Preserving Lexical Scoping When Dynamically Embedding Languages

Félix Ribeiro, Hisham Muhammad,
André Murbach Maidl, Roberto Ierusalimschy

Department of Computer Science – PUC-Rio – Rio de Janeiro – Brazil
{fribeiro,hisham,amaidl,roberto}@inf.puc-rio.br

Abstract. There are various situations in which one may want to embed source code from one language into another, for example when combining relational query languages with application code or when performing staged meta-programming. Typically, one will want to transfer data between these languages. We propose an approach in which the embedded code shares variables with the host language, preserving lexical scoping rules even after the code is converted into an intermediate representation. We demonstrate this approach through a module for meta-programming using Lua as both embedded and host languages, which decompiles Lua functions to their AST form and can later rebuild them preserving scoping rules of the decompilation site. Our method requires no special annotation of functions to be translated and is implemented as a library, requiring no source pre-processing or changes to the host language execution environment.

Keywords: Lua, Domain-Specific Languages, Embedded Languages

1 Introduction

Domain-Specific Languages (DSLs) are a way to simplify the development of programs through the aggregation of domain knowledge into a programming language. A Domain-Specific Language is a programming language that includes features to express the semantics of a domain, often adding specific syntax. Examples of DSLs are TEX for text processing, MATLAB for performing numerical computations, SQL for querying relational databases and regular expressions for pattern matching in text.

The use of DSLs frequently happens in combination with other languages, so that some aspects of a problem are handled with the DSL while other parts are developed in a general-purpose language [7]. One way to do this is to embed source code written in the domain-specific language into the source code of the application, which is written in another language. We have then the notion of a *host language* and an *embedded language*. SQL and regular expressions are examples of languages which are often used in this fashion.

Embedding source code of one language into another poses challenges. Typically, a language parser does not have support for handling chunks of code written in another language intermixed with the source code. Common approaches

to handle the source code of two languages in a single source file are to either pull the processing back to a step prior to the parsing of the main language, using pre-processing, or to push it forward by storing the code written in the embedded language as strings in the host language source code, which are sent to the embedded language for processing only at run time.

This approach of storing code as strings, while popular, has some inconveniences. For instance, it is not possible to detect syntactical errors while compiling the code. Embedding languages should also allow programmers to transfer data between these languages, taking care to keep data in sync. For these reasons, solutions based on meta-programming, where the embedded language can be manipulated at a higher level of abstraction than strings, are more interesting.

Multi-Stage Programming (MSP) [14,15,16] is a meta-programming approach that helps embedding a programming language in a host language in a well-organized way. It defines constructs for quoting and escaping source code that produce code objects, which are valid objects stored in the host language but can also be invoked to execute the embedded language. A major benefit of MSP is that it does not delay error verification to run-time. One can detect syntactical errors and even type errors in the embedded code during compile-time. Another benefit of MSP is that we can use program specialization to reduce the costs of abstractions [14].

Using MSP to embed languages inside imperative languages can be hard, because in these languages programmers can move code objects so they are used outside of the scope of the binder of their free variables [17]. In purely functional languages we do not have this problem due to the absence of side effects [9].

In this work, we propose an approach for meta-programming in which the embedded code shares variables with the host language, preserving lexical scoping rules even after the code is converted into an intermediate representation. In our proposed method, the host language uses closures to share data with the embedded language, replacing variable references with function calls in the generated code. This way, we ensure that variables always match the scope of their declarations.

We demonstrate this approach through a module for meta-programming using Lua as both embedded and host languages. Our module decompiles Lua functions to their Abstract Syntax Tree (AST) form and can later rebuild them preserving scoping rules of the decompilation site. For simplicity, our implementation only supports functions that contain a single expression. We call these functions *lambda functions*.

Our method requires no special annotation of functions to be translated and is implemented as a library, requiring no source pre-processing or changes to the host language execution environment. When an AST describes a function that uses variables from an external local scope, it includes information about the context where this function was defined.

We organize this paper in five sections. In Section 2 we review related work in the field of multi-stage programming. In Section 3 we demonstrate our approach.

