora, pois ele deve ser o mais natural possível para os programadores que já estão familiarizados com essa linguagem.
Neste trabalho nós apresentamos e formalizamos Typed Lua, um sistema de tipos opcional para Lua, o qual introduz novas características para tipar estaticamente alguns idiomas e características de Lua. Embora Lua compartilhe várias características com outras linguagens dinamicamente tipadas, em particular JavaScript, Lua também possui várias características não usuais, as quais não estão presentes nos sistemas de tipos dessas linguagens. Essas características incluem funções com aridade flexível, atribuições múltiplas, funções que são sobrecarregadas no número de valores de retorno e a evolução incremental de registros e objetos. Nós discutimos como Typed Lua tipa estaticamente essas características e também discutimos nossas decisões de projeto. Finalmente, apresentamos uma avaliação de resultados, a qual obtivemos ao usar Typed Lua para tipar código Lua existente.

Palavras–chave

Linguagens de script. Lua. Sistemas de tipos. Sistemas de tipos opcionais. Tipagem gradual.

Contents

List of Figures

List of Tables

1 Introduction

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

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

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

Ideally, programming languages should offer programmers the option to choose between static and dynamic typing: *optional type systems* [?] and *gradual typing* [?] are two similar approaches for blending static and dynamic typing in the same language. The aim of both approaches is to offer programmers the option to use type annotations where static typing is needed, allowing the incremental migration from dynamic to static typing. The difference between these two approaches is the way they treat run-time semantics. While optional type systems do not affect run-time semantics, gradual typing uses run-time checks to ensure that dynamically typed code does not violate the invariants of statically typed code.

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

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

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

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

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

So far, Typed Lua is a Lua extension that allows statically typed code to coexist and interact with dynamically typed code through optional type annotations. In addition, it adds default constructs that programmers can use to better structure Lua programs. The Typed Lua compiler warns programmers about type errors, but always generates Lua code that runs in unmodified Lua implementations. Programmers can enjoy some of the benefits of static types even without converting existing Lua modules to Typed Lua – they can export a statically typed interface to a dynamically typed module, and statically typed users of the module can use the Typed Lua compiler to check their use of the module. Thus, implementing an optional type system for Lua offers Lua programmers one way to obtain most of the advantages of static typing without compromising the simplicity and flexibility of dynamic typing. We have an implementation of the Typed Lua compiler that is available online[1].

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

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

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

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

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

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

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

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

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 [?] 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. [?] 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 [?] performs implicit coercions and run-time checks to replace the `dynamic` and `typecase` constructs that were proposed by Abadi et al. [?]. 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 [?] 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 [?] 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 [?] 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 [?] 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. [?] 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 [?] 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 [?].

Strongtalk [?, ?] 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 [?] 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 [?] and TypeScript [?] 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 [?, ?] to formalize optional type systems.

2.3 Gradual Typing

The main goal of gradual typing [?] is to allow programmers to choose between static and dynamic typing in the same language. To do that, Siek and Taha [?] extended the simply typed lambda calculus with the dynamic type ?, as we can see in Figure **??**. 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

$$
\begin{array}{lll}
T ::= & & \text{TYPES:} \\
& \textbf{number} & \textit{base type number} \\
& \mid \textbf{string} & \textit{base type string} \\
& \mid ? & \textit{dynamic type} \\
& \mid T \rightarrow T & \textit{function types} \\
e ::= & & \text{EXPRESSIONS:} \\
& l & \textit{literals} \\
& \mid x & \textit{variables} \\
& \mid \lambda x{:}T.e & \textit{abstractions} \\
& \mid e_1 e_2 & \textit{application}
\end{array}
$$

Figure 2.1: Syntax of the gradually-typed lambda calculus

the dynamic type, and disallows conversions between inconsistent types [**?**]. 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.

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

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

Figure 2.2: The consistency relation

Figure **??** 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 **??** 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 $T_1 \rightarrow T_2$. 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.

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

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

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

Figure 2.3: Gradual type system gradually-typed lambda calculus

Gradual typing [?] is similar to two previous approaches [?, ?], 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 [?] rely on the consistency relation, Abadi et al. [?] rely on type equality with explicit projections plus injections, and Thatte [?] 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 [?] 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 **??** 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 **??** 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 [**?**, **?**] 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 [?].

Herman et al. [?] 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. [?] use the coercion calculus outlined by Henglein [?] 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 [?] developed *blame calculus* as a way to handle this issue, and Ahmed et al. [?] 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 [?].

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. [?]. Thus, Siek et al. [?] extended the coercion calculus outlined by Herman et al. [?] with blame tracking to achieve an implementation of the blame calculus that is space efficient. After that, Siek and Wadler [?] 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 [?, ?] is a research project that evaluates the costs of gradual typing in Python. Gradualtalk [?] is a gradually-typed Smalltalk that introduces a new cast insertion strategy for gradually-typed objects [?]. Grace [?, ?] 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 [?]. ActionScript [?] is one the first languages that incorporated gradual typing to its implementation and Perl 6 [?] 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 [?] and Alore [?] 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 [?], 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 [?] is an optional type system for Clojure that is now adopting the gradual typing slogan.

Flanagan [?] 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 [?] 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 [?] 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 [?] 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 [?] 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. [?] 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 [?]. 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 [?] 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.

We end this chapter presenting a summary of the statistics that we just presented:

- table constructor
  - only use literal keys: 73%
  - empty: 26%
  - only use non-literal keys: 1%
- table indexing
  - reads: 86%
    * use literals: 93%
    * use non-literals: 7%
  - writes: 14%
    * use literals: 71%
    * use non-literals: 29%
- iteration over tables
  - files that call `pairs`: 23%
  - files that call `ipairs`: 21%
  - files that use the numeric for along with the length operator: 10%
- function declarations
  - perform dynamic type checking in their input parameters: 9%
    * asserting input parameters: 50%
    * overloading input parameters: 50%
  - return multiple values: 10%
    * return `nil` plus something else: 50%
    * return `false` plus something else: 10%
    * return other multiple values: 40%

- are method declarations: 23%
- are variadic: 8%

- object-oriented programming
  - projects that extend metatables and declare methods: 45%
- modules
  - files that define modules: 61%
  - files that are plain scripts: 39%

3
Typed Lua

Typed Lua is an optional type system for Lua, and its main goal is to provide static type checking for Lua. To do that, Typed Lua extends the syntax of Lua 5.3 to introduce optional type annotations, and performs local type inference [?] to detect more precise types for unannotated expressions. Even though the compiler warns the programmer about type errors, it always removes the type annotations to generate Lua code that runs in unmodified Lua implementations.

Another goal of Typed Lua is to be backwards compatible with Lua. This means that any Lua code is valid Typed Lua code. To be backwards compatible with Lua, the syntactic extensions introduced by Typed Lua do not include new reserved words. Appendix ?? presents the complete syntax of Typed Lua in extended BNF.

We use the consistent-subtyping relation of gradual typing [?, ?] to formalize Typed Lua, though it does not insert run-time checks in the gradual typing style. In gradual typing, run-time checks inspect the interaction between dynamically typed and statically typed code to guarantee that dynamically typed code does not violate statically typed code during run-time. We did not insert run-time checks at this moment because they can decrease run-time performance [?]. We believe that a careful evaluation of run-time checks should be done before inserting them in the type system. However, this evaluation is out of scope of this work.

Unlike Dart [?] and TypeScript [?], we are designing Typed Lua aiming soundness to make it possible to switch Typed Lua from optional typing to gradual typing in the future. A sound type system is a prerequisite to insert run-time checks after static type checking, because a sound type system ensures that statically typed code will not throw type errors during run-time.

In this chapter we use some examples of Typed Lua code to show how they relate to Lua. These examples give an informal overview of our optional type system. In the next chapter we will use typing rules to present the formalization of the most interesting features of our optional type system. All the examples that we present in this chapter run in our Typed Lua compiler.

3.1 Optional type annotations

Lua values can have one of eight tags: *nil*, *boolean*, *number*, *string*, *function*, *table*, *userdata*, and *thread*. Typed Lua includes types for the first six. Typed Lua also includes a syntactical extension that programmers can use to define the types of *userdata*. We use this syntactical extension to define the type *thread*. In this section we present the Typed Lua types that may appear on annotations. We explain all Typed Lua types and syntactical extensions in this chapter.

Types

$$
\begin{array}{rcl}
\mathit{type} & ::= & \mathit{primarytype}\ [\text{‘}\texttt{?}\text{’}] \\
\mathit{primarytype} & ::= & \mathit{literaltype} \mid \mathit{basetype} \mid \textbf{nil} \mid \textbf{value} \mid \textbf{any} \mid \textbf{self} \mid \mathit{Name} \\
 & & \mid \mathit{functiontype} \mid \mathit{tabletype} \mid \mathit{primarytype}\ \text{‘}\texttt{|}\text{’}\ \mathit{primarytype} \\
\mathit{literaltype} & ::= & \textbf{false} \mid \textbf{true} \mid \mathit{Int} \mid \mathit{Float} \mid \mathit{String} \\
\mathit{basetype} & ::= & \textbf{boolean} \mid \textbf{integer} \mid \textbf{number} \mid \textbf{string} \\
\mathit{functiontype} & ::= & \mathit{tupletype}\ \text{‘}\texttt{->}\text{’}\ \mathit{rettype} \\
\mathit{tupletype} & ::= & \text{‘}\texttt{(}\text{’}\ [\mathit{typelist}]\ \text{‘}\texttt{)}\text{’} \\
\mathit{typelist} & ::= & \mathit{type}\ \{\text{‘}\texttt{,}\text{’}\ \mathit{type}\}\ [\text{‘}\texttt{*}\text{’}] \\
\mathit{rettype} & ::= & \mathit{type} \mid \mathit{uniontuple}\ [\text{‘}\texttt{?}\text{’}] \\
\mathit{uniontuple} & ::= & \mathit{tupletype} \mid \mathit{uniontuple}\ \text{‘}\texttt{|}\text{’}\ \mathit{uniontuple} \\
\mathit{tabletype} & ::= & \text{‘}\texttt{\{}\text{’}\ [\mathit{tabletypebody}]\ \text{‘}\texttt{\}}\text{’} \\
\mathit{tabletypebody} & ::= & \mathit{maptype} \mid \mathit{recordtype} \\
\mathit{maptype} & ::= & [\mathit{keytype}\ \text{‘}\texttt{:}\text{’}]\ \mathit{type} \\
\mathit{keytype} & ::= & \mathit{basetype} \mid \textbf{value} \\
\mathit{recordtype} & ::= & \mathit{recordfield}\ \{\text{‘}\texttt{,}\text{’}\ \mathit{recordfield}\}\ [\text{‘}\texttt{,}\text{’}\ \mathit{type}] \\
\mathit{recordfield} & ::= & [\textbf{const}]\ \mathit{literaltype}\ \text{‘}\texttt{:}\text{’}\ \mathit{type}
\end{array}
$$

Figure 3.1: The concrete syntax of Typed Lua types

Figure **??** presents the concrete syntax of Typed Lua types in extended BNF. We classify Typed Lua types into two categories: *first-level types* and *second-level types*. First-level types consist of *type* and represent Lua values, while second-level types consist of either *tupletype* or *rettype* and represent the type of expression lists, multiple assignments, and function applications. First-level types include literal types, base types, the type `nil`, the top type `value`, the dynamic type `any`, the self type `self`, named types, function types, table types, and union types. Second-level types include the type void `()`, vararg types, tuple types, and unions of tuple types.

Typed Lua uses subtyping to order types. Any first-level type is a subtype of `value`. Union types are supertypes of their parts. The base types `boolean`, `integer`, `number`, and `string` are supertypes of their respective literal types. The base type `integer` is subtype of `number`. Function types are related by contravariance on the input and covariance on the output. Table types have width subtyping, with depth subtyping on `const` fields. Tuple and vararg types are covariant. Unions of tuple types are also supertypes of their parts. We will present the formalization of the subtyping relation in Section **??**.

Typed Lua uses consistent-subtyping to check the interaction among the dynamic type `any` and other types. The dynamic type `any` is a subtype of `value`, but it is neither a supertype nor a subtype of any other type. Our consistent-subtyping relationship follows the standards defined by the gradual typing of objects [**?**, **?**]. In practice, we can pass a value of the dynamic type anytime we want a value of some other type, and we can pass any value where a value of the dynamic type is expected, but the compiler tracks these operations, and the programmer can choose to be warned about them. We will discuss the formalization of the consistent-subtyping relation in Section **??**.

Before we start discussing examples of Typed Lua code, it is worth mentioning that there is a subtle difference between the dynamic type `any` and the top type `value`. Although both types mean that they accept a value of any other type, the type `value` is not a good option for handling the interaction between dynamically typed and statically typed code. Gradual typing uses the dynamic type `any` to identify where it should insert run-time checks for asserting that dynamically typed code does not violate statically typed code. Typed Lua also uses the dynamic type `any` in this sense, though it is an optional type system. More precisely, we use `any` instead of `value` to allow programmers blending dynamic and static typing because we use the consistent-subtyping relation to formalize our optional type system, as it is a first step to switch Typed Lua from optional typing to gradual typing in the future.

Typed Lua allows optional type annotations in variable and function declarations. We use the following example to illustrate how we can annotate a function declaration and a variable declaration:

```
local function succ (n:integer):integer
  return n + 1
end
local x:integer = 7
x = succ(x)
print(x)      --> 8
```

Typed Lua uses local type inference to assign more specific types to some unannotated declarations. More precisely, Typed Lua can infer the type of local variables and the return type of local functions that are not recursive. The inference that we implement in Typed Lua is quite simple, as it uses only the type of the local expression. For local variables, Typed Lua uses the type of the initialization expression to assign a more specific type to an unannotated local variable. For local functions, Typed Lua uses the type of the returned expression to assign a more specific type to an unannotated return type.

This means that we can rewrite the previous example to use local type inference for inferring the return type of `succ` and also for inferring the type of `x`:

```
local function succ (n:integer)
  return n + 1
end
local x = 7
x = succ(x)
print(x)      --> 8
```

In this example, the compiler uses local type inference to assign the type `integer` to the local variable `x` and to the return type of the local function `succ`, making this example compile without any warnings. Local type inference always uses the most general type. In this example, the compiler does not use the literal type `7`, instead of the base type `integer`, because this would generate a warning when we try to assign other integer value to the variable `x`. Still, programmers can use literal types in type annotations if they need a variable that has a very specific type. In Section **??** we will see in more detail that literal types are essential to type Lua tables.

Typed Lua assigns the dynamic type `any` to the unannotated declarations that it does not infer a more specific type. More precisely, Typed Lua does not infer more specific types to the input parameters of function declarations and to the return type of recursive functions. The Typed Lua compiler cannot infer them because it performs type checking in a single step that simulates the program execution. We could have split type checking into two steps to try solving this limitation, but it would have implications in the mechanisms that Typed Lua uses to handle the discrimination of union types and the refinement of table types. We will discuss these features in Section **??** and Section **??**, respectively.

We use the following example to illustrate type annotations in the declaration of a recursive function:

```
local function factorial (n:integer):integer
  if n == 0 then
    return 1
  else
    return n * factorial(n - 1)
  end
end
```

This example compiles without any warnings because we annotated the return type of `factorial`. Local type inference cannot use the type of the returned expression when it includes a function call to the function that is being type checked, as its return type is still unknown to the type checker. For this reason, we need to annotate the return type of recursive local functions to inform the type checker what type it should use while type checking their body.

We use the following example to illustrate the omission of type annotations in the input parameters of a local function, and also to show that Typed Lua allows programmers to combine statically typed code with dynamically typed code:

```
local function absolute (n:integer):integer
  if n < 0 then
    return -n
  else
    return n
  end
end

local function distance (x, y)
  return absolute(x - y)
end
```

The function `distance` receives two parameters of type `any` and returns a value of type `integer`. The compiler assigns the dynamic type `any` to the input parameters of `distance` because they do not have type annotations and the compiler does not use global type inference, as we mentioned previously. Even though we did not annotate the return type of `distance`, the compiler is able to infer its return type because it is local and not recursive.

In this example, Typed Lua cannot guarantee that `distance` is never going to call `absolute` with a parameter that is not an integer, because in the semantics of Lua the minus operator can result in a value that is not an integer number. In fact, we can overload the minus operator to return a value that is not even a number. However, we can call `absolute` inside `distance` because the subtraction expression `x - y` has type `any`, and it is consistent with type `integer`. Still, the dynamic type `any` may be hiding a value of a type that is not an integer, making the dynamically typed code break the guarantees provided by the statically typed code. This is a typical example where run-time checks would ensure safety between the interaction of dynamically typed and statically typed code.

Even though Typed Lua can type check recursive functions when we annotate their return type, it has some limitations for type checking mutually recursive functions, even if we annotate their return type. We will use the following example for discussing this limitation and also for presenting an alternative solution:

```
local even, odd

function even (n:integer):boolean
  if n == 0 then return true
  elseif n > 0 then return odd(n - 1)
  else return odd(n + 1)
  end
end

function odd (n:integer):boolean
  if n == 0 then return false
  elseif n > 0 then return even(n - 1)
  else return even(n + 1)
  end
end
```

This example shows an attempt of annotating a common idiom that Lua programmers use for defining mutually recursive functions, but it generates compile-time warnings. The problem is related to the fact that Lua does not split a program into declarations and statements, and also to the fact that Typed Lua performs type checking in a single step. This means that forwarding the declaration of a local variable assigns the type `any` to it, making the compiler warn the programmers about function calls to this variable, as they can assign any value to a variable that has the dynamic type. In this example, the Typed Lua compiler generates warnings when we call `even` and `odd`.

One way to overcome these warnings is to predeclare these functions with an empty body before the actual declaration. Even though this is a verbose solution and the Typed Lua compiler has an option to suppress warnings related to the dynamic type `any`, it ensures that further assignments to forwarded local variables will not change their type. Next we show how we can apply this alternative solution to the previous example, making it compile without any warnings:

```
local function even (n:integer):boolean return true end
local function odd (n:integer):boolean return false end

function even (n:integer):boolean
  if n == 0 then return true
  elseif n > 0 then return odd(n - 1)
  else return odd(n + 1)
  end
end

function odd (n:integer):boolean
  if n == 0 then return false
  elseif n > 0 then return even(n - 1)
  else return even(n + 1)
  end
end
```

3.2 Functions

Lua has first-class functions, and they have some peculiarities. First, the number of arguments of a function call does not need to match the arity of the function declaration, as Lua silently drops extra arguments after evaluating them, or uses `nil` to replace missing arguments. Second, functions can return any number of values, and this number of returned values may not be statically known. Third, Lua has multiple assignment, and the semantics of argument passing is the same of the multiple assignment, that is, calling a function is like doing a multiple assignment where the left-hand side is the parameter list and the right-hand side is the argument list.

Typed Lua uses second-level types to encode function types and to preserve these peculiarities. We call them second-level because these types do not correspond to actual Lua values and we cannot use them to type variables or parameters. Second-level types represent tuple types that can appear in multiple assignment. Since the semantics of argument passing is the same of multiple assignment, second-level types also appear in function types.

Function types

```
functiontype ::= tupletype '->' rettype
   tupletype ::= '(' [typelist] ')'
    typelist ::= type {',' type} ['*']
     rettype ::= type | uniontuple ['?']
  uniontuple ::= tupletype | uniontuple '|' uniontuple
```

Figure 3.2: The concrete syntax of Typed Lua function types

Figure **??** shows that Typed Lua uses second-level types in the definition of function types. A second-level type is either the type void `()`, a tuple of first-level types optionally ending in a variadic type, or a union of these tuples. The type void `()` represents the type of the empty tuple. A variadic type `t*` is a generator for a sequence of values of the union type `t|nil`. Union of tuple types appear in the return type of function types to represent overloading on the return type. We will explain union types in more detail in the next section. We can use only one first-level type `t` in the return type because it is syntactic sugar to the tuple type `(t)`.

Typed Lua provides two operation modes: the *default* mode and the *strict* mode. In the default mode, the compiler adds a variadic tail to the type of the parameter list and to the return type if the programmer does not specify one. The default mode has this behavior to preserve the semantics of function calls in Lua, that is, it discards extra arguments and uses `nil` to replace missing arguments. In the strict mode, the compiler does not add a variadic tail to the type of the parameter list and to the return type. Instead, it preserves the type defined by the programmer to catch arity mismatch.

We will use the following function to illustrate how these two operation modes work:

```
local function sum (x:integer, y:integer):integer
  return x + y
end
```

In the default mode, `sum` has type `(integer, integer, value*) -> (integer, nil*)`. The compiler adds `value*` to the type of the parameter list to discard extra arguments. For instance, the call `sum(1, 2, 3)` compiles without any warnings because Typed Lua uses `value*` to drop the extra argument `3`.

In the strict mode, `sum` has type `(integer, integer) -> (integer)`. The compiler does not add `value*` to the type of the parameter list to catch arity mismatch. For instance, the call `sum(1, 2, 3)` compiles with a warning because we are passing an extra argument `3` to `sum`.

Even though these operation modes affect function calls, they do not affect other multiple assignments. This means that both operation modes discard extra arguments in multiple assignments and they also use `nil` to replace missing arguments in multiple assignments. We made this design decision to avoid being to restrictive, that is, we did not want to give unnecessary warnings to programmers, as discarded values do not have any implications during run-time.

As an example,

```
local x:integer, y:integer = 1, 2, 3
```

compiles without any compile-time warnings in both operation modes. This means that the left-hand side of this local declaration has the tuple type `(integer, integer, value*)`, while the expression list in the right-hand side of this local declaration has the tuple type `(1, 2, 3, nil*)` in both operation modes. The literal type `3` is consistent with `value`, and the type `nil*` generates a value of type `nil` that is also consistent with `value`.

As another example,

```
local x:integer, y:integer = sum(2, 2)
```

compiles with one compile-time warning in both of the operation modes. This means that the left-hand side of this local declaration has the tuple type `(integer, integer, value*)`, while the expression list in the right-hand side of this local declaration has the tuple type `(integer, nil*)` in both operation modes. The type `nil*` generates a value of type `nil` that is not consistent with `integer`, the type of `y`.

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. We will use the following function to illustrate the interaction between multiple returns and expression lists:

```
local function m ():(integer, string)
  return 2, "foo"
end
```

As an example,

```
local x:integer, y:integer, z:string = m(), m()
```

compiles without compile-time warnings in both operation modes. This happens because in the right-hand side of the multiple assignment, only the first value produced by the first call to `m` gets used, so the type of the right-hand side is `(integer, integer, string, nil*)`, which is consistent with the type of the left-hand side `(integer, integer, string, value*)`.

Typed Lua always includes `nil*` in the end of the type of an expression list that does not end in a variadic type. This behavior preserves the semantics of Lua on replacing missing values, and it is necessary when we omit optional parameters in a function call. We will discuss optional parameters in the next section.

We can also type variadic functions. For instance,

```
local function v (...:string):(string*)
  return ...
end
```

has type `(string*) -> (string*)` in both operation modes. The function call `v()` compiles without any compile-time warnings in the default and strict modes because `(value*)` and `()` are consistent with `(string*)`. The call `v("hello", "world")` compiles without any warnings in the default and strict modes because `("hello", "world", nil*)` and `("hello", "world")` are consistent with `(string*)`. Calling `v(...)` compiles without any warnings in both operation modes because the type of the argument list is `(string*)`, assuming that the vararg expression (`...`) has type `string`.

3.3 Unions

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

Optional values are unions of some type `t` and `nil`, and Typed Lua includes the syntactic sugar `t?` to represent them because they appear quite often. The concrete syntax `t?` is syntactic sugar for `t|nil`. Optional values can appear when a function has optional parameters and when the program reads a value from an array or a map. The following example shows a function that has an optional parameter:

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

Although the parameter `greeting` is optional, and has type `string|nil`, the concatenation does not generate a warning because we used the short-circuiting `or` operator to declare a new variable `greeting` that is guaranteed to have type `string`. In Lua, only the values `nil` and `false` represent a false condition, so programmers often use the `or` operator as a common idiom to assign a default value to an optional parameter. Typed Lua uses the following rule to type this idiom: if the left-hand side of `or` has type `t|nil` and the right-hand side has type `t` then the `or` expression has type `t`.

In fact, we do not need to declare a new variable `greeting` that shadows the optional parameter:

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

Typed Lua allows the assignment `v = v or e` to change the type of `v` from `t|nil` to `t` if it matches the following rules: the type of `e` is a subtype of `t`, and the variable `v` is local to the current function and it is not being assigned in another function. The change only affects the type of `v` in the remainder of the current scope. Any assignment to `v` restores its type back to the original type. In the case of `greeting`, the assignment changes its type from `string|nil` to `string`.

After we define the function `message`, we can call it with a missing argument in both operation modes:

```
print(message("Typed Lua"))          --> Hello Typed Lua
print(message("Typed Lua", "Hi "))   --> Hi Typed Lua
```

The type of the function `message` is `(string, string|nil, value*) -> (string, nil*)` in the default mode, and `(string, string|nil) -> (string)` in the strict mode. The function call `message("Typed Lua")` compiles without any compile-time warnings in both modes, because the argument list `("Typed Lua")` has type `("Typed Lua", nil*)` that is consistent with the input type `(string, string|nil, value*)` in the default mode, and that is also consistent with the input type `(string, string|nil)` in the strict mode. The compiler includes `nil*` in the tail of an argument list in both modes, as it is necessary to replace any optional parameters that may appear in a function declaration. In Section **??** we will formalize this behavior.

Lua programmers often overload the input parameters of functions, and use the `type` function to inspect the tag of the input parameters to take different actions depending on what those tags are. The simplest case overloads on just a single parameter:

```
local function overload (s1:string, s2:string|integer)
  if type(s2) == "string" then
    return s1 .. s2
  else
    -- string.rep : (string, integer, string?) -> (string)
    return string.rep(s1, s2)
  end
end
```

Typed Lua has a small set of `type` predicates that allows programmers to constrain the type of a local variable inside a condition. This example uses the predicate `type(v) == "string"` that constrains the type of `v` from `string|t` to `string` when the predicate is true and `t` otherwise. This is a simplified form of *flow typing* [?, ?]. As with `or`, the variable must be local to the function and it is not being assigned 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. For instance, the predicates can discriminate a union that combines a table type with a base type, or a table type with a function type, or two base types, but they cannot discriminate a union that combines two different table types, or two different function types.

Lua programmers also overload the return type of functions, usually for signaling the occurrence of errors. In this idiom, a function returns its normal set of return values when it successfully finished its execution, and it returns `nil` plus an error message or other data that describes the error when something failed during its execution. Next, we show an example:

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

Typed Lua also includes a syntactic sugar for this idiom: we can annotate the return type of `idiv` with `(integer, integer)?` to denote the same union. The parentheses are always necessary in this case, because `t?` is syntactic sugar for `t|nil`, while `(t)?` is syntactic sugar for `(t)|(nil, string)`.

