3
Type Inference and Optimization

The previous chapter presented a basic compiler from Lua to the CLR and some variations of it, changing the runtime representation of Lua values and the treatment of functions that return multiple values. All variations of the basic compiler have in common the fact that they did not need any analysis of the source code beyond basic analysis to tie the use of local variables with their definitions.

This chapter presents a more complex variation of the basic compiler, using information derived from a *type inference* algorithm, a kind of static analysis that tries to assign a type to each variable and expression in the program. If types are precise enough, the compiler can use more efficient runtime representations for values, and can generate more efficient code for operations.

In Section 3.1, we give an overview of the problem of typing inference in the Lua programming language, and describe our type inference algorithm. In Section 3.2, we show how the compiler uses the type information extracted by the algorithm. In Section 3.3, we review related work on type inference and type-related analysis for dynamic programming languages, and discuss how our work differs from this other work.

3.1
Type Inference For Lua

Lua is a dynamically typed language, which combines lack of type annotations with runtime type checking. This imposes several constraints in the representation of Lua values for the Lua compilers we presented in the previous chapter: all numbers have to use the same underlying representation as other values, and any operation involving two numbers converting from this representation to native CLR numbers, doing the operation, and converting back to the common representation; all polymorphic operations have to be dispatched through virtual methods, a form of dynamic dispatch natively supported by the CLR; all functions need to be able to take any number of arguments of any type; all function applications can produce any number of

values of any type; finally, all tables have to allow keys and values of any type.

We can use more efficient representations and can generate more efficient code if we are sure that variables and expressions have more precise types. In an extreme case, if we are sure that expression e_1 can only be a number and that expression e_2 can only be a number, we can safely make both expressions evaluate to `double`, so the expression e_1+e_2 compiles to the following Common Intermediate Language code, where $C(e)$ is the code for evaluating expression e and leaving the result on the top of the CLR's data stack:

$C(e_1)$
$C(e_2)$
add

Contrast this with the code when we can not be sure e_1 and e_2 are numbers, which is the following CIL code in the "box", "intern", and "prop" compilers (we elide the case where either expression did not evaluate to a number):

$C(e_1)$
dup
isinst `double`
brfalse *add1*
unbox `double`
$C(e_2)$
dup
isinst `double`
brfalse *add2*
unbox `double`
add
box `double`
br *out*
add1: ...
br *out*
add2: ...
out: *ldots*

What was just a simple addition now involves type checking, unboxing and reboxing the result.

Another interesting case is the compilation of expressions such as $e[c]$ where c is a string literal and we are sure that e evaluates to a table whose keys are all statically known and include c (a record, in other words). This is a common expression in Lua because of the e.name syntactic sugar for tables. In

this case, we can represent the table as a sealed CLR class (a heap-allocated record), and the expression compiles to the following simple code, where t is the type of the record we synthesized for the table, and **ldfld** is a very efficient field access operation (in practice an indexed memory fetch):

$$C(e)$$
$$\textbf{ldfld}\ t::c$$

Contrast with the following code, where **callvirt** is a dynamically dispatched call to a function that does a hashtable lookup, and we elide the special case where e evaluates to a number:

$$\begin{array}{rl} & C(e) \\ & \textbf{dup} \\ & \textbf{isinst}\ \texttt{double} \\ & \textbf{br}\ num \\ & \textbf{castclass}\ \texttt{Lua.Reference} \\ & \textbf{ldsfld}\ \texttt{InternedSymbols}::c \\ & \textbf{callvirt}\ \texttt{Lua.Reference::get\_Item(Lua.Symbol)} \\ & \textbf{br}\ out \\ num: & \ldots \\ out: & \ldots \end{array}$$

We want an algorithm that can extract from the program the type information necessary for this kind of optimization. The specific algorithm we use is a form of *type inference*. A type inference algorithm uses syntax-directed typing rules to build and solve a set of constraints on the types of the program [Damas and Milner, 1982]. The solution ideally assigns the most precise type for each expression and variable that still satisfies all typing rules. All Lua programs have to be typable by our type inference algorithm, the variation will only be in the degree of precision of this typing; programs that make more use of Lua's dynamism will necessarily have more imprecise types but will still be well-typed, that is, our type inference will not introduce errors in correct programs.

Our type system will assign type $\mathcal{D}$, the *dynamic* type, to all variables and expressions whose precise type can only be known at runtime. We say these variables and expressions hold and evaluate to *tagged values* from the way runtime type checking is traditionally implemented (e.g. in the Lua interpreter), by representing a dynamic value as a tagged union. Any operation on a tagged value involves a check of the tag and dispatching based on this tag. An abstract class and concrete subclasses, the representation we used on the last chapter, is a kind of tagged union. Typing all variables and expressions

with type $\mathcal{D}$ produces a valid typing for any well-formed Lua program, but without any static type optimizations.

We are interested in using better representations than tagged unions in our compiler, so we will introduce several *untagged types* in addition to $\mathcal{D}$. We will assign an untagged type to any variable and expression for which we can infer a precise type. These variables and expressions hold and evaluate to *untagged values*, so the implementation can use different and incompatible representations for tagged and untagged types. For example, if we infer the untagged type **Number** to variable x then we know x will only hold untagged numbers, and we can choose a representation accordingly (`double` in the compiler we present in Section 3.2). If we infer type $\mathcal{D}$ to x then x can potentially hold any tagged value, even if at runtime it will only hold tagged numbers, so we have to use the fallback dynamic representation (`object` in the case of the CLR).

We could eliminate the distinction between tagged and untagged values, and use the inferred type information only to eliminate type checks and dynamic dispatch, which is what *soft typing* approaches do [Wright and Cartwright, 1997], but we would lose important optimization opportunities. For example, in the code fragments we presented in the beginning of this section we would only be able to eliminate the type checks and the dynamic dispatch, but the unboxing, reboxing and the hashtable lookup would still be there.

We will have untagged types for the first-order values booleans, numbers, strings, and nil, and also for the higher-order values tables and functions. Threads and userdata do not have untagged types as a simplification (our simplified Lua core in Section 3.1.4 does not have threads and userdata). Lua functions can return multiple values in some syntactical contexts, so we will also introduce a "second-class" tuple type for these situations. Our table types will be a family of related types, corresponding to the different ways tables get used in Lua programs: records, sparse arrays, hash tables, or a combination of these.

Our type system will also have a *coercion relation* $\rightsquigarrow$ that applies when the values of a type can be coerced to values of another type without error (with a runtime conversion if the two types do not share the same representation). The coercion relation lets part of an expression have an untagged type even if the expression as a whole needs to have type $\mathcal{D}$. It also allows us to introduce *singleton types* for literals that appear in the program, which in turn lets us infer record-like types for tables that are only indexed by literals. And also allows us to introduce *nullable types*, which are unions of an untagged type and

the type of nil, useful for typing table indexing expressions, as indexing a non-existent value in Lua evaluates to nil instead of raising an error. The coercion relation is similar to a subtyping relation with regard to contravariance, so coercions from function and table types will be severely restricted. Section 3.1.2 gives the full coercion relation and elaborates on these issues.

There is no ML-style *parametric* polymorphism [Cardelli and Wegner, 1985, Damas and Milner, 1982] in our type system, for pragmatic reasons that we elaborate on Section 3.1.3. The lack of polymorphic types in our type system will not make programs fail to type check and compile, because our type system has $\mathcal{D}$ as a fallback type. The worst that can happen is a loss of precision, with a corresponding loss of runtime efficiency. This is different from the type systems of languages of the ML family, where lack of polymorphic types can make useful programs not compile at all.

Finding a valid and precise typing for a program is the job of our type inference algorithm. The core of the algorithm is a traversal of a program's abstract syntax tree, typing the expressions in the tree from the leafs to the root. If there are several valid typings for an expression then the algorithm will use the most precise one. Coercions in the typing rules lets the type inference assign a less precise type to an expression while having more precise types in its parts. The contravariance restrictions on coercion of function and table types add another complication, though; to assign a type to an expression the algorithm may need to change the type of other expressions that already have been typed.

For example, a function that has already been typed as **Number** $\rightarrow$ **Number** has to be applied to a **String**. In this case, the algorithm needs to change the type of the function's parameter from **Number** to $\mathcal{D}$. This may induce a change in the return type of the function, and in other expressions that have already been typed. To deal with the situations like these, the algorithm is *iterative*, and does traversals of the program's tree until the types of all terms have converged. Termination is guaranteed because types always change from more precise to less (according to coercion). Once the type of a term becomes $\mathcal{D}$ it will remain $\mathcal{D}$.

This "mutability" of function and table types is reflected in our typing rules by non-determinism in the typing rules for constructors of these values. The type of a function can have more parameter types than the function's arity, for example, and the type of the table constructor can be any valid table type. Section 3.1.5 details the algorithm and gives a non-trivial example of its iterative type assignment.

