# Reasoning About Foreign Function Interfaces Without Modelling the Foreign Language

**Alexi Turcotte**
Northeastern University
Boston, MA, USA

**Ellen Arteca**
Northeastern University
Boston, MA, USA

**Gregor Richards**
University of Waterloo
Waterloo, Ontario, Canada

## Abstract

Foreign function interfaces (FFIs) allow programs written in one language (called the *host* language) to call functions written in another language (called the *guest* language), and are widespread throughout modern programming languages, with C FFIs being the most prevalent. Unfortunately, reasoning about C FFIs can be very challenging, particularly when using traditional methods which necessitate a full model of the guest language in order to guarantee anything about the whole language. To address this, we propose a framework for defining whole language semantics of FFIs without needing to model the guest language, which makes reasoning about C FFIs feasible. We show that with such a semantics, one can guarantee some form of soundness of the overall language, as well as attribute errors in well-typed host language programs to the guest language. We also present an implementation of this scheme, Poseidon Lua, which shows a speedup over a traditional Lua C FFI.

**2012 ACM Subject Classification** .

**Keywords and phrases** .

**Digital Object Identifier** 10.4230/LIPIcs...

**Acknowledgements** The authors would like to thank Rafi Turas for writing the implementation of these techniques in Poseidon Lua. We'd also like to thank Hugo Musso Gualandi for his valuable discussions/feedback. This work was partially funded by NSERC.

## 1 Introduction

Often, programming languages are designed with a specific purpose or task in mind. For example, domain specific languages (DSLs) exist for a variety of domains (e.g., querying databases), and a programmer will often choose a DSL when solving a problem that falls in its domain. But when a programmer wants to write code which touches on several domains, they turn to more general purpose languages (e.g., Java) to give them the tools they need to do everything they need to do, even though the language might be worse at any one given task as compared to a DSL written specifically for it. With so many programming languages to choose from, not only is picking the right language non-trivial, picking the "wrong" language may come back to haunt you.

To make choosing a language easier, many programming languages are equipped to interoperate with other languages, and one of the most common forms of interoperation is

© .;
licensed under Creative Commons License CC-BY

the foreign function interface (FFI). FFIs allow code written in one language (called the *host* language) to call functions written in another language (called the *guest* language), and also interface with data from the guest language, typically accomplished with wrapper code surrounding guest language values and regulating access to them. By and large the most common form of language interoperation is the C FFI since C is so fast; C FFI's are available for Python, Lua and many other dynamic languages.

Semantically, interfacing with C exposes one to all of C's foibles and irregularities: Memory accesses can fail, return values of an incorrect type, or cause system-specific undefined behavior. As such, FFI's are usually avoided in language semantics, and assumed to be either benign or absent. Unfortunately, proving properties of the behavior of a C FFI using conventional techniques is challenging: Of the existing body of work on formal specification of language interoperation, some are designed with a very specific use case in mind [6][1], and others propose general frameworks [16] which are difficult to use when reasoning about interoperation with C; these general approaches rely on fully defined semantics for all interoperating languages, which is infeasible when one of those languages is C.

In this paper, we aim to describe what behavioral guarantees remain true in the presence of an FFI, how a language hosting an FFI can guarantee its own type correctness at the interface, and how that can motivate the implementation of an FFI. We propose a framework which allows typed languages with a C FFI to be formalized and easily reasoned about *without a full model of C*. Our approach relies on a merger of the guest and host language's type systems, which allows us reason statically about the whole language and the host language's use of the FFI. Additionally, without a model of C, our semantics are *nondeterministic*—as there's no telling what an arbitrary C function might do—and we develop a novel method to reason about these nondeterministic semantics. In principle, this approach works well with other languages too, though our model of C's memory and C's types in the host language make languages with similar memory behavior to C's most suitable.

As an example of our framework in action, we also present both the semantics and implementation of Poseidon Lua, a Typed Lua C FFI. In Poseidon Lua, Typed Lua interfaces with C by holding direct pointers to C data, and is equipped to dereference these pointers, cast them, allocate C data directly, as well as call arbitrary C functions. We prove conditional soundness of Poseidon Lua, and prove that if anything "goes wrong" in well-typed Poseidon Lua programs, C code is at fault for the error. Interestingly, merging the type systems of the constituent languages eliminates the need for wrapper code around guest language values, which contributes to improved overall performance.

The main contributions of this paper are:

- a framework for merging type systems of guest and host language to allow interoperation that can be easily reasoned about;
- a semantics for Poseidon Lua, a Typed Lua C FFI, implemented with our framework;
- an actual implementation of Poseidon Lua;
- improved performance results over the previously existing Lua C FFI.

## 2 Background

In this section, we will provide requisite background for understanding our proposed framework, as well as our prototype implementation, Poseidon Lua. We will begin with an overview of foreign function interfaces, as we are describing a framework for reasoning about them. We will also discuss taint analysis, since the concept of taint features prominently in our semantics. We will then discuss Lua, Typed Lua, and Featherweight Lua, as all are crucial to understanding our language Poseidon Lua. We end the section with a quick highlight of some related work.

## 2.1 Foreign Function Interfaces

A foreign function interface (FFI) is a framework in which code written in one language (called the *host* language) may call code written in another language (called the *guest* language) as well as interface with data from that guest language. In an FFI, the guest language typically exports an API of available functions to the host language, and the host language calls said functions through the function interface. In addition to this function interface, a *data interface* is required to manage the use of one language's data in the other language.

FFIs are prevalent in modern programming. They date back to Common Lisp [11], which first introduced the concept of calling functions written in another language. Many dynamic languages, such as Python [22] and Perl [19], have easy-to-use C FFIs, allowing programmers to quickly and easily call functions written in C, a language known for its speed. In fact, C FFIs are very common, particularly in systems where performance is critical: Scientific computing environments, such as MATLAB [15] and Julia [9], carry out intensive numeric computations and simulations, and often programmers turn to external C functions available through an FFI to speed up the running time of their computationally intensive programs. This provides the user with an easy-to-use scripting language front end which may not be very performant, but with the ability to call fast functions when speed becomes an issue.

Most C FFIs interface with C in environments where C has access to all memory, including that of the host language, but there are exceptions where C is an embedded language with restricted access. A popular such system is Emscripten [27]. Emscripten is a source-to-source compiler from LLVM to JavaScript; its goal is to provide a way to run code on the web which can be compiled with LLVM but not natively run in browsers. Since JavaScript can run in essentially any web setting, compiling a language such as C to JavaScript would enable it to run reliably on a browser. With Emscripten, this can be done by first compiling the original source code down to LLVM, and then translating this to JavaScript. In terms of semantics, C is isolated to its own heap, and cannot interfere with JavaScript's; this is the same setup we have in our semantics, where C is isolated from the host.

Idiomatic usage of FFIs is to minimize the data interface between the languages to the point where only primitive, scalar values are passed between the languages, as sharing actual structured data has unfortunate behavior: Often, if the FFI even has the capability to allow the host language to store pointers to guest structures, they are mediated through a wrapper. This wrapper problem is insidious: Consider, for example a list. With each access to the next element of a list, a new wrapper must be allocated, and the old wrapper discarded, so a series of simple accesses instead becomes a series of allocations. If the FFI has no capability to access structured guest data, as in Lua's inbuilt C FFI, the programmer has to write a C accessor for every member they want to access. While the definition of these accessors can be automated, they still incur the FFI to actually access the data, as the accessors are written in C.

Formally specifying FFIs (and language interoperation in general) is not unknown to the research community. One example is early work by M. Abadi and coauthors [1], which explores dynamic typing in a statically typed language, a mixing of two very different language paradigms. Other work by K. Gray [6] tackles the problem of multi-language object extension, and presents a sound calculus modelling the language interoperability and the semantics of objects written in one language being extended in another. Additional work by J. Matthews and R. B. Findler [16] realizes whole language semantics by defining full semantics for host and guest languages, and uses *boundaries* to explicitly regulate value conversions. For our purposes, these approaches are either too specific [1][6], or do not generalize to reasoning about languages with a C FFI [16]. One particular work has a similar motivation to ours and has a fairly generalizable approach: linking types presented by Patterson and Ahmed [21]. This is discussed below in Section 2.4.

## 2.2 Dynamic Taint Analysis

Introduced by Newsome and Song in their paper [18], dynamic taint analysis is a technique initially developed for tracing potential error propagation through a system, in order to detect exploits on commodity software. The idea is that some data sources are considered *untrusted*, and data which originates from these sources is labelled with taint. This allows for the tracking of potential errors, and also can be used to restrict what the tainted data can be used for. In addition, if there is an error in the program that involves some of the tainted data, information on what potentially caused the error is all available as taint information.

