Fabio Mascarenhas de Queiroz

Optimized Compilation of a Dynamic Language to a Managed Runtime Environment

PhD Dissertation

DEPARTMENT OF COMPUTER SCIENCE

Graduate Program in Computer Science

Rio de Janeiro
September 2009

Pontifícia Universidade Católica
do Rio de Janeiro

Fabio Mascarenhas de Queiroz

Optimized Compilation of a Dynamic Language to a Managed Runtime Environment

PhD Dissertation

Dissertation presented to the Graduate Program in Computer Science of Departamento de Informática, PUC–Rio as partial fulfillment of the requirements for the degree of Doctor in Philosophy in Computer Science

Advisor: Prof. Roberto Ierusalimschy

Rio de Janeiro
September 2009

Pontifícia Universidade Católica do Rio de Janeiro

Fabio Mascarenhas de Queiroz

Optimized Compilation of a Dynamic Language to a Managed Runtime Environment

Dissertation presented to the graduate program in Computer Science of Departamento de Informática, PUC–Rio as partial fulfillment of the requirements for the degree of Doctor in Philosophy in Computer Science. Approved by the following commission:

Rio de Janeiro — September 4, 2009

Fabio Mascarenhas de Queiroz

Fabio Mascarenhas de Queiroz graduated from the Universidade Federal da Bahia (Salvador, Bahia) in Computer Science. He then obtained a Master degree at PUC–Rio in programming languages, and has now finished his Ph. D. at PUC–Rio, also in programming languages.

Bibliographic data

Queiroz, Fabio Mascarenhas de

Optimized Compilation of a Dynamic Language to a Managed Runtime Environment / Fabio Mascarenhas de Queiroz; orientador: Roberto Ierusalimschy. — Rio de Janeiro : PUC–Rio, Departamento de Informática, 2009.

v., 97 f: il. ; 29,7 cm

1. Tese de Doutorado - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia.

1. Informática – Dissertação. 2. Linguagens de Programação. 3. Compiladores. 4. Inferência de Tipos. 5. Ambientes de Execução Gerenciada. 6. Desempenho. 7. Linguagens Dinâmicas. 8. Lua. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

I would like to thank my advisor, Professor Roberto Ierusalimschy, for his advice and insights in crucial moments of this work, and for the many intellectually stimulating, and entertaining, conversations that we had through these almost seven years that I have been his student, both Master's and Ph. D.'s.

I also would like to thank my friends and colleagues at Lablua, Sérgio Medeiros and Hisham Muhammad, for providing a productive and fun work environment.

I also thank my friends, my family, and specially Cristina, for the constant encouragement, prodding, and for enduring me in when I was my most stressed and least social.

Finally, I would like ot thank CNPq and FAPERJ for their financial support in respectively the first and second half of my doctorate, without which this work could not have been done.

Abstract

Queiroz, Fabio Mascarenhas de; Ierusalimschy, Roberto. **Optimized Compilation of a Dynamic Language to a Managed Runtime Environment**. Rio de Janeiro, 2009. 97p. PhD Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Managed runtime environments have become popular targets for compilers of high-level programming languages. They provide a high-level type system with enforced runtime safety, as well as facilities such as garbage collection, possibly sandboxed access to services of the underlying platform, multithreading, and a rich library of data structures and algorithms. But managed runtime environments lack a clear performance model, which hinders attempts at optimizing the compilation of any language that does not have a direct mapping to the runtime environments' semantics. This is aggravated if the language is dynamically typed.
We assert that it is possible to build a compiler for a dynamic language that targets a managed runtime environment so that it rivals a compiler that targets machine code directly in efficiency of the code it generates. This dissertation presents such a compiler, describing the optimizations that were needed to build it, and benchmarks that validate these optimizations. Our optimizations do not depend on runtime code generation, only on information that is statically available from the source program. We use a novel type inference analysis to increase the amount of information available.

Keywords

Programming Languages. Compilers. Type Inference. Managed Runtime Environments. Benchmarking. Dynamic Languages. Common Language Runtime. Lua.

Resumo

Queiroz, Fabio Mascarenhas de; Ierusalimschy, Roberto. **Compilação Otimizada de uma Linguagem Dinâmica para um Ambiente de Execução Gerenciada**. Rio de Janeiro, 2009. 97p. Tese de Doutorado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Ambientes de Execução Gerenciada tornaram-se alvos populares para compiladores de linguagens de programação de alto nível. Eles provêem um sistema tipos de alto nível com segurança de memória garantida, assim como facilidades como coleta de lixo, acesso a serviços da plataforma subjacente (possivelmente através de uma sandbox), multithreading, e uma rica biblioteca de estruturas de dados e algorithmos, mas não possuem um modelo de desempenho claro, o que atrapalha tentativas de otimização de qualquer linguagem que não tenha um mapeamento direto na semântica do ambiente de execução, especialmente se a linguagem é dinamicamente tipada.
Nós afirmamos que é possível construir um compilador para uma linguagem dinâmica que tem como alvo um ambiente de execução gerenciada que rivaliza um compilador que tem como alvo linguagem de máquina na eficiência do código que ele gera. Essa tese apresenta um compilador com tal característica, descrevendo as otimizações necessárias para sua construção, e testes de desempenho que validam essas otimizações. Nossas otimizações não dependem de geração de código em tempo de execução, apenas em informação estaticamente disponível no código fonte. Nós usamos uma nova análise de inferência de tipos para aumentar a quantidade de informação disponível.

Palavras–chave

Linguagens de Programação. Compiladores. Inferência de Tipos. Ambientes de Execução Gerenciada. Desempenho. Linguagens Dinâmicas. Lua.

Contents

List of Figures

List of Tables

1 Introduction

Managed runtime environments have become popular targets for compilers of high-level programming languages. Reasons for adoption of these runtimes include a safe execution environment for foreign code, easier interoperability, and their existing libraries. These managed runtimes provide a high-level type system with enforced runtime safety, as well as facilities such as garbage collection, possibly sandboxed access to services of the underlying platform, multithreading, and a rich library of data structures and algorithms. Examples of managed runtimes include Microsoft's Common Language Runtime [Microsoft, 2005], the Java Virtual Machine [Lindholm and Yellin, 1999], and more recently the JavaScript runtimes present in web browsers [ECMA, 1999, Manolescu et al., 2008].

As these runtimes are higher level than the usual compiler targets such as machine languages and intermediate languages close to the hardware, they inevitably lead to an impedance mismatch between the semantics of the language that is being compiled and the semantics of the target runtime, which translates to inefficiency in the generated code, changes in the language, or both. The lack of a clear performance model for these runtimes, which can have great variation even among different implementations of a particular runtime, also hinder attempts at optimizing the generated code of any language that does not have a direct mapping to the semantics of the runtime. Writing an optimizing compiler for these runtimes has to involve guesswork and experimentation.

The problem of efficient compilation is worse when compiling dynamically-typed languages to statically typed runtimes, such as the CLR and JVM. In this case, all of the operations of the language have to be compiled using runtime type checks or using the virtual dispatch mechanism of the runtime. If the language is object-oriented then it cannot use the native method dispatch mechanism of the runtime, and has to implement its own. Implementing arithmetic operations is particularly troublesome, as the runtimes usually do not have tagged unions value types, so numbers have to be boxed inside heap-allocated objects and treated as references.

Compiling to a dynamically-typed runtime is also problematic unless the semantics of the types and operations of the source language exactly match the semantics of corresponding types and operations of the target. This kind of semantic match is very rare among high-level languages. In practice, some form of wrapping and runtime checking, or even more radical program transformations, such as trampolines for tail call optimization, are still necessary.

Compiling dynamically-typed languages to machine language or low-level languages also needs runtime type checks and dynamic dispatch, but the low-level nature of the target language means these operations are more efficient than their equivalent on managed runtimes and their performance characteristics are better understood. When compiling to a machine language you have a performance model, the performance model of the target processor, that is missing in the intermediate languages of the managed runtimes.

Nevertheless, we assert that it is possible to generate efficient code from a dynamically-typed source language to a managed runtime. By efficient we mean at least as fast as the same code executed by a good native interpreter for the source language and, in a modern managed runtime with a good optimizing JIT compiler, matching or exceeding the performance of code generated by a good optimizing compiler for the source language. We support our assertion by implementing an optimizing compiler for the Lua programming language that targets the Microsoft Common Language Runtime and benchmarking this compiler against the Lua interpreter and an optimizing Lua JIT compiler.

Lua is a dynamically-typed language that has relatively simple semantics and a very efficient[1] interpreter implemented in C [Ierusalimschy, 2006]. Lua has some advanced features such as extensible semantics, anonymous functions with full lexical scoping (similar to the lexical scoping present in the Scheme language), tail call optimization, and full asymmetric coroutines [Ierusalimschy et al., 2007, de Moura et al., 2004]. It has a simple type system: nil, floating point numbers, immutable strings, booleans, functions, and tables (associative arrays), the latter having syntactical and semantic support for use as numeric arrays, as user-defined data types, and as a way to implement object orientation using prototypes or classes. Section 1.1 is a brief primer on the language.

Our approach for creating a compiler for the Lua programming language was to start with a very simple compiler, and a suite of small to medium length benchmarks that include Lua idioms used by actual programs. We then built variations of this first compiler, validating optimizations against this benchmark suite. This is a continuation of previous work we published

[1]Compared to other interpreters for dynamically-typed languages.

in Mascarenhas and Ierusalimschy [2008].

Benchmarking was an essential part of our approach; the CLR intermediate code that our compilers generate passes through the CLR optimizers on conversion to native code, and we could not know beforehand how a particular piece of CLR code would perform. This makes it very difficult to accurately assess the impact of even simple changes in the compiler. In the worst case, what we can think is an optimization may in fact make real programs run slower. Benchmarks helped assess the impact of our changes, and anomalies found in the results of some of the benchmarks are evidence of the unpredictability introduced by the lack of a performance model. A benchmark suite is also useful for programmers, being a source of tips on how to write efficient code for the compiler.

Our simplest compilers did not use static program analysis, so the scope of optimizations they implemented was limited to what was possible with extremely local information. The compilers had different mappings of Lua types to CLR types, such as using value types versus using boxing and reference types and interning strings. The compilers also had different ways to implement the return of multiple values from a function call.

Our more advanced compilers used type inference to extract more static information out of Lua programs. Having more information let us generate better code. For example, if we can statically determine the runtime types of each variable and parameter then we can avoid boxing and type checking for every variable that we are certain can only hold numbers. Sufficiently precise type information let the compiler synthesize CLR classes for Lua tables, transforming what was a hash lookup in the simpler compilers to a simple address lookup.

We did both local and interprocedural type inference. The local inference does not cross function boundaries and is much simpler to implement, but the information it obtains is very imprecise. Languages that use local type inference use explicit type annotation of function arguments to get more precise results. Interprocedural type inference was harder to specify and implement, but yielded much better results in our benchmarks. The basic problem of an interprocedural type inference of Lua programs is the same as of other languages with first-class functions: which functions are callable at each call site is initially unknown, and has to be found during inference. To know which function is callable at a call site you need the type of the variable referencing the function, or the expression that yields it.

Type inference is a complex algorithm that can subtly alter the behavior of a program if specified and implemented incorrectly. We did a formal specification of our typing rules and inference algorithm with regards to an operational semantics of a subset of Lua to make the specification of our type inference more precise and easier to understand. The actual inference algorithm works on the full Lua language, but leaves parts of a program outside of this subset dynamically typed.

Related work on implementing compilers for dynamic typed languages that target managed runtimes uses approaches that are based on runtime optimization using the code generation and dynamic code loading facilities of the managed runtimes. These approaches adapt the concept of polymorphic inline caches [Deutsch and Schiffman, 1984, Hölzle and Ungar, 1994], while our approach uses compile-time optimization via static analysis. Examples of the former include Microsoft's Dynamic Language Runtime for the CLR [Chiles and Turner, 2009] and the invokedynamic opcode for the JVM [Sun Microsystems, 2008]. The implementation of approaches such as the DLR and invokedynamic is very complex, and their performance characteristics are even more opaque than the performance characteristics of the underlying runtime, due to the extra level of indirection in the compilation.

The following sections are a small primer on the Lua language and the CLR. The rest of the dissertation is organized as follows: Chapter 2 presents our basic Lua compiler and the variations that do not depend on interprocedural type inference. Chapter 3 is a presentation of our type inference algorithm for the Lua language, and how we used it in our Lua compiler; Chapters 2 and 3 also discuss related work. Chapter 4 presents our benchmark suite, our benchmark results and the analysis of these results. Finally, Chapter 5 states our conclusions and outlines possible future work.

1.1 A Lua Primer

Lua [Ierusalimschy, 2006] is a scripting language designed to be easily embedded in applications, and used as a configuration and extension language. Lua has a simple syntax, and combines common characteristics of imperative languages, such as loops, assignment, and mutable data structures, with features such as associative arrays, first-class functions, lexical scoping, and coroutines.

Lua is dynamically typed, and has eight types: `nil`, *number*, *string*, *boolean*, *table*, *function*, *userdata*, and *thread*. The *nil* type has a single value, `nil`, and represents an uninitialized or invalid reference; values of type *string* are immutable character strings; the *boolean* type has two values, `true` and `false`, but every value except `false` and `nil` has a true boolean value for the purposes of tests in the language; values of type *table* are associative arrays; the *userdata* type is the type of external values from the application that is embedding Lua, and *thread* is the type of coroutines.

Table indexes can be any value except `nil`. Lua also has syntactic sugar for using tables as records and objects. The expression `tab.field` is syntactic sugar for `tab["field"]`. Lua functions are first-class values; storing functions in tables is the base of Lua's support for object-oriented programming. The expression `tab:method(x)` is a method call and is syntactical sugar for `tab.method(tab, x)`. This syntactic sugar also works for defining methods, as in the fragment below:

```
function tab:method(x)
  -- method body
end
```

The fragment above is the same as the following one:

```
tab.method = function (self, x)
  -- method body
end
```

The behavior of Lua values can be extended using *metatables*. A metatable is a table with functions that modify the behavior of the table or the type it is attached to. Each table can have an attached metatable, but for the other types there can only be one metatable per type. A common use of metatables is to implement single inheritance for objects, as in the following fragment:

```
local obj = setmetatable({ a = 0 },
  { __index = parent })
function parent:method(x)
  self.a = self.a + x
end
obj:method(2)
print(obj.a) -- 2
```

In the fragment above, whenever the code tries to read a field from `obj` and this field does not exist, Lua looks it up in the `__index` field of the metatable, so the value of this field works as the parent object. Other metatable fields modify operations such as assignment to a field, arithmetic operations, and comparisons.

First-class functions, lexical scoping, and imperative assignment interact in the same way as they do in Scheme [Steele, 1978, Adams et al., 1998].

The following fragment creates a counter and returns two functions, one to increment and one to decrement the counter. Both functions share the same variable in the enclosing lexical scope, and each pair of functions returned by make_counter shares a `counter` variable distinct from the one of the other pairs:

```
function make_counter()
  local counter = 0
  return function ()
           counter = counter + 1
           return counter
         end,
         function ()
           counter = counter - 1
           return counter
         end
end
```

The fragment above also shows how Lua functions can return more than one value. In most function calls Lua just takes the first returned value and discards the others, but if the function call is part of a multiple assignment, as in `inc, dec = make_counter()`, or if you are composing functions, as in `use_counter(make_counter())`, then Lua will use the extra values.

A function call that has more or less arguments the the arity of the function is legal in Lua. Extra arguments are simply ignored, and missing arguments have the value `nil`.

1.2 The Common Language Runtime

The CLR [Microsoft, 2005] is a managed runtime created to be a common target platform for different programming languages, with the goal to make it easier for those languages to interoperate. It has an intermediate language and shared execution environment with resources such as a garbage-collected heap, threading, a library of common data structures, and a security system with code signing. The CLR also has an object-oriented type system augmented with parametric types with support for reflection and tagging of types with metadata [Yu et al., 2004].

CLR types can be *value types* or *reference types*. Value types are the primitive types (numbers and booleans) and structures; assignment of value types copies the value. Reference types are classes, interfaces, delegates, and arrays. Assignment of reference types copies the reference to the value, and the value is kept in the heap. The CLR has a single-inheritance object system, rooted in the `object` type, but classes can implement *interfaces* which are types that only have abstract methods. *Delegates* are typed function pointers. Each value type has a corresponding reference type, used for *boxing* the value type in the heap.

The CLR execution engine is a stack-based intermediate language with about 200 opcodes, called Common Intermediate Language, abbreviated as CIL or just IL. The basic unit of execution is the method; each method has an activation record kept in an execution stack. The activation record has the local variables and arguments of the method being executed, metadata about the method, and the evaluation stack of the execution engine. IL opcodes implement operations such as transferring values between local variables, arguments, or fields and the evaluation stack, creation of new objects, method calls, arithmetic, and branching.

2
"Naive" Compilation

This chapter presents our basic approach for compiling Lua programs to the CLR, along with a series of variations of the basic approach that try to improve performance of generated code without having to perform non-trivial static analysis on the source code (type inference or data-flow based optimizations). We focus on different ways of mapping Lua's values and operations to the Common Language Runtime. Evaluation on the effectiveness of our choices is deferred to Chapter 4, where we present our benchmarks. Although benchmarking was intertwined with the process of coming up with the compilers in this chapter, and has informed this process, we believe that presenting the benchmarks and discussion about them separately leads to a better exposition.

The lack of static analysis and data-flow optimizations results in relatively naive code generation. But the code we generate still has to interact with the CLR's optimizer and JIT compiler, and this interaction is hard to predict even for relatively naive code. The CLR specification [Microsoft, 2005] has no performance model for its intermediate language, and even the performance of a particular implementation depends on the architecture of the machine where it is executing [Blajian et al., 2006, Morrison, 2008a]. The Java Virtual Machine, another managed runtime environment, has the same lack of a clear performance model [Gu et al., 2006, Georges et al., 2007]. We believe that the preferred approach with this lack of performance models is to generate straightforward code and then experiment to discover the impact of any changes, instead of trying to generate optimized code from what we think will be the behavior of the runtime.

Section 2.1 presents the basic compiler and how it is implemented. Section 2.2 presents several variations on this basic compiler, detailing their changes an the rationale for them. Section 2.3 reviews related work on compilation of Lua programs and on compilation of other dynamic languages to managed runtime environments.