In Section 4 we formalize the semantics of our approach. In Section 5 we present our conclusions.

2 Related Work

Meta-programming is the concept of writing programs that manipulate program code as data, producing other programs. This allows programmers to improve code performance or expressiveness by defining transformations over code. Lisp [11] pioneered meta-programming by introducing a mechanism of *quotation*: expressions marked with the operator `'` are not evaluated, and are treated as data. Later Lisp dialects like Common Lisp and Scheme include *quasi-quotation*, represented with the operator `` ` ``, that allows parts of the quoted expression to be "escaped" (with the `,` operator). The combination of quasi-quotation and escaping powers the macro system of those languages [1]. This feature, however, does not preserve scoping rules.

Multi-Stage Programming [14,15,16] is similar to the quasi-quotation mechanism, but it takes lexical scoping into account. It features three constructs that programmers can use to annotate code: *brackets*, *escape*, and *run*. We will use MetaOCaml [2], an OCaml extension with MSP support through these three staging constructs, to briefly explain these constructs. *Brackets*, marked with `.<>.`, avoid the execution of a computation, constructing an object instead that represents the marked block of code:

```
let x = 1 + 1;;
let y = .< 1 + 1 >.;;
```

In the above example, `x` has type `int` and `y` has type `int code`. This means that the expression `x + y` is invalid code in MetaOCaml, as the types of both variables do not match. *Escapes*, marked as `.~`, combine small delayed computations for building bigger ones:

```
let z = .< x + .~y >.;;
```

Here, the code `.< x + .~y >.` binds a new delayed computation `2 + (1 + 1)` to `z`. *Run*, using the prefix operator `.!`, executes staged code. In the example below, the program will compile and execute the code inside `z`, assigning the integer `4` to `r`:

```
let r = .!z;;
```

Implementing DSLs is one of the most interesting applications of MSP [3]. Implementing efficient DSLs, either as interpreters or as compilers, is not an easy task. The MSP constructs allow programmers to implement a DSL as a staged interpreter, which translates the DSL code to the host language code, allowing DSLs to run as efficiently as the host code, taking advantage of the optimizations of the underlying compiler [15].

```
1    class Program {
2      delegate int sumY (int arg);
3      Expression <sumY> boo () {
4        int y = 1;
5        Expression <sumY> treesumY = x => x + y;
6        y = y + 1;
7        return treesumY;
8      }
9      int foo () {
10       int y = 10;
11       Expression <sumY> ret = boo();
12       return ret.Compile()(40); // returns 42
13     }
14   }
```

Fig. 1. Lexical scoping in variables referenced in expression trees in C#.

Mint [17] is a MSP extension to Java. Even though MSP ensures correctness while embedding languages using purely functional languages, the same is not that straightforward when we try to use MSP for embbeding imperative languages. The problem of embedding a language in an imperative language is related to side effects, as programmers can move code objects beyond the bound scope of free variables inadvertedly, a problem known as scope extrusion. Mint extends the semantics of the escape construct to impose some restrictions on side effects, not allowing side effects to appear inside a escape construct when these side effects interact with delayed code.

LINQ (Language Integrated-Query) [12] is a set of features that extends C#, allowing programmers to perform queries and manipulate data over different kinds of data storage such as XML and MDF. One can also use LINQ with data structures such as lists and arrays.

In .NET, C# and Visual Basic define a restricted type of anonymous function called an *expression lambda*, which is a function that consists of a single expression. LINQ works as an embedded DSL [7] where anonymous functions are used extensively, and was the motivating use case for the introduction of expression lambdas. When one assigns an expression lambda to a variable of type `Expression<TDelegate>`, .NET creates an AST corresponding to that expression, called an *expression tree* [1]. Expression trees can also be created programatically, manipulating node objects via the API of the `Expression` class.

Expression lambdas can access external local variables, and they respect lexical scope, regardless if they are used to declare anonymous functions or only to produce an expression tree. Figure 1 illustrates how lexical scoping is preserved

[1] Note that in C# parlance, *lambda expression* is a more general term that can refer to both single-expression anonymous functions called *expression lambdas* and multi-statement functions called *statement lambdas*. Conversion to expression trees is only supported for expression lambdas.

in expression trees. Free variable `y` in line 5 references the declaration from line 4, even when the expression tree returned in line 11 is compiled into a function in line 12.