The idea of dynamic taint analysis can be generalized to the tracking or propagation of any tagged (tainted) data in a program. In this work, we adapt the concept of taint to reasoning about a C FFI without modelling C: when a C call occurs, we cannot say what *will* happen, but we can reason about what *could* happen. We can model arbitrary C calls by tagging any data which could have been modified by the call with taint information identifying it, and should an error occur involving any of this data, the taint can point to the call which tampered with the data. Note that this is a property of the semantics for the purpose of proofs; we do not demand that an implementation track dynamic taint. This is explained in detail in Sections 3.2 and 4.

## 2.3 The Base for Poseidon Lua

Later in this work, we will be presenting Poseidon Lua, a Typed Lua C FFI. In this section, we present variants of Lua, the host language in Poseidon Lua. First we discuss the Lua language itself, before turning our attention to its variants and extensions.

### 2.3.1 Lua

Lua is a lightweight dynamic imperative scripting language with lexical scoping and first class functions. Lua is extensible, and offers many metaprogramming mechanisms to facilitate adaptation of the language. Its main data structure is an associative array known as the table, which can stand in for most common data structures, such as arrays, records, and objects. The functionality of tables can be further augmented through metamethods, which are essentially hooks for the Lua compiler. Classic object-oriented programming patterns, such as methods and constructors, can be easily encoded in Lua with these table extensions. A C FFI was developed for Lua by Facebook [3]: called luaffifb, it is a standard C FFI which wraps C data for use by Lua. Note that we did not implement Poseidon Lua on top of LuaJIT [20], as the implementation merely serves as a demonstration of the semantics, and JIT compilers are less amenable to such modifications. Also, LuaJIT offers the same sort of data interface that we do, but without types and with boxed references to C structures—our techniques would thus apply to it for better performance.

Our approach to reasoning about FFIs involves embedding the type system of the guest into the host language, but Lua has no type system to embed into! For this reason, Lua is not the host language in Poseidon Lua—as we need a type system, we chose Typed Lua as a base.

### 2.3.2 Typed Lua

Lua is a dynamic language, and as is often the case with these languages (see TypeScript [17] and Typed Racket [25]), there have been a few attempts at adding types in some form. One such example with Lua specifically is Tidal Lock [12], a static analyzer relying on simple type annotations. Another is Typed Lua, an optional type system for Lua [14].

In their design of Typed Lua, Maidl et al. performed an automated analysis of existing Lua programs to obtain a clear picture of how programmers use the language; they paid close attention to idiomatic Lua code to ensure that their design aligned with conventional language use. Typed Lua is optionally typed, which means that the type annotations are removed when code is compiled. Typed Lua accounts for a large subset of Lua, but a few parts are omitted, namely polymorphic functions and table types, and certain uses of the `setmetatable` function. The type system of Poseidon Lua largely matches Typed Lua's, and a full discussion will appear in Section 4.1.

Like other optionally and gradually typed languages, a program written in Typed Lua has an initial stage of *type compilation*. First, the Typed Lua code gets translated (i.e., compiled) to its corresponding Lua program, and it's during this first phase of compilation that the type information is used. At "type compile" time, typed code can be checked statically for type errors before being translated. The type information has no effect on the generated Lua code; Typed Lua programs are type checked by the compiler, and if they are well-typed, the compiler simply erases the types, generating plain Lua. Then, this Lua code is compiled to bytecode and run on the Lua virtual machine.

This multistage process means that there are two distinct versions of Lua involved in running a Typed Lua program. For clarity, in our discussion of Poseidon Lua we will use the following terminology: Typed Lua will be referred to as the *typed language* or the *user language*, since this is the language in which the programmer will be writing programs. Then, the *untyped language* or the *runtime language* refers to the subset of Lua resulting from the compilation of user language programs and additional expressions needed to deal with C. Both of these languages' grammar and operational semantics are given in Section 4.

In giving a prototype using our framework we needed to develop a formal representation of Poseidon Lua. Poseidon Lua is formalized using a *core calculus* based on Featherweight Lua (FWLua) [10], itself a core calculus of Lua. Details on FWLua are given in the next section.

### 2.3.3 Featherweight Lua

There have been a few formal specifications of Lua. First, a semantics was developed by M. Soldevila and coauthors [24] to gain a deeper understanding of Lua programs; it was mechanized in PLT Redex [4] using reduction semantics with evaluation contexts. Another semantics, not unlike Featherweight Java [8] and LambdaJS [7], proposes a core calculus for Lua. Called Featherweight Lua (FWLua) [10], this semantics focuses on formalizing what authors deem to be the essential features of Lua: first-class functions, tables, and metatables. Remaining Lua features, including expression sequencing and control structures, are shown to reduce into FWLua through an extensive desugaring process. The FWLua specification [10] also provides a reference interpreter written in Haskell.

The principle goal of FWLua is to capture core Lua idioms, and a crucial aspect of the Lua language is its table construct. Under the hood, Lua handles table access and table write with `rawsget` and `rawset` functions, respectively; these are not typically written by the programmer, but are part of how Lua drives table functionality. In their design of FWLua, the authors modelled table access and table write wholly with these **rawget** and **rawset** operations, and together with other basic semantic constructs (e.g., functions and binary operations) propose functions which mimic the semantics of full-fledged Lua. For example, to capture Lua's scoping rules, FWLua reserves certain tables to be so-called "scope tables": the `_local` table is one such example and is always accessible, and changes whenever a new scope is entered while keeping a reference to its outer scope in its `_outer` member. This way, variable access (say, of `x`) is desugared into a function which first searches through `_local`, and if `x` is not present in `_local`, then it searches recursively through `_local._outer`, and so on until `x` is located, producing `nil` if `x` is not found. This proved challenging to reason about, so we chose to promote variables to first-class language members.

To contrast Lua and FWLua, consider the following, which illustrates table construction in Lua:

```
local t = {}
t.x -- nil, uninitialized table members are nil
t.x = 42 -- t.x is now 42
t[0] = "hello" -- tables may be indexed like arrays
t["hi"] = 3.14 -- equivalent to t.hi
```

As you can see, tables can be accessed in a variety of ways in Lua, and have syntax which specifically supports different access styles, be it array-style or record-style. Tables are incrementally constructed, and can be extended at any time, much like dynamic object extension in JavaScript or other dynamic languages. In FWLua, the above translates to:

```
rawset(_local, "t", {})
rawget(rawget(_local, "t"), "x")
rawset(rawget(_local, "t"), "x", 42)
rawset(rawget(_local, "t"), 0, "hello")
rawset(rawget(_local, "t"), "hi", "hello")
```

As you can see, the **rawset** and **rawget** functions are used to write and read from a table, respectively. As we mentioned earlier, FWLua desugars variables into special table members: The table `_local` deals with local variables, and the table `_ENV` deals with global variables.

## 2.4 Related Work: Linking Types

*Linking types*, presented by Patterson and Ahmed [21], consider a different approach to reasoning about language interoperation. This work considers the languages working together as components within a larger language, which itself encompasses behavior of one language as well as the added behavior of making calls to the other language. Linking types themselves are designed to allow programmers to express and reason about one language's features in another (possibly) less expressive language which has no concept of those features. With linking types, the programmer can annotate a program to indicate where it interfaces with more expressive code in the linked language. Then, with these types, reasoning about the behavior of the whole program becomes possible.

Although both their and our work are motivated by the same essential problem, they both require modelling of both languages and focus more on the language of types than on semantics or proofs. In our work, we take a notably different approach in deciding not to model the behavior of the guest language, and instead work with the semantics of the point of intersection (i.e. the boundary between host and guest), using nondeterminism to consider the potential outcomes of the guest language calls. We believe that our types could be expressed in terms of linking types with no meaningful change to our semantics or proofs, but have not investigated this.

## 3 The Problem

FFIs are ubiquitous in programming languages, and out of these C FFIs are by far the most popular. Unfortunately, if one wants to make any guarantees about programs using a C FFI, using traditional methods of reasoning is challenging. These necessitate a full semantic model of the guest language to show anything about the overall system, and defining a formal semantics for C is very involved. Further, any such semantics will be compiler-dependent. For example, while the CompCert [2] project was groundbreaking in their implementation of a formally verified C compiler, their guarantees are limited to C programs compiled with this compiler, and do not hold for C programs compiled on other compilers (such as `gcc`).

Hypothetically, if we had a whole language semantics for a system with a C FFI, what might we be interested to show? One result of interest would be some form of type soundness for the host language, to ensure that the inclusion of the FFI in the semantics didn't cause any strange issues. Additionally, we might like to show that if any failures occur in a well-typed program calling a C FFI, then C is in some way *at fault* for the failure. In this work, we show that we can get these results *even without a full model of C!*

To achieve this, we will need to be able to reason statically about *use* of the FFI (i.e., the host's interface with the guest). The function interface of an FFI exports function handles, so we can at least check that functions are being called and used correctly, even if we don't know exactly what they do. However, the data interface of FFIs is typically built up *dynamically*, and cannot be reasoned about statically. Indeed, in a conventional FFI, wrappers are built up at runtime as values flow from one language to another, and dynamically regulate access to underlying data.