A typical client of this function would use it as follows:

```
local q, r = idiv(a, b)
if q then
  print(a == b * q + r)
else
  print("ERROR: " .. r)
end
```

When Typed Lua finds a union of tuples in the right-hand side of a declaration, it assigns *projection types* to the variables that appear unannotated in the left-hand side of the declaration. Projection types do not appear in type annotations, but Typed Lua uses them to project unions of tuple types into unions of first-level types that have a dependency relation. We will discuss projection types in more detail in Section **??**.

So far, Typed Lua replaces projection types with the union of the corresponding component of each tuple, when it infers that a local variable has a projection type. In our example, the variables `q` and `r` get projection types that map to the first and second components of the union of tuple types `(integer, integer, nil*)|(nil, string, nil*)`. This means that variables `q` and `r` have types `integer|nil` and `integer|string`, respectively.

If a variable with a projection type appears in a `type` predicate, it discriminates all tuples in the projected union. In our example, the projected union has type `(integer, integer, nil*)|(nil, string, nil*)` outside of the `if` statement, but `(integer, integer, nil*)` inside the `then` block, and `(nil, string, nil*)` inside the `else` block. Thus, variable `q` has type `integer|nil` and variable `r` has type `integer|string` outside of the `if` statement; but variable `q` has type `integer` and variable `r` also has type `integer` inside the `then` block; and variable `q` has type `nil` while variable `r` has type `string` inside the `else` block.

We could have used `math.type(q) == "integer"` or even `math.type(r) == "integer"` as the predicate of our example, as both predicates would produce the same result. However, the form that appears in our example is much more succinct and idiomatic. Note that the type `integer` is restricted to Lua 5.3, as we use the function `math.type` to decide whether a number is integer.

Typed Lua does not allow assignments to variables that hold a projection type. Unrestricted assignment to these variables would be unsound, as it could break the dependency relation among the types in each tuple that is part of the union.

The overloading mechanism of Typed Lua has a limitation: the return type cannot depend on the input types. Next, we show an example:

```
local function limitation (x:number|string)
  if type(x) == "number" then
    return x + x
  else
    return x .. x
  end
end
```

This example means that we cannot write a function that is guaranteed to return a number when we pass a number and guaranteed to return a string when we pass a string. Even though we could have used an intersection type `(number) -> (number) & (string) -> (string)` to express the type of this function, intersection types require more sophisticated flow typing to check whether a function has this type, and we still need to work on this problem.

3.4 Tables

Tables are the only mechanism that Lua has to build data structures; they are associative arrays where any value (except `nil`) can be used as a key. Programmers can use tables to represent tuples, arrays (dense or sparse), records, graphs, modules, objects, etc. Lua has syntactic sugar for indexing tables as records: `t.k` is syntactic sugar for `t["k"]`. In this section, we show how Typed Lua types tables that encode maps, arrays, and records.

Figure **??** shows that the concrete syntax of Typed Lua table types is restricted to either the type of the empty table, maps, arrays, or records with an optional array part. We made this design choice due to the results that we obtained about the usage of the table constructor, which we discussed in Section **??**. More precisely, those results indicated that programmers seldom define a table constructor that is neither an empty table nor a table that contains only literal keys. This means that our design can type check most of the usages of the table constructor. Later in this chapter we will show that we use the syntax of records to type modules and objects. We will also show that we need the `const` modifier while typing object-oriented code.

Table types

$$\begin{aligned}
\textit{tabletype} &::= \text{‘}\{\text{’}\ [\textit{tabletypebody}]\ \text{‘}\}\text{’} \\
\textit{tabletypebody} &::= \textit{maptype} \mid \textit{recordtype} \\
\textit{maptype} &::= [\textit{keytype}\ \text{‘}\texttt{:}\text{’}]\ \textit{type} \\
\textit{keytype} &::= \textit{basetype} \mid \textbf{value} \\
\textit{recordtype} &::= \textit{recordfield}\ \{\text{‘}\texttt{,}\text{’}\ \textit{recordfield}\}\ [\text{‘}\texttt{,}\text{’}\ \textit{type}] \\
\textit{recordfield} &::= [\textbf{const}]\ \textit{literaltype}\ \text{‘}\texttt{:}\text{’}\ \textit{type}
\end{aligned}$$

Figure 3.3: The concrete syntax of Typed Lua table types

The table type {`k:t`} represents the type of a map from values of type `k` to values of type `t|nil`. Table types that represent maps always include the type `nil`, because Lua returns `nil` when we use a non-existing key to index a table. The types of the keys are restricted to either base types or the type `value` due to the statistics of the table constructor, which we discussed in Section **??**. More precisely, those results indicated that programmers seldom use non-literal keys in the definition of a table constructor. For typing maps, base types can type check most of the cases where programmers use a table constructor to initialize a map, while the type `value` is still an option to allow programmers to type check uncommon table constructors. This means that this restriction to key types in maps has no impact in the usability of Typed Lua. Next, we show one example of table type to type a map from strings to integers:

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

The second line of this example raises a warning, because we are attempting to assign a value of type `integer|nil` to a variable that accepts only values of type `integer`. Although the field `bar` does not exist in `t`, the third line of this example does not raise a warning, because the annotated type matches the type of the values that can be stored in `t`. The last line shows that the `or` idiom is also useful to give a default value to a missing table field.

The table type {`t`} represents the type of an array that maps values of type `integer` to values of type `t|nil`. In other words, Typed Lua handles arrays as syntactic sugar to the table type {`integer:t`}. Next, we show one example of table type to type an array of strings:

```
local days:{string} = { "Sunday", "Monday", "Tuesday",
                        "Wednesday", "Thursday", "Friday",
                        "Saturday" }
local i = 5
local x = days[1]                 --> x gets "Sunday"
local y = days[8]                 --> y gets nil
local z = days[i]                 --> z gets "Thursday"
```

In this example, the type of `i` is `integer`, while the type of `x`, `y`, and `z` is `string|nil`. An inconvenient aspect of making the types of maps and arrays always include the type `nil` is to overload the programmers, as they need to use the logical `or` operator or the `if` statement to narrow the type of the elements they are accessing.

The table type $\{l_1{:}t_1, \ldots, l_n{:}t_n\}$ represents the type of a record that maps literal types $l_i, \ldots, l_n$ to values of types $t_i, \ldots, t_n$. Most programming languages treat records as maps from names to types, but we chose to use literal types instead of names due to the statistics results that we obtained about the usage of the table constructor, which we discussed in Section **??**. This means that we can use the syntax of records to type a list that has a fixed number of elements, like the variable `days` from the previous example.

When we know that a list has fixed elements, we can leave the variable declaration unannotated and let local type inference assign a more specific table type to the variable. If we remove the annotation in the previous example, the compiler uses the syntax of records to infer the following table type to `days`:

```
{ 1:string, 2:string, 3:string, 4:string,
  5:string, 6:string, 7:string }
```

When we use the syntax of records to type an array, the compiler raises a warning when we try to access an index that is out of bounds. In the previous example, the expressions `days[8]` and `days[i]` would raise warnings if we had used the records syntax, as both the literal type `8` and the base type `integer` would not map to any value.

We can also use the syntax of records to type heterogeneous tuples:

```
local album:{1:string, 2:integer, 3:string} =
  { "Transformer", 1972, "Lou Reed" }
```

Next, we show one example of table type to type a record that maps names to strings:

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

In these two previous examples, local type inference would infer the very same table types that we used to annotate the variables `album` and `person`.

Lua programmers often build records incrementally, that is, they usually declare a local variable with an empty table, and then use assignment to add fields to this table:

```
local person = {}
person.firstname = "Lou"
person.lastname = "Reed"
print("bye bye " .. person.firstname)  --> bye bye Lou
```

In this example, we want to refine the type of `person` as we build the table: starting with {}, and then refining to {`"firstname":string`}, and finally reaching {`"firstname":string, "lastname":string`}. This type change is trickier than the one that we introduced for narrowing union types, as we are not just allowing the programmer to change the type of the variable `person`, but we are actually allowing the programmer to change the type of the value that `person` points to.

Typed Lua tags a variable that holds a table type as either *unique*, *open*, *fixed*, or *closed*. If a variable gets its type from a table constructor then it is *unique*, otherwise it is *closed*. Any alias to *unique* variables makes them *open*, as we use the tag *open* to keep track of *unique* variables that are aliased. In the previous example, `person` has an *unique* type because it has no alias. In Section **??** we will show that we use the tag *fixed* to describe the type of classes. Typed Lua also has different subtyping rules that handle these different tags, which we explain in Section **??**.

Typed Lua uses three rules to decide whether it is sound to allow field assignment to change the type of *unique* and *open* variables: the variable must be local to the current block, the new type must only add new fields, and the variable cannot have been assigned in another function.

Before discussing each one of these rules, it is worth discussing why Typed Lua needs *unique* and *open* table types. Even though both *unique* and *open* table types allow the refinement of table types, we need both tags to increase usability and to avoid some unsafe behaviors.

If we did not have *unique* table types and *open* table types had the same behavior that they have now, we would not be able to type check some safe constructions, like in the following example:

```
local t:{"x":integer, "y":integer?} = { x = 1, y = 2 }
```

Typed Lua type checks this example because the table constructor has an *unique* table type {"x":1, "y":2} that is consistent with the type in the annotation. However, if the table constructor had an *open* table type, it would not type check because it does not allow us to convert a table field from `integer` to `integer?`. We need this kind of behavior to avoid unsafe constructions while aliasing *unique* table types, like in the following example:

```
local t1 = { x = 1 }
local t2:{"x":integer} = t1
local t3:{"x":integer?} = t1      --> compile-time warning
t3.x = nil
```

Typed Lua generates a compile-time warning in this example because aliasing `t1` changes its type from *unique* to *open*. First, Typed Lua assigns an *unique* table type to `t1`, but then it changes its type to *open* before aliasing `t1` to `t2`. Even though it is safe aliasing `t1` to `t2`, it is not safe aliasing `t1` to `t3`, as it allows removing the value that is stored in the field `x` from `t1` through an assignment to field `x` from `t3`. Typed Lua uses *open* table types to prevent this kind of unsafe behavior, as it would be allowed by *unique* table types due to their loose subtyping rules.

Back to the rules that Typed Lua uses to allow the refinement of table types, the following example shows that it is not sound to allow the type of the fields to change:

```
local person = { firstname = "Lewis", lastname = "Reed" }
person.middlename = "Allan"
person.firstname = 1942           --> compile-time warning
person.lastname = 2013            --> compile-time warning
print("bye bye " .. person.firstname)
```

In this example, both third and fourth lines are unsound because they are trying to change the type of fields that already exist.

It is also not sound to allow changing the type of an alias:

```
local person = {}
local bogus = person
person.firstname = "Lou"
person.lastname = "Reed"
bogus.firstname = true            --> compile-time warning
print("bye bye " .. person.firstname)
```

In this example, the fifth line is unsound because it allows the programmer to change the type of the value that is stored in `person.firstname`, and makes the last line break the guarantees provided by the statically typed code. In the first line, we assign an empty table to `person`. In the second line, we assign the type of `person` to `bogus`. In the third and fourth lines we change the type of `person`. In the fifth line we try to change the type of `bogus` from {} to {`"firstname":boolean`}, but if we allow this type change we also allow changing the value that is stored in `person.firstname`, regardless of its type. Taking the changes individually looks fine, but the truth is that aliasing makes one of them unsound.

We can create aliases to *open* variables, but they are always *closed*. Any mutation in the original reference and in the aliased reference is not a problem, as the type of the original reference can only add new fields. For mutable fields this means that the type of a field cannot change after it is added to the type of a table. In our previous example, the second line changes the type of `person` from *unique* to *open* and makes the type of `bogus` *closed*.

The location of the change also matters, as the next example shows:

```
local person = { lastname = "Reed" }
local function spoil ()
  person.firstname = nil          --> compile-time warning
end
person.firstname = "Lou"
spoil()
print("bye bye " .. person.firstname)
```

In this example, the third line is unsound because this line allows the programmer to change the type of `person` outside of the scope that `person` was declared. The call to `spoil` erases the field `firstname` from `person`, and makes the last line break the guarantees provided by the statically typed code.

In Section **??** we present the formalization of the rules that type check the refinement of table types.

3.5 Type aliases and interfaces

Typed Lua includes *type aliases* for allowing programmers to define their own data types. Figure **??** shows the concrete syntax of the `typealias` construct.

As an example, the following declaration defines the type `Person` as an alias to the table type {`"firstname":string, "lastname":string`} in the remainder of the current scope:

```
local typealias Person = { "firstname":string,
                           "lastname":string }
```

Type aliases

$$\mathit{typealias} ::= [\textbf{local}]\ \textbf{typealias}\ \mathit{Name}\ \text{‘}\texttt{=}\text{’}\ \mathit{type}$$

Figure 3.4: The concrete syntax of Typed Lua type aliases

Typed Lua also includes *interfaces* as syntactic sugar to aliases for table types that specify records, as writing table types can be unwieldy when records get bigger and the types of record fields get more complicated. Figure **??** shows the concrete syntax of the `interface` construct. In Section **??** we will show that we can also use *interface* to document the type of objects, and that *methodtype* is syntactic sugar to express the type of methods.

Interfaces

$$\begin{aligned}
\mathit{interface} &::= [\textbf{local}]\ \textbf{interface}\ \mathit{typedec} \\
\mathit{typedec} &::= \mathit{Name}\ \{\mathit{decitem}\}\ \textbf{end} \\
\mathit{decitem} &::= \mathit{idlist}\ \text{‘}\texttt{:}\text{’}\ \mathit{idtype} \\
\mathit{idtype} &::= \mathit{type} \mid \mathit{methodtype} \\
\mathit{idlist} &::= \mathit{id}\ \{\text{‘}\texttt{,}\text{’}\ \mathit{id}\} \\
\mathit{id} &::= [\textbf{const}]\ \mathit{Name}
\end{aligned}$$

Figure 3.5: The concrete syntax of Typed Lua interfaces

As an example, we can use the following `interface` to define the type `Person` from the previous example:

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

After we define the type `Person`, we can use it in type annotations:

```
local function byebye (person:Person):string
  return "Goodbye " .. 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(byebye(user1))             --> Goodbye Lewis Reed
print(byebye(user2))             --> compile-time warning
print(byebye(user3))             --> Goodbye Lou Reed
print(byebye(user4))             --> compile-time warning
```

This example shows that our optional type system is structural rather than nominal, that is, it checks the structure of types instead of their names, and it uses subtyping and consistent-subtyping for checking types.

Even though the interface declaration may look redundant due to the type alias declaration, it has a more convenient syntax for declaring table types that express records. For this reason, it is worth having two different constructs, one specifically for records and another for more general types.

The `interface` and `typealias` constructs also allow the declaration of recursive types. For instance, the following interface defines a type for singly-linked lists of integers:

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

Now we can use `Element` to annotate a function that inserts an element at the beginning of a list:

```
local function insert (e:Element?, v:integer):Element
  return { info = v, next = e }
end
```

We need an explicit type declaration in the return type because the Typed Lua compiler cannot infer recursive types, though it has subtyping rules to check that the inferred return type matches the annotation.

Now we can write a program that declares a list, inserts some elements in it, and then traverses the list printing each element:

```
local l:Element?

l = insert(l, 4)
l = insert(l, 3)
l = insert(l, 2)
l = insert(l, 1)

while l do
  print(l.info)
  l = l.next
end
```

Note that the type of `l` is `Element` inside the `while` body, and the assignment `l = l.next` restores the type of `l` to `Element|nil`. This means that this example compiles without any warnings, because the `while` statement also uses the small set of `type` predicates that we mentioned in Section ??.

As another example of recursive type, the following type alias defines a type for the abstract syntax tree of a language of simple arithmetic expressions:

```
local typealias Exp = {"tag":"Number", 1:number}
                    | {"tag":"Add", 1:Exp, 2:Exp}
                    | {"tag":"Sub", 1:Exp, 2:Exp}
                    | {"tag":"Mul", 1:Exp, 2:Exp}
                    | {"tag":"Div", 1:Exp, 2:Exp}
```

The type `Exp` is a recursive type that resembles the algebraic data types from functional programming. The small set of `type` predicates that we mentioned in Section ?? also includes a specific rule that is not based on tags. This specific rule lets programmers discriminate unions of table types and resembles the pattern matching from functional programming. We can use this feature to write an evaluation function for our simple language:

```
local function eval (e:Exp):number
  if e.tag == "Number" then return e[1]
  elseif e.tag == "Add" then return eval(e[1]) + eval(e[2])
  elseif e.tag == "Sub" then return eval(e[1]) - eval(e[2])
  elseif e.tag == "Mul" then return eval(e[1]) * eval(e[2])
  elseif e.tag == "Div" then return eval(e[1]) / eval(e[2])
  end
end
```

When Typed Lua finds the predicate `v[`e_1`] == `e_2, it checks whether the type of the variable `v` is an union of table types, whether the type of e_1 is a literal type $\mathtt{l}_1$, and whether the type of $\mathtt{e}_2$ is a literal type $\mathtt{l}_2$. If that is the case, it constrains the type of `v` inside the `if` to the table type that has a key of type $\mathtt{l}_1$ which maps to a value of type $\mathtt{l}_2$. In our example, the expression `e.tag == "Add"` makes the Typed Lua compiler constrain the type of `e` to `{"tag":"Add", 1:Exp, 2:Exp}`. If we did not add this special predicate, we would not be able to traverse an AST, as any attempt to index an union raises a compile-time warning.

3.6 Modules

Lua does not set policies on how programmers should define modules, but it provides mechanisms for organizing a program in modules. To load a module, Lua first checks whether the module is already loaded. When the module is not loaded, Lua executes its source file, and the value returned is the module. Although programmers can use functions for defining modules, our survey from Section ?? confirms that the convention among Lua programmers is to use tables for defining modules. In this convention, the fields of the table are the functions and other values that the module exports.

An idiomatic way for defining modules in Lua is to declare only locals and return a table constructor at the end of the source file. The returned constructor includes the members that the module should export. The following example illustrates this case in Typed Lua:

```
local RADIANS_PER_DEGREE = 3.14159 / 180.0
local function deg (x:number):number
  return x / RADIANS_PER_DEGREE
end
local function rad (x:number):number
  return x * RADIANS_PER_DEGREE
end
local function pow (x:number, y:number):number
  return x ^ y
end
return {
  deg = deg,
  rad = rad,
  pow = pow,
}
```

In this example, Typed Lua uses the type of the table constructor to allow the programmer to build the type of the module. The local variable

`RADIANS_PER_DEGREE` is private because we did not include it in the list of exported members. The `return` at the end of the source file gives the type of the module:

```
{ "deg":(number) -> (number),
  "rad":(number) -> (number),
  "pow":(number, number) -> (number) }
```

Typed Lua always fixes *unique* and *open* table types that appear in a top-level return statement. This means that modules have *fixed* table types in Typed Lua. This behavior ensures that classes always have *fixed* table types. In the next section we will show that we build classes in a similar way that we use to build modules, and we will also show that classes should have *fixed* table types to allow safe single inheritance.

Another idiomatic way for defining modules in Lua is to declare an empty table at the begin of the source file, populate this table with the members that the module should export, and return this table at the end of the source file. The following example illustrates this case in Typed Lua:

```
local mymath = {}
local RADIANS_PER_DEGREE = 3.14159 / 180.0
mymath.deg = function (x:number):number
  return x / RADIANS_PER_DEGREE
end
mymath.rad = function (x:number):number
  return x * RADIANS_PER_DEGREE
end
mymath.pow = function (x:number, y:number):number
  return x ^ y
end
return mymath
```

In this example, Typed Lua uses the refinement of table types to incrementally build the type of the module `mymath`. The variable `RADIANS_PER_DEGREE` is private because we declared it as local to the module. The `return` at the end of the source file gives the type of the module, which is the same type of the variable `mymath`. This module has the same type of the module from the previous example.

After we define the module `mymath`, regardless of the adopted style, users can use it in the standard way, but with the static type checking provided by Typed Lua. Next we show an example:

```
local mymath = require "mymath"
print(mymath.pow(2, 3))          --> 8
print(mymath.pow(2, "foo"))      --> compile-time warning
```

In Typed Lua, `require` is a primitive that statically type checks a given module to infer its type. This means that the type of its input parameter must be a literal string, as Typed Lua uses this literal string to find the source file that implements the given module. To statically type check a module, Typed Lua follows the same rules that Lua follows to load a module. Typed Lua first checks whether the module is already statically type checked. When the module is not yet statically type checked, Typed Lua statically type checks its source file, and the type returned is the type of the module. Typed Lua raises a compile-time warning when it cannot find the source file of a given module.

After we use `require` to statically type check a module, Typed Lua can statically type check the usage of this module. In our previous example, the call to `require` assigns the type of the module `mymath` to the local `mymath`, so Typed Lua can catch misuses of the module.

The way that Typed Lua handles modules using *fixed* table types is also relevant for supporting object-oriented programming, as we discuss in the next section.

3.7 Object-Oriented Programming

Lua provides minimal support for object-oriented programming. The basic mechanism is the : syntactic sugar for method definitions and method calls. In the case of method definitions, the Lua compiler translates `function obj:method(args) end` into an operation that assigns a function to the field `method` in `obj`. This function includes a first parameter named `self` plus any other parameters. In the case of method calls, the Lua compiler translates `obj:method(args)` into an operation that evaluates `obj`, uses `method` to index `obj`, and then calls `obj.method` with the result of evaluating `obj` as the first argument, followed by the result of evaluating the argument list in the original expression.

Typed Lua uses *closed* table types along with the type `self` to represent objects. We use the following example to discuss this feature:

```
local Shape = { x = 0.0, y = 0.0 }
function Shape:move (dx:number, dy:number)
  self.x = self.x + dx
  self.y = self.y + dy
end
```

Typed Lua assigns the type `self` to the implicit parameter `self`. The type `self` represents the type of the *receiver* in method definitions and method calls. In this example, Typed Lua binds the type `self` to the *closed* table type {`"x":number, "y":number`} inside `move`. This example also uses the refinement of table types to build the type of `Shape`:

```
{ "x":number, "y":number,
  "move":(self, number, number) -> () }
```

While `:` is syntactic sugar in Lua, Typed Lua uses it to check method calls, binding any occurrence of the type `self` in the type of the method to the receiver. Indexing a method and not immediately calling it with the correct receiver is a compile-time warning:

```
Shape.move(Shape, 10, 10) --> Shape.x = 10 and Shape.y = 10
Shape:move(5, 20)         --> Shape.x = 5 and Shape.y = 20
local p = Shape.move      --> compile-time warning
```

Lua has a mechanism for self-like (or JavaScript-like) delegation of missing table fields. After the call `setmetatable(t1, { __index = t2 })`, Lua looks up in `t2` for any missing fields of `t1`. As we mentioned in Section **??**, Lua programmers often use this mechanism to simulate classes. We will use the following example to discuss this feature:

```
local Shape = { x = 0.0, y = 0.0 }

const function Shape:new (x:number, y:number):self
  local s:self = 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

return Shape
```

In Typed Lua, `setmetatable` is a strict primitive that obeys three typing rules. The reason for being so strict with `setmetatable` is because it is the only mechanism that Typed Lua has to create classes as prototype objects.

The first `setmetatable` rule appears in our previous example. In a `setmetatable` expression `setmetatable({},{__index = id})`, if `id` has type `self` then the expression also has type `self`. We will explain the other two rules while we explain how Typed Lua type checks single inheritance.

In our example, we are using the refinement of table types to build the type of the variable `Shape`, as it should work as our class. This means that the local `Shape` has type `{"x":number, "y":number}` inside the definition of `new`. This also makes Typed Lua bind the type of the local `Shape` to the type `self` inside the definition of `new`, allowing us to access fields `x` and `y`. After the definition of `new`, the local `Shape` has type `{"x":number, "y":number, const "new":(self, number, number) -> (self)}`, which is the type that Typed Lua binds to the type `self` inside the definition of `move`. We use the `const` annotation in the type of the methods because it is necessary for covariance among object types to work. Finally, the top-level `return` statement fixes the type of `Shape`, because *fixed* table types do not allow width subtyping, and this behavior allows Typed Lua to use the refinement of table types to type check single inheritance, as we will see in this section.

A return statement that appears in the top-level also exports an interface which we can use in type annotations. For instance, our previous example exports the following interface:

```
interface Shape
  x, y:number
  const new:(number, number) => (self)
  const move:(number, number) => ()
end
```

We use a double arrow instead of a single arrow because it is syntactic sugar for defining the type of methods, as it includes an implicit first input type `self`. The double arrow can appear only inside the declaration of interfaces and userdata, which we will introduce in the next section.

The style of classes definition from the previous example allows us to use `require` for creating prototype objects that work as classes, along with an alias to the object type. This allows us to use the exported alias in type annotations, as we show in the following example:

```
local Shape = require "shape"
local shape1 = Shape:new(0, 5)
local shape2:Shape = Shape:new(10, 10)
```

Note that `Shape` has a *fixed* table type, because it describes the type of the class `Shape`, but `shape1` and `shape2` have *closed* table types, because they describe the type of objects from the class Shape. Now, we will see that we need *fixed* table types to allow setmetatable to simulate single inheritance. We use the following example to discuss single inheritance in Typed Lua:

```
local Shape = require "shape"

local Circle = setmetatable({}, { __index = Shape })

Circle.radius = 0.0

const function Circle:new (x:number,
                           y:number,
                           radius:value):self
  local c:self = setmetatable(Shape:new(x, y),
                              { __index = self })
  c.radius = tonumber(radius) or 0.0
  return c
end

const function Circle:area ():number
  return math.pi * self.radius * self.radius
end

return Circle
```

In this example, we use the other two setmetatable rules. The second rule appears to trigger the refinement of table types, as we need this rule to add new methods and also to override existing methods in Circle. The third rule appears to redefine the constructor new.

In a setmetatable expression setmetatable({},{__index = id}), if id has a *fixed* table type then it is safe to produce an equivalent *open* table type, as *fixed* table types do not allow hiding fields. In our example, it is safe to use the type of Shape to initialize Circle, opening the type of Circle to allow the refinement of table types to build the type of the class Circle.

In a setmetatable expression setmetatable(e,{__index = id}), if the expression e has a *closed* table type t, id has type self, and the type bound to self is a subtype of t, then the expression has type self. Note how we can use this rule to call the constructor of Shape 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 the input parameter `radius` has to be quite permissive. Also note that this is the only form of refinement that allows changing the type of a table field, if the new type is a subtype of the previous type.

Like in the example that we introduced the class `Shape`, in the example that we introduced the class `Circle`, the top-level return statement exports an interface which we can use in type annotations:

```
interface Circle
  x, y, radius:number
  const new:(number, number, value) => (self)
  const move:(number, number) => ()
  const area:() => (number)
end
```

Even though the class `Circle` is not a subtype of the class `Shape`, because *fixed* table types do not have width subtyping, the objects that have the exported type `Circle` are subtypes of the objects that have the exported type `Shape`, because *closed* table types have width subtyping. We discuss *fixed* and *closed* table types in more detail in the next chapter.

