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

Typed Lua includes types that can appear in annotations and some special types that cannot appear in annotations, though they play special role in type checking some Lua idioms and handling flow typing. Special types cannot appear in annotations because they do not have a corresponding concrete syntax, so we also isolated them in the formalization to make the implementation easier. First we present the types that compose the type language of our type system, and then we present the special types, indicating why they are necessary.

Figure 4.1 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 the input and output of functions. 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, and recursive types. Second-level types include tuple types and unions of tuple

Type Language

$$
\begin{array}{rll}
F ::= & & \text{FIRST-LEVEL TYPES:} \\
& L & \textit{literal types} \\
& \mid B & \textit{base types} \\
& \mid \textbf{nil} & \textit{nil type} \\
& \mid \textbf{value} & \textit{top type} \\
& \mid \textbf{any} & \textit{dynamic type} \\
& \mid \textbf{self} & \textit{self type} \\
& \mid F_1 \cup F_2 & \textit{union types} \\
& \mid S_1 \to S_2 & \textit{function types} \\
& \mid \{\overline{F{:}V}\}_{unique|open|fixed|closed} & \textit{table types} \\
& \mid x & \textit{type variables} \\
& \mid \mu x.F & \textit{recursive types} \\
L ::= & & \text{LITERAL TYPES:} \\
& \textbf{false} \mid \textbf{true} \mid \textit{int} \mid \textit{float} \mid \textit{string} & \\
B ::= & & \text{BASE TYPES:} \\
& \textbf{boolean} \mid \textbf{integer} \mid \textbf{number} \mid \textbf{string} & \\
V ::= & & \text{VALUE TYPES:} \\
& F \mid \textbf{const}\ F & \\
S ::= & & \text{SECOND-LEVEL TYPES:} \\
& P & \textit{tuple types} \\
& \mid S_1 \sqcup S_2 & \textit{unions of tuple types} \\
P ::= & & \text{TUPLE TYPES:} \\
& F* & \textit{variadic types} \\
& \mid F \times P & \textit{pair types}
\end{array}
$$

Figure 4.1: The abstract syntax of Typed Lua types

types. Tuple types include variadic types and pair types. Types are ordered by a subtype relationship that we introduce in Section 4.2, 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.

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

Typed Lua uses the type **self** to represent the *receiver* in object-oriented method definitions and method calls. As we mentioned in Section 3.7, we need the type **self** to prevent programs from indexing a method without calling it with the correct receiver.

Union types $F_1 \cup F_2$ 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 end 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. A variadic type $F*$ represents a sequence of values of type $F \cup$ **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 2.5. 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 $\{\overline{F{:}V}\}_{unique|open|fixed|closed}$, where the notation $\overline{F{:}V}$ denotes the list $F_1{:}V_1, ..., F_n{:}V_n$. Each V_i represents the type of the value that table keys of type F_i map to. Value types represent mutable fields by default, but we can use the **const** type to make them represent immutable fields. Making a field **const** does not guarantee that its value cannot change, as the table may have aliases with a non-**const** type for that field. Typed Lua needs immutable fields to enable depth subtyping between table types.

We also use the tags *unique*, *open*, *fixed*, and *closed* to classify table types.

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. The tags *unique*, *open*, and *fixed* represent tables with no keys that do not inhabit one of the table's key types, but with different guarantees about the reference to a value of that type. A reference to an *unique* table is guaranteed to point to a table that has no other references to it. In particular, the type of the table constructor has this tag, as it allows greater flexibility in reinterpreting its type. A reference to an *open* table is guaranteed to have only *closed* references pointing to the same table, a guarantee that still lets the type system reinterpret the type of a reference, but with more restrictions. A reference to a *fixed* table can have any number of *fixed* or *closed* references point to it, so its type cannot change anymore. In particular, the type of a class has this tag in our type system.

A *fixed* table type guarantees that there are no keys with a type that is not one of its key types. Even though this guarantee allows type-safe iteration on *fixed* table types, it forbids width subtyping that is necessary for object-oriented programming, so *closed* table types remove this guarantee to allow width subytping between other table types and *closed* table types. This means that objects have *closed* table types, while their classes have *fixed* 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 4.3 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 $\mu x.F$, where F is a first-level type that x represents. For instance, $\mu x.\{\text{“}info\text{”} : \mathbf{integer}, \text{“}next\text{”} : x \cup \mathbf{nil}\}_{closed}$ is a type for singly-linked lists of integers. In Section 3.5 we mentioned that we can use the following interface declaration as an alias to this type:

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

With recursive types we finish the discussion about Typed Lua types, and we begin the discussion about special types.

Figure 4.2 presents the special types that Typed Lua includes for typing some Lua idioms and flow typing. Typed Lua splits special types into three categories: *expression types*, *expression list types*, and *filter result types*. Expression types represent the type of expressions in the type system, which can

Special Types

$T ::=$		EXPRESSION TYPES:
	F	*first-level types*
	$\mid \phi(F_1, F_2)$	*filter types*
	$\mid \pi_i^x$	*projection types*
$E ::=$		EXPRESSION LIST TYPES:
	$T*$	*variadic types*
	$\mid T \times E$	*pair types*
$R ::=$		FILTER RESULT TYPES:
	void	*void type*
	$\mid F$	*first-level types*

Figure 4.2: The special types used by Typed Lua

be any first-level type, a filter type, or a projection type. Expression list types represent the type of lists of expressions, which are tuples of expression types that end with a variadic type. Filter result types are either the type **void** or a first-level type. In Section 4.3 we will show that Typed Lua uses filter types in flow typing, being the type **void** a type with no values, used by the type system as a way to detect branches that are unreachable due to flow typing.

