A Typed Lua Calculus

Fabien Fleutot

February 16, 2013

This article proposes a sound type system for the Lua programming language. It's intended to be later combined with gradual typing and partial type inference, so that users can blend statically and dynamically typed program fragments, as best suits their development needs.

Lua [**?**] is a dynamic, imperative programming language, similar in expressiveness to other modern languages such as Python, Ruby or Javascript; compared to these, Lua specifically shines by its embeddability, frugality in terms of hardware resources, tight integration with C; its speed performances are also noteworthy [**?**], especially when run through a JIT compiler [**?**]. Finally, there is, by design, a proper subset of the language which is widely acknowledged as beginner-friendly [**?**]. Due to those qualities, Lua is widely used in domains such as high-performance video games, embedded devices or highly user-customizable systems.

Static typing is not always beginner-friendly; however, a well designed, statically typed third party API is generally easier to use, because many usage mistakes can be caught sooner, either when compiling or immediately by a type-checking IDE. While introducing a mandatory static type system in Lua would ruin several of the language's key features, supporting optional types would significantly improve the experience of many users.

We aim at offering such a fine-grained integration of static and dynamic program fragments, by building upon the research on gradual typing [**?**]. As a first step, we propose a type system, inspired by theoretical studies of records and objects typing [**?**, **?**], which accepts a significant proportion of idiomatic Lua programs.

A preliminary, necessary step is the definition of a formal calculus capturing the key semantic characteristics of Lua. Then we will propose a type system for this calculus, which allows to check a properly annotated program against illegal operations. We'll then hint at ways to integrate gradual typing in this system, and partial inference to lighten the amount of necessary annotations.

1 Calculus

This section defines a calculus, intended to capture Lua's defining features. A compromise shall be found between faithfulness to Lua, simplicity of the

semantic rules, and ability to support a type system.

1.1 Notations

- $\Sigma[x \leftarrow y]$, with Σ a mathematical function, denotes the function which to x associates y, and to all other values x' associates $\Sigma(x')$ if defined.
- $E[v \leftarrow E']$, with E and E' terms, and v a variable possibly occurring in E, denotes the term E in which all occurrences of v are replaced with E'.
- If E denotes an element of a given kind, $\bar{E}$ denotes a sequence of such elements. A sequence has zero, one or several elements, which are ordered and not necessarily unique.
- $\varnothing$ denotes an empty sequence. When it enhances understanding, it can be subscripted with the type of elements the sequence might have contained. For instance, an empty sequence of expressions might be denoted either $\varnothing$ or $\varnothing_E$.
- $(x_n)^{\forall n \in [1 \ldots m]}$ denotes the sequence of all elements x_n for successive values of n $1, 2, ..., m$. Boundaries 1 and m are inclusive.
- Sequences are concatenated with a semicolon between parentheses: $(\bar{E}_1; \bar{E}_2)$. They can also be concatenated with single elements: $(E; \bar{E})$.

1.2 Terms definition

In a first step, we'll introduce a calculus which captures Lua's defining features, such as tables, functions, multi-value returns, local variables. We will not deal with classic structures (if/then/else statements, for loops etc.) which can be added later in a rather straightforward way.

We also won't explicitly support features which can be easily encoded. "`...`" trailing function arguments, method invocations, globals, combined local variable declaration+assignment are intentionally left out of the calculus.

We'll also consider it mandatory for function bodies to end with a `return nil`, to save a specific rule about non-returning functions (the Lua compiler performs this transformation for the same reason).

Finally, we distinguish function applications as expressions $f(\bar{x})_E$ and as statements $f(\bar{x})_S$: the latter discard whatever value they might have returned. This distinction is trivial to do when encoding a program into the calculus, and simplifies the calculus' semantic rules.

$$
\begin{array}{rcll}
E & ::= & L & (\textit{E-Left}) \\
 & | & P & (\textit{E-Primitive}) \\
 & | & E(\bar{E})_E & (\textit{E-Apply}) \\
 & | & \texttt{function(}\bar{v}\texttt{)}\ \bar{S}\ \texttt{end} & (\textit{E-Function}) \\
 & | & \{\ ([E^k_n] = E^v_n)^{\forall n \in [1 \ldots m]}\ \} & (\textit{E-Table}) \\
 & & & \\
L & ::= & v & (\textit{E-Variable}) \\
 & | & E[E] & (\textit{E-Index}) \\
 & & & \\
P & ::= & \langle \text{string} \rangle & (\textit{E-String}) \\
 & | & \langle \text{number} \rangle & (\textit{E-Number}) \\
 & | & \texttt{true} & (\textit{E-True}) \\
 & | & \texttt{false} & (\textit{E-False}) \\
 & | & \texttt{nil} & (\textit{E-Nil}) \\
 & | & t & (\textit{E-TableRef}) \\
 & | & f & (\textit{E-ClosureRef}) \\
 & & & \\
S & ::= & \texttt{local}\ \bar{v} & (\textit{S-Local}) \\
 & | & \bar{L} = \bar{E} & (\textit{S-Assign}) \\
 & | & E(\bar{E})_S & (\textit{S-Apply}) \\
 & | & \texttt{return}\ \bar{E} & (\textit{S-Return})
\end{array}
$$

Expression elements are sorted into categories:

- S denotes statements;
- E encompasses all expressions;
- P denotes primary expressions: expressions which evaluate to themselves, without side effect; we have $P \subset E$.
- L denotes left-hand-side values, i.e. values which can legally appear to the left of a $\bar{L} = \bar{E}$ assignment statement; we have $L \subset E$ and $L \cap P = \{\ \}$.

Table and function references t and f aren't part of Lua: they are what tables and functions are evaluated to, so that the notion of identity and mutation sharing in the calculus remain faithful to Lua.

1.3 Semantics

We'll now describe how calculus terms are evaluated. We'll do so in big-step ("natural") semantics, i.e. we give a proposition of the form "assumptions $\vdash$ term to evaluate $\Rightarrow$ modified assumptions, fully evaluated term". If no such reduction statement can be proved, then the term to evaluate is erroneous. This differs from small-step semantics, where one defines a single reduction step, and the evaluation of a program is defined as the repeated application of the reduction step until no more reduction is possible.

More formally, the proposition $\Sigma_0, X_0 \Rightarrow \Sigma_1, X_1$ reads "element X_0, when evaluated under environment Σ_0, reduces to X_1 and changes the environment into Σ_1". To clarify rules, this operator has been separated into $\overset{E}{\Rightarrow}, \overset{\bar{E}}{\Rightarrow}, \overset{S}{\Rightarrow}, \overset{\bar{S}}{\Rightarrow}, \overset{L}{\Rightarrow}, \overset{\bar{L}}{\Rightarrow}$ depending on the kind of terms on which it operates.

Rules will be presented the usual way in logic: as fraction bars, with the premises over the bar, and the proved conclusion under it. $\frac{a \;\; b \;\; c}{d}$ means that proposition d is proved if we can provide a proof of propositions a, b and c (the "premises"). A rule with nothing above the line, such as $\frac{}{d}$, is an axiom: a proposition considered true with no need for any further proof. A complete proof is therefore a tree of propositions, with the goal proposition as the tree's root, and axioms as its leaves.

Identity of tables and closures In Lua, tables and functions have an identity, i.e. two structurally equal tables are not equal, unless they're a shared reference to a same local variable (or table element) holding the same value. For instance, in "`a={ }; b={ }`", `a` and `b` are not equal. Similarly, extensionally equal functions are not equal in the Lua sense.

To reflect this in the calculus, we'll let (*E-Function*) and (*E-Table*) terms evaluate to a fresh reference f or t every time they're evaluated; the association between the reference and its definition will be kept in the environment store Σ. As a result, equality works in the calculus as in Lua; for instance, in "`a={ }; b=a`", `a` and `b` are indeed equal.

Moreover, in order to make mutable variables and first class functions cohabit seamlessly, Lua defines up-values, i.e. local variables which outlive their syntactical scope. To faithfully represent this in the calculus, we'll perform an α-renaming to fresh variable names every time we encounter a "`local` $\bar{v}$" statement. This way, all variables live forever in the calculus, although only up-values might be referred to out of their scope. Put otherwise, the formal calculus never performs garbage collection.