Terra [4] is a multi-stage language for high-performance computing. It uses Lua as a host language and defines extensions for staged computation. Lua functions that run in the Lua interpreter are declared using standard Lua syntax, with the `function` construct. Staged code is declared as Terra functions, using the `terra` statement. Terra functions use similar syntax to Lua, but they are statically typed and compiled into native code using LLVM. Lua code can manipulate Terra types and functions as Lua objects. Terra also features a `quote` statement for quoting blocks of Terra code as expression objects and brackets (`[]`) as the escape operator for evaluating Lua code inside a Terra function.

When a Terra function is declared, all Lua expressions escaped inside it and Lua variables are replaced by the results of their evaluation. A Terra function, therefore, does not form a closure with respect to free Lua variables. This design trades lexical scoping for the guarantee that compiled Terra code does not need to call back into the Lua interpreter during execution.

Metalua [5] is a Lua compiler that supports compile-time meta-programming, a mechanism that allows programmers to interact with the compiler through a macro system [6]. Metalua extends Lua 5.1 to provide methods for transforming Lua code into Abstract Syntax Trees, but this code cannot contain references to local variables of an outer scope.

Our implementation generates program ASTs in the same format as Metalua, but including information about enclosing local variables. While Metalua handles arbitrary Lua code syntactically marked for quoting, our module operates only on restricted functions, but requires no quoting.

3 Lua2AST

Lua2AST is a Lua module that is able to generate ASTs given a restricted form of Lua functions, that we named lambda functions. Lambda functions are defined as functions that contain in its body a single `return` statement containing an expression. This expression can be of any kind and can also use variables of the outer lexical scope.

Lua supports functions as a first-class value. Function objects are proper closures, and are internally implemented by storing along with each function a internal set of boxed references any `local` variables belonging to outer lexical scopes. In Lua, these references are called *upvalues* [8]. Upvalues implement proper lexical scoping and are generally transparent to the Lua programmer, but they can be directly manipulated through Lua's C API and through its debug API. Lua2AST can produce a Lua function object given an AST, and references to variables in the resulting function match the lexical scoping rules of the call site where the AST was originally generated. As we will see below, this is done using the debug API to correct upvalue references in the generated code.

Lua2AST uses two external Lua libraries in its implementation: LuaDec and Lua-Parser. Luadec [13] is a Lua decompiler that takes a Lua binary chunk and returns a string with equivalent Lua source code. Lua-Parser [10] generates a Lua table representing the code AST given a string of Lua source code. Lua2AST works by decompiling the input function with LuaDec, producing an AST with Lua-Parser and finally resolving upvalue references in this AST, producing an annotated AST with additional information that allows the library to recreate the function's original environment.

Our approach to preserve variable references is to generate auxiliary closures when converting the function into AST format. These auxiliary closures are stored in the AST data structure. When compiling the AST back into a function, variable references are replaced by function calls to these closures.

This approach presents two major advantages to usual methods for adding staged computation to existing languages. Firstly, our implementation is done entirely as a library. By internally using a decompiler, we can operate directly on Lua function objects without having to use a source code preprocessor. This results in a non-intrusive approach: we did not need to create language extensions and we did not need to modify the Lua virtual machine.

Secondly, our approach is particularly suitable for a dynamic language. If Lua2AST was implemented as a static pass over the input source code, it would not be possible to transform dynamically-loaded functions into ASTs. Since Lua2AST operates entirely at runtime, we are able to operate over any suitable lambda function, including dynamically-generated Lua functions, such as those loaded during program execution using the `dostring` function.

Below, we will discuss the implementation in further detail, covering the two main functions of the Lua2AST API: `lua2ast.toAST` and `lua2ast.compile`.