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 3.3. This means that `type` predicates use filter types of the form $\phi(F_1, F_2)$ to discriminate local variables that are bound to union types. In a filter type $\phi(F_1, T_2)$, F_1 is the original type and F_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 4.3 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 3.3. 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.

4.2 Subtyping

Our type system uses subtyping [Car84, AC96] to order types and consistent-subtyping [ST07, SVB13] 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 2.3, 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 $\Sigma \vdash T_1 <: T_2$, where T_1 and T_2 are two types of the same kind. 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**:

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

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

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

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**:

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

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:

$$\frac{\Sigma \vdash F_1 <: F \quad \Sigma \vdash F_2 <: F}{\Sigma \vdash F_1 \cup F_2 <: F}\ (\text{S-UNION1}) \qquad \frac{\Sigma \vdash F <: F_1}{\Sigma \vdash F <: F_1 \cup F_2}\ (\text{S-UNION2}) \qquad \frac{\Sigma \vdash F <: F_2}{\Sigma \vdash F <: F_1 \cup F_2}\ (\text{S-UNION3})$$

The first rule shows that a union type $F_1 \cup F_2$ is a subtype of F if both F_1 and F_2 are subtypes of F; and the other rules show that a type F is a subtype of a union type $F_1 \cup F_2$ if F is a subtype of either F_1 or F_2.

The subtyping rule for function types is also standard:

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

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 subtyping rule for pair types is the standard covariant rule:

$$\frac{\Sigma \vdash F_1 <: F_2 \quad \Sigma \vdash P_1 <: P_2}{\Sigma \vdash F_1 \times P_1 <: F_2 \times P_2}\ (\text{S-PAIR1})$$

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

The rule S-VARARG1 handles subtyping between two variadic types:

$$\frac{\Sigma \vdash F_1 \cup \mathbf{nil} <: F_2 \cup \mathbf{nil}}{\Sigma \vdash F_1* <: F_2*}\ (\text{S-VARARG1})$$

This rule shows that F_1* is a subtype of F_2* if $F_1 \cup \mathbf{nil}$ is a subtype of

$F_2 \cup \mathbf{nil}$. 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 $\mathbf{nil} \not<: \mathbf{number}$.

The other rules handle subtyping between a varidic type and a pair type:

$$\frac{\Sigma \vdash F_1 \cup \mathbf{nil} <: F_2 \quad \Sigma \vdash F_1* <: P_2}{\Sigma \vdash F_1* <: F_2 \times P_2}\ (\text{S-VARARG2}) \qquad \frac{\Sigma \vdash F_1 <: F_2 \cup \mathbf{nil} \quad \Sigma \vdash P_1 <: F_2*}{\Sigma \vdash F_1 \times P_1 <: F_2*}\ (\text{S-VARARG3})$$

These rules state the conditions when tuple types of different length are compatible. 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:

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

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:

$$\frac{\forall i\ \exists j \quad \Sigma \vdash F_j <: F'_i \quad \Sigma \vdash F'_i <: F_j \quad \Sigma \vdash V_j <:_c V'_i}{\Sigma \vdash \{\overline{F{:}V}\}_{fixed|closed} <: \{\overline{F'{:}V'}\}_{closed}}\ (\text{S-TABLE1})$$

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:

$$\frac{\Sigma \vdash F_1 <: F_2 \quad \Sigma \vdash F_2 <: F_1}{\Sigma \vdash F_1 <:_c F_2}\ (\text{S-FIELD1}) \qquad \frac{\Sigma \vdash F_1 <: F_2}{\Sigma \vdash \mathbf{const}\ F_1 <:_c \mathbf{const}\ F_2}\ (\text{S-FIELD2})$$

$$\frac{\Sigma \vdash F_1 <: F_2}{\Sigma \vdash F_1 <:_c \mathbf{const}\ F_2}\ (\text{S-FIELD3})$$

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,

$$\{\text{“}x\text{”} : 1, \text{“}y\text{”} : 2\}_{closed} \not<: \{\text{“}x\text{”} : \mathbf{integer}, \text{“}y\text{”} : \mathbf{integer} \cup \mathbf{nil}\}_{closed}$$

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,

$$\{\text{“}x\text{”} : \mathbf{integer}, \text{“}y\text{”} : \mathbf{integer}\}_{closed} \not<:$$
$$\{\text{“}x\text{”} : \mathbf{integer}, \text{“}y\text{”} : \mathbf{integer} \cup \mathbf{nil}\}_{closed}$$