We can use both exported aliases in type annotations, as we show in the following example:

```
local Circle = require "circle"
local circle1 = Circle:new(0, 5, 10)
local circle2:Circle = Circle:new(10, 10, 10)
local circle3:Shape = circle1
local circle4:Shape = circle2
print(circle2:area())             --> 314.15926535898
print(circle3:area())             --> compile-time warning
```

In all examples, if we erase all type and `const` annotations, they become valid Lua code, which run with the same semantics as Typed Lua code.

The current version of Typed Lua has some limitations regarding the use of `setmetatable` that are on plans for future work. One limitation is that Typed Lua does not have polymorphism, so programmers cannot hide the calls to `setmetatable` behind nicer abstractions, as some Lua libraries do. Another limitation is that Typed Lua does not support operator overloading, so programmers cannot use `setmetatable` to change the behavior of predefined operations, as some Lua libraries do.

Our classes system also does not support multiple inheritance and does not offer privacy rules, but these limitations are not on plans for future work anyway.

3.8 Description files

Typed Lua allows programmers to create description files for exporting statically typed interfaces to dynamically typed modules. This means that programmers can have some of the benefits of static types even without converting existing Lua modules to Typed Lua, as a dynamically typed module can export a statically typed interface, and statically typed users of the module have their use of the module checked by the compiler.

Furthermore, Typed Lua also allows programmers to create description files for exporting statically typed interfaces for Lua modules that are written in C.

The complete syntax of description files

$$
\begin{array}{rcl}
\mathit{description} & ::= & \mathit{desclist} \\
\mathit{desclist} & ::= & \mathit{descitem}\ \{\mathit{descitem}\} \\
\mathit{descitem} & ::= & \mathit{typedid} \mid \mathit{interface} \mid \mathit{userdata} \mid \mathit{typealias} \\
\mathit{typedid} & ::= & [\mathbf{const}]\ \mathit{id}\ \text{‘}\texttt{:}\text{’}\ \mathit{type} \\
\mathit{interface} & ::= & \mathbf{interface}\ \mathit{typedec} \\
\mathit{userdata} & ::= & \mathbf{userdata}\ \mathit{typedec} \\
\mathit{typedec} & ::= & \mathit{Name}\ \{\mathit{decitem}\}\ \mathbf{end} \\
\mathit{decitem} & ::= & \mathit{idlist}\ \text{‘}\texttt{:}\text{’}\ \mathit{idtype} \\
\mathit{idtype} & ::= & \mathit{type} \mid \mathit{methodtype} \\
\mathit{idlist} & ::= & \mathit{id}\ \{\text{‘}\texttt{,}\text{’}\ \mathit{id}\} \\
\mathit{id} & ::= & [\mathbf{const}]\ \mathit{Name} \\
\mathit{typealias} & ::= & \mathbf{typealias}\ \mathit{Name}\ \text{‘}\texttt{=}\text{’}\ \mathit{type}
\end{array}
$$

Figure 3.6: The concrete syntax of Typed Lua description files

Figure **??** shows the complete syntax of Typed Lua description files in extended BNF. A Typed Lua description file defines the table type that represents a certain module and the type names that are exported along with this table type. We can export a type name through the declaration of either an interface, an userdata, or a type alias. As we mentioned in Section **??**, a type alias declaration creates an alias to a more general type, while an interface declaration creates an alias to a table type that represents a record. An userdata declaration is similar to an interface declaration, but it also includes its name as the *brand* of the table type. The Typed Lua compiler uses this brand to combine structural with nominal type checking, so two userdata that export exactly the same members, but do not have the same name, are not subtype of each other, because they do not share the same brand.

The following example shows the description file for `lmd5` [?], a MD5 digest library for Lua that is written in C:

```
userdata md5_context
  __tostring : (self) -> (string)
  clone : (self) -> (self)
  digest : (self, value) -> (string)
  new : (self) -> (self)
  reset : (self) -> (self)
  update : (self, string*) -> (self)
  version : string
end

__tostring : (md5_context) -> (string)
clone : (md5_context) -> (md5_context)
digest : (md5_context|string, value) -> (string)
new : () -> (md5_context)
reset : (md5_context) -> (md5_context)
update : (md5_context, string*) -> (md5_context)
version : string
```

This description file exports the type `md5_context` through an `userdata` declaration and the table type that represents the type of the module. Now we can use the Typed Lua compiler to check for type errors in our use of the `lmd5` library:

```
local m = require "md5"
local x = m.new()
local y = x:clone()
local z = m.clone("foo")          --> compile-time warning
print(x:digest() == m.digest(y))  --> true
```

The Typed Lua compiler searches for a description file when it cannot find the respective Typed Lua file that is the argument of `require`. In this example, the call to `require` assigns to the local `m` the table type that the description file of the `lmd5` library exports. Thus, the compiler raises a compile-time warning in the fourth line, as the function `clone` expects a value of type `md5_context` instead of a value of type `string`.

The description files are the mechanism that we used to include the typing of the Lua standard library inside Typed Lua. In Chapter **??** we will discuss the issues that we found while typing the Lua standard library and other case studies, which include the `lmd5` library.

4
The type system

In the previous chapter we presented an informal overview of Typed Lua. We showed that programmers can use Typed Lua to combine static and dynamic typing in the same code, and it allows them to incrementally migrate from dynamic to static typing. This is a benefit to programmers that use dynamically typed languages to build large applications, as static types detect many bugs during the development phase, and also provide better documentation.

In this chapter we present the abstract syntax of Typed Lua types, the subtyping rules, and the most interesting typing rules. Besides its practical contributions, Typed Lua also has some interesting contributions to the field of optional type systems for scripting languages. They are novel type system features that let Typed Lua cover several Lua idioms and features, such as refinement of tables, multiple assignment, and multiple return values.

4.1 Types

Figure **??** presents the abstract syntax of Typed Lua types. Typed Lua splits types into two categories: *first-level types* and *second-level types*. First-level types represent first-class Lua values and second-level types represent tuples of values that appear in assignments and function applications. First-level types include literal types, base types, the type **nil**, the top type **value**, the dynamic type **any**, the type **self**, union types, function types, table types, recursive types, filter types, and projection types. Second-level types include tuple types and unions of tuple types. Tuple types include the type **void**, variadic types, and pair types. Types are ordered by a subtype relationship that we introduce in Section **??**, so Lua values may belong to several distinct types.

Literal types represent the type of literal values. They can be the boolean values **false** and **true**, an integer value, a floating point value, or a string value. We will see that literal types are important in our treatment of table types as records.

Type Language

$T ::=$		FIRST-LEVEL TYPES:
	L	*literal types*
	$\mid B$	*base types*
	$\mid \mathbf{nil}$	*nil type*
	$\mid \mathbf{value}$	*top type*
	$\mid \mathbf{any}$	*dynamic type*
	$\mid \mathbf{self}$	*self type*
	$\mid T_1 \cup T_2$	*disjoint union types*
	$\mid S_1 \to S_2$	*function types*
	$\mid \{K_1{:}V_1, ..., K_n{:}V_n\}_{unique\mid open\mid fixed\mid closed}$	*table types*
	$\mid x$	*type variables*
	$\mid \mu x.T$	*recursive types*
	$\mid \phi(T_1, T_2)$	*filter types*
	$\mid \pi_i^x$	*projection types*
$L ::=$		LITERAL TYPES:
	$\mathbf{false} \mid \mathbf{true} \mid \mathit{int} \mid \mathit{float} \mid \mathit{string}$	
$B ::=$		BASE TYPES:
	$\mathbf{boolean} \mid \mathbf{integer} \mid \mathbf{number} \mid \mathbf{string}$	
$K ::=$		KEY TYPES:
	$L \mid B \mid \mathbf{value}$	
$V ::=$		VALUE TYPES:
	$T \mid \mathbf{const}\ T$	
$S ::=$		SECOND-LEVEL TYPES:
	P	*tuple types*
	$\mid S_1 \sqcup S_2$	*unions of tuple types*
$P ::=$		TUPLE TYPES:
	$\mathbf{void}$	*void type*
	$\mid T*$	*variadic types*
	$\mid T \times P$	*pair types*

Figure 4.1: The abstract syntax of Typed Lua types

Typed Lua includes four base types: **boolean**, **integer**, **number**, and **string**. The base types **boolean** and **string** represent the values that Lua tags as `boolean` and `string` during run-time. Lua 5.3 introduced two internal representations to the tag `number`: `integer` for integer numbers and `float` for real numbers. Lua does automatic promotion of `integer` values to `float` values as needed. We introduced the base type **number** to represent `float` values, and the base type **integer** to represent `integer` values. In the next section we will show that **integer** is a subtype of **number**. This allows programmers to keep using `integer` values where `float` values are expected.

The type **nil** is the type of `nil`, the value that Lua uses for undefined variables, missing parameters, and missing table keys.

The type **value** is the top type, which represents any Lua value. In Section **??** we will show that this type, along with variadic types, helps the type system to drop extra values on assignments and function calls, thus preserving the semantics of Lua in these cases.

Typed Lua uses the type **self** to represent the *receiver* in object-oriented method definitions and method calls. As we mentioned in Section **??**, we need the type **self** to prevent programs from indexing a method without calling it with the correct receiver.

Union types $T_1 \cup T_2$ represent data types that can hold a value of two different types.

Function types have the form $S_1 \rightarrow S_2$ and represent Lua functions, where S is a second-level type.

Second-level types are either tuple types or unions of tuple types. Tuple types are tuples of first-level types that can end with either an empty tuple or with a variadic type. Typed Lua needs second-level types because tuples are not first-class values in Lua, only appearing on argument passing, multiple returns, and multiple assignments. The type **void** is the type of an empty tuple. A variadic type $T*$ represents a sequence of values of type $T \cup \textbf{nil}$; it is the type of a vararg expression. Second-level types include unions of tuples because Lua programs usually overload the return type of functions to denote error, as we mentioned in Section **??**. For clarity, we use the symbol $\sqcup$ to represent the union between two different tuple types. Note that $\cup$ represents the union between two first-level types, while $\sqcup$ represents the union between two tuple types.

Back to first-level types, table types represent the various forms that Lua tables can take. The syntactical form of table types is $\{K_1{:}V_1, ..., K_n{:}V_n\}_{tag}$, where each K_i represents the type of a table key, and each V_i represents the type of the value that table keys of type K_i map to. Key types can only be literal types, base types, or the top type. We made this restriction to the type of the keys because the statistics that we discussed in Section **??** showed that most of the tables are records, lists, and hashes. The type **value** is an option when we need a loose table type. For instance, $\{\textbf{value} : \textbf{value}\}_{closed}$ represents the type of a table in which both indices and values can have any type. Value types can be any first-level type, and can optionally include the **const** type to denote immutable values.

We also use the tags *unique*, *open*, *fixed*, and *closed* to classify table types. The tag *unique* represents tables with no keys that do not inhabit one of the table's key types, and with no alias. In particular, the type of the table constructor has this tag. The tag *open* represents *unique* table types that have at least one alias. The tag *fixed* represents *unique* table types that we do not know how many aliases they have. In particular, the type of a class has this tag. The tag *closed* represents table types that do not provide any guarantees about keys with types not listed in the table type. In particular, in the concrete syntax, type annotations, interface declarations, and userdata declarations always describe *closed* table types. In the next sections we explain in more detail why we need different table types.

Any table type has to be *well-formed*. Informally, a table type is well-formed if key types do not overlap. In Section **??** we formalize the definition of well-formed table types. We delay the proper formalization of well-formed table types because we use consistent-subtyping in this formalization.

Recursive types have the form $\mu x.T$, where T is a first-level type that x represents. For instance, $\mu x.\{\text{“}\mathit{info}\text{”} : \mathbf{integer}, \text{“}\mathit{next}\text{”} : x \cup \mathbf{nil}\}_{closed}$ is a type for singly-linked lists of integers. In Section **??** we mentioned that we can use the following interface declaration as an alias to this type:

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

Typed Lua includes filter types as a way to discriminate the type of local variables inside conditions. Our type system uses filter types to formalize the `type` predicates that we mentioned in Section **??**. This means that `type` predicates use filter types of the form $\phi(T_1, T_2)$ to discriminate local variables that are bound to union types. In a filter type $\phi(T_1, T_2)$, T_1 is the original type and T_2 is the discriminated type.

Typed Lua includes projection types as a way to project unions of tuple types into unions of first-level types. In Section **??** we will show in more detail how our type system uses them as a mechanism for handling unions of tuple types, when they appear in the right-hand side of the declaration of local variables, as we mentioned in Section **??**. We also show how this feature allows our type system to constrain the type of a local variable that depends on the type of another local variable.

Typed Lua includes the dynamic type **any** for allowing programmers to mix static and dynamic typing.

4.2 Subtyping

Our type system uses subtyping [?, ?] to order types and consistent-subtyping [?, ?] to allow the interaction between statically and dynamically typed code. We explain the subtyping and consistent-subtyping rules throughout this section. However, we focus the discussion on the definition of subtyping because, as we mentioned in Section ??, we can combine the consistency and subtyping relations to achieve consistent-subtyping. The differences between subtyping and consistent-subtyping are the way they handle the dynamic type, and the fact that subtyping is transitive, but consistent-subtyping is not.

We present the subtyping rules as a deduction system for the subtyping relation $\Sigma \vdash T_1 <: T_2$. The variable Σ is a set of pairs of recursion variables. We need this set to record the hypotheses that we assume when checking recursive types.

The subtyping rules for literal types and base types include the rules for defining that literal types are subtypes of their respective base types, and that **integer** is a subtype of **number**:

$$\begin{array}{ccc} \text{(S-FALSE)} & \text{(S-TRUE)} & \text{(S-STRING)} \\ \Sigma \vdash \textbf{false} <: \textbf{boolean} & \Sigma \vdash \textbf{true} <: \textbf{boolean} & \Sigma \vdash \textit{string} <: \textbf{string} \end{array}$$

$$\begin{array}{ccc} \text{(S-INT1)} & \text{(S-INT2)} & \text{(S-FLOAT)} \\ \Sigma \vdash \textit{int} <: \textbf{integer} & \Sigma \vdash \textit{int} <: \textbf{number} & \Sigma \vdash \textit{float} <: \textbf{number} \end{array}$$

$$\begin{array}{c} \text{(S-INTEGER)} \\ \Sigma \vdash \textbf{integer} <: \textbf{number} \end{array}$$

Subtyping is reflexive and transitive; therefore, we could have omitted the rule S-INT2. More precisely, we could have defined a transitive rule for first-level types instead of defining specific rules for transitive cases. For instance, a transitive rule would allow us to derive that

$$\frac{\Sigma \vdash 1 <: \textbf{integer} \quad \Sigma \vdash \textbf{integer} <: \textbf{number}}{\Sigma \vdash 1 <: \textbf{number}}$$

However, we are using the subtyping rules as the template for defining the consistent-subtyping rules, and consistent-subtyping is not transitive. More precisely, we want the subtyping and consistent-subtyping rules to differ only in the way they handle the dynamic type. Thus, we define the subtyping rules using an algorithmic approach that is close to the implementation, as this approach allows us to use subtyping to easily formalize consistent-subtyping.

Our type system includes the top type **value**, so any first-level type is a subtype of **value**:

$$\text{(S-VALUE)}$$
$$\Sigma \vdash T <: \textbf{value}$$

Many programming languages include a bottom type to represent an empty value that programmers can use as a default expression, and we could have used the type **nil** for this role. However, making **nil** the bottom type would lead to several expressions that would pass the type checker, but that would fail during run-time in the presence of a `nil` value. Thus, our type system does not have a bottom type, and **nil** is a subtype only of itself and of **value**.

Another type that is only a subtype of itself and of the type **value** is the type **self**.

The subtyping rules for union types are standard:

$$\frac{\Sigma \vdash T_1 <: T \quad \Sigma \vdash T_2 <: T}{\Sigma \vdash T_1 \cup T_2 <: T}\text{(S-UNION1)} \qquad \frac{\Sigma \vdash T <: T_1}{\Sigma \vdash T <: T_1 \cup T_2}\text{(S-UNION2)} \qquad \frac{\Sigma \vdash T <: T_2}{\Sigma \vdash T <: T_1 \cup T_2}\text{(S-UNION3)}$$

The first rule shows that a union type $T_1 \cup T_2$ is a subtype of T if both T_1 and T_2 are subtypes of T; and the other rules show that a type T is a subtype of a union type $T_1 \cup T_2$ if T is a subtype of either T_1 or T_2.

The subtyping rule for function types is also standard:

$$\text{(S-FUNCTION)}$$
$$\frac{\Sigma \vdash S_3 <: S_1 \quad \Sigma \vdash S_2 <: S_4}{\Sigma \vdash S_1 \rightarrow S_2 <: S_3 \rightarrow S_4}$$

The rule S-FUNCTION shows that subtyping between function types is contravariant on the type of the parameter list and covariant on the return type. In the previous section we explained why our type system uses second-level types to represent the type of the parameter list and the return type. Now, we explain their subtyping rules.

The type **void** is a subtype of itself and of a variadic type:

$$\text{(S-VOID2)}$$
$$\Sigma \vdash \textbf{void} <: T*$$

A variadic type $T*$ represents a sequence of values of type $T \cup \textbf{nil}$, and the rule S-VOID2 handles the case where a given sequence is empty.

The subtyping rule for pair types is the standard covariant rule:

$$\text{(S-PAIR)}\quad \frac{\Sigma \vdash T_1 <: T_2 \quad \Sigma \vdash P_1 <: P_2}{\Sigma \vdash T_1 \times P_1 <: T_2 \times P_2}$$

The subtyping rules for variadic types are not so obvious. We need six different subtyping rules for variadic types to handle all the cases where they can appear.

The rule S-VARARG1 is a special rule for handling the case where we give a sequence of **nil** to the empty tuple:

$$\text{(S-VARARG1)}\quad \Sigma \vdash \mathbf{nil}* <: \mathbf{void}$$

The rule S-VARARG2 handles the case where both tuple types end with variadic types:

$$\text{(S-VARARG2)}\quad \frac{\Sigma \vdash T_1 \cup \mathbf{nil} <: T_2 \cup \mathbf{nil}}{\Sigma \vdash T_1* <: T_2*}$$

This rule shows that T_1* is a subtype of T_2* if $T_1 \cup \mathbf{nil}$ is a subtype of $T_2 \cup \mathbf{nil}$. It explicitly includes **nil** in both sides because otherwise **nil**$*$ would not be a subtype of several other variadic types. For instance, **nil**$*$ would not be a subtype of **number**$*$, as **nil** $\not<:$ **number**.

The other rules handle the cases where only one tuple type ends with a variadic type:

$$\text{(S-VARARG3)}\quad \frac{\Sigma \vdash T_1 \cup \mathbf{nil} <: T_2}{\Sigma \vdash T_1* <: T_2 \times \mathbf{void}} \qquad \text{(S-VARARG4)}\quad \frac{\Sigma \vdash T_1 <: T_2 \cup \mathbf{nil}}{\Sigma \vdash T_1 \times \mathbf{void} <: T_2*}$$

$$\text{(S-VARARG5)}\quad \frac{\Sigma \vdash T_1* <: T_2 \times \mathbf{void} \quad \Sigma \vdash T_1* <: P_2}{\Sigma \vdash T_1* <: T_2 \times P_2}$$

$$\text{(S-VARARG6)}\quad \frac{\Sigma \vdash T_1 \times \mathbf{void} <: T_2* \quad \Sigma \vdash P_1 <: T_2*}{\Sigma \vdash T_1 \times P_1 <: T_2*}$$

Note that the case where both tuple types end with the type **void** does not require any special rule. In the next section we will show that we use the subtyping rules for variadic types, along with the types **value** and **nil**, to make our type system reflect the semantics of Lua on discarding extra parameters and replacing missing parameters.

The subtyping rules for unions of tuple types are similar to the subtyping rules for unions of first-level types:

$$\text{(S-UNION4)}\ \frac{\Sigma \vdash S_1 <: S \quad \Sigma \vdash S_2 <: S}{\Sigma \vdash S_1 \sqcup S_2 <: S} \qquad \text{(S-UNION5)}\ \frac{\Sigma \vdash S <: S_1}{\Sigma \vdash S <: S_1 \sqcup S_2} \qquad \text{(S-UNION6)}\ \frac{\Sigma \vdash S <: S_2}{\Sigma \vdash S <: S_1 \sqcup S_2}$$

Back to the subtyping rules between first-level types, the subtyping rule among a *fixed* or *closed* table type and another *closed* table type resembles the standard subtyping rule between records:

$$\text{(S-TABLE1)}\ \frac{\forall i \in 1..n\ \exists j \in 1..m \quad \Sigma \vdash K_j <: K'_i \quad \Sigma \vdash K'_i <: K_j \quad \Sigma \vdash V_j <:_c V'_i}{\Sigma \vdash \{K_1{:}V_1, ..., K_m{:}V_m\}_{fixed|closed} <: \{K'_1{:}V'_1, ..., K'_n{:}V'_n\}_{closed}}\ m \geq n$$

The rule S-TABLE1 allows width subtyping and introduces the auxiliary relation $<:_c$ to handle depth subtyping on the type of the values stored in the table fields. We need an auxiliary relation because the subtyping of the type of the values stored in the table fields changes according to the tags of the table types. We define the relation $<:_c$ as follows:

$$\text{(S-FIELD1)}\ \frac{\Sigma \vdash V_1 <: V_2 \quad \Sigma \vdash V_2 <: V_1}{\Sigma \vdash V_1 <:_c V_2} \qquad \text{(S-FIELD2)}\ \frac{\Sigma \vdash V_1 <: V_2}{\Sigma \vdash \textbf{const}\ V_1 <:_c \textbf{const}\ V_2}$$

$$\text{(S-FIELD3)}\ \frac{\Sigma \vdash V_1 <: V_2}{\Sigma \vdash V_1 <:_c \textbf{const}\ V_2}$$

These rules allow depth subtyping on **const** fields. The rule S-FIELD1 defines that mutable fields are invariant, while the rule S-FIELD2 defines that immutable fields are covariant. The rule S-FIELD3 defines that it is safe to promote fields from mutable to immutable. We do not include a rule that allows promoting fields from immutable to mutable because this would be unsafe due to variance.

There is a limitation on *closed* table types that led us to introduce *open* and *unique* table types. If the table constructor had a *closed* table type, then programmers would not be able to use it to initialize a variable with a table type that describes a more general type. For instance,

```
local t:{"x":integer, "y":integer?} = { x = 1, y = 2 }
```

would not type check, as the type of the table constructor would not be a subtype of the type in the annotation. More precisely,

$$\{``x" : 1, ``y" : 2\}_{closed} \not<: \{``x" : \textbf{integer}, ``y" : \textbf{integer} \cup \textbf{nil}\}_{closed}$$

Simply promoting the type of each table value to its supertype would not overcome this limitation, as it still would give to the table constructor a *closed* table type without covariant mutable fields. Thus, programmers would not be able to use the table constructor to initialize a variable with a table type that includes an optional field. Using the previous example,

$$\{``x" : \textbf{integer}, ``y" : \textbf{integer}\}_{closed} \not<: \\ \{``x" : \textbf{integer}, ``y" : \textbf{integer} \cup \textbf{nil}\}_{closed}$$

We introduced *unique* table types to avoid this limitation, as they represent the type of tables with no keys that do not inhabit one of the table's key types, and with no alias. In particular, this is the case of the table constructor. The following subtyping rule defines the subtyping relation among *unique* table types and *closed* table types:

$$\text{(S-TABLE2)}\quad \frac{\begin{array}{c}\forall i \in 1..m\ \forall j \in 1..n\ \Sigma \vdash K_i <: K'_j \to \Sigma \vdash V_i <:_u V'_j \\ \forall j \in 1..n\ \ \nexists i \in 1..m\ \Sigma \vdash K_i <: K'_j \to \Sigma \vdash \textbf{nil} <:_o V'_j\end{array}}{\Sigma \vdash \{K_1{:}V_1, ..., K_m{:}V_m\}_{unique} <: \{K'_1{:}V'_1, ..., K'_n{:}V'_n\}_{closed}}$$

The rule S-TABLE2 allows width subtyping and covariant keys. It allows covariant keys because we also want to use *unique* table types as a way to join table fields that inhabit *closed* table types. For instance, we want to use the table constructor to initialize a variable with a table type that describes a hash.

The rule S-TABLE2 introduced the auxiliary relations $<:_u$ and $<:_o$. The first allows depth subtyping on all fields, while the second allows the omission of optional fields. We define them as follows:

$$\text{(S-FIELD4)}\ \frac{\Sigma \vdash V_1 <: V_2}{\Sigma \vdash V_1 <:_u V_2} \qquad \text{(S-FIELD5)}\ \frac{\Sigma \vdash V_1 <: V_2}{\Sigma \vdash \textbf{const}\ V_1 <:_u \textbf{const}\ V_2} \qquad \text{(S-FIELD6)}\ \frac{\Sigma \vdash V_1 <: V_2}{\Sigma \vdash V_1 <:_u \textbf{const}\ V_2}$$

$$\text{(S-FIELD7)}\ \frac{\Sigma \vdash \textbf{nil} <: V}{\Sigma \vdash \textbf{nil} <:_o V} \qquad \text{(S-FIELD8)}\ \frac{\Sigma \vdash \textbf{nil} <: V}{\Sigma \vdash \textbf{nil} <:_o \textbf{const}\ V}$$

Using *unique* table types to represent the type of the table constructor allows our type system to type check the previous example. More precisely,

$$\{\text{“}x\text{”} : 1, \text{“}y\text{”} : 2\}_{unique} <: \{\text{“}x\text{”} : \mathbf{integer}, \text{“}y\text{”} : \mathbf{integer} \cup \mathbf{nil}\}_{closed}$$

Even though we allow width subtyping between *unique* and *closed* table types, we do not allow it among *unique* and other table types because it would violate our definition of these other table types:

$$\text{(S-TABLE3)}\quad \frac{\begin{array}{c}\forall i \in 1..m \\ \exists j \in 1..n\ \Sigma \vdash K_i <: K'_j \wedge \Sigma \vdash V_i <:_u V'_j \\ \forall j \in 1..n\ \ \nexists i \in 1..m\ \Sigma \vdash K_i <: K'_j \rightarrow \Sigma \vdash \mathbf{nil} <:_o V'_j\end{array}}{\Sigma \vdash \{K_1{:}V_1, ..., K_m{:}V_m\}_{unique} <: \{K'_1{:}V'_1, ..., K'_n{:}V'_n\}_{unique|open|fixed}}$$

The rule that handles subtyping between *open* and *closed* table types allows width subtyping:

$$\text{(S-TABLE4)}\quad \frac{\begin{array}{c}\forall i \in 1..m\ \forall j \in 1..n\ \Sigma \vdash K_i <: K'_j \rightarrow \Sigma \vdash V_i <:_c V'_j \\ \forall j \in 1..n\ \ \nexists i \in 1..m\ \Sigma \vdash K_i <: K'_j \rightarrow \Sigma \vdash \mathbf{nil} <:_o V'_j\end{array}}{\Sigma \vdash \{K_1{:}V_1, ..., K_m{:}V_m\}_{open} <: \{K'_1{:}V'_1, ..., K'_n{:}V'_n\}_{closed}}$$

However, the rule that handles subtyping among *open* and *open* or *fixed* table types does not allow width subtyping:

$$\text{(S-TABLE5)}\quad \frac{\begin{array}{c}\forall i \in 1..m \\ \exists j \in 1..n\ \Sigma \vdash K_i <: K'_j \wedge \Sigma \vdash V_i <:_c V'_j \\ \forall j \in 1..n\ \ \nexists i \in 1..m\ \Sigma \vdash K_i <: K'_j \rightarrow \Sigma \vdash \mathbf{nil} <:_o V'_j\end{array}}{\Sigma \vdash \{K_1{:}V_1, ..., K_m{:}V_m\}_{open} <: \{K'_1{:}V'_1, ..., K'_n{:}V'_n\}_{open|fixed}}$$

The rules S-TABLE4 and S-TABLE5 allow joining fields plus omitting optional fields. Both rules use $<:_c$ to allow depth subtyping on **const** fields only.