3.1 Function lua2ast.toAST(*func*)

The function `toAST` generates an AST from a Lua function. It takes a Lua function as a parameter, which must be a lambda function. The function's return is a Lua table that represents an AST. This table follows a standardized format for Lua ASTs that was originally defined by the Metalua project [5]. If the received function uses upvalues, this AST will be decorated with additional data, so that upvalue references can be later reconstructed.

The function `toAST` initially calls the LuaDec decompiler to produce a source code representation of the given function. This string is sent to the `parse` function of the Lua-Parser library, producing the AST that represents the code. The AST as returned by Lua-Parser, however, would not be sufficient to reconstruct the function with proper scoping rules. Simply rebuilding the plain AST into source code and loading into Lua would produce a function where all local variables of outer scopes would turn into global variable references, since in Lua undeclared variables are treated as globals by default.

The next step, therefore, is to detect locals of outer scopes and to annotate them in the AST. This is done by scanning variable references in the AST and matching them to the list of upvalues of the function object. Firstly, we find the

parameters of the function and store them in a set. Then, we locate the free variables of the function, which are indentifiers in our expression tree that are not in the set of function arguments. These free identifiers may be references to outer locals or references to global variables. Any outer local will have a matching entry in the internal list of upvalues of the closure. We look for this entry using `debug.getupvalue()`, a function of Lua's standard library that allows us to perform introspection of a function's upvalues. When the variable is found, we decorate the AST node.

To do this decoration in our AST, we create a closure which will hold a reference to our desired variable. To do so, we use the following helper function:

```
local function newclosure()
    local temp
    return function () return temp end
end
```

This function produces a new closure that contains an upvalue and merely returns it. We then use the function `debug.upvaluejoin()`, also from the standard library. This function gets an upvalue from a Lua closure and make it refer to another upvalue from a different function. We take the upvalue from our desired variable and join it with the upvalue for the `temp` variable of our newly-created closure. We then store this auxiliary closure in the AST node that identifies the free variable.

Figure 2 illustrates the use of the `lua2ast.toAST()` function. The Lua code on this example operates equivalently to the code on Figure 1. For illustration purposes, the code also calls `lua2ast.print()`, which dumps the AST in textual format, following the syntax of Metalua. It represents node types with names such as `‘Function`; node data is represented as strings such as `"x"`. The output produced by the call at line 7 would be as follows:

```
{ ‘Function{ { ‘Id "x" },
      { ‘Return{ ‘Op{ "add", ‘Id "x", ‘Id "y"}}}
}}
```

Node `‘Id "y"` is internally decorated with a closure that returns the value of `y` defined in line 4.

3.2 Function lua2ast.compile(*ast*)

This function takes an AST and returns a new function object that is a result of the AST's compilation. When used with ASTs generated by `lua2ast.toAST()`, it will use the additional decoration to produce variable references with proper lexical scope.

Function `lua2ast.toAST()` works by generating source code, compiling it and then using the standard debug library's facilities to attach the auxiliary closures to the generated function's upvalue slots.

```
1    local lua2ast = require "lua2ast"
2
3    function boo()
4        local y = 1
5        local treesumY = lua2ast.toAST(function(x) return x + y end)
6        y = y + 1
7        lua2ast.print(treesumY)
8        return treesumY
9    end
10
11   function foo()
12       local y = 10
13       local ret = boo()
14       return lua2ast.compile(ret)(40) -- returns 42
15   end
```

Fig. 2. Lua2AST usage example

Proceeding with the example of Figure 2, the AST returned in line 13 would be initially converted into the following source code (Lua uses double-brackets for multi-line strings):

```
[[ local y
   return function(x) return x + y() end ]]
```

Prior to the reconstructed source code of the functions, we add declarations of local variables for each outer local variable referenced in the function. Note also that references for these variables are replaced by function calls in the body of the function.

We then compile this source code using Lua's standard function `loadstring()` and run it to obtain its return value: a Lua function object. Note that in the value of local `y` is not assigned in the source code. Calling this function at this point would result in an error as the upvalue for `y` points to a variable with the value of `nil`.