Program environment The program's environment, i.e. the assumptions under which a term is evaluated, is represented by a triplet of functions $(\Sigma^L, \Sigma^T, \Sigma^F)$. They keep track respectively of local variables' content, tables' content and closures. Since in most cases the details of the environment don't matter, the triplet is often shortened as Σ.

Formally, the three functions have the following types:

- $\Sigma^L : v \mapsto P$ is a function from variable names to the evaluated expressions they hold;
- $\Sigma^T : (t \times P) \mapsto P$ is a function from table references and key expressions, to the value expressions held under this key in this table;
- $\Sigma^F : f \mapsto (\bar{v} \times \bar{S})$ is a function from function references f to function definitions `function(`$\bar{v}$`)` $\bar{S}$ `end`.

The environment could also be used capture I/O operations, to describe side effects other than variable value changes. This wouldn't significantly change the calculus' semantic properties, though, and hence will be left out.

Assignment to environment To properly define variable assignment, we'll need an operator $\Leftarrow$ on environments, which denotes the update of what's bound to variables and table fields in Σ. Indeed, an assignment statement can update a mix of local variables and table contents, so we need to describe the simultaneous update of Σ^L and Σ^T. The operator $(\Sigma^L, \Sigma^T) \Leftarrow (\bar{L}, \bar{P})$ is defined inductively as follows:

$$\begin{array}{rcll}
(\Sigma^L, \Sigma^T) \Leftarrow (\varnothing_L, \bar{P}) & = & (\Sigma^L, \Sigma^T) & (A\text{-}\varnothing_L) \\
(\Sigma^L, \Sigma^T) \Leftarrow (\bar{L}, \varnothing_P) & = & (\Sigma^L, \Sigma^T) \Leftarrow (\bar{L}, \texttt{nil}) & (A\text{-}\varnothing_P) \\
(\Sigma^L, \Sigma^T) \Leftarrow ((v; \bar{L}), (P; \bar{P})) & = & (\Sigma^L[v \leftarrow P], \Sigma^T) \Leftarrow (\bar{L}; \bar{P}) & (A\text{-}Local) \\
(\Sigma^L, \Sigma^T) \Leftarrow (t[P_k], \bar{L}; P_v, \bar{P}) & = & (\Sigma^L, \Sigma^T[(t, P_k) \leftarrow P_v]) \Leftarrow (\bar{L}; \bar{P}) & (A\text{-}Table)
\end{array}$$

Intuitively, $\Leftarrow$ stores variable assignments in Σ^L and table writings in Σ^T, thanks to the two last rules (the two first ones are structural). It expects its third argument to be a list of left-values, i.e. either variables or indexing of a primitive term by another. As a notation facility, we'll allow to transparently pass an extra Σ^F argument to $\Leftarrow$. This lets use it directly on complete environments Σ. Formally, the operator is overloaded as follows:

$$(\Sigma^L, \Sigma^T, \Sigma^F) \Leftarrow (\bar{L}, \bar{P}) = (\Sigma^L_\star, \Sigma^T_\star, \Sigma^F) \text{ iff } (\Sigma^L, \Sigma^T) \Leftarrow (\bar{L}, \bar{P}) = (\Sigma^L_\star, \Sigma^T_\star)$$

.

TODO: *forbid* $E_L[\texttt{nil}] = E_R$

Statements sequences evaluation $\varnothing_S$ denotes an empty sequence of statements. It is also the result of a sequence which didn't return anything:

$$\frac{}{\Sigma, \varnothing_S \overset{\bar{S}}{\Rightarrow} \Sigma, \varnothing_S} \quad (ES\text{-}\varnothing)$$

If the first element of a sequence doesn't evaluate into a `return`, then the result of the sequence is that of the following statements (plus any side effect caused on Σ by the first statement):

$$\frac{\Sigma, S^1 \overset{S}{\Rightarrow} \Sigma_1, \varnothing \quad \Sigma_1, \bar{S} \overset{\bar{S}}{\Rightarrow} \Sigma_2, S^\star}{\Sigma, (S^1; \bar{S}) \overset{\bar{S}}{\Rightarrow} \Sigma_2, S^\star} \quad (ES\text{-}\bar{\varnothing})$$

However, if a statement evaluates to `return`, the rest of the sequence isn't evaluated:

$$\frac{\Sigma, S^1 \overset{S}{\Rightarrow} \Sigma_1, \texttt{return } \bar{P}}{\Sigma, (S^1; \bar{S}) \overset{\bar{S}}{\Rightarrow} \Sigma_1, \texttt{return } \bar{P}} \quad (ES\text{-}\overline{\texttt{return}})$$

Statements

Return statements Return statements evaluate their returning values. When a function body evaluates to `return` $\bar{P}$, it will be "unwrapped" back into a $\bar{P}$ by (*EE-Apply*).

$$\frac{\Sigma, \bar{E} \overset{\bar{E}}{\Rightarrow} \Sigma_1, \bar{P}}{\Sigma, \texttt{return}\ \bar{E} \overset{S}{\Rightarrow} \Sigma_1, \texttt{return}\ \bar{P}} \quad (\textit{ES-Return})$$

Local variables creation Local variable creations are immediately α-renamed:

$$\frac{\left(\Sigma^L\left[\bar{w} \leftarrow \overline{\texttt{nil}}\right], \Sigma^T, \Sigma^F\right), \bar{S}[\bar{v} \leftarrow \bar{w}] \overset{\bar{S}}{\Rightarrow} \Sigma_1, S^\star \quad \bar{w} \text{ free in } \Sigma^L}{(\Sigma^L, \Sigma^T, \Sigma^F), (\texttt{local}\ \bar{v}; \bar{S}) \overset{\bar{S}}{\Rightarrow} \Sigma_1, S^\star} \quad (\textit{ES-Local})$$

The point is to handle upvalues (references to variables defined outside of the function body) correctly. Consider for instance the following program, featuring an up-value `u`:

```
local u=1
local f = function(x)
  u=u+1
  return x+u
end
_ENV["a"], _ENV["b"] = f(1), f(2)
```

The first line `local u` will be evaluated only once, and therefore occurrences of `u` in `f` will all be α-renamed to the same fresh variable: they will indeed be shared as expected. Conversely, in the following program:

```
local f = function(x)
  local u; u=1
  u=u+1
  return x+u
end
_ENV["a"], _ENV["b"] = f(1), f(2)
```

The `local u` statement will be evaluated twice, each time being renamed in a different fresh variable, and no sharing can occur.

Evaluation of assignment contains a subtlety: the value-receiving fields and variables, on the left of the "=" sign, must not be fully evaluated. Instead, they need to be reduced to a "left-normal" form (variables or index to a table), which is done by a distinct evaluation operator $\overset{\bar{L}}{\Rightarrow}$; once both left and right sides of "=" are evaluated, modifying the environment adequately is left to the $\Leftarrow$ operator defined above:

$$\frac{\Sigma, \bar{L} \overset{\bar{L}}{\Rightarrow} \Sigma_1, \bar{L}^\star \qquad \Sigma_1, \bar{E} \overset{\bar{E}}{\Rightarrow} \Sigma_2, \bar{P}}{\Sigma_1, \bar{L} = \bar{E} \overset{S}{\Rightarrow} (\Sigma_2 \Leftarrow (\bar{L}^\star, \bar{P})), \varnothing_S} \quad (\textit{ES-Assign})$$

We'll define $\overset{\bar{L}}{\Rightarrow}$ in terms of $\overset{L}{\Rightarrow}$, which operates on a single expression. Variables are considered fully evaluated when the occur on the left of "=":

$$\frac{}{\Sigma, v \overset{L}{\Rightarrow} \Sigma, v} (\textit{EL-v})$$

Indexed values have the table and its key evaluated, but the field-content-accessing operation isn't performed:

$$\frac{\Sigma, E_T \overset{E}{\Rightarrow} \Sigma_1, (P_T^1; \bar{P}_T) \qquad \Sigma_1, E_k \overset{E}{\Rightarrow} \Sigma_2, (P_k^1; \bar{P}_k)}{\Sigma, E_T[E_k] \overset{L}{\Rightarrow} \Sigma_2, P_T^1[P_k^1]} (\textit{EL-Index})$$