We introduced *fixed* table types because we needed a safe way to represent the type of classes that can allow single inheritance through the refinement of table types. The rule that handles subtyping between *fixed* table types does not allow width subtyping, joining fields, and omitting fields, but it allows depth subtyping on **const** fields:

$$\frac{\forall i \in 1..n\ \exists j \in 1..n \quad \Sigma \vdash K_j <: K'_i \quad \Sigma \vdash K'_i <: K_j \quad \Sigma \vdash V_j <:_c V'_i}{\Sigma \vdash \{K_1{:}V_1, ..., K_n{:}V_n\}_{fixed} <: \{K'_1{:}V'_1, ..., K'_n{:}V'_n\}_{fixed}}\ \text{(S-TABLE6)}$$

In the next section we will show in more detail how our type system uses these tags to handle the refinement of table types.

We use the *Amber rule* [**?**] to define subtyping between recursive types:

$$\frac{\Sigma[x_1 <: x_2] \vdash T_1 <: T_2}{\Sigma \vdash \mu x_1.T_1 <: \mu x_2.T_2}\ \text{(S-AMBER)} \qquad \frac{x_1 <: x_2 \in \Sigma}{\Sigma \vdash x_1 <: x_2}\ \text{(S-ASSUMPTION)}$$

The rule S-AMBER also uses the rule S-ASSUMPTION to check whether $\mu x_1.T_1 <: \mu x_2.T_2$. Both rules use the set of assumptions Σ, where each assumption is a pair of recursion variables. The rule S-AMBER extends Σ with the assumption $x_1 <: x_2$ to check whether $T_1 <: T_2$. The rule S-ASSUMPTION allows the rule S-AMBER to check whether an assumption is valid.

A recursive type may appear inside a first-level type, and our type system includes subtyping rules to handle subtyping between recursive types and other first-level types:

$$\frac{\Sigma \vdash T_1 <: [x \mapsto \mu x.T_2]T_2}{\Sigma \vdash T_1 <: \mu x.T_2}\ \text{(S-UNFOLDR)} \qquad \frac{\Sigma \vdash [x \mapsto \mu x.T_1]T_1 <: T_2}{\Sigma \vdash \mu x.T_1 <: T_2}\ \text{(S-UNFOLDL)}$$

As an example, the rule S-UNFOLDR allows our type system to type check the function `insert` from Section **??**:

```
local function insert (e:Element?, v:integer):Element
  return { info = v, next = e }
end
```

that is, the type checker uses the rule S-UNFOLDR to verify whether the type of the table constructor is a subtype of `Element`:

$$\{\text{“}info\text{”} : \textbf{integer},\ \text{“}next\text{”} : \mu x.\{\text{“}info\text{”} : \textbf{integer}, \text{“}next\text{”} : x \cup \textbf{nil}\}_{closed} \cup \textbf{nil}\}_{unique} <: \mu x.\{\text{“}info\text{”} : \textbf{integer}, \text{“}next\text{”} : x \cup \textbf{nil}\}_{closed}$$

Filter types are subtypes only of themselves and of **value**. More precisely, a filter type $\phi(T_1, T_2)$ is a subtype of the same filter type $\phi(T_1, T_2)$, which shares the same types T_1 and T_2, and it is also a subtype of **value**.

Projection types are subtypes only of themselves and of **value**. More precisely, a projection type π_i^x is a subtype of the same projection type π_i^x, which shares the same union of tuples x and the same index i, and it is also a subtype of **value**.

The dynamic type **any** is neither the bottom nor the top type, but a separate type that is subtype only of itself and of **value**.

Even though the dynamic type **any** does not interact with subtyping, it does interact with consistent-subtyping. We present the consistent-subtyping rules as a deduction system for the consistent-subtyping relation $\Sigma \vdash T_1 \lesssim T_2$. As in the subtyping relation, Σ is also a set of pairs of recursion variables. We define the consistent-subtyping rules for the dynamic type **any** as follows:

$$\begin{array}{cc} \text{(C-ANY1)} & \text{(C-ANY2)} \\ \Sigma \vdash T \lesssim \textbf{any} & \Sigma \vdash \textbf{any} \lesssim T \end{array}$$

If we had set the type **any** as both bottom and top types of our subtyping relation, then any type T_1 would be a subtype of any other type T_2. The consequence of this is that all programs would type check without errors. This would happen due to the transitivity of subtyping, that is, we would be able to down-cast any type T_1 to **any** and then up-cast **any** to any other type T_2. The rules C-ANY1 and C-ANY2 are the rules that allow the dynamic type to interact with other first-level types, and thus allow dynamically typed code to coexist with statically typed code. Because of these two rules, consistent-subtyping cannot be transitive. These two rules are the only rules that differ between subtyping and consistent-subtyping, if we implement the subtyping rules as we do in this section.

In the implementation of Typed Lua we also use consistent-subtyping to normalize and simplify union types, though we let union types free in the formalization. For instance, the union type `boolean|any` results in the type `any`, because `boolean` is consistent-subtype of `any`. Another example is the union type `number|nil|1` that results in the union type `number|nil`, because `1` is consistent-subtype of `number`.

4.3 Typing rules

In this section we use a reduced core of Typed Lua to present the most interesting rules of our type system. These rules type check multiple assignment, table refinement, and overloading on the return type of functions. Appendix **??** presents the full set of typing rules.

Our core limits control flow to if and while statements; it has explicit type annotations, explicit scope for variables, explicit method declarations, and explicit method calls. Here is a list of features that are not present in our reduced core:

– labels and goto statements (they are difficult to handle along with our simplified form of *flow typing*, and they are out of scope for now);

– explicit blocks (we are already using explicit scope for variables);

– other loop structures such as repeat-until, numeric for, and generic for (we can use while to express them);

– table fields other than $[e_1] = e_2$ (we can use this form to express the missing forms);

– arithmetic operators other than + (other arithmetic operators have similar typing rules);

– relational operators other than == and < (inequality has similar typing rules to == and other relational operators have similar typing rules to <);

– bitwise operators other than & (other bitwise operators have similar typing rules).

Our reduced core does not lose much expressiveness, as it can express any Lua program except those that use labels and goto statements.

Figure **??** presents the abstract syntax of core Typed Lua. It splits the syntactic categories as follows: s are statements, e are expressions, l are left-hand values, k are literal constants, el are expression lists, me are expressions with multiple results, a are function and method applications, f are function declarations, pl are parameter lists, id are variable names, T are first-level types, and S are second-level types. The notation $\overline{id{:}T}$ denotes the list $id_1{:}T_1, ..., id_n{:}T_n$.

Our reduced core includes two statements for declaring local variables, one with and another without type annotations. While we use the former to formalize how our type system handles the declaration of annotated variables, we use the latter to formalize how our type system handles the declaration of unannotated variables through local type inference and also the introduction of projection types.

Our reduced core also includes a truncation operator $\lfloor\rfloor$ for function applications, method applications, and the vararg expression. We use $\lfloor a \rfloor_0$ to

Abstract Syntax

$s ::=$		STATEMENTS:
	$\textbf{skip}$	*skip*
	$\mid s_1 \; ; \; s_2$	*sequence*
	$\mid \overline{l} = el$	*multiple assignment*
	$\mid \textbf{while}\ e\ \textbf{do}\ s \mid \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2$	*control flow*
	$\mid \textbf{local}\ \overline{id{:}T} = el\ \textbf{in}\ s$	*variable declaration*
	$\mid \textbf{local}\ \overline{id} = el\ \textbf{in}\ s$	*variable declaration*
	$\mid \textbf{rec}\ id{:}T = f\ \textbf{in}\ s$	*recursive function*
	$\mid \textbf{return}\ el$	*return*
	$\mid \lfloor a \rfloor_0$	*application with no results*
	$\mid \textbf{fun}\ id_1{:}id_2\ (pl){:}S\ s\ ;\ \textbf{return}\ el$	*method declaration*
$e ::=$		EXPRESSIONS:
	$\textbf{nil}$	*nil*
	$\mid k$	*other literals*
	$\mid id$	*variable access*
	$\mid e_1[e_2]$	*table access*
	$\mid <T>\ id$	*type coercion*
	$\mid f$	*function declaration*
	$\mid \{\ \overline{[e_1] = e_2}\ \} \mid \{\ \overline{[e_1] = e_2}, me\ \}$	*table constructor*
	$\mid e_1 + e_2 \mid e_1\ ..\ e_2 \mid e_1 == e_2 \mid e_1 < e_2$	*binary operations*
	$\mid e_1\ \&\ e_2 \mid e_1\ \textbf{and}\ e_2 \mid e_1\ \textbf{or}\ e_2$	*binary operations*
	$\mid \textbf{not}\ e \mid \#\ e$	*unary operations*
	$\mid \lfloor me \rfloor_1$	*expressions with one result*
$l ::=$		LEFT-HAND VALUES:
	id_l	*variable assignment*
	$\mid e_1[e_2]$	*table assignment*
	$\mid id[k]\ <T>$	*type coercion*
$k ::=$		LITERAL CONSTANTS:
	$\textbf{false} \mid \textbf{true} \mid int \mid float \mid string$	
$el ::=$		EXPRESSION LISTS:
	$\overline{e} \mid \overline{e}, me$	
$me ::=$		MULTIPLE RESULTS:
	a	*application*
	$\mid ...$	*vararg expression*
$a ::=$		APPLICATIONS:
	$e(el)$	*function application*
	$\mid e{:}n(el)$	*method application*
$f ::=$		FUNCTION DECLARATIONS:
	$\textbf{fun}\ (pl){:}S\ s\ ;\ \textbf{return}\ el$	
$pl ::=$		PARAMETER LISTS:
	$\overline{id{:}T} \mid \overline{id{:}T}, ...{:}T$	

Figure 4.2: The abstract syntax of Typed Lua

denote function and method applications that produce no value, because they appear as statements. We use $\lfloor me \rfloor_1$ to denote function applications, method applications, and vararg expressions that produce only one value, even if they return multiple values.

We also include two kinds of type coercions in our core language: the left-hand value $id[k]$ <T> and the expression <T> id. Both allow the refinement of table types. We also split variable names into two categories to have safe aliasing of tables in the presence of refinement. We use id when variable names appear as expressions and id_l when variable names appear as left-hand values.

Even though we can assign only first-level types to variables, functions and methods can return unions of second-level types, and our type system should be able to project these unions of second-level types into unions of first-level types. We use two different environments to handle this feature. The first environment is the type environment Γ that maps variables to first-level types. We use $\Gamma_1[id \mapsto T]$ to extend the environment Γ_1 with the variable id that maps to type T. The second environment is the projection environment Π that maps projection variables to second-level types. We use $\Pi[x \mapsto S]$ to extend the environment Π with the projection variable x that maps to type S. In Section **??** we will show how our type system uses the projection environment Π for handling projection types, and also for projecting unions of second-level types into unions of first-level types.

We present the typing rules as a deduction system for two typing relations, one for typing statements and another for typing expressions.

We use the relation $\Gamma_1, \Pi \vdash s, \Gamma_2$ for typing statements. This relation means that given a type environment Γ_1 and a projection environment Π, we can check that a statement s produces a new type environment Γ_2.

We use the relation $\Gamma_1, \Pi \vdash e : T, \Gamma_2$ for typing expressions. This relation means that given a type environment Γ_1 and a projection environment Π, we can check that an expression e has type T and produces a new type environment Γ_2.

(a) Assignment and function application

Lua has multiple assignment, and our type system uses three different kinds of typing rules to type check this feature. It uses typing rules that type check the different forms of expression lists that can appear in the right-hand side, a typing rule that type checks a list of left-hand values, and a general rule that uses consistent-subtyping to check whether the type of the right-hand side is consistent with the type of the left-hand side.

As an example, lets assume that x and y are variables in the environment with types **integer** and **string**. Let us see how our type system type checks the following assignment:

$$x, y = 1, \text{“}foo\text{”}$$

First, our type system type checks the expression list in the right-hand side of the assignment. In our example, the right-hand side of the assignment has type $1 \times \text{“}foo\text{”} \times \textbf{nil}*$. Note that our type system includes the type **nil**$*$ to replace missing values. The rules that type check expression lists introduce the type **nil**$*$ to let the right-hand side produce fewer values than expected in the left-hand side. Our example uses the rule T-EXPLIST1 to type check the right-hand side of the assignment. The rule T-EXPLIST1 is the rule that type checks an expression list where all expressions can only produce a single value:

$$\text{(T-EXPLIST1)}$$
$$\frac{\Gamma_1, \Pi \vdash e_i : T_i, \Gamma_{i+1} \quad \Gamma_f = merge(\Gamma_1, ..., \Gamma_{n+1}) \quad n = \mid \overline{e} \mid}{\Gamma_1, \Pi \vdash \overline{e} : T_1 \times ... \times T_n \times \textbf{nil}*, \Gamma_f}$$

In Section **??** we will show that table refinement can change the type environment while typing an expression or a left-hand value. Thus, the rules that type check lists of expressions and lists of left-hand values use a partial auxiliary function *merge* to collect all environment changes in a new environment Γ_m, if there are no conflicts. In Section **??** we will also show that we can only change the environment to add new table fields in a table type, and we cannot change the type of a variable or a table field which is already present in a table type.

After type checking the right-hand side, our type system type checks the list of left-hand values. In our example, the left-hand side of the assignment has type **integer** $\times$ **string** $\times$ **value**$*$. Note that our type system uses the type **value**$*$ to discard extra values. The rule that type checks lists of left-hand values introduces the type **value**$*$ to let the right-hand side produce more values than expected in the left-hand side. Our example uses the rule T-LHSLIST to type check a list of left-hand values:

$$\text{(T-LHSLIST)}$$
$$\frac{\Gamma_1, \Pi \vdash l_i : T_i, \Gamma_{i+1} \quad \Gamma_f = merge(\Gamma_1, ..., \Gamma_{n+1}) \quad n = \mid \overline{l} \mid}{\Gamma_1, \Pi \vdash \overline{l} : T_1 \times ... \times T_n \times \textbf{value}*, \Gamma_f}$$

After type checking the right-hand side and the left-hand side of an assignment, our type system checks whether their types are consistent. The rule T-ASSIGNMENT is the general rule that expresses this idea:

$$\text{(T-ASSIGNMENT)}$$
$$\frac{\Gamma_1, \Pi \vdash el : S_1, \Gamma_2 \quad \Gamma_2, \Pi \vdash \bar{l} : S_2, \Gamma_3 \quad S_1 \lesssim S_2}{\Gamma_1, \Pi \vdash \bar{l} = el, \Gamma_3}$$

Back to our example, it type checks through rule T-ASSIGNMENT because

$$1 \times \text{``}foo\text{''} \times \mathbf{nil}* \lesssim \mathbf{integer} \times \mathbf{string} \times \mathbf{value}*$$

As another example, lets assume that x, y, and z are variables in the environment with types **integer**, **string**, and **string** $\cup$ **nil**. The assignment

$$x, y, z = 1, \text{``}foo\text{''}$$

type checks because

$$1 \times \text{``}foo\text{''} \times \mathbf{nil}* \lesssim \mathbf{integer} \times \mathbf{string} \times (\mathbf{string} \cup \mathbf{nil}) \times \mathbf{value}*$$

Note how **nil**$*$ replaces any missing values. This example type checks because **nil**$*$ produces as many **nil** values as we need, and **nil** is consistent with **string** $\cup$ **nil**, which is the type of z.

Conversely, the assignment

$$x = 1, \text{``}foo\text{''}$$

type checks because

$$1 \times \text{``}foo\text{''} \times \mathbf{nil}* \lesssim \mathbf{integer} \times \mathbf{value}*$$

Note how **value**$*$ discards extra values. This example type checks because **value**$*$ discards as many extra values as we need, and "foo" is consistent with **value**.

Our type system includes **nil**$*$ in the type of an expression list only if its type does not end in another variadic type $T*$. For instance, the rules T-EXPLIST2 and T-EXPLIST3 handle the case where an expression list ends with an expression that may produce multiple values. The former rule includes the type **nil**$*$, but the latter rule does not:

$$\frac{\Gamma_1, \Pi \vdash e_i : T_i, \Gamma_{i+1} \quad \Gamma_1, \Pi \vdash me : T_{n+1} \times ... \times T_{n+m} \times \mathbf{void}, \Gamma_{n+2} \qquad \Gamma_f = merge(\Gamma_1, ..., \Gamma_{n+2}) \quad n = |\ \overline{e}\ |}{\Gamma_1, \Pi \vdash \overline{e}, me : T_1 \times ... \times T_{n+m} \times \mathbf{nil}*, \Gamma_f}\ \text{(T-EXPLIST2)}$$

$$\frac{\Gamma_1, \Pi \vdash e_i : T_i, \Gamma_{i+1} \quad \Gamma_1, \Pi \vdash me : T_{n+1} \times ... \times T_{n+m}*, \Gamma_{n+2} \qquad \Gamma_f = merge(\Gamma_1, ..., \Gamma_{n+2}) \quad n = |\ \overline{e}\ |}{\Gamma_1, \Pi \vdash \overline{e}, me : T_1 \times ... \times T_{n+m}*, \Gamma_f}\ \text{(T-EXPLIST3)}$$

Rules for function applications are similar to the rule for multiple assignment. The rule T-APPLY1 handles the case where function applications are expressions that produce multiple values:

$$\frac{\Gamma_1, \Pi \vdash e : S_1 \rightarrow S_2, \Gamma_2 \quad \Gamma_2, \Pi \vdash el : S_3, \Gamma_3 \quad S_3 \lesssim S_1}{\Gamma_1, \Pi \vdash e(el) : S_2, \Gamma_3}\ \text{(T-APPLY1)}$$

We also use the rule T-APPLY1 as the base case for the rules that handle the cases where function applications are either statements that produce no value or expressions that produce only one value. The rule T-STMAPPLY1 discards the produced values, while the rule T-EXPAPPLY1 uses the auxiliary function *first* to ensure that only one value is produced:

$$\frac{\Gamma_1, \Pi \vdash e(el) : S, \Gamma_2}{\Gamma_1, \Pi \vdash \lfloor e(el) \rfloor_0, \Gamma_2}\ \text{(T-STMAPPLY1)} \qquad \frac{\Gamma_1, \Pi \vdash e(el) : S, \Gamma_2}{\Gamma_1, \Pi \vdash \lfloor e(el) \rfloor_1 : first(S), \Gamma_2}\ \text{(T-EXPAPPLY1)}$$

We can define *first* inductively as follows:

$$\begin{aligned} first(\mathbf{void}) &= \mathbf{nil} \\ first(T*) &= T \cup \mathbf{nil} \\ first(T \times P) &= T \\ first(S_1 \sqcup S_2) &= first(S_1) \cup first(S_2) \end{aligned}$$

As an example, let us assume that f is a local function in the environment, and that f has type $\mathbf{string} \times (\mathbf{integer} \cup \mathbf{nil}) \times (\mathbf{integer} \cup \mathbf{nil}) \times \mathbf{value}* \rightarrow \mathbf{integer}*$. The function call

$$f(\text{``}foo\text{''})$$

type checks through the rule T-APPLY1, because

$$\text{"}foo\text{"} \times \mathbf{nil}{*} \lesssim \mathbf{string} \times (\mathbf{integer} \cup \mathbf{nil}) \times (\mathbf{integer} \cup \mathbf{nil}) \times \mathbf{value}{*}$$

and the function call

$$f(\text{"}foo\text{"}, 1, 2, 3)$$

also type checks through the rule T-APPLY1, because

$$\text{"}foo\text{"} \times 1 \times 2 \times 3 \times \mathbf{nil}{*} \lesssim \mathbf{string} \times (\mathbf{integer} \cup \mathbf{nil}) \times (\mathbf{integer} \cup \mathbf{nil}) \times \mathbf{value}{*}$$

Our type system also catches arity mismatch. To do that, we end the input type of a function with type **void** instead of **value**$*$. For instance, let us assume that f has type $\mathbf{string} \times (\mathbf{integer} \cup \mathbf{nil}) \times (\mathbf{integer} \cup \mathbf{nil}) \times \mathbf{void} \rightarrow \mathbf{integer}{*}$. The function call

$$f(\text{"}foo\text{"})$$

type checks through the rule T-APPLY1, because

$$\text{"}foo\text{"} \times \mathbf{nil}{*} \lesssim \mathbf{string} \times (\mathbf{integer} \cup \mathbf{nil}) \times (\mathbf{integer} \cup \mathbf{nil}) \times \mathbf{void}$$

but the function call

$$f(\text{"}foo\text{"}, 1, 2, 3)$$

does not type check through the rule T-APPLY1, because

$$\text{"}foo\text{"} \times 1 \times 2 \times 3 \times \mathbf{nil}{*} \not\lesssim \mathbf{string} \times (\mathbf{integer} \cup \mathbf{nil}) \times (\mathbf{integer} \cup \mathbf{nil}) \times \mathbf{void}$$

We just mentioned that when our type system type checks an expression list, it always includes **nil**$*$ in the end of the type of this expression list if its type does not end in a variadic type. This behavior preserves the semantics of Lua on replacing missing values, and it is necessary when we omit optional parameters in a function call, like the previous example showed.

Using **nil**$*$ in the end of the type of expression lists also allows our type system to catch arity mismatch in function calls without optional parameters. For instance, let us assume that f has type $\mathbf{integer} \times \mathbf{integer} \times \mathbf{void} \rightarrow \mathbf{integer} \times \mathbf{void}$. The function call

$$f(1)$$

does not type check through the rule T-APPLY1, because

$$1 \times \mathbf{nil}{*} \not\lesssim \mathbf{integer} \times \mathbf{integer} \times \mathbf{void}$$

and the function call

$$f(1, 2, 3)$$

also does not type check through the rule T-APPLY1, because

$$1 \times 2 \times 3 \times \mathbf{nil}* \not\lesssim \mathbf{integer} \times \mathbf{integer} \times \mathbf{void}$$

(b) Tables and refinement

Our abstract syntax reduces the syntactic forms of the table constructor into two forms: $\{\ \overline{[e_1] = e_2}\ \}$ and $\{\ \overline{[e_1] = e_2}, me\ \}$. The first uses a list of table fields $([e_1] = e_2)_1, ..., ([e_1] = e_2)_n$. The second uses a list of table fields and an expression that can produce multiple values.

The simplest expression involving tables is the empty table constructor. Its type checking rule is straightforward:

$$\text{(T-CONSTRUCTOR1)}$$
$$\Gamma_1, \Pi \vdash \{\} : \{\}_{unique}, \Gamma_1$$

As a more interesting example, let us see how our type system type checks the table constructor $\{[1] = \text{“}x\text{”}, [2] = \text{“}y\text{”}, [3] = \text{“}z\text{”}\}$.

First, our type system uses the auxiliary relation $\Gamma_1, \Pi \vdash [e_1] = e_2 : (K, V), \Gamma_2$ to type check each table field. This auxiliary relation means that given a type environment Γ_1 and a projection environment Π, checking a table field $[e_1] = e_2$ produces a pair (K, V) and a new type environment Γ_2. A pair (K, V) means that e_1 has type K and e_2 has type V, where K is the type of the key and V is the type of the value.

After type checking each table field, our type system uses each pair (K, V) to build the table type that express the type of a given constructor, and uses the predicate *wf* to check whether this table type is well-formed. Formally, a table type is well-formed if it obeys the following rule:

$$\forall i\ \nexists j\ i \neq j \wedge K_i \lesssim K_j$$

Well-formed table types avoid ambiguity. For instance, this rule detects that the table type $\{1 : \mathbf{number}, \mathbf{integer} : \mathbf{string}, \mathbf{any} : \mathbf{boolean}\}$ is ambiguous, because the type of the value stored by key 1 can be **number**, **string**, or **boolean**, as $1 \lesssim 1$, $1 \lesssim \mathbf{integer}$, and $1 \lesssim \mathbf{any}$. Moreover, the type of the value stored by a key of type **integer**, which is not the literal type 1, can be **number** or **boolean**, as $\mathbf{integer} \lesssim \mathbf{integer}$, and $\mathbf{integer} \lesssim \mathbf{any}$.

Well-formed table types also do not allow *unique* and *open* table types to appear in the type of the values. We made this restriction because our type system does not keep track of aliases to table fields. This means that allowing *unique* and *open* table types to appear in the type of a value would allow the creation of unsafe aliases. The rules that type check table fields use the auxiliary function *close* to close the type of the values in a table type.

The rule T-CONSTRUCTOR2 uses these steps to type check a table constructor with a non-empty list of table fields:

$$\text{(T-CONSTRUCTOR2)}\quad \frac{\begin{array}{c}\Gamma_1, \Pi \vdash ([e_1] = e_2)_i : (K_i, V_i), \Gamma_{i+1} \\ T = \{K_1{:}V_1, ..., K_n{:}V_n\}_{unique} \quad wf(T) \quad n = |\ \overline{[e_1] = e_2}\ | \\ \Gamma_f = merge(\Gamma_1, ..., \Gamma_{n+1})\end{array}}{\Gamma_1, \Pi \vdash \{\ \overline{[e_1] = e_2}\ \} : T, \Gamma_f}$$

Back to our example, the constructor $\{[1] = \text{“}x\text{”}, [2] = \text{“}y\text{”}, [3] = \text{“}z\text{”}\}$ has type $\{1 : \text{“}x\text{”}, 2 : \text{“}y\text{”}, 3 : \text{“}z\text{”}\}_{unique}$ through rule T-CONSTRUCTOR2.