The final step of `lua2ast.compile()` is to fix the upvalue references to make them point to the auxiliary closures created by `lua2ast.toAST()` and stored in the AST table. For that, we use the standard function `debug.setupvalue()`, which takes a closure, an upvalue index and a Lua value, and sets the variable pointed by the upvalue to the given value. It is worth pointing out, however, that by setting this value we are not fixing the value of the original variable reference, since we replaced it in the newly generated function with a call to a proxy function, which is being fixed in its stead. We formalize this process in the following section.

Once the upvalues are fixed, `lua2ast.compile()` returns the function. In line 14 of Figure 2 we see that the result of the compilation is then further applied, and the reconstructed function runs according to the scope of the original function declared in line 5.

$$e = b \mid x \mid \text{let } x = e \text{ in } e \mid x := e \mid e(e) \mid \text{fun}(x)\{e\} \mid e \text{ op } e \mid \text{toAST}(e) \mid \text{compile}(a)$$
$$v = b \mid \langle \Gamma, x, e \rangle \mid a$$
$$a = [\text{fn } x\ a] \mid [\text{base } b] \mid [\text{var } x\ \langle \Gamma, x, e \rangle] \mid [\text{op } a\ a]$$

Fig. 3. Syntax of our version of Lua Core, extended with constructs to specify Lua2AST

4 Semantics

In this section, we specify the behavior of functions `lua2ast.toAST()` and `lua2ast.compile()` by using the formalization of a subset of Lua semantics, presented in [4] as Lua Core. We use the same formal framework of that work in order to properly compare and contrast our approach for multi-stage programming to that employed by Terra.

Lua Core depicts the notions of lexical scoping, closures and side-effects present in Lua, and is therefore mostly sufficient for our purposes. We extend this specification with an arbitrary "binary operator" expression, mimicking Lua operators supported by Lua2AST. This way, we have a recursive rule through which we can model Lua expressions as trees, to be later converted to ASTs. We also include *toAST*() and *compile*() as core language operations so we can specify their semantics separately from plain functions.

The syntax of our version of Lua Core is presented in Figure 3. A Lua expression (e) can be either a base value (b), a variable (x), a scoped variable definition (let $x = e$ in e, with $e_1; e_2$ as sugar for let $_ = e_1$in e_2), a variable assignment ($x := e$), an application ($e(e)$), a function definition ($\text{fun}(x)\{e\}$), an operation on expressions (e op e), or one of the special invocations *toAST(e)* and *compile(a)*. Lua values (v) can be base values (b), Lua ASTs (a) or closures. A closure is represented as a triple $\langle \Gamma, x, e \rangle$, consisting of a namespace $\Gamma : x \to p$ (mapping variable names x to memory positions p), an input argument x and an expression body e. A Lua AST for a function consists of a root node ($[\text{fn } x\ a]$) which may contain nodes that wrap base values ($[\text{base } b]$), operations ($[\text{op } a\ a]$), and variables ($[\text{var } x\ \langle \Gamma, x, e \rangle]$). As we will see below, the fact that variables are wrapped by a node containing a closure is central to our approach.

In Figure 4, we present the rules for evaluating Lua Core over an environment Σ, which is a tuple (Γ, S) containing a namespace $\Gamma : x \to p$ and a store $S : p \to v$ that maps memory positions to values[2]. We use $\xrightarrow{L}: (e \times \Sigma) \to (v \times \Sigma)$ for the evaluation of Lua expressions as in [4]. Rules for $\xrightarrow{L}$ presented here are equivalent to those in that work: LVAL and LVAR evaluate values and variables; LLET describes variable scoping, by evaluating e_2 in an environment created by adding the result of evaluating e_1 and assigning it to local variable x;

[2] The semantics of Lua Core in [4] is based on an environment $\Sigma = (\Gamma, S, F)$ where F is specific to Terra functions. In our presentation, we removed F. Rules reused from [4] were adapted accordingly.