In the rest of this section we give a detailed description of the types,

coercion relation, and the main typing rules of our type system, and also detail the main parts of the type inference algorithm.

3.1.1 Type Language

In the previous section we have already introduced the first type of our type system, $\mathcal{D}$. A value having type $\mathcal{D}$ means we can only know its type at runtime, so values of type $\mathcal{D}$ have to carry their type information in their runtime representation, which is why we called them tagged values. Our type system will guarantee that any variable that can hold a tagged value or any expression that can produce one will have type $\mathcal{D}$. In the rest of this section we will present the other types in our type system.

Let us start with *singleton* types, the types of constants. Singleton types are **nil**, **true**, and **false**, plus a singleton type for each number and string (we will use n to mean a numeric singleton type and s to mean a string singleton type). Typing literals and constants with these singleton types will let us infer record types for some tables.

The next types we will introduce are **Bool**, **Number**, and **String**, for values that are known at runtime to be booleans, numbers, or strings, respectively. Notice that while literal string "foo" has the **"foo"** singleton type, the coercion relation effectively also gives it the **String** type because "foo" $\rightsquigarrow$ **String**.

Table types can take two forms. The first form is a type with the template $\mathcal{D} \mapsto \upsilon$, where υ is a type. These are hash tables with dynamic keys and values of type υ. The other form is a conjunction $\tau_1 \mapsto \upsilon_1 \wedge \ldots \wedge \tau_n \mapsto \upsilon_n$ with $n \geq 1$ and $\tau_k \neq \mathcal{D}$, meaning a table where the keys can have any of the types $\tau_1, \ldots, \tau_n$ and the values any of the types $\upsilon_1, \ldots, \upsilon_n$, with keys of type τ_k having values of type υ_k.

Table types as defined above can be ambiguous. For example, in the type $2 \mapsto$ **Number** $\wedge$ **Number** $\mapsto$ **String** the type of the value corresponding the key 2 could be **Number** or **String**, as 2 can be interpreted as having singleton type 2 or type **Number** (because of coercion). To remove this ambiguity we restrict the types of the keys so that for any distinct key types τ_i and τ_j there is no type σ with $\sigma \rightsquigarrow \tau_i$ and $\sigma \rightsquigarrow \tau_j$.

To talk about function types we will first define tuple types, which correspond to heterogeneous (and immutable) lists of values. Tuples are not first-class values in Lua. They have temporary existence as the result of evaluating a list of expressions (the rvalue of an assignment or the arguments of a function application), and sometimes can be returned as the result of a

function application, but the elements of a tuple cannot be other tuples. The size of a tuple size may not be statically know, so our type system has to reflect this. We will give the type **empty** to empty tuples. Non-empty tuples of known size have types of the form $\tau_1 \times \ldots \times \tau_n$ with $n \geq 1$ (τ_k can not be a tuple type, naturally). Tuples with a known minimum but unknown maximum size have types of the form $\tau_1 \times \ldots \times \tau_n \times \mathcal{D}^*$ (possibly $\mathcal{D}^*$).

We can now define function types as types of the form $\tau \rightarrow \upsilon$, where τ and υ are tuple types. Variadic functions are functions where the domain type is a tuple type of unknown maximum length, so variadic arguments of a function always have type $\mathcal{D}$ in our type system.

Situations where a value can either be nil or some other untagged value are common in our type system because of the way Lua tables work. In Lua, indexing a table with a key that does not exist is not an error, but returns **nil**. This means that even if all assignments to keys of type τ have type υ there is the possibility of indexing the table and getting **nil**. We introduce *nullable* types $\tau?$ to represent the union of τ and **nil** (a value of type $\tau?$ is either a value of type τ or **nil**). To simplify our type inference, we restrict the type τ in $\tau?$ to simple, table, and function types.

Our type system also types statements, not just expressions. The type of a single statement is the singleton type **void** if it is not a **return** statement. The type of a block of statements is also **void** if no **return** statements are present in the block.

Finally, we need a way to define recursive types (to be able to have types for things such as linked lists and trees); we use $\mu\alpha.\tau$ for recursive types, where τ is a function or table type with α appearing anywhere a function or table type could appear. For example, $\mu\alpha.(1 \mapsto \textbf{Number} \wedge 2 \mapsto \alpha?)$ represents single linked lists of numbers.

Figure 3.1 summarizes our complete type language.

3.1.2 Types and Coercion

The core of our typing rules and type inference algorithm is a coercion relation $\tau \rightsquigarrow \upsilon$ between two types τ and υ that holds whenever values of type τ can be coerced into values of type υ. This coercion means that values of type τ can be converted to values of type υ, or it means that the both types τ and υ share the same runtime representation, depending on the how we map types to concrete representations.

The coercion relation is reflexive and transitive, and Figure 3.2 lists its base cases.

tagged types

dynamic	$::= \mathcal{D}$
dynamic list	$::= \mathcal{D}^*$

untagged types

singleton	$::= n$, s, **nil**, **true**, **false**
simple	$::=$ **Bool**, **Number**, **String**
table	$::= \tau_1 \mapsto \upsilon_1 \wedge \ldots \wedge \tau_n \mapsto \upsilon_n$ with $n \geq 1$, where $\forall k.\tau_k \neq \mathcal{D}$ and $\forall i, j, \sigma.(i \neq j \wedge \sigma \rightsquigarrow \tau_i) \rightarrow \sigma \not\rightsquigarrow \tau_j$
	$::= \mathcal{D} \mapsto \upsilon$
function	$::= \tau \rightarrow \upsilon$, where τ and υ are tuple types
nullable	$::= \tau?$, where τ is a simple, function, or table type
recursive	$::= \mu\alpha.\tau$, where τ is a function or table type with α standing in for τ

tuple types

tuple	$::= \tau_1 \times \ldots \times \tau_n$ with $n \geq 1$
	$::= \tau_1 \times \ldots \times \tau_n \times \mathcal{D}^*$ with $n \geq 0$
	$::=$ **empty**

Figure 3.1: Type Language

The simple and nullable coercions are straightforward. Any untagged first-order type can be coerced to $\mathcal{D}$, and a nullable type can also be coerced to $\mathcal{D}$ if its base type can be coerced. Tables and functions are a different matter; only tables that map tagged values to tagged values and functions that take tagged values and return a dynamic list can be coerced to $\mathcal{D}$, because there are no coercions from $\mathcal{D}$ to untagged types and functions and tables are contravariant on their domain and key types, respectively.

The coercion rules for tuples are best understood in the context of the typing rules that employ them, so we will defer the explanation to the next section, where we describe and explain some the essential type rules of our type system.

The purpose of the coercion relation is to balance the need of inferring precise types with the need of inferring types for all correct Lua programs (which often means using $\mathcal{D}$). A coercion constraint in a typing rule means that a part of the expression being typed can have a more precise type than the whole expression. For example, coercion allows the type system to type an argument in a function application with a function of type $\mathcal{D}^* \rightarrow \mathcal{D}^*$ with a more precise type such as **Number**.

simple coercions

$$\mathbf{true} \rightsquigarrow \mathbf{Bool}$$
$$\mathbf{false} \rightsquigarrow \mathbf{Bool}$$
$$n \rightsquigarrow \mathbf{Number}$$
$$s \rightsquigarrow \mathbf{String}$$

nullable coercions

$$\mathbf{nil} \rightsquigarrow \tau?$$
$$\tau \rightsquigarrow \tau?$$

tagging coercions

$$\mathbf{nil} \rightsquigarrow \mathcal{D}$$
$$\mathbf{Bool} \rightsquigarrow \mathcal{D}$$
$$\mathbf{Number} \rightsquigarrow \mathcal{D}$$
$$\mathbf{String} \rightsquigarrow \mathcal{D}$$
$$\tau? \rightsquigarrow \mathcal{D} \text{ iff } \tau \rightsquigarrow \mathcal{D}$$
$$\mathcal{D} \mapsto \mathcal{D} \rightsquigarrow \mathcal{D}$$
$$\tau_1 \times \ldots \times \tau_n \rightarrow \mathcal{D}^* \text{ with } n > 0 \rightsquigarrow \mathcal{D} \text{ iff } \tau_1 = \mathcal{D} \wedge \ldots \wedge \tau_{n-1} = \mathcal{D} \wedge (\tau_n = \mathcal{D} \vee \tau_n = \mathcal{D}^*)$$

tuple coercions

$$\mathbf{empty} \rightsquigarrow \mathbf{nil} \times \ldots \times \mathbf{nil}$$
$$\tau_1 \times \ldots \times \tau_n \rightsquigarrow \upsilon_1 \times \ldots \times \upsilon_n \text{ iff } \tau_k \rightsquigarrow \upsilon_k$$
$$\tau_1 \times \ldots \times \tau_n \rightsquigarrow \tau_1 \times \ldots \times \tau_n \times \text{nil}$$
$$\tau_1 \times \ldots \times \tau_n \times \mathcal{D}^* \rightsquigarrow \upsilon_1 \times \ldots \times \upsilon_n \times \mathcal{D}^* \text{ iff } \tau_k \rightsquigarrow \upsilon_k$$
$$\tau_1 \times \ldots \times \tau_n \times \mathcal{D}^* \rightsquigarrow \tau_1 \times \ldots \times \tau_n \times \mathcal{D} \times \mathcal{D}^*$$
$$\tau_1 \times \ldots \times \tau_n \rightsquigarrow \mathcal{D}^* \text{ iff } \tau_k \rightsquigarrow \mathcal{D}$$
$$\tau_1 \times \ldots \times \tau_n \times \mathcal{D}^* \rightsquigarrow \mathcal{D}^* \text{ iff } \tau_k \rightsquigarrow \mathcal{D}$$

Figure 3.2: Coercion Relation

3.1.3 Monomorphism Restriction

Lua's primitive operations exhibit *ad-hoc* [Cardelli and Wegner, 1985] instead of parametric polymorphism. A simple function like the function foo in the following fragment has no single polymorphic type:

```
function foo(a, b)
  return a[1], b[2]
end
```

Function foo can work on any Lua type, via extensible semantics. If indexing was restricted to tables then *foo* still would not be polymorphic, as there is insufficient information to know if the table a is a record, an array, a hashtable, a set, or other structures that Lua tables can emulate, each with a different polymorphic type.