In order to fully guarantee the host language's use of the C FFI correct, we need the data interface to be static, and we can achieve this by embedding C's type system into the type system of the host language. This way, the host language can express C types and statically check its own use of C data instead of relying on runtime wrapper code like in traditional approaches. As it happens, with this scheme wrappers are no longer necessary, and their removal results in improved performance; this is discussed further in Section 5.

It's not enough to have a system in place to statically reason about the host language's use of the C FFI, as we still need to consider how we can model calls to C when we have no model of the C code, and how we can reason about the resulting semantics. The mechanisms which enable this are *taint* and *nondeterminism*, discussed next.

### 3.1 Taint and Nondeterminism

Put simply, without a model of C code, C calls are *nondeterministic*: In this scheme, a well-typed call to a C function could arbitrarily fail or succeed, as there's no telling exactly *what* the function does (e.g., a C function could dereference a null pointer or otherwise crash the program) or *what* the function returns. To account for this, at least two semantic rules for guest language calls are required: one modelling a *successful* call where the function didn't crash and returned correctly, and another modelling *failure*, where the function failed to do so (or, more generally, failed to successfully pass execution back to the host language program). Note that the rule for failure must have strictly more permissive preconditions that any rule modelling a successful call, as failure must always be an option.

Unfortunately, this simple model of nondeterministic success and failure does not fully account for all effects that C can have. For instance, execution a C function could free some memory that the host program has access to while still terminating and returning successfully, and the next dereference of a pointer to that memory would fail or return unexpected values. To fully account for this case where a successful C call has detrimental side effects, we need some additional mechanism to indicate to subsequent reductions that the function may have tampered with some data.

To model the fact that C code may unexpectedly modify data, we use the concept of *taint*; here, even successful calls to C functions will taint the memory locations which may have been modified, indicating the possible presence of a modification which could cause issues for the next access to this data (e.g., if C deallocated the memory at this location). The presence of taint at a memory location indicates that use of the location is nondeterministic: the next use of the location could either succeed, indicating that no fatal modification was made, or fail, indicating that the C call which prompted the taint to be added modified the location in such a way as to cause an error on access—either way, the effect will only be observed on the *next* access. Success in accessing a tainted location does *not* mean that the value at that location is the value that was there before it became tainted, it just means that the access did not crash; C could still have changed the value in a way that was not fatal to the program. Crucially, successfully using a tainted location will *clean* or *remove* the taint, as from that moment until the next C call we are sure that the location is not somehow broken, and that its value will not change (unless overwritten by Lua).

In summary, nondeterminism and taint together enable us to express the effects that C may have on the host language program *without* modelling C. Note that since we use a nondeterministic semantics for C and thus avoid modelling its behavior, in principle this approach works well with other languages. However, our model of C's memory and C's types in the host language make languages with similar memory behavior to C's most suitable.

To demonstrate this framework, we will present the semantics of Poseidon Lua, a Typed Lua C FFI. A high-level description of Poseidon Lua will be given in the next section.

## 3.2 Overview of Poseidon Lua

Essentially, Poseidon Lua is Typed Lua with a C FFI. It is fine-grained relative to standard FFIs: Unlike traditional FFIs, in Poseidon Lua the type systems of Lua and C are merged through a Lua pointer type, and the language has syntax with which the Lua programmer can allocate and manipulate these pointers. Specifically, Poseidon Lua allows you to: allocate and use C data, cast said pointers, and call C functions. The formal semantics are discussed fully in Section 4.

In our semantics of Poseidon Lua, Lua directly holds C values through a *pointer* to some location in a C store, which is separate from Lua's store. Structs are laid out in the C store as they would be in C, taking up space proportional to the number of struct members; these members can then be accessed with an offset equal to its position in the list of struct members (like accessing elements in an array). As explained, with no model of C, C function calls are nondeterministic, and successful calls taint everything in the C store, since the function could have modified any memory C has access to in a way which breaks a later access—for this reason, our formalization includes optional taint information in the C store. Access to clean (i.e., taint-free) locations in the C store are deterministic, while accesses to tainted locations are not, and in the event of successful access to a tainted location the taint can be removed and future accesses to that same location become deterministic (at least, until the next call to a C function).

In addition to modelling possibly errant C calls, taint allows us to model C's undefined behavior. One classic example of this is casting pointers in C. In Poseidon Lua, as in C, pointers to C values may be downcast. To model this in our formal semantics, we include types in the C store, alongside taint and the values themselves—the C store is thus a list of triples of (*value*, *type*, *optional taint*). This way, we can model the cast of a Lua pointer (to a C value) to some type $T$ by changing the type held at the pointer's location in the C store to $T$. But that's not quite enough, as casting pointers is undefined behavior in C, and we can use taint to cleanly capture this: Once cast, the location becomes tainted, and the next access to that location is nondeterministic. In this scenario, taint indicates the cast location's potential for undefined behavior when it is accessed.

Another use of taint in Poseidon Lua is in our modelling of allocation of C pointers. In C, the `calloc` function initializes the allocated memory with `0`s, so in allocating a pointer to a pointer, one is actually allocating a pointer to a `0` (which is to be treated as a pointer)! Indeed, if one were to dereference the second pointer, one would be dereferencing `0` which leads to a segmentation fault in most circumstances (`0`, of course, is `NULL` in C). To achieve this in our semantics, we taint the allocated memory location when a (Lua pointer to a) C pointer is being allocated, to indicate the potential failure of the next access to this location.

Even though we don't model C, we do make some assumptions about C's behavior: For one, we assume that C does not touch Lua's memory, and that its effects are contained to an explicitly defined C store: in other words, the shared memory has clearly defined bounds. This mirrors reality in most other FFIs, where guest code and data is not aware of host code and data. However, it is technically possible for C code to violate this assumption. We also make a simplifying assumption that all allocation and access is by word, which reduces the complexity of C data accesses without loss of generality. We require that C doesn't write new or mutate existing Lua code, otherwise we would have to scrutinize existing expressions that have yet to be reduced and would be unable to prove anything. We additionally make no explicit mention of the stack pointer, which would needlessly complicate function calls and returns for no real benefit. Further, C functions cannot call Lua functions in our formalization, so as to package all of C's effects into one black box; this is possible through callbacks, but would again be very complex without meaningfully improving the semantics. Finally, we disregard threads, which avoids needing to reason about the effects of concurrency on top of the effect of C, a layer of complexity which is outside of the scope of this project.

## 4 Semantics

Poseidon Lua is our proof-of-concept for the ideas discussed in Section 3. Having highlighted some of the stranger corners of our formal specification of Poseidon Lua in Section 3.2, we will now discuss the C FFI in its entirety.

In Poseidon Lua, Lua primarily interacts with C by calling C functions, and our merger of the two languages necessitates that C values be a part of the broader language. To represent these C values, Typed Lua has a concept of a Lua pointer to a C value, which is Lua's window to accessing C data. This means that Lua never deals directly with C values per se, and instead deals with pointers to these values. With pointers to C values as first-class citizens in Poseidon Lua, we implement the additional functionality of allocating C data as well as downcasting C pointers, both directly from Lua code without needing to call C.

We start by describing the type system in detail, and follow with a presentation of a core calculus which models the language. Then, we discuss the typing and reduction relations before concluding with a discussion of soundness and other interesting proven results.

$$
\begin{array}{llll}
T & ::= & \textbf{nil} & \textit{nil type} \\
 & | & \textbf{value} & \textit{top type} \\
 & | & \textbf{ref}\ T & \textit{reference type} \\
 & | & T_1 \cup T_2 & \textit{union type} \\
 & | & L & \textit{literal type} \\
 & | & B & \textit{base type} \\
 & | & T_1 \rightarrow_L T_2 & \textit{function type} \\
 & | & \{f_1, ..., f_n\} & \textit{table type} \\
 & | & \textbf{ptr}_{\textbf{L}}\ T_C & \textit{Lua pointer type} \\
T_C & ::= & \textbf{int} & \textit{C integer type} \\
 & | & T_C^1 \rightarrow_C T_C^2 & \textit{C function type} \\
 & | & \textbf{ptr}_{\textbf{C}}\ T_C & \textit{C pointer type} \\
 & | & \{s_1 : T_C^1, ..., s_n : T_C^n\} & \textit{C struct type}
\end{array}
\qquad
\begin{array}{llll}
f & ::= & s : T & \textit{fields} \\
 & | & \textbf{const}\ s : T & \textit{const fields} \\
L & ::= & < \textit{booleans} > & \textit{literals} \\
 & | & < \textit{numbers} > & \\
 & | & < \textit{strings} > & \\
B & ::= & \textbf{boolean} & \textit{base types} \\
 & | & \textbf{number} & \\
 & | & \textbf{string} &
\end{array}
$$

**Figure 1** The Poseidon Lua type system.

## 4.1 Type Systems

Poseidon Lua's type system is a combination of Typed Lua's [14] and C's type systems. For illustrative purposes, we chose a subset of C's type system which highlights some of C's interesting features without getting bogged down in the low-level details; we only formalized integers, pointers, structs, and functions. These are not limitations of the concept, merely simplifications made to the formalization. The story is similar with Typed Lua's type system; our function type only has a single argument type, and multivariate functions are curried to repeated application of single variable functions, by which a single argument function type suffices. In fleshing out this type system for our core calculus, we found no need for Typed Lua's type variables, recursive types, and projection types, and were able to greatly simplify their table type. Further, to simplify reasoning about Lua, we only allow string indexing in tables. Again, these are not limitations of the language, and are only simplifications for the purposes of formalization.