$$v, \Sigma \xrightarrow{L} v, \Sigma \ (\textsc{LVal}) \qquad b, \Sigma \xrightarrow{D} [\text{base } b] \ (\textsc{DBase})$$

$$\frac{\Sigma = (\Gamma, S)}{x, \Sigma \xrightarrow{L} S(\Gamma(x)), \Sigma} \ (\textsc{LVar}) \qquad \frac{\Sigma = (\Gamma, S)}{x, \Sigma \xrightarrow{D} [\text{var } x \ \langle \Gamma, \_, x \rangle]} \ (\textsc{DVar})$$

$$\frac{\begin{array}{c} e_1, \Sigma_1 \xrightarrow{L} v_1, (\Gamma_2, S_2) \quad p \,\text{fresh} \\ e_2, (\Gamma_2[x \leftarrow p], S_2[p \leftarrow v_1]) \xrightarrow{L} v_2, (\Gamma_3, S_3) \end{array}}{\text{let } x = e_1 \text{ in } e_2, \Sigma_1 \xrightarrow{L} v_2, (\Gamma_2, S_3)} \ (\textsc{LLet}) \qquad \frac{e_1, \Sigma \xrightarrow{D} a_1 \quad e_2, \Sigma \xrightarrow{D} a_2}{e_1 \text{ op } e_2, \Sigma \xrightarrow{D} [\text{op } a_1 \, a_2]} \ (\textsc{DOp})$$

$$\frac{\begin{array}{c} e_1, \Sigma_1 \xrightarrow{L} \langle \Gamma_1, x, e_3 \rangle, \Sigma_2 \\ e_2, \Sigma_2 \xrightarrow{L} v_1, (\Gamma_3, S_3) \quad p \text{ fresh} \\ e_3, (\Gamma_1[x \leftarrow p], S_3[p \leftarrow v_1]) \xrightarrow{L} v_2, (\Gamma_4, S_4) \end{array}}{e_1(e_2), \Sigma_1 \xrightarrow{L} v_2, (\Gamma_3, S_4)} \ (\textsc{LApp}) \qquad \frac{e, \Sigma \xrightarrow{D} a}{\langle \Gamma, x, e \rangle, \Sigma \xrightarrow{D} [\text{fn } x \, a]} \ (\textsc{DFn})$$

$$[\text{base } b], \Sigma \xrightarrow{C} b, \Sigma \ (\textsc{CBase})$$

$$\frac{e_1, \Sigma_1 \xrightarrow{L} v, (\Gamma, S) \quad \Gamma(x) = p}{x \coloneqq e, \Sigma \xrightarrow{L} v, (\Gamma, S[p \leftarrow v])} \ (\textsc{LAsn})$$

$$\frac{\Sigma = (\Gamma, S)}{\text{fun}(x)\{e\}, \Sigma \xrightarrow{L} \langle \Gamma, x, e \rangle, \Sigma} \ (\textsc{LFun}) \qquad \frac{\begin{array}{c} \Sigma_1 = (\Gamma, S) \\ p \,\text{fresh} \\ \Sigma_2 = (\Gamma[x \leftarrow p], S[p \leftarrow f]) \end{array}}{[\text{var } x \ f], \Sigma_1 \xrightarrow{C} x(\_), \Sigma_2} \ (\textsc{CVar})$$

$$\frac{\begin{array}{c} e_1, \Sigma_1 \xrightarrow{L} v_1, \Sigma_2 \quad e_2, \Sigma_2 \xrightarrow{L} v_2, \Sigma_3 \\ v_3 = Op(v_1, v_2) \end{array}}{e_1 \text{ op } e_2, \Sigma_1 \xrightarrow{L} v_3, \Sigma_3} \ (\textsc{LOp}) \qquad \frac{a_1, \Sigma_1 \xrightarrow{C} e_1, \Sigma_2 \quad a_2, \Sigma_2 \xrightarrow{C} e_2, \Sigma_3}{[\text{op } a_1 \, a_2], \Sigma_1 \xrightarrow{C} e_1 \text{ op } e_2, \Sigma_3} \ (\textsc{COp})$$