One way to assign polymorphic types to *foo* would be to type each call site of *foo* separately, assuming that we are sure that those call sites only call *foo*, so we could get enough information to resolve the ad-hoc polymorphism of the indexing operator at compile-time. Each inferred type for *foo* would lead to at least one different compilation of *foo*, because the use of different representations may force more than one compiled version for the same parametric type. If *foo* is parametric on two types then we need $m \times n$ versions where m is the number of representations of one of the types and n the number of representations of the other. If a call site of *foo* is inside a function *bar* and *bar* itself is polymorphic, then the call sites of *foo* in each polymorphic version of *bar* have to be typed separately.

Even if we accept the increased code size of having several compiled versions of the same function, polymorphic type inference in the presence of assignment and mutable data structures is unsound in the general case. Restrictions in the inference algorithm can restore soundness, but at the cost of greater complexity of the type inference (greater complexity of implementation, greater complexity of understanding by the user, and greater complexity in the algorithmic sense) [Leroy and Weis, 1991]. Restricting our type inference to monomorphic types does not mean restricting the set of programs that are typable, only the precision of the type inference, so we decided to forego the extra complexity of polymorphic types.

3.1.4 Typing Rules

We use a simplified core of the Lua language to make the presentation of the typing rules in this section easier. This simplified core removes syntactic sugar, reduces control flow statements to just if and while statements, makes variable scope explicit, and splits function application in three different operators, $f(el)_0$ when we discard return values (function application as a statement), $f(el)_1$ when we want exactly one return value (the first, or **nil** if the function returned no values), and $f(el)_n$ when we want all return values.

Appendix A gives an operational semantics for our simplified core, modeling extensible semantics (metamethods, see Section 1.1) through special primitives. The simplified semantics just capture how the extensible semantics influences the typing of operations and do not try to specify their precise behavior.

Figure 3.3 describes the abstract syntax of our core Lua. The syntactic categories are as follows: s are statements, l are lvalues, el are expression lists, me are multi-expressions (single expressions that can evaluate to multiple

$$
\begin{array}{rcl}
s & ::= & s_1; s_2 \mid \textbf{skip} \mid \textbf{return}\ el \mid e(el)_0 \mid \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \mid \\
 & & \textbf{while}\ e\ \textbf{do}\ s \mid \textbf{local}\ \vec{x} = el\ \textbf{in}\ s \mid \textbf{rec}\ x = f\ \textbf{in}\ s \mid \vec{l} = \\
 & & el \\
l & ::= & x \mid e_1[e_2] \\
el & ::= & \textbf{nothing} \mid \vec{e} \mid me \mid \vec{e}, me \\
me & ::= & e(el)_n \mid r_n \\
e & ::= & v \mid e_1[e_2] \mid e_1 \oplus e_2 \mid e_1 == e_2 \mid e_1 < e_2 \mid e_1\ \textbf{and}\ e_2 \mid \\
 & & e_1\ \textbf{or}\ e_2 \mid \textbf{not}\ e \mid e(el)_1 \mid r_1 \\
v & ::= & c \mid f \mid \{\} \\
f & ::= & \textbf{fun}()\ b \mid \textbf{fun}(r)\ b \mid \textbf{fun}(\vec{x})\ b \mid \textbf{fun}(\vec{x}, r)\ b \\
b & ::= & s; \textbf{return}\ el \\
c & ::= & n \mid \text{“”} \mid \text{“}a_1 \ldots a_n\text{”} \mid \textbf{nil} \mid \textbf{true} \mid \textbf{false} \\
n & ::= & \textit{<decimal numerals>} \\
a & ::= & \textit{<characters>}
\end{array}
$$

Figure 3.3: Abstract Syntax

values), e are expressions, v are values, f are function constructors, b are function bodies, and the remaining categories are for literals. The expressions r_1 and r_n are *rest expressions*, to access variadic arguments (the formal parameter r in the function constructors is the variadic argument list). The notation $\vec{x}$ denotes the non-empty list $x_1, \ldots, x_n$.

We will give the typing rules as a deduction system for the typing relation $\Gamma \vdash t : \tau$. The relation means that the syntactical term t has type τ given the type environment Γ, which maps variables to types. We say $\Gamma[x \mapsto \tau]$ to mean environment Γ extended so it maps x to τ while leaving all other mappings intact.

We start with the rules for assignment. The main constraint of our typing system is that all valid Lua programs have to typecheck. The assignment

$$x, y = z + 2$$

is correct Lua code, where Lua at runtime *adjusts* the result of the expression list to have the same length as the number of lvalues, dropping extra values and using **nil** for missing ones. The rules for assignment, ASSIGN-DROP and ASSIGN-FILL, have to take adjustment into account:

$$\frac{\Gamma \vdash l_k : \tau_k \quad \Gamma \vdash el : \upsilon_1 \times \ldots \times \upsilon_m \quad m \geq |\vec{l}| \quad \upsilon_k \rightsquigarrow \tau_k}{\Gamma \vdash \vec{l} = el : \textbf{void}}$$

$$\frac{\Gamma \vdash l_k : \tau_k \quad \Gamma \vdash el : \upsilon_1 \times \ldots \times \upsilon_m \quad m < |\vec{l}| \quad \upsilon_k \rightsquigarrow \tau_k \quad \textbf{nil} \rightsquigarrow \tau_l \quad l > m}{\Gamma \vdash \vec{l} = el : \textbf{void}}$$

The ASSIGN-DROP rule ignores the types of extra values, and ASSIGN-

FILL uses **nil** as the types of missing values. Each value's type needs to be able to be coerced into the corresponding lvalue's type. The assignment itself has type **void**. The intuition behind the rule is that if there is more than one assignment to the same lvalue (the same variable, for example), then the type of the lvalue must be a type that all the values in the several assignments can be coerced to. In the worst case this means $\mathcal{D}$, but the job of the type inference will be to find a more precise type if it is available.

The typing rules for simple expression lists, EL-EMPTY and EL, are straightforward:

$$\frac{}{\Gamma \vdash \textbf{nothing} : \textbf{empty}}$$

$$\frac{\Gamma \vdash e_k : \tau_k \quad n = |\vec{e}|}{\Gamma \vdash \vec{e} : \tau_1 \times \ldots \times \tau_n}$$

Assignments with empty expression lists will use rule ASSIGN-FILL.

Adjustment is different in the special case where the last expression in a expression list is a function application or rest expression, as these can produce multiple values. In the assignment

$$x, y, z = a + 1, f(\textbf{empty})_n$$

if the function application produces no values then y and z will get **nil**, if it produces a single value then y will get this value and z will get **nil**, and if it produces two or more values then y and z will get the first two values produced and the rest is ignored.

First we will consider the case where the number of values the multi-expression produces is statically known, which is covered by rules EL-MEXP-EMPTY and EL-MEXP:

$$\frac{\Gamma \vdash e_k : \tau_k \quad \Gamma \vdash me : \textbf{empty} \quad n = |\vec{e}|}{\Gamma \vdash \vec{e}, me : \tau_1 \times \ldots \times \tau_n}$$

$$\frac{\Gamma \vdash e_k : \tau_k \quad \Gamma \vdash me : \upsilon_1 \times \ldots \times \upsilon_m \quad n = |\vec{e}|}{\Gamma \vdash \vec{e}, me : \tau_1 \times \ldots \times \tau_n \times \upsilon_1 \times \ldots \times \upsilon_m}$$

There are analogous rules MEXP-EMPTY and MEXP for when the expression list is just the multi-expression.