Our types are given in Figure 1, and explained in detail throughout this section. Type ordering is as follows:

- **value** is a supertype of all types;
- **nil** is the type of Lua's `nil` value, and is a subtype of all base types;
- union types are supertypes of their members;
- literal types are the types of literals (e.g. the literal type of `5` is *5*), and base types are the more general typical types of these literals (e.g. the base type of `5` is *numeric*)—that said, literal types are subtypes of their corresponding base types;
- function types are contravariant in their argument types, and covariant in their return types;
- table types have **width subtyping**: A table type $T$ is a supertype of a table type $T'$ which has a superset of all of the fields of $T$ (in other words, adding extra fields preserves the subtyping relationship);
- table types have **depth subtyping** only on **const** fields: If a table type $T$ has a **const** field $x$ with type $T_x$, and a table type $T'$ has all the same fields as $T$ except that field $x$ has type $T'_x$, where $T'_x <: T_x$, then $T' <: T$ (in other words, **const** field types may be specialized while preserving the subtyping relationship)

C's types are included in the Typed Lua type system (and made accessible to the user) via the “Lua pointer” C type $\mathbf{ptr_L}\ T_C$; here, $\mathbf{ptr_L}$ denotes a Lua pointer type, and $T_C$ is the C type being pointed to (e.g., $\mathbf{ptr_L}$ **int** is a Lua pointer to a C integer). As explained above, Lua only ever deals with *pointers* to C values, and not C values themselves: the only access to C values is through this pointer. C's type system is consequently entirely self contained, and is a strict subset of Lua's with no ability to reference Lua types. In some sense, C is “plugged” in to Lua through the $\mathbf{ptr_L}\ T_C$ type.

While we don't formally model C, we do need some information on C functions in order to ensure that everything shakes out properly at runtime. For example, in our semantics we model C functions as black boxes with no function body, and we ask for parameter and return types for these functions to ensure that they are called with correctly-typed arguments, even though the function bodies themselves are not modeled. What this means is that we can make sure that the functions are called correctly, but are not responsible for their internal behavior. Indeed, FFIs typically export function types as part of their API and may not always export their code—this is the situation modeled by our semantics. This is also analogous to a user calling a library for which the source code is not provided, even when the library is written in the same language as the “library host” language.

## 4.2 The Language

In this section, we present a core calculus modelling Poseidon Lua, akin to FWLua [10]. We will discuss the language of expressions, both typed and untyped, before moving on to the typing judgment and reduction relation.

We present *two* languages (in the same manner as Typed Lua, recall from Section 2.3.2): The language of *untyped expressions* $E$, also known as the language of *runtime expressions*, is the language that will actually reduce at runtime, and the language of *typed expressions* $TE$ is the language that programmers will interface with and program in, with a few minor caveats which will be discussed in time. Roughly, the typed language corresponds to Typed Lua with our added C FFI, and the untyped language corresponds to a subset of Lua with additional expressions for C interoperation. We begin with the typed language $TE$.

### 4.2.1 Typed Language

Figure 2 presents the language of typed expressions, representing the language that the programmer will be interfacing with, with some notable exceptions. The *Lua dereference* and *location update* expressions, and the *Lua location* value are not explicitly written by the programmer; they are artifacts of our typing judgment which will be presented in Section 4.3. We sometimes refer to the aforementioned expressions as *intermediate expressions*; the typed language without these is the *user language*.

These expressions largely describe a core calculus of Typed Lua, with the exception of the following C expressions:

- *C downcast* denotes the cast of expression $te$ to C type $T_C$;
- *C allocation* allocates a *C pointer* to a value of C type $T_C$;
- *C deref* is used to dereference the C pointer expression $te$;
- *C function* describes a C function with type signature $T_C$. The type $T_C$ is required by the type transformation to type these functions, as it cannot leverage the function body (as is the case with traditional functions);
- *C pointer* is a pointer to location $n$ in the C store, with expected C type $T_C$;

$$
\begin{array}{rcll}
te & ::= & v_t & \textit{value} \\
 & | & \{s_1 = v_1, ..., s_n = v_n\} & \textit{table} \\
 & | & \textbf{let}\, x : T := te_1 \,\textbf{in}\, te_2 & \textit{let binding} \\
 & | & x := te & \textit{variable update} \\
 & | & \textbf{loc}\, n := te & \textit{location update} \\
 & | & \textbf{deref}\, te & \textit{Lua dereference} \\
 & | & te_1 \, op \, te_2 & \textit{binary operation} \\
 & | & te_1(te_2) & \textit{function call} \\
 & | & x & \textit{variable} \\
 & | & te_1.te_2 & \textit{dot access} \\
 & | & te_1.te_2 := te_3 & \textit{dot update} \\
 & | & \textbf{cast}\, te\, T_C & \textit{C downcast} \\
 & | & \textbf{calloc}\, T_C & \textit{C allocation} \\
 & | & \textbf{deref}_{\mathbf{C}}\, te & \textit{C deref} \\
 & | & te_1;\ te_2 & \textit{sequence}
\end{array}
\qquad
\begin{array}{rcll}
v_t & ::= & \textbf{nil} & \textit{nil value} \\
 & | & r & \textit{register} \\
 & | & c & \textit{constant} \\
 & | & \textbf{loc}\, n & \textit{Lua location} \\
 & | & \lambda x : T.te & \textit{Lua function} \\
 & | & \textbf{cfun}\, T_C & \textit{C function} \\
 & | & \textbf{ptr}\, n\, T_C & \textit{C pointer} \\
r & ::= & \textbf{reg}\, n & \textit{table store loc} \\
c & ::= & n & \textit{number} \\
 & | & b & \textit{boolean} \\
 & | & s & \textit{string} \\
op & ::= & +, -, *, / & \textit{arithmetic} \\
 & | & \leq, <, \geq, > & \textit{order} \\
 & | & \wedge, \vee & \textit{boolean} \\
 & | & .. & \textit{concatenation} \\
 & | & == & \textit{equality}
\end{array}
$$

**Figure 2** The language of typed expressions.

- Access to C structs is done through the *dot access* and *dot update* expressions (so long as $te_1$ is a C struct), and calling C functions is done through the *function call* expression (so long as $te_1$ is a C function).

Besides the C expressions, the typed language is standard or otherwise directly analogous to some untyped expression, which we will discuss in more detail shortly.

Typed expressions will all compile into equivalent runtime expressions where the types have been erased. We explore this runtime language next.

### 4.2.2 Untyped Language

The untyped language describes the expressions which will reduce/evaluate at runtime. Generally speaking, they are analogous to some equivalent typed expression where the types have been erased. This language essentially describes a core calculus of Lua, based on FWLua (described in Section 2.3.3), though we added sequencing, let bindings, variables, table literals, and of course C interoperability. The full language can be found in Figure 3.

FWLua is a core calculus of Lua, and a number of minor modifications were required when adapting FWLua to describe Typed Lua, particularly with tables. Recall that tables are the principle data structure in Lua; as discussed previously, FWLua desugars all of Lua's table manipulation into the dual **rawget** and **rawset** constructs. For the purposes of formalization, we needed to relax FWLua's extreme desugaring; one example of this being the table literal (*table*) expression. FWLua handles table construction incrementally: an empty table is first created and stored, and then it is populated with the values at the programmer's discretion. Unfortunately, this scheme fails in typed languages, as the empty table is not a subtype of any non-empty tables, so we include a table literal to allow the expression of a full table when needed for assignments.