As another example, the constructor $\{[\text{“}x\text{”}] = 1, [\text{“}y\text{”}] = \{[\text{“}z\text{”}] = 2\}\}$ has type $\{\text{“}x\text{”} : 1, \text{“}y\text{”} : \{\text{“}z\text{”} : 2\}_{closed}\}_{unique}$ through rule T-CONSTRUCTOR2. The inner table is *closed* to prevent the creation of unsafe aliases.

After presenting some typing rules of the table constructor, we start the discussion of the rules that define the most unusual feature of our type system: the refinement of table types. The first kind of refinement allows programmers to add new fields to *unique* or *open* table types through field assignment. For instance, in Section **??** we presented the following example:

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

We can translate this example to our reduced core as follows:

$$\begin{array}{l}\textbf{local}\ person = \{\}\ \textbf{in} \\ \quad person[\text{“}firstname\text{”}]\ <\textbf{string}> = \text{“}Lou\text{”}; \\ \quad person[\text{“}lastname\text{”}]\ <\textbf{string}> = \text{“}Reed\text{”}\end{array}$$

In this example, we assign the type $\{\}_{unique}$ to the variable $person$, then we refine its type to $\{\text{“}firstname\text{”} : \textbf{string}\}_{unique}$, and then we refine its type to $\{\text{“}firstname\text{”} : \textbf{string}, \text{“}lastname\text{”} : \textbf{string}\}_{unique}$. Rule T-REFINE type checks this use of refinement:

$$\text{(T-REFINE)}\quad \frac{\begin{array}{c}\Gamma_1(id) = \{K_1{:}V_1, ..., K_n{:}V_n\}_{open|unique} \\ \Gamma_1, \Pi \vdash k : K, \Gamma_2 \quad \nexists i \in 1..n\ K \lesssim K_i \quad V = close(T)\end{array}}{\Gamma_1, \Pi \vdash id[k]{<}T{>} : V, \Gamma_2[id \mapsto \{K_1{:}V_1, ..., K_n{:}V_n, K{:}V\}_{open|unique}]}$$

We restricted the refinement of table types to include only literal keys, because its purpose is to make it easier the construction of table types that represent records.

We use the refinement of table types to handle the declaration of new global variables. In Lua, the assignment `v = v + 1` translates to `_ENV["v"] = _ENV["v"] + 1` when `v` is not a local variable, where `_ENV` is a table that stores the global environment. For this reason, Typed Lua treats accesses to global variables as field accesses to an *open* table in the top-level scope. In the following examples we assume that $\_ENV$ is in the environment and has type $\{\}_{open}$.

As an example,

$$\_ENV[\text{“}x\text{”}] <\textbf{string}> = \text{“}foo\text{”} \;;\; \_ENV[\text{“}y\text{”}] <\textbf{integer}> = 1$$

uses field assignment to add fields “x” and “y” to $\_ENV$. Therefore, after these field assignments $\_ENV$ has type $\{\text{“}x\text{”} : \textbf{string}, \text{“}y\text{”} : \textbf{integer}\}_{open}$.

We do not allow the refinement of table types to add a field if it is already present in the table's type. For instance,

$$\_ENV[\text{“}x\text{”}] <\textbf{string}> = \text{“}foo\text{”} \;;\; \_ENV[\text{“}x\text{”}] <\textbf{integer}> = 1$$

does not type check, as we are trying to add “x” twice.

We also do not allow the refinement of table types to introduce fields with table types that are not *closed*. For instance,

$$\_ENV[\text{“}x\text{”}] <\{\}_{unique}> = \{\}$$

refines the type of $\_ENV$ from $\{\}_{open}$ to $\{\text{“}x\text{”} : \{\}_{closed}\}_{open}$. Currently, our type system can only track *unique* and *open* table types that are bound to local variables.

We can also use multiple assignment to refine table types:

$$\_ENV[\text{“}x\text{”}] <\textbf{string}>, \_ENV[\text{“}y\text{”}] <\textbf{integer}> = \text{“}foo\text{”}, 1$$

This example type checks because all the environment changes are consistent, and “foo” $\times 1 \times \textbf{nil}* \lesssim \textbf{string} \times \textbf{integer} \times \textbf{value}*$. By consistent we mean that we are only adding new fields. Nevertheless, the next example does not type check because it tries to add the same field to $\_ENV$, but with different types:

$$\_ENV[\text{“}x\text{”}] <\textbf{string}>, \_ENV[\text{“}x\text{”}] <\textbf{integer}> = \text{“}foo\text{”}, 1$$

Aliasing an *unique* or an *open* table type can produce either a *closed* or a *fixed* table type, depending on the context that we are using a variable. As we mentioned in Sections **??** and **??**, we need *fixed* table types to type classes in object-oriented programming. In the implementation we fix the aliasing of *unique* and *open* table types that appear in a top-level return statement, and in other cases we close the aliasing of *unique* and *open* table types. However, in the formalization we chose to define this behavior in a not deterministic way, as it makes easier the presentation of this behavior.

As an example,

$$\begin{array}{l}
\textbf{local}\ a : \{\}_{unique} = \{\}\ \textbf{in} \\
\quad \textbf{local}\ b : \{\}_{open} = a\ \textbf{in} \\
\quad\quad a[\text{“}x\text{”}]\ <\textbf{string}> = \text{“}foo\text{”}; \\
\quad\quad b[\text{“}x\text{”}]\ <\textbf{integer}> = 1
\end{array}$$

does not type check, as aliasing a produces the type $\{\}_{closed}$ that is not a subtype of $\{\}_{open}$, the type of b. Our type system has this behavior to warn programmers about potential unsafe behaviors after this kind of alias. In this example, it is unsafe to add the field “x” to b, as it changes the value that is stored in the field “x” of a.

Rules T-IDREAD1 and T-IDREAD2 define this non-deterministic behavior. Rule T-IDREAD1 uses the auxiliary function *close* to produce a *closed* alias. It also uses the auxiliary function *open* to change the type of the original reference from *unique* to *open*, because aliasing an *unique* table type while keeping the original reference *unique* can be unsafe. Rule T-IDREAD2 uses the auxiliary function *fix* to produce a *fixed* alias. It also uses *fix* to change the type of the original reference to *fixed*, because a *fixed* table type does not allow width subtyping. We define these rules as follows:

$$\text{(T-IDREAD1)}\quad \frac{\Gamma_1(id) = T_1 \quad T_2 = read(\Pi, T_1)}{\Gamma_1, \Pi \vdash id : close(T_2), \Gamma_1[id \mapsto open(T_1)]}$$

$$\text{(T-IDREAD2)}\quad \frac{\Gamma_1(id) = T_1 \quad T_2 = read(\Pi, T_1)}{\Gamma_1, \Pi \vdash id : fix(T_2), \Gamma_1[id \mapsto fix(T_1)]}$$

Both rules use the auxiliary function *read* because they may be accessing an identifier that is bound to a filter or projection type. As we mentioned in Section **??**, our type system includes a small set of `type` predicates that allow programmers to discriminate union types, and our type system uses filter and projection types in the definition of these predicates to handle the discrimination of unions types. While filter types discriminate unions of first-level types, projection types discriminate unions of second-level types and project unions of second-level types into unions of first-level types. We can define *read* as follows:

$$\begin{aligned} \mathit{read}(\Pi, \phi(T_1, T_2)) &= T_2 \\ \mathit{read}(\Pi, \pi_i^x) &= \mathit{proj}(\Pi(x), i) \\ \mathit{read}(\Pi, T) &= T \end{aligned}$$

The function *read* uses the auxiliary function *proj* to project a union of first-level types, based on an union of second-level types and an index from a projection type. In Section **??** we will discuss how our type system uses projection types to handle overloaded return types. We can define *proj* as follows:

$$\begin{aligned} \mathit{proj}(T_1 \times ... \times T_n \times T{*}, i) &= T_i \quad \text{if } i <= n \\ \mathit{proj}(S_1 \sqcup S_2, i) &= \mathit{proj}(S_1, i) \cup \mathit{proj}(S_2, i) \end{aligned}$$

We also need to close *unique* and *open* tables that appear in the left-hand side of assignments, as leaving them *unique* and *open* would allow the creation of unsafe references.

As an example,

$$\begin{array}{l} \textbf{local}\ a : \{\}_{unique} = \{\}\ \textbf{in} \\ \quad \textbf{local}\ b : \{\}_{open} = \{\}\ \textbf{in} \\ \quad\quad b = a; \\ \quad\quad a[\text{“}x\text{”}]\ \textbf{<string>} = \text{“}foo\text{”}; \\ \quad\quad b[\text{“}x\text{”}]\ \textbf{<integer>} = 1 \end{array}$$

does not type check because we cannot add the field “x” to b, as its type is *closed*. Aliasing a changes the type of a from *unique* to *open*, and that is the reason why we can add the field “x” to the type of a. Aliasing a also produces the type $\{\}_{closed}$, which is the same type that b has in left-hand side of the assignment. After the assignment, the type of b is *closed* and thus does not allow changing the value that is stored in the field “x” of a.

Rule T-IDWRITE defines this behavior:

$$\text{(T-IDWRITE)}\quad \frac{\Gamma_1(id) = T_1 \quad T_2 = \mathit{write}(T_1)}{\Gamma_1, \Pi \vdash id_l : \mathit{close}(T_2), \Gamma_1[id \mapsto \mathit{close}(T_2)]}$$

This rule uses the auxiliary function *write* because it may be accessing an identifier that is bound to a filter type. As we mentioned in Sections **??** and **??**, assignments restore discriminated union types to their original types, and function *write* works in this purpose. We can define *write* as follows:

$$\begin{aligned} write(\phi(T_1, T_2)) &= T_1 \\ write(T) &= T \end{aligned}$$

Our type system also has different rules for type checking table indexing to avoid changing table types in these operations, as they cannot create aliases:

$$\text{(T-INDEX1)}\quad \frac{\begin{array}{c}\Gamma_1(id) = T \quad read(\Pi, T) = \{K_1{:}V_1, ..., K_n{:}V_n\} \\ \Gamma_1, \Pi \vdash e_2 : K, \Gamma_2 \quad \exists i \in 1..n\ K \lesssim K_i\end{array}}{\Gamma_1, \Pi \vdash id[e_2] : V_i, \Gamma_2}$$

$$\text{(T-INDEX2)}\quad \frac{\begin{array}{c}\Gamma_1, \Pi \vdash e_1 : \{K_1{:}V_1, ..., K_n{:}V_n\}, \Gamma_2 \\ \Gamma_2, \Pi \vdash e_2 : K, \Gamma_3 \quad \exists i \in 1..n\ K \lesssim K_i\end{array}}{\Gamma_1, \Pi \vdash e_1[e_2] : V_i, \Gamma_3}$$

A second form of refinement happens when we want to use an *unique* or *open* table type in a context that expects a *fixed* or *closed* table type with a different shape. This kind of refinement allows programmers to add optional fields or merge existing fields. To do that, Typed Lua includes a type coercion expression $<T>$ *id*. For instance, we can use this type coercion expression to make the following example type check:

```
**local** $a : \{\}_{unique} = \{\}$ **in**
  $a[“x”]$ <**string**> $=$ “$foo$”;
  $a[“y”]$ <**string**> $=$ “$bar$”;
  **local** $b : \{“x” : \textbf{string}, “y” : \textbf{string} \cup \textbf{nil}\}_{closed} =$
    $<\{“x” : \textbf{string}, “y” : \textbf{string} \cup \textbf{nil}\}_{open}>\ a$ **in** $a[“z”]$ <**integer**> $= 1$
```

We can use a to initialize b because the coercion converts the type of a from $\{“x” : \textbf{string}, “y” : \textbf{string}\}_{unique}$ to $\{“x” : \textbf{string}, “y” : \textbf{string}{\cup}\textbf{nil}\}_{open}$, and results in $\{“x” : \textbf{string}, “y” : \textbf{string} \cup \textbf{nil}\}_{closed}$, which is a subtype of $\{“x” : \textbf{string}, “y” : \textbf{string} \cup \textbf{nil}\}_{closed}$, the type of b. We can continue to refine the type of a after aliasing it to b, as it still holds an *open* table. At the end of this example, a has type $\{“x” : \textbf{string}, “y” : \textbf{string} \cup \textbf{nil}, “z” : \textbf{integer}\}_{open}$.

Rule T-COERCE defines the behavior of the type coercion expression:

$$\text{(T-COERCE)}\quad \frac{\Gamma_1(id) <: T \quad \Gamma_1[id \mapsto T], \Pi \vdash id : T_1, \Gamma_2}{\Gamma_1, \Pi \vdash <T>\ id : T_1, \Gamma_2}$$

Note that rule T-COERCE only allows changing the type of a variable if the new type is a supertype of the previous type, and the resulting type is always *fixed* or *closed* to prevent the creation of unsafe aliases.

We also need to make sure to close all *unique* and *open* table types before we type check a nested scope. To do that, our type system uses some auxiliary functions to change the type of variables before type checking a nested scope and also to change the type of assigned and referenced variables after type checking a nested scope. The function *crall* closes all *unique* and *open* table types; it also restores filter types to their original types. The function *crset* closes a given set of free assigned variables, which is given by the function *fav*, and it also uses this set to restore filter types to their original types. The function *openset* changes from *unique* to *open* a given set of referenced variables, which is given by the function *rv*.

As an example,

```
local $a : \{\}_{unique}, b : \{\}_{unique} = \{\}, \{\}$ in
  local $f$ : integer $\times$ void $\rightarrow$ integer $\times$ void =
    fun ($x$ : integer) : integer $\times$ void
      $b = a$ ; return $x + 1$
  in $a[\text{"}x\text{"}]$ <integer> = 1 ; $b[\text{"}x\text{"}]$ <string> = "$foo$" ; $f(a[\text{"}x\text{"}])$
```

does not type check because we cannot add the field "x" to b, as its type is closed. The assignment $b = a$ type checks because, at that point, a and b have the same type: $\{\}_{closed}$. Their type was closed by *crall* before type checking the function body. Their type would be restored to $\{\}_{unique}$ after type checking the function body, but that assignment also triggers other two type changes. First, the function *fav* includes b in the set of variables that should be closed by *crset*. Then, the function *rv* includes a in the set of variables that should change from *unique* to *open* by *openset*. After declaring f, a has type $\{\}_{open}$ and b has type $\{\}_{closed}$, so we can refine the type of a, but we cannot refine the type of b.

Rule T-FUNCTION1 illustrates this case:

$$\text{(T-FUNCTION1)}\quad \frac{\begin{array}{c} crall(\Gamma_1[\overline{id} \mapsto \overline{T}]), \Pi[\rho \mapsto S] \vdash s, \Gamma_2 \\ \Gamma_3 = openset(crset(\Gamma_1, fav(\textbf{fun}\ (\overline{id{:}T}){:}S\ s)), rv(\textbf{fun}\ (\overline{id{:}T}){:}S\ s)) \end{array}}{\Gamma_1, \Pi \vdash \textbf{fun}\ (\overline{id{:}T}){:}S\ s : \overline{T} \times \textbf{void} \rightarrow S, \Gamma_3}$$

This rule also extends the environment Π, bounding the special variable ρ to the return type S. Rule T-RETURN uses the type that is bound to ρ in Π to type check return statements:

$$\text{(T-RETURN)}\quad \frac{\Gamma_1 \vdash el : S_1, \Gamma_2 \quad \Pi(\rho) = S_2 \quad S_1 \lesssim S_2}{\Gamma_1 \vdash \mathbf{return}\ el, \Gamma_2}$$

(c) Projections

Lua programmers often overload the return type of functions to denote errors, and our type system uses projection types to handle this idiom.

As an example, let us assume that *idiv* and *print* are functions in the environment. The function *idiv* has type

$$\mathbf{integer} \times \mathbf{integer} \times \mathbf{void} \to (\mathbf{integer} \times \mathbf{integer} \times \mathbf{void}) \sqcup (\mathbf{nil} \times \mathbf{string} \times \mathbf{void})$$

As we mentioned in Section **??**, *idiv* performs integer division. In case of success, it returns two integers: the result and the remainder. In case of failure, it returns **nil** plus an error message that describes the error. The function *print* is a variadic function of type $\mathbf{value}* \to \mathbf{void}$. Let us see how our type system type checks the following program:

$$\begin{array}{l} \mathbf{local}\ q, r = idiv(1, 2)\ \mathbf{in} \\ \quad \mathbf{if}\ q\ \mathbf{then}\ \lfloor print(q + r) \rfloor_0\ \mathbf{else}\ \lfloor print(\text{“}ERROR: \text{”}\ ..\ r) \rfloor_0 \end{array}$$

First, our type system uses the auxiliary relation $\Gamma_1, \Pi \vdash el : S_1, \Gamma_2, (x, S_2)$ for type checking $idiv(1, 2)$. This relation means that given a type environment Γ_1 and a projection environment Π, we can check that an expression list el has type S_1 and produces a new type environment Γ_2 and produces a pair (x, S_2). This pair means that the last expression of an expression list el produces an union of second-level types S_2 that should be bound to a variable x in the projection environment Π, as the resulting type of this expression is a tuple of projection types π_i^x. In our example, our type system uses rule T-EXPLIST4 for type checking $idiv(1, 2)$:

$$\text{(T-EXPLIST4)}\quad \frac{\begin{array}{c} \Gamma_1, \Pi \vdash e_i : T_i, \Gamma_{i+1} \quad \Gamma_1, \Pi \vdash me : S, \Gamma_{n+2} \\ S = T_{n+1} \times ... \times T_{n+m} \times \mathbf{void} \sqcup T'_{n+1} \times ... \times T'_{n+m} \times \mathbf{void} \\ \Gamma_f = merge(\Gamma_1, ..., \Gamma_{n+2}) \quad n = |\, \overline{e}\, | \end{array}}{\Gamma_1, \Pi \vdash \overline{e}, me : T_1 \times ... \times T_n \times \pi_1^x \times ... \times \pi_m^x \times \mathbf{nil}*, \Gamma_f, (x, S)}$$

Note that $idiv(1, 2)$ has type $\pi_1^x \times \pi_2^x \times \mathbf{nil}*$ and produces the pair

$$(x, (\mathbf{integer} \times \mathbf{integer} \times \mathbf{nil}*) \sqcup (\mathbf{nil} \times \mathbf{string} \times \mathbf{nil}*))$$

In the rule that type checks the declaration of unannotated variables, our type system uses the pair (x, S_2) to bound a union of second-level types S_2 to a variable x in the projection environment Π. In our example, declaring q and r bounds the projection type π_1^x to q and bounds the projection type π_2^x to r, where the projection variable x bounds to

$$(\mathbf{integer} \times \mathbf{integer} \times \mathbf{nil}*) \sqcup (\mathbf{nil} \times \mathbf{string} \times \mathbf{nil}*)$$

in the projection environment Π. Rule T-LOCAL2 illustrates this intuition:

$$\text{(T-LOCAL2)}$$
$$\frac{\begin{array}{c}\Gamma_1, \Pi \vdash el : S_1, \Gamma_2, (x, S_2) \\ \Gamma_2[id_1 \mapsto infer(S_1, 1), ..., id_n \mapsto infer(S_1, n)], \Pi[x \mapsto S_2] \vdash s, \Gamma_3 \quad n = \mid \overline{id} \mid\end{array}}{\Gamma_1, \Pi \vdash \mathbf{local}\ \overline{id} = el\ \mathbf{in}\ s, \Gamma_3 - \{\overline{id}\}}$$

This rule uses the auxiliary function *infer* to get the most general first-level types of each variable that should be introduced in the type environment for type checking s. After type checking the statement s, rule T-LOCAL2 produces a new type environment Γ_3 without the variables that it introduced before type checking s. We can define *infer* as follows:

$$infer(T_1 \times ... \times T_n*, i) = \begin{cases} general(T_i) & \text{if } i < n \\ general(T_n \cup \mathbf{nil}) & \text{if } i >= n \end{cases}$$

$$\begin{aligned}
general(\mathbf{false}) &= \mathbf{boolean} \\
general(\mathbf{true}) &= \mathbf{boolean} \\
general(int) &= \mathbf{integer} \\
general(float) &= \mathbf{number} \\
general(string) &= \mathbf{string} \\
general(T_1 \cup T_2) &= general(T_1) \cup general(T_2) \\
general(S_1 \to S_2) &= general2(S_1) \to general2(S_2) \\
general(\{K_1{:}V_1, ..., K_n{:}V_n\}_{tag}) &= \{K_1{:}general(V_1), ..., K_n{:}general(V_n)\}_{tag} \\
general(\mu x.T) &= \mu x.general(T) \\
general(T) &= T
\end{aligned}$$

$$\begin{aligned}
general2(\mathbf{void}) &= \mathbf{void}\\
general2(T*) &= general(T)*\\
general2(T \times P) &= general(T) \times general2(P)\\
general2(S_1 \sqcup S_2) &= general2(S_1) \sqcup general2(S_2)
\end{aligned}$$

After assigning projection types to q and r, reading q will use the projection type π_1^x to project the type of q into the union type $\mathbf{integer} \cup \mathbf{nil}$, while reading r will use the projection type π_2^x to project the type of r into the union type $\mathbf{integer} \cup \mathbf{string}$. Now, we may want to discriminate these variables to check whether the function call returned with success.

Introducing a projection variable x in the projection environment allows our type system to discriminate projection types π_i^x. The rule T-IF3 shows the case where our type system discriminates a projection type based on the tag `nil`:

$$\text{(T-IF3)}\quad \frac{\begin{array}{c}\Gamma_1(id) = \pi_i^x \quad \Pi(x) = S\\ closeall(\Gamma_1), \Pi[x \mapsto fpt(S, \mathbf{nil}, i)] \vdash s_1, \Gamma_2\\ closeall(\Gamma_1), \Pi[x \mapsto gpt(S, \mathbf{nil}, i)] \vdash s_2, \Gamma_3\\ \Gamma_4 = openset(closeset(\Gamma_1, fav(s_1) \cup fav(s_2)), rv(s_1) \cup rv(s_2))\end{array}}{\Gamma_1, \Pi \vdash \mathbf{if}\ id\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \Gamma_4}$$

Rule T-IF3 uses the auxiliary functions *fpt* and *gpt* to filter a projection x, affecting all variables that bind to the same projection. For instance, our previous example type checks through rule T-IF3, because it uses the information provided by the projection type π_1^x, which is the type of q, to make the rule T-IF3 use the function call

$$fpt((\mathbf{integer} \times \mathbf{integer} \times \mathbf{nil}*) \sqcup (\mathbf{nil} \times \mathbf{string} \times \mathbf{nil}*), \mathbf{nil}, 1)$$

to discriminate the projection x to the single tuple $\mathbf{integer} \times \mathbf{integer} \times \mathbf{nil}*$ inside the **if** branch, and the function call

$$gpt((\mathbf{integer} \times \mathbf{integer} \times \mathbf{nil}*) \sqcup (\mathbf{nil} \times \mathbf{string} \times \mathbf{nil}*), \mathbf{nil}, 1)$$

to discriminate the projection x to the single tuple $\mathbf{nil} \times \mathbf{string} \times \mathbf{nil}*$ inside the **else** branch. Thus, reading q and r projects π_1^x to **integer** and π_2^x to **integer** inside the **if** branch, but it projects π_1^x to **nil** and π_2^x to **string** inside the **else** branch. Outside the condition, q and r use the original projection, that is, they project to **integer** ∪ **nil** and **integer** ∪ **string**, respectively.

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 [?] 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 [?], 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 [?] and LuaFileSystem [?] 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 [?], Typical [?], and Mod 11 [?]. 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.

Case study	**easy**	**poly**	**over**	**hard**	**Total**
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 **??** 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. This function 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 have parametric polymorphism.

The *over* category shows the percentage of members that we gave a static type that is not precise enough, as these members are overloaded functions that require intersection types to describe their precise static types. 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 we could not give a precise static type. For instance, the function `assert` from the Lua standard library is in this category because it can return any number of values of any type, being difficult to type this behavior. Still, we gave this function the type `(value*) -> (any*)`.

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 [?]. 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 ?? summarizes the evaluation results for the Lua Standard Library (version 5.3).

Module	**easy**	**poly**	**over**	**hard**	**Total**
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

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 system 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 [?] in the form of *coroutines* [?], and effect systems [?] 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 **??** summarizes the evaluation results for MD5.

Module	**easy**	**poly**	**over**	**hard**	**Total**
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 documentation 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 [?]. 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 ?? summarizes the evaluation results for LuaSocket.

Module	**easy**	**poly**	**over**	**hard**	**Total**
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 [?].

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 **??** summarizes the evaluation results for LuaFileSystem.

Module	**easy**	**poly**	**over**	**hard**	**Total**
lfs	89%	0%	11%	0%	19

Table 5.5: 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.

5.5 HTTP Digest

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

Module	**easy**	**poly**	**over**	**hard**	**Total**
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 **??** summarizes the evaluation results for Typical.

Module	**easy**	**poly**	**over**	**hard**	**Total**
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 **??** summarizes the evaluation results for Typical.

Module	**easy**	**poly**	**over**	**hard**	**Total**
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 ?? summarizes its evaluation results.

Module	**easy**	**poly**	**over**	**hard**	**Total**
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 [?, ?] 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 **??**. 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`.

6
Related Work

In this chapter we review related work, and we split it into two sections: in the first section we review other Lua projects, while in the second section we review other projects that are not related to Lua.

6.1 Other Lua projects

Metalua [?] is a Lua compiler that supports compile-time metaprogramming (CTMP). CTMP is a kind of macro system that allows the programmers to interact with the compiler [?]. Metalua extends Lua 5.1 syntax to include its macro system, and allows programmers to define their own syntax. Metalua can provide syntactical support for several object-oriented styles, and can also provide syntax for turning simple type annotations into run-time assertions.

MoonScript [?] is a programming language that supports class-based object-oriented programming. MoonScript compiles to idiomatic Lua code, but it does not perform compile-time type checking.

LuaInspect [?] is a tool that uses MetaLua to perform some code analysis. For instance, it flags unknown global variables and table fields, it checks the number of function arguments against signatures, and it infers function return values. However, it does not try to analyze object-oriented code and it does not perform compile-time type checking.

Tidal Lock [?] is a prototype of another optional type system for Lua, which is written in Metalua. Tidal Lock covers a little subset of Lua. Statements include declaration of local variables, multiple assignment, function application, and the return statement. This means that Tidal Lock does not include any control-flow statement. Expressions include primitive literals, table indexing, function application, function declaration, and the table constructor, but they do not include binary operations.

A remarkable feature of Tidal Lock is the refinement of table types. This feature inspired us to also include it in Typed Lua, but in a simpler way and with different formalization.

The table type from Tidal Lock can only represent records, that is, it cannot describe hash tables and arrays yet, though we can refine them. Tidal Lock also includes field types to describe the type of the fields of a table type. The field types describe if a table field is mutable or immutable in a table type. Field types are the feature that allow the refinement of table types in Tidal Lock.

Tidal Lock is also a structural type system that relies on subtyping and local type inference. However, it does not support union types, recursive types, and variadic types. It also does not type any object-oriented idiom.