When the number of values the multi-expression produces is not statically known it will have type $\mathcal{D}^*$ or $\tau_1 \times \ldots \times \tau_n \times \mathcal{D}^*$. We need corresponding expression list rules EL-VAR-1 and EL-VAR-2:

$$\frac{\Gamma \vdash e_k : \tau_k \quad \Gamma \vdash me : \mathcal{D}^* \quad n = |\vec{e}|}{\Gamma \vdash \vec{e}, me : \tau_1 \times \ldots \times \tau_n \times \mathcal{D}^*}$$

$$\frac{\Gamma \vdash e_k : \tau_k \quad \Gamma \vdash me : \upsilon_1 \times \ldots \times \upsilon_m \times \mathcal{D}^* \quad n = |\vec{e}|}{\Gamma \vdash \vec{e}, me : \tau_1 \times \ldots \times \tau_n \times \upsilon_1 \times \ldots \times \upsilon_m \times \mathcal{D}^*}$$

We now add new rules ASSIGN-VAR-DROP and ASSIGN-VAR-FILL for assignment that will correctly handle variable-length expression lists:

$$\frac{\Gamma \vdash l_k : \tau_k \quad \Gamma \vdash el : \upsilon_1 \times \ldots \times \upsilon_m \times \mathcal{D}^* \quad m \geq |\vec{l}| \quad \upsilon_k \rightsquigarrow \tau_k}{\Gamma \vdash \vec{l} = el : \mathbf{void}}$$

$$\frac{\Gamma \vdash l_k : \tau_k \quad \Gamma \vdash el : \upsilon_1 \times \ldots \times \upsilon_m \times \mathcal{D}^* \quad m < |\vec{l}| \quad \upsilon_k \rightsquigarrow \tau_k \quad \tau_l = \mathcal{D} \quad l > m}{\Gamma \vdash \vec{l} = el : \mathbf{void}}$$

The rule ASSIGN-VAR-FILL covers the interesting case, and comes from our previous definition of $\mathcal{D}^*$ as a list of tagged values, so it is natural that the lvalues of $\mathcal{D}^*$ need to have type $\mathcal{D}$. In the assignment

$$x, y, z = a + 1, f(\mathbf{empty})_n,$$

if $f(\mathbf{empty})_n$ has type $\mathcal{D}^*$ then both y an z will have type $\mathcal{D}$.

Let us move to the rules for the typing of functions and function application. Lua also adjusts the length of argument lists to the number of formal parameters, so the code fragment (given in the abstract syntax of Figure 3.3) below is correct Lua code:

$$\begin{array}{l}\mathbf{local}\ f = \mathbf{fun}(x)\ \mathbf{return}\ x + 2\ \mathbf{in} \\ \quad \mathbf{local}\ g = \mathbf{fun}(x,\ y)\ \mathbf{return}\ x + y\ \mathbf{in} \\ \quad\quad \mathbf{local}\ h = g\ \mathbf{in} \\ \quad\quad\quad \mathbf{if}\ z\ \mathbf{then\ return}\ h(2,3)\ \mathbf{else}\ h = f;\ \mathbf{return}\ h(3,2)\end{array}$$

One way the above code fragment can typecheck, given the typing and coercion rules we have until now, is to have the type of h be $\mathcal{D}$ while f has type $\mathcal{D} \rightarrow \mathcal{D}^*$ and g has type $\mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}^*$, both types coercible to $\mathcal{D}$. But ideally we want the possibility of more precise types. A solution is to have h, f and g all have the same type, $\mathbf{Number} \times \mathbf{Number} \rightarrow \mathbf{Number}$. This is possible with the following rules, FUN-EMPTY and FUN, for (non-variadic) function definitions, so the type of the function's domain can have more components than the number of formal parameters:

$$\frac{\Gamma \vdash s; \mathbf{return}\ el : \upsilon}{\Gamma \vdash \mathbf{fun}()\ s; \mathbf{return}\ el : \tau_1 \times \ldots \times \tau_n \rightarrow \upsilon}$$

$$\frac{\Gamma[\vec{x} \mapsto \vec{\tau}] \vdash s; \mathbf{return}\ el : \upsilon}{\Gamma \vdash \mathbf{fun}(\vec{x})\ s; \mathbf{return}\ el : \tau_1 \times \ldots \times \tau_n \rightarrow \upsilon \quad n \geq |\vec{x}|}$$

The typing of a function application depends on the type of the function expression, whether it is a non-variadic function type, a variadic function type,

or other type that can be coerced to $\mathcal{D}$. The first case is similar to typing an assignment, and is covered by rules APP-DROP, APP-FIL, APP-VAR-DROP, and APP-VAR-FILL:

$$\frac{\Gamma \vdash f : \tau_1 \times \ldots \times \tau_n \to \sigma \quad \Gamma \vdash el : \upsilon_1 \times \ldots \times \upsilon_m \quad m \geq n \quad \upsilon_k \rightsquigarrow \tau_k}{\Gamma \vdash f(el)_n : \sigma}$$

$$\frac{\begin{array}{c}\Gamma \vdash f : \tau_1 \times \ldots \times \tau_n \to \sigma \\ \Gamma \vdash el : \upsilon_1 \times \ldots \times \upsilon_m \quad m < n \quad \upsilon_k \rightsquigarrow \tau_k \quad \mathbf{nil} \rightsquigarrow \tau_l \quad l > m\end{array}}{\Gamma \vdash f(el)_n : \sigma}$$

$$\frac{\Gamma \vdash f : \tau_1 \times \ldots \times \tau_n \to \sigma \quad \Gamma \vdash el : \upsilon_1 \times \ldots \times \upsilon_m \times \mathcal{D}^* \quad m \geq n \quad \upsilon_k \rightsquigarrow \tau_k}{\Gamma \vdash f(el)_n : \sigma}$$

$$\frac{\begin{array}{c}\Gamma \vdash f : \tau_1 \times \ldots \times \tau_n \to \sigma \\ \Gamma \vdash el : \upsilon_1 \times \ldots \times \upsilon_m \times \mathcal{D}^* \quad m < n \quad \upsilon_k \rightsquigarrow \tau_k \quad \tau_l = \mathcal{D} \quad l > m\end{array}}{\Gamma \vdash f(el)_n : \sigma}$$

Similar rules cover $f(el)_0$, where the type of the application is always **void**, and $f(el)_1$, where the type is **nil** if $\Gamma \vdash f(el)_n : \mathbf{empty}$, $\mathcal{D}$ if $\Gamma \vdash f(el)_n : \mathcal{D}^*$, and τ_1 if $\Gamma \vdash f(el)_n : \tau_1 \times \ldots \times \tau_n$ or $\Gamma \vdash f(el)_n : \tau_1 \times \ldots \times \tau_n \times \mathcal{D}^*$.

The return type of a function depends on the types of **return** expression lists in the function body. We use a trick where the type of a block with no **return** statements has type **void**, but a block with a return statement has the type of the **return** statement. For blocks with more than one **return** statements we give the same type to all the **return** statements using the rule RETURN:

$$\frac{\Gamma \vdash el : \tau \quad \tau \rightsquigarrow \upsilon}{\Gamma \vdash \mathbf{return}\ el : \upsilon}$$

Figure 3.2 has the coercion rules for tuples, derived from how adjustment works. The last two coercion rules cover the case where a function must have return type $\mathcal{D}^*$ because the function has to be coerced into $\mathcal{D}$.

Function application when the type of the function expression is not a function type is typed by rule APP-DYN:

$$\frac{\Gamma \vdash f : \tau \quad \Gamma \vdash el : \upsilon \quad \tau \rightsquigarrow \mathcal{D} \quad \upsilon \rightsquigarrow \mathcal{D}^*}{\Gamma \vdash f(el)_n : \mathcal{D}^*}$$

The rule means that the expression list can be any expression list that produces tagged values (or values that can be coerced into tagged values), and the application can return any number of tagged values.

Typing tables has similarities to typing functions. The typing for a table constructor {} depends on how the rest of the program uses the tables created by that constructor. The type system also needs to be flexible enough to let

the type inference algorithm synthesize precise enough types even when the same expression can evaluate to different functions, or tables from different constructors. Take the code below, where x can be a table created by the first or the second table constructor:

$$\begin{array}{l}\textbf{local } f = \textbf{fun}(x) \textbf{ return } x.\text{foo } \textbf{in} \\ \quad \textbf{local } a = \{\} \textbf{ in} \\ \quad\quad \textbf{local } b = \{\} \textbf{ in} \\ \quad\quad\quad a.\text{foo} = 3;\ a.\text{bar} = \text{“}s\text{”};\ b.\text{foo} = 5;\ \textbf{return } f(a), f(b)\end{array}$$

We will have the first and second table constructors (and, by extension, variables a and b) sharing the same precise type $\text{“}foo\text{”} \mapsto \textbf{Number}? \wedge \text{“}bar\text{”} \mapsto \textbf{String}?$. We are trading precision for possibly greater memory consumption in the representation of the tables created by the second table constructor (because of the unused “bar” field).