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:

$$\text{(S-TABLE2)}$$
$$\frac{\begin{array}{c}\forall i\, \forall j \quad \Sigma \vdash F_i <: F'_j \rightarrow \Sigma \vdash V_i <:_u V'_j \\ \forall j \nexists i \quad \Sigma \vdash F_i <: F'_j \rightarrow \Sigma \vdash \mathbf{nil} <:_o V'_j\end{array}}{\Sigma \vdash \{\overline{F{:}V}\}_{unique} <: \{\overline{F'{:}V'}\}_{closed}}$$

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. More precisely, this rule states that it is safe to

recast the table type $\{\text{“}x\text{”} : \mathbf{integer}, \text{“}y\text{”} : \mathbf{integer}, \text{“}z\text{”} : \mathbf{integer}\}_{unique}$ to $\{\mathbf{string} : \mathbf{integer} \cup \mathbf{nil}\}_{closed}$, as long as the new type becomes inaccessible with the original type.

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:

$$\frac{\text{(S-FIELD4)}\quad \Sigma \vdash F_1 <: F_2}{\Sigma \vdash F_1 <:_u F_2} \qquad \frac{\text{(S-FIELD5)}\quad \Sigma \vdash F_1 <: F_2}{\Sigma \vdash \mathbf{const}\ F_1 <:_u \mathbf{const}\ F_2}$$

$$\frac{\text{(S-FIELD6)}\quad \Sigma \vdash F_1 <: F_2}{\Sigma \vdash \mathbf{const}\ F_1 <:_u F_2} \qquad \frac{\text{(S-FIELD7)}\quad \Sigma \vdash F_1 <: F_2}{\Sigma \vdash F_1 <:_u \mathbf{const}\ F_2}$$

$$\frac{\text{(S-FIELD8)}\quad \Sigma \vdash \mathbf{nil} <: F}{\Sigma \vdash \mathbf{nil} <:_o F} \qquad \frac{\text{(S-FIELD9)}\quad \Sigma \vdash \mathbf{nil} <: F}{\Sigma \vdash \mathbf{nil} <:_o \mathbf{const}\ F}$$

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:

$$\frac{\begin{array}{c}\text{(S-TABLE3)}\\ \forall i\ \exists j \quad \Sigma \vdash F_i <: F'_j \wedge \Sigma \vdash V_i <:_u V'_j \\ \forall j\ \nexists i \quad \Sigma \vdash F_i <: F'_j \rightarrow \Sigma \vdash \mathbf{nil} <:_o V'_j\end{array}}{\Sigma \vdash \{\overline{F{:}V}\}_{unique} <: \{\overline{F'{:}V'}\}_{unique|open|fixed}}$$

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

$$\frac{\begin{array}{c}\text{(S-TABLE4)}\\ \forall i\ \forall j \quad \Sigma \vdash F_i <: F'_j \rightarrow \Sigma \vdash V_i <:_c V'_j \\ \forall j\ \nexists i \quad \Sigma \vdash F_i <: F'_j \rightarrow \Sigma \vdash \mathbf{nil} <:_o V'_j\end{array}}{\Sigma \vdash \{\overline{F{:}V}\}_{open} <: \{\overline{F'{:}V'}\}_{closed}}$$

However, the rule that handles subtyping among *open* and *open* or *fixed*

table types does not allow width subtyping:

$$\text{(S-TABLE5)}$$
$$\frac{\begin{array}{c}\forall i\ \exists j \quad \Sigma \vdash F_i <: F'_j \wedge \Sigma \vdash V_i <:_c V'_j \\ \forall j\ \nexists i \quad \Sigma \vdash F_i <: F'_j \rightarrow \Sigma \vdash \mathbf{nil} <:_o V'_j\end{array}}{\Sigma \vdash \{\overline{F{:}V}\}_{open} <: \{\overline{F'{:}V'}\}_{open|fixed}}$$

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:

$$\text{(S-TABLE6)}$$
$$\frac{\begin{array}{c}\forall i\ \exists j \quad \Sigma \vdash F_i <: F'_j \quad \Sigma \vdash F'_j <: F_i \quad \Sigma \vdash V_i <:_c V'_j \\ \forall j\ \exists i \quad \Sigma \vdash F_i <: F'_j \quad \Sigma \vdash F'_j <: F_i \quad \Sigma \vdash V_i <:_c V'_j\end{array}}{\Sigma \vdash \{\overline{F{:}V}\}_{fixed} <: \{\overline{F'{:}V'}\}_{fixed}}$$

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* [Car86] to define subtyping between recursive types:

$$\text{(S-AMBER)}\qquad \frac{\Sigma[x_1 <: x_2] \vdash F_1 <: F_2}{\Sigma \vdash \mu x_1.F_1 <: \mu x_2.F_2}$$

$$\text{(S-ASSUMPTION)}\qquad \frac{x_1 <: x_2 \in \Sigma}{\Sigma \vdash x_1 <: x_2}$$

The rule S-AMBER also uses the rule S-ASSUMPTION to check whether $\mu x_1.F_1 <: \mu x_2.F_2$. 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 $F_1 <: F_2$. 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:

$$\text{(S-UNFOLDR)}\qquad \frac{\Sigma \vdash F_1 <: [x \mapsto \mu x.F_2]F_2}{\Sigma \vdash F_1 <: \mu x.F_2}$$

$$\text{(S-UNFOLDL)}\qquad \frac{\Sigma \vdash [x \mapsto \mu x.F_1]F_1 <: F_2}{\Sigma \vdash \mu x.F_1 <: F_2}$$