Sol [?] is an experimental optional type system for Lua. Its type system is similar to ours, as it includes literal types, union types, and function types that handle variadic functions. However, it does not handle the refinement of tables and it includes different types for tables. Sol types tables as lists, maps, and objects. Its object types handle a specific object-oriented idiom that Sol introduces.

Lua Analyzer [?] is an optional type system for Lua that is specially designed to work in the Löve Studio, an IDE for game developing using the Löve framework. It works in Lua 5.1 only, and uses type annotations inside comments. It is unsound by design because its dynamic type is both top and bottom in the subtyping relation.

Lua Analyzer shares some features with Typed Lua, and also has some interesting features that we do not have in Typed Lua. It has similar rules for handling the `or` idiom and discriminating union types inside conditions. However, these rules are limited to the `nil` tag only. It also includes different types for typing tables. It includes regular record types that maps names to types, array types, and map types. Even though it does not support the refinement of tables, it allows the definition of nominal table types that simulate classes. This system allows it to type check custom class systems, which are common in Lua. Function types also support multiple return values and variadic functions, but they do not support overloading the return type. Recently, it included experimental support for type aliases and generics.

Luacheck [?] is a tool that performs static analysis on Lua code. It can flag access to undeclared globals and unused local variables, but it does not perform static type checking.

Ravi [?] is an experimental Lua dialect. Ravi introduces optional static typing for Lua to improve run-time performance. To do that, Ravi extends the Lua Virtual Machine to include new operations that take into account static type information. Currently, Ravi extends the Lua Virtual Machine to support few types: `integer`, `number`, arrays of integers, and arrays of numbers.

6.2 Other projects

Typed Racket [?] is a statically typed version of the Racket language, which is a Scheme dialect. The main purpose of Typed Racket is to allow programmers to combine untyped modules, which are written in Racket, with typed modules, which are written in Typed Racket. It also uses local type inference to deduce the type of unannotated expressions.

The main feature of Typed Racket's type system is *occurrence typing* [?]. It is a novel way to use type predicates in control flow statements to refine union types. Occurrence typing is not sound in the presence of mutation. As these kinds of checks are common in other languages, related systems have appeared [?, ?, ?].

The type system of Typed Racket also includes function types, recursive types, and structure types. Its function types also handle multiple return values, and there is also a way to describe function types that have optional arguments. Its structure types are similar to our interfaces, as they describe record types. The type system is also structural and based on subtyping. It also includes the dynamic type `Any`, which is the top type in the system. Typed Racket also supports polymorphic functions and data structures.

Typed Clojure [?] is an optional type system for Clojure. Although Clojure is a Lisp dialect that runs on the Java Virtual Machine, Common Language Runtime, and JavaScript, Typed Clojure runs only on the Java Virtual Machine. Perhaps, this restriction pushed Typed Clojure to support Java classes and some Java types such as `Long`, `Double`, and `String`. Typed Clojure also provides optional type annotations and uses local type inference to deduce the type of unannotated expressions. It also assigns the type `Any` to unannotated function parameters, which is the top type in the type system.

The type system of Typed Clojure includes polymorphic function types, union types, intersection types, lists, vectors, maps, sets, and recursive types. Function types can also have rest parameters, which are similar to our variadic types, but can only appear on the input parameter of function types. In fact, its function types cannot return multiple results. It also uses occurrence typing to allow control flow statements to refine union types. The type system is also structural and based on subtyping.

Dart [?] is a new class-based object-oriented programming language. It includes optional type annotations and compiles to JavaScript. The type system of Dart is nominal and includes base types, function types, lists, and maps. It also supports generics, and the programmer can define generic functions, lists, and maps. Unlike Typed Lua, Dart is unsound by design.

Even though Dart has optional typing and static types by default do not affect run-time semantics, it has an execution mode that affects run-time. The *checked mode* inserts run-time assertions that verifies whether static types match run-time tags. The *production mode* is the default execution mode that does not include any assertions.

TypeScript [?] is a JavaScript extension that includes optional type annotations and class-based object-oriented programming. It also uses local type inference to deduce the type of unannotated expressions. The type system of TypeScript is structural, based on subtyping, and supports generics. It includes the dynamic type, primitive types, union types, function types, array types, tuple types, recursive types, and object types. Unlike Typed Lua, TypeScript uses arrays to represent variadic functions and multiple return values.

Even though TypeScript is unsound by design, Bierman et al. [?] shows how to make TypeScript sound. They use a reduced core of TypeScript to formalize a sound type system for TypeScript, but also to formalize its current unsound type system.

TeJaS [?] is a framework for the construction of different type systems for JavaScript. The authors created a base type system for JavaScript with extensible typing rules that allow the experimentation of different static analysis. They used TeJaS to create a type system that simulates the type system of TypeScript.

Politz et al. [?] proposes semantics and types for objects with first-class member names, a well-known feature from scripting languages. Their type system uses string patterns to describe the members of an object, and define a complex subtyping relation to validate these patterns. They also provide an implementation of their system to JavaScript.

Gradualtalk [?] is a Smalltalk dialect that supports gradual typing. The type system combines nominal and structural typing. It includes function types, union types, structural types, nominal types, a self type, and parametric polymorphism. The type system also relies on subtyping and consistent-subtyping.

Gradualtalk inserts run-time checks that ensure dynamically typed code does not violate statically typed code. Allende et al. [?] perform a careful evaluation about cast insertion in Gradualtalk. They report that usually cast insertions impact on execution performance, so Gradualtalk also has an option that allows programmers to turn them off, downgrading Gradualtalk to an optional type system.

Reticulated Python [?] is a Python compiler that supports gradual typing. The type system is structural and based on subtyping. It includes base types, the dynamic type, list types, dictionary types, tuple types, function types, set types, object types, class types, and recursive types. It includes class and object types to differentiate the type of class declarations and instances, respectively. It also uses local type inference. Besides static type checking, Reticulated Python also introduces three different approaches for inserting run-time assertions.

Mypy [?] is an optional type system for Python. The type system of mypy is similar to the type system of Reticulated Python, but mypy does not insert run-time checks and it has parametric polymorphism. In contrast, Reticulated Python can type variadic functions, but mypy cannot. Recently, Guido van Rossum, Python's author, proposed a standard syntax for type annotations in Python [?] that is extremely inspired by mypy [?]. The main goal of this proposal is to make easier building static analysis tools for Python. Typing [?] is a tool that is being developed to implement this proposal.

Hack [?] is a new programming language that runs on the Hip Hop Virtual Machine (HHVM). The HHVM is a virtual machine that executes Hack and PHP programs. We can view Hack as an extension to PHP that combines static and dynamic typing. The type system of Hack includes generics, nullable types, collections, and function types.

The Ruby Type Checker [?] is a library that performs type checking during run-time. The library provides type annotations that the programmer can use on classes and methods. Its type system includes nominal types, union types, intersection types, method types, parametric polymorphism, and type casts.

Grace [?] is an object-oriented language with optional typing. Grace is not a dynamically typed language that has been extended with an optional type system, but a language that has been designed from scratch to have both static and dynamic typing. Homer et al. [?] 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.

7
Conclusions

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

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

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

A key feature in optional type systems is usability. This means that optional type systems should not change the idioms that programmers are already familiar with. Instead, optional type systems should fit existing idioms to statically type check them. Designing a too simple type system can overload programmers by forcing them to change the way they program in the language to fit the type system, while designing a too complex type system can overload programmers with types and error messages that are hard to understand, even if type inference removes the necessity of annotating the program with these complex types. The most challenging aspect of designing optional type systems is to find the right amount of complexity for a type system that feels natural to the programmers.

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

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

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

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

A Glossary

bottom type It is a type that is subtype of all types.

closed table type It is the type of table annotations in Typed Lua. A closed table type does not provide any guarantees about keys with types not listed in the table type. It also does not allow table refinement to add fields to the table type.

coercion It is a relation that allows converting values from one type to values of another type without error.

consistency Gradual typing uses the consistency relation to check the interaction among the dynamic type and other types. This relation allows us combining dynamic and static typing in the same language, but still catching static type errors. The consistency relation is reflexive and symmetric, but it is not transitive to prevent that casts from a static type to the dynamic type can be combined with casts from the dynamic type to another static type.

consistent-subtyping It is a relation that combines consistency and subtyping, allowing the definition of gradual type systems for object-oriented languages. Like the consistency relation, it is reflexive and symmetric, but it is not transitive.

contravariant Subtyping is contravariant when it reverses the subtyping order, that is, a subtyping rule is contravariant when it orders types from more generic types (supertypes) to more specific types (subtypes).

covariant Subtyping is covariant when it preserves the subtyping order, that is, a subtyping rule is covariant when it orders types from more specific types (subtypes) to more general types (supertypes).

depth subtyping It allows the supertype to vary the type of individual fields in the subtype.

dynamic type It is a type that allows combining dynamic and static typing in the same code. It is neither the bottom nor the top type in the subtyping relation, but a subtype only of itself. Gradual typing uses the dynamic type along with the consistency relation to identify the parts of the code where run-time casts should be inserted to prevent that dynamically typed code violates statically typed code.

filter type It is a type that allows Typed Lua to discriminate the type of local variables inside control flow statements, as these variables are bound to unions of first-level types.

fixed table type It is the type of classes in Typed Lua. It is a table type that does not allow width subtyping to make single inheritance safe. It also does not allow table refinement to add fields to the table type.

flow typing It is a combination of static typing and flow analysis to allow variables to have different types at different parts of the program.

free assigned variable It is a free variable that appears in an assignment.

gradual type system It is a type system that uses the consistency relation instead of type equality to perform static type checking.

gradual typing It is an approach that uses a gradual type system to allow static and dynamic typing in the same code, but inserting run-time checks between statically typed and dynamically typed code. These run-time checks ensure that dynamically typed code does not violate statically typed code. More precisely, gradual typing allows programmers to change between dynamic and static typing without changing the dynamic or the static behavior of the program.

invariant Subtyping is invariant when it does not allow ordering types, that is, it is a way to define type equality through subtyping.

metatable It is a Lua table that allows changing the behavior of other tables it is attached to.

nominal type system It is a type system that uses the name of the types to check the compatibility among them.

open table type It is the type of the tables with keys that do not inhabit one of the table's key types, and have at least one alias. It also allows table refinement to add fields to the table type.

optional type system It is a type system that allows combining static and dynamic typing in the same language, but without affecting the run-time semantics. This means that the programmer has the option to use or not use its static analysis to check for static type errors, as these errors will be caught by the run-time semantics anyway.

projection environment It is an environment that Typed Lua uses to handle unions of second-level types that are bound to projection types.

projection type It is a type that allows Typed Lua to discriminate the type of local variables that have a dependency relation, as they project an union of second-level types into unions of first-level types. These unions of second-level types can describe the type of several local variables. A projection type uses an index to define which components of a union of second-level types should be used to project the union of first-level types of a local variable. When programmers discriminate a projection type, they are discriminating the union of second-level types that is bound to this projection type, affecting all variables that bind to the same projection type, which also bind to the same union of second-level types.

prototype object It is an object that works like a class, that is, it is an object from which other objects inherit its attributes.

self-like delegation It is a technique to implement inheritance in dynamically typed languages through prototype objects. In this technique, when an object tries to access an attribute that is not present, it looks for this attribute in its parent object.

sound type system It is a type system that does not type check all programs that contain a type error.

structural type system It is a type system that uses the structure of types to check the compatibility among them.

table refinement It is an operation from Typed Lua that allows programmers to change a table type to include new fields or to specialize existing fields.

top type It is a type that is supertype of all types.

type environment It is an environment that Typed Lua uses to assign variable names to first-level types.

type tag In dynamically typed languages, a type tag describes the type of a value during run-time.

unique table type It is the table type that describes the type of a table constructor. It is the type of the tables with keys that do not inhabit one of the table's key types, and that does not have any alias. It also allows table refinement to add fields to the table type.

unsound type system It is a type system that type checks certain programs that contain type errors.

userdata It is a Lua data type that allows Lua variables to hold values from applications or libraries that are written in C.

vararg expression It is a Lua expression that can result in an arbitrary number of values.

variadic function It is a Lua function that can receive an arbitrary number of arguments.

variance It is the way types are ordered.

width subtyping It allows the subtype to have fields that do not exist in the supertype.

B
The syntax of Typed Lua

This appendix presents the complete syntax of Typed Lua.

chunk ::= *block*

block ::= {*stat*} [*retstat*]

stat ::= ‘`;`’
| *varlist* ‘`=`’ *explist*
| *functioncall*
| *label*
| **break**
| **goto** *Name*
| **do** *block* **end**
| **while** *exp* **do** *block* **end**
| **repeat** *block* **until** *exp*
| **if** *exp* **then** *block* {**elseif** *exp* **then** *block*} [**else** *block*] **end**
| **for** *Name* ‘`=`’ *exp* ‘`,`’ *exp* [‘`,`’ *exp*] **do** *block* **end**
| **for** *namelist* **in** *explist* **do** *block* **end**
| [**const**] **function** *funcname funcbody*
| **local function** *Name funcbody*
| **local** *namelist* [‘`=`’ *explist*]
| [**local**] **typealias** *Name* ‘`=`’ *type*
| [**local**] **interface** *typedec*

retstat ::= **return** [*explist*] [‘`;`’]

label ::= ‘`::`’ *Name* ‘`::`’

funcname ::= *Name* {‘`.`’ *Name*} [‘`:`’ *Name*]

varlist ::= [**const**] *var* {‘`,`’ [**const**] *var*}

var ::= *Name* | *prefixexp* ‘`[`’ *exp* ‘`]`’ | *prefixexp* ‘`.`’ *Name*

namelist ::= *Name* [‘`:`’ *type*] {‘`,`’ *Name* [‘`:`’ *type*]}

```
         explist ::= exp {‘,’ exp}
             exp ::= nil | false | true | Number | String | ‘...’ | functiondef
                   | prefixexp | tableconstructor | exp binop exp | unop exp
       prefixexp ::= var | functioncall | ‘(’ exp ‘)’
    functioncall ::= prefixexp args | prefixexp ‘:’ Name args
            args ::= ‘(’ [explist] ‘)’ | tableconstructor | String
     functiondef ::= function funcbody
        funcbody ::= ‘(’ [parlist] ‘)’ [‘:’ rettype] block end
         parlist ::= namelist [‘,’ ‘...’ [‘:’ type]] | ‘...’ [‘:’ type]
tableconstructor ::= ‘{’ [fieldlist] ‘}’
       fieldlist ::= [const] field {fieldsep [const] field} [fieldsep]
           field ::= ‘[’ exp ‘]’ ‘=’ exp | Name ‘=’ exp | exp
        fieldsep ::= ‘,’ | ‘;’
           binop ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘//’ | ‘^’ | ‘%’
                   | ‘&’ | ‘~’ | ‘|’ | ‘>>’ | ‘<<’ | ‘..’
                   | ‘<’ | ‘<=’ | ‘>’ | ‘>=’ | ‘==’ | ‘~=’
                   | and | or
            unop ::= ‘-’ | not | ‘#’ | ‘~’
         typedec ::= Name {decitem} end
         decitem ::= idlist ‘:’ idtype
          idtype ::= type | methodtype
          idlist ::= id {‘,’ id}
              id ::= [const] Name
            type ::= primarytype [‘?’]
     primarytype ::= literaltype | basetype | nil | value | any | self | Name
                   | functiontype | tabletype | primarytype ‘|’ primarytype
     literaltype ::= false | true | Int | Float | String
        basetype ::= boolean | integer | number | string
    functiontype ::= tupletype ‘->’ rettype
       tupletype ::= ‘(’ [typelist] ‘)’
        typelist ::= type {‘,’ type} [‘*’]
         rettype ::= type | uniontuple [‘?’]
      uniontuple ::= tupletype | uniontuple ‘|’ uniontuple
```

tabletype ::= ‘`{`’ [*tabletypebody*] ‘`}`’

tabletypebody ::= *maptype* | *recordtype*

maptype ::= [*keytype* ‘`:`’] *type*

keytype ::= *basetype* | **value**

recordtype ::= *recordfield* {‘`,`’ *recordfield*} [‘`,`’ *type*]

recordfield ::= [**const**] *literaltype* ‘`:`’ *type*

methodtype ::= *tupletype* ‘`=>`’ *rettype*

C
The type system of Typed Lua

This appendix presents the complete type system of Typed Lua.

C.1 Subtyping rules

(S-LITERAL)

$$\Sigma \vdash L <: L$$

(S-FALSE)

$$\Sigma \vdash \textbf{false} <: \textbf{boolean}$$

(S-TRUE)

$$\Sigma \vdash \textbf{true} <: \textbf{boolean}$$

(S-STRING)

$$\Sigma \vdash \textit{string} <: \textbf{string}$$

(S-INT1)

$$\Sigma \vdash \textit{int} <: \textbf{integer}$$

(S-INT2)

$$\Sigma \vdash \textit{int} <: \textbf{number}$$

(S-FLOAT)

$$\Sigma \vdash \textit{float} <: \textbf{number}$$

(S-BASE)

$$\Sigma \vdash B <: B$$

(S-INTEGER)

$$\Sigma \vdash \textbf{integer} <: \textbf{number}$$

(S-NIL)

$$\Sigma \vdash \textbf{nil} <: \textbf{nil}$$

(S-VALUE)

$$\Sigma \vdash T <: \textbf{value}$$

(S-ANY)

$$\Sigma \vdash \textbf{any} <: \textbf{any}$$

(S-SELF)

$$\Sigma \vdash \textbf{self} <: \textbf{self}$$

(S-UNION1)

$$\frac{\Sigma \vdash T_1 <: T \quad \Sigma \vdash T_2 <: T}{\Sigma \vdash T_1 \cup T_2 <: T}$$

(S-UNION2)

$$\frac{\Sigma \vdash T <: T_1}{\Sigma \vdash T <: T_1 \cup T_2}$$

(S-UNION3)

$$\frac{\Sigma \vdash T <: T_2}{\Sigma \vdash T <: T_1 \cup T_2}$$

(S-FUNCTION)

$$\frac{\Sigma \vdash S_3 <: S_1 \quad \Sigma \vdash S_2 <: S_4}{\Sigma \vdash S_1 \rightarrow S_2 <: S_3 \rightarrow S_4}$$

(S-VOID1)

$$\Sigma \vdash \textbf{void} <: \textbf{void}$$

(S-VOID2)

$$\Sigma \vdash \textbf{void} <: T*$$

(S-PAIR)

$$\frac{\Sigma \vdash T_1 <: T_2 \quad \Sigma \vdash P_1 <: P_2}{\Sigma \vdash T_1 \times P_1 <: T_2 \times P_2}$$

(S-VARARG1)

$$\Sigma \vdash \textbf{nil}* <: \textbf{void}$$

(S-VARARG2)

$$\frac{\Sigma \vdash T_1 \cup \textbf{nil} <: T_2 \cup \textbf{nil}}{\Sigma \vdash T_1* <: T_2*}$$

(S-VARARG3)

$$\frac{\Sigma \vdash T_1 \cup \textbf{nil} <: T_2}{\Sigma \vdash T_1* <: T_2 \times \textbf{void}}$$

(S-VARARG4)

$$\frac{\Sigma \vdash T_1 <: T_2 \cup \textbf{nil}}{\Sigma \vdash T_1 \times \textbf{void} <: T_2*}$$

(S-VARARG5)

$$\frac{\Sigma \vdash T_1{*} <: T_2 \times \mathbf{void} \quad \Sigma \vdash T_1{*} <: P_2}{\Sigma \vdash T_1{*} <: T_2 \times P_2}$$

(S-VARARG6)

$$\frac{\Sigma \vdash T_1 \times \mathbf{void} <: T_2{*} \quad \Sigma \vdash P_1 <: T_2{*}}{\Sigma \vdash T_1 \times P_1 <: T_2{*}}$$

(S-UNION4)

$$\frac{\Sigma \vdash S_1 <: S \quad \Sigma \vdash S_2 <: S}{\Sigma \vdash S_1 \sqcup S_2 <: S}$$

(S-UNION5)

$$\frac{\Sigma \vdash S <: S_1}{\Sigma \vdash S <: S_1 \sqcup S_2}$$

(S-UNION6)

$$\frac{\Sigma \vdash S <: S_2}{\Sigma \vdash S <: S_1 \sqcup S_2}$$

(S-TABLE1)

$$\frac{\forall i \in 1..n\ \exists j \in 1..m \quad \Sigma \vdash K_j <: K'_i \quad \Sigma \vdash K'_i <: K_j \quad \Sigma \vdash V_j <:_c V'_i}{\Sigma \vdash \{K_1{:}V_1, ..., K_m{:}V_m\}_{fixed|closed} <: \{K'_1{:}V'_1, ..., K'_n{:}V'_n\}_{closed}}\ m \geq n$$

(S-TABLE2)

$$\frac{\begin{array}{c}\forall i \in 1..m\ \forall j \in 1..n\ \Sigma \vdash K_i <: K'_j \rightarrow \Sigma \vdash V_i <:_u V'_j \\ \forall j \in 1..n \quad \nexists i \in 1..m\ \Sigma \vdash K_i <: K'_j \rightarrow \Sigma \vdash \mathbf{nil} <:_o V'_j\end{array}}{\Sigma \vdash \{K_1{:}V_1, ..., K_m{:}V_m\}_{unique} <: \{K'_1{:}V'_1, ..., K'_n{:}V'_n\}_{closed}}$$

(S-TABLE3)

$$\frac{\begin{array}{c}\forall i \in 1..m \\ \exists j \in 1..n\ \Sigma \vdash K_i <: K'_j \land \Sigma \vdash V_i <:_u V'_j \\ \forall j \in 1..n \quad \nexists i \in 1..m\ \Sigma \vdash K_i <: K'_j \rightarrow \Sigma \vdash \mathbf{nil} <:_o V'_j\end{array}}{\Sigma \vdash \{K_1{:}V_1, ..., K_m{:}V_m\}_{unique} <: \{K'_1{:}V'_1, ..., K'_n{:}V'_n\}_{unique|open|fixed}}$$

(S-TABLE4)

$$\frac{\begin{array}{c}\forall i \in 1..m\ \forall j \in 1..n\ \Sigma \vdash K_i <: K'_j \rightarrow \Sigma \vdash V_i <:_c V'_j \\ \forall j \in 1..n \quad \nexists i \in 1..m\ \Sigma \vdash K_i <: K'_j \rightarrow \Sigma \vdash \mathbf{nil} <:_o V'_j\end{array}}{\Sigma \vdash \{K_1{:}V_1, ..., K_m{:}V_m\}_{open} <: \{K'_1{:}V'_1, ..., K'_n{:}V'_n\}_{closed}}$$

(S-TABLE5)

$$\frac{\begin{array}{c}\forall i \in 1..m \\ \exists j \in 1..n \ \Sigma \vdash K_i <: K'_j \wedge \Sigma \vdash V_i <:_c V'_j \\ \forall j \in 1..n \quad \nexists i \in 1..m \ \Sigma \vdash K_i <: K'_j \rightarrow \Sigma \vdash \mathbf{nil} <:_o V'_j\end{array}}{\Sigma \vdash \{K_1{:}V_1, ..., K_m{:}V_m\}_{open} <: \{K'_1{:}V'_1, ..., K'_n{:}V'_n\}_{open|fixed}}$$

(S-TABLE6)

$$\frac{\forall i \in 1..n \ \exists j \in 1..n \quad \Sigma \vdash K_j <: K'_i \quad \Sigma \vdash K'_i <: K_j \quad \Sigma \vdash V_j <:_c V'_i}{\Sigma \vdash \{K_1{:}V_1, ..., K_n{:}V_n\}_{fixed} <: \{K'_1{:}V'_1, ..., K'_n{:}V'_n\}_{fixed}}$$

(S-FIELD1)

$$\frac{\Sigma \vdash V_1 <: V_2 \quad \Sigma \vdash V_2 <: V_1}{\Sigma \vdash V_1 <:_c V_2}$$

(S-FIELD2)

$$\frac{\Sigma \vdash V_1 <: V_2}{\Sigma \vdash \mathbf{const}\ V_1 <:_c \mathbf{const}\ V_2}$$

(S-FIELD3)

$$\frac{\Sigma \vdash V_1 <: V_2}{\Sigma \vdash V_1 <:_c \mathbf{const}\ V_2}$$

(S-FIELD4)

$$\frac{\Sigma \vdash V_1 <: V_2}{\Sigma \vdash V_1 <:_u V_2}$$

(S-FIELD5)

$$\frac{\Sigma \vdash V_1 <: V_2}{\Sigma \vdash \mathbf{const}\ V_1 <:_u \mathbf{const}\ V_2}$$

(S-FIELD6)

$$\frac{\Sigma \vdash V_1 <: V_2}{\Sigma \vdash V_1 <:_u \mathbf{const}\ V_2}$$

(S-FIELD7)

$$\frac{\Sigma \vdash \mathbf{nil} <: V}{\Sigma \vdash \mathbf{nil} <:_o V}$$

(S-FIELD8)

$$\frac{\Sigma \vdash \mathbf{nil} <: V}{\Sigma \vdash \mathbf{nil} <:_o \mathbf{const}\ V}$$

(S-AMBER)

$$\frac{\Sigma[x_1 <: x_2] \vdash T_1 <: T_2}{\Sigma \vdash \mu x_1.T_1 <: \mu x_2.T_2}$$

(S-ASSUMPTION)

$$\frac{x_1 <: x_2 \in \Sigma}{\Sigma \vdash x_1 <: x_2}$$

(S-UNFOLDR)

$$\frac{\Sigma \vdash T_1 <: [x \mapsto \mu x.T_2]T_2}{\Sigma \vdash T_1 <: \mu x.T_2}$$

(S-UNFOLDL)

$$\frac{\Sigma \vdash [x \mapsto \mu x.T_1]T_1 <: T_2}{\Sigma \vdash \mu x.T_1 <: T_2}$$

(S-FILTER)

$$\Sigma \vdash \phi(T_1, T_2) <: \phi(T_1, T_2)$$

(S-PROJECTION)

$$\Sigma \vdash \pi_i^x <: \pi_i^x$$

(C-ANY1)

$$\Sigma \vdash T \lesssim \textbf{any}$$

(C-ANY2)

$$\Sigma \vdash \textbf{any} \lesssim T$$

C.2 Typing rules

(T-SKIP)

$$\Gamma_1, \Pi \vdash \textbf{skip}, \Gamma_1$$

(T-SEQ)

$$\frac{\Gamma_1, \Pi \vdash s_1, \Gamma_2 \quad \Gamma_2, \Pi \vdash s_2, \Gamma_3}{\Gamma_1, \Pi \vdash s_1 \ ; \ s_2, \Gamma_3}$$

(T-ASSIGNMENT)

$$\frac{\Gamma_1, \Pi \vdash el : S_1, \Gamma_2 \quad \Gamma_2, \Pi \vdash \bar{l} : S_2, \Gamma_3 \quad S_1 \lesssim S_2}{\Gamma_1, \Pi \vdash \bar{l} = el, \Gamma_3}$$