These typing rules for the table constructor are CONS and CONS-DYN:

$$\frac{\forall i, j, \sigma.(i \neq j \wedge \sigma \rightsquigarrow \tau_i) \rightarrow \sigma \not\rightsquigarrow \tau_j \quad \textbf{nil} \rightsquigarrow \upsilon_k}{\Gamma \vdash \{\} : \tau_1 \mapsto \upsilon_1 \wedge \ldots \wedge \tau_n \mapsto \upsilon_n}$$

$$\frac{\textbf{nil} \rightsquigarrow \upsilon}{\Gamma \vdash \{\} : \mathcal{D} \mapsto \upsilon}$$

They basically restate the rules for creating table types in our type language given on Figure 3.1, with the added restriction that **nil** has to be coercible to any type used as a value. This added restriction comes from the behavior of Lua tables where indexing a non-existent key returns **nil** instead of being an error. Without this restriction the type system becomes unsound, as we could type as τ (with $\textbf{nil} \not\rightsquigarrow \tau$) an expression that evaluates to **nil** at runtime. Lua has other kinds of table constructors that can lift this restriction in some cases, and in the end of this section we discuss a nuance of Lua's semantics that, while not removing this restriction, at least lessens its effects in most Lua programs.

Indexing a table uses the rule INDEX:

$$\frac{\Gamma \vdash e_1 : \tau_1 \mapsto \upsilon_1 \wedge \ldots \wedge \tau_n \mapsto \upsilon_n \quad \Gamma \vdash e_2 : \sigma \quad \sigma \rightsquigarrow \tau_k}{\Gamma \vdash e_1[e_2] : \upsilon_k}$$

The restriction on the types of table keys guarantees that τ_k is unique. The INDEX rule types both indexing expressions and indexing assignments (indexing in lvalue position), although we will see in the next section that they are treated differently by the type inference algorithm.

There is also an INDEX-DYN rule for indexing non-tables, analogous to the APP-DYN rule:

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \upsilon \quad \tau \rightsquigarrow \mathcal{D} \quad \upsilon \rightsquigarrow \mathcal{D}}{\Gamma \vdash e_1[e_2] : \mathcal{D}}$$

The $\mathbf{nil} \rightsquigarrow \upsilon$ restriction on types of table values means that expressions such as $t[e_1][e_2]$ cannot have precise types using just the rules we gave, as $t[e_1]$ cannot have a type τ where τ is a table type in our typing rules; the closest to a table type $t[e_1]$ can have is $\tau?$ where τ is a table type. So the expression $t[e_1][e_2]$ always has to use the INDEX-DYN rule. This is how Lua's semantics work in the general case, as the user can extend the behavior of the **nil** value. But in practice extending the behavior of **nil** in this manner is forbidden (the user has to use Lua's `debug` library for that), because changing the behavior of **nil** can break library and third-party code that depends on the standard behavior. So it is safe to add rules to get precise type inference for expressions such as $t[e_1][e_2]$ (and expressions such as the $t[e](el)$ application), like INDEX-NIL:

$$\frac{\Gamma \vdash e_1 : (\tau_1 \mapsto \upsilon_1 \wedge \ldots \wedge \tau_n \mapsto \upsilon_n)? \quad \Gamma \vdash e_2 : \sigma \quad \sigma \rightsquigarrow \tau_k}{\Gamma \vdash e_1[e_2] : \upsilon_k}$$

This last rule is type safe, as the **nil** in values of type $\tau?$ is the untagged **nil**, which the compiler can make sure has the standard **nil** behavior.

The complete set of typing rules is in Appendix B. In the next section we will outline part of the type inference algorithm based on these rules.

3.1.5 Type Inference

The type system we outlined in the previous section allows us to assign more precise types to a Lua program than just $\mathcal{D}$, and lets us check if these types lead to a well-typed program (assigning type **Number** to an expression that can possibly hold a string at runtime is against the typing rules, for example). But the type system is not constructive: it can only check if a typing is valid, not produce one. Assigning valid and precise types to programs is the task of our type inference algorithm.

Our type inference algorithm finds types for the program's variables and expressions by recursively trying to apply the typing rules with *type variables* instead of just types. A type variable is a reference to a type (or another type variable), and the type the variable refers to can change during the course of the inference, and this change is always from more precise to less precise types. The algorithm proceeds from the root node of the program's abstract

syntax tree to its leafs, using the typing rules and an update procedure for type variables that is based on the coercion relation of Section 3.1.2. Several syntactical constructs have different typing rules, and the algorithm has to choose one based on information that may later change. The algorithm does multiple passes over the syntax tree until no type variables have changed (we say that the types have *converged*). We will later see this gives us the benefit of a straightforward implementation of aliasing for function and table types when our type inference has to force different functions or tables to have the same type.

In the exposition of the algorithm below we always represent type variables with upper-case letters, with $\mathcal{V}(X)$ being the value that the type variable X refers to and $X := \tau$ an update of type variable X. A fresh (unassigned) variable has the special value ε. During the course of the inference we need to change table types, adding or removing pairs of key and value types. To make it easier to follow the algorithm, we use different letters to indicate invariants that some type variables can have. We use T and U for table types. Different table constructors may need to have the same type, so T always holds another type variable which we will call P or Q. So the following always holds for table types:

$$\begin{aligned} \mathcal{V}(T) &= P \\ \mathcal{V}(P) &= \tau_1 \mapsto X_1 \wedge \ldots \wedge \tau_n \mapsto X_n. \end{aligned}$$

Similarly, we use the letter F for function types. Functions need a similar indirection for the types of their return values, and we use the letters R and S for the type variable used for the return type. So the following always holds for function types:

$$\begin{aligned} \mathcal{V}(F) &= X_1 \times \ldots \times X_n \rightarrow \times R \\ \mathcal{V}(R) &= Y_1 \times \ldots \times Y_n \text{ or } Y_1 \times \ldots \times Y_n \times \mathcal{D}^* \text{ or } \mathcal{D}^* \end{aligned}$$

Each syntactical term t has an implicit type variable that we will refer as $[[t]]$.

Let us now present an example of type inference for the fragment

$$\begin{array}{l} \textbf{local } f = \textbf{fun}(x) \textbf{ return } x.\text{foo } \textbf{in} \\ \quad \textbf{local } a = \textbf{\{\}} \textbf{ in} \\ \quad\quad \textbf{local } b = \textbf{\{\}} \textbf{ in} \\ \quad\quad\quad a.\text{foo} = 3; a.\text{bar} = \text{“}s\text{”}; b.\text{foo} = 5; \textbf{return } f(a), f(b) \end{array}$$

that we used in the last section. In the first iteration we have the function

definition getting a fresh function type F with $\mathcal{V}(F) = X_{F_1} \rightarrow \times R$, and F gets assigned to f. The first table constructor gets a fresh table type T with $\mathcal{V}(T) = P$, and T gets assigned to a, while the second table constructor gets a fresh table type U with $\mathcal{V}(U) = Q$, and U gets assigned to b. After the first assignment statement we have $\mathcal{V}(P) = \text{“}foo\text{”} \mapsto X_{P_1}$ and $\mathcal{V}(X_{P_1}) = 3$. After the second assignment we have $\mathcal{V}(P) = \text{“}foo\text{”} \mapsto X_{P_1} \wedge \text{“}bar\text{”} \mapsto X_{P_2}$ with $\mathcal{V}(X_{P_2}) = \text{“}s\text{”}$. After the third statement we have $\mathcal{V}(Q) = \text{“}foo\text{”} \mapsto X_{Q_1}$ and $\mathcal{V}(X_{Q_1}) = 5$. After Processing the first expression in the expression list of the last statement we have $\mathcal{V}(X_{F_1}) = T$, as $\mathcal{V}(X_{F_1})$ was ε. After processing the second expression we have aliasing of T and U, so we have $\mathcal{V}(T) = \mathcal{V}(U) = P'$ where P' is a fresh type variable. The expression list produces $\varepsilon \times \varepsilon$.

In the second iteration we have $\mathcal{V}(X_{F_1}) = U$ (which is now an alias of T), so the indexing expression in the function body now sets P' to $\text{“}foo\text{”} \mapsto X_{P'_1}$ but still has type ε, so R is still ε. The first table constructor now makes $\mathcal{V}(X_{P'_1}) = \textbf{nil}$ so T respects the invariant of table types. The second table constructor does not change anything, as U is an alias of T. The first assignment updates $X_{P'_1}$ to **Number**?, as **Number?** is the most precise type that both **nil** and 3 can be coerced to. After the second statement we have $\mathcal{V}(P') = \text{“}foo\text{”} \mapsto X_{P'_1} \wedge \text{“}bar\text{”} \mapsto X_{P'_2}$ and $\mathcal{V}(X_{P'_2}) = \text{“}s\text{”}$. The third assignment now does not change anything as $5 \rightsquigarrow$ **Number**?. There is no more aliasing in the last statement as both T and U have the same value P', but the type of the expression list is still $\varepsilon \times \varepsilon$.