$$\frac{e_1, \Sigma_1 \xrightarrow{L} \langle \Gamma, x, e_2 \rangle, \Sigma_2 \quad \langle \Gamma, x, e_2 \rangle, \Sigma_2 \xrightarrow{D} a}{\text{toAST}(e_1), \Sigma_1 \xrightarrow{L} a, \Sigma_2} \ (\textsc{LAst}) \qquad \frac{\begin{array}{c} \Sigma_1 = (\Gamma_1, S_1) \\ a, \Sigma_1 \xrightarrow{C} e, (\Gamma_2, S_2) \end{array}}{[\text{fn } x \, a], \Sigma_1 \xrightarrow{C} \langle \Gamma_2, x, e \rangle, (\Gamma_1, S_2)} \ (\textsc{CFn})$$

$$\frac{a, \Sigma_1 \xrightarrow{C} e, \Sigma_2 \quad e \xrightarrow{L} v}{\text{compile}(a), \Sigma_1 \xrightarrow{L} v, \Sigma_2} \ (\textsc{LComp})$$

Fig. 4. Rules $\xrightarrow{L}$ for the evaluation of Lua expressions, $\xrightarrow{D}$ for decompiling Lua expressions into ASTs, and $\xrightarrow{C}$ for compiling ASTs back into expressions.

LApp describes function application, propagating side effects; LAsn evaluates assignments; LFun evaluates function declarations. Our work adds new rules for $\xrightarrow{L}$: LOp describes the evaluation of an arbitrary binary operator, with semantics given by some function $Op()$; LAst describes the evaluation of $toAST()$; LComp evaluates $compile()$.

We also add two other relations: rules for decompiling a Lua function into an AST ($\xrightarrow{D}: (e \times \Sigma) \to a$) and rules for compiling ASTs back into Lua functions ($\xrightarrow{C}: (a \times \Sigma) \to (e \times \Sigma)$). These are used in LAst and LComp, respectively.

The decompilation function $\xrightarrow{D}$ takes an expression and an environment and produces an AST. Since $toAST()$ is a pure function, Σ does not figure in the codomain of $\xrightarrow{D}$. Note that $\xrightarrow{D}$ is defined only for base values (DBase), variables (DVar), the binary operator (DOp), and the initial function (DFn), mirroring the implementation of $toAST()$ in Lua2AST, which only supports functions containing these elements. Its rules deconstruct the body of the function and build the corresponding AST. Of particular interest is rule DVar, which stores in the AST node a newly created closure, which returns the value of x given the original function's environment.

The compilation function $\xrightarrow{C}$ takes an AST and an environment and produces a closure and a new environment. For each of the four decompilation rules there is a complementary compilation rule: CBase, CVar, COp and CFn. Rule CVar translates nodes representing variable references into a function call to the closure created by rule DVar. CVar assigns this closure to a variable x in the resulting environment, and produces a function call to this closure instead of a variable reference. Rule CFn returns a closure representing the entire compiled function and a new environment. This environment contains an unmodified namespace Γ_1 and a new store S_2, which includes any closures created for keeping variable references. The extended namespace Γ_2 produced by the compilation is used as the namespace of the resulting function's closure.

As a result of running $compile()$, all variable references that existed in the original function that was decompiled and was now recompiled were replaced by calls to newly-created closures that merely return the value of the corresponding variables. These closures use the original namespace from decompilation time (Γ in DVar), so the variable references are bound to the addresses they have in the lexical scope where decompilation takes place. Any variable x stored in an AST will only be evaluated when the compiled function returned by $compile(a)$ is called.

By replacing variable references to function calls to the wrapper closures, we ensure that the evaluation of variables (ultimately happening within the wrapper closures) are based on their original namespaces. This is different from the approach taken by Terra [4], where evaluation of Lua variables is done when the Terra code is generated. LINQ [12] preserves the lexical scoping of reconstructed function objects like our work does, but in our case staging happens entirely at run time.

5 Conclusion

In this work we presented an approach for multi-stage programming, through which the lexical scope of variables can be preserved by replacing variable references in the generated representation of the embedded language with closures

from the host language. When the intermediate representation is later converted into executable form, calls to these closures are produced, ensuring access to the variable in the correct context.

We implemented a module that demonstrates this approach. Our implementation uses a decompiler to convert, at runtime, Lua functions into an abstract syntax tree form decorated with closures that capture the lexical environment of free variables. The module is then able to compile the AST back into Lua, ensuring that the resulting function accesses the correct variables even if compiled at a different call site.