Each compiler has a short name that we use to refer to it throughout this chapter and the rest of the dissertation. Table 2.1 lists the names for each compiler along with its defining characteristic and the section where the compiler is described. References to these names in the text are quoted.

Name	Description
base	Basic Compiler, Section 2.1
single	Single Return Values, Section 2.2.1
box	Boxed Numbers, Section 2.2.2
intern	Interned Strings, Section 2.2.3
prop	Local Type Propagation, Section 2.2.4
infer	Type Inference, Section 3.2

Table 2.1: Compiler names

2.1 Basic Compiler

The basic LuaCLR compiler directly generates CLR Intermediate Language code from an annotated (and desugared) Lua abstract syntax tree. There is no separate intermediate language between Lua and IL, so describing the compiler is a matter of describing the representation of Lua's types by CLR types and of describing the IL code for Lua's operations. We present the representation of Lua values in the next section, and Section 2.1.2 covers how the operations are implemented.

2.1.1 Representing Lua Values

Lua is a dynamically typed language, so values of any of its types can be operands to its operations, and Lua will raise a runtime error if the operands are incompatible. This means that Lua types need an uniform representation and this representation must carry type information that is checked at runtime. Having to use an uniform representation automatically precludes directly using the CLR's primitive types (several types of numbers, and booleans), but still leaves several approaches for representing Lua types.

The approach we use in the basic compiler is essentially the same as the representation we used in Lua2IL [Mascarenhas and Ierusalimschy, 2005], and is similar to the representation the Lua interpreter uses [Ierusalimschy et al., 2005]. The Lua interpreter uses a C *struct* with a type tag and a C *union* with the actual value. The value itself may be a double precision floating point number, a boolean, an external pointer, or a GC-managed pointer. Other types of structs and unions represent values of the GC-managed types, such as functions and tables.

The CLR does not have unions, just structures and classes. We use a `Lua.Value` structure as our uniform representation for all Lua values. A field of this structure is a reference to an instance of a subclass of the abstract `Lua.Reference` class. The other field of this structure is a double precision floating point number (CLR type `double`). The first field acts as a tag that identifies if the value is a number or not; if the first field is `null` then the value is a number, with the number stored in the second field.

Each of the other Lua types is represented by a subclass of `Lua.Reference`. The *nil* type is the singleton class `Lua.Nil`, so the sole instance of this class is the representation of the value `nil`. The `Lua.Bool` class represents booleans, with a single instance for `true` and a single instance for `false`. Strings are represented by instances of the `Lua.String` class, which encapsulates CLR strings, as both Lua and CLR strings are immutable. An important difference between our representation and the Lua interpreter representation is that it is possible to have several instances of the same string with our representation, while the Lua interpreter *interns* strings so there is only one instance of each string. Our representation makes string creation more efficient, at the cost of slower equality testing (and consequently slower table lookups).

We represent tables with instances of the `Lua.Table` class. The internal implementation of this class is similar to the representation used by the Lua interpreter, which has separate hash table and array parts to optimize some accesses using integer indexes. A description of the algorithm is in Ierusalimschy et al. [2005], and implementing it for the CLR is straightforward.

Each function in the program source is represented by its own subclass of `Lua.Reference`. Instances of the function's class are the closures created during execution of the program. Lua functions are first class values that can reference and modify variables in their enclosing lexical scopes, so the closures we create have instance variables that hold references to variables in enclosing scopes that the function uses. In the next section we will cover the internals of closures in greater depth. Each closure also has an instance variable to hold the *environment table*, used to look up the value of global variables.

An important characteristic of our chosen representations is that Lua code never manipulates "native" CLR objects directly, only instances of the `Lua.Value` type. We will see in the next section that this leads to simpler code generation and more efficient generated code. The drawback is that it makes it harder to interface Lua with other CLR code. To interface with other code we can wrap CLR values in a subclass of `Lua.Reference` that delegates operations to the actual CLR type of the value (using reflection, for example).

Any Lua values passed to CLR functions can also be converted to equivalent primitive CLR values. This wrapping can be made transparent to the user, and we have already used this wrapping both in Lua2IL and in a bridge from the Lua interpreter to the CLR [Mascarenhas and Ierusalimschy, 2005, 2004].

2.1.2 Code Generation

The code the "base" compiler generates for most operations is similar: there is code to test if both operands are numbers, then code that does the operation inline if both operands are numbers, and finally a call to a function in the runtime library that does the operation if one of the operands is not a number. These functions of the runtime library are all virtual methods of the base `Lua.Reference` class. So each operation involves a type check and then either the inlined operation or a virtual dispatch to a method that implements the operation.

We inline arithmetic and relational operations on numbers because they are just a single IL instruction. Instead of generating tests and inlined operations we could just compile each operation to a call to a runtime library function that would do the checks, and rely on the CLR JIT's inliner to inline this code. But the type checks and inlined operations are straightforward to generate.

Lua functions are first-class values with lexical scoping, as we saw in Section 1.1. Distinct functions that use the same variable share the same memory location for the variable, instead of each having a distinct memory location that receives a copy of the variable's value at the moment the function is defined, like lexically scoped variables in Python [Mertz, 2001] and the "final" restriction of Java's inner classes [Gosling et al., 2005].

The "base" compiler uses the CLR stack for local variables and argument passing, but the CLR denies access to local variables outside of the current stack frame, and also does not allow references to local variable locations that escape the function where the local variable is defined. To get around these limitations we have to allocate in the heap all local variables that are used outside the function that defines them, and keep just references to these heap cells in the stack. Each closure also has a kind of *display* [Aho et al., 1986, Friedman et al., 2001] that holds references to heap cells of all variables in enclosing scopes that it uses. The basic compiler implements the display as fields in the function's type, so they are instance variables of the closures. When instantiating the function at runtime (at the point in the program where the function is defined) the closure's constructor takes one argument for each field in its display, and sets the fields in the newly created instance.

For example, take this fragment:

```
local w
function f(x)
  return function (y)
    return function (z) return w + x + y + z end
  end
end
```

The class of the innermost function has fields for variables w, x, and y. The class of its enclosing function has fields for variables w and x, and the class of the outermost function has a field for variable w. All these fields hold a cell for a single Lua value. When instantiating the innermost closure, we emit code to call its constructor passing the value of the w and x fields of the enclosing closure, as well as the value of the y stack slot, which is also a reference to a heap cell holding the value of y. Using references to these heap cells allows two closures to share the same variable. The type of these heap cells is `Lua.UpValue`.

Lua does not have arity errors when calling functions; extra arguments are simply ignored, and missing arguments have value *nil*. Function calls in the basic LuaCLR compiler are compiled to a call to a virtual method of `Lua.Reference` called `Invoke`. `Invoke` is overloaded; `Lua.Reference` has different versions of `Invoke` for different numbers of arguments in the call site, from zero arguments to a maximum fixed by the compiler, plus a version of `Invoke` that takes an array of arguments. The version of `Invoke` that takes an array is used in call sites with more arguments than the maximum or with call sites where the number of arguments is unknown at compile-time. The latter case may happen because Lua functions can return a variable number of values, and when a function call is the last argument to another function call all the values returned by the first call get passed to other call (like when composing functions).

Each function always has a version of `Invoke` that matches the arity of the function, and this version has the code generated from the function body. Other versions of `Invoke` delegate to it, adjusting the number of arguments as necessary. This means that calling a function with a small number of arguments translates to a single CLR virtual call if the number of arguments matches the arity of function. If there is a mismatch there is an adjustment, but then the adjusted call to `Invoke` is a statically dispatched call, not another virtual call.

The classes of all three functions in the example shown in the previous page have `Invoke` methods similar to the ones shown in the fragment of C# code below[1]:

```
Lua.Value[] Invoke() {
  return this.Invoke(Lua.Nil.Instance);
}
Lua.Value[] Invoke(Lua.Value v1) {
  ... code that implements the function ...
}
Lua.Value[] Invoke(Lua.Value v1, Lua.value v2) {
  return this.Invoke(v1);
}
... versions of Invoke with more parameters ...
Lua.Value[] Invoke(Lua.Value[] vs) {
  if(vs.Length > 0)
    return this.Invoke(vs[0]);
  else
    return this.Invoke(Lua.Nil.Instance);
}
```

2.2 Variations of the Basic Compiler

There are several changes that we can make in how we represent Lua types and generate code for operations on these types that trade complexity in the implementation of the compiler for faster programs. In the following sections we present variations of the "base" compiler and the rationale for these variations.

2.2.1 Single Return Values

Lua functions can return any number of values, so all functions in the "base" compiler return an array of `Lua.Value`, even if they only return a single value or the function call needs just the first one. This means that every function call has to allocate an array on the heap to store the return values of the call.

There are several situations where the compiler can be sure that it only needs the first return value, or none at all. Some of these situations are when

[1]We use C# code just to make the fragment shorter, the compiler actually generates IL directly.

a function call is on the right side of an assignment, as in `x = f()`, when a function call is used as a statement, when a function call is not the last argument of another function call, as in `g(f(), x)`, and when a function call is used in a binary operation.

Our first variation of the basic compiler exploits this property by compiling each function twice; one version returns an array, just like in "base", while the other returns a single `Lua.Value` (possibly *nil*). So instead of a `Invoke` method the functions have a `InvokeM` method, for multiple values, and a `InvokeS` method, for single values. They are overloaded to take different number of arguments, as in the basic compiler. In the code for *f*'s `InvokeS`, the code generated for all `return` statements returns the first expression in the statement and just evaluates and discards the others, while in the code for *f*'s `InvokeM` a `return` statement allocates an array, stores the values of its expressions in this array, and then returns it.

For example, in the function call `g(f(), f())` the compiler emits a call to a `InvokeS` method in the first call to *f*, and a call to a `InvokeM` method in the second call to *f*, as all values returned by the second call have to be passed as arguments to the call to *g*.

2.2.2 Boxed Numbers

The previous compiler still uses a `Lua.Value` structure as a uniform representation for Lua values. This structure has two fields: one is used if the value is a number, and the other is a pointer to other types of values, which are all instances of subclasses of the `Lua.Reference` abstract class. This arrangement tries to mimic the representation used by the Lua interpreter (which uses a union instead of a structure), and avoids having to store numbers in the heap.

We will see in Section 2.3 that other compilers for dynamic languages on the CLR choose to represent all values as pointers to objects in the heap, with most compilers using the CLR base `object` type as the common denominator for the values of the language. We follow a similar approach in the variation we present in this section.

Instead of having a `Lua.Value` structure we will use `object` as the type of Lua values; all values are now pointers to either a boxed `double` in the heap, in case the value is a number, or a pointer to an instance of one of the subclasses of `Lua.Reference`, for all other types of values. The representation of these other types remain unchanged, except for trivial changes in the signatures of methods and the internal representation of tables to deal with `object` instead of `Lua.Value`.

All operations involving numbers in the "base" and "single" compilers first check if the value is a number by checking if the `Lua.Reference` field of the `Lua.Value` structure is `null`, and then unpack the number from the structure, by loading the `double` field. In the "box" compiler these operations become a type test, to see if the value is of type `double`, and an unboxing operation if it is. These are the following fragment in the CLR intermediate language:

```
isinst double
brfalse notnum
unbox double
```

After doing the operation, the "base" and "single" compilers had to store `null` in the `Lua.Reference` field and the number that is the result of the primitive operation in the `double` field of the `Lua.Reference` structure that holds the result of the operation. The "box" compiler just does a **box** `double` IL operation that takes the result and boxes it.

In operations that do not involve numbers, the "base" and "single" compilers had to unpack the operand by loading the `Lua.Reference` field from the `Lua.Value` structure and then invoking the correct virtual method. In the "box" compiler this becomes a cast to `Lua.Reference` followed by the virtual dispatch.

This variation presents a case where the high-level nature of the intermediate language of a managed runtime environment and a lack of information on how the optimizer of this runtime translates this intermediate language to machine code makes it hard to assess if a particular change improves or not the execution time of compiled programs.

The number of intermediate language instructions to do each operation in the "base", "single", and "box" compilers is roughly the same, but the cost of these instructions is unpredictable. We have to use benchmarks to evaluate how each approach performs. The change of unboxed to boxed representations can have wildly different performance characteristics depending on how the runtime's heap allocator and garbage collector work, and even whether the runtime optimizes boxing and unboxing of some numbers. For example, one possible optimization a runtime can do is to pre-allocate a range of boxed numbers, like small integers, and keep reusing them instead of allocating new objects in the heap.

2.2.3 Interned Strings

The Lua interpreter *interns* all strings so each string has only one copy. This lets the test of whether two strings are equal be a simple test of pointer equality, instead of a test of the strings' contents. Tests of string equality are always used when indexing tables with string keys, so interning strings makes indexing using string keys faster. Indexing operations with string keys are a common operation in Lua, specially in OO-style code, as field accesses (`obj.field`) and method calls (`obj:method()`) get desugared to indexing operations with string keys.

Our previous compilers represent Lua strings with the `Lua.String` subclass of `Lua.Reference`, and the internal representation is just a CLR `string`. Like Lua strings, CLR strings are immutable, but the CLR specification does not dictate how strings should be implemented, so implementations are free to intern strings or not. Not interning is a better choice when string creation (as a result of concatenation, for example, or slicing) is more common than testing equality. In this section we present a variation of the "box" compiler that tries to optimize equality tests, and benchmarking this variation against "box" tells whether a particular CLR implementation is interning its strings or not.

One optimization we can do is to implement the equality test between two `Lua.String` objects as a pointer equality test first, with the CLR string equality test done only when the pointer test fails. We then make sure all string literals in the program, including desugared field and method names, have only one instance. This completely avoids regular string comparison in indexing operations if the particular `Lua.String` key is already in the table and there are no hash collisions.

The optimization we actually do in the "intern" compiler is applicable to a greater number of indexing operations; we add a new type for interned strings that we call *symbols*, the type `Lua.Symbol`. This is a subtype of `Lua.String`, the only difference being that symbols are interned in a global hashtable, so two symbols can safely be tested for equality by pointer equality. All string literals are symbols, and tables intern all strings that are used as keys, so all internal equality tests in a hash lookup are done between symbols. Any indexing operation that has a symbol key only needs pointer equality, even in the case of collisions or if the key is not in the hash. This includes all indexing operations caused by Lua's OO syntactical sugar.

Operations that create new strings still create instances of `Lua.String` instead of `Lua.Symbol`, so they do not have to pay the cost of interning strings.

If the CLR implementation does not intern strings then this variation should speed up OO-style code without slowing down code that does string processing.

2.2.4 Local Type Propagation

Working with boxed numbers can have a big performance impact on code that does a lot of arithmetic, even if the runtime tries to optimize boxing and unboxing operations and has a good memory allocator and garbage collector. The `Lua.Value` representation of the "base" and "single" compilers avoids boxing and unboxing, but wastes memory and uses a runtime feature, structured value types allocated in the stack, that also relies on optimizations by the runtime. If we statically know that we are only dealing with numbers we can avoid both boxing and structures and use the native `double` type of the CLR directly.

In Chapter 3 we present a way of extracting this information with *type inference*, and show how it can have other uses besides just avoiding boxing of numbers. In this section we present a much simplified, and local, version of type inference. It is local because it does not try to infer types across function call boundaries. It types variables and expressions in a function as one of seven simple types: nil, boolean, number, string, function, table, and the type *any*, which means that the variable (or expression) can have any type. There are no structural types: having a type "function" or "table" just means that operations can be dispatched to methods of `Lua.Function` or `Lua.Table` without doing type checks; all functions have type "function" and all tables have type "table". Function calls and indexing operations always have type "any", and "any" is also the type of the parameters of a function.

The typing rules are straightforward. For assignment there are three cases, which depends on the types of the lvalue (the left side of the assignment) and the rvalue (the right side of the assignment):

1. If the lvalue does not have a type yet, then it gets the type of the rvalue;
2. if the lvalue has the same type as the rvalue, then nothing changes;
3. if the lvalue and rvalue have different types, then the lvalue now has type "any".

The rules for binary arithmetic operations are that the type of the operation is "number" if the type of both operands is "number", otherwise it is "any". In Lua, as a consequence of the metatables we mentioned in Section 1.1, binary arithmetic operations are dispatched to user-defined operations if one of the operands is not a number, and our type inference has to assume that these operations can return anything.

The representation that the "prop" compiler uses for most types is the same as the one the "intern" compiler uses. The only type that has a different representation is "number", which uses the `double` type of the CLR. The type "any" is an `object` pointer to either a boxed `double` or an instance of a subclass of `Lua.Reference`, and other types continue to be represented as subclasses of `Lua.Reference`.

We implement the type inference as an iterated traversal of each the abstract syntax tree of each function. The inference stops when the types converge. All the typing rules have the invariant that if the type of a term is "any" it cannot change, and if the type of a term is not "any" then it can either stay the same or change to "any", so convergence is guaranteed.

Changes to the code generator are straightforward, with simpler code for arithmetic expressions when the operands are numbers, and elimination of type checks prior to calls to methods of `Lua.Reference` whenever possible.

2.3 Related Work

In this section we review previous work on compiling dynamic languages to managed runtime environments. We focus on compilers targeting the Common Language Runtime first, as it was the first managed runtime specifically created as a shared runtime for different high-level languages, with emphasis on interoperation among these languages [Hamilton, 2003].

In 1999 and 2000 Microsoft sponsored the development of a compiler for the Python scripting language, written by Mark Hammond and Greg Stein. The compiler supported most of the Python language and allowed Python programs to interface with CLR code, but the authors judged the performance to be "so low as to render the current implementation useless for anything beyond demonstration purposes" [Hammond, 2000]. This poor performance was for "both the compiler itself, and the code generated by the compiler" [Hammond, 2000]. The authors abandoned the effort in 2002.

Python for .NET used a type mapping similar to the one we used on our basic compiler, with a CLR structure representing Python values, but the authors noted that "simple arithmetic expressions could take hundreds or thousands of Intermediate Language instructions", so the operations were inefficient. All the operations were dispatched to the Python for .NET runtime, as the compiler did not do any inlining.