With this we can easily define $\overset{\bar{L}}{\Rightarrow}$, which chains $\overset{L}{\Rightarrow}$ operations by stringing their environment modifications together:

$$\frac{\Sigma, L_1 \overset{L}{\Rightarrow} \Sigma_1, L_1^\star \qquad \Sigma_1, \bar{L} \overset{\bar{L}}{\Rightarrow} \Sigma_2, \bar{L}^\star}{\Sigma, (L_1; \bar{L}) \overset{\bar{L}}{\Rightarrow} \Sigma_2, (L_1^\star; \bar{L}^\star)} (\textit{EL-}\bar{L}) \qquad \frac{}{\Sigma, \varnothing_L \overset{\bar{L}}{\Rightarrow} \Sigma, \varnothing_L} (\textit{EL-}\varnothing)$$

Function applications in statment contexts Function application in a statement context is pretty similar to function application in an expression context, except that any returned result is thrown out. The actual β-reduction is therefore delegated to (*EE-Apply*), defined later:

$$\frac{\Sigma, E_f(\bar{E})_E \overset{E}{\Rightarrow} \Sigma_1, \bar{P}}{\Sigma, E_f(\bar{E})_S \overset{S}{\Rightarrow} \Sigma_1, \varnothing_S} \quad (\textit{ES-Apply})$$

Expression sequences In Lua, when a single expression evaluates into several results, only the first result is kept, except for the last one which is entirely appended to the resulting multi-value. For instance, if we consider `f = function(a,b,c) return a,b,c end`, the sequence `f(10,11,12), f(20,21,22), f(30,31,32)` will evaluate to `10, 20, 30, 31, 32`.

The rule below chains expression evaluations by stringing their environment modifications together, and discards extraneous returned values:

$$\frac{(\forall n \in [1...m])\ \Sigma_{n-1}, E_n \overset{E}{\Rightarrow} \Sigma_n, (P_n^1; \bar{P}_n)}{\Sigma_0, (E_n)^{\forall n \in [1...m]} \overset{\bar{E}}{\Rightarrow} \Sigma_m, ((P_n^1)^{\forall n \in [1...m]}; \bar{P}_m)} \quad (\textit{EE-Sequence})$$

Expressions

Variables and primitives Variables are replaced by their content from the store; primitives are their own evaluation:

$$\frac{\Sigma^L(v) = P}{(\Sigma^L, \Sigma^T, \Sigma^F), v \overset{E}{\Rightarrow} P} \quad (EE\text{-}v) \qquad \frac{}{\Sigma, P \overset{E}{\Rightarrow} \Sigma, P} \quad (EE\text{-}Primitive)$$

Function application For function applications, function definitions are retrieved from Σ^F. We define the evaluation by transforming the function parameters into local variables, assigned to the arguments' values. Notice that the arguments are evaluated before their assignment, although (*EE-Assign*) would have evaluated them anyway. The reason is, the arguments need to be evaluated outside of the function's scope; otherwise, capture problems could occur (think for instance of `f=function(x) return x end; local x; f(x)`):

$$\frac{\begin{array}{c}\Sigma, E_f \overset{E}{\Rightarrow} (\Sigma_1^L, \Sigma_1^T, \Sigma_1^F), f \\ \Sigma_1^F(f) = \texttt{function(}\bar{v}\texttt{)}\ \ \bar{S}\ \ \texttt{end} \\ (\Sigma_1^L, \Sigma_1^T, \Sigma_1^F), \bar{E} \overset{\bar{E}}{\Rightarrow} \Sigma_2, \bar{P} \\ \Sigma_2, (\texttt{local}\ \ \bar{v}; \bar{v} = \bar{P}; \bar{S}) \overset{\bar{S}}{\Rightarrow} \Sigma_3, \texttt{return}\ \ \bar{P}\end{array}}{\Sigma, E_f(\bar{E})_E \overset{E}{\Rightarrow} \Sigma_3, \bar{P}} \quad (EE\text{-}Apply)$$

Function creations As seen in (*EE-Apply*), function definitions are retrieved from Σ^F. They're stored in it, under a fresh name f, when the `function ... end` expression is found:

$$\frac{E_f = \texttt{function(}\bar{v}\texttt{)}\ \ \bar{S}\ \ \texttt{end} \qquad f \text{ free in } \Sigma^F}{(\Sigma^L, \Sigma^T, \Sigma^F), E_f \overset{E}{\Rightarrow} (\Sigma^L, \Sigma^T, \Sigma^F[f \leftarrow E_f]), f} \quad (EE\text{-}Function)$$

Literal tables To evaluate a literal table, we evaluate every key and value in order, chaining environment modifications, then store the (key, value) evaluated pairs in Σ^T:

$$\frac{\begin{array}{c}t \text{ free in } \Sigma_{2m}^T \\ (\forall n \in [1...m]) \left\{\begin{array}{l}\Sigma_{2n-2}, E_k^n \Rightarrow \Sigma_{2n-1}, (P_k^n; \bar{P}_k^n) \\ \Sigma_{2n-1}, E_v^n \Rightarrow \Sigma_{2n}\ \ , (P_v^n; \bar{P}_v^n)\end{array}\right. \\ \Sigma_\star^T = \Sigma_{2m}^T[(t, P_k^n) \leftarrow P_v^n]^{\forall n \in [1...m]}\end{array}}{\Sigma_0, ([E_k^n] = E_v^n)^{\forall n \in [1...m]} \overset{E}{\Rightarrow} (\Sigma_{2m}^L, \Sigma_\star^T, \Sigma_{2m}^F), t} \quad (EE\text{-}Table)$$

Accessing table contents When a value is indexed, it must evaluate to a table reference; then the value associated with the corresponding key is retrieved from the store Σ^T:

$$\frac{\Sigma, E_T \Rightarrow \Sigma_1, t \quad \Sigma_1, E_k \Rightarrow \Sigma_2, (P_k^1; \bar{P}_k) \quad \Sigma_2^T(t, P_k^1) = P_v}{\Sigma, E_T[E_k] \Rightarrow \Sigma_2, P_v} \quad (\textit{EE-Index})$$

In Lua, a key which has never been set in a table is associated with value `nil`. To reflect this, an evaluation must start with all table/key pairs associated with `nil`:

$$(\forall t)(\forall P)\ \Sigma^T(t, P) = \texttt{nil}$$

1.4 How evaluation can fail

The operational semantics above defines the result of programs, as long as they:

- don't get stuck in infinite recursion;
- don't try to index a non-table (cf. premises of (*EE-Index*), which is the only evaluation rule applying to terms of the form $E[E]$, and requires the indexed object to be a reference to a table);
- don't try to apply a non-function (cf. premises of (*EE-Apply*), which is the only evaluation rule applying to terms of the form $E(\bar{E})$, and requires the applied object to be a reference to a function;);
- always return a value from a function, i.e. all function bodies, when parameters are substituted with arguments, evaluate to a `return` $\bar{P}$ statement value. It's easily proved that by appending a `return nil` at the end of the function's body, a function body evaluating to $\varnothing_S$ will evaluate to `return nil` instead.

The last point is very easily addressed, and the first one is well known as undecidable; the most reasonable definition of static correctness, for a program, is to provably perform no indexing of a non-table value, and no function call on a non-function. A sound type system for the present calculus will provide formal proofs that a given term cannot involve such incorrect sub-terms. The design of such a type system is the subject of the next section.

2 Static type system

In the previous section, we've seen that we wanted a type system to prevent indexing of non-tables as well as application of non-functions. We've also mentioned that a type system which eliminates all incorrect terms will also eliminate some correct ones (a direct consequence of the calculus' Türing-completeness, easily demonstrated by encoding the λ-calculus in it). This section will try to find a reasonable compromise: a type system which accepts a lot of "reasonable" terms, catches all incorrect ones, and doesn't force too much book-keeping on users.