As an example, the rule S-UNFOLDR allows our type system to type check the function `insert` from Section 3.5:

```
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. More precisely, a filter type $\phi(F_1, F_2)$ is a subtype of the same filter type $\phi(F_1, F_2)$, which shares the same types F_1 and F_2.

Projection types are subtypes only of themselves. 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.

The subtyping rules for expression list types are similar to the subtyping rules for tuple types.

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 $\Sigma \vdash T_1 \lesssim T_2$, where T_1 and T_2 are two types of the same kind. 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:

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

If we had set the type **any** as both bottom and top types of our subtyping relation, then any type F_1 would be a subtype of any other type F_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 F_1 to **any** and then up-cast **any** to any other type F_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 Type checking

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 C 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.

Abstract Syntax

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

Figure 4.3: The abstract syntax of Typed Lua

Figure 4.3 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, F are first-level types, and S are second-level types. The notation $\overline{id{:}F}$ denotes the list $id_1{:}F_1, ..., id_n{:}F_n$.

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 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 $id[e] <V>$ and the expression $<F> id$. 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 expression types, as the type of an expression can be a first-level type, a filter type, or a projection type. 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 4.3 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.

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 $1 \times \text{“}foo\text{”} \times \mathbf{nil}*$. Note that our type system includes the type $\mathbf{nil}*$ to replace missing values. The rules that type check expression lists introduce the type $\mathbf{nil}*$ 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:

$$\frac{\Gamma_1, \Pi \vdash e_i : F_i, \Gamma_{i+1} \quad \Gamma_f = merge(\Gamma_1, ..., \Gamma_{n+1}) \quad n = \mid \overline{e} \mid}{\Gamma_1, \Pi \vdash \overline{e} : F_1 \times ... \times F_n \times \mathbf{nil}*, \Gamma_f} \text{(T-EXPLIST1)}$$

Later, in this 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_f, if there are no conflicts. 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:

$$\text{(T-LHSLIST)}$$
$$\frac{\Gamma_1, \Pi \vdash l_i : F_i, \Gamma_{i+1} \quad \Gamma_f = merge(\Gamma_1, ..., \Gamma_{n+1}) \quad n = |\ \bar{l}\ |}{\Gamma_1, \Pi \vdash \bar{l} : F_1 \times ... \times F_n \times \textbf{value}*, \Gamma_f}$$

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

$$1 \times \text{“}foo\text{”} \times \textbf{nil}* \lesssim \textbf{integer} \times \textbf{string} \times \textbf{value}*$$

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

$$1 \times \text{“}foo\text{”} \times \textbf{nil}* \lesssim \textbf{integer} \times \textbf{string} \times (\textbf{string} \cup \textbf{nil}) \times \textbf{value}*$$

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**.

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:

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

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 *proj* to ensure that only the first value is produced:

$$\frac{\text{(T-STMAPPLY1)}}{\Gamma_1, \Pi \vdash e(el) : S, \Gamma_2}{\Gamma_1, \Pi \vdash \lfloor e(el) \rfloor_0, \Gamma_2} \qquad \frac{\text{(T-EXPAPPLY1)}}{\Gamma_1, \Pi \vdash e(el) : S, \Gamma_2}{\Gamma_1, \Pi \vdash \lfloor e(el) \rfloor_1 : proj(S, 1), \Gamma_2}$$

We can define *proj* inductively as follows:

$$\begin{aligned}
proj(S_1 \sqcup S_2, i) &= proj(S_1, i) \cup proj(S_2, i) \\
proj(F*, i) &= nil(F) \\
proj(F \times P, 1) &= F \\
proj(F \times P, i) &= proj(P, i-1) \\
proj(E*, i) &= nil(E) \\
proj(T \times E, 1) &= T \\
proj(T \times E, i) &= proj(E, i-1) \\
nil(T) &= \begin{cases} T & \text{if } \mathbf{nil} \lesssim T \\ T \cup \mathbf{nil} & \text{otherwise} \end{cases}
\end{aligned}$$

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

$$f(\text{“}foo\text{”})$$

type checks through the rule T-APPLY1, because

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

but the function call

$$f(\text{“}foo\text{”}, 1, 2, 3)$$

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

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

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 $\mathbf{integer} \times \mathbf{integer} \times \mathbf{nil}* \rightarrow \mathbf{integer} \times \mathbf{nil}*$. The function call

$$f(1)$$

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

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