Common Larceny [Clinger, 2005] was an early Scheme compiler for the

CLR that compiled MacScheme assembly code, used as an intermediate language in the Larceny family of Scheme compilers [Clinger and Hansen, 1994], to CIL instructions. It used instances of a `SchemeObject` class to represent Scheme values, so all values were pointers to objects in the heap; Common Larceny preallocated booleans, characters, and small integers. Common Larceny used its own stack for both values and control information, to make the implementation of Scheme's first-class continuations easier. The performance of the compiled Scheme code was found to be similar to the performance of the code running on the MzScheme interpreter [Clinger, 2005].

Bigloo.NET [Bres et al., 2004] was a later Scheme compiler for the CLR which, like Common Larceny, was part of a family of Scheme compilers that targeted different platforms (the Bigloo family [Serrano and Weis, 1995]). Bigloo.NET was heavily derived from the BiglooJVM compiler [Serpette and Serrano, 2002]. Its usual representation for Scheme values was a pointer (of `object` type) to a value in the heap that could be a boxed number, a byte array (for strings), or instances of classes that represented other Scheme types. Bigloo.NET could also use an interprocedural flow analysis [Serrano and Feeley, 1996] augmented with type annotations to use unboxed representations for scalar values.

In contrast to Common Larceny, Bigloo.NET used the CLR's stack for control-flow, argument passing, and local variable storage, and did not have a complete implementation of Scheme continuations. Bigloo.NET implemented all closures in the same Scheme module as instances of the same class derived from `bigloo.procedure`, with an index that identified the specific closure's entry point plus an array for the closure's display. The code for all the functions in a module was compiled to different entry points in the same CLR method, indexed by a switch statement, because the authors claimed this was better than having each closure be a separate subclass of `bigloo.procedure` for code that makes heavy use of closures. Performance of Bigloo.NET was found to be two to six times slower than the performance of BiglooC, a compiler of the Bigloo family that compiles Scheme to C code, and about twice as slow as BiglooJVM.

Lua2IL was another project for running Lua code inside the CLR, and worked by translating Lua 5.0 bytecodes to CIL instructions, with the help of a support runtime [Mascarenhas and Ierusalimschy, 2005]. Lua2IL used the same mapping from Lua types to CLR types as our "base" compiler, with a structure holding either a Lua number of a reference to other Lua types, and the other types represented by CLR classes with a common subclass. Lua2IL also inlined operations on numbers.

Where Lua2IL differs form our "base" compiler is on the treatment of local variables, function arguments and upvalues. Lua2IL kept a parallel stack for Lua values in the CLR heap and threaded this stack through the compiled Lua code, but still used the CLR stack for control. Code generated by Lua2IL had performance similar to the same code when executed by the Lua 5.0.2 interpreter.

The research prototype of IronPython, another Python compiler for the CLR, showed that Python could have good performance in the CLR if the compiler was designed with careful consideration to performance [Hugunin, 2004]. IronPython boxed numbers and cast other types to `object`, like our "boxed numbers" compiler, and also inlined common operations.

Later versions of IronPython have focused on generalizing its runtime to other dynamic languages, building a *Dynamic Language Runtime* on top of the CLR [Hugunin, 2008, Chiles and Turner, 2009]. The DLR uses a generalization of inline caches [Deutsch and Schiffman, 1984, Hölzle and Ungar, 1994, Hölzle et al., 1991] to implement operations, called *dynamic sites*. Its runtime system generates CIL code that makes heavy use of static method calls, relying on the CLR JIT for inlining and optimization. The implementation of dynamic sites uses a complex runtime system and extensive runtime code generation [Turner and Chiles, 2009]. Recent work on IronPython has moved to mixed-mode execution, with an interpreter for DLR syntax trees as the main mode of execution and the compiled dynamic sites for hotspots, as the extensive code generation needed by the dynamic sites was found to be too heavyweight [Hugunin, 2009].

Currently Microsoft has compilers that target the DLR for Python (the IronPython compilers cited above), Ruby [Lam, 2009], and JavaScript [Dhamija, 2007], and the set of the DLR features is biased towards these languages. In particular, there is no native support for multiple return values from functions, as these three languages implement multiple return values using tuples or arrays. Tail call optimization is also not present in the DLR, despite the CLR supporting for it. The semantics of Lua require tail call optimization, and efficient implementation of Lua function calls requires support for returning multiple values. Another issue is the implementation of lexical scoping and varargs in the DLR, which revert to building stack frames in the heap instead of using the CLR stack.

IronScheme [Pritchard, 2009] is a Scheme R^{6}RS [Sperber et al., 2007] compiler for the CLR in active development that uses the DLR, but it uses a modified version of the DLR that implements the extra functionality needed to compile the Scheme language. The author estimates that the modified DLR used by the IronScheme compiler uses only 20% of code the original DLR, and wants to investigate the viability of writing his own code generator instead of using the DLR [Vastbinder, 2008].

Phalanger [Benda et al., 2006] is a PHP compiler for the CLR. The unit of execution in PHP is a script, and Phalanger compiles a script to a CLR class, with the body of the script as a `Main` method of this class and other functions declared in the script as static methods of the class. Each function is compiled to two methods, one that takes the function's arguments as formal parameters, so uses the CLR stack, and another method that takes arguments in an array and delegates to the other method, for function calls where the target is unknown. Phalanger represents all values as a pointer of type `object`, using a combination of boxing for PHP types that have corresponding CLR primitive types (such as numbers), and subclasses of a `PhpObject` class for other PHP types such as classes and interfaces.

Cuni et al. [2009] use a generalization of polymorphic inline caches called *flexswitches* to implement a JIT compiler for a toy dynamic language targeting the CLR, so there are two layers of JIT compilation, the flexswitch-based JIT compiler and the CLR JIT compiler. While a polymorphic inline cache optimizes just the specific operation at the site of the cache, and does not affect other operations, a flexswitch can call back to the flexswitch-based JIT compiler to generate specialized code for code reachable from the flexswitch. This approach can generate code that is very efficient, but the extra level of compilation and recompilation adds considerable overhead, with most of the total running time of the microbenchmarks in the paper being compilation overhead.

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(e1)
C(e2)
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(e1)
       dup
       isinst double
       brfalse add1
       unbox double
       C(e2)
       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):

```
$C(e)$
**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:

```
      $C(e)$
      dup
      isinst double
      br num
      castclass Lua.Reference
      ldsfld InternedSymbols::c
      callvirt Lua.Reference::get_Item(Lua.Symbol)
      br out
num:  ...
out:  ...
```

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 $\text{“}foo\text{”} \rightsquigarrow \textbf{String}$.

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 $2 \mapsto \textbf{Number} \wedge \textbf{Number} \mapsto \textbf{String}$ 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, $\mu\alpha.(1 \mapsto \mathbf{Number} \wedge 2 \mapsto \alpha?)$ 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

$$
\begin{aligned}
\textbf{true} &\leadsto \textbf{Bool} \\
\textbf{false} &\leadsto \textbf{Bool} \\
n &\leadsto \textbf{Number} \\
s &\leadsto \textbf{String}
\end{aligned}
$$

nullable coercions

$$
\begin{aligned}
\textbf{nil} &\leadsto \tau? \\
\tau &\leadsto \tau?
\end{aligned}
$$

tagging coercions

$$
\begin{aligned}
\textbf{nil} &\leadsto \mathcal{D} \\
\textbf{Bool} &\leadsto \mathcal{D} \\
\textbf{Number} &\leadsto \mathcal{D} \\
\textbf{String} &\leadsto \mathcal{D} \\
\tau? &\leadsto \mathcal{D} \text{ iff } \tau \leadsto \mathcal{D} \\
\mathcal{D} \mapsto \mathcal{D} &\leadsto \mathcal{D} \\
\tau_1 \times \ldots \times \tau_n \rightarrow \mathcal{D}^* \text{ with } n > 0 &\leadsto \mathcal{D} \text{ iff } \tau_1 = \mathcal{D} \wedge \ldots \wedge \tau_{n-1} = \\
&\quad \mathcal{D} \wedge (\tau_n = \mathcal{D} \vee \tau_n = \mathcal{D}^*)
\end{aligned}
$$

tuple coercions

$$
\begin{aligned}
\textbf{empty} &\leadsto \textbf{nil} \times \ldots \times \textbf{nil} \\
\tau_1 \times \ldots \times \tau_n &\leadsto \upsilon_1 \times \ldots \times \upsilon_n \text{ iff } \tau_k \leadsto \upsilon_k \\
\tau_1 \times \ldots \times \tau_n &\leadsto \tau_1 \times \ldots \times \tau_n \times \text{nil} \\
\tau_1 \times \ldots \times \tau_n \times \mathcal{D}^* &\leadsto \upsilon_1 \times \ldots \times \upsilon_n \times \mathcal{D}^* \text{ iff } \tau_k \leadsto \upsilon_k \\
\tau_1 \times \ldots \times \tau_n \times \mathcal{D}^* &\leadsto \tau_1 \times \ldots \times \tau_n \times \mathcal{D} \times \mathcal{D}^* \\
\tau_1 \times \ldots \times \tau_n &\leadsto \mathcal{D}^* \text{ iff } \tau_k \leadsto \mathcal{D} \\
\tau_1 \times \ldots \times \tau_n \times \mathcal{D}^* &\leadsto \mathcal{D}^* \text{ iff } \tau_k \leadsto \mathcal{D}
\end{aligned}
$$

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:

$$\frac{\Gamma \vdash l_k : \tau_k \quad \Gamma \vdash el : v_1 \times \ldots \times v_m \quad m \geq |\vec{l}| \quad v_k \rightsquigarrow \tau_k}{\Gamma \vdash \vec{l} = el : \textbf{void}}$$

$$\frac{\Gamma \vdash l_k : \tau_k \quad \Gamma \vdash el : v_1 \times \ldots \times v_m \quad m < |\vec{l}| \quad v_k \rightsquigarrow \tau_k \quad \textbf{nil} \rightsquigarrow \tau_l \quad l > m}{\Gamma \vdash \vec{l} = el : \textbf{void}}$$

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:

$$\frac{}{\Gamma \vdash \mathbf{nothing} \colon \mathbf{empty}}$$

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

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

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

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:

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

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

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:

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

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

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

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

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

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

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

if $f(\textbf{empty})_n$ 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:

```
local $f$ = fun($x$) return $x + 2$ in
  local $g$ = fun($x$, $y$) return $x + y$ in
    local $h = g$ in
      if $z$ then return $h(2, 3)$ else $h = f$; return $h(3, 2)$
```

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 $\mathcal{D} \to \mathcal{D}^*$ and g has type $\mathcal{D} \times \mathcal{D} \to \mathcal{D}^*$, 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, **Number** × **Number** → **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:

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

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

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:

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

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

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

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

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:

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

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:

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

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:

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

We will have the first and second table constructors (and, by extension, variables a and b) sharing the same precise type "*foo*" $\mapsto$ **Number**?$\wedge$"*bar*" $\mapsto$ **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 **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 **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

```
**local** $f$ = **fun**($x$) **return** $x$.foo **in**
  **local** $a$ = **{}** **in**
    **local** $b$ = **{}** **in**
      $a$.foo = 3; $a$.bar = “$s$”; $b$.foo = 5; **return** $f(a), f(b)$
```

that we used in the last section. In the first iteration we have the function definition getting a fresh function type F with $\mathcal{V}(F) = X_{F_1} \to \times R$, 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 $\mathcal{V}(P) =$ "foo" $\mapsto X_{P_1}$ and $\mathcal{V}(X_{P_1}) = 3$. After the second assignment we have $\mathcal{V}(P) =$ "foo" $\mapsto X_{P_1} \wedge$ "bar" $\mapsto X_{P_2}$ with $\mathcal{V}(X_{P_2}) =$ "s". After the third statement we have $\mathcal{V}(Q) =$ "foo" $\mapsto X_{Q_1}$ 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 "foo" $\mapsto X_{P'_1}$ 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 $\mathcal{V}(P') =$ "foo" $\mapsto X_{P'_1} \wedge$"bar" $\mapsto X_{P'_2}$ and $\mathcal{V}(X_{P'_2}) =$ "s". 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**?) $\to$ **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:     $\Gamma := \{\}$
3:     repeat
```

```
4:         INFERSTEP($\Gamma$, $root$)
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($\Gamma, \langle l_1, \ldots, l_n = el\rangle$)
2:     INFERSTEP($\Gamma, el$)
3:     let $\langle \upsilon_1 \times \ldots \times \upsilon_m\rangle = \mathcal{V}([[el]])$
4:     if $m \geq n$ then
5:         for $k := 1, n$ do
6:             INFERSTEP($\Gamma, l_k$)
7:             UPDATE($\upsilon_k$, $\mathcal{V}([[l_k]])$)
8:         end for
9:     else
10:        for $k := 1, m$ do
11:            INFERSTEP($\Gamma, l_k$)
12:            UPDATE($\upsilon_k$, $\mathcal{V}([[l_k]])$)
13:        end for
14:        for $k := m + 1, n$ do
15:            INFERSTEP($\Gamma, l_k$)
16:            UPDATE(nil, $\mathcal{V}([[l_k]])$)
17:        end for
18:    end if
19:    $[[l_1, \ldots, l_n = el]]$ := 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($\Gamma, \langle e_1, \ldots, e_n \rangle$)
2:     for $k := 1, n$ do
3:         INFERSTEP($\Gamma, e_k$)
4:     end for
5:     $[[e_1, \ldots, e_n]] := \mathcal{V}([[e_1]]) \times \ldots \times \mathcal{V}([[e_n]])$
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 $\tau \overset{\mathcal{V}}{\leadsto} \upsilon'$ and $\upsilon \overset{\mathcal{V}}{\leadsto} \upsilon'$, where $\overset{\mathcal{V}}{\leadsto}$ is the coercion relation lifted for type variables. For example, this is the definition of UPDATE when $\tau = \textbf{nil}$, used in INFERSTEP for assignments:

```
1: procedure UPDATE($\textbf{nil}, X$)
2:     match $\mathcal{V}(X)$ with
3:         case $\varepsilon$: $X := \textbf{nil}$
4:         case $n$: $X := \textbf{Number}?$
5:         case $s$: $X := \textbf{String}?$
6:         case true | false: $X := \textbf{Bool}?$
7:         case $\mathcal{D} \mid \tau?$: break
8:         otherwise: $X := \mathcal{V}(X)?$
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 $\textbf{nil} \overset{\mathcal{V}}{\leadsto} X$ 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 $\tau \leadsto \mathcal{D}$ 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($F, \langle X$ when $\mathcal{V}(X) = \mathcal{D} \rangle$)
```

```
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($\Gamma, \langle \mathbf{fun}(x_1, \ldots, x_n)\ s; \mathbf{return}\ el\rangle$)
2:     if $\mathcal{V}([[\mathbf{fun}(x_1, \ldots, x_n)\ s; \mathbf{return}\ el]]) = \varepsilon$ then
3:         let $R = \mathbf{newvar}$
4:         let $\tau = \underbrace{\mathbf{newvar} \times \ldots \times \mathbf{newvar}}_{n} \to R$
5:         let $F = \mathbf{newvar}\ \tau$
6:         $[[\mathbf{fun}(x_1, \ldots, x_n)\ s; \mathbf{return}\ el]] := F$
7:     end if
8:     let $F = \mathcal{V}([[\mathbf{fun}(x_1, \ldots, x_n)\ s; \mathbf{return}\ el]])$
9:     let $\langle X_1 \times \ldots \times X_m \to R\rangle = \mathcal{V}(F)$
10:    INFERSTEP($\Gamma[x_1 \mapsto X_1, \ldots, x_n \mapsto X_n, rv \mapsto R], \langle s; \mathbf{return}\ el\rangle$)
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:     $P := \mathcal{D} \mapsto (\mathbf{newvar}\ \mathcal{D})$
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($\Gamma, \langle\{\}\rangle$)
2:     if $\mathcal{V}([[\{\}]]) = \varepsilon$ then
3:         let $P$ = newvar
4:         let $T$ = newvar
5:         let $T := P$
6:         $[[\{\}]] := T$
7:     end if
8:     let $T = \mathcal{V}([[\{\}]])$
9:     let $P = \mathcal{V}(T)$
10:    if $\mathcal{V}(P) \neq \varepsilon$ then
11:        let $\langle \tau_1 \mapsto X_1 \wedge \ldots \wedge \tau_n \mapsto X_n \rangle = \mathcal{V}(P)$
12:        for $k := 1, n$ do
13:            UPDATE(nil, $X_k$)
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($F, \langle X$ when $\mathcal{V}(X) = G\rangle$)
2:     if $\mathcal{V}(F) \neq \mathcal{V}(G)$ then
3:         let $\langle X_1 \times \ldots \times X_m \rightarrow R\rangle = \mathcal{V}(F)$
4:         let $\langle Y_1 \times \ldots \times Y_n \rightarrow S\rangle = \mathcal{V}(G)$
5:         let $k$ = max $m, n$
```

```
 6:         let $R' =$ newvar
 7:         $G := \underbrace{\textbf{newvar} \times \ldots \times \textbf{newvar}}_{k} \rightarrow R'$
 8:         $F := \mathcal{V}(G)$
 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($\mapsto T, \langle X$ when $\mathcal{V}(X) = U\rangle$)
2:     if $\mathcal{V}(T) \neq \mathcal{V}(U)$ 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($\Gamma, \langle l$ when $l = \langle e_1[e_2]\rangle\rangle$)
 2:     INFERSTEP($\Gamma, e_1$)
 3:     INFERSTEP($\Gamma, e_2$)
 4:     match $\mathcal{V}([[e_1]])$ with
 5:         case $T$:
 6:             match FIND($\mathcal{V}([[e_2]]), \mathcal{V}(T)$) with
 7:                 case $\varepsilon$:
 8:                     let $X =$ newvar
 9:                     UNION($\mathcal{V}([[e2]]) \mapsto X, \mathcal{V}(T)$)
10:                     $[[l]] := X$
11:                 end
12:                 case $X$: $[[l]] := X$
13:             end match
14:         end
15:         case otherwise:
16:             UPDATE($\mathcal{V}([[e_1]])$, newvar $\mathcal{D}$)
```

```
            UPDATE($\mathcal{V}([[e_2]])$, **newvar** $\mathcal{D}$)
            $[[l]] :=$ **newvar** $\mathcal{D}$
        **end**
    **end match**
**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 $\upsilon \overset{\mathcal{V}}{\leadsto} \tau_k$. 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$(\langle \upsilon \mapsto X \rangle, P)$:

```
**procedure** UNION($\langle \upsilon \mapsto X \rangle, P$)
    **if** $\mathcal{V}(P) = \varepsilon$ **then**
        $P := \upsilon \mapsto X$
    **else**
        **let** $\langle \tau_1 \mapsto X_1 \wedge \ldots \wedge \tau_n \mapsto X_n \rangle = \mathcal{V}(P)$
        **let** $P' =$ **newvar** $(\upsilon \mapsto X)$
        **for** $k := 1, n$ **do**
            **if** $\tau_k \overset{\mathcal{V}}{\leadsto} \upsilon$ **then**
                UPDATE($\mathcal{V}(X_k), X$)
            **else**
                $P' := \mathcal{V}(P') \wedge \tau_k \mapsto X_k$
            **end if**
        **end for**
        $P := \mathcal{V}(P')$
    **end if**
**end procedure**
```

The implementations of FIND and $\tau \overset{\mathcal{V}}{\leadsto} \upsilon$ are straightforward. The implementation of $\overset{\mathcal{V}}{\leadsto}$ 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.