2.1 Notations

$<:$ denotes subtyping. $\mathbb{T}_1 <: \mathbb{T}_2$ means that $\mathbb{T}_1$ is a subtype of $\mathbb{T}_2$, i.e. a term of type $\mathbb{T}_1$ can be used everywhere a term of type $\mathbb{T}_2$ is expected.

2.2 Type System

$$
\begin{array}{rcll}
\mathbb{E} & ::= & [\overline{P : \mathbb{F}} | \mathbb{F}] & (\textit{TE-Table}) \\
& | & \mathbb{P} & (\textit{TE-Primitive}) \\
& | & \bar{\mathbb{E}} \to \bar{\mathbb{E}} & (\textit{TE-Function}) \\
& | & \top & (\textit{TE-Top}) \\
\\
\mathbb{P} & ::= & \texttt{nil} \mid \texttt{boolean} \mid \texttt{number} \mid \texttt{string} & \\
\\
\mathbb{F} & ::= & \texttt{just}\ \mathbb{E} & (\textit{TF-Just}) \\
& | & \texttt{currently}\ \mathbb{E} & (\textit{TF-Currently}) \\
& | & \texttt{var}\ \mathbb{E} & (\textit{TF-Var}) \\
& | & \texttt{const}\ \mathbb{E} & (\textit{TF-Const}) \\
& | & \texttt{field} & (\textit{TF-Field}) \\
\\
\mathbb{S} & ::= & \texttt{return}\ \bar{\mathbb{E}} & (\textit{TS-Return}) \\
& | & \varnothing_{\mathbb{S}} & (\textit{TS-None})
\end{array}
$$

Lua key features to respect The type system should capture as many Lua-specific idioms as possible. Tables are of course central in Lua, and quite similar in some respects to objects of calculi such as Abadi & Cardelli's [**?**], Fisher Honsell & Mitchell [**?**], or Rémy's [**?**]. Among others, they are defined with no native notion of classes. The most striking Lua-specific features are:

- Lua tables—and therefore Lua itself—are deeply imperative. There are primitives in Lua to alter a table, but neither to copy nor to functionally update it; despite closures and tail-call optimization, idiomatic Lua code is not functional. This contrasts with most theoretical studies of object-oriented calculi, which inherit from λ-calculus a preference for functional primitives and idioms.

- Lua tables are arbitrary value→value hashtables, whereas object calculi typically index object fields with labels taken from a separate (enumerable) set. In the first version of our type system, we'll only type tables whose keys have primitive types `string`, `number` or `boolean`. Homogeneously typed hashtables should be easy to add at a later stage—they mostly behave as simplified functions, type-wise; but arbitrarily mixed tables, acting as a raw mix of records and hashtables, are untypable in general.

- Lua makes a distinction between statements and expressions. This means that two distinct type kinds $\mathbb{S}$ and $\mathbb{E}$ are defined. This is again in contrast with most theoretical calculi, which are purely expression-based.

- There's no notion of "undefined label": all table keys except `nil` are defined in all tables; keys which haven't been explicitly assigned are associated with the value `nil`.
- A key consequence is that an table's type changes during its lifetime: when created all its fields have type `nil`, then these field types are modified as meaningful values are put in them. Whereas most calculi allow to add fields to object, ours will allow to change their type.
- Lua functions take multiple arguments, and return multiple values.
- Expressions can be collected into expression sequences, to be used in assignments, function calls and function results. They get their dedicated type kind $\bar{\mathbb{E}}$.

Beyond those specific needs, the type system will include many staples of modern type systems, such as structural subtyping, functions contravariant in their arguments and covariant in their results, covariant read-only table fields, invariant read+write fields.

Types will not be nullable (there won't be any legal way to derive `nil` $<:$ $\mathbb{E}$ for any $\mathbb{E}$ other than `nil`): catching "NPE" (*Null Pointer Exceptions*, as nicknamed by traumatized Java developers) has been done for ages in ML-inspired languages; Hoare himself, who first introduced implicitly nullable types in Algol, called them "*my billion dollar mistake*" [**?**]. However, since nullable function arguments and results are an important idiom in Lua, future versions of the calculus will have to support either an explicit `nullable` type modifier, or a generic union type, allowing to type e.g. an optional number as `nil|number`.

Tables Lua tables accept all values as keys except `nil`, and there's no notion of unset/undefined key; it's legal to request `foo["bar"]` even if the key `"bar"` has never been set in `foo`: it will return `nil`. So in practice, all but a finite set of keys in a given table will return a value other than `nil`. To reflect this in the type system, a table type will contain a default field type, shared by all but a finite number of its fields; this type will usually be set to some variant of `nil`. The other key/type pairs are listed explicitly. For instance, [`"x"` : `const number`|`const nil`] describes a table such as `{["x"]=1}`, with a field `"x"` of type `const number`, and all other fields left to `nil`.

In the current version, field keys are limited to values of atomic types `string`, `number` or `boolean`. Tables using other keys will not be typable statically. Some future extensions are possible, and will be studied separately.

In contrast with type systems inspired by Abadi & Cardelli's, we won't introduce a $\zeta(s)$ self-type binder in the type system [**?**]: it substantially complicates it, mostly to allow a functional-style use of objects that doesn't seem to correspond to any widespread Lua idiom.

Table types are considered equal modulo fields reordering, and expansion of the default field type; the following types are all considered equal:

$$
\begin{aligned}
& [P_1 : \mathbb{F}_1; P_2 : \mathbb{F}_2 | \mathbb{F}_d] \\
= & [P_2 : \mathbb{F}_2; P_1 : \mathbb{F}_1 | \mathbb{F}_d] && (\textit{reordering}) \\
= & [P_1 : \mathbb{F}_1; P_2 : \mathbb{F}_2; P_3 : \mathbb{F}_d | \mathbb{F}_d] && (\textit{default expansion})
\end{aligned}
$$

Moreover, it's illegal for a field key to appear more than once in the same table: $[P_1 : \mathbb{F}_1; P_1 : \mathbb{F}_2 | \mathbb{F}_d]$ is not a well-formed type.

Finally, we'll admit as a shortcut that $[\overline{P : \mathbb{F}}]$ means $[\overline{P : \mathbb{F}} | \texttt{field}]$.

Field types Field types are expression types with a prefix modifier: `just` $\mathbb{E}$, `currently` $\mathbb{E}$, `var` $\mathbb{E}$, `const` $\mathbb{E}$, and simply `field` without a type parameter. They give control over field variance, i.e. they prevent some operations, but in exchange allow more permissive subtyping, and hence allow to use tables in more contexts. `var`, `const` and `field` will be familiar to people who studied structural subtyping, and offer the expected variance properties. `just` $\mathbb{E}$ and `currently` $\mathbb{E}$ are more unusual, and allow to change a value's type for an unrelated one, under specific conditions.

Read-write fields `var` $\mathbb{E}$ is the type of a field which can be read and written with values of type $\mathbb{E}$. It's not covariant, i.e. even if $\mathbb{E}_1 <: \mathbb{E}_2$, we don't have $[P : \texttt{var}\ \mathbb{E}_1] <: [P : \texttt{var}\ \mathbb{E}_2]$. To see why, let's consider the subtyping relationship `positive` <: `number` [1]. If we had $[P : \texttt{var positive}] <: [P : \texttt{var number}]$, we could take a table of type $[P : \texttt{var positive}]$, partially forget its type through subtyping into $[P : \texttt{var number}]$, then write a negative number in its field P. Other parts of the program, which retained the more precise type $[P : \texttt{var positive}]$ for the same table, might break because they take for granted that P's content is positive.

Read-only fields If we promise not to overwrite a field, however, we can make it covariant. This is the purpose of `const` $\mathbb{E}$ fields: the type system will prevent from updating such fields, but in exchange, whenever we have $\mathbb{E}_1 <: \mathbb{E}_2$, we also get $\texttt{const}\ \mathbb{E}_1 <: \texttt{const}\ \mathbb{E}_2$. Adapting the previous example, $[P : \texttt{const positive}]$ is a subtype of $[P : \texttt{const number}]$, because when reading its P field, one gets a `positive` which is indeed a `number`; since we can't write in it, there's no danger of putting a negative number where the type system expects to find only positive ones.