and the function call

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

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

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

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 $\{\}$; it always has type $\{\}_{unique}$.

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, Typed Lua uses the auxiliary relation $\Gamma_1, \Pi \vdash [e_1] = e_2 : (F, V), \Gamma_2$ 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 (F, V) and a new type environment Γ_2. A pair (F, V) means that e_1 has type F and e_2 has type V, where F is the type of the key and V is the type of the field value.

After type checking each table field, our type system uses each pair (F, 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. The predicate *wf* also uses the auxiliary predicate *tag* to forbid *unique* and *open* fields. Formally, we can define *wf* inductivelly as follows:

$$\begin{aligned}
wf(\{\overline{F : V}\}_{unique|open|fixed|closed}) &= \forall i\ ((\nexists j\ i \neq j\ \wedge\ F_i \lesssim F_j)\ \wedge\ wf(V_i) \wedge \\
&\qquad \neg tag(V_i, unique)\ \wedge\ \neg tag(V_i, open)) \\
wf(\mathbf{const}\ F) &= wf(F) \\
wf(F_1 \cup F_2) &= wf(F_1)\ \wedge\ wf(F_2) \\
wf(\mu x.F) &= wf(F) \\
wf(S_1 \to S_2) &= wf(S_1)\ \wedge\ wf(S_2) \\
wf(S_1 \sqcup S_2) &= wf(S_1)\ \wedge\ wf(S_2) \\
wf(F*) &= wf(F) \\
wf(F \times P) &= wf(F)\ \wedge\ wf(P) \\
wf(F) &= \top \quad \text{for all other cases}
\end{aligned}$$

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 field 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. Due to this restriction, the rule that type check table fields use the auxiliary function *fix* (in the definition of *vt*) to change any *unique* and *open* table types used in the field initializer to *fixed*. Rule T-FIELD defines this behavior:

$$\text{(T-FIELD)}$$
$$\frac{\Gamma_1, \Pi \vdash e_2 : V, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_1 : F, \Gamma_3}{\Gamma_1, \Pi \vdash [e_1] = e_2 : (F, vt(F, V)), \Gamma_3}$$

The rule T-FIELD uses the auxiliary function *vt* to adjust the type of the value according to the type of the key. More precisely, *vt* includes the type **nil** in the type of the value when the type of the key is not a literal type. It also uses *fix* to prevent *unique* and *open* table types to appear in the type of the field values. We define these functions as follows:

$$\begin{aligned} vt(L, V) &= fix(V) \\ vt(F_1, F_2) &= nil(fix(F_2)) \\ vt(F_1, \mathbf{const}\ F_2) &= \mathbf{const}\ nil(fix(F_2)) \end{aligned}$$

$$\begin{aligned} fix(F_1 \cup F_2) &= fix(F_1) \cup fix(F_2) \\ fix(\{\overline{F{:}V}\}_{unique|open}) &= \{\overline{F{:}V}\}_{fixed} \\ fix(F) &= F \end{aligned}$$

The rule T-CONSTRUCTOR1 uses these steps to type check a table constructor with a list of table fields that do not end with an expression that potentially returns multiple values:

$$\text{(T-CONSTRUCTOR1)}$$
$$\frac{\begin{array}{c}\Gamma_1, \Pi \vdash ([e_1] = e_2)_i : (F_i, V_i), \Gamma_{i+1} \quad T = \{F_1{:}V_1, ..., F_n{:}V_n\}_{unique} \\ wf(T) \quad n = |\ \overline{[e_1] = e_2}\ | \quad \Gamma_f = merge(\Gamma_1, ..., \Gamma_{n+1})\end{array}}{\Gamma_1, \Pi \vdash \{\ \overline{[e_1] = e_2}\ \} : T, \Gamma_f}$$

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-CONSTRUCTOR1. The subtyping rule for *unique* table types allows us assigning this table to a variable with a more general type such as $\{1 : \mathbf{string}, 2 : \mathbf{string}, 3 : \mathbf{string}\}_{closed}$ or even $\{\mathbf{integer} : \mathbf{string} \cup \mathbf{nil}\}_{closed}$.

As another example, the constructor $\{[\text{“}x\text{”}] = 1, [\text{“}y\text{”}] = \{[\text{“}z\text{”}] = 2\}\}$ has type $\{\text{“}x\text{”} : 1, \text{“}y\text{”} : \{\text{“}z\text{”} : 2\}_{fixed}\}_{unique}$ through rule T-CONSTRUCTOR1. The inner table is *fixed* 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 3.4 we presented the following example:

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

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

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

In this example, we assign the type $\{\}_{unique}$ to the variable $person$, then we refine its type to $\{\text{“}firstname\text{”} : \mathbf{string}\}_{unique}$, and then we refine its type to $\{\text{“}firstname\text{”} : \mathbf{string}, \text{“}lastname\text{”} : \mathbf{string}\}_{unique}$. Rule T-REFINE1 type checks this use of refinement:

$$\frac{\begin{array}{c}\text{(T-REFINE1)}\\ \Gamma_1(id) = \{\overline{F{:}V}\}_{unique}\\ \Gamma_1, \Pi \vdash e : F_{new}, \Gamma_2 \quad \nexists i \in 1..n\ F_{new} \lesssim F_i \quad V_{new} = vt(F_{new}, V) \quad n = |\overline{F{:}V}|\end{array}}{\Gamma_1, \Pi \vdash id[e]{<}V{>} : V_{new}, \Gamma_2[id \mapsto \{\overline{F{:}V}, F_{new}{:}V_{new}\}_{unique}]}$$

The rule for refining *open* table types is similar, changing only the tag in the type of id:

$$\frac{\begin{array}{c}\text{(T-REFINE2)}\\ \Gamma_1(id) = \{\overline{F{:}V}\}_{open}\\ \Gamma_1, \Pi \vdash e : F_{new}, \Gamma_2 \quad \nexists i \in 1..n\ F_{new} \lesssim F_i \quad V_{new} = vt(F_{new}, V) \quad n = |\overline{F{:}V}|\end{array}}{\Gamma_1, \Pi \vdash id[e]{<}V{>} : V_{new}, \Gamma_2[id \mapsto \{\overline{F{:}V}, F_{new}{:}V_{new}\}_{open}]}$$

Our type system also includes analogous rules for adding methods to *unique* and *open* tables as a side-effect of type checking a method declaration, but we will not discuss them in this section for brevity.

We use the refinement of table types to handle the declaration of new global variables. In Lua, the assignment `v = v + 1` translates to the statement `_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,

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

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

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,

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

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 neither *fixed* nor *closed*. For instance,

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

refines the type of $\_ENV$ from $\{\}_{open}$ to $\{\text{“}x\text{”} : \{\}_{fixed}\}_{open}$. 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:

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

This example type checks because all the environment changes are consistent, and $\text{“}foo\text{”} \times 1 \times \mathbf{nil}* \lesssim \mathbf{string} \times \mathbf{integer} \times \mathbf{value}*$. By consistent we mean that we are only adding new fields. More precisely, the first coercion expression refines the type of $\_ENV$ to $\{\text{“}x\text{”} : \mathbf{string}\}_{open}$, while the second coercion expression refines the type of $\_ENV$ to $\{\text{“}y\text{”} : \mathbf{integer}\}_{open}$. Merging the two yields $\{\text{“}x\text{”} : \mathbf{string}, \text{“}y\text{”} : \mathbf{integer}\}_{open}$. Nevertheless, the next example does not type check because it tries to add the same field to $\_ENV$,

but with different types:

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

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 3.7 and 4.2, 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,

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

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:

$$\frac{\begin{array}{c}\text{(T-IDREAD1)} \\ \Gamma_1(id) = F\end{array}}{\Gamma_1, \Pi \vdash id : close(F), \Gamma_1[id \mapsto open(F)]}$$

$$\frac{\begin{array}{c}\text{(T-IDREAD2)} \\ \Gamma_1(id) = F\end{array}}{\Gamma_1, \Pi \vdash id : fix(F), \Gamma_1[id \mapsto fix(F)]}$$

We do not need to close *unique* and *open* tables that appear in the left-hand side of assignments, because T-IDREAD1 and T-IDREAD2 are

sufficient to forbid the creation of an alias to another *unique* or *open* table. For this reason, identifiers that appear in the left-hand side of assignments have their own rule T-IDWRITE1:

$$\frac{\text{(T-IDWRITE1)}\quad \Gamma_1(id) = F}{\Gamma_1, \Pi \vdash id_l : F, \Gamma_1}$$

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. These rules also use the auxiliary function *rconst* to strip the **const** type from the type of the value, if present. We define these rules as follows:

$$\frac{\text{(T-INDEXREAD1)}\quad \Gamma_1(id) = \{\overline{F{:}V}\} \quad \Gamma_1, \Pi \vdash e_2 : F, \Gamma_2 \quad \exists i \in 1..n \; F \lesssim F_i \quad n = |\overline{F{:}V}|}{\Gamma_1, \Pi \vdash id[e_2] : rconst(V_i), \Gamma_2}$$

$$\frac{\text{(T-INDEXREAD2)}\quad \Gamma_1, \Pi \vdash e_1 : \{\overline{F{:}V}\}, \Gamma_2 \quad \Gamma_2, \Pi \vdash e_2 : F, \Gamma_3 \quad \exists i \in 1..n \; F \lesssim F_i \quad n = |\overline{F{:}V}|}{\Gamma_1, \Pi \vdash e_1[e_2] : rconst(V_i), \Gamma_3}$$

Rule T-INDEXREAD1 defines the case where using an identifier to index a table does not create an alias, while rule T-INDEXREAD2 defines the case for indexing expressions where the expression denoting the table is not an identifier. The rules for indexing left-hand values are similar to these rules, except that they ensure that the field is not **const**.

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 $<F>\ id$. For instance, we can use this type coercion expression to make the following example type check:

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

We can use a to initialize b because the coercion converts the type of a from $\{\text{“}x\text{”} : \textbf{string}, \text{“}y\text{”} : \textbf{string}\}_{unique}$ to $\{\text{“}x\text{”} : \textbf{string}, \text{“}y\text{”} : \textbf{string} \cup \textbf{nil}\}_{open}$, and results in $\{\text{“}x\text{”} : \textbf{string}, \text{“}y\text{”} : \textbf{string} \cup \textbf{nil}\}_{closed}$, which is a subtype of $\{\text{“}x\text{”} : \textbf{string}, \text{“}y\text{”} : \textbf{string} \cup \textbf{nil}\}_{closed}$, 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 $\{\text{“}x\text{”} : \mathbf{string}, \text{“}y\text{”} : \mathbf{string} \cup \mathbf{nil}, \text{“}z\text{”} : \mathbf{integer}\}_{open}$.