4
Benchmarks

This chapter presents our suite of benchmarks and the results of those benchmarks on different variations of our Lua to CLR compiler and different implementations of the CLR. We analyze the performance impact of our changes to build a partial performance model of these implementations. We also benchmark our compilers against the Lua interpreter on x86, a Lua x86 JIT compiler, and a Microsoft-developed Python to CLR compiler. These benchmarks respectively serve to find out how well Lua can perform on the CLR relative to other Lua implementations, and how well our approach for compiling a dynamic language in the CLR works relative to a compiler using the Microsoft DLR.

Section 4.1 describes the programs we used in our benchmarks and the Lua operations and idioms that they exercise. Section 4.2 gives the results and analysis of our benchmarking of the different variations of our Lua to CLR compiler, and Section 4.3 compares compilers with other Lua implementations and with a Python compiler for the CLR. Appendix C has tables with running times for all of the benchmarks in this chapter.

4.1
Benchmark Programs

Our first suite of benchmarks is a suite of small (less than fifty lines of code) numerical benchmarks mostly taken from a suite of bencharks that compare several programming languages [Brent A. Fulgham and Isaac Gouy, 2009]. They are useful because implementations of these benchmarks are readily available for several programming languages, they have few dependencies on Lua's standard library, and they have no dependencies on the platform facilities. These benchmarks are naive, but useful for testing the impact of specific optimizations. Small benchmarks are useful for comparing performance of different implementations, and to guide programmers on how to tailor their programs to get the best performance out of a particular implementation [Gabriel, 1985].

The benchmarks are listed on Table 4.1, with a brief description of what they do and what they exercise (what are the main influences on their results).

Name	Description
binary-trees	allocation and traversal of binary trees, exercises small records, memory allocation and GC
fannkuch	array permutations with a small array, exercises array operations
fib-iter	fibonnaci function, iterative algorithm, exercises simple arithmetic and loops
fib-memo	fibonnaci function, recursive algorithm with memoization, exercises array operations on a variable-sized array and first-class functions
fib-rec	fibonnaci function, recursive algorithm, exercises recursion
mandelbrot	mandelbrot fractal, exercises floating point arithmetic and iteration
n-body	newtonian gravity simulation, exercises floating point arithmetic on records and arrays
n-sieve	sieve of Eratosthenes using an array, exercises array access
n-sieve-bits	sieve of Eratosthenes using bitfields, exercises floating point arithmetic and arrays
partial-sum	iterative summation of series, exercises floating point arithmetic and built-in functions with iteration
recursive	several recursive functions, exercises recursion
spectral-norm	spectral norm of an infinite matrix, exercises arithmetic involving several cooperating functions

Table 4.1: First benchmark suite

The overlapping coverage of these benchmarks of the first suite is on purpose; similar benchmarks should respond in a similar way to changes in the implementation of the compiler.

We also implemented a second suite of benchmarks. It is a set of variations on the Richards Benchmark [Richards, 1999], a medium-sized benchmark. The benchmark implements the kernel of a very simple operating system, with a task dispatcher, input/output devices, and worker tasks that communicate via message passing. The benchmark has implementations for several programming languages. Its output is deterministic, so it is trivial to verify that a particular implementation is correct; the simulation uses pseudorandom numbers, but the pseudorandom number generator is part of the benchmark.

We have six different implementations of the benchmark, with different ways of implementing the core task dispatcher; the implementations have around three hundred lines of Lua code each. For comparison, the C version of the benchmark has four hundred lines of code, and the Python version has about four hundred and fifty.

Our implementations of the Richards benchmarks are listed on Table 4.2.

Name	Description
richards	the closest to the C implementation, it uses an explicit state string and a sequence of ifs inside an endless loop as the core of the dispatcher; the state string simulates a bitfield
richards-tail	has a state machine using tail calls for the state transitions instead of an endless loop, with the code to implement each state factored out to a different function, but otherwise is identical to the previous one
richards-oo	embeds the dispatcher logic inside each task as several methods, and keeps the state local in each task object; state transition is via a trampoline: each task returns the next task to be processed
richards-oo-tail	the previous implementation using tail calls instead of the trampoline, each task calls the next one to be processed directly
richards-oo-meta	like richards-oo but uses a parent table to hold the methods, and a metatable associated with each task object that delegates method calls to this parent table
richards-oo-cache	also uses a parent table, but the metatable caches each method in the task object itself after the first use, to speed up subsequent lookups

Table 4.2: Second benchmark suite

4.2 Benchmarking the Variations

This section compares the different variations of our Lua compiler on different implementations of the CLR. We present a series of graphs of performance improvement (or worsening) relative to the base compiler described in Section 2.1. All graphs show a base 2 logarithm of the running time of the benchmark compiled by the base compiler divided by the running time of the benchmark compiled by each of the other compilers (e.g. -1 means the benchmark took twice as long, 2 means it took a fourth of the time). The other compilers are identified by the short names listed on Table 2.1. All of the tests are done on a computer with an AMD Phenom 8400 processor with 2Gb of RAM and executing in x86 (32-bit) mode.

The first CLR implementation we benchmarked is Microsoft .NET 3.5 SP1, the current version of the CLR released by Microsoft. Figure 4.1 shows the results of the first benchmark suite on this version of the CLR.

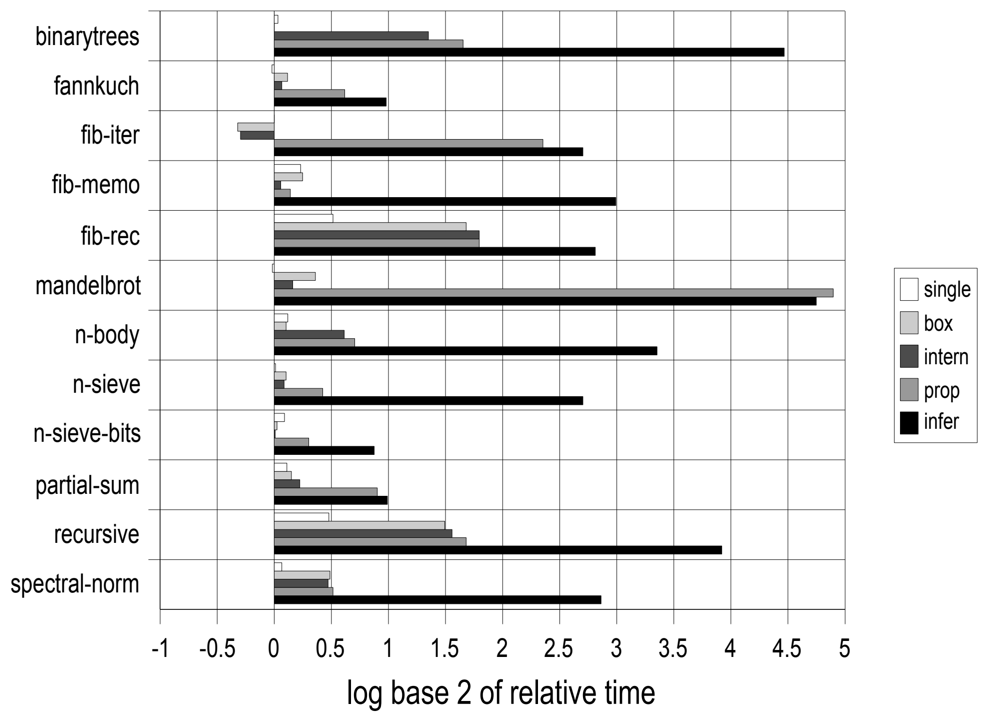

Figure 4.1: .NET 3.5 SP1 Comparison

The results show barely any improvement with the optimization of the function calls that only need to return a single value when we are using a structure to represent Lua values. Changing the representation to avoid structures and use boxing instead improves most benchmarks, specially the ones that make heavy use of function calls. The CLR JIT does not optimize code that uses structures as well as other code, so the performance gain in avoiding allocation of the unnecessary arrays gets lost in the noise [Morrison, 2008b].

Interning strings leads to a good improvement in the benchmarks that use records, confirming that this implementation of the CLR does not intern strings. Local type propagation shows a big improvement only in the benchmarks where the core of the benchmark is a numerical loop inside a single function, but for these benchmarks it is about as good as the full type inference. The biggest improvement comes from doing full type inference on the programs. The running time for the benchmarks of the first suite is dominated by boxing, type checking, and dispatch, and type inference is eliminating most of those.

The results for the second suite of benchmarks are on Figure 4.2. Neither the base compiler nor the single return value compiler could run the *richards-oo-tail*; they both blow the stack because the CLR JIT is not compiling the

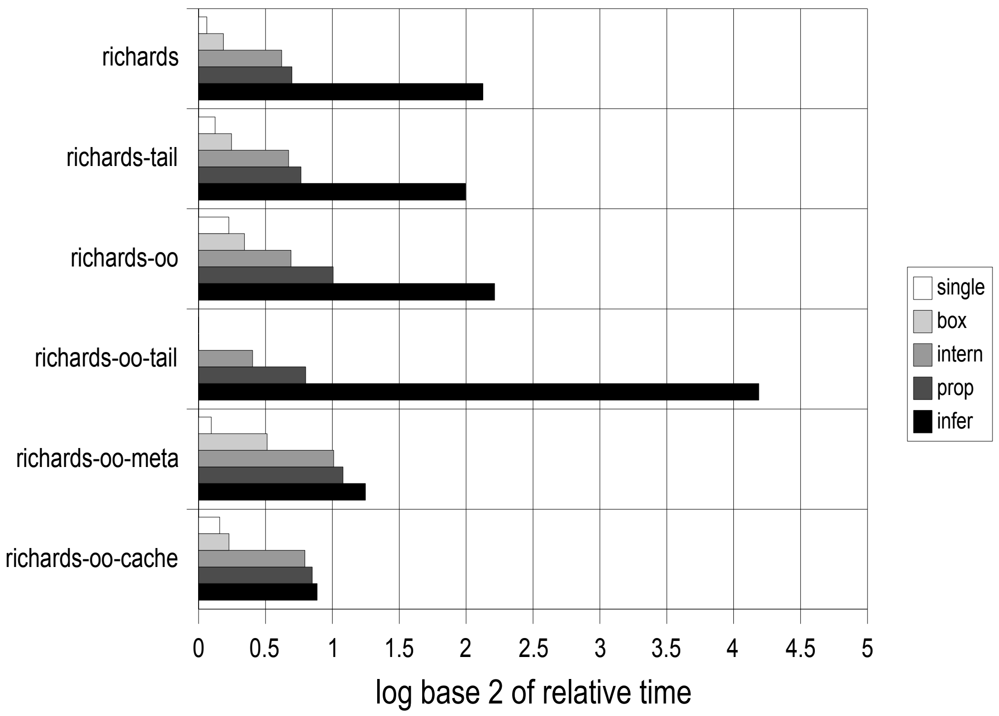

Figure 4.2: .NET 3.5 SP1 Comparison, Richards benchmarks

tail calls in them as actual tail calls, and they continue using stack space. The .NET JIT treats the tail call opcodes as just a suggestion, and ignores it in several situations [Broman, 2007]. Only the JIT on version 4.0 of .NET for the x64 platform will always honor the tail call opcodes [Richins, 2009]. The results for *richards-oo-tail* on Figure 4.2 are relative to the "box" compiler.

All of the benchmarks in this suite do a great number of string equality tests, so interning strings shows a good improvement for all of them. Local type propagation shows a modest improvement, but the biggest improvement again comes from doing full type inference. The big improvement in the *richards-oo-tail* benchmark for type inference relative to the improvement in *richards-oo* is due to another anomaly of tail calls in the .NET implementation of the CLR, where a tail call to a function with a different number of arguments than the current function interacts badly with the code that synchronizes with the thread running the garbage collector [Borde, 2005]; type inference is unifying all of the mutually recursive functions in *richards-oo-tail* to have the same signature, avoiding the problem and thus increasing the amount of improvement in relation to the other compilers.

The benchmarks that use metatables, *richards-oo-meta* and *richards-oo-cache*, show little improvement with type inference, as our type inference algorithm always infers the most general table type for tables that have a

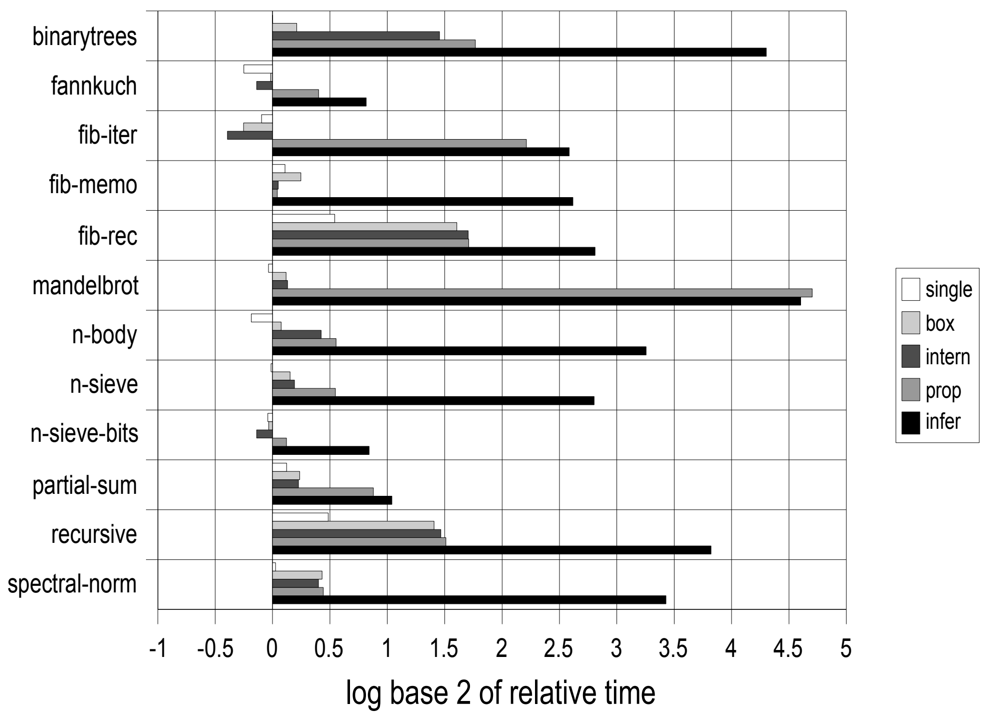

Figure 4.3: .NET 4.0 Beta 1 Comparison

metatable attached. This ends up spreading to functions used as methods.

We also ran our benchmarks on Microsoft .NET 4.0 Beta 1, the current beta of the next version of Microsoft's implementation of the CLR. The results are on Figure 4.3 and Figure 4.4. They are about the same as the results for .NET 3.5 SP1, showing that the behavior of the JIT compiler in both versions is similar. Microsoft .NET 4.0 Beta 1 has the same issues with tail calls that .NET 3.5 SP1 has, so our "base" and "single" compilers also cannot complete the *richards-oo-tail* benchmark.

Finally, we ran our benchmarks on Mono 2.4, an open-source implementation of the CLR. Figure 4.5 shows the results for the first suite of benchmarks. They are very different from the results of both Microsoft implementations, showing the different performance models of Mono and .NET, even though they both are implementations of the same managed runtime environment. Boxing in Mono performs much worse than using structures, and when using structures there is a good improvement in code that uses function calls when avoiding unnecessary array allocations. The best results are still obtained by doing full type inference.

Figure 4.6 shows the results for the suite of Richards benchmark on Mono 2.4. Arithmetic is not as critical for the benchmarks in this suite as in the benchmarks of the first suite, so the "box" compiler, which also optimizes

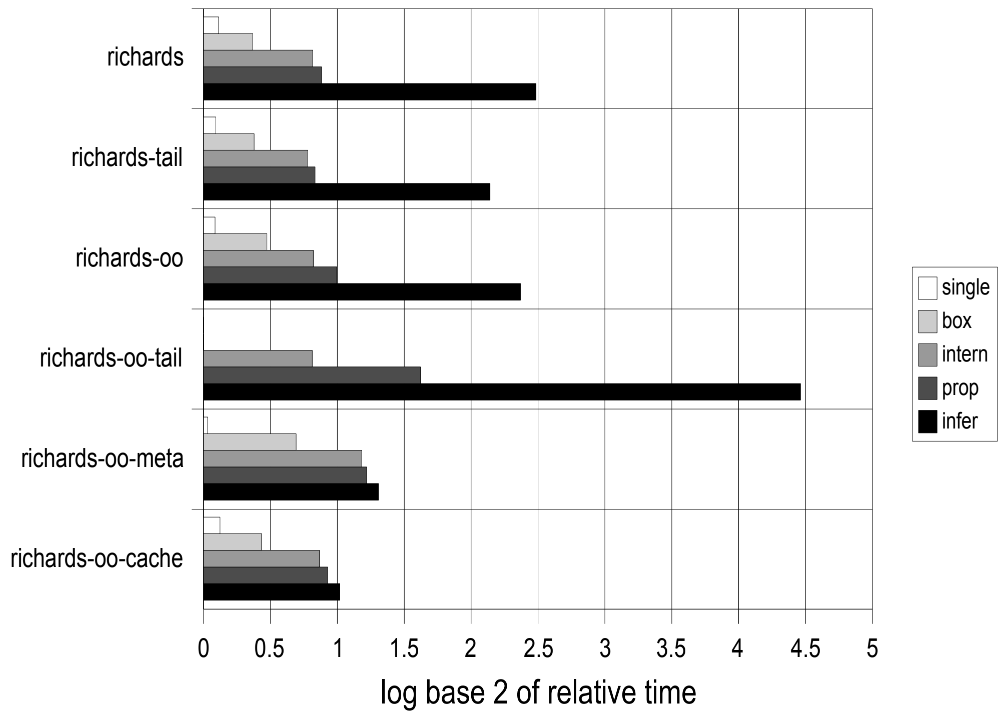

Figure 4.4: .NET 4.0 Beta 1 Comparison, Richards benchmarks

returning single values, shows an improvement in these benchmarks despite using boxed numbers. Mono also does not intern its strings, so we get a good improvement when doing that, relative to the "box" compiler. But none of the benchmarks that use tail calls ran on Mono without blowing the stack.