Contravariant fields Symmetrically, we could promise not to read a field, and get contravariance in exchange (whenever $\mathbb{E}_1 <: \mathbb{E}_2$, we get $[P : \texttt{writeonly}\ \mathbb{E}_2] <: [P : \texttt{writeonly}\ \mathbb{E}_1]$). However, this seems of limited practical use, so we'll leave this out of the type system.

[1] Positive numbers are numbers, but not the other way around. This example of subtyping has been chosen because it's hopefully familiar and intuitive for everyone; but it's only intended to illustrate variance issues, and the calculus won't have a specific `positive` type.

All fields If we promise neither to read nor to write a given field, it becomes bivariant: whether $\mathbb{E}_1 <: \mathbb{E}_2$ or $\mathbb{E}_2 <: \mathbb{E}_1$, or even if $\mathbb{E}_1$ and $\mathbb{E}_2$ are incomparable, we'd still have `field` $\mathbb{E}_1 <:$ `field` $\mathbb{E}_2$. We'll therefore simply write it `field`: no need to keep $\mathbb{E}$ in it, since it isn't used anyway. It's the super-type of all other type fields, and it acts as the `private` modifier does in C++ inspired languages: you can neither override nor use a field of this type.

Type-changing field types `currently` $\mathbb{E}$ means that a field currently has type $\mathbb{E}$, but that this type can be changed without breaking the program. This is an unusually liberal typing rule, and as such, it will only be allowed under strictly controlled circumstances. Most notably, a table type with some `currently` fields will have to be used linearly: if several variables allowed access to the same `currently` field, one variable could change the field's content type without the other variable's knowledge, and break the program in unpredictable ways. Hence, it will be mandatory to weaken the type of `currently` $\mathbb{E}$ fields into `field`, before using them in non-linear ways, thus preventing both read and write operations on it.

`currently` $\mathbb{E}$ field types are intended to allow idioms such as "`x={ };x.f_1= E1; ...; x.f_n=En`"; the type of `x` in this program will change at each statement of this sequence, from [|`currently nil`] to [`"f_1"` : `currently` $\mathbb{E}_1$;...; `"f_n"` : `currently` $\mathbb{E}_n$|`currently nil`].

We mentioned a criterion of linearity: there must be at most one reference to a `currently` $\mathbb{E}$ field. Otherwise, one reference might change the type of the field's content without the other references' knowledge. The typing rules will be designed in such a way that whenever extra references to it are created, these will be typed as `field`, i.e. inaccessible.

Unreferenced field types Finally, we need a type indicating that an object is completely unreferrenced, and can therefore be stored safely into a `currently` $\mathbb{E}$ field. For instance, in `x={foo=1}`, if the right-hand-side table was typed [`"foo"` : `currently number`|`currently nil`], it would have to be weakened into [|`field`] before being stored in x, in case there was already a reference to it.

Therefore, we distinguisg `currently` $\mathbb{E}$ the type of a field referenced once, and `just` $\mathbb{E}$ the type of a field which isn't referenced at all. In the example above, the right-hand-side is typed [`"foo"` : `just number`|`just nil`], and weakened into [`"foo"` : `currently number`|`currently nil`] when stored into `x`. A further `y=x` statement would see the type stored in `y` weakened to [|`field`].

2.3 Subtyping rules

The subtyping relationship is a partial order, defined as the smallest transitive closure of the rules listed in this subsection.

Structural rules The subtyping relationship, defined over expression, field and statement types, is reflexive and transitive; $\top$ is the biggest expression type:

$$\frac{}{\mathbb{E} <: \top}(<: \top)$$

$$\frac{}{\mathbb{E} <: \mathbb{E}} \quad \frac{}{\mathbb{F} <: \mathbb{F}} \quad \frac{}{\mathbb{S} <: \mathbb{S}} \quad (<:\mathit{Refl})$$

$$\frac{\mathbb{E}_1 <: \mathbb{E}_2 \quad \mathbb{E}_2 <: \mathbb{E}_3}{\mathbb{E}_1 <: \mathbb{E}_3} \quad \frac{\mathbb{F}_1 <: \mathbb{F}_2 \quad \mathbb{F}_2 <: \mathbb{F}_3}{\mathbb{F}_1 <: \mathbb{F}_3} \quad \frac{\mathbb{S}_1 <: \mathbb{S}_2 \quad \mathbb{S}_2 <: \mathbb{S}_3}{\mathbb{S}_1 <: \mathbb{S}_3}$$

$$(<: \mathit{Trans})$$

Fields subtyping We have `var` $\mathbb{E}$ <: `const` $\mathbb{E}$, `const`'s covariance, and `field` the top field type:

$$\frac{}{\mathbb{F} <: \texttt{field}}(<:\mathit{Field})$$

$$\frac{}{\texttt{var}\ \mathbb{E} <: \texttt{const}\ \mathbb{E}}(<:\mathit{Const}) \qquad \frac{\mathbb{E}_1 <: \mathbb{E}_2}{\texttt{const}\ \mathbb{E}_1 <: \texttt{const}\ \mathbb{E}_2}(<:\mathit{Const}^+)$$

$$\frac{}{\texttt{just}\ \mathbb{E} <: \texttt{currently}\ \mathbb{E}}(<:\mathit{Currently}) \qquad \frac{}{\texttt{just}\ \mathbb{E} <: \texttt{var}\ \mathbb{E}}(<:\mathit{Var})$$

$$\frac{\mathbb{E}_1 <: \mathbb{E}_2}{\texttt{just}\ \mathbb{E}_1 <: \texttt{just}\ \mathbb{E}_2}(<:\mathit{Just}^+)$$

We do *not* have `currently` $\mathbb{E}$ <: `var` $\mathbb{E}$. Indeed, mutable field types are not a special case of variable fields: the latter can be used with less restrictions when linearity cannot be guaranteed. For instance, if `x` : [`var number`], its content can be assigned to `y` with "`x=y`", and `y` will also have type [`var number`]. However, if `x` had type [`currently number`], `y` would only get type [`field`], because the following statement might be e.g. "`x=false`": one cannot count on `y` keeping its `number` type.

Functions subtyping Functions are contravariant in their arguments, and covariant in their results:

$$\frac{\bar{\mathbb{E}}^2_? <: \bar{\mathbb{E}}^1_? \quad \bar{\mathbb{E}}^1_! <: \bar{\mathbb{E}}^2_!}{\bar{\mathbb{E}}^1_? \to \bar{\mathbb{E}}^1_! <: \bar{\mathbb{E}}^2_? \to \bar{\mathbb{E}}^2_!}(<:\mathit{Function})$$

Tables subtyping Subtyping between tables is directly lifted from field subtyping; unreferenced tables, marked with a prime, can be considered as regular table whenever suitable.

$$\frac{(\forall n \in [0...m])\ \mathbb{F}^a_n <: \mathbb{F}^b_n}{[(P_n : \mathbb{F}^a_n)^{\forall n \in [1...m]} | \mathbb{F}^a_0] <: [(P_n : \mathbb{F}^b_n)^{\forall n \in [1...m]} | \mathbb{F}^a_0]}(<:\mathit{Table})$$

Expression sequences subtyping Subtyping between expression sequences is only defined between sequences of the same length. Typing rules will pad sequence with `nil`s on the right whenever appropriate:

$$\frac{(\forall n \in [1...m])\ \mathbb{E}_1^n <: \mathbb{E}_2^n}{(\mathbb{E}_1^n)^{\forall n \in [1...m]} <: (\mathbb{E}_2^n)^{\forall n \in [1...m]}} (<: \bar{\mathbb{E}})$$

Statements subtyping Statement types are either $\varnothing_{\mathbb{S}}$, or of the form `return` $\bar{\mathbb{E}}$. In the latter case, subtyping is lifted from expression sequences subtyping:

$$\frac{\bar{\mathbb{E}}_1 <: \bar{\mathbb{E}}_2}{\texttt{return}\ \bar{\mathbb{E}}_1 <: \texttt{return}\ \bar{\mathbb{E}}_2} (<: \mathit{Return})$$

2.4 Typing rules

This section gives the typing rules, which allow to determine the belonging of terms to certain types. To do that, we'll enrich the calculus with a couple of type annotations. We won't discuss the possibility to algorithmically infer some of those.