In the third iteration we still have $\mathcal{V}(X_{F_1}) = U$, but the indexing expression in the function body now has type **Number**?, so $\mathcal{V}(R) =$ **Number**?. The first table constructor changes $X_{P'_2}$ from “s” to **String**?, to restore the invariant of table types. The second table constructor does not change anything. The three assignments now do not change anything either, as $3 \rightsquigarrow$ **Number**?, “s” $\rightsquigarrow$ **String**? and $5 \rightsquigarrow$ **Number**?. In the last statement the type of the expression list now is **Number**? $\times$ **Number**?.

In the fourth iteration no type variables change, and the algorithm stops. In the final assignments (eliminating the type variables) we have f with type (“foo” $\mapsto$ **Number**? $\wedge$ “bar” $\mapsto$ **String**?) $\rightarrow$ **Number**? and both a and b having type “foo” $\mapsto$ **Number**? $\wedge$ “bar” $\mapsto$ **String**?. The whole fragment has type **Number**? $\times$ **Number**?. It is straightforward to check that this is a correct typing in our type system.

The entry point of the algorithm is the procedure INFER:

```
1: procedure INFER(root)
2:     Γ := {}
3:     repeat
```

```
4:          INFERSTEP(Γ, root)
5:      until converge
6:  end procedure
```

Procedure INFERSTEP corresponds to one iteration of the type inference algorithm, taking a type environment, which is a mapping from identifiers to type variables, and a syntactical term. We will give parts of its definition using pattern matching on terms to simplify the exposition. Let us start with the definition of INFERSTEP for assignment statements, covering rules ASSIGN-DROP and ASSIGN-FILL:

```
1:  procedure INFERSTEP(Γ, ⟨l_1, ..., l_n = el⟩)
2:      INFERSTEP(Γ, el)
3:      let ⟨υ_1 × ... × υ_m⟩ = V([[el]])
4:      if m ≥ n then
5:          for k := 1, n do
6:              INFERSTEP(Γ, l_k)
7:              UPDATE(υ_k, V([[l_k]]))
8:          end for
9:      else
10:         for k := 1, m do
11:             INFERSTEP(Γ, l_k)
12:             UPDATE(υ_k, V([[l_k]]))
13:         end for
14:         for k := m + 1, n do
15:             INFERSTEP(Γ, l_k)
16:             UPDATE(nil, V([[l_k]]))
17:         end for
18:     end if
19:     [[l_1, ..., l_n = el]] := void
20: end procedure
```

All definitions of INFERSTEP follow a similar structure, derived from the typing rule it implements. In the definition above, for type inference of assignments, we begin by recursively inferring the type of the expression list. If there are more rvalues than lvalues, we recursively infer the type for each lvalue, and update its type variable (we will see that $\mathcal{V}([[l_k]])$ is always a type variable) with the type of the corresponding rvalue, and ignore the others. This corresponds to rule ASSIGN-DROP. If there are more lvalues than rvalues, we do the above, and update the type variables of any remaining lvalues with **nil**. This corresponds to rule ASSIGN-FILL. Extending the definition of

INFERSTEP given above to cover rules ASSIGN-VAR-DROP and ASSIGN-VAR-FILL is straightforward.

This is the definition of INFERSTEP for simple expression lists:

```
1: procedure INFERSTEP(Γ, ⟨e_1, ..., e_n⟩)
2:     for k := 1, n do
3:         INFERSTEP(Γ, e_k)
4:     end for
5:     [[e_1, ..., e_n]] := V([[e_1]]) × ... × V([[e_n]])
6: end procedure
```

The definition above recursively infers the types of each expression in the expression list and assigns a tuple of these types as the type of the expression list. The definition implements typing rule EL.

The UPDATE(τ, X) procedure is the core of the type inference algorithm. This procedure updates X from its current value υ to a υ' so that $\tau \overset{\mathcal{V}}{\rightsquigarrow} \upsilon'$ and $\upsilon \overset{\mathcal{V}}{\rightsquigarrow} \upsilon'$, where $\overset{\mathcal{V}}{\rightsquigarrow}$ is the coercion relation lifted for type variables. For example, this is the definition of UPDATE when $\tau =$ **nil**, used in INFERSTEP for assignments:

```
1: procedure UPDATE(nil, X)
2:     match V(X) with
3:         case ε: X := nil
4:         case n: X := Number?
5:         case s: X := String?
6:         case true | false: X := Bool?
7:         case D | τ?: break
8:         otherwise: X := V(X)?
9:     end match
10: end procedure
```

The first case, where X is unassigned, is a common case for all UPDATE definitions. Then come three cases where X holds a singleton type, so we update X to the corresponding nullable type. Then comes the case where **nil** $\overset{\mathcal{V}}{\rightsquigarrow} X$ already holds, with X holding $\mathcal{D}$ or a nullable type, so we do nothing. For other values of X we update X so it holds the corresponding nullable type.

Another case of UPDATE is the one where $\mathcal{V}(X)$ is $\mathcal{D}$. In this case, the UPDATE does nothing if t is a scalar type, as $\tau \rightsquigarrow \mathcal{D}$ already holds for all scalar types. The interesting subcases are where τ is a function or table type. This is the definition for τ as a function type:

```
1: procedure UPDATE(F, ⟨X when V(X) = D⟩)
```

2: **let** $\langle Y_1 \times \ldots \times Y_n \to R \rangle = \mathcal{V}(F)$
3: **for** $k := 1, n$ **do**
4: $Y_k := \mathcal{D}$
5: **end for**
6: $R := \mathcal{D}^*$
7: **end procedure**

The only way to let a function type be coerced to $\mathcal{D}$ is to have all its parameters have type $\mathcal{D}$ and its return type be $\mathcal{D}^*$, so we force the function type to be $\mathcal{D} \times \ldots \times \mathcal{D} \to \mathcal{D}^*$. It is easier to see that the above procedure works if we examine part of INFERSTEP for function definitions:

1: **procedure** INFERSTEP($\Gamma, \langle$**fun**$(x_1, \ldots, x_n)\ s;$ **return** $el\rangle$)
2: **if** $\mathcal{V}([[$**fun**$(x_1, \ldots, x_n)\ s;$ **return** $el]]) = \varepsilon$ **then**
3: **let** $R =$ **newvar**
4: **let** $\tau = \underbrace{\textbf{newvar} \times \ldots \times \textbf{newvar}}_{n} \to R$
5: **let** $F =$ **newvar** τ
6: $[[$**fun**$(x_1, \ldots, x_n)\ s;$ **return** $el]] := F$
7: **end if**
8: **let** $F = \mathcal{V}([[$**fun**$(x_1, \ldots, x_n)\ s;$ **return** $el]])$
9: **let** $\langle X_1 \times \ldots \times X_m \to R \rangle = \mathcal{V}(F)$
10: INFERSTEP($\Gamma[x_1 \mapsto X_1, \ldots, x_n \mapsto X_n, rv \mapsto R], \langle s;$ **return** $el\rangle$)
11: **end procedure**

In the above INFERSTEP procedure we construct an initial function type if this is the first iteration; this function type has the structure we outlined in the beginning of this section (and the structure that the UPDATE procedure we gave above expects). Then we deconstruct the function type to get the type variables for each parameter and for the return values, and recursively infer types in the body using an extended type environment. We inject the return type variable in the environment so type inference for statements (and in particular **return** statements) can change the return type of the enclosing function directly. Notice the parallel with the rule FUN we gave in the previous section.

Another interesting UPDATE(τ, X) subcase when $\mathcal{V}(X) = \mathcal{D}$ is the subcase for table types:

1: **procedure** UPDATE($T, \langle X$ when $\mathcal{V}(X) = \mathcal{D}\rangle$)
2: **let** $P = \mathcal{V}(T)$
3: $P := \mathcal{D} \mapsto ($**newvar** $\mathcal{D})$
4: **end procedure**

In the above procedure we are forcing the table to have type $\mathcal{D} \mapsto \mathcal{D}$. We use a new type variable to hold the second $\mathcal{D}$ to respect the structure for table types that we gave in the beginning of the section. Again, it is easier to understand UPDATE if we examine INFERSTEP for table constructors:

```
1: procedure INFERSTEP(Γ, ⟨{}⟩)
2:     if V([[{}]]) = ε then
3:         let P = newvar
4:         let T = newvar
5:         let T := P
6:         [[{}]] := T
7:     end if
8:     let T = V([[{}]])
9:     let P = V(T)
10:    if V(P) ≠ ε then
11:        let ⟨τ1 ↦ X1 ∧ ... ∧ τn ↦ Xn⟩ = V(P)
12:        for k := 1, n do
13:            UPDATE(nil, Xk)
14:        end for
15:    end if
16: end procedure
```

Like we did for function definitions, we make an empty table type with the structure we outlined in the beginning of this section if this is the first iteration. We then deconstruct the table type and enforce the invariant that **nil** has to be able to be coerced to the types of the table's values.