The differences between the Mono and .NET implementations are big enough to change the optimal compilation: for Mono the best approach is to combine type inference with a structure as a fallback uniform representation, while for .NET it is to combine type inference with a uniform representation that uses boxing, the one we actually implemented. Our previous work showed that the penalty for using structures was even greater in a previous version of the .NET CLR [Mascarenhas and Ierusalimschy, 2008]; the current version of the .NET CLR is just over one year old at the time of the writing of this dissertation, and is the first to have improved code generation for programs that use structures [Morrison, 2008b]. The performance characteristics vary not only between competing implementations of the same runtime, but also between different versions of the same implementation, so the best approach for building a compiler that targets a managed runtime environment can depend on a specific version of a specific implementation of the runtime.

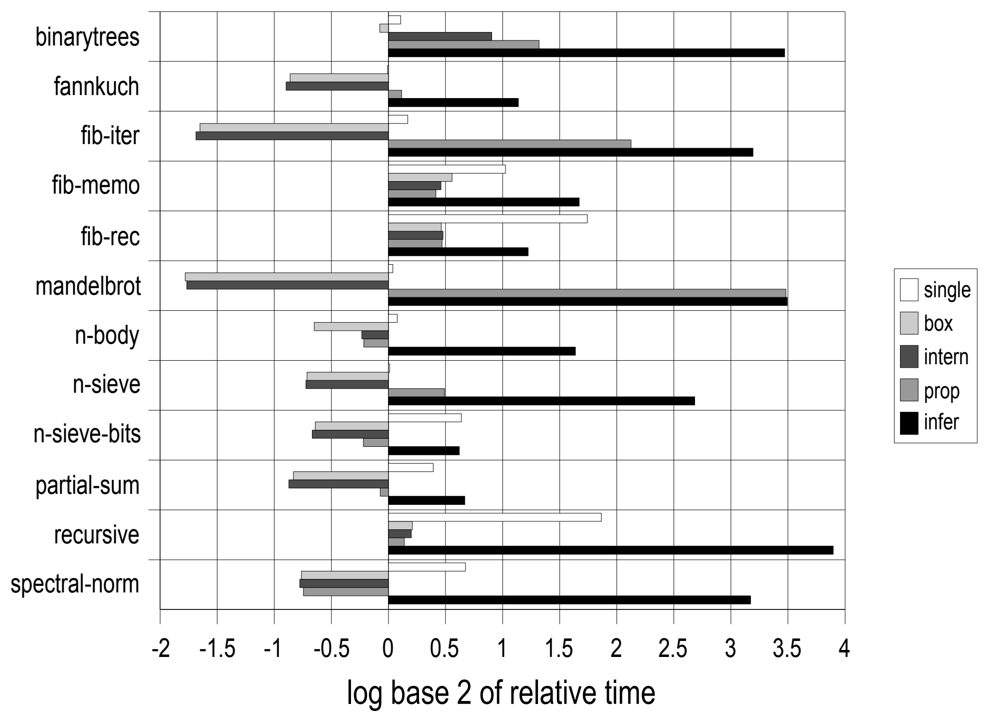

Figure 4.5: Mono 2.4 Comparison

4.3 Other Benchmarks

This section compares our Lua compilers first with other Lua implementations and then with IronPython 2.0, a Python compiler for the CLR that we already reviewed in Section 2.3. We first show the results of benchmarking our Lua compilers that were able to run all of the benchmarks in both suites (“box”, “intern”, “prop”, and “infer”) against version 5.1.4 of the Lua interpreter and LuaJIT 1.1.5, a JIT compiler for Lua 5.1. The graphs for this benchmark show the performance of each of our compilers, plus LuaJIT, relative to the Lua interpreter, also as the base 2 logarithm of the relative running times.

Figure 4.7 shows the results of the first benchmark suite. With type inference our last compiler is able to generate, with the help of the .NET JIT, code that performs better than LuaJIT for most benchmarks, and better than the Lua interpreter for all benchmarks. Local type propagation, in the three benchmarks where it was most useful, gets similar results. Our other two compilers still outperform the Lua interpreter in several benchmarks, but are worse than the interpreter in benchmarks that depend on floating point arithmetic, by a factor of two in some cases. The Lua interpreter is very efficient doing floating point computations, as it always works with unboxed numbers.

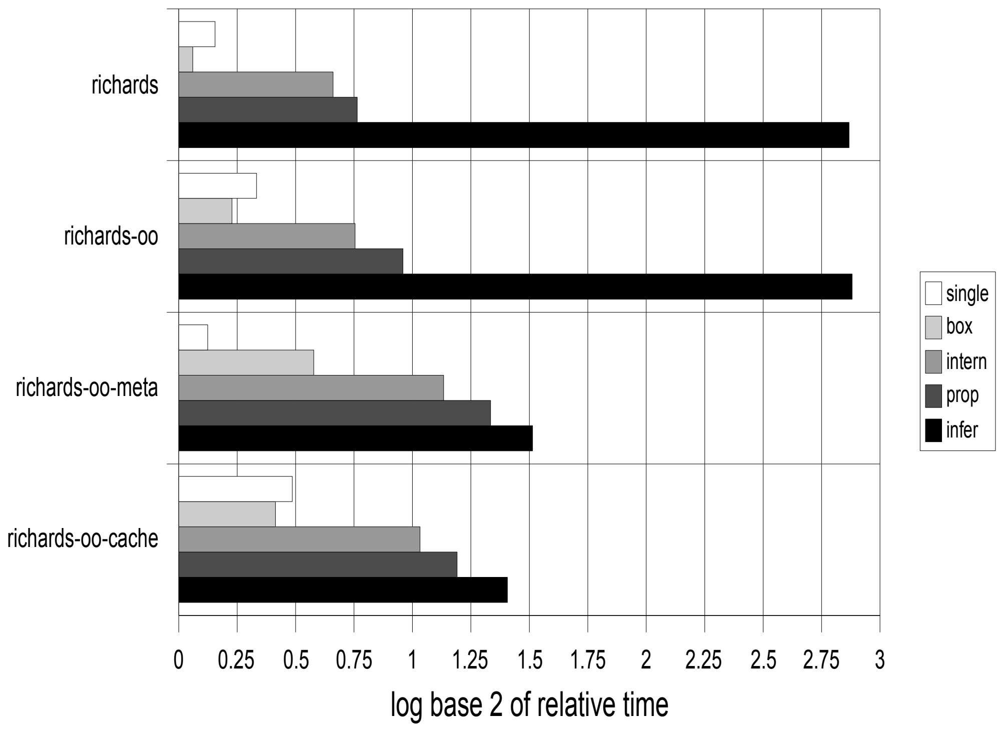

Figure 4.6: Mono 2.4 Comparison, Richards benchmarks

The results of the second benchmark suite are on Figure 4.8. The issue with tail calls in the .NET CLR is clearer to see here, as we are now comparing against the Lua interpreter. While the "box" compiler is about a factor of two slower than the Lua interpreter in most benchmarks, for the *richards-oo-tail* benchmark it is approximately ten times slower than the Lua interpreter. The gap narrows for the "intern" and "prop" compilers but is gone only for the last compiler, where this benchmark performs similarly to the *richards-oo* benchmark. This is consistent with the reason for the anomaly that we discussed in the previous section.

Combining the results of both benchmark suites, we have the benchmarks running in at most twice the time as the Lua interpreter, but usually running in a similar amount of time (except for the outlier, the *richards-oo-tail* benchmark). The performance is on par or exceeds the performance of a x86 Lua JIT compiler when our type inference algorithm is able to assign more precise types. These results support our assertion in Chapter 1 that it is possible to generate efficient code from a dynamically-typed source language to a managed runtime.

Our last benchmark compares our compilers with IronPython 2.0, a Python compiler for the CLR that we reviewed in Section 2.3. For this benchmark we added the *richards-oo* benchmark to the benchmarks of the

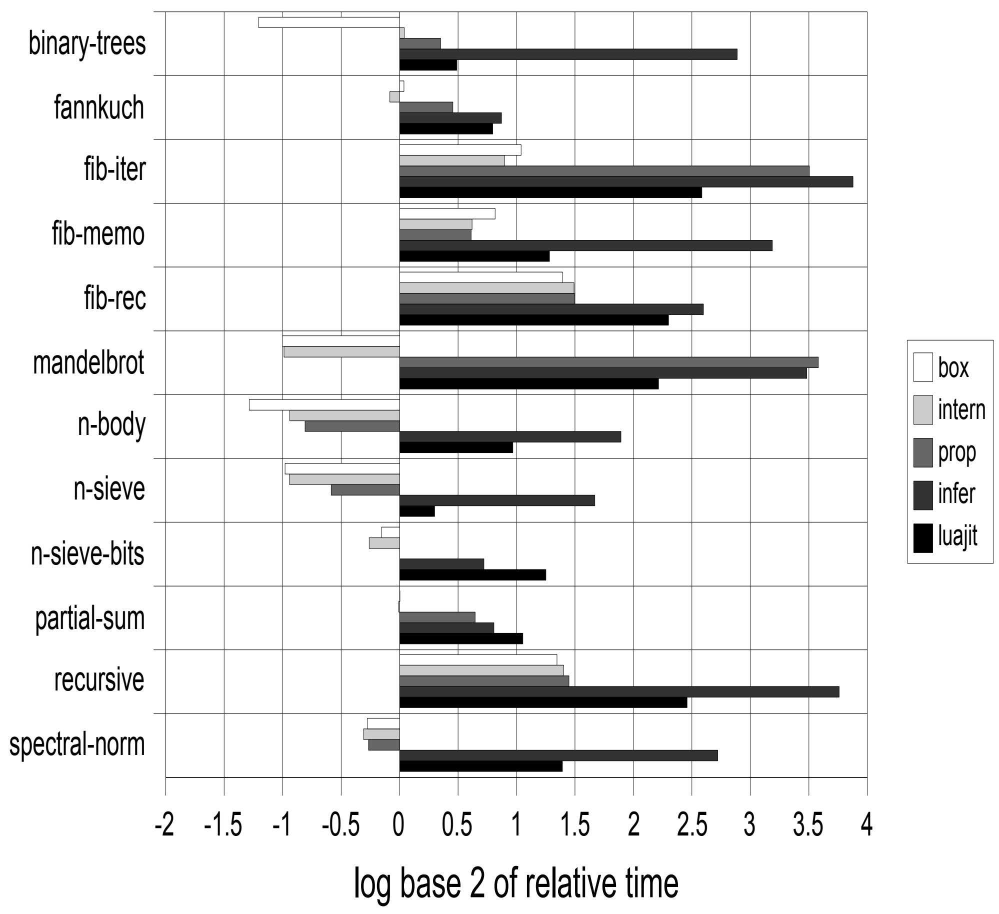

Figure 4.7: Comparison with Lua 5.1.4

first suite, as there is a single implementation of the Richards benchmark for Python. Figure 4.9 shows the performance relative to our "box" compiler, again by showing the base 2 logarithm of the relative running times ("ipy" is the IronPython compiler).

Python has separate array and dictionary types, while Lua arrays (in the absence of precise type inference) are an optimization of tables. This explains the better performance of IronPython in the *binarytrees*, *fib-memo* and *n-sieve* benchmarks, relative to our compilers that do not have type inference. Our compilers outperform IronPython in the other benchmarks even without type inference. With type inference we outperform IronPython in all of the benchmarks, in almost all of them by more than a factor of four, as the code IronPython generates is still doing runtime type checking and still boxing numbers.

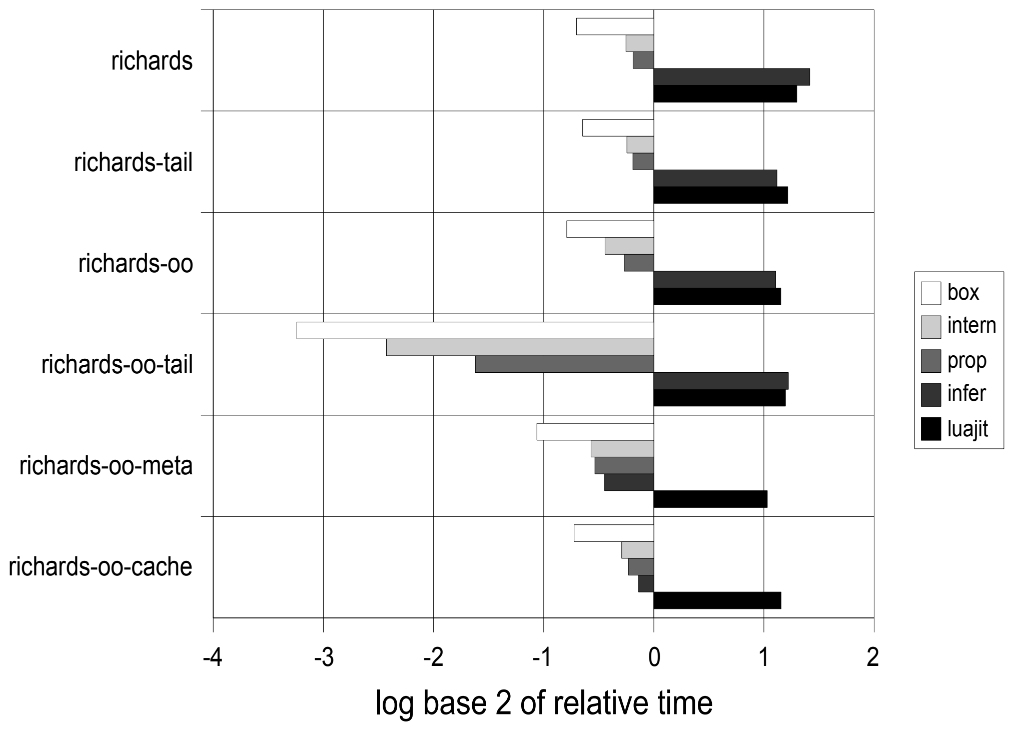

Figure 4.8: Comparison with Lua 5.1.4, Richards benchmarks

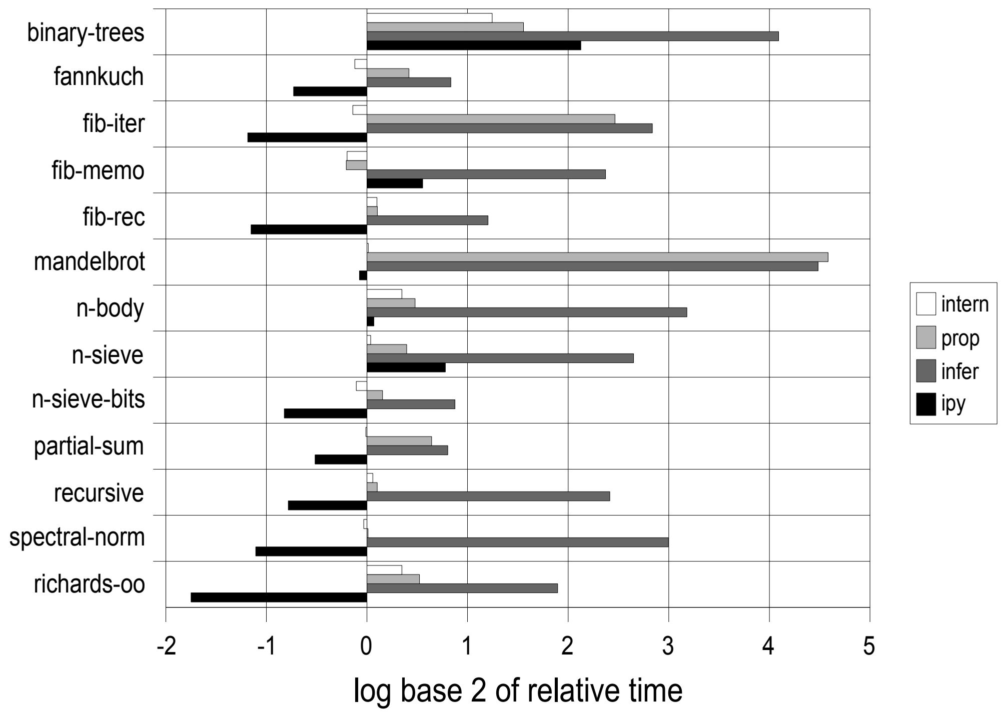

Figure 4.9: Comparison with IronPython

5
Conclusions

Writing an optimizing compiler for a managed runtime involves guesswork and experimentation. Instead of targeting low-level machine code with a clear performance model we are targeting a high-level language with its own type system, runtime library, and optimizing Just-In-Time compiler. Not only it is difficult to predict how a particular approach will perform, but the performance can vary among different implementations of the managed runtime, or even different versions of the same implementation.

We have shown the difficulty in compiling to a managed runtime by building a series of compilers for the Lua programming language that targets the Common Language Runtime. We built several compilers with different ways to represent Lua types in the CLR type system and different ways to compile Lua operations, and then benchmarked these compilers on different implementations of the CLR. Our benchmarks show how the best approach for compiling Lua to the CLR depends on what implementation of the CLR we are targeting.