(T-METHOD1)

$$\frac{\begin{array}{c} \Gamma_1(id_1) = T_s \quad T_s = \{K_1{:}V_1, ..., K_n{:}V_n\}_{unique|open} \\ \Gamma_1, \Pi \vdash id_2 : L, \Gamma_2 \quad \nexists i \in 1..n \ L \lesssim K_i \\ crall(\Gamma_1[self \mapsto \textbf{self}, \overline{id} \mapsto \overline{T}, \sigma \mapsto T_s]), \Pi[\rho \mapsto S] \vdash s, \Gamma_3 \\ T_o = \{K_1{:}V_1, ..., K_n{:}V_n, L{:}\textbf{const self} \times \overline{T} \times \textbf{void} \rightarrow S\}_{unique|open} \\ \Gamma_4 = openset(crset(\Gamma_1[id_1 \mapsto T_o], fav(\textbf{fun}\ (\overline{id{:}T}){:}S\ s)), rv(\textbf{fun}\ (\overline{id{:}T}){:}S\ s)) \end{array}}{\Gamma_1, \Pi \vdash \textbf{fun}\ id_1{:}id_2\ (\overline{id{:}T}){:}S\ s, \Gamma_4}$$

(T-METHOD2)

$$\frac{\begin{array}{c}\Gamma_1(id_1) = T_s \quad T_s = \{K_1{:}V_1, ..., K_n{:}V_n\}_{unique|open} \\ \Gamma_1, \Pi \vdash id_2 : L, \Gamma_2 \quad \nexists i \in 1..n\ L \lesssim K_i \\ crall(\Gamma_1[self \mapsto \mathbf{self}, \overline{id} \mapsto \overline{T}, ... \mapsto T, \sigma \mapsto T_s]), \Pi[\rho \mapsto S] \vdash s, \Gamma_3 \\ T_o = \{K_1{:}V_1, ..., K_n{:}V_n, L{:}\mathbf{const\ self} \times \overline{T} \times T{*} \to S\}_{unique|open} \\ \Gamma_4 = openset(crset(\Gamma_1[id_1 \mapsto T_o], fav(\mathbf{fun}\ (\overline{id{:}T}){:}S\ s)), rv(\mathbf{fun}\ (\overline{id{:}T}){:}S\ s))\end{array}}{\Gamma_1, \Pi \vdash \mathbf{fun}\ id_1{:}id_2\ (\overline{id{:}T}, ...{:}T){:}S\ s, \Gamma_4}$$

(T-METHOD3)

$$\frac{\begin{array}{c}\Gamma_1(id_1) = T_s \quad T_s = \{K_1{:}V_1, ..., K_n{:}V_n\}_{unique|open} \quad \Gamma_1, \Pi \vdash id_2 : L, \Gamma_2 \\ crall(\Gamma_1[self \mapsto \mathbf{self}, \overline{id} \mapsto \overline{T}, \sigma \mapsto T_s]), \Pi[\rho \mapsto S] \vdash s, \Gamma_3 \\ \exists i \in 1..n\ L <: K_i \wedge K_i <: L \quad \mathbf{const\ self} \times \overline{T} \times \mathbf{void} \to S <: V_i \\ V_i = \mathbf{const\ self} \times \overline{T} \times \mathbf{void} \to S \\ T_o = \{K_1{:}V_1, ..., K_n{:}V_n\}_{unique|open} \\ \Gamma_4 = openset(crset(\Gamma_1[id_1 \mapsto T_o], fav(\mathbf{fun}\ (\overline{id{:}T}){:}S\ s)), rv(\mathbf{fun}\ (\overline{id{:}T}){:}S\ s))\end{array}}{\Gamma_1, \Pi \vdash \mathbf{fun}\ id_1{:}id_2\ (\overline{id{:}T}){:}S\ s, \Gamma_4}$$

(T-METHOD2)

$$\frac{\begin{array}{c}\Gamma_1(id_1) = T_s \quad T_s = \{K_1{:}V_1, ..., K_n{:}V_n\}_{unique|open} \quad \Gamma_1, \Pi \vdash id_2 : L, \Gamma_2 \\ crall(\Gamma_1[self \mapsto \mathbf{self}, \overline{id} \mapsto \overline{T}, ... \mapsto T, \sigma \mapsto T_s]), \Pi[\rho \mapsto S] \vdash s, \Gamma_3 \\ \exists i \in 1..n\ L <: K_i \wedge K_i <: L \quad \mathbf{const\ self} \times \overline{T} \times T{*} \to S <: V_i \\ V_i = \mathbf{const\ self} \times \overline{T} \times T{*} \to S \\ T_o = \{K_1{:}V_1, ..., K_n{:}V_n\}_{unique|open} \\ \Gamma_4 = openset(crset(\Gamma_1[id_1 \mapsto T_o], fav(\mathbf{fun}\ (\overline{id{:}T}){:}S\ s)), rv(\mathbf{fun}\ (\overline{id{:}T}){:}S\ s))\end{array}}{\Gamma_1, \Pi \vdash \mathbf{fun}\ id_1{:}id_2\ (\overline{id{:}T}, ...{:}T){:}S\ s, \Gamma_4}$$

(T-WHILE1)

$$\frac{\begin{array}{c}\Gamma_1, \Pi \vdash e : T, \Gamma_2 \quad closeall(\Gamma_2), \Pi \vdash s, \Gamma_3 \\ \Gamma_4 = openset(closeset(\Gamma_2, fav(s)), rv(s))\end{array}}{\Gamma_1, \Pi \vdash \mathbf{while}\ e\ \mathbf{do}\ s, \Gamma_4}$$

(T-WHILE2)

$$\frac{\begin{array}{c}\Gamma_1(id) = T_1 \cup T_2 \\ closeall(\Gamma_1[id \mapsto \phi(T_1 \cup T_2, filter(T_1 \cup T_2, \mathbf{nil}))]), \Pi \vdash s, \Gamma_2 \\ \Gamma_3 = openset(closeset(\Gamma_1[id \mapsto T_1 \cup T_2], fav(s)), rv(s))\end{array}}{\Gamma_1, \Pi \vdash \mathbf{while}\ id\ \mathbf{do}\ s, \Gamma_3}$$

(T-WHILE3)

$$\frac{\begin{array}{c}\Gamma_1(id) = \pi_i^x \quad \Pi(x) = S \\ closeall(\Gamma_1), \Pi[x \mapsto fpt(S, \mathbf{nil}, i)] \vdash s, \Gamma_2 \\ \Gamma_3 = openset(closeset(\Gamma_1, fav(s)), rv(s))\end{array}}{\Gamma_1, \Pi \vdash \mathbf{while}\ id\ \mathbf{do}\ s, \Gamma_3}$$

(T-IF1)

$$\frac{\begin{array}{c}\Gamma_1, \Pi \vdash e : T, \Gamma_2 \\ closeall(\Gamma_2), \Pi \vdash s_1 : \Gamma_3 \\ closeall(\Gamma_2), \Pi \vdash s_2 : \Gamma_4 \\ \Gamma_5 = openset(closeset(\Gamma_2, fav(s_1) \cup fav(s_2)), rv(s_1) \cup rv(s_2)))\end{array}}{\Gamma_1 \vdash \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \Gamma_5}$$

(T-IF2)

$$\frac{\begin{array}{c}\Gamma_1(id) = T_1 \cup T_2 \\ closeall(\Gamma_1[id \mapsto \phi(T_1 \cup T_2, filter(T_1 \cup T_2, \mathbf{nil}))]), \Pi \vdash s_1, \Gamma_2 \\ closeall(\Gamma_1[id \mapsto \phi(T_1 \cup T_2, \mathbf{nil})]), \Pi \vdash s_2, \Gamma_3 \\ \Gamma_4 = openset(closeset(\Gamma_1[id \mapsto T_1 \cup T_2], fav(s_1) \cup fav(s_2)), rv(s_1) \cup rv(s_2))\end{array}}{\Gamma_1, \Pi \vdash \mathbf{if}\ id\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \Gamma_4}$$

(T-IF3)

$$\frac{\begin{array}{c}\Gamma_1(id) = \pi_i^x \quad \Pi(x) = S \\ closeall(\Gamma_1), \Pi[x \mapsto fpt(S, \mathbf{nil}, i)] \vdash s_1, \Gamma_2 \\ closeall(\Gamma_1), \Pi[x \mapsto gpt(S, \mathbf{nil}, i)] \vdash s_2, \Gamma_3 \\ \Gamma_4 = openset(closeset(\Gamma_1, fav(s_1) \cup fav(s_2)), rv(s_1) \cup rv(s_2))\end{array}}{\Gamma_1, \Pi \vdash \mathbf{if}\ id\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \Gamma_4}$$

(T-IF4)

$$\frac{\begin{array}{c}\Gamma_1(id) = T_1 \cup T_2 \\ closeall(\Gamma_1[id \mapsto \phi(T_1 \cup T_2, \mathbf{boolean})]), \Pi \vdash s_1, \Gamma_2 \\ closeall(\Gamma_1[id \mapsto \phi(T_1 \cup T_2, filter(T_1 \cup T_2, \mathbf{boolean}))]), \Pi \vdash s_2, \Gamma_3 \\ \Gamma_4 = openset(closeset(\Gamma_1[id \mapsto T_1 \cup T_2], fav(s_1) \cup fav(s_2)), rv(s_1) \cup rv(s_2))\end{array}}{\Gamma_1, \Pi \vdash \mathbf{if}\ type(id) == \text{``}boolean\text{''}\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \Gamma_4}$$

(T-IF5)

$$\frac{\begin{array}{c}\Gamma_1(id) = \pi_i^x \quad \Pi(x) = S \\ \mathit{closeall}(\Gamma_1), \Pi[x \mapsto \mathit{gpt}(S, \textbf{boolean}, i)] \vdash s_1, \Gamma_2 \\ \mathit{closeall}(\Gamma_1), \Pi[x \mapsto \mathit{fpt}(S, \textbf{boolean}, i)] \vdash s_2, \Gamma_3 \\ \Gamma_4 = \mathit{openset}(\mathit{closeset}(\Gamma_1, \mathit{fav}(s_1) \cup \mathit{fav}(s_2)), \mathit{rv}(s_1) \cup \mathit{rv}(s_2))\end{array}}{\Gamma_1, \Pi \vdash \textbf{if}\ \mathit{type}(id) == \text{``}\mathit{boolean}\text{''}\ \textbf{then}\ s_1\ \textbf{else}\ s_2, \Gamma_4}$$

(T-IF6)

$$\frac{\begin{array}{c}\Gamma_1(id) = T_1 \cup T_2 \\ \mathit{closeall}(\Gamma_1[id \mapsto \phi(T_1 \cup T_2, \textbf{integer})]), \Pi \vdash s_1, \Gamma_2 \\ \mathit{closeall}(\Gamma_1[id \mapsto \phi(T_1 \cup T_2, \mathit{filter}(T_1 \cup T_2, \textbf{integer}))]), \Pi \vdash s_2, \Gamma_3 \\ \Gamma_4 = \mathit{openset}(\mathit{closeset}(\Gamma_1[id \mapsto T_1 \cup T_2], \mathit{fav}(s_1) \cup \mathit{fav}(s_2)), \mathit{rv}(s_1) \cup \mathit{rv}(s_2))\end{array}}{\Gamma_1, \Pi \vdash \textbf{if}\ \mathit{type}(id) == \text{``}\mathit{integer}\text{''}\ \textbf{then}\ s_1\ \textbf{else}\ s_2, \Gamma_4}$$

(T-IF7)

$$\frac{\begin{array}{c}\Gamma_1(id) = \pi_i^x \quad \Pi(x) = S \\ \mathit{closeall}(\Gamma_1), \Pi[x \mapsto \mathit{gpt}(S, \textbf{integer}, i)] \vdash s_1, \Gamma_2 \\ \mathit{closeall}(\Gamma_1), \Pi[x \mapsto \mathit{fpt}(S, \textbf{integer}, i)] \vdash s_2, \Gamma_3 \\ \Gamma_4 = \mathit{openset}(\mathit{closeset}(\Gamma_1, \mathit{fav}(s_1) \cup \mathit{fav}(s_2)), \mathit{rv}(s_1) \cup \mathit{rv}(s_2))\end{array}}{\Gamma_1, \Pi \vdash \textbf{if}\ \mathit{type}(id) == \text{``}\mathit{integer}\text{''}\ \textbf{then}\ s_1\ \textbf{else}\ s_2, \Gamma_4}$$

(T-IF8)

$$\frac{\begin{array}{c}\Gamma_1(id) = T_1 \cup T_2 \\ \mathit{closeall}(\Gamma_1[id \mapsto \phi(T_1 \cup T_2, \textbf{number})]), \Pi \vdash s_1, \Gamma_2 \\ \mathit{closeall}(\Gamma_1[id \mapsto \phi(T_1 \cup T_2, \mathit{filter}(T_1 \cup T_2, \textbf{number}))]), \Pi \vdash s_2, \Gamma_3 \\ \Gamma_4 = \mathit{openset}(\mathit{closeset}(\Gamma_1[id \mapsto T_1 \cup T_2], \mathit{fav}(s_1) \cup \mathit{fav}(s_2)), \mathit{rv}(s_1) \cup \mathit{rv}(s_2))\end{array}}{\Gamma_1, \Pi \vdash \textbf{if}\ \mathit{type}(id) == \text{``}\mathit{number}\text{''}\ \textbf{then}\ s_1\ \textbf{else}\ s_2, \Gamma_4}$$

(T-IF9)

$$\frac{\begin{array}{c}\Gamma_1(id) = \pi_i^x \quad \Pi(x) = S \\ \mathit{closeall}(\Gamma_1), \Pi[x \mapsto \mathit{gpt}(S, \textbf{number}, i)] \vdash s_1, \Gamma_2 \\ \mathit{closeall}(\Gamma_1), \Pi[x \mapsto \mathit{fpt}(S, \textbf{number}, i)] \vdash s_2, \Gamma_3 \\ \Gamma_4 = \mathit{openset}(\mathit{closeset}(\Gamma_1, \mathit{fav}(s_1) \cup \mathit{fav}(s_2)), \mathit{rv}(s_1) \cup \mathit{rv}(s_2))\end{array}}{\Gamma_1, \Pi \vdash \textbf{if}\ \mathit{type}(id) == \text{``}\mathit{number}\text{''}\ \textbf{then}\ s_1\ \textbf{else}\ s_2, \Gamma_4}$$

(T-IF10)

$$\frac{\begin{array}{c}\Gamma_1(id) = T_1 \cup T_2 \\ closeall(\Gamma_1[id \mapsto \phi(T_1 \cup T_2, \mathbf{string})]), \Pi \vdash s_1, \Gamma_2 \\ closeall(\Gamma_1[id \mapsto \phi(T_1 \cup T_2, filter(T_1 \cup T_2, \mathbf{string}))]), \Pi \vdash s_2, \Gamma_3 \\ \Gamma_4 = openset(closeset(\Gamma_1[id \mapsto T_1 \cup T_2], fav(s_1) \cup fav(s_2)), rv(s_1) \cup rv(s_2))\end{array}}{\Gamma_1, \Pi \vdash \mathbf{if}\ type(id) == \text{“}string\text{”}\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \Gamma_4}$$

(T-IF11)

$$\frac{\begin{array}{c}\Gamma_1(id) = \pi_i^x \quad \Pi(x) = S \\ closeall(\Gamma_1), \Pi[x \mapsto gpt(S, \mathbf{string}, i)] \vdash s_1, \Gamma_2 \\ closeall(\Gamma_1), \Pi[x \mapsto fpt(S, \mathbf{string}, i)] \vdash s_2, \Gamma_3 \\ \Gamma_4 = openset(closeset(\Gamma_1, fav(s_1) \cup fav(s_2)), rv(s_1) \cup rv(s_2))\end{array}}{\Gamma_1, \Pi \vdash \mathbf{if}\ type(id) == \text{“}string\text{”}\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \Gamma_4}$$

(T-LOCAL1)

$$\frac{\Gamma_1, \Pi \vdash el : S, \Gamma_2 \quad S \lesssim \overline{T} \times \mathbf{value}{*} \quad \Gamma_2[\overline{id} \mapsto \overline{T}], \Pi \vdash s, \Gamma_3}{\Gamma_1, \Pi \vdash \mathbf{local}\ \overline{id{:}T} = el\ \mathbf{in}\ s, \Gamma_3 - \{\overline{id}\}}$$

(T-LOCAL2)

$$\frac{\begin{array}{c}\Gamma_1, \Pi \vdash el : S_1, \Gamma_2, (x, S_2) \\ \Gamma_2[id_1 \mapsto infer(S_1, 1), ..., id_n \mapsto infer(S_1, n)], \Pi[x \mapsto S_2] \vdash s, \Gamma_3 \quad n = |\ \overline{id}\ |\end{array}}{\Gamma_1, \Pi \vdash \mathbf{local}\ \overline{id} = el\ \mathbf{in}\ s, \Gamma_3 - \{\overline{id}\}}$$

(T-LOCALREC)

$$\frac{\Gamma_1[id \mapsto T], \Pi \vdash f : T_1, \Gamma_2 \quad T_1 \lesssim T \quad \Gamma_2, \Pi \vdash s, \Gamma_3}{\Gamma_1, \Pi \vdash \mathbf{rec}\ id{:}T = f\ \mathbf{in}\ s, \Gamma_3 - \{id\}}$$

(T-RETURN)

$$\frac{\Gamma_1 \vdash el : S_1, \Gamma_2 \quad \Pi(\rho) = S_2 \quad S_1 \lesssim S_2}{\Gamma_1 \vdash \mathbf{return}\ el, \Gamma_2}$$

(T-STMAPPLY1)

$$\frac{\Gamma_1, \Pi \vdash e(el) : S, \Gamma_2}{\Gamma_1, \Pi \vdash \lfloor e(el) \rfloor_0, \Gamma_2}$$

(T-STMINVOKE1)

$$\frac{\Gamma_1, \Pi \vdash e{:}n(el) : S, \Gamma_2}{\Gamma_1, \Pi \vdash \lfloor e{:}n(el) \rfloor_0, \Gamma_2}$$

(T-NIL)

$$\Gamma_1, \Pi \vdash \textbf{nil} : \textbf{nil}, \Gamma_1$$

(T-FALSE)

$$\Gamma_1, \Pi \vdash \textbf{false} : \textbf{false}, \Gamma_1$$

(T-TRUE)

$$\Gamma_1, \Pi \vdash \textbf{true} : \textbf{true}, \Gamma_1$$

(T-INT)

$$\Gamma_1, \Pi \vdash \mathit{int} : \mathit{int}, \Gamma_1$$

(T-FLOAT)

$$\Gamma_1, \Pi \vdash \mathit{float} : \mathit{float}, \Gamma_1$$

(T-STR)

$$\Gamma_1, \Pi \vdash \mathit{string} : \mathit{string}, \Gamma_1$$

(T-IDREAD1)

$$\frac{\Gamma_1(id) = T_1 \quad T_2 = read(\Pi, T_1)}{\Gamma_1, \Pi \vdash id : close(T_2), \Gamma_1[id \mapsto open(T_1)]}$$

(T-IDREAD2)

$$\frac{\Gamma_1(id) = T_1 \quad T_2 = read(\Pi, T_1)}{\Gamma_1, \Pi \vdash id : fix(T_2), \Gamma_1[id \mapsto fix(T_1)]}$$

(T-INDEX1)

$$\frac{\begin{array}{c}\Gamma_1(id) = T \quad read(\Pi, T) = \{K_1{:}V_1, ..., K_n{:}V_n\} \\ \Gamma_1, \Pi \vdash e_2 : K, \Gamma_2 \quad \exists i \in 1..n \; K \lesssim K_i\end{array}}{\Gamma_1, \Pi \vdash id[e_2] : V_i, \Gamma_2}$$

(T-INDEX2)

$$\frac{\begin{array}{c}\Gamma_1, \Pi \vdash e_1 : \{K_1{:}V_1, ..., K_n{:}V_n\}, \Gamma_2 \\ \Gamma_2, \Pi \vdash e_2 : K, \Gamma_3 \quad \exists i \in 1..n \; K \lesssim K_i\end{array}}{\Gamma_1, \Pi \vdash e_1[e_2] : V_i, \Gamma_3}$$

(T-INDEX3)

$$\frac{\Gamma_1, \Pi \vdash e_1 : \mathbf{any}, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : t, \Gamma_3}{\Gamma_1, \Pi \vdash e_1[e_2] : \mathbf{any}, \Gamma_3}$$

(T-COERCE)

$$\frac{\Gamma_1(id) <: T \quad \Gamma_1[id \mapsto T], \Pi \vdash id : T_1, \Gamma_2}{\Gamma_1, \Pi \vdash \,<\!T\!>\, id : T_1, \Gamma_2}$$

(T-FUNCTION1)

$$\frac{\begin{array}{c} crall(\Gamma_1[\overline{id} \mapsto \overline{T}]), \Pi[\rho \mapsto S] \vdash s, \Gamma_2 \\ \Gamma_3 = openset(crset(\Gamma_1, fav(\mathbf{fun}\ (\overline{id{:}T}){:}S\ s)), rv(\mathbf{fun}\ (\overline{id{:}T}){:}S\ s)) \end{array}}{\Gamma_1, \Pi \vdash \mathbf{fun}\ (\overline{id{:}T}){:}S\ s : \overline{T} \times \mathbf{void} \rightarrow S, \Gamma_3}$$

(T-FUNCTION2)

$$\frac{\begin{array}{c} crall(\Gamma_1[\overline{id} \mapsto \overline{T}, ... \mapsto T]), \Pi[\rho \mapsto S] \vdash s, \Gamma_2 \\ \Gamma_3 = openset(crset(\Gamma_1, fav(\mathbf{fun}\ (\overline{id{:}T}){:}S\ s)), rv(\mathbf{fun}\ (\overline{id{:}T}){:}S\ s)) \end{array}}{\Gamma_1, \Pi \vdash \mathbf{fun}\ (\overline{id{:}T}, ...{:}T){:}S\ s : \overline{T} \times T* \rightarrow S, \Gamma_3}$$

(T-CONSTRUCTOR1)

$$\Gamma_1, \Pi \vdash \{\} : \{\}_{unique}, \Gamma_1$$

(T-CONSTRUCTOR2)

$$\frac{\begin{array}{c} \Gamma_1, \Pi \vdash ([e_1] = e_2)_i : (K_i, V_i), \Gamma_{i+1} \\ T = \{K_1{:}V_1, ..., K_n{:}V_n\}_{unique} \quad wf(T) \quad n = |\ \overline{[e_1] = e_2}\ | \\ \Gamma_f = merge(\Gamma_1, ..., \Gamma_{n+1}) \end{array}}{\Gamma_1, \Pi \vdash \{\ \overline{[e_1] = e_2}\ \} : T, \Gamma_f}$$

(T-CONSTRUCTOR3)

$$\frac{\begin{array}{c} \Gamma_1, \Pi \vdash ([e_1] = e_2)_i : (K_i, V_i), \Gamma_{i+1} \\ \Gamma_1, \Pi \vdash me : T_{n+1} \times ... \times T_{n+m} \times T_{n+m+1}*, \Gamma_{n+2} \\ T = \{K_1{:}V_1, ..., K_n{:}V_n, 1 : T_{n+1}, ..., m : T_{n+m}, \mathbf{integer} : T_{n+m+1} \cup \mathbf{nil}\}_{unique} \\ wf(T) \quad n = |\ \overline{[e_1] = e_2}\ | \quad \Gamma_f = merge(\Gamma_1, ..., \Gamma_{n+2}) \end{array}}{\Gamma_1, \Pi \vdash \{\ \overline{[e_1] = e_2}\ \} : T, \Gamma_f}$$

(T-FIELD1)

$$\frac{\Gamma_1, \Pi \vdash e_2 : T_2, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_1 : T_1, \Gamma_3 \quad T_1 <: \mathbf{boolean}}{\Gamma_1, \Pi \vdash [e_1] = e_2 : (T_1, \mathit{close}(T_2)), \Gamma_3}$$

(T-FIELD2)

$$\frac{\Gamma_1, \Pi \vdash e_2 : T_2, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_1 : T_1, \Gamma_3 \quad T_1 <: \mathbf{number}}{\Gamma_1, \Pi \vdash [e_1] = e_2 : (T_1, \mathit{close}(T_2)), \Gamma_3}$$

(T-FIELD3)

$$\frac{\Gamma_1, \Pi \vdash e_2 : T_2, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_1 : T_1, \Gamma_3 \quad T_1 <: \mathbf{string}}{\Gamma_1, \Pi \vdash [e_1] = e_2 : (T_1, \mathit{close}(T_2)), \Gamma_3}$$

(T-FIELD4)

$$\frac{\Gamma_1, \Pi \vdash e_2 : T_2, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_1 : T_1, \Gamma_3}{\Gamma_1, \Pi \vdash [e_1] = e_2 : (\mathbf{value}, \mathit{close}(T_2)), \Gamma_3}$$

(T-ARITH1)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T_1, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T_2, \Gamma_3 \quad T_1 <: \mathbf{integer} \quad T_2 <: \mathbf{integer}}{\Gamma_1, \Pi \vdash e_1 + e_2 : \mathbf{integer}, \Gamma_3}$$

(T-ARITH2)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T_1, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T_2, \Gamma_3 \quad T_1 <: \mathbf{integer} \quad T_2 <: \mathbf{number}}{\Gamma_1, \Pi \vdash e_1 + e_2 : \mathbf{number}, \Gamma_3}$$

(T-ARITH3)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T_1, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T_2, \Gamma_3 \quad T_1 <: \mathbf{number} \quad T_2 <: \mathbf{integer}}{\Gamma_1, \Pi \vdash e_1 + e_2 : \mathbf{number}, \Gamma_3}$$

(T-ARITH4)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T_1, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T_2, \Gamma_3 \quad T_1 <: \mathbf{number} \quad T_2 <: \mathbf{number}}{\Gamma_1, \Pi \vdash e_1 + e_2 : \mathbf{number}, \Gamma_3}$$

(T-ARITH5)

$$\frac{\Gamma_1, \Pi \vdash e_1 : \mathbf{any}, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T, \Gamma_3}{\Gamma_1, \Pi \vdash e_1 + e_2 : \mathbf{any}, \Gamma_3}$$

(T-ARITH6)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : \textbf{any}, \Gamma_3}{\Gamma_1, \Pi \vdash e_1 + e_2 : \textbf{any}, \Gamma_3}$$

(T-CONCAT1)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T_1, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T_2, \Gamma_3 \quad T_1 <: \textbf{string} \quad T_2 <: \textbf{string}}{\Gamma_1, \Pi \vdash e_1 \mathrel{..} e_2 : \textbf{string}, \Gamma_3}$$