Typing environments The rules exposed below use typing environments $\Gamma : v \mapsto \mathbb{F}$, functions from variables to field types (rather than, as one could have expected, expression types). Indeed, being able to separate constants from variables in the type system is valuable, and more importantly, the linearity issues handled by the `currently` modifier occur with local variables as well as with table fields.

This section won't address variable shadowing issues[2]: since we work on static terms which we don't evaluate, an appropriate α-renaming before typing can ensure that no such shadowing occurs.

Notations

- $\Gamma[v \leftarrow \mathbb{F}]$ is the function which, to v, associates $\mathbb{F}$, and to all other values $w \in \mathrm{dom}(\Gamma) \backslash \{v\}$ associates $\Gamma(w)$.
- This definition is extended homomorphically to sequences of variables and values $\Gamma[\bar{v} \leftarrow \bar{\mathbb{F}}]$.
- The empty environment, i.e. the function with an empty domain, is written $\varnothing_\Gamma$.
- $\Gamma \vdash T : \mathbb{T}$ means that under the assumptions in environment Γ, term T has type $\mathbb{T}$.

[2] i.e. homonymies, as in "`local x; x=function(x) return x end`", where there are two distinct variables which both share the name `x`.

- The notation $\Gamma \vdash E \therefore \mathbb{F}$ describes the types of left-hand sides operands in "=" assignment statements; these variables and table fields must retain field types rather than expression types, because their field qualifier indicates whether and how they can be updated. It means "under the assumptions Γ, and in an assignment's left-hand side context, expression E has type $\mathbb{F}$".
- We'll refer to expressions which can syntactically appear on the left-hand-side of an assignments, and in a "$\therefore$" judgment, as "slots". Those are local variables (*E-Variable*) and indexed tables (*E-Index*).

Calculus extensions The untyped calculus is extended with a couple of annotations which will allow to insert typing hints at appropriate places in programs. Syntactically, typing annotations will make heavy use of the pound # ascii character. This character, unused in Lua where type annotations may occur, will hopefully make typing annotations stand out visually, and make it easy to preprocess them out of a program's sources, so that it can be used by other interpreters.

- statement $\#\texttt{return } \bar{\mathbb{E}}; \bar{S}$ weakens the type of statements sequence $\bar{S}$ to statement type $\texttt{return } \bar{\mathbb{E}}$.
- $\texttt{function(}\overline{v\ \#\mathbb{E}}\texttt{)}\ \bar{S}\ \texttt{end}$ allows to give type annotations to function parameters, which are notoriously hard to infer effectively.
- Left-hand sides of assignments take a type field, allowing to handle `currently` field type updates, and linearity issues: $\overline{L\ \#\mathbb{F}} = \bar{\mathbb{E}}$.

$$\frac{}{\Sigma, (\#\texttt{return } \bar{\mathbb{E}}; \bar{S}) \Rightarrow \Sigma, \varnothing_S} \qquad \frac{\Sigma_1, \texttt{function(}\bar{v}\texttt{)}\ \bar{S}\ \texttt{end} \Rightarrow \Sigma_2, f}{\Sigma_1, \texttt{function(}\overline{v\ \#\mathbb{E}}\texttt{)}\ \bar{S}\ \texttt{end} \Rightarrow \Sigma_2, f}$$

$$\frac{\Sigma_1, (\texttt{local } \bar{L} = \bar{E}; \bar{S}) \Rightarrow \Sigma_2, S}{\Sigma_1, (\texttt{local } \overline{L\ \#\mathbb{F}} = \bar{E}; \bar{S}) \Rightarrow \Sigma_2, S}$$

Expression sequences Expressions in Lua can evaluate into multiple values. When concatenating expressions in a sequence, Lua only keeps the first value of each expression's evaluation, except for the last one which is expanded (*EE-Sequence*). For instance, if we have $\texttt{a} : (\mathbb{E}^1_a; \mathbb{E}^2_a)$, $\texttt{b} : (\mathbb{E}^1_b; \mathbb{E}^2_b)$ and $\texttt{c} : (\mathbb{E}^1_c; \mathbb{E}^2_c)$, then the type of the sequence $(\texttt{a}; \texttt{b}; \texttt{c})$ is $(\mathbb{E}^1_a; \mathbb{E}^1_b; \mathbb{E}^1_c; \mathbb{E}^2_c)$. A noteworthy property is that the number of elements in the type might be bigger than the number of expressions in the sequence. Another is that appending `nil`types at the end of a sequence type doesn't change it: `number, number, nil` must be treated as equal to `number, number`. This will be ensured by every rule effectively combining expression type sequences.

$$\frac{(\forall n \in [1...m]) \quad \Gamma \vdash E_n : (\mathbb{E}^1_n; \bar{\mathbb{E}}_n)}{\Gamma \vdash \bar{E} : (\mathbb{E}^1_n)^{\forall n \in [1...m]}; \bar{\mathbb{E}}_m} (TR\text{-}\bar{E})$$

Primitive expressions

$$\frac{}{\Gamma \vdash \langle \text{number} \rangle : \texttt{number}} \quad \frac{}{\Gamma \vdash \langle \text{string} \rangle : \texttt{string}}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{boolean}} \quad \frac{}{\Gamma \vdash \texttt{false} : \texttt{boolean}}$$
$$(TR\text{-}P)$$

Statement sequences Statement sequences appear in function bodies. What we need to know about them, besides the fact that they don't fail during evaluation, is the type of the expression sequences they return. Therefore we have two families of statement types: $\varnothing_{\mathbb{S}}$ for terms which don't return, and $\texttt{return}\ \bar{\mathbb{E}}$ for terms returning a sequence of expressions of type $\bar{\mathbb{E}}$.

$$\frac{\Gamma \vdash S : \varnothing_{\mathbb{S}} \quad \Gamma \vdash \bar{S} : \mathbb{S}}{\Gamma \vdash (S; \bar{S}) : \mathbb{S}}(TR\text{-}\varnothing_{\mathbb{S}}) \qquad \frac{\Gamma \vdash S : \texttt{return}\ \bar{\mathbb{E}}}{\Gamma \vdash (S; \bar{S}) : \texttt{return}\ \bar{\mathbb{E}}}(TR\text{-}\texttt{return})$$

$$\frac{\Gamma \vdash \bar{S} : \texttt{return}\ \bar{\mathbb{E}}_1 \quad \bar{\mathbb{E}}_1 <: \bar{\mathbb{E}}_2}{\Gamma \vdash (\#\texttt{return}\ \bar{\mathbb{E}}_2; \bar{S}) : \texttt{return}\ \bar{\mathbb{E}}_2}(TR\text{-}\#\texttt{return})$$

Variables Variable types are remembered in the environment Γ. They're slots, and have a field type $\mathbb{F}$ rather than an expression type $\mathbb{E}$. This allows to use them on the left-hand-side of assignements, to remember whether they're constant, private, or whether their current type can be updated. Field type judgments use the operator "$\therefore$", to avoid being confused with expression type judgments using ":".

$$\frac{\Gamma(v) = \mathbb{F}}{\Gamma \vdash v \therefore \mathbb{F}}(TR\text{-}\therefore)$$

When a slot L is used as a normal expression rather than an assignment's left-hand-side, its field type can be projected into an expression type:

$$\frac{\Gamma \vdash L \therefore \texttt{currently}\ \mathbb{E}}{\Gamma \vdash L : \mathbb{E}} \quad \frac{\Gamma \vdash L \therefore \texttt{var}\ \mathbb{E}}{\Gamma \vdash L : \mathbb{E}} \quad \frac{\Gamma \vdash L \therefore \texttt{const}\ \mathbb{E}}{\Gamma \vdash L : \mathbb{E}} \quad (TR\text{-}L)$$

There's no rule to project type `field`: it's never legal to use such a field in an expression's context. We'll see that no rule to project `just` types is needed either, because there's no legal way to derive a `just` type for a left-hand=side expression.