The choice of implementation approach and implementation target influences not only the performance of our Lua implementation but also its semantics, as tail call optimization does not work for some combinations of implementations of our compiler and implementations of the CLR, even though it should have worked by the CLR standard. There are other corners of Lua's semantics that are problematic to implement in the CLR: weak tables, finalizers, and coroutines, but we already covered these in Mascarenhas and Ierusalimschy [2005], so we focused on the efficient implementation of Lua's core semantics for the Lua compilers of this dissertation.

The influence of the implementation target on the semantics of Lua has parallels to the work on definitional interpreters in Reynolds [1998], where the closer the semantics of the language that you want to interpret is to the semantics of the language you are using to write the interpreter the simpler the interpreter can be. Where the semantics differ you have to implement the correct semantics in terms of the underlying language. With compilers for high-level targets such as managed runtime environments, the closer the semantics of the language you are compiling to the semantics of the runtime the more direct the compilation, and where the semantics differ you need extra scaffolding and support to implement the correct semantics.

Lua is a dynamically typed language, and the CLR is a statically typed runtime, so in most of our compilers all of Lua's operations had to be compiled using runtime type checks and the virtual dispatch mechanism of the CLR. We also had to use a unified representation for Lua values that either wasted memory and interacted in a less than optimal way with the JIT of one of the CLR implementations we tested, or had to store all numbers in boxes in the heap instead of using the efficient native representation for floating-point numbers that the CLR has.

We specified a type system and type inference algorithm for Lua that can statically assign more precise types to several kinds of Lua operations. We implemented this algorithm in one of our compilers, and used its output to generate more efficient representations for Lua values and better performing code. Analysis of the output of the type inference algorithm and the performance gains showed that the type inference algorithm correctly infers precise types for most variables and operations in our benchmarks.

Compared to other Lua implementations, our best combination of Lua compiler without type inference and CLR implementation has the level of performance of version 5.1.4 of the Lua interpreter, being worse by a factor of less than two in benchmarks that are heavily dependent on floating point computation, and faster by a little over a factor of two in benchmarks that are heavily dependent on recursion.

With type inference, our best combination of Lua compiler and CLR implementation outperforms the Lua interpreter and performs better than version 1.1.5 of the LuaJIT compiler for most benchmarks. Our results show that it is possible to get good performance out of a dynamic language in a managed runtime if the managed runtime has a good implementation.

We also benchmarked our Lua compilers against IronPython 2.0, a Python compiler for the CLR that uses a different implementation approach based on runtime generation of specialized code, in contrast to our simpler approach that only uses offline compilation. Without type inference our best compiler performs equal or better than IronPython in almost all benchmarks; with type inference our best compiler outperforms IronPython by a large margin in all benchmarks. We believe our approach is the best one for compiling Lua on the CLR given the current state of the Dynamic Language Runtime that IronPython is built on.

Future implementations of the CLR may change the impact of some of our implementation decisions, so the specific results may change, but this only restates our general thesis that the optimal implementation approach depends on the specific implementations that are the targets.

Type inference was the key to the optimizations with the most impact on performance, and this suggests directions for future research. Our type inference algorithm works just as badly across module boundaries as the local type propagation of Section 2.2.4 works across function call boundaries. Parameters of exported functions in our type system always have the dynamic type, and imported functions also always return values of this type.

Recent work on *gradual typing* [Siek and Taha, 2006, 2007] and *Typed Scheme* [Tobin-Hochstadt and Felleisen, 2008] may lead to an approach combining type annotations in module boundaries with type inference used for intra-module optimization. Gradual typing is a type system where parts of a program may be annotated with precise types, and parts not annotated have a dynamic type. The type system guarantees that type errors only occur in the dynamically-typed portions of the program. Typed Scheme is a gradual typing system for Scheme that lets the programmer mix statically and dynamically typed Scheme code. The Typed Scheme runtime is still the same runtime as regular Scheme; the type annotations provide static type safety, not increased performance through removal of type checks or better representations.

Gradual typing may be orthogonal to the type system we use for our type inference algorithm. Siek and Vachharajani [2008] has already show that gradual typing is compatible with Hindley-Milner type inference, and Herman et al. [2007] use techniques taken from Heinglein's work on dynamic typing that we reviewed in Section 3.3.1 [Henglein, 1992a]. Combing gradual typing and our type inference should make it possible to have inference working across modules with minimal type annotations, as well as increasing the static type safety of Lua programs.

Bibliography

N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson. Revised^5 report on the algorithmic language Scheme. *SIGPLAN Notices*, 33(9):26–76, 1998. ISSN 0362-1340. 1.1

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1986. ISBN 0-201-10088-6. 2.1.2

Alexander Aiken and Manuel Fähndrich. Dynamic typing and subtype inference. In *FPCA '95: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 182–191, New York, NY, 1995. ACM. 3.3.1

Randall D. Beer. Preliminary report on a practical type inference system for Common Lisp. *SIGPLAN Lisp Pointers*, 1(2):5–11, 1987. ISSN 1045-3563. 3.3.2

Jan Benda, Tomas Matousek, and Ladislav Prosek. Phalanger: Compiling and running PHP applications on the Microsoft .NET platform. In *Proceedings of the .NET Technologies Conference*, pages 31–38, 2006. 2.3

Gregory Blajian, Roger Eggen, Maurice Eggen, and Gerald Pitts. Mono versus .NET: A comparative study of performance for distributed processing. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 45–51, 2006. 2

Shri Borde. Tail call performance on x86, 2005. Available at `http://blogs.msdn.com/shrib/archive/2005/01/25/360370.aspx`. 4.2

Brent A. Fulgham and Isaac Gouy. The computer language benchmarks game, 2009. Available at http://shootout.alioth.debian.org. 4.1

Yannis Bres, Bernard Serpette, and Manuel Serrano. Bigloo.NET: compiling Scheme to .NET CLR. *Journal of Object Technology*, 9(3):71–94, October 2004. 2.3

David Broman. Tail call JIT conditions, 2007. Available at `http://blogs.msdn.com/davbr/pages/tail-call-jit-conditions.aspx`. 4.2

Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, 1985. ISSN 0360-0300. 3.1, 3.1.3

Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 278–292, New York, NY, 1991. ACM. 3.3.1

Bill Chiles and Alex Turner. Dynamic language runtime, 2009. Available at `http://dlr.codeplex.com/`. 1, 2.3

William Clinger. Common Larceny. In *Proceedings of the 2005 International Lisp Conference*, pages 101–107, 2005. 2.3

William D. Clinger and Lars Thomas Hansen. Lambda, the ultimate label or a simple optimizing compiler for Scheme. In *LFP '94: Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 128–139, New York, NY, 1994. ACM. 2.3

Antonio Cuni, Davide Ancona, and Armin Rigo. Faster than c#: efficient implementation of dynamic languages on .net. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 26–33, New York, NY, 2009. ACM. 2.3

Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212, New York, NY, 1982. ACM. 3.1, 3.3.1, 3.3.2

Ana Lúcia de Moura, Noemi Rodriguez, and Roberto Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, 2004. 1

L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 297–302, New York, NY, 1984. ACM. 1, 2.3

Jitendra Dhamija. Introducing managed JScript, 2007. Available at `http://blogs.msdn.com/jscript/archive/2007/05/07/introducing-managed-jscript.aspx`. 2.3

ECMA. ECMAScript language specification, 1999. Available at http://www.ecma-international.org/publications/standards/Ecma-262.htm. 1

Daniel P. Friedman, Christopher T. Haynes, and Mitchell Wand. *Essentials of programming languages (2nd ed.).* MIT, Cambridge, MA, 2001. ISBN 0-262-06217-8. 2.1.2

Richard P. Gabriel. *Performance and evaluation of LISP systems.* MIT, Cambridge, MA, 1985. ISBN 0-262-07093-6. 4.1

Richard P. Gabriel and Kent M. Pitman. Endpaper: Technical issues of separation in function cells and value cells. *Lisp and Symbolic Computation*, 1(1):81–101, 1988. 3.3.2

Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA '07: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 57–76, New York, NY, 2007. ACM. 2

Carsten K. Gomard. Partial type inference for untyped functional programs. In *LFP '90: Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 282–287, New York, NY, 1990. ACM. 3.3.1

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison Wesley, 2005. ISBN 978-0321246783. 2.1.2

Dayong Gu, Clark Verbrugge, and Etienne M. Gagnon. Relative factors in performance analysis of Java virtual machines. In *VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 111–121, New York, NY, 2006. ACM. 2

Jennifer Hamilton. Language integration in the Common Language Runtime. *SIGPLAN Notices*, 38(2):19–28, 2003. ISSN 0362-1340. 2.3

Mark Hammond. Python for .NET: lessons learned, 2000. Available at `http://word-to-html.com/convertions/Python_for_NET.html`. 2.3

Fritz Henglein. Dynamic typing. In *ESOP'92: Proceedings of the 4th European Symposium on Programming*, pages 233–253, London, UK, 1992a. Springer-Verlag. 3.3.1, 5

Fritz Henglein. Global tagging optimization by type inference. In *LFP '92: Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 205–215, New York, NY, 1992b. ACM. 3.3.1

Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: translating Scheme to ML. In *FPCA '95: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 192–203, New York, NY, 1995. ACM. 3.3.1

David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Proceedings of the Symposium on Trends in Functional Programming*, pages 1–16, April 2007. 5

Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 326–336, New York, NY, 1994. ACM Press. 1, 2.3

Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38, London, UK, 1991. Springer-Verlag. 2.3

Jim Hugunin. IronPython: A fast Python implementation for .NET and Mono. In *Proceedings of PyCon DC 2004*, 2004. Available at `http://www.python.org/pycon/dc2004/papers/9`. 2.3

Jim Hugunin. Ironpython 2.0, 2008. Available at `http://ironpython.codeplex.com/Release/ProjectReleases.aspx?ReleaseId=8365`. 2.3

Jim Hugunin. Ironpython 2.6 alpha 1, 2009. Available at `http://ironpython.codeplex.com/Release/ProjectReleases.aspx?ReleaseId=22982`. 2.3

Roberto Ierusalimschy. *Programming in Lua, Second Edition*. Lua.Org, 2006. ISBN 8590379825. 1, 1.1

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The implementation of Lua 5.0. *Journal of Universal Computer Science*, 11(7): 1159–1176, 7 2005. 2.1.1

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *HOPL III: Proceedings of the third ACM SIGPLAN Conference on History of Programming Languages*, pages 2–1–2–26, New York, NY, 2007. ACM Press. 1

John Lam. IronRuby, 2009. Available at `http://ironruby.net`. 2.3

Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 291–302, New York, NY, 1991. ACM. 3.1.3

Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Prentice Hall, 1999. ISBN 978-0201432947. 1

Robert A. MacLachlan. The Python compiler for CMU Common Lisp. In *LFP '92: Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 235–246, New York, NY, 1992. ACM. 3.3.2

Dragos Manolescu, Brian Beckman, and Benjamin Livshits. Volta: Developing distributed applications by recompiling. *IEEE Software*, 25(5):53–59, 2008. ISSN 0740-7459. 1

Fabio Mascarenhas and Roberto Ierusalimschy. Running Lua scripts on the CLR through bytecode translation. *Journal of Universal Computer Science*, 11(7):1275–1290, 2005. 2.1.1, 2.3, 5

Fabio Mascarenhas and Roberto Ierusalimschy. Efficient compilation of Lua for the CLR. In *SAC '08: Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 217–221, New York, NY, 2008. ACM. 1, 4.2

Fabio Mascarenhas and Roberto Ierusalimschy. LuaInterface: Scripting the .NET CLR with Lua. *Journal of Universal Computer Science*, 10(7):892–909, 2004. 2.1.1

David Mertz. Charming Python: Functional programming in Python, 2001. Avaliable at `http://www.ibm.com/developerworks/library/l-prog2.html`. 2.1.2

Microsoft. ECMA C# and Common Language Infrastructure standards, 2005. Available at `http://msdn.microsoft.com/net/ecma/`. 1, 1.2, 2

Vance Morrison. Measure early and often for performance. *MSDN Magazine*, 9(4), 2008a. Available at `http://msdn.microsoft.com/en-us/magazine/cc500596.aspx`. 2

Vance Morrison. What's coming in .NET runtime performance in version v3.5 SP1, 2008b. Available at `http://blogs.msdn.com/vancem/archive/2008/05/12/what-s-coming-in-net-runtime-performance-in-version-v3-5-sp1.aspx`. 4.2, 4.2

Llewellyn Pritchard. IronScheme, 2009. Available at `http://www.codeplex.com/IronScheme`. 2.3

John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher Order and Symbolic Computation*, 11(4):363–397, 1998. ISSN 1388-3690. 5

Martin Richards. Richards Benchmark, 1999. Available at `http://www.cl.cam.ac.uk/~mr10/Bench.html`. 4.1

Grant Richins. Tail call improvements in .NET framework 4, 2009. Available at `http://blogs.msdn.com/clrcodegeneration/archive/2009/05/11/tail-call-improvements-in-net-framework-4.aspx`. 4.2

Bernard Paul Serpette and Manuel Serrano. Compiling Scheme to JVM bytecode: a performance study. *SIGPLAN Notices*, 37(9):259–270, 2002. ISSN 0362-1340. 2.3

Manuel Serrano and Marc Feeley. Storage use analysis and its applications. In *ICFP '96: Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, New York, NY, 1996. ACM. 2.3, 3.3.2

Manuel Serrano and Pierre Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In *Proceedings of the Static Analysis Symposium*, pages 366–381, 1995. 2.3

Olin Shivers. Control flow analysis in Scheme. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 164–174, New York, NY, 1988. ACM. 3.3.2

Jeremy Siek and Walid Taha. Gradual typing for objects. In *Proceedings of ECOOP 2007: European Conference on Object-Oriented Programming*, pages 2–27, 2007. 5

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 81–92, September 2006. 5

Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *DLS '08: Proceedings of the 2008 Symposium on Dynamic languages*, pages 1–12, New York, NY, 2008. ACM. 5

Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Richar Kelsey, Willian Clinger, Jonathan Rees, Robert Bruce Findler, and Jacob Matthews. Revised^6 report on the algorithmic language Scheme, 2007. Available at `http://www.r6rs.org`. 2.3

Guy L. Steele, Jr. Rabbit: A compiler for Scheme. Technical report, MIT, Cambridge, MA, 1978. 1.1

Sun Microsystems. Supporting dynamically typed languages on the Java platform, 2008. JSR 292, available at `http://jcp.org/en/jsr/detail?id=292`. 1

Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. *SIGPLAN Notices*, 43(1):395–406, 2008. ISSN 0362-1340. 5

Alex Turner and Bill Chiles. Sites, binders, and dynamic object interop spec, 2009. Available at `http://dlr.codeplex.com/Project/Download/FileDownload.aspx?DownloadId=68830`. 2.3

James Vastbinder. A .NET Triumvirate: IronScheme, IronLisp, and Xacc, 2008. Available at `http://www.infoq.com/news/2008/01/leppie-ironscheme`. 2.3

Andrew K. Wright and Robert Cartwright. A practical soft type system for scheme. *ACM Transasctions on Programming Languagens and Systems (TOPLAS)*, 19(1):87–152, 1997. ISSN 0164-0925. 3.1, 3.3.1

Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20(1):166–207, 1998. ISSN 0164-0925. 3.3.2

Dachuan Yu, Andrew Kennedy, and Don Syme. Formalization of generics for the .NET Common Language Runtime. *SIGPLAN Notices*, 39(1):39–51, 2004. ISSN 0362-1340. 1.2

A
Operational Semantics

This appendix presents a big-step operational semantics for the simplified Lua language in Figure 3.3 as a series of inference rules for relations $\xrightarrow{s}$ (statements and expression lists), $\xrightarrow{l}$ (lvalues), and $\xrightarrow{e}$ (expressions). Each relation is from a memory $\mathcal{M}$, an environment $\mathcal{E}$ and a term a to another memory $\mathcal{M}'$ and a value v that depends on the relation.

A memory $\mathcal{M}$ is a function $\mathbf{Loc} \rightarrow \mathbf{Value} \cup \{\bot\}$. $\mathbf{Loc}$ is the set of locations. $\mathbf{Value}$ is the set of Lua values and also the set of output values of the relation $\xrightarrow{e}$. Lua values are numbers, strings, **nil**, **true**, **false**, closures, which are a pair of environment $\mathcal{E}$ and function term f, and tables, which are locations pointing to functions $\mathbf{Value} \rightarrow \mathbf{Loc} \cup \{\bot\}$.

The set of output values for relation $\xrightarrow{l}$ is $\mathbf{Loc}$, and for relation $\xrightarrow{s}$ it is the set of value tuples $\mathbf{Value}^*$.

An environment $\mathcal{E}$ is a function $\mathbf{Var} \rightarrow \mathbf{Loc}$ that maps variables to locations in memory.

The functions index, app, arith and less are primitives that model part of Lua's extensible semantics. The functions index, arith and less take a memory and two values and return a memory and a value, and app takes a memory, a value and a tuple and returns a memory and another tuple. The output memory of these functions is the same as the input memory for all locations that are $\bot$ in the input memory and are not part of any of the input values.