Our function expression is unchanged from FWLua, though we must include a new C function expression to allow FFI calls. Unlike the Lua function, which is a traditional lambda expression, the C function has far less information in it—indeed, it has no function body!

$$
\begin{array}{rcll}
e & ::= & v & \textit{value} \\
 & | & \{s_1 = v_1, ..., s_n = v_n\} & \textit{table} \\
 & | & \textbf{rawget}\, e_1\, e_2 & \textit{table select} \\
 & | & \textbf{rawset}\, e_1\, e_2\, e_3 & \textit{table update} \\
 & | & e_1\, op\, e_2 & \textit{binary operation} \\
 & | & e_1(e_2) & \textit{Lua fun. appl.} \\
 & | & x & \textit{variable} \\
 & | & x := e & \textit{var. assignment} \\
 & | & \textbf{loc}\, n := e & \textit{location update} \\
 & | & \textbf{deref}\, e & \textit{Lua dereference} \\
 & | & \textbf{let}\, x := e_1\, \textbf{in}\, e_2 & \textit{let binding} \\
 & | & \textbf{cget}\, e\, n\, T_C & \textit{C store access} \\
 & | & \textbf{cset}\, e_1\, n\, e_2\, T_C & \textit{C store update} \\
 & | & \textbf{ccall}\, e_1\, e_2\, T_C\, \beta & \textit{C function call} \\
 & | & \textbf{calloc}\, T_C\, \beta & \textit{C allocation} \\
 & | & \textbf{cast}\, e\, T_C\, \beta & \textit{C downcast} \\
 & | & e_1;\ e_2 & \textit{sequence} \\
 & | & \textbf{err}\, \beta & \textit{error expression}
\end{array}
$$

$$
\begin{array}{rcll}
v & ::= & \textbf{nil}_{\mathbf{L}} & \textit{nil value} \\
 & | & r & \textit{register} \\
 & | & c & \textit{constant} \\
 & | & \textbf{loc}\, n & \textit{Lua store loc.} \\
 & | & \textbf{ptr}_{\mathbf{L}}\, n\, T_C & \textit{C store pointer} \\
 & | & \lambda x.e & \textit{Lua function} \\
 & | & \textbf{cfun} & \textit{C function} \\
v_C & ::= & \textbf{ptr}_{\mathbf{C}}\, n & \textit{C store pointer} \\
 & | & n & \textit{C number literal}
\end{array}
$$

**Figure 3** The language of untyped, runtime expressions.

Most of the information needed for a C call is stored in the C function call expression itself.

For accesses into C structs, we have the **cget** and **cset** expressions, analogous to **rawget** and **rawset**. **cget** and **cset** are also used for accessing and writing to C pointers, which will be discussed in more detail in Section 4.4. In $\textbf{cget}\, e\, n\, T_C$, $e$ is a pointer into the C store, $n$ is the offset of the access, and $T_C$ is the type that the **cget** is expecting to read. Similarly in $\textbf{cset}\, e_1\, n\, e_2\, T_C$, $e_1$ is a pointer into the C store, $n$ is an offset, $e_2$ is the value to write, and $T_C$ is the type that the **cset** is expecting the store to contain at the referenced pointer (recall that we store type information for each pointer in the C store).

To call functions, programmers may write a standard function application as $te_1(te_2)$ in the typed language of Figure 2. The type transformation can, depending on the type of $te_1$, transform the application into either a Lua function application or a C call. The Lua function call expression $e_1(e_2)$ is straightforward, so let us focus on the C call: In $\textbf{ccall}\ e_1\ e_2\ T_C\ \beta$, $e_1$ is the C function being called, $e_2$ is the argument to that function, $T_C$ is the function's type, and $\beta$ is an identifier associated with the call (its line of code). The type is necessary since C calls exhibit nondeterministic behavior, and we can leverage $T_C$ to reason about the value that is returned from the function. The line of code information $\beta$ is related to taint, which we will describe fully when giving the semantics of the calls.

There are also a few expressions for functionality unique to C. As one might expect, $\textbf{calloc}\ T_C\ \beta$ allocates something of C type $T_C$, and $\beta$ is the identifier uniquely associated with the allocation, which allows a trace-back if a runtime error occurs. $\textbf{cast}\ e\ T_C\ \beta$ downcasts the pointer $e$ to type $T_C$, and again $\beta$ is a unique identifier associated with the cast.

## 4.3 Typing Judgment

Making a distinction between typed and untyped languages (or user and runtime languages) makes sense in many optionally or gradually typed languages, where a typed language is compiled into an untyped language which will be the one executing at runtime (recall the two stage compilation process described in the context of Typed Lua in Section 2.3.2). In these settings the typing judgment often needs to be modified to connect the languages together. We define a *type transformation* relation, a modification of the standard *typing judgment* relation, which transforms/compiles a typed expression into its corresponding untyped expression:

$$\Gamma, K \vdash te : T \rightsquigarrow e \tag{1}$$

Here, $\Gamma$ is the typing environment, which assigns types to variables, and $K$ is the typing context, containing information about the various store typings. Our runtime environment contains three stores: a table store for Lua tables, a C store for C values, and a variable store for variables. $K$ can thus be broken up into three store typings: $\Sigma_T$ describing the table store, $\Sigma_C$ for the C store, and $\Sigma_V$ for the variable store. Roughly speaking, the type transformation takes a typed expression $te$ and "compiles" it into an untyped expression $e$, assigning to it type $T$ in the context of $\Gamma$ and $K$.

In the following typing rules, some auxiliary functions will appear in the preconditions to simplify the notation. They are as follows:

- $\mathit{goodLayout}(n, T_C, \Sigma_C)$ checks to see if location $n$ in the C store typing $\Sigma_C$ represents type $T_C$. If $T_C$ is a primitive type or a pointer type, this succeeds if $\Sigma_C(n) = T_C$. As for structs, recall that they are laid out contiguously in the store: If $T_C$ is a struct type (for example, $\{s_1 : T_C^1, ..., s_n : T_C^n\}$), then each of the fields must be present in $\Sigma_C$ with the correct type, i.e. for all fields $s_i$ we must have $\Sigma_C(n + i) = T_C^i$.
- $\mathit{offsetForType}(s, T_C)$ computes the offset of member $s$ in structure type $T_C$. Our formalization of the C store lays out structs according to their type, and this function relates their type ($T_C$) to their layout in the store.

As we mentioned, in Poseidon Lua, Lua can interact with C in the following ways: allocation and access of C data, C function calls, and casting of C pointers. In this section we will focus on the typing rules for the expressions describing this FFI. The full typing rules are given in Appendix A.1.

We will first consider the rule for allocation of C data.

$$\frac{\mathit{validType}(T_C) \qquad \beta\ \mathit{unused}}{\Gamma, K \vdash \mathbf{calloc}\, T_C\ : \mathbf{ptr_L}\, T_C \rightsquigarrow \mathbf{calloc}\, T_C\ \beta} \quad (\text{TT\_CAlloc})$$

In Poseidon Lua, programmers can allocate Lua pointers to C data types (here, $T_C$), provided that the type is *valid* for allocation. For this to be the case, $T_C$ must either be a primitive type, pointer type, or struct (itself recursively made up of valid types). This prevents programmers from making nonsensical statements, such as allocating C functions in Lua. The $\beta$ here is needed when allocating C pointers: In C, allocating a pointer to a pointer can cause issues if the innermost pointer is not properly initialized, due to the default values that C inserts (pointer values are often initialized to `0`, which is an invalid memory address for C to access). This semantics will be dealt with in due course, and the inclusion of $\beta$ in the **calloc** expression is crucial to achieving the desired behavior—this will be further discussed in Section 4.4.

Having seen C allocation, we turn our attention to typing (Lua pointers to) C values:

$$\frac{\begin{array}{c} n < \mathit{length}(\Sigma_C) \\ \mathit{goodLayout}(n, T, \Sigma_C) \end{array}}{\Gamma, (\Sigma_T, \Sigma_C, \Sigma_V) \vdash \mathbf{ptr_L}\, n\, T_C : \mathbf{ptr_L}\, T_C \rightsquigarrow \mathbf{ptr_L}\, n\, T_C} \quad (\text{TT\_Lua\_Ptr})$$

C values are always "hidden behind" a Lua pointer in Poseidon Lua, and so from Lua's point of view all C values have some $\mathbf{ptr_L}$ type. In the expression $\mathbf{ptr_L}\, n\, T_C$, $n$ is the location referenced by the pointer, and $T_C$ specifies the type that the location is intended to have. The type information is required since structures do not directly inhabit the C store, and so accessing a structure would be impossible with a simpler rule, since $\Sigma_C(n)$ will never have a struct type; the type information allows us to check to see if location $n$ does in fact correspond to $T_C$ using the *goodLayout* auxiliary function, and only allow the pointer to type if it does. The typing rule for dereferencing these pointers follows.

$$\frac{\Gamma, K \vdash te : \mathbf{ptr_L}\, T_C \rightsquigarrow e \qquad \mathit{validForCDeref}(T_C) \qquad T_L = \mathit{coerceCType}(T_C)}{\Gamma, K \vdash \mathbf{deref_C}\ te : T_L \rightsquigarrow \mathbf{cget}\ e\ 0\ T_C} \text{ (TT\_VAR\_C\_DEREF)}$$

Here, beyond ensuring that $te$ is in fact a Lua pointer, we need to ensure that it is a pointer to a type that we can dereference. The C store is made up entirely of primitives and pointers, so we disallow dereferencing of things of another type (for example, we cannot dereference a C function pointer). Because our type transformation deals with Lua types only, we need to coerce $T_C$ into a Lua type to type this expression: Indeed, at runtime the dereference will coerce the value it obtains from the C store, and the coercion at this level allows such an expression to type. Note also the untyped expression corresponding to the dereference: **cget** can play the part of either simple dereferencing and also struct field access, depending on the value of its offset parameter (here, `0`). An offset of `0` indicates that we are either getting the first member in a struct, or simply dereferencing a pointer to non-struct data.