In the last section we discussed how the type system allows us to keep precise types for functions and tables even if different function definitions need to have the same type. In the type inference algorithm it is the job of the UPDATE procedure to unify different function types (and table types) when this occurs. The iterative nature of our algorithm and the structure we use for these types make this a simple procedure, though; we can build a new fresh type for the aliased types, and just make sure for function types that the new type preserves the invariant that we have at least as many parameter types than formal parameters. This is the aliasing UPDATE for function types:

```
1: procedure UPDATE(F, ⟨X when V(X) = G⟩)
2:     if V(F) ≠ V(G) then
3:         let ⟨X1 × ... × Xm → R⟩ = V(F)
4:         let ⟨Y1 × ... × Yn → S⟩ = V(G)
5:         let k = max m, n
```

```
6:        let R' = newvar
7:        G := newvar × ... × newvar (k) → R'
8:        F := V(G)
9:    end if
10: end procedure
```

In the above procedure we make F and G hold the same fresh type variables for parameters and return types, effectively aliasing F and G. The aliasing UPDATE for table types is simpler:

```
1: procedure UPDATE(↦ T, ⟨X when V(X) = U⟩)
2:    if V(T) ≠ V(U) then
3:        let P = newvar
4:        T := P
5:        U := P
6:    end if
7: end procedure
```

Again, we just make the two table types hold the same (fresh) type variable, effectively aliasing T and U.

The INFERSTEP procedure for function application is analogous to assignment. The INFERSTEP procedure for indexing operations is more interesting, as it is the procedure responsible for enforcing the invariants on types of table keys. This is part of the INFERSTEP procedure for indexing in lvalue position:

```
1: procedure INFERSTEP(Γ, ⟨l when l = ⟨e1[e2]⟩⟩)
2:    INFERSTEP(Γ, e1)
3:    INFERSTEP(Γ, e2)
4:    match V([[e1]]) with
5:        case T:
6:            match FIND(V([[e2]]), V(T)) with
7:                case ε:
8:                    let X = newvar
9:                    UNION(V([[e2]]) ↦ X, V(T))
10:                   [[l]] := X
11:               end
12:               case X: [[l]] := X
13:           end match
14:       end
15:       case otherwise:
16:           UPDATE(V([[e1]]), newvar D)
```

```
17:             UPDATE(V([[e2]]), newvar D)
18:             [[l]] := newvar D
19:         end
20:     end match
21: end procedure
```

In the indexing expression $e_1[e_2]$, if the type of e_1 is a table type T, we try to find which the type for values with keys of type $\mathcal{V}([[e_2]])$. This uses the auxiliary function FIND. FIND(υ, P)searches for a pair $\tau_k \mapsto X$ in P so that $\upsilon \overset{\mathcal{V}}{\rightsquigarrow} \tau_k$. If there is such a pair then FIND returns X, otherwise it returns ε. If FIND returns ε then we extend the table's type with the new key type and a fresh type variable, using the auxiliary procedure UNION$(\langle \upsilon \mapsto X \rangle, P)$:

```
1: procedure UNION(⟨υ ↦ X⟩, P)
2:     if V(P) = ε then
3:         P := υ ↦ X
4:     else
5:         let ⟨τ1 ↦ X1 ∧ ... ∧ τn ↦ Xn⟩ = V(P)
6:         let P' = newvar (υ ↦ X)
7:         for k := 1, n do
8:             if τk ⇝ υ then
9:                 UPDATE(V(Xk), X)
10:            else
11:                P' := V(P') ∧ τk ↦ Xk
12:            end if
13:        end for
14:        P := V(P')
15:    end if
16: end procedure
```

The implementations of FIND and $\tau \overset{\mathcal{V}}{\rightsquigarrow} \upsilon$ are straightforward. The implementation of $\overset{\mathcal{V}}{\rightsquigarrow}$ has to alias its arguments if they are both function or table types, otherwise aliasing in other places may break the invariants on key types.

The INFERSTEP procedure for indexing in rvalue position is similar to INFERSTEP for indexing in lvalue position: we just replace $[[l]] := X$ with $[[e_1[e_2]]] := \mathcal{V}(X)$ and $[[l]] :=$ **newvar** $\mathcal{D}$ with $[[e_1[e_2]]] := \mathcal{D}$. INFERSTEP for indexing can also be trivially extended to cover the alternative INDEX-NIL rule we presented in the last section.

In the next section we will show how the types inferred by the type inference algorithm can lead to a variation of the compiler we presented on

Chapter 2 that generates exploits type information to generate more efficient code.

3.2 Compiling Types

This section presents a variation on the "intern" compiler we presented in Section 2.2.3. Its code generator uses information extracted by the type inference algorithm to generate optimized representations for Lua's values and specialized code for Lua's operations.

The representation of tagged types (corresponding to type $\mathcal{D}$) is unchanged, so any operation that involves values of type $\mathcal{D}$ continues generating the same code as before. We then have two issues to tackle: first, representation of untagged types, second; code generation for operations on untagged types and coercions.

3.2.1 Untagged Representations

Representing singleton types and simple types is straightforward, as each of them has an analogue in the CLR: **Number** is `double`, **String** is `Lua.Symbol`, each numeric and string singleton type has a corresponding literal, and **true**, **false**, and **Bool** are respectively the literals `true`, `false`, and the CLR type `bool`. The singleton type **nil** is the special value `null`.

We represent function types using a pair of *delegate* types, which are CLR's analogue of function types. We use two types instead of just one so we can keep the optimization of Section 2.2.1 for function calls that only need a single value; one delegate type returns a single value, the other delegate type returns a tuple. Function return values are the only place we need a representation of a tuple type, as tuples are not first class values. We represent a tuple type $\tau_1 \times \ldots \times \tau_n \times \mathcal{D}^*$ with a CLR class having a field v_k of type corresponding to the representation of type τ_k for each type in the static part of the tuple, and having a member r of type `object`[] for the dynamic part of the tuple.

We represent the actual functions as CLR classes that implement one "invoke" method for each of the two delegate types that represent the function's type, and have fields for the function's display. Functions with types that can be coerced to $\mathcal{D}$ also subclass the `Function` type of dynamic Lua functions.

The representations of table types are CLR classes, but the specifics depend on their characteristics. For each singleton key type τ_k with corresponding

value type v_k we have a field v_{τ_k} of type corresponding to the representation of type v_k. This is the *record* part of the table.

If the table has **Number** $\mapsto v$ as part of its type, its CLR class has a member a of type `SparseArray<T>`, where `T` is the representation of type v. The `SparseArray<X>` type is a polymorphic specialization of Lua tables (with an array part and a hash part) for numeric keys, parametrized over the type of its values.

If the table has **String** $\mapsto v$ as part of its type, its CLR class has a member s of type `Dictionary<Lua.Symbol, T>`, where `T` is the representation of type v. `Dictionary` is the CLR's type for polymorphic hash tables (from its base class library).

For **Bool** $\mapsto v$ we do the same as if the table has both **true** $\mapsto v$ and **false** $\mapsto v$ as part of its type, and generate code accordingly. A table with $\tau \mapsto v$ as part of its type, where τ is a function type, gets a field f of type `Dictionary<Delegate, T>` where `T` is the representation of type v. If the table has $\tau \mapsto v$ as part of its type, where τ is a table type, it gets a field t of type `Dictionary<T, U>`, where `T` and `U` are the representation of types τ and v, respectively.

Tables of type $\mathcal{D} \mapsto v$ are represented by the CLR class `HTable<T>`, where `T` is the representation of type v. Class `HTable<X>` is a polymorphic version of `Table`, the class of dynamic Lua tables, that implements the same protocol as `Table` plus methods for accessing elements parametrized by type X.

We can use the same types for both τ and $\tau?$ in most cases, because the CLR's reference semantics allows `null` as a valid value for any reference type. The exception is **Number**?, because `double` is a value type. In this case, we use the boxed version of `double`, a reference type.

CLR's structured reference types (which include classes and delegates) are naturally recursive, so representing recursive types is straightforward.

3.2.2 Code Generation

It is straightforward to generate code for Lua operations that uses the type information. For each operation there is a *fast* case, where the operation has simple semantics for the types involved, like arithmetic with numbers, indexing with records, and applications with functions, and a *dynamic* case where you coerce the operands to $\mathcal{D}$ and then do the dynamically dispatched operation. Generating code for the dynamic dispatch is the same as for the compiler on Section 2.2.3.

In the fast case the code for the operation is often just a single CIL instruction, as in the two examples in the beginning of Section 3.1. In some cases the code for the operation itself is a no-op, like **and** or **or** with a first operand that is known to be neither **nil** nor **false**. There are also some edge cases like arithmetic operations with one number operand and one operand of other type, where we can generate better specialized code than just naively treating it as the dynamic case.