A.1
Semantic Rules

Rule SKIP:

$$\mathcal{M}, \mathcal{E}, \mathbf{skip} \xrightarrow{s} \mathcal{M}, \bot$$

Rule SEQ-RETURN:

$$\frac{\mathcal{M}, \mathcal{E}, s_1 \xrightarrow{s} \mathcal{M}', v \quad v \neq \bot}{\mathcal{M}, \mathcal{E}, s_1; s_2 \xrightarrow{s} \mathcal{M}', v}$$

Rule SEQ:

$$\frac{\mathcal{M}, \mathcal{E}, s_1 \xrightarrow{s} \mathcal{M}', \perp \quad \mathcal{M}', \mathcal{E}, s_2 \xrightarrow{s} \mathcal{M}'', v}{\mathcal{M}', \mathcal{E}, s_1; s_2 \xrightarrow{s} \mathcal{M}'', v}$$

Rule RETURN:

$$\frac{\mathcal{M}, \mathcal{E}, el \xrightarrow{s} \mathcal{M}', v}{\mathcal{M}, \mathcal{E}, \textbf{return } el \xrightarrow{s} \mathcal{M}', v}$$

Rule IF-FALSE:

$$\frac{\mathcal{M}, \mathcal{E}, e \xrightarrow{e} \mathcal{M}', \textbf{false} \quad \mathcal{M}', \mathcal{E}, s_2 \xrightarrow{s} \mathcal{M}'', v}{\mathcal{M}, \mathcal{E}, \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \xrightarrow{s} \mathcal{M}'', v}$$

Rule IF-NIL:

$$\frac{\mathcal{M}, \mathcal{E}, e \xrightarrow{e} \mathcal{M}', \textbf{nil} \quad \mathcal{M}', \mathcal{E}, s_2 \xrightarrow{s} \mathcal{M}'', v}{\mathcal{M}, \mathcal{E}, \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \xrightarrow{s} \mathcal{M}'', v}$$

Rule IF-TRUE:

$$\frac{\mathcal{M}, \mathcal{E}, e \xrightarrow{e} \mathcal{M}', v \quad \mathcal{M}', \mathcal{E}, s_1 \xrightarrow{s} \mathcal{M}'', u \quad v \neq \textbf{false} \quad v \neq \textbf{nil}}{\mathcal{M}, \mathcal{E}, \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \xrightarrow{s} \mathcal{M}'', u}$$

Rule WHILE-FALSE:

$$\frac{\mathcal{M}, \mathcal{E}, e \xrightarrow{e} \mathcal{M}', \textbf{false}}{\mathcal{M}, \mathcal{E}, \textbf{while } e \textbf{ do } s \xrightarrow{s} \mathcal{M}', \perp}$$

Rule WHILE-FALSE:

$$\frac{\mathcal{M}, \mathcal{E}, e \xrightarrow{e} \mathcal{M}', \textbf{nil}}{\mathcal{M}, \mathcal{E}, \textbf{while } e \textbf{ do } s \xrightarrow{s} \mathcal{M}', \perp}$$

Rule WHILE-RETURN:

$$\frac{\mathcal{M}, \mathcal{E}, e \xrightarrow{e} \mathcal{M}', v \quad \mathcal{M}', \mathcal{E}, s \xrightarrow{s} \mathcal{M}'', u \quad v \neq \textbf{false} \quad v \neq \textbf{nil} \quad u \neq \perp}{\mathcal{M}, \mathcal{E}, \textbf{while } e \textbf{ do } s \xrightarrow{s} \mathcal{M}'', u}$$

Rule WHILE-TRUE:

$$\frac{\begin{array}{c}\mathcal{M}, \mathcal{E}, e \xrightarrow{e} \mathcal{M}', v \quad \mathcal{M}', \mathcal{E}, s \xrightarrow{s} \mathcal{M}'', \perp \quad \mathcal{M}'', \mathcal{E}, \textbf{while } e \textbf{ do } s \xrightarrow{s} \mathcal{M}''', u \\ v \neq \textbf{false} \quad v \neq \textbf{nil}\end{array}}{\mathcal{M}, \mathcal{E}, \textbf{while } e \textbf{ do } s \xrightarrow{s} \mathcal{M}''', u}$$

Rule LOCAL-1:

$$\frac{\mathcal{M}, \mathcal{E}, el \xrightarrow{s} \mathcal{M}', \perp \quad \mathcal{M}'[\vec{m} \mapsto \vec{\textbf{nil}}], \mathcal{E}[\vec{x} \mapsto \vec{m}], s \xrightarrow{s} \mathcal{M}'', v \quad \mathcal{M}'(m_k) = \perp}{\mathcal{M}, \mathcal{E}, \textbf{local } \vec{x} = el \textbf{ in } s \xrightarrow{s} \mathcal{M}'', v}$$

Rule LOCAL-2:

$$\frac{\mathcal{M}, \mathcal{E}, el \xrightarrow{s} \mathcal{M}', \vec{v} \quad \mathcal{M}'[\vec{m} \mapsto \vec{v_a}], \mathcal{E}[\vec{x} \mapsto \vec{m}], s \xrightarrow{s} \mathcal{M}'', u \quad \mathcal{M}'(m_k) = \bot}{\mathcal{M}, \mathcal{E}, \textbf{local}\ \vec{x} = el\ \textbf{in}\ s \xrightarrow{s} \mathcal{M}'', v}$$

where $v_{a_k} = v_k$ if $k \leq |\vec{v}|$ and $v_{a_k} = \textbf{nil}$ otherwise and $|\vec{m}| = |\vec{x}|$.

Rule ASSIGN-1:

$$\frac{\mathcal{M}, \mathcal{E}, l_1 \xrightarrow{l} \mathcal{M}_1, m_1 \ldots \mathcal{M}_{k-1}, \mathcal{E}, l_k \xrightarrow{l} \mathcal{M}', m_k \quad \mathcal{M}', \mathcal{E}, el \xrightarrow{s} \mathcal{M}'', \bot}{\mathcal{M}, \mathcal{E}, \vec{l} = el \xrightarrow{s} \mathcal{M}''[\vec{m} \mapsto \vec{\textbf{nil}}], \bot}$$

Rule ASSIGN-2:

$$\frac{\mathcal{M}, \mathcal{E}, l_1 \xrightarrow{l} \mathcal{M}_1, m_1 \ldots \mathcal{M}_{k-1}, \mathcal{E}, l_k \xrightarrow{l} \mathcal{M}', m_k \quad \mathcal{M}', \mathcal{E}, el \xrightarrow{s} \mathcal{M}'', \vec{v}}{\mathcal{M}, \mathcal{E}, \vec{l} = el \xrightarrow{s} \mathcal{M}''[\vec{m} \mapsto \vec{v_a}], \bot}$$

where $v_{a_k} = v_k$ if $k \leq |\vec{v}|$ and $v_{a_k} = \textbf{nil}$ otherwise and $|\vec{m}| = |\vec{l}|$.

Rule APP-STAT:

$$\frac{\mathcal{M}, \mathcal{E}, e(el)_1 \xrightarrow{e} \mathcal{M}', v}{\mathcal{M}, \mathcal{E}, e(el)_0 \xrightarrow{s} \mathcal{M}', \bot}$$

Rule VAR-LVAL:

$$\mathcal{M}, \mathcal{E}, x \xrightarrow{s} \mathcal{E}(x)$$

Rule VAR-RVAL:

$$\mathcal{M}, \mathcal{E}, x \xrightarrow{e} \mathcal{M}(\mathcal{E}(x))$$

Rule CONS:

$$\mathcal{M}, \mathcal{E}, \{\} \xrightarrow{e} \mathcal{M}[t \mapsto \lambda x.\bot], t \quad \mathcal{M}(t) = \bot$$

Rule TAB-LVAL:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}', t \quad \mathcal{M}', \mathcal{E}, e_2 \xrightarrow{e} \mathcal{M}'', v \quad t \in \textbf{Table} \quad \mathcal{M}''(t)(v) \neq \bot}{\mathcal{M}, \mathcal{E}, e_1[e_2] \xrightarrow{s} \mathcal{M}'', \mathcal{M}''(t)(v)}$$

Rule TAB-LVAL-NEW:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}', t \quad \mathcal{M}', \mathcal{E}, e_2 \xrightarrow{e} \mathcal{M}'', v \quad t \in \textbf{Table} \quad \mathcal{M}''(t)(v) = \bot}{\mathcal{M}, \mathcal{E}, e_1[e_2] \xrightarrow{s} \mathcal{M}''[t \mapsto \mathcal{M}''(t)[v \mapsto l], l \quad \mathcal{M}''(l) = \bot}$$

Rule TAB-RVAL:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}', t \quad \mathcal{M}', \mathcal{E}, e_2 \xrightarrow{e} \mathcal{M}'', v \quad t \in \textbf{Table} \quad \mathcal{M}''(t)(v) \neq \bot}{\mathcal{M}, \mathcal{E}, e_1[e_2] \xrightarrow{s} \mathcal{M}'', \mathcal{M}''(\mathcal{M}''(t)(v))}$$

Rule TAB-RVAL-NIL:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}', t \quad \mathcal{M}', \mathcal{E}, e_2 \xrightarrow{e} \mathcal{M}'', v \quad t \in \mathbf{Table} \quad \mathcal{M}''(t)(v) = \perp}{\mathcal{M}, \mathcal{E}, e_1[e_2] \xrightarrow{s} \mathcal{M}'', \mathbf{nil}}$$

Rule TAB-META:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}', v_1 \quad \mathcal{M}', \mathcal{E}, e_2 \xrightarrow{e} \mathcal{M}'', v_2 \quad v_1 \notin \mathbf{Table}}{\mathcal{M}, \mathcal{E}, e_1[e_2] \xrightarrow{e} \text{index}(\mathcal{M}'', v_1, v_2)}$$

Rule EL-EMPTY:

$$\mathcal{M}, \mathcal{E}, \mathbf{nothing} \xrightarrow{s} \mathcal{M}, \perp$$

Rule EL:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}_1, v_1 \ldots \mathcal{M}_{k-1}, \mathcal{E}, e_k \xrightarrow{e} \mathcal{M}', v_k \quad k = |\vec{v}|}{\mathcal{M}, \mathcal{E}, \vec{e} \xrightarrow{s} \mathcal{M}', \vec{v}}$$

Rule EL-MEXP:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}_1, v_1 \ldots \mathcal{M}_{k-1}, \mathcal{E}, e_k \xrightarrow{e} \mathcal{M}', v_k \quad \mathcal{M}', \mathcal{E}, me \xrightarrow{s} \mathcal{M}'', u \quad k = |\vec{v}|}{\mathcal{M}, \mathcal{E}, \vec{e}, me \xrightarrow{s} \mathcal{M}'', \vec{v}u}$$

Rule APP-CLOSURE:

$$\frac{\begin{array}{c}\mathcal{M}, \mathcal{E}, e \xrightarrow{e} \mathcal{M}', \langle \mathcal{E}', \mathbf{fun}(\vec{x})\ b \rangle \quad \mathcal{M}', \mathcal{E}, el \xrightarrow{s} \mathcal{M}'', \vec{v} \\ \mathcal{M}''[\vec{m} \mapsto \vec{v_a}], \mathcal{E}'[\vec{x} \mapsto \vec{m}], b \xrightarrow{s} \mathcal{M}''', u \quad \mathcal{M}''(m_k) = \perp\end{array}}{\mathcal{M}, \mathcal{E}, e(el)_n \xrightarrow{s} \mathcal{M}''', u}$$

where $v_{a_k} = v_k$ if $k \leq |\vec{v}|$ and $v_{a_k} = \mathbf{nil}$ otherwise and $|\vec{m}| = |\vec{x}|$.

Rule APP-META:

$$\frac{\mathcal{M}, \mathcal{E}, e \xrightarrow{e} \mathcal{M}', v_1 \quad \mathcal{M}', \mathcal{E}, el \xrightarrow{s} \mathcal{M}'', v_2 \quad v_1 \notin \mathbf{Closure}}{\mathcal{M}, \mathcal{E}, e(el)_n \xrightarrow{s} \text{app}(\mathcal{M}'', v_1, v_2)}$$

Rule APP-FIRST-NIL:

$$\frac{\mathcal{M}, \mathcal{E}, e(el)_n \xrightarrow{s} \mathcal{M}', \perp}{\mathcal{M}, \mathcal{E}, e(el)_1 \xrightarrow{e} \mathcal{M}', \mathbf{nil}}$$

Rule APP-FIRST:

$$\frac{\mathcal{M}, \mathcal{E}, e(el)_n \xrightarrow{s} \mathcal{M}', \vec{v}}{\mathcal{M}, \mathcal{E}, e(el)_1 \xrightarrow{e} \mathcal{M}', v_1}$$

Rule ARITH:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}', v_1 \quad \mathcal{M}', \mathcal{E}, e_2 \xrightarrow{e} \mathcal{M}'', v_2 \quad v_1 \in \textbf{Number} \quad v_2 \in \textbf{Number}}{\mathcal{M}, \mathcal{E}, e_1 \oplus e_2 \xrightarrow{e} \mathcal{M}'', \mathcal{E}, v1 \oplus v_2}$$

Rule ARITH-META:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}', v_1 \quad \mathcal{M}', \mathcal{E}, e_2 \xrightarrow{e} \mathcal{M}'', v_2 \quad v_1 \notin \textbf{Number} \vee v_2 \notin \textbf{Number}}{\mathcal{M}, \mathcal{E}, e_1 \oplus e_2 \xrightarrow{e} \text{arith}(\mathcal{M}'', v1, v_2)}$$

Rule EQ-TRUE:

$$\frac{\mathcal{M}, env, e_1 \xrightarrow{e} \mathcal{M}', v_1 \quad \mathcal{M}', \mathcal{E}, e_2 \xrightarrow{e} v_2 \quad v_1 = v_2}{\mathcal{M}, \mathcal{E}, e_1 == e_2 \xrightarrow{e} \mathcal{M}'', \textbf{true}}$$

Rule EQ-FALSE:

$$\frac{\mathcal{M}, env, e_1 \xrightarrow{e} \mathcal{M}', v_1 \quad \mathcal{M}', \mathcal{E}, e_2 \xrightarrow{e} v_2 \quad v_1 \neq v_2}{\mathcal{M}, \mathcal{E}, e_1 == e_2 \xrightarrow{e} \mathcal{M}'', \textbf{false}}$$

Rule LESS-TRUE:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}', v_1 \quad \mathcal{M}', \mathcal{E}, e_2 \xrightarrow{e} \mathcal{M}'', v_2 \quad v_1 \in \textbf{Number} \quad v_2 \in \textbf{Number} \quad v_1 < v_2}{\mathcal{M}, \mathcal{E}, e_1 \oplus e_2 \xrightarrow{e} \mathcal{M}'', \mathcal{E}, \textbf{true}}$$

Rule LESS-FALSE:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}', v_1 \quad \mathcal{M}', \mathcal{E}, e_2 \xrightarrow{e} \mathcal{M}'', v_2 \quad v_1 \in \textbf{Number} \quad v_2 \in \textbf{Number} \quad v_1 \not< v_2}{\mathcal{M}, \mathcal{E}, e_1 \oplus e_2 \xrightarrow{e} \mathcal{M}'', \mathcal{E}, \textbf{false}}$$

Rule LESS-META:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}', v_1 \quad \mathcal{M}', \mathcal{E}, e_2 \xrightarrow{e} \mathcal{M}'', v_2 \quad v_1 \notin \textbf{Number} \vee v_2 \notin \textbf{Number}}{\mathcal{M}, \mathcal{E}, e_1 \oplus e_2 \xrightarrow{e} \text{less}(\mathcal{M}'', \mathcal{E}, v1 \oplus v_2)}$$

Rule AND-NIL:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}', v_1 \quad v_1 = \textbf{nil}}{\mathcal{M}, \mathcal{E}, e_1 \textbf{ and } e_2 \xrightarrow{e} \mathcal{M}', \textbf{nil}}$$

Rule AND-FALSE:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}', v_1 \quad v_1 = \textbf{false}}{\mathcal{M}, \mathcal{E}, e_1 \textbf{ and } e_2 \xrightarrow{e} \mathcal{M}', \textbf{false}}$$

Rule AND:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}', v_1 \quad \mathcal{M}', \mathcal{E}, e_2 \xrightarrow{e} \mathcal{M}'', v_2 \quad v_1 \neq \textbf{false} \quad v_1 \neq \textbf{nil}}{\mathcal{M}, \mathcal{E}, e_1 \textbf{ and } e_2 \xrightarrow{e} \mathcal{M}'', v2}$$

Rule OR-NIL:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}', \textbf{nil} \quad \mathcal{M}', \mathcal{E}, e_2 \xrightarrow{e} \mathcal{M}'', v_2}{\mathcal{M}, \mathcal{E}, e_1 \textbf{ or } e_2 \xrightarrow{e} \mathcal{M}'', v2}$$

Rule OR-FALSE:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}', \textbf{false} \quad \mathcal{M}', \mathcal{E}, e_2 \xrightarrow{e} \mathcal{M}'', v_2}{\mathcal{M}, \mathcal{E}, e_1 \textbf{ or } e_2 \xrightarrow{e} \mathcal{M}'', v2}$$

Rule OR:

$$\frac{\mathcal{M}, \mathcal{E}, e_1 \xrightarrow{e} \mathcal{M}', v_1 \quad v_1 \neq \textbf{false} \quad v_1 \neq \textbf{nil}}{\mathcal{M}, \mathcal{E}, e_1 \textbf{ or } e_2 \xrightarrow{e} \mathcal{M}', v1}$$

Rule NOT-NIL:

$$\frac{\mathcal{M}, \mathcal{E}, e \xrightarrow{e} \mathcal{M}', \textbf{nil}}{\mathcal{M}, \mathcal{E}, \textbf{not } e \xrightarrow{e} \mathcal{M}', \textbf{true}}$$

Rule NOT-FALSE:

$$\frac{\mathcal{M}, \mathcal{E}, e \xrightarrow{e} \mathcal{M}', \textbf{false}}{\mathcal{M}, \mathcal{E}, \textbf{not } e \xrightarrow{e} \mathcal{M}', \textbf{true}}$$

Rule NOT-TRUE:

$$\frac{\mathcal{M}, \mathcal{E}, e \xrightarrow{e} \mathcal{M}', v \quad v \neq \textbf{false} \quad v \neq \textbf{nil}}{\mathcal{M}, \mathcal{E}, \textbf{not } e \xrightarrow{e} \mathcal{M}', \textbf{false}}$$

Rule FUN:

$$\mathcal{M}, \mathcal{E}, \textbf{fun}(\vec{x})\ b \xrightarrow{e} \mathcal{M}, \langle \mathcal{E}, \textbf{fun}(\vec{x})\ b \rangle$$

B
Typing Rules

This appendix presents the full set of typing rules, including those already mentioned in Section 3.1.4.