Rules T-COERCE1 and T-COERCE2 define the behavior of the type coercion expression:

$$\frac{\Gamma_1(id) <: F \quad tag(F, closed)}{\Gamma_1, \Pi \vdash <F> id : F, \Gamma_1[id \mapsto reopen(F)]} \text{(T-COERCE1)} \qquad \frac{\Gamma_1(id) <: F \quad tag(F, fixed)}{\Gamma_1, \Pi \vdash <F> id : F, \Gamma_1[id \mapsto F]} \text{(T-COERCE2)}$$

Note that the coercion rules only allow 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. If the coercion is to a *closed* table type the type of the table changes to an *open* table type with the same shape, but if the coercion is to a *fixed* table type the table has to assume the same type.

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 *closeall* closes all *unique* and *open* table types. The function *closeset* closes a given set of free assigned variables, which is given by the function *fav*. The function *openset* changes from *unique* to *open* a given set of referenced variables, which is given by the function *frv*.

As an example,

$$\begin{array}{l}\mathbf{local}\ a : \{\}_{unique}, b : \{\}_{unique} = \{\}, \{\}\ \mathbf{in} \\ \quad \mathbf{local}\ f : \mathbf{integer} \times \mathbf{nil}* \to \mathbf{integer} \times \mathbf{nil}* = \\ \quad\quad \mathbf{fun}\ (x : \mathbf{integer}) : \mathbf{integer} \times \mathbf{nil}* \\ \quad\quad\quad b = a\ ;\ \mathbf{return}\ x + 1 \\ \quad \mathbf{in}\ a[\text{“}x\text{”}] <\mathbf{integer}> = 1\ ;\ b[\text{“}x\text{”}] <\mathbf{string}> = \text{“}foo\text{”}\ ;\ \lfloor f(a[\text{“}x\text{”}])\rfloor_0\end{array}$$

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 *closeall* 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 *closeset*. Then, the function *frv* 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 types non-variadic function declarations, and it illustrates this case:

$$\frac{\begin{array}{c}\text{(T-FUNCTION1)}\\ closeall(\Gamma_1)[\overline{id \mapsto F}], \Pi[\rho \mapsto S] \vdash s, \Gamma_2 \\ \Gamma_3 = openset(\Gamma_1, frv(\textbf{fun}\ (\overline{id{:}F}){:}S\ s)) \\ \Gamma_4 = closeset(\Gamma_3, fav(\textbf{fun}\ (\overline{id{:}F}){:}S\ s))\end{array}}{\Gamma_1, \Pi \vdash \textbf{fun}\ (\overline{id{:}F}){:}S\ s : F_1 \times ... \times F_n \times \textbf{nil}* \to S, \Gamma_4}$$

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:

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

The rules for declaring variadic functions and recursive functions are similar to T-FUNCTION1, and we did not discuss them in this section for brevity.

Projections

Lua programmers often overload the return type of functions to denote errors, returning `nil` and an error message in case of error instead of the usual return values, 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. As we mentioned in Section 3.3, *idiv* performs integer division and has type

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

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 $\textbf{value}* \to \textbf{nil}*$. Let us also assume that a and b are local variables in the environment, and that both have type **integer**. Let us see how our type system type checks the following program:

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

First, our type system uses the auxiliary relation $\Gamma_1, \Pi \vdash el : E, \Gamma_2, (x, S)$ for type checking $idiv(a, b)$. 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 E and produces a new type environment Γ_2 and produces a pair (x, S). This pair means that the last expression of an expression list *el* produces an union of second-level types S that should be bound to a fresh 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-EXPLIST3 for type checking $idiv(a, b)$:

$$\text{(T-EXPLIST3)}$$
$$\frac{\begin{array}{c}\Gamma_1, \Pi \vdash e_i : F_i, \Gamma_{i+1} \quad \Gamma_1, \Pi \vdash me : S, \Gamma_{n+2} \\ S = F_{n+1} \times ... \times F_{n+m} \times \mathbf{nil}* \sqcup F'_{n+1} \times ... \times F'_{n+m} \times \mathbf{nil}* \\ \Gamma_f = merge(\Gamma_1, ..., \Gamma_{n+2}) \quad n = \mid \overline{e} \mid\end{array}}{\Gamma_1, \Pi \vdash \overline{e}, me : F_1 \times ... \times F_n \times \pi_1^x \times ... \times \pi_m^x \times \mathbf{nil}*, \Gamma_f, (x, S)}$$

Note that $idiv(a, b)$ 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) to bound a union of second-level types S 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:

$$\text{(T-LOCAL2)}$$
$$\frac{\begin{array}{c}\Gamma_1, \Pi \vdash el : E, \Gamma_2, (x, S) \\ \Gamma_3 = \Gamma_2[id_1 \mapsto infer(E, 1), ..., id_n \mapsto infer(E, n)] \\ \Gamma_3, \Pi[x \mapsto S] \vdash s, \Gamma_4 \quad n = \mid \overline{id} \mid\end{array}}{\Gamma_1, \Pi \vdash \mathbf{local}\ \overline{id} = el\ \mathbf{in}\ s, (\Gamma_4 - \{\overline{id}\})[\overline{id \mapsto \Gamma_2(id)}]}$$