Table fields In this rule, the variable ϕ denotes a set of key/field types, plus the table's default type.

$$\frac{\Gamma \vdash E_T : [P : \mathbb{F}_K; \phi]}{\Gamma \vdash E_T[P] \therefore \mathbb{F}_K}(TR\text{-}[\ ])$$

In some cases, an expansion of the default field type might be needed, e.g. we have $\frac{\Gamma \vdash E : [|\texttt{const nil}]}{\Gamma \vdash E[\texttt{"x"}] \therefore \texttt{const nil}}$, because $[|\texttt{const nil}] = [\texttt{"x"} : \texttt{const nil} | \texttt{const nil}]$.

Literal tables Literal tables have all their fields typed with `just` modifiers. When the literal table will be stored in a variables, the field types will be weakened into either `currently`, `var`, `const` or `field` types.

$$\frac{(\forall n \in [1...m]) \quad \Gamma \vdash E_n : (\mathbb{E}^1_n; \bar{\mathbb{E}}_n)}{\Gamma \vdash \{([P_n] = E_n)^{\forall n \in [1...m]}\} : [(P_n : \texttt{just } \mathbb{E}^1_n)^{\forall n \in [1...m]} | \texttt{just nil}]} (\textit{TR-Table})$$

Local variable declarations Unlike in most type systems, newly created variables are given the `nil` type, rather than the type of their future content. This is because assignment statements change the type of the variables on which they operate, as we'll see below. Besides, typing as non-`nil` a variable while it contains `nil` wouldn't be sound.

$$\frac{\Gamma[\bar{v} \leftarrow \overline{\texttt{currently nil}}] \vdash S : \mathbb{S}}{\Gamma \vdash (\texttt{local } \bar{v}; \bar{S}) : \mathbb{S}} (\textit{TR-Local})$$

Assignments Assignments can change the type of `currently` variables and fields in Γ; they can also perform weakenings, essentially changing `var` field types into `const`s. They must be preventing from altering `const` and `field` slots. To type them, we'll need two auxiliary predicates:

- $\Gamma \vdash \text{update}(L, \mathbb{F}) = [\sigma]$ checks whether variable/field L is allowed to have its current type changed into $\mathbb{F}$. If it is, it returns a substitution $[\sigma]$ over environments, so that $\Gamma[\sigma]$ is the typing environment in effect after the assignment has been performed.
- $\mathbb{F}_L \rhd \mathbb{F}_R$ checks whether the content of slot of type $\mathbb{F}_R$ can be stored in a slot of type $\mathbb{F}_L$. It is, as we'll see, a subset of $:>$ the opposite of the subtyping relationship.

The former will prevent from changing the type of `var` fields, and from changing the content of `field` or `const` fields: there will be no rule allowing to derive $\Gamma \vdash \text{update}(L, \texttt{const } \mathbb{E}) = [\sigma]$; moreover, $\Gamma \vdash \text{update}(L, \texttt{var } \mathbb{E}) = [\,]$ will only be derivable from $\Gamma \vdash L \therefore \texttt{var } \mathbb{E}$, and will only produce empty type substitutions $[\,]$. Once update() has allowed an assignment based on the field's former and new types, $\rhd$ checks that the content put in the field is consistent with the new type, to prevent such unsound assignments as `v #var number="abc"`.

The following rule allows to change the content of a `var` slot, as long as its type isn't changed:

$$\frac{\Gamma \vdash L \therefore \texttt{var } \mathbb{E}}{\Gamma \vdash \text{update}(L, \texttt{var } \mathbb{E}) = [\,]} (\textit{UP-}\texttt{var})$$

`currently` slots can change the type of the value they contain, but the slot type itself can also be changed, into a `var`, a `const` or even a `field`.

$$\frac{\Gamma \vdash v \therefore \texttt{currently } \mathbb{E}}{\Gamma \vdash \text{update}(v, \mathbb{F}) = [v \leftarrow \mathbb{F}]}(UP\text{-}\texttt{cur})$$

`currently` fields within tables pose an additional difficulty: if the field's type changes, the type of the table containing it also changes. Therefore, the table itself must also be stored in a `currently` slot, etc. recursively until we reach a top-level `currently` variable. The typing of assignments to those fields is therefore defined recursively, with $(UP\text{-}\texttt{cur})$ as a base case, and $(UP\text{-}\texttt{cur}[\ ])$ below as the inductive rule:

$$\frac{\begin{array}{l}\Gamma \vdash L \therefore \texttt{currently } [P : \texttt{currently } \mathbb{E}; \phi] \\ \Gamma \vdash \text{update}(L, \texttt{currently } [P : \mathbb{F}; \phi]) = [\sigma]\end{array}}{\Gamma \vdash \text{update}(L[P], \mathbb{F}) = [\sigma]}(UP\text{-}\texttt{cur}[\ ])$$

As a usage example, let's consider an object `x` with a field `y` currently containing a number, and updated to a string variable. To make the proof tree terser, we'll use the following definitions for `x`'s former type $\mathbb{F}_1$, its new type $\mathbb{F}_2$, and the typing environment before assignment Γ respectively:

$$\begin{array}{l}\mathbb{F}_1 = \texttt{currently } [\texttt{"y"} : \texttt{currently number}] \\ \mathbb{F}_2 = \texttt{currently } [\texttt{"y"} : \texttt{var string}] \\ \Gamma = \{\texttt{x} \mapsto \mathbb{F}_1\}\end{array}$$

The soundness of environmnent substitution $[x \leftarrow \mathbb{F}_2]$ is computed by $(UP\text{-}\texttt{cur})$ over `x`; from there, it's concluded by $(UP\text{-}\texttt{cur}[\ ])$ that `x["y"]` can also cause this substitution, because both `x` and `x["y"]` are `currently` slots:

$$\frac{\dfrac{\Gamma(\texttt{x}) = \mathbb{F}_1}{\Gamma \vdash \texttt{x} \therefore \mathbb{F}_1}(TR\text{-}\therefore) \qquad \dfrac{\dfrac{\Gamma(\texttt{x}) = \mathbb{F}_1}{\Gamma \vdash \texttt{x} \therefore \mathbb{F}_1}(TR\text{-}\therefore)}{\Gamma \vdash \text{update}(\texttt{x}, \mathbb{F}_2) = [x \leftarrow \mathbb{F}_2]}(UP\text{-}\texttt{cur})}{\Gamma \vdash \text{update}(\texttt{x["y"]}, \mathbb{F}_2) = [x \leftarrow \mathbb{F}_2]}(UP\text{-}\texttt{cur}[\ ])$$

Because `currently` fields are only usable when they're inside other `currently` fields all the way up to a variable, there's no point having types such as $v \therefore \texttt{var } [P : \texttt{currently } \mathbb{E}; \phi]$: it wouldn't allow anything more than $v \therefore \texttt{var } [P : \texttt{var } \mathbb{E}; \phi]$.

$\mathbb{F}_L \rhd \mathbb{F}_R$ checks whether what's stored in a field has an appropriate expression type. It also keeps track of linearity, forcing to transform `just` $\mathbb{E}$ into `currently` $\mathbb{E}$, and `currently` $\mathbb{E}$ into `field`. The relation is expressed between two fields rather than a field and an expression, to ease its recursive over tables (last rule below):

$$\frac{}{\texttt{currently } \mathbb{E} \rhd \texttt{just } \mathbb{E}} \qquad \frac{}{\texttt{field} \rhd \texttt{currently } \mathbb{E}}$$

$$\frac{}{\texttt{var } \mathbb{E} \rhd \texttt{var } \mathbb{E}} \qquad \frac{}{\texttt{const } \mathbb{E} \rhd \texttt{const } \mathbb{E}}$$

$$\frac{(\forall n \in [0...m]) \quad \mathbb{F}^L_n \rhd \mathbb{F}^R_n}{\texttt{currently } [(P_n : \mathbb{F}^L_n)^{\forall n \in [1...m]} | \mathbb{F}^L_0] \rhd \texttt{just } [(P_n : \mathbb{F}^R_n)^{\forall n \in [1...m]} | \mathbb{F}^R_0]}$$

$$\frac{\mathbb{F}_L \rhd \mathbb{F}_{R1} \quad \mathbb{F}_{R1} :> \mathbb{F}_{R2}}{\mathbb{F}_L \rhd \mathbb{F}_{R2}}$$

(*Accept*)

Notice that although $\rhd$ is a subset of $:>$, it isn't an order relayionship: it isn't idempotent (e.g. `just` $\mathbb{E} \not\rhd$`just` $\mathbb{E}$). By using the composition with $:>$, we can choose to store a `just` field inside a table into either a `currently` or a `var` one; the former will allow to change the field's type, but any copy of it can't be used (it will have to be further weakened into `field`); the latter will lock the type's content, but allows to make and use further copies.