Rule SKIP:

$$\Gamma \vdash \textbf{skip} \colon \textbf{void}$$

Rule SEQ-VOID:

$$\frac{\Gamma \vdash s_1 \colon \textbf{void} \quad \Gamma \vdash s_2 \colon \textbf{void}}{\Gamma \vdash s_1; s_2 \colon \textbf{void}}$$

Rule SEQ-1:

$$\frac{\Gamma \vdash s_1 \colon \tau \quad \Gamma \vdash s_2 \colon \textbf{void}}{\Gamma \vdash s_1; s_2 \colon \tau}$$

Rule SEQ-2:

$$\frac{\Gamma \vdash s_1 \colon \textbf{void} \quad \Gamma \vdash s_2 \colon \tau}{\Gamma \vdash s_1; s_2 \colon \tau}$$

Rule SEQ-BOTH:

$$\frac{\Gamma \vdash s_1 \colon \tau_1 \quad \Gamma \vdash s_2 \colon \tau_2 \quad \tau_1 \rightsquigarrow \upsilon \quad \tau_2 \rightsquigarrow \upsilon}{\Gamma \vdash s_1; s_2 \colon \upsilon}$$

Rule IF-VOID:

$$\frac{\Gamma \vdash e \colon \tau_e \quad \Gamma \vdash s_1 \colon \textbf{void} \quad \Gamma \vdash s_2 \colon \textbf{void}}{\Gamma \vdash \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \colon \textbf{void}}$$

Rule IF-1:

$$\frac{\Gamma \vdash e \colon \tau_e \quad \Gamma \vdash s_1 \colon \tau \quad \Gamma \vdash s_2 \colon \textbf{void}}{\Gamma \vdash \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \colon \tau}$$

Rule IF-2:

$$\frac{\Gamma \vdash e \colon \tau_e \quad \Gamma \vdash s_1 \colon \textbf{void} \quad \Gamma \vdash s_2 \colon \tau}{\Gamma \vdash \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \colon \tau}$$

Rule IF-BOTH:

$$\frac{\Gamma \vdash e \colon \tau_e \quad \Gamma \vdash s_1 \colon \tau_1 \quad \Gamma \vdash s_2 \colon \tau_2 \quad \tau_1 \rightsquigarrow \upsilon \quad \tau_2 \rightsquigarrow \upsilon}{\Gamma \vdash \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \colon \upsilon}$$

Rule WHILE-VOID:

$$\frac{\Gamma \vdash e\colon \tau \quad \Gamma \vdash s\colon \mathbf{void}}{\Gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ s\colon \mathbf{void}}$$

Rule WHILE:

$$\frac{\Gamma \vdash e\colon \tau \quad \Gamma \vdash s\colon \upsilon}{\Gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ s\colon \upsilon}$$

Rule RETURN:

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

Rule LOCAL-DROP-VOID:

$$\frac{\Gamma \vdash el : \upsilon_1 \times \ldots \times \upsilon_m \quad \Gamma[\vec{x} \mapsto \vec{\tau}] \vdash s : \mathbf{void} \quad m \geq |\vec{x}| \quad \upsilon_k \rightsquigarrow \tau_k}{\Gamma \vdash \mathbf{local}\ \vec{x} = el\ \mathbf{in}\ s\colon \mathbf{void}}$$

Rule LOCAL-DROP:

$$\frac{\Gamma \vdash el : \upsilon_1 \times \ldots \times \upsilon_m \quad \Gamma[\vec{x} \mapsto \vec{\tau}] \vdash s : \omega \quad m \geq |\vec{x}| \quad \upsilon_k \rightsquigarrow \tau_k}{\Gamma \vdash \mathbf{local}\ \vec{x} = el\ \mathbf{in}\ s\colon \omega}$$

Rule LOCAL-FILL-VOID:

$$\frac{\Gamma \vdash el : \upsilon_1 \times \ldots \times \upsilon_m \quad \Gamma[\vec{x} \mapsto \vec{\tau}] \vdash s : \mathbf{void} \quad m < |\vec{x}| \quad \upsilon_k \rightsquigarrow \tau_k \quad \mathbf{nil} \rightsquigarrow \tau_l \quad l > m}{\Gamma \vdash \mathbf{local}\ \vec{x} = el\ \mathbf{in}\ s\colon \mathbf{void}}$$

Rule LOCAL-FILL:

$$\frac{\Gamma \vdash el : \upsilon_1 \times \ldots \times \upsilon_m \quad \Gamma[\vec{x} \mapsto \vec{\tau}] \vdash s : \omega \quad m < |\vec{x}| \quad \upsilon_k \rightsquigarrow \tau_k \quad \mathbf{nil} \rightsquigarrow \tau_l \quad l > m}{\Gamma \vdash \mathbf{local}\ \vec{x} = el\ \mathbf{in}\ s\colon \omega}$$

Rule LOCAL-VAR-DROP-VOID:

$$\frac{\Gamma \vdash el : \upsilon_1 \times \ldots \times \upsilon_m \times \mathcal{D}^* \quad \Gamma[\vec{x} \mapsto \vec{\tau}] \vdash s : \mathbf{void} \quad m \geq |\vec{x}| \quad \upsilon_k \rightsquigarrow \tau_k}{\Gamma \mathbf{local}\ \vec{x} = el\ \mathbf{in}\ s\colon \mathbf{void}}$$

Rule LOCAL-VAR-DROP:

$$\frac{\Gamma \vdash el : \upsilon_1 \times \ldots \times \upsilon_m \times \mathcal{D}^* \quad \Gamma[\vec{x} \mapsto \vec{\tau}] \vdash s : \omega \quad m \geq |\vec{x}| \quad \upsilon_k \rightsquigarrow \tau_k}{\Gamma \mathbf{local}\ \vec{x} = el\ \mathbf{in}\ s\colon \omega}$$

Rule LOCAL-VAR-FILL-VOID:

$$\frac{\begin{array}{c}\Gamma \vdash el : \upsilon_1 \times \ldots \times \upsilon_m \times \mathcal{D}^* \quad \Gamma[\vec{x} \mapsto \vec{\tau}] \vdash s : \mathbf{void} \quad m < |\vec{x}| \\ \upsilon_k \rightsquigarrow \tau_k \quad \tau_l = \mathcal{D} \quad l > m\end{array}}{\Gamma \vdash \mathbf{local}\ \vec{x} = el\ \mathbf{in}\ s\colon \mathbf{void}}$$

Rule LOCAL-VAR-FILL:

$$\frac{\Gamma \vdash el : \upsilon_1 \times \ldots \times \upsilon_m \times \mathcal{D}^* \quad \Gamma[\vec{x} \mapsto \vec{\tau}] \vdash s : \omega \quad m < |\vec{x}| \\ \upsilon_k \leadsto \tau_k \quad \tau_l = \mathcal{D} \quad l > m}{\Gamma \vdash \textbf{local}\ \vec{x} = el\ \textbf{in}\ s : \omega}$$

Rule ASSIGN-DROP:

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

Rule ASSIGN-FILL:

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

Rule ASSIGN-VAR-DROP:

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

Rule ASSIGN-VAR-FILL:

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

Rule EL-EMPTY:

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

Rule EL:

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

Rule EL-MEXP-EMPTY:

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

Rule EL-MEXP:

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

Rule EL-VAR-1:

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

Rule EL-VAR-2:

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

Rule APP-DROP:

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

Rule APP-FILL:

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

Rule APP-VAR-DROP:

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

Rule APP-VAR-FILL:

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

Rule APP-DYN:

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

Rule APP-STAT:

$$\frac{\Gamma \vdash e(el)_n\colon \tau}{\Gamma \vdash e(el)_0\colon \mathbf{void}}$$

Rule APP-FIRST:

$$\frac{\Gamma \vdash e(el)_n\colon \tau_1 \times \ldots \times \tau_n}{\Gamma \vdash e(el)_1\colon \tau_1}$$

Rule APP-FIRST-NIL:

$$\frac{\Gamma \vdash e(el)_n\colon \mathbf{empty}}{\Gamma \vdash e(el)_1\colon \mathbf{nil}}$$

Rule APP-FIRST-DYN:

$$\frac{\Gamma \vdash e(el)_n\colon \mathcal{D}^*}{\Gamma \vdash e(el)_1\colon \mathcal{D}}$$

Rule CONS:

$$\frac{\forall i, j, \sigma.(i \neq j \land \sigma \leadsto \tau_i) \to \sigma \not\leadsto \tau_j \quad \mathbf{nil} \leadsto \upsilon_k}{\Gamma \vdash \{\} : \tau_1 \mapsto \upsilon_1 \land \ldots \land \tau_n \mapsto \upsilon_n}$$

Rule CONS-DYN:

$$\frac{\mathbf{nil} \leadsto \upsilon}{\Gamma \vdash \{\} : \mathcal{D} \mapsto \upsilon}$$

Rule INDEX:

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

Rule INDEX-DYN:

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

Rule ARITH-NUM:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \leadsto \mathbf{Number} \quad \tau_2 \leadsto \mathbf{Number}}{\Gamma \vdash e_1 \oplus e_2 : \mathbf{Number}}$$

Rule ARITH-DYN:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \not\leadsto \mathbf{Number} \lor \tau_2 \not\leadsto \mathbf{Number} \quad \tau_1 \leadsto \mathcal{D} \quad \tau_2 \leadsto \mathcal{D}}{\Gamma \vdash e_1 \oplus e_2 : \mathcal{D}}$$

Rule EQ:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \leadsto \tau_2 \lor \tau_2 \leadsto \tau_1}{\Gamma \vdash e_1 == e_2 : \mathbf{Bool}}$$

Rule LESS-NUM:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \leadsto \mathbf{Number} \quad \tau_2 \leadsto \mathbf{Number}}{\Gamma \vdash e_1 < e_2 : \mathbf{Bool}}$$

Rule LESS-DYN:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \not\leadsto \mathbf{Number} \lor \tau_2 \not\leadsto \mathbf{Number} \quad \tau_1 \leadsto \mathcal{D} \quad \tau_2 \leadsto \mathcal{D}}{\Gamma \vdash e_1 < e_2 : \mathcal{D}}$$

Rule AND-NIL:

$$\frac{\Gamma \vdash e_1 : \mathbf{nil}}{\Gamma \vdash e_1 \ \mathbf{and} \ e_2 : \mathbf{nil}}$$

Rule AND-FALSE:

$$\frac{\Gamma \vdash e_1 : \mathbf{false}}{\Gamma \vdash e_1 \ \mathbf{and} \ e_2 : \mathbf{false}}$$

Rule AND-TRUE:

$$\frac{\Gamma \vdash e_1 \colon \tau_1 \quad \Gamma \vdash e_1 \colon \tau_2 \quad \tau_1 \neq \textbf{false} \quad \textbf{nil} \not\leadsto \tau_1}{\Gamma \vdash e_1 \ \textbf{and} \ e_2 \colon \tau_2}$$

Rule AND:

$$\frac{\Gamma \vdash e_1 \colon \tau_1 \quad \Gamma \vdash e_1 \colon \tau_2 \quad \tau_1 \neq \textbf{nil} \quad \textbf{nil} \leadsto \tau_1 \quad \tau_1 \leadsto \upsilon \quad \tau_2 \leadsto \upsilon}{\Gamma \vdash e_1 \ \textbf{and} \ e_2 \colon \upsilon}$$

Rule OR-NIL:

$$\frac{\Gamma \vdash e_1 \colon \textbf{nil} \quad \Gamma \vdash e_2 \colon \tau}{\Gamma \vdash e_1 \ \textbf{or} \ e_2 \colon \tau}$$

Rule OR-FALSE:

$$\frac{\Gamma \vdash e_1 \colon \textbf{false} \quad \Gamma \vdash e_2 \colon \tau}{\Gamma \vdash e_1 \ \textbf{or} \ e_2 \colon \tau}$$

Rule OR-TRUE:

$$\frac{\Gamma \vdash e_1 \colon \tau \quad \tau \neq \textbf{false} \quad \textbf{nil} \not\leadsto \tau}{\Gamma \vdash e_1 \ \textbf{or} \ e_2 \colon \tau}$$

Rule OR:

$$\frac{\Gamma \vdash e_1 \colon \tau_1 \quad \Gamma \vdash e_2 \colon \tau_2 \quad \tau_1 \neq \textbf{nil} \quad \textbf{nil} \leadsto \tau_1 \quad \tau_1 \leadsto \upsilon \quad \tau_2 \leadsto \upsilon}{\Gamma \vdash e_1 \ \textbf{or} \ e_2 \colon \upsilon}$$

Rule NOT-NIL:

$$\frac{\Gamma \vdash e \colon \textbf{nil}}{\Gamma \vdash \textbf{not} \ e \colon \textbf{true}}$$

Rule NOT-FALSE:

$$\frac{\Gamma \vdash e \colon \textbf{false}}{\Gamma \vdash \textbf{not} \ e \colon \textbf{true}}$$

Rule NOT-TRUE:

$$\frac{\Gamma \vdash e \colon t \quad \tau \neq \textbf{false} \quad \textbf{nil} \not\leadsto \tau}{\Gamma \vdash \textbf{not} \ e \colon \textbf{false}}$$

Rule NOT:

$$\frac{\Gamma \vdash e \colon t \quad \tau \neq \textbf{nil} \quad \textbf{nil} \leadsto \tau}{\Gamma \vdash \textbf{not} \ e \colon \textbf{Bool}}$$

Rule FUN-EMPTY:

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

Rule FUN:

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

C
Collected Benchmark Results

	base	single	box	intern	prop	infer
binary-trees	2.140	1.987	2.256	1.143	0.857	0.193
fannkuch	1.825	1.835	3.315	3.397	1.684	0.829
fib-iter	0.567	0.504	1.780	1.824	0.130	0.062
fib-memo	0.468	0.230	0.318	0.340	0.351	0.147
fib-rec	1.806	0.540	1.311	1.297	1.304	0.773
mandelbrot	1.632	1.588	5.608	5.551	0.146	0.145
n-body	2.656	2.516	4.169	3.123	3.084	0.853
n-sieve	1.901	1.887	3.115	3.138	1.352	0.296
nsieve-bits	0.883	0.567	1.377	1.403	1.028	0.574
partial-sum	0.647	0.493	1.152	1.184	0.680	0.407
recursive	1.639	0.450	1.418	1.425	1.488	0.110
spectral-norm	1.894	1.186	3.212	3.245	3.175	0.210
richards	0.343	0.308	0.329	0.217	0.202	0.047
richards-tail	N/A	N/A	N/A	N/A	N/A	N/A
richards-oo	0.383	0.304	0.327	0.227	0.197	0.052
richards-oo-tail	N/A	N/A	N/A	N/A	N/A	N/A
richards-oo-meta	0.691	0.634	0.463	0.315	0.274	0.242
richards-oo-cache	0.493	0.352	0.370	0.241	0.216	0.186

Table C.1: Benchmark running times for Mono 2.4, in seconds

	base	single	box	intern	prop	infer
binary-trees	0.686	0.671	0.686	0.269	0.218	0.031
fannkuch	1.373	1.392	1.267	1.312	0.895	0.696
fib-iter	0.189	0.189	0.236	0.232	0.037	0.029
fib-memo	0.183	0.156	0.154	0.176	0.166	0.023
fib-rec	0.920	0.644	0.287	0.265	0.265	0.131
mandelbrot	1.102	1.115	0.858	0.985	0.037	0.041
n-body	1.952	1.798	1.817	1.277	1.197	0.191
n-sieve	1.336	1.326	1.244	1.258	0.996	0.205
n-sieve-bits	0.365	0.343	0.359	0.363	0.296	0.199
partial-sum	0.314	0.291	0.283	0.269	0.168	0.158
recursive	0.712	0.511	0.253	0.242	0.222	0.047
spectral-norm	1.178	1.125	0.840	0.850	0.825	0.162
richards	0.240	0.230	0.211	0.156	0.148	0.055
richards-tail	0.255	0.234	0.215	0.160	0.150	0.064
richards-oo	0.255	0.218	0.201	0.158	0.127	0.055
richards-oo-tail	N/A	N/A	1.094	0.827	0.628	0.060
richards-oo-meta	0.425	0.398	0.298	0.211	0.201	0.179
richards-oo-cache	0.281	0.252	0.240	0.162	0.156	0.152

Table C.2: Benchmark running times for .NET 3.5 SP1, in seconds

	base	single	box	intern	prop	infer
binary-trees	0.691	0.690	0.597	0.252	0.203	0.035
fannkuch	1.182	1.406	1.196	1.301	0.895	0.671
fib-iter	0.204	0.218	0.243	0.268	0.044	0.034
fib-memo	0.178	0.165	0.150	0.172	0.173	0.029
fib-rec	0.968	0.665	0.318	0.297	0.296	0.138
mandelbrot	1.093	1.120	1.007	0.998	0.042	0.045
n-body	1.747	1.986	1.658	1.303	1.190	0.183
n-sieve	1.437	1.450	1.292	1.259	0.983	0.206
nsieve-bits	0.346	0.356	0.354	0.381	0.318	0.193
partial-sum	0.331	0.304	0.281	0.283	0.180	0.161
recursive	0.735	0.525	0.277	0.266	0.258	0.052
spectral-norm	1.152	1.131	0.854	0.873	0.848	0.107
richards	0.280	0.259	0.217	0.159	0.152	0.050
richards-tail	0.278	0.261	0.214	0.162	0.156	0.063
richards-oo	0.279	0.263	0.201	0.158	0.140	0.054
richards-oo-tail	N/A	N/A	1.212	0.690	0.394	0.055
richards-oo-meta	0.470	0.460	0.291	0.207	0.202	0.190
richards-oo-cache	0.308	0.283	0.228	0.169	0.162	0.152

Table C.3: Benchmark running times for .NET 4.0 Beta 1, in seconds

	lua	luajit
binary-trees	0.259	0.184
fannkuch	1.228	0.707
fib-iter	0.499	0.083
fib-memo	0.264	0.109
fib-rec	0.835	0.170
mandelbrot	0.503	0.108
n-body	0.680	0.347
n-sieve	0.655	0.532
n-sieve-bits	0.318	0.134
partial-sum	0.282	0.136
recursive	0.704	0.128
spectra-lnorm	0.705	0.269
richards	0.133	0.054
richards-tail	0.137	0.059
richards-oo	0.116	0.052
richards-oo-tail	0.128	0.056
richards-oo-meta	0.140	0.068
richards-oo-cache	0.138	0.062

Table C.4: Benchmark running times for Lua 5.1.4 and LuaJIT 1.1.5, in seconds

binary-trees	0.137
fannkuch	1.983
fib-iter	0.553
fib-memo	0.102
fib-rec	0.706
mandelbrot	1.061
n-body	1.581
n-sieve	0.753
n-sieve-bits	0.626
partial-sum	0.402
recursive	0.476
spectral-norm	1.838
richards	0.676

Table C.5: Benchmark running times for IronPython 2.0, in seconds