We consider C functions next.

$$\frac{}{\Gamma, K \vdash \mathbf{cfun}\,(ct_1 \to_C ct_2) : (ct_1 \to_C ct_2) \rightsquigarrow \mathbf{cfun}} \text{ (TT\_C\_FUNCTION)}$$

Here, note that the C function expression contains the whole type of the function, and without a body the function trivially types. Type information is necessary because we don't model C's semantics: In typical typing rules for functions, the return type can be determined thanks to the function body, and we have no such body to rely on here. In some sense, this is in line with what one would expect when dealing with FFIs, since part of their API is the full type of the exported functions.

Let us consider how one calls these functions:

$$\frac{\Gamma, K \vdash te_1 : (T \to_C T') \rightsquigarrow e_1 \qquad \Gamma, K \vdash te_2 : T \rightsquigarrow e_2 \qquad \beta\ \mathit{unused}}{\Gamma, K \vdash te_1(te_2) : T' \rightsquigarrow \mathbf{ccall}\, e_1\, e_2\, T'\, \beta} \text{ (TT\_C\_FUN\_APPL)}$$

In rule TT_C_FUN_APPL, we type the function application according to its return type. Note the $T'$ in the compiled (on the right of the $\rightsquigarrow$) C call: The untyped call requires the return type for reduction to be possible, and we will discuss this in more detail in Section 4.4. Since C calls are sources of taint, we include $\beta$ as an identifier uniquely associated with the call, which corresponds to the line of code occupied by the call. In the event of a failure, we can determine which call (and, thus, which function handle) is to blame.

We will now consider reading from and writing to C structs. First, reading:

$$\frac{\Gamma, K \vdash te_1 : \mathbf{ptr_L}\, T_1 \rightsquigarrow e_1 \qquad \mathit{structType}(T_1) \qquad \Gamma, K \vdash te_2 : s \rightsquigarrow e_2 \qquad s \in T_1 \qquad n = \mathit{offsetForType}\,(s, T_1)}{\Gamma, K \vdash te_1.te_2 : T_1(s) \rightsquigarrow \mathbf{cget}\ e_1\ n\ T_1(s)} \text{ (TT\_C\_Dot\_Access)}$$

Here, if $te_1$ types to $\mathbf{ptr_L}\ T_1$, $T_1$ is a struct type, and $te_2$ types to a string literal $s$ which is a field name in struct $T_1$, then the C struct member access types. Note that $te_1$ must be a Lua pointer to a C struct, as C structs themselves are not allowed in Poseidon Lua unless they are behind a Lua pointer. Also, the resulting **cget** is given the offset of field $s$ in $T_1$ (determined with the *offsetForType* auxiliary function), since the C store lays out struct members linearly in an array form.

Second, C struct member update:

$$\frac{\Gamma, K \vdash te_1 : \mathbf{ptr_L}\, T_1 \rightsquigarrow e_1 \qquad \mathit{structType}\,(T_1) \qquad \Gamma, K \vdash te_2 : s \rightsquigarrow e_2 \qquad \Gamma, K \vdash te_3 : T_1(s) \rightsquigarrow e_3 \qquad s \in T_1 \qquad n = \mathit{offsetForType}\,(s, T_1)}{\Gamma, K \vdash te_1.te_2 := te_3 : \mathbf{value} \rightsquigarrow \mathbf{cset}\ e_1\ n\ e_2\ T_1(s)} \text{ (TT\_C\_Dot\_Update)}$$

As before, if $te_1$ is a Lua pointer to a C struct type $T_1$, and $te_2$ is a string $s$ which is a member of that struct, and $te_3$ is appropriately typed, we can type the C struct update. We again emit an offset (in place of $te_2$), which the **cset** will use when writing to the C store.

Finally, Poseidon Lua allows C values to be downcast, and they type as follows:

$$\frac{\Gamma, K \vdash te : \mathbf{ptr_L}\, T'_C \rightsquigarrow e \qquad \beta\ \mathit{unused}}{\Gamma, K \vdash \mathbf{ccast}\, te\, T_C\ : T_C \rightsquigarrow \mathbf{ccast}\, e\, T_C\, \beta} \text{ (TT\_C\_Cast)}$$

Here, we notice that casting must be done through the Lua pointer, and so long as $T_C$ is a C type we allow the cast to go through. There is no mention of $T_C$ and $T'_C$ being compatible types, as C freely allows casting of pointers, and the cast merely changes the way that the bits referred to by the pointer are read. As with previous mentions of $\beta$, it features here to allow errors caused by the cast to be easily traced back to the cast.

At this point, we have explored each of the typing rules associated with Poseidon Lua's C FFI. In many cases, such as in TT_C_Fun_Appl, these rules transferred some type information to their analogous runtime expressions in order to drive the runtime functionality of the system. We discuss reduction of runtime expressions next.

## 4.4 Operational Semantics

The *reduction relation* on untyped expressions, describing the execution of programs, is:

$$e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e' \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \tag{2}$$

Here, $e$ and $e'$ are expressions in the untyped language, $\sigma_T$ and $\sigma'_T$ are *table stores*, $\sigma_C$ and $\sigma'_C$ are *C stores*, $\sigma_V$ and $\sigma'_V$ are *variable stores*. At a high level, the table store $\sigma_T$ is a list of Lua tables, the variable store $\sigma_V$ is a list of values, and finally the C store $\sigma_C$ is a list of $(v, T_C, \beta?)$ triples, where $v$ is a C value, $T_C$ is its type, and $\beta?$ is optional taint information ($\emptyset$ represents no taint, or a clean location). As we mentioned in Section 3.2, the unusual inclusion of type information in the runtime C store is required to properly model C downcast semantics.

To simplify notation, we sometimes write the reduction relation as:

$$e \,/\, \mathcal{S} \rightarrow e' \,/\, \mathcal{S}' \tag{3}$$

We refer to $\mathcal{S}$ and $\mathcal{S}'$ above as the *runtime environment*; the set of all the stores making up the state/context of the reduction.

It will be necessary to differentiate between C stores based on whether or not they are tainted; for this purpose, we say that a C store is *clean* if none of the elements of the store are themselves tainted. To simplify discussion of tainted environments, we say that a runtime environment is clean if its C store is also clean.

At the very highest level, we are formalizing a system wherein Lua code can interface with C in the following manner: allocating C data, reading from and writing to some shared memory with C, downcasting C values, and calling C functions.

Our formalization of Lua is based on FWLua [10], and we adapted their big-step semantics to a more standard small-step equivalent. For our discussion of FWLua, see Section 2.3.3. In order to mechanize our formalization, some simplifying modifications to FWLua were required, namely the promotion of variables from syntactic sugar to full-fledged language members. Of course, Lua allows you to declare and use variables, but FWLua desugars variables into access to a special store carried around at runtime. Poseidon Lua requires that FFI calls be made only from well-typed code, and so we adapted the type system of Typed Lua [14], with some modifications made possible by our simplified semantics for Lua.

Notable in Poseidon Lua is the merger of Typed Lua's and C's type systems through the Lua pointer type, and consequently the intermixing of values from both Lua and C. Lua makes reference to C values through the *Lua pointer* expression, and can both access and change the data contained in these pointers, as well as cast them to some C type. Lua may also allocate Lua pointers to C values through the **calloc** expression, without needing to make a **ccall**.

We will now turn our attention to the operational semantics of Poseidon Lua, with a focus on the C FFI, mirroring discussion of the typing judgment in Section 4.3. The full reduction rules are given in Appendix A.2. We start with the semantics of allocating C data. Consider:

$$\frac{n = \mathit{length}(\sigma_C) \qquad \sigma'_C = \sigma_C + \mathit{layoutTypeAndTaint}(T_C, \beta)}{\mathbf{calloc}\, T_C\, \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow \mathbf{ptr}\, n\, T_C \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \quad (\text{R\_CAlloc})$$

The $\mathbf{calloc}\, T_C\, \beta$ expression allocates enough memory in the C store $\sigma_C$ to accommodate a value of type $T_C$. The function *layoutTypeAndTaint* lays out type $T_C$ and taints pointer members (as per our earlier discussion in Section 3.2). If $T_C$ either is or contains a C pointer type, then we taint that location (with taint information $\beta$) to indicate to our system that its behavior is undefined until it is successfully accessed or written to. If $T_C$ is a primitive or pointer type, then we simply produce a triplet containing a default value (this is `0` for pointers), the type $T_C$, and taint if $T_C$ is a pointer type, and if $T_C$ is a struct, we lay out each of its members in a similar fashion. Following allocation, a C pointer with the location of the beginning of the newly allocated memory is produced.

Compared with C allocation, C calls have intricate semantics as we do not attempt to model the bodies of arbitrary C functions. Instead, we treat the C functions like black boxes, and consequently C function calls exhibit *nondeterministic* semantics, as any well-typed C call can either succeed or fail if the function body is made up of arbitrary C code (recall that we consider a call successful if it returns to executing the host language with some value of the expected type). In the event of successful execution, we concern ourselves with the return value and the call's potential effects on the rest of the C data. Recall our discussion that even if a call is successful, the function code might have altered the C store in a variety of ways (such as freeing some existing memory), and we must account for this possibility. We will first consider the reduction rule for a successful C call.

$$\frac{\mathit{value}\,(v_2) \qquad v = \mathit{makeValueOfType}\,(ct_2) \qquad \sigma'_C = \mathit{taintCStore}\,(\sigma_C, \beta)}{\mathbf{ccall}\ \mathbf{cfun}\ v_2\ ct_2\ \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V\ \rightarrow v \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \quad (\text{R\_CCall\_Worked})$$

Here, **ccall cfun** $v_2$ $ct_2$ $\beta$ calls a C function **cfun** with argument $v_2$. In this case, the call succeeds, and $\mathit{makeValueOfType}\,(ct_2)$ gives us $v$, something of type $ct_2$. Of course, since it's possible that the call tampered with the C store, we taint the store with taint information $\beta$, corresponding to the line of code of this function call. This notifies subsequent accesses to these memory locations of potential tampering, which modifies the semantics of those accesses. C function calls can also fail:

$$\frac{\mathit{value}\,(v_2) \qquad \sigma'_C = \mathit{taintCStore}\,(\sigma_C, \beta)}{\mathbf{ccall}\ \mathbf{cfun}\ v_2\ ct_2\ \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V\ \rightarrow \mathbf{err}\ \beta \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \quad (\text{R\_CCall\_Failed})$$

To capture that both success and failure are possible outcomes, we ensure that the premises of both rules are simultaneously satisfied: When all of R_CCall_Worked's preconditions are met, so are R_CCall_Failed's (and vice-versa). The **err** $\beta$ expression is the result of the failing call, and indicates through taint information $\beta$ which call is to blame for the failure.

Having seen the intricacies of C calls, we will turn our attention to the semantics of casting C pointers, another source of taint. For brevity, we only present the rule for casting a clean location (the other rule is not notably different). Consider:

$$\frac{\begin{array}{c} n < \mathit{length}(\sigma_C) \\ \sigma_C(n) = (v, T_C, \emptyset) \qquad \sigma'_C = \mathit{update}\,(\sigma_C, n, (v, T'_C, \beta)) \end{array}}{\mathbf{ccast}\ (\mathbf{ptr_L}\, n\, T_C)\ T'_C\ \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V\ \rightarrow \mathbf{ptr_L}\, n\, T'_C \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \quad (\text{R\_CCast})$$

Here, the location $n$ in $\sigma_C$ is updated with the new type $T'_C$ and taint information associated with the cast (thanks to the *update* auxiliary function—*update(s,l,v)* reads as "update $s$ at location $l$ with value $v$"). In C, casting a pointer merely changes how the bits being pointed to are read, and the cast may even cause an error; we achieve similar semantics with taint. When attempting to read location $n$ in $\sigma_C$ after it was cast, taint indicates that the access should be nondeterministic. To keep our system as general as possible, we don't attempt to model the cast per se, and the next read will replace $v$ with a new value of type $T'_C$ if successful, or fail with an error. We discuss the semantics of accesses next.

Thus far, we focused on the introduction of taint and fairly direct sources of nondeterminism, and we will turn our attention to taint's effect on the semantics of our system, as well as how it can be removed from the runtime environment. As an example, recall our semantics for C casts: When casting a location to some type $T_C$, the location becomes tainted. Now, imagine that the next use of the location is to store something of type $T_C$ in it; if this write succeeds, from then on we are sure about the value present at the location. Such an operation is said to *clean* the taint from the location; in our formalization, taint represents uncertainty about a C value, and once we become certain of it (e.g., we have accessed the value and no errors have occurred) we can safely remove the taint.

In more formal terms, the presence of taint at a location in $\sigma_C$ indicates that accessing that location yields nondeterministic results. To capture this, we ensure that a read or write to a tainted location can reduce to more than one expression *under the same premises*; namely, said read or write can succeed or fail.

Consider the following semantics for accessing a clean location in $\sigma_C$:

$$\frac{\sigma_C(n+o) = (v_C, T_C, \emptyset) \qquad v_{out} = coerceToLua(v_C)}{\textbf{cget}\,(\textbf{ptr}_\textbf{L}\, n\, T'_C)\ o\ T_C\ /\ \sigma_T\ /\ \sigma_C\ /\ \sigma_V\ \rightarrow v_{out}\ /\ \sigma_T\ /\ \sigma_C\ /\ \sigma_V} \text{ (R\_CGet\_No\_Taint)}$$

Here, the expression $\textbf{cget}\,(\textbf{ptr}_\textbf{L}\, n\, T'_C)\, o\, T_C$ accesses $\sigma_C$ at location $n$ with offset $o$, and is expecting something of type $T_C$. In this reduction rule, location $n+o$ in $\sigma_C$ is clean, and so the (well-typed) store access cannot fail. The access steps to $v_{out}$, which is the Lua equivalent of the C value contained in $\sigma_C$, determined through the *coerceToLua* auxiliary function. Note that the pointer's type ($T'_C$) does not necessarily need to match the expected type of the access ($T_C$); this is because **cget**s can be used for struct member access, where $T'_C$ would be a struct type and $T_C$ would be the type of the member.

$coerceToLua\,(v_C)$ is a function which takes a C value $v$ and coerces it to a Lua value. If $v_C$ is a C integer, then it is coerced to a Lua constant with the same numeric value. If $v_C$ is a C pointer $\textbf{ptr}_\textbf{C}\ m\ ct$, then it is coerced into a Lua pointer $\textbf{ptr}_\textbf{Lua}\ m\ ct$ (to the same location). Otherwise, the coercion fails.

Note the presence of a type $T_C$ in the **cget** expression. A condition of reading (and writing) from $\sigma_C$ is that the type specified for the read must match the type held in $\sigma_C$. This allows us to enforce the correct use of downcast locations, as the cast changes the type in $\sigma_C$, and future reads (and writes) must specify the new type.

We will now consider accesses to tainted locations, which can either fail or succeed. First, consider a successful access:

$$\frac{\begin{array}{c}\sigma_C(n+o) = (v, T_C, \beta) \qquad v' = makeValueOfType\,(T_C)\\ \sigma'_C = update\,(\sigma_C, n+o, (v', T_C, \emptyset) \qquad v_{out} = coerceToLua(v'))\end{array}}{\textbf{cget}\,(\textbf{ptr}\, n\, T'_C)\, o\, T_C\ /\ \sigma_T\ /\ \sigma_C\ /\ \sigma_V\ \rightarrow v_{out}\ /\ \sigma_T\ /\ \sigma_C\ /\ \sigma_V} \text{ (R\_CGet\_Taint\_Works)}$$

Here, we access $\sigma_C$ at location $n$ with offset $o$, and are expecting something of type $T_C$ as before. However, $\sigma_C(n+o)$ is tainted, resulting in nondeterminism (i.e. we do not know whether an access to this value will fail or succeed). In this reduction rule, we deal with the case of a successful access to tainted locations. Here, a successful access returns some value of the appropriate type (thanks to the *makeValueOfType* auxiliary function). The C store at $n+o$ is cleaned and updated with the new value; from this moment on, use of this location is deterministic. Note that the value was observed to be *something* of type $T_C$, though not necessarily the same value that was in that location before the C call which initially necessitated the addition of the taint.

The following reduction rule deals with failing access:

$$\frac{\sigma_C(n+o) = (v, T_C, \beta)}{\textbf{cget}\,(\textbf{ptr}\, n\, T'_C)\, o\, T_C\ /\ \sigma_T\ /\ \sigma_C\ /\ \sigma_V\ \rightarrow \textbf{err}\ \beta\ /\ \sigma_T\ /\ \sigma_C\ /\ \sigma_V} \text{ (R\_CGet\_Taint\_Fails)}$$

Here, the access fails, reporting the taint information identifying the call which tampered with this data. Note that satisfaction of rule R_CGet_Taint_Works's preconditions implies satisfaction of this rule's preconditions—this ensures that access to tainted locations can fail in any situation that it can succeed.

Similar to **cget**, **cset** has nondeterministic semantics when dealing with tainted locations. First, consider writes to clean locations:

$$\frac{\sigma_C(n+o) = (v, T_C, \emptyset) \qquad \mathit{value}(v_2) \\ v_{put} = \mathit{coerceToC}(v_2) \qquad \sigma'_C = \mathit{update}(\sigma_C, n+o, (v_{put}, T_C, \emptyset))}{\mathbf{cset}\ (\mathbf{ptr_L}\, n\, T'_C)\ o\ v_2\ T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow v_2 / \,\sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \text{(R\_CSet\_No\_Taint)}$$

In the expression $\mathbf{cset}\ (\mathbf{ptr_L}\, n\, T'_C)\, o\, v_2\, T_C$, we write $v_2$ to location $n$ with offset $o$ in $\sigma_C$, and we expect the location to have type $T_C$. Since location $n+o$ in $\sigma_C$ is clean, the store update cannot fail.

Note that we must first coerce $v_2$ to a C value $v_{put}$ to store it in $\sigma_C$. $\mathit{coerceToC}(v_2)$ is similar to the *coerceToLua* function, though it coerces Lua values to C instead. For example, if $v_2$ is a numeric constant, the function produces a C integer with the same numeric value, and if $v_2$ is a Lua pointer $\mathbf{ptr_L}\ m\ ct$, an equivalent C pointer $\mathbf{ptr_C}\ m\ ct$ is produced.

The rule for **cset**s on tainted locations is given below:

$$\frac{\sigma_C(n+o) = (v, T_C, \beta) \qquad \mathit{value}(v_2) \\ v_{put} = \mathit{coerceToC}(v_2) \qquad \sigma'_C = \mathit{update}(\sigma_C, n+o, (v_{put}, T_C, \emptyset))}{\mathbf{cset}\ (\mathbf{ptr_L}\, n\, T'_C)\ o\ v_2\ T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow v_2 \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V}$$
(R_CSet_Taint_Works)

Here, we again coerce $v_2$ to a C value $v_{put}$ to location $n$ with offset $o$ in $\sigma_C$, and we expect the location to have type $T_C$. However, $\sigma_C(n+o)$ is tainted, and so we are in a state of nondeterminism. In rule R_CSet_Taint_Works, the write succeeds: We update $\sigma_C(n+o)$ with the new value $v_{put}$ and clean the taint. Of course, failure is always an option:

$$\frac{\sigma_C(n+o) = (v, T_C, \beta)}{\mathbf{cset}\ (\mathbf{ptr_L}\, n\, T'_C)\ o\ v_2\ T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow \mathbf{err}\ \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V} \text{(R\_CSet\_Taint\_Fails)}$$

In this parallel case to T_CSet_Taint_Works, the write fails, and reports the taint information stored at $\sigma_C(n+o)$.

By now, we have explored each of the reduction relations related to Poseidon Lua's C FFI. In Section 3, we claimed that even without a model of C, as is the case in our system, the merger of the type systems of C and Typed Lua allows us to prove meaningful and interesting results about the language as a whole. The next section presents the results which we have proved, and sketches the proofs.

## 4.5 Proofs

There are two major results that we would like to prove about our semantics of Poseidon Lua. First, we would like to show some form of *soundness*, though clearly we can't have traditional type safety due to interoperation with C. Even so, we designed our semantics in such a way as to *track* C's effect on the overall system, and we can leverage that to show (conditional) soundness of the host language. Note that our proofs are mechanized in Coq, and this code in included in the artifact; a brief sketch of each proof is given here, but for the full details refer to the code.

We start with a sketch of preservation.

▶ **Theorem 1** (Preservation). *For all $K$, $te$, $T$, $e$, $S$, and $S'$ such that $\{\}, K \vdash te : T \rightsquigarrow e$, $e \,/\, S \rightarrow e' \,/\, S'$, $S$ is well-typed with respect to $K$, and both environments $S$ and $S'$ are clean, then there exists store typing $K'$, typed expression $te'$, and type $T'$ such that $\{\}, K' \vdash te' : T' \rightsquigarrow e'$ with $T' <: T$, $S'$ is well-typed with respect to $K'$, and $K'$ extends $K$.*

**Proof sketch.** Standard proof by induction on the typing derivation $\{\}, K \vdash te : T \rightsquigarrow e$. Any case where the error expression is reached is in violation of the runtime environments $S$ and $S'$ being clean, as taint is required in order to get an error. ◀

Essentially, the statement of preservation for Poseidon Lua differs from traditional statements of preservation in the stipulation that the runtime environments $S$ and $S'$ be clean. Clean environments ensure that the C error expression cannot be reached, and that the semantics are deterministic, as it's the presence of taint which begets nondeterminism.

We can similarly show progress.

▶ **Theorem 2** (Progress). *For all $K$, $te$, $T$, $e$, and $S$ such that $\{\}, K \vdash te : T \rightsquigarrow e$, $S$ is well-typed with respect to $K$, and $S$ is clean, then either $e$ is a value, or there exists clean environment $S'$, and expression $e'$ such that $e \,/\, S \rightarrow e' \,/\, S'$.*

**Proof sketch.** Another standard proof by induction on the typing derivation $\{\}, K \vdash te : T \rightsquigarrow e$. As with preservation, any case where the error expression is reached is in violation of the runtime environments $S$ and $S'$ being clean. ◀

As was the case in preservation, the statement of progress here is distinguished by the requirement that runtime environment $S$ be clean. With a clean $S'$, progress connects cleanly with preservation, allowing us to show soundness of Poseidon Lua contingent on clean environments. A sketch of soundness follows.

▶ **Theorem 3** (Soundness). *For all $K$, $te$, $T$, $e$, and $S$ such that $\{\}, K \vdash te : T \rightsquigarrow e$, either $e$ diverges, or there exists clean environment $S'$, and value $v$ such that $e \,/\, S \rightarrow^* v \,/\, S'$ and all intermediate environments are clean.*

**Proof sketch.** A standard proof, which basically amounts to applications of progress and preservation, and the intermediate environment of each step in the chain of reductions is guaranteed to be clean by construction (in a sense, progress generates clean environments). ◀

Roughly speaking, Theorem 3 states that Poseidon Lua programs in clean environments do not get stuck. The restriction to clean environments is due to the guest language, C, potentially interfering with the host language: C calls taint the environment, and accessing tainted values can lead to a stuck state even in well-typed programs. This isn't to say that you can't use C at all, as allocating simple pointers and structs does not taint the environment, and it is equally valid if some taint was once present and had been cleaned by successful accesses or writes.

Unfortunately, our statement of soundness doesn't say much for the realistic use case of Poseidon Lua (and C FFIs in general), as these systems are designed to call C code. That said, we are not without options: as before, our inclusion of taint allows us to reason about C's effects on the overall language. Crucially, failing C reductions result in the error expression **err** $\beta$, and the taint information $\beta$ can be used to identify the true culprit for the crash, even if that culprit was some earlier, seemingly unrelated expression. In short, we can show that C is to blame for failures in well-typed Poseidon Lua programs.

▶ **Theorem 4** (Always Blame C). *If the error expression* **err** $\beta$ *is reached, then there exists some C expression which is to blame.*

**Proof sketch.** Effectively, this can be shown by construction of our semantics. **err** $\beta$ can only be reached through reduction from a C expression, and the only way that such a reduction can occur is if there was some taint in the runtime environment. In **err** $\beta$, $\beta$ is

```
p = calloc Point
cCall1(p)
cCall2(p)
cCall3(p)
print(p.x)
```

```
p = calloc Point
cCall1(p)
print(p.x)
cCall2(p)
print(p.x)
cCall3(p)
print(p.x)
```

**Figure 4** Illustrative example.

taint information which identifies some C call, cast, or allocation (as those are the only expressions which can taint), and it's the identified expression that will be blamed. ◀

At a high level, Theorem 4 indicates that runtime errors in well-typed Poseidon Lua are attributable to C. This signifies that our interoperation scheme does not allow for any additional errors which are the fault of the host language, and any errors introduced by the C FFI can be traced back to C.

Taken together, Theorems 3 and 4 are analogous to soundness of static code and the gradual guarantee in gradually typed languages [26][23], though the context is otherwise quite different. This similarity betrays a certain connection between gradual typing and language interoperation, a connection equally noted by aforementioned work on linking types [21].

As we know, program execution in a tainted environment is nondeterministic. In this state, many executions are possible, and they can be categorized as follows: the program either terminates successfully, terminates unsuccessfully, or it executes until the environment is cleaned of taint. Interestingly, executions which clean the taint actually *reclaim* soundness, and are deterministic at least until the next C call.

We can show one other interesting result about Poseidon Lua programs which call C. First, recall that only clean locations gain taint when a C call occurred; this ensures proper error tracking in the event of multiple C calls possibly tainting the same data. For an illustrative example, consider the code in Figure 4.

Assume the leftmost program fails at the access to `p.x`, blaming `cCall1` and identifying it as