The technique we present here is general, and its core principle is not dependent on specificities of Lua. It could be implemented in other languages using other methods, such as source code pre-processing. However, the run-time manipulation of function objects made possible by decompilation, as opposed to compile-time manipulation of the source tree, allows us to perform multi-stage programming dynamically, operating on any suitable functions, even if they were created via dynamic code generation. This makes our approach particularly suitable for dynamic languages.

Our implementation also exploited Lua's facilities for manipulating a closure's list of upvalues, which allowed the construction of the generated functions purely through manipulation of Lua function objects, without having to resort to low-level bytecode generation. The only bytecode-level manipulation performed by Lua2AST is read-only, and is restricted to the decompiler module. Our implementation required no language extensions and no modifications to the Lua VM.

We also specified the operational semantics for the transformations performed by Lua2AST, in order to show how the lexical environment of variables is correctly preserved, and to contrast it with related work from the literature on multi-stage programming.

This work presents many possibilities for future extensions. The current implementation is a proof-of-concept that demonstrates the technique, and can be extended to support more of the host language's grammar. Another future work we envision is the development of different code-generation back-ends, supporting other languages. This would allow, for example, using Lua functions for writing prepared statements for database query languages.

References

1. Bawden, A.: Quasiquotation in Lisp. In Danvy, O., ed.: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 1999). Number NS-99-1 in BRICS Note Series, San Antonio, Texas (1999) 4–12
2. Calcagno, C., Taha, W., Huang, L., Leroy, X.: Implementing multi-stage languages using ASTs, gensym, and reflection. In: Generative Programming and Component Engineering, Springer (2003) 57–76
3. Czarnecki, K., O'Donnell, J.T., Striegnitz, J., Taha, W.: DSL implementation in MetaOCaml, Template Haskell, and C++. In: Domain-Specific Program Generation. Springer (2004) 51–72

4. DeVito, Z., Hegarty, J., Aiken, A., Hanrahan, P., Vitek, J.: Terra: A multi-stage language for high-performance computing. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '13, New York, NY, USA, ACM (2013) 105–116
5. Fleutot, F.: Metalua: Static meta-programming for Lua. `https://github.com/fab13n/metalua` (2007) [Visited on February 2015].
6. Fleutot, F., Tratt, L.: Contrasting compile-time meta-programming in Metalua and Converge. In: Proceedings of the Workshop on Dynamic Languages and Applications. (2007)
7. Fowler, M.: Domain Specific Languages. 1st edn. Addison-Wesley Professional (2010)
8. Ierusalimschy, R.: Programming in Lua, Second Edition. Lua.Org (2006)
9. Kameyama, Y., Kiselyov, O., Shan, C.c.: Closing the stage: From staged code to typed closures. In: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation. PEPM '08, New York, NY, USA, ACM (2008) 147–157
10. Maidl, A.M.: lua-parser: a Lua 5.3 parser written with LPeg. `https://github.com/andremm/lua-parser` (2013) [Visited on April 2015].
11. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part I. Communications of ACM **3**(4) (April 1960) 184–195
12. Microsoft: LINQ. `https://msdn.microsoft.com/en-us/library/bb397926.aspx` (2013) [Visited on April 2015].
13. Muhammad, H.: LuaDec: a decompiler for the Lua language. `http://luadec.luaforge.net/` (2006) [Visited on April 2015].
14. Taha, W.: Multi-stage programming: Its theory and applications. PhD thesis, Oregon Graduate Institute of Science and Technology (1999)
15. Taha, W.: A gentle introduction to multi-stage programming. In: Domain-Specific Program Generation. Springer (2004) 30–50
16. Taha, W.: A gentle introduction to multi-stage programming, part II. In: Generative and Transformational Techniques in Software Engineering II. Springer (2008) 260–290
17. Westbrook, E., Ricken, M., Inoue, J., Yao, Y., Abdelatif, T., Taha, W.: Mint: Java multi-stage programming using weak separability. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '10, New York, NY, USA, ACM (2010) 400–411