The code for simple coercions is a no-op, as the representation of a singleton type is the same as the simple type they can be coerced to. Nullable coercions for most types are also no-ops, due to CLR reference semantics regarding **null**, and coercion from **Number** to **Number**? is just boxing.

Coercion to $\mathcal{D}$ is a no-op for tables and functions, boxing for numbers, wrapping in the `Symbol` type for strings, and selecting the corresponding singleton value for **Bool** and **nil**. Coercing nullable types to $\mathcal{D}$ is a no-op for **Number**? and the same operation as coercing the non-nullable type for the other types.

Tuples are not first class values, so they usually only have ephemeral existence in the CLR evaluation stack. Tuple coercions then involve coercing individual tuple elements as we pushed on the stack, and then pushing additional elements as needed. In cases where we have to create a tuple object so we can return it as the result of a function call, we generate the code for the tuple coercion, then call the corresponding tuple object constructor. The typing rules guarantee that these cases only occur when generating code for **return**.

3.3 Related Work

This section is a review of some of the previous approaches for extracting type information from dynamically typed programs. We divide the approaches in two, presented in this order: type inference and flow analysis. Type inference approaches work directly on the abstract syntax tree of a program, and assign a type in a formally defined type language to each syntactical term. Flow analysis works on a *control flow graph*, built incrementally from an entry point in the program (using the nodes in the syntax tree and the information obtained by the analysis as input), and tracks the flow of abstract values through this graph; types are just a kind of abstract value.

The type information discovered by flow analysis is always used for optimization, by eliminating runtime type checks and usually also optimizing representation of values. This is not the case for the type inference approaches

we review in this section; some of them have the goal of optimizing the program, like the type inference we presented in this chapter and the flow analysis approaches, but most are primarily for checking types to discover potential runtime errors. In the section where we review the type inference approaches we note which of these goals (optimization or checking) is the primary goal for each approach.

3.3.1 Type Inference

Type inference algorithms for dynamically typed languages are not new. Gomard [1990] describes a two-level lambda calculus with a monomorphic type system that adds a type *untyped* (similar to our type $\mathcal{D}$), and an annotated version of each primitive operation that works on values of this type. He also presents an extension of the unification-based algorithm W [Damas and Milner, 1982] that, on a type error (unification failure), annotates the primitive where the error occurred and retries the algorithm until it succeeds. One possible application he gives for this modified algorithm W is to avoid doing type checks in dynamically typed code.

His type system forces these annotations to propagate to subexpressions, though, while our coercions can be localized to only part of an expression, increasing precision. Our type system also has a richer type language that cannot be fitted into his framework without losing precision. Extending his framework to support the same level of precision we achieve would lead to a more complex type system and inference algorithm that would be harder to understand.

Global tagging optimization [Henglein, 1992b,a] adds a type `Dynamic` (similar to our type $\mathcal{D}$) to a fairly complete subset of Scheme extended with coercions, and a type inference algorithm that finds out which coercions give the most precise types for a program using an extension of unification, and it is used to generate code for Scheme that uses more efficient data representations and avoids type checking. The algorithm can be implemented with a single pass over the program. His type system has polymorphic primitives but all inferred types are monomorphic. The algorithm is efficient and reasonably simple, but is very specific to its type system, and cannot accommodate Lua's ad-hoc polymorphic primitives or adjustment of expression lists.

The type system and inference algorithm in Henglein and Rehof [1995] extends the work of Henglein [1992b,a] with polymorphism for inferred types and modular type inference (it can infer types of functions without knowing how they are used), by incorporating polymorphic coercions as part of a

function's polymorphic type. These polymorphic coercions have coercion parameters that are analogous to type parameters in polymorphic types. The type system also replaces the type `Dynamic` of Henglein [1992b] with a sum type where each type constructor in the type language appears once and only once. The goal is code optimization through better data representation and avoidance of runtime checks, but the use of polymorphism requires generation of specialized code.

The inference algorithm in Henglein and Rehof [1995] is still based on unification, but has a complex intermediate step between unification and generalization of type variables; this intermediate step simplifies the parameters of the polymorphic coercions. Without this extra step the number of parameters to be generalized can be exponential on the size of the function. Polymorphic coercion parameters are an elegant way of combining parametric polymorphism with dynamic typing, but it is not useful in our case, as a polymorphic type system is a poor fit for Lua due to the reasons we gave on Section 3.1.

Aiken and Fähndrich [1995] give an alternative formulation for Henglein's global tagging optimization [Henglein, 1992b], which also has the goal of optimizing representations. They model coercions with subtyping constraints on a type system where each type has a structural part and a tagging part, and the structural part allows both union and intersection types. The inference algorithm generates a set of constraints from the program's syntax tree, solves these constraints, and maps these back to coercions. The algorithm is more general than the algorithm in Henglein [1992b], and can accommodate richer (but still monomorphic) type systems at the cost of cubical instead of linear time complexity, but the constraint language cannot express the ad-hoc polymorphism in our type system without sacrificing precision.

Soft typing [Cartwright and Fagan, 1991] presents a type system and inference algorithm for a functional language that has polymorphic types and union types (using an encoding of sum types as polymorphic types so they work with regular unification), where the inference algorithm inserts runtime type checks when algorithm W finds a type error.

Wright and Cartwright [1997] have a more sophisticated soft typing system for full Scheme. This system has extensions to deal with features present in Scheme but not in the idealized functional language used in Cartwright and Fagan [1991], including imperative features, and not only inserts runtime type checks, but also flags applications of primitives that can never succeed, and tries to expose readable types to the programmer (the encoding used for union types in Cartwright and Fagan [1991] makes it harder to understand what the

types mean).

The primary goal of both soft typing approaches is to find possible errors in programs by exposing the necessary runtime checks to the programmer; optimizing the program by removing unnecessary runtime checks is a side effect. Soft typing approaches also assume a uniform runtime representation for all types. The goal of our type system, on the other hand, is to optimize Lua programs, primarily by using optimized representations of Lua values. It is ill suited for finding programming errors.

3.3.2 Flow Analysis

Iterative flow analysis has been used by optimizing compilers for Scheme and Lisp to extract type information from dynamically typed programs. Beer [1987] presents an inference system that uses local data flow analysis to infer types in Common Lisp programs, aided by type declarations for formal parameters. The inference system has, for each node in the flow graph, a function that computes the type of the node's output from the type of the node's input. Initially all types (except those flowing from formal parameters and constants) are empty, and analysis of the flow graph is iterated until reaching a fixed point. Optional type declarations act as filters for the types of corresponding control flow nodes. An implementation of type inference using these techniques is present in the "Python" CMU Common Lisp compiler [MacLachlan, 1992]. The information extracted by the analysis is used in code optimization.

Lua, in contrast to Common Lisp, does not have type declarations, so a local data flow analysis like the above would need to assign the maximal type (an analogue to $\mathcal{D}$) to formal parameters. The result is that in most cases all local variables and expressions will also have this maximal type, rendering inference useless. Our type inference is also syntax-directed instead of depending on computing a data flow graph, which is a harder problem in a higher-order language with a single namespace (*Lisp-1*), as Lua and Scheme, than in languages with separate scalar and function namespaces (*Lisp-2*), as Common Lisp [Gabriel and Pitman, 1988, Shivers, 1988].

Storage Use Analysis [Serrano and Feeley, 1996] uses a global data flow analysis to infer types in a Scheme dialect by computing a subset of the (finite) set of possible abstract values (one abstract value for each scalar type, closure, and structured data constructor in the program). The analysis has a special abstract value $\top$ for values external to the program. The analysis is done by iterated traversal of the call graph of an intermediate form of the

program where all closure creation has been made explicit and higher-order function application has been converted to closure calls. The call graph is not constructed explicitly, but traversed implicitly from the syntactical structure of the transformed program. The authors use the information extracted by their analysis for optimizing data representation.

Our type inference algorithm is similar to the Storage Use Analysis algorithm in that their "type system" is also monomorphic, but the actual details of the inference algorithm are very different: we do not require a transformation to make closure creation and use explicit, and traverse the syntax tree instead of the call graph. Their type system is also implicitly specified by the algorithm, and their treatment of structured types is ad-hoc, while our type system is specified separately as a deduction system. We believe this makes our type system and inference easier to reason about, both for the compiler writer and for the programmer. Although our type inference's purpose is program optimization, it is trivial to make it output readable types for the programmer, using the type language in Section 3.1.1.

Polymorphic Splitting [Wright and Jagannathan, 1998] is a global flow analysis for Scheme that mimics ML's let-polymorphism [Damas and Milner, 1982] by splitting different occurrences of the same let-bound closure in different abstract values instead of having they all be the same abstract value. The analysis explicitly constructs a flow graph from the program's syntax tree, and propagates abstract values along this graph. The authors use the results of this analysis to eliminate runtime checks in Scheme programs, but, due to polymorphism, assume a uniform data representation, so the results of the analysis are unsuited for the optimizations that our type inference enables.