This rule uses the auxiliary function *infer* to get the most general 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_4 without the variables that it introduced before type checking s. We can define *infer* as follows:

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

$$
\begin{aligned}
general(\textbf{false}) &= \textbf{boolean} \\
general(\textbf{true}) &= \textbf{boolean} \\
general(int) &= \textbf{integer} \\
general(float) &= \textbf{number} \\
general(string) &= \textbf{string} \\
general(F_1 \cup F_2) &= general(F_1) \cup general(F_2) \\
general(S_1 \to S_2) &= general2(S_1) \to general2(S_2) \\
general(\{F_1{:}V_1, ..., F_n{:}V_n\}_{tag}) &= \{F_1{:}general(V_1), ..., F_n{:}general(V_n)\}_{tag} \\
general(\mu x.F) &= \mu x.general(F) \\
general(T) &= T
\end{aligned}
$$

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

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 $\textbf{integer} \cup \textbf{nil}$, while reading r will use the projection type π_2^x to project the type of r into the union type $\textbf{integer} \cup \textbf{string}$. Our type system defines this behavior through rule T-IDREAD4, which uses the auxiliary function *proj* to project an union of second-level types into an union of first-level types:

$$
\frac{\Gamma_1(id) = \pi_i^x}{\Gamma_1, \Pi \vdash id : proj(\Pi(x), i), \Gamma_1} \text{(T-IDREAD4)}
$$

Now, we may want to discriminate q and r 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, as they are a general way to not compromise the dependency between the types of q and r after discriminating one of them, so flow typing can narrow the type of both variables by testing just one of them because the projection types of both variables bound to the same projection variable.

The rule T-IF5 shows the case where our type system discriminates a projection type based on the tag `nil`. It uses the auxiliary functions *fopt* and *fipt* to filter a projection x, affecting all variables that bind to the same projection. More precisely, the former function filters out the tuples that

contain a type F in the i-th component, while the latter function filters out the tuples that do not contain F in the i-th component. We define T-IF5 as follows:

$$\text{(T-IF5)}$$
$$\frac{\begin{array}{c}\Gamma_1(id) = \pi_i^x \\ S_t = fopt(\Pi(x), \mathbf{nil}, i) \quad S_e = fipt(\Pi(x), \mathbf{nil}, i) \\ \Gamma_1, \Pi[x \mapsto S_t] \vdash s_1, \Gamma_2 \\ \Gamma_1, \Pi[x \mapsto S_e] \vdash s_2, \Gamma_3 \\ \Gamma_4 = join(\Gamma_2, \Gamma_3)\end{array}}{\Gamma_1, \Pi \vdash \mathbf{if}\ id\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \Gamma_4}$$

Our previous example type checks through rule T-IF5, because it uses the information provided by the projection type π_1^x, which is the type of q, to make the rule T-IF5 use the function call

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

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

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

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 $\mathbf{integer} \cup \mathbf{nil}$ and $\mathbf{integer} \cup \mathbf{string}$, respectively.

Our type system also includes rules that check whether a branch is unreachable. Rules T-IF6 and T-IF7 respectively cover the case where the **else** branch is unreachable and the case where the **then** branch is unreachable, because either the projected type of π_i^x is not a supertype of **nil** or it is **nil**. We define these rules as follows:

$$\text{(T-IF6)}$$
$$\frac{\begin{array}{c}\Gamma_1(id) = \pi_i^x \\ S_t = fopt(\Pi(x), \mathbf{nil}, i) \\ fit(proj(\Pi(x), i), \mathbf{nil}) = \mathbf{void} \\ \Gamma_1, \Pi[x \mapsto S_t] \vdash s_1, \Gamma_2\end{array}}{\Gamma_1, \Pi \vdash \mathbf{if}\ id\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \Gamma_2}$$

$$\text{(T-IF7)}$$
$$\frac{\begin{array}{c}\Gamma_1(id) = \pi_i^x \\ S_e = fipt(\Pi(x), \mathbf{nil}, i) \\ fot(proj(\Pi(x), i), \mathbf{nil}) = \mathbf{void} \\ \Gamma_1, \Pi[x \mapsto S_e] \vdash s_2, \Gamma_2\end{array}}{\Gamma_1, \Pi \vdash \mathbf{if}\ id\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \Gamma_2}$$

Typed Lua does not allow assignments to left-hand values that are bound to a projection type. This kind of assignment would be unsound, because it

could break the dependency relation that the components of each tuple of the union have. For instance, the following example does not type check:

$$\begin{array}{l}\textbf{local } q, r = idiv(a, b) \textbf{ in} \\ \quad r = \text{“}foo\text{”}; \\ \quad \textbf{if } q \textbf{ then } \lfloor print(q + r) \rfloor_0 \textbf{ else } \lfloor print(\text{“}ERROR: \text{”} \mathbin{..} r) \rfloor_0\end{array}$$

In this example, the projected type of r outside of the **if** statement is **integer** $\cup$ **string**, so the assignment looks fine. However, the projected type of r inside the **if** branch is **integer**, not matching the **string** value that r has after the assignment.