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 [PT00] 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 B presents the complete syntax of Typed Lua in extended BNF.

We use the consistent-subtyping relation of gradual typing [ST07, SVB13] 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 [AFT13]. 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 [Goo11] and TypeScript [Mic12], 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

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*

Figure 3.1: The concrete syntax of Typed Lua types

Figure 3.1 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 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 4.2.

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 [ST07, SVB13]. 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 4.2.

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 3.4 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 3.3 and Section 3.4, 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 3.2 shows that Typed Lua uses second-level types in the definition of function types. A second-level type is either a tuple of first-level types optionally ending in a variadic type or a union of these tuples. A variadic type `t*` is a generator for a sequence of values of the union type `t|nil`. The empty tuple `()` is syntactic sugar to `(nil*)`, as we will show that Typed Lua always ends a tuple type in a variadic type. 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 modes of operation: the *default* mode and the *strict* mode. In the default mode, the compiler adds `value*` to the type of the parameter list and `nil*` to the return type when the programmer does not specify a variadic tail. It has this behavior to preserve the semantics of function calls in Lua, that is, it uses `value*` to discard extra arguments and `nil*` to replace missing arguments. In the strict mode, the compiler adds `nil*` to both parts of the function type. Even though it still uses `nil*` to replace missing values, it also uses `nil*` instead of `value*` to catch arity mismatch.

We will use the following function to illustrate how these two 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, nil*) -> (integer, nil*)`. The compiler adds `nil*` 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 modes of operation affect function calls, they do not affect other multiple assignments. This means that both 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 too 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 modes of operation. 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 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 modes of operation. 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 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 modes of operation. 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 modes of operation. The function call `v()` compiles without any compile-time warnings in both modes because the type of the argument list is `(nil*)`, which is consistent with `(string*)`. The call `v("hello", "world")` also compiles without any warnings in both modes because the type of the argument list is `("hello", "world", nil*)`, which is 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 modes of operation:

```
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, nil*) -> (string, nil*)` in the strict mode. The 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, 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 4.3 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* [GSK11, THF10]. 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 4.3.

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 3.3 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 2.5. 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

tabletype ::= ‘{’ [*tabletypebody*] ‘}’
tabletypebody ::= *maptype* | *recordtype*
maptype ::= [*keytype* ‘:’] *type*
keytype ::= *basetype* | **value**
recordtype ::= *recordfield* {‘,’ *recordfield*} [‘,’ *type*]
recordfield ::= [**const**] *literaltype* ‘:’ *type*

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 2.5. 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 2.5. 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 table types as either *unique*, *open*, *fixed*, or *closed* to identify which operations are safe. Type annotations define *closed* table types, as this tag describes table types that do not provide any guarantees about keys that do not appear in the table type. In contrast, *unique*, *open*, and *fixed* table types guarantee that a table does not contain any keys which are not listed in its table type, though they provide different guarantees over their references. An *unique* table type guarantees that it has only one reference. In particular, this is the case of table constructors, so they always have *unique* table types. In the previous example, `person` has an *unique* type because it has no alias. Any alias to *unique* tables makes them *open*, as we use this tag to keep track of *unique* aliasing. This means that *open* table types can only have *closed* aliases, and it is guaranteed that only one reference remains *open*. Finally, *fixed* table types can have any number of *closed* and *fixed* aliases, so its type cannot change. In Section 3.7 we will show that we use *fixed* table types to describe the type of classes. Typed Lua also has different subtyping rules that handle these different tags, which we explain in Section 4.2.

Typed Lua uses three rules to check whether field assignment can change the type of *unique* and *open* tables: the table must be assigned to a local variable to the current block, the new type must only add new fields, and the variable cannot have been assigned in another function.

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 forbidden to change the type of existing fields:

```
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 not valid because they are trying to change the type of fields that already exist. Even though it would be sound to allow changing the type of fields in *unique* table types because they cannot have any alias, this could still lead to a weak static type checking

approach. In other words, allowing the type of the fields to evolve from a base type to an union type or even to the dynamic type could hide some type errors, so our design choice was to keep it as precise as we could. Moreover, it would be unsound to allow changing the type of fields in *open* table types because they can have *closed* aliases, so it made more sense to have the same rule to both table types.

The following example shows that it is not allowed to change 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* tables, 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 4.3 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 3.4 shows the concrete syntax of the `typealias` construct.

Type aliases

typealias ::= [**local**] **typealias** *Name* '=' *type*

Figure 3.4: The concrete syntax of Typed Lua type aliases

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 }
```

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 3.5 shows the concrete syntax of the `interface` construct. In Section 3.7 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.

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:

Interfaces

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

Figure 3.5: The concrete syntax of Typed Lua interfaces

```
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 3.3.

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 3.3 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[`$\mathtt{e}_1$`] == `$\mathtt{e}_2$, it checks whether the type of the variable `v` is an union of table types, whether the type of $\mathtt{e}_1$ 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 2.5 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

2.5, 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.

Figure 3.6 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 3.5, a type alias declaration creates an alias to a more general type, while an

The complete syntax of description files

$$
\begin{aligned}
\textit{description} &::= \textit{desclist} \\
\textit{desclist} &::= \textit{descitem}\ \{\textit{descitem}\} \\
\textit{descitem} &::= \textit{typedid} \mid \textit{interface} \mid \textit{userdata} \mid \textit{typealias} \\
\textit{typedid} &::= [\textbf{const}]\ \textit{id}\ \text{`:'}\ \textit{type} \\
\textit{interface} &::= \textbf{interface}\ \textit{typedec} \\
\textit{userdata} &::= \textbf{userdata}\ \textit{typedec} \\
\textit{typedec} &::= \textit{Name}\ \{\textit{decitem}\}\ \textbf{end} \\
\textit{decitem} &::= \textit{idlist}\ \text{`:'}\ \textit{idtype} \\
\textit{idtype} &::= \textit{type} \mid \textit{methodtype} \\
\textit{idlist} &::= \textit{id}\ \{\text{`,'}\ \textit{id}\} \\
\textit{id} &::= [\textbf{const}]\ \textit{Name} \\
\textit{typealias} &::= \textbf{typealias}\ \textit{Name}\ \text{`='}\ \textit{type}
\end{aligned}
$$

Figure 3.6: The concrete syntax of Typed Lua description files

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` [dF14], 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 5 we will discuss the issues that we found while typing the Lua standard library and other case studies, which include the `lmd5` library.