(T-CONCAT2)

$$\frac{\Gamma_1, \Pi \vdash e_1 : \textbf{any}, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T, \Gamma_3}{\Gamma_1, \Pi \vdash e_1 \mathrel{..} e_2 : \textbf{any}, \Gamma_3}$$

(T-CONCAT3)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : \textbf{any}, \Gamma_3}{\Gamma_1, \Pi \vdash e_1 \mathrel{..} e_2 : \textbf{any}, \Gamma_3}$$

(T-EQUAL)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T_1, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T_2, \Gamma_3}{\Gamma_1, \Pi \vdash e_1 == e_2 : \textbf{boolean}, \Gamma_3}$$

(T-ORDER1)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T_1, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T_2, \Gamma_3 \quad T_1 <: \textbf{number} \quad T_2 <: \textbf{number}}{\Gamma, \Pi \vdash e_1 < e_2 : \textbf{boolean}, \Gamma_3}$$

(T-ORDER2)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T_1, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T_2, \Gamma_3 \quad T_1 <: \textbf{string} \quad T_2 <: \textbf{string}}{\Gamma_1, \Pi \vdash e_1 < e_2 : \textbf{boolean}}$$

(T-ORDER3)

$$\frac{\Gamma_1, \Pi \vdash e_1 : \textbf{any}, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T, \Gamma_3}{\Gamma_1, \Pi \vdash e_1 < e_2 : \textbf{any}, \Gamma_3}$$

(T-ORDER4)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : \textbf{any}, \Gamma_3}{\Gamma_1, \Pi \vdash e_1 < e_2 : \textbf{any}, \Gamma_3}$$

(T-BITWISE1)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T_1, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T_2, \Gamma_3 \quad T_1 <: \textbf{integer} \quad T_2 <: \textbf{integer}}{\Gamma_1, \Pi \vdash e_1 \mathbin{\&} e_2 : \textbf{integer}, \Gamma_3}$$

(T-BITWISE2)

$$\frac{\Gamma_1, \Pi \vdash e_1 : \textbf{any}, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T, \Gamma_3}{\Gamma_1, \Pi \vdash e_1 \mathbin{\&} e_2 : \textbf{any}, \Gamma_3}$$

(T-BITWISE3)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : \textbf{any}, \Gamma_3}{\Gamma_1, \Pi \vdash e_1 \mathbin{\&} e_2 : \textbf{any}, \Gamma_3}$$

(T-AND1)

$$\frac{\Gamma_1, \Pi \vdash e_1 : \textbf{nil}, \Gamma_2}{\Gamma_1, \Pi \vdash e_1 \textbf{ and } e_2 : \textbf{nil}, \Gamma_2}$$

(T-AND2)

$$\frac{\Gamma_1, \Pi \vdash e_1 : \textbf{false}, \Gamma_2}{\Gamma_1, \Pi \vdash e_1 \textbf{ and } e_2 : \textbf{false}, \Gamma_2}$$

(T-AND3)

$$\frac{\Gamma_1, \Pi \vdash e_1 : \textbf{nil} \cup \textbf{false}, \Gamma_2}{\Gamma_1, \Pi \vdash e_1 \textbf{ and } e_2 : \textbf{nil} \cup \textbf{false}, \Gamma_2}$$

(T-AND4)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T_1, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T_2, \Gamma_3 \quad \textbf{nil} \not\lesssim T_1 \quad \textbf{false} \not\lesssim T_1}{\Gamma_1, \Pi \vdash e_1 \textbf{ and } e_2 : T_2, \Gamma_3}$$

(T-AND5)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T_1, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T_2, \Gamma_3}{\Gamma_1, \Pi \vdash e_1 \textbf{ and } e_2 : T_1 \cup T_2, \Gamma_3}$$

(T-OR1)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T, \Gamma_2 \quad \textbf{nil} \not\lesssim T \quad \textbf{false} \not\lesssim T}{\Gamma_1, \Pi \vdash e_1 \textbf{ or } e_2 : T, \Gamma_2}$$

(T-OR2)

$$\frac{\Gamma_1, \Pi \vdash e_1 : \textbf{nil}, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T, \Gamma_3}{\Gamma_1, \Pi \vdash e_1 \textbf{ or } e_2 : T, \Gamma_3}$$

(T-OR3)

$$\frac{\Gamma_1, \Pi \vdash e_1 : \textbf{false}, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T, \Gamma_3}{\Gamma_1, \Pi \vdash e_1 \textbf{ or } e_2 : T, \Gamma_3}$$

(T-OR4)

$$\frac{\Gamma_1, \Pi \vdash e_1 : \textbf{nil} \cup \textbf{false}, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T, \Gamma_3}{\Gamma_1, \Pi \vdash e_1 \textbf{ or } e_2 : T, \Gamma_3}$$

(T-OR5)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T_1, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T_2, \Gamma_3}{\Gamma_1, \Pi \vdash e_1 \textbf{ or } e_2 : \mathit{filter}(\mathit{filter}(T_1, \textbf{nil}), \textbf{false}) \cup T_2, \Gamma_3}$$

(T-NOT1)

$$\frac{\Gamma_1, \Pi \vdash e : \textbf{nil}, \Gamma_2}{\Gamma_1, \Pi \vdash \textbf{not } e : \textbf{true}, \Gamma_2}$$

(T-NOT2)

$$\frac{\Gamma_1, \Pi \vdash e : \textbf{false}, \Gamma_2}{\Gamma_1, \Pi \vdash \textbf{not } e : \textbf{true}, \Gamma_2}$$

(T-NOT3)

$$\frac{\Gamma_1, \Pi \vdash e : \textbf{nil} \cup \textbf{false}, \Gamma_2}{\Gamma_1, \Pi \vdash \textbf{not } e : \textbf{true}, \Gamma_2}$$

(T-NOT4)

$$\frac{\Gamma_1, \Pi \vdash e : T \quad \textbf{nil} \not\lesssim T \quad \textbf{false} \not\lesssim T}{\Gamma_1, \Pi \vdash \textbf{not } e : \textbf{false}, \Gamma_2}$$

(T-NOT5)

$$\frac{\Gamma_1, \Pi \vdash e : T, \Gamma_2}{\Gamma_1, \Pi \vdash \textbf{not } e : \textbf{boolean}, \Gamma_2}$$

(T-LEN1)

$$\frac{\Gamma_1, \Pi \vdash e : T, \Gamma_2 \quad T <: \textbf{string}}{\Gamma_1, \Pi \vdash \#\, e : \textbf{integer}, \Gamma_2}$$

(T-LEN2)

$$\frac{\Gamma_1, \Pi \vdash e : T, \Gamma_2 \quad T <: \{\}_{\mathit{closed}}}{\Gamma_1, \Pi \vdash \#\, e : \textbf{integer}, \Gamma_2}$$

(T-LEN3)

$$\frac{\Gamma_1, \Pi \vdash e : \textbf{any}, \Gamma_2}{\Gamma_1, \Pi \vdash \#\, e : \textbf{any}, \Gamma_2}$$

(T-EXPAPPLY1)

$$\frac{\Gamma_1, \Pi \vdash e(el) : S, \Gamma_2}{\Gamma_1, \Pi \vdash \lfloor e(el) \rfloor_1 : \mathit{first}(S), \Gamma_2}$$

(T-EXPINVOKE1)

$$\frac{\Gamma_1, \Pi \vdash e{:}n(el) : S, \Gamma_2}{\Gamma_1, \Pi \vdash \lfloor e{:}n(el) \rfloor_1 : \mathit{first}(S), \Gamma_2}$$

(T-EXPDOTS)

$$\frac{\Gamma_1, \Pi \vdash ... : T*, \Gamma_2}{\Gamma_1, \Pi \vdash \lfloor ... \rfloor_1 : T \cup \textbf{nil}, \Gamma_2}$$

(T-IDWRITE)

$$\frac{\Gamma_1(id) = T_1 \quad T_2 = \mathit{write}(T_1)}{\Gamma_1, \Pi \vdash id_l : \mathit{close}(T_2), \Gamma_1[id \mapsto \mathit{close}(T_2)]}$$

(T-REFINE)

$$\frac{\begin{array}{c}\Gamma_1(id) = \{K_1{:}V_1, ..., K_n{:}V_n\}_{open|unique} \\ \Gamma_1, \Pi \vdash k : K, \Gamma_2 \quad \nexists i \in 1..n\ K \lesssim K_i \quad V = \mathit{close}(T)\end{array}}{\Gamma_1, \Pi \vdash id[k]{<}T{>} : V, \Gamma_2[id \mapsto \{K_1{:}V_1, ..., K_n{:}V_n, K{:}V\}_{open|unique}]}$$

(T-LHSLIST)

$$\frac{\Gamma_1, \Pi \vdash l_i : T_i, \Gamma_{i+1} \quad \Gamma_f = \mathit{merge}(\Gamma_1, ..., \Gamma_{n+1}) \quad n = |\ \bar{l}\ |}{\Gamma_1, \Pi \vdash \bar{l} : T_1 \times ... \times T_n \times \textbf{value}*, \Gamma_f}$$

(T-EXPLIST1)

$$\frac{\Gamma_1, \Pi \vdash e_i : T_i, \Gamma_{i+1} \quad \Gamma_f = \mathit{merge}(\Gamma_1, ..., \Gamma_{n+1}) \quad n = |\ \bar{e}\ |}{\Gamma_1, \Pi \vdash \bar{e} : T_1 \times ... \times T_n \times \textbf{nil}*, \Gamma_f}$$

(T-EXPLIST2)

$$\frac{\begin{array}{c}\Gamma_1, \Pi \vdash e_i : T_i, \Gamma_{i+1} \quad \Gamma_1, \Pi \vdash me : T_{n+1} \times ... \times T_{n+m} \times \textbf{void}, \Gamma_{n+2} \\ \Gamma_f = \mathit{merge}(\Gamma_1, ..., \Gamma_{n+2}) \quad n = |\ \bar{e}\ |\end{array}}{\Gamma_1, \Pi \vdash \bar{e}, me : T_1 \times ... \times T_{n+m} \times \textbf{nil}*, \Gamma_f}$$

(T-EXPLIST3)

$$\frac{\begin{array}{c}\Gamma_1, \Pi \vdash e_i : T_i, \Gamma_{i+1} \quad \Gamma_1, \Pi \vdash me : T_{n+1} \times ... \times T_{n+m}*, \Gamma_{n+2} \\ \Gamma_f = merge(\Gamma_1, ..., \Gamma_{n+2}) \quad n = \mid \overline{e} \mid\end{array}}{\Gamma_1, \Pi \vdash \overline{e}, me : T_1 \times ... \times T_{n+m}*, \Gamma_f}$$

(T-EXPLIST4)

$$\frac{\begin{array}{c}\Gamma_1, \Pi \vdash e_i : T_i, \Gamma_{i+1} \quad \Gamma_1, \Pi \vdash me : S, \Gamma_{n+2} \\ S = T_{n+1} \times ... \times T_{n+m} \times \textbf{void} \sqcup T'_{n+1} \times ... \times T'_{n+m} \times \textbf{void} \\ \Gamma_f = merge(\Gamma_1, ..., \Gamma_{n+2}) \quad n = \mid \overline{e} \mid\end{array}}{\Gamma_1, \Pi \vdash \overline{e}, me : T_1 \times ... \times T_n \times \pi^x_1 \times ... \times \pi^x_m \times \textbf{nil}*, \Gamma_f, (x, S)}$$

(T-APPLY1)

$$\frac{\Gamma_1, \Pi \vdash e : S_1 \rightarrow S_2, \Gamma_2 \quad \Gamma_2, \Pi \vdash el : S_3, \Gamma_3 \quad S_3 \lesssim S_1}{\Gamma_1, \Pi \vdash e(el) : S_2, \Gamma_3}$$

(T-APPLY2)

$$\frac{\Gamma_1, \Pi \vdash e : \textbf{any}, \Gamma_2 \quad \Gamma_2, \Pi \vdash el : S, \Gamma_3}{\Gamma_1, \Pi \vdash e_1(el) : \textbf{any}*, \Gamma_3}$$

(T-INVOKE1)

$$\frac{\begin{array}{c}\Gamma_1, \Pi \vdash e : T_s, \Gamma_2 \\ \Gamma_2[\sigma \mapsto T_s], \Pi \vdash e[id] : \textbf{const}\ S_1 \rightarrow S_2, \Gamma_3 \\ \Gamma_3[\sigma \mapsto T_s], \Pi \vdash el : S_3, \Gamma_4 \quad T_s \times S_3 \lesssim S_1\end{array}}{\Gamma_1, \Pi \vdash e{:}id(el) : [\sigma \mapsto T]S_2, \Gamma_4}$$

(T-INVOKE2)

$$\frac{\Gamma_1, \Pi \vdash e : \textbf{any}, \Gamma_2 \quad \Gamma_2, \Pi \vdash el : S, \Gamma_3}{\Gamma_1, \Pi \vdash e{:}id(el) : \textbf{any}*, \Gamma_3}$$

(T-DOTS)

$$\frac{\Gamma_1(...) = T}{\Gamma_1, \Pi \vdash ... : T*, \Gamma_1}$$

(T-SELF)

$$\frac{\Gamma_1, \Pi \vdash e : \textbf{self}, \Gamma_2 \quad \Gamma_2(\sigma) = T}{\Gamma_1, \Pi \vdash e : T, \Gamma_2}$$

(T-SETMETATABLE1)

$$\frac{\Gamma(id) = \textbf{self}}{\Gamma, \Pi \vdash \mathit{setmetatable}(\{\}, \{[\text{“\_\_}\mathit{index}\text{”}] = id\}) : \textbf{self}, \Gamma}$$

(T-SETMETATABLE2)

$$\frac{\Gamma(id) = \{K_1{:}V_1, ..., K_n{:}V_n\}_{\mathit{fixed}}}{\Gamma, \Pi \vdash \mathit{setmetatable}(\{\}, \{[\text{“\_\_}\mathit{index}\text{”}] = id\}) : \{K_1{:}V_1, ..., K_n{:}V_n\}_{\mathit{open}}, \Gamma}$$

(T-SETMETATABLE3)

$$\frac{\Gamma_1, \Pi \vdash e = T, \Gamma_2 \quad T = \{K_1{:}V_1, ..., K_n{:}V_n\}_{\mathit{closed}} \quad \Gamma_1(id) = \textbf{self} \quad \Gamma_1(\sigma) <: T}{\Gamma_1, \Pi \vdash \mathit{setmetatable}(e, \{[\text{“\_\_}\mathit{index}\text{”}] = id\}) : \textbf{self}, \Gamma_2[\sigma \mapsto T]}$$

(T-UNFOLD)

$$\frac{\Gamma_1, \Pi \vdash e : \mu x.T, \Gamma_2}{\Gamma_1, \Pi \vdash e : [x \mapsto \mu x.T]T, \Gamma_2}$$

(T-FOLD)

$$\frac{\Gamma_1, \Pi \vdash e : [x \mapsto \mu x.T]T, \Gamma_2}{\Gamma_1, \Pi \vdash e : \mu x.T, \Gamma_2}$$

(T-TERNARY)

$$\frac{\Gamma_1, \Pi \vdash e_1 : T_1, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : T_2, \Gamma_3 \quad \Gamma_3, \Pi \vdash e_3 : T_2, \Gamma_4}{\Gamma_1, \Pi \vdash e_1 \textbf{ and } e_2 \textbf{ or } e_3 : T_2, \Gamma_4}$$

C.3 Auxiliary functions

$$\begin{aligned}
first(\textbf{void}) &= \textbf{nil} \\
first(T*) &= T \cup \textbf{nil} \\
first(T \times P) &= T \\
first(S_1 \sqcup S_2) &= first(S_1) \cup first(S_2)
\end{aligned}$$

$$\begin{aligned}
read(\Pi, \phi(T_1, T_2)) &= T_2 \\
read(\Pi, \pi_i^x) &= proj(\Pi(x), i) \\
read(\Pi, T) &= T
\end{aligned}$$

$$write(\phi(T_1, T_2)) = T_1$$

$$write(T) = T$$

$$infer(T_1 \times ... \times T_n*, i) = \begin{cases} general(T_i) & \text{if } i < n \\ general(T_n \cup \textbf{nil}) & \text{if } i >= n \end{cases}$$

$$\begin{aligned}
general(\textbf{false}) &= \textbf{boolean} \\
general(\textbf{true}) &= \textbf{boolean} \\
general(int) &= \textbf{integer} \\
general(float) &= \textbf{number} \\
general(string) &= \textbf{string} \\
general(T_1 \cup T_2) &= general(T_1) \cup general(T_2) \\
general(S_1 \to S_2) &= general2(S_1) \to general2(S_2) \\
general(\{K_1{:}V_1, ..., K_n{:}V_n\}_{tag}) &= \{K_1{:}general(V_1), ..., K_n{:}general(V_n)\}_{tag} \\
general(\mu x.T) &= \mu x.general(T) \\
general(T) &= T
\end{aligned}$$

$$\begin{aligned}
general2(\textbf{void}) &= \textbf{void} \\
general2(T*) &= general(T)* \\
general2(T \times P) &= general(T) \times general2(P) \\
general2(S_1 \sqcup S_2) &= general2(S_1) \sqcup general2(S_2)
\end{aligned}$$

$$\begin{aligned}
proj(T_1 \times ... \times T_n \times T*, i) &= T_i \quad \text{if } i <= n \\
proj(S_1 \sqcup S_2, i) &= proj(S_1, i) \cup proj(S_2, i)
\end{aligned}$$

$$\begin{aligned}
filter(T_1 \cup T_2, T_1) &= filter(T_2, T_1) \\
filter(T_1 \cup T_2, T_2) &= filter(T_1, T_2) \\
filter(T_1 \cup T_2, T_3) &= filter(T_1, T_3) \cup filter(T_2, T_3) \\
filter(T_1, T_2) &= T_1
\end{aligned}$$

$$\begin{aligned}
close(T_1 \cup T_2) &= close(T_1) \cup close(T_2) \\
close(\{K_1{:}V_1, ..., K_n{:}V_n\}_{unique|open}) &= \{K_1{:}V_1, ..., K_n{:}V_n\}_{closed} \\
close(T) &= T
\end{aligned}$$

$$\begin{aligned}
fix(T_1 \cup T_2) &= fix(T_1) \cup fix(T_2) \\
fix(\{K_1{:}V_1, ..., K_n{:}V_n\}_{unique|open}) &= \{K_1{:}V_1, ..., K_n{:}V_n\}_{fixed} \\
fix(T) &= T
\end{aligned}$$

$$\begin{aligned}
open(T_1 \cup T_2) &= open(T_1) \cup open(T_2) \\
open(\{K_1{:}V_1, ..., K_n{:}V_n\}_{unique}) &= \{K_1{:}V_1, ..., K_n{:}V_n\}_{open} \\
open(T) &= T
\end{aligned}$$

$$fpt(T_1 \times .... \times T_n{*} \sqcup T'_1 \times ... \times T'_n{*}, T, i) = T'_1 \times ... \times T'_n{*}$$

$$gpt(T_1 \times .... \times T_n{*} \sqcup T'_1 \times ... \times T'_n{*}, T, i) = T_1 \times ... \times T_n{*}$$

$$closeall(\Gamma[id_1 \mapsto T_1, ..., id_n \mapsto T_n]) = \Gamma[id_1 \mapsto close(T_1), ..., id_n \mapsto close(T_n)]$$

$$crall(\Gamma[id_1 \mapsto T_1, ..., id_n \mapsto T_n]) = \Gamma[id_1 \mapsto close(write(T_1)), ..., id_n \mapsto close(write(T_n))]$$

$$closeset(\Gamma, \{id_1, ..., id_n\}) = \Gamma[id_1 \mapsto close(\Gamma(id_1)), ..., id_n \mapsto close(\Gamma(id_n))]$$

$$crset(\Gamma, \{id_1, ..., id_n\}) = \Gamma[id_1 \mapsto close(write(\Gamma(id_1))), ..., id_n \mapsto close(write(\Gamma(id_n)))]$$

$$openset(\Gamma, \{id_1, ..., id_n\}) = \Gamma[id_1 \mapsto open(\Gamma(id_1)), ..., id_n \mapsto open(\Gamma(id_n))]$$

$$\begin{aligned}
&merge(\Gamma_1[id_1 \mapsto T_1, ..., id_n \mapsto T_n], \Gamma_2[id_1 \mapsto T'_1, ..., id_n \mapsto T'_n]) = \\
&\qquad \Gamma_3[id_1 \mapsto merget(T_1, T'_1), ..., id_n \mapsto merget(T_n, T'_n)]
\end{aligned}$$

$$\begin{aligned}
&merget(\{K_1{:}V_1, ..., K_n{:}V_n, K{:}V\}_{tag}, \{K_1{:}V_1, ..., K_n{:}V_n, K'{:}V'\}_{tag}) = \\
&\qquad\qquad\qquad \{K_1{:}V_1, ..., K_n{:}V_n, K{:}V, K'{:}V'\}_{tag} \\
&\qquad\qquad\qquad\qquad\qquad merget(T, T') = T' \text{ if } \quad T <: T'
\end{aligned}$$

$$\begin{aligned}
fav(\textbf{skip}) &= \emptyset \\
fav(s_1 \; ; \; s_2) &= fav(s_1) \cup fav(s_2) \\
fav(\bar{l} = el) &= fav(\bar{l}) \cup fav(el) \\
fav(\textbf{while}\; e \;\textbf{do}\; s) &= fav(s) \\
fav(\textbf{if}\; e \;\textbf{then}\; s_1 \;\textbf{else}\; s_2) &= fav(s_1) \cup fav(s_2) \\
fav(\textbf{local}\; \overline{id{:}T} = el \;\textbf{in}\; s) &= fav(s) \\
fav(\textbf{local}\; \overline{id} = el \;\textbf{in}\; s) &= fav(el) \cup fav(s) \\
fav(\textbf{rec}\; id{:}T = f \;\textbf{in}\; s) &= fav(f) \cup fav(s) \\
fav(\textbf{return}\; el) &= fav(el) \\
fav(\lfloor a \rfloor_0) &= fav(a) \\
fav(\textbf{fun}\; id_1{:}id_2\; (pl){:}S\; s \;;\; \textbf{return}\; el) &= fav(s) \cup fav(el) \\
fav(\textbf{nil}) &= \emptyset \\
fav(k) &= \emptyset \\
fav(id) &= \emptyset \\
fav(e_1[e_2]) &= \emptyset \\
fav(<T>\; id) &= \emptyset \\
fav(\textbf{fun}\; (pl){:}S\; s \;;\; \textbf{return}\; el) &= fav(s) \cup fav(el) \\
fav(\{\; \overline{[e_1] = e_2}\; \}) &= fav(\overline{e_1}) \cup fav(\overline{e_2}) \\
fav(\{\; \overline{[e_1] = e_2}, me\; \}) &= fav(\overline{e_1}) \cup fav(\overline{e_2}) \cup fav(me) \\
fav(e_1 + e_2) &= fav(e_1) \cup fav(e_2) \\
fav(e_1 \;..\; e_2) &= fav(e_1) \cup fav(e_2) \\
fav(e_1 == e_2) &= fav(e_1) \cup fav(e_2) \\
fav(e_1 < e_2) &= fav(e_1) \cup fav(e_2) \\
fav(e_1 \;\&\; e_2) &= fav(e_1) \cup fav(e_2) \\
fav(e_1 \;\textbf{and}\; e_2) &= fav(e_1) \cup fav(e_2) \\
fav(e_1 \;\textbf{or}\; e_2) &= fav(e_1) \cup fav(e_2) \\
fav(\textbf{not}\; e) &= fav(e)
\end{aligned}$$

$$
\begin{aligned}
fav(\#\ e) &= fav(e) \\
fav(\lfloor me \rfloor_1) &= fav(me) \\
fav(id_l) &= \{id\} \\
fav(id[k]\ {<}T{>}) &= \emptyset \\
fav(e(el)) &= fav(e) \cup fav(el) \\
fav(e{:}n(el)) &= fav(e) \cup fav(el) \\
fav(...) &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
rv(\mathbf{skip}) &= \emptyset \\
rv(s_1\ ;\ s_2) &= rv(s_1) \cup rv(s_2) \\
rv(\bar{l} = el) &= rv(\bar{l}) \cup rv(el) \\
rv(\mathbf{while}\ e\ \mathbf{do}\ s) &= rv(e) \cup rv(s) \\
rv(\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2) &= rv(e) \cup rv(s_1) \cup rv(s_2) \\
rv(\mathbf{local}\ \overline{id{:}T} = el\ \mathbf{in}\ s) &= rv(el) \cup rv(s) \\
rv(\mathbf{local}\ \overline{id} = el\ \mathbf{in}\ s) &= rv(el) \cup rv(s) \\
rv(\mathbf{rec}\ id{:}T = f\ \mathbf{in}\ s) &= rv(f) \cup rv(s) \\
rv(\mathbf{return}\ el) &= rv(el) \\
rv(\lfloor a \rfloor_0) &= rv(a) \\
rv(\mathbf{fun}\ id_1{:}id_2\ (pl){:}S\ s\ ;\ \mathbf{return}\ el) &= rv(s) \cup rv(el) \\
rv(\mathbf{nil}) &= \emptyset \\
rv(k) &= \emptyset \\
rv(id) &= \{id\} \\
rv(e_1[e_2]) &= \emptyset \\
rv({<}T{>}\ id) &= \emptyset \\
rv(\mathbf{fun}\ (pl){:}S\ s\ ;\ \mathbf{return}\ el) &= rv(s) \cup rv(el) \\
rv(\{\ \overline{[e_1] = e_2}\ \}) &= rv(\overline{e_1}) \cup rv(\overline{e_2}) \\
rv(\{\ \overline{[e_1] = e_2}, me\ \}) &= rv(\overline{e_1}) \cup rv(\overline{e_2}) \cup rv(me) \\
rv(e_1 + e_2) &= rv(e_1) \cup rv(e_2) \\
rv(e_1\ ..\ e_2) &= rv(e_1) \cup rv(e_2) \\
rv(e_1 == e_2) &= rv(e_1) \cup rv(e_2) \\
rv(e_1 < e_2) &= rv(e_1) \cup rv(e_2) \\
rv(e_1\ \&\ e_2) &= rv(e_1) \cup rv(e_2)
\end{aligned}
$$

$$
\begin{aligned}
rv(e_1 \textbf{ and } e_2) &= rv(e_1) \cup rv(e_2) \\
rv(e_1 \textbf{ or } e_2) &= rv(e_1) \cup rv(e_2) \\
rv(\textbf{not } e) &= rv(e) \\
rv(\#\ e) &= rv(e) \\
rv(\lfloor me \rfloor_1) &= rv(me) \\
rv(id_l) &= \emptyset \\
rv(id[k] <T>) &= \emptyset \\
rv(e(el)) &= rv(e) \cup rv(el) \\
rv(e{:}n(el)) &= rv(e) \cup rv(el) \\
rv(...) &= \emptyset
\end{aligned}
$$