It is possible, but pointless, to put a `currently` field in a `var` one: the outer `var` one will prevent from modifying the inner one, thus making it strictly less usable than a `var` field (no type modification and no usable copy).

Equipped with these rules, we can now type assignments. But as an intermediate step, we'll spell the simpler rule for the special case where both left-hand side and right-hand side sequences have only one element:

$$\frac{\Gamma \vdash E : \mathbb{E} \qquad \mathbb{F} \rhd \texttt{just } \mathbb{E} \qquad \Gamma \vdash \text{update}(L, \mathbb{F}) = [\sigma] \qquad \Gamma[\sigma] \vdash \bar{S} : \mathbb{S}}{\Gamma \vdash (L \ \#\mathbb{F} = E; \bar{S}) : \mathbb{S}}$$

To paraphrase, it must be possible (1) E must be well typed; (2) this type must be legal to store in a field of L's new type $\mathbb{F}$; (3) it must be legal to substitute L's former type with the new one $\mathbb{F}$; (4) the rest of the sequence $\bar{S}$ must be typable with the type substitution $[\sigma]$ applied in Γ.

The actual rule, although more intimidating because it deals with sequences of possibly different lengths, is not more sophisticated. The two upper premices define a type $\bar{\mathbb{E}}$ corresponding to $\mathbb{E}$ above, with some `nil`-padding if needed to match the right-hand-side's length. The lower one involving $\rhd$ and update() corresponds to premices (2) and (3), applied to each (left, right) pair; and the final premice chains all substitutions together to type the following statements:

$$\frac{\begin{array}{l}(\forall n \in [1...p])\ \Gamma \vdash E_n : \mathbb{E}_n \quad (\forall n \in [p+1...m])\ \mathbb{E}_n = \texttt{nil} \\ (\forall n \in [1...m])\ \mathbb{F}_n \rhd \texttt{just } \mathbb{E}_n \text{ and } \Gamma \vdash \text{update}(L_n, \mathbb{F}_n) = [\sigma_n]\end{array} \quad \Gamma[\sigma_1...\sigma_n] \vdash \bar{S} : \mathbb{S}}{\Gamma \vdash ((L_n \ \#\mathbb{F}_n)^{\forall n \in [1...m]} = (E_n)^{\forall n \in [1...p]}; \bar{S}) : \mathbb{S}}$$

(*TR-Assign*)

TODO: *Substitution conflicts within an assignment aren't handled, e.g.* `x #[|currently nil] = { }; x.a, x.b = "A", "B"`. *The simplest solution is to mandate that substitutions have disjoint domains within an assignment.*

Functions Functions break linearity: they assign arguments to parameters, which can create a second reference to a table. Moreover, due to Lua's support for full closures, they capture variables defined outside of them (these variables are called "upvalues" in Lua). Functions don't use upvalues' content immediately, where the function is defined; instead, they'll use them whenever they're called, and in between, the content of a `currently` field or variable might have been changed arbitrarily.

For this reason, when typing the function's body, we weaken every upvalue through $\rhd$, so that it doesn't rely on any variable having a field type of the form `currently` $\mathbb{E}$. We elect to type parameters as `var` slots, thus allowing to change their content in the function's body. It would have been possible to forbid it by typing them with `const`[3].

$$\frac{\Gamma_{\text{in}} \rhd \Gamma_{\text{out}} \quad \Gamma_{\text{in}}[\bar{v} \leftarrow \texttt{var}\ \bar{\mathbb{E}}_?] \vdash \bar{S} : \texttt{return}\ \bar{\mathbb{E}}_!}{\Gamma_{\text{out}} \vdash \texttt{function(}\overline{v\ \#\mathbb{E}_?}\texttt{)}\ \ \bar{S}\ \ \texttt{end} : \mathbb{E}_? \to \bar{\mathbb{E}}_!}\ (\textit{TR-Function})$$

(this rule generalizes $\rhd$ over environments, in the obvious way: $\Gamma_1 \rhd \Gamma_2$ iff Γ_1 has the same domain D as Γ_2, and $(\forall v \in D)\ \Gamma_1(v) \rhd \Gamma_2(v)$)

Function applications As we did for (*TR-Assign*), let's demystify the function application rule, by first giving the simplified version with one parameter, one argument and one result: (1) what's applied must be a function, (2) the argument must be well typed, and (3) this argument type must be compatible with the parameter:

$$\frac{\Gamma \vdash E_f : (\mathbb{E}^?) \to (\mathbb{E}^!) \qquad \Gamma \vdash E : \mathbb{E}^a \qquad \mathbb{E}^a <: \mathbb{E}^?}{\Gamma \vdash E_f(E^a)_E : (\mathbb{E}^!)}$$

This rule is extended to multiple parameters / arguments / results almost trivially; the only point to take care of is that the argument types sequence might be padded with `nil` types, if it's shorter than the parameter types sequence:

$$\frac{\begin{array}{c}\Gamma \vdash E_f : (\mathbb{E}^?_n)^{\forall n \in [1...m]} \to (\mathbb{E}^!_n)^{\forall n \in [1...p]} \\ (\forall n \in [1...q])\ \Gamma \vdash E^a_n : \mathbb{E}^a_n \quad (\forall n \in [q+1...m])\ \mathbb{E}^a_n = \texttt{nil} \\ (\forall n \in [1...m])\ \mathbb{E}^a_n <: \mathbb{E}^?_n\end{array}}{\Gamma \vdash E_f((E^a_n)^{\forall n \in [1...q]})_E : (\mathbb{E}^!_n)^{\forall n \in [1...p]}}\quad (\textit{TR-Apply-E})$$

Finally, function applications in a statement context are typed by discarding applying (*TR-Apply-E*), then forgetting the results and replacing them with the non-returning-statement type $\varnothing_{\mathbb{S}}$:

$$\frac{\Gamma \vdash E_f(\bar{E})_E : \bar{\mathbb{E}}}{\Gamma \vdash E_f(\bar{E})_S : \varnothing_{\mathbb{S}}}\quad (\textit{TR-Apply-S})$$

[3] I wonder whether typing parameters as `currently` inside the function's body would have been admissible. It's not trivial: we must not allow to change a field in a table, so we must be sure there's no `currently` table fields.

3 Future work

This section sums up a list tasks remaining to be completed, in order to turn this calculus into a useful type-checker for actual Lua programs.

- Implement a type checker for the existing calculus.
- Produce a soundness proof (formal demonstration that a typed term can't cause a runtime type error).
- Combine with gradual typing.
- Define a pragmatic syntax: allow to omit easily guessed type annotations, have sensible defaults for missing indications such as field type modifiers, etc.
- Consider an alternative, Lua compatible syntax for types which fits into Lua comments. This version would be backward-compatible with plain Lua compilers, and would certainly present similarities with Luadoc-like tools.
- Support for (limited) type inference. Unwritten types must not be all interpreted as dynamic types; otherwise, no type checking at all would occur in programs which aren't fully annotated. A reasonable compromize would probably be that unannotated function parameters are dynamic, but unnanotated locals are to be guessed through inference.
- Clarify what typed programs look like. Not all Lua programming styles will be supported: for instance, modules which pollute global variables will probably not be accepted. The choices about what's acceptable for the type system must not only be made: their rationales and their price must be carefully justified.
- Extend the calculus with missing parts of Lua. Some should be easy to incorporate in the type system (loop statements, pseudo-fields based on `__index` tables); others would have to remain dynamically typed (binary operators, varargs...). Here gradual typing really saves the day: constructs which no current type system handles satisfactorily, such as covariant binary operators, can be left dynamically typed without making the whole type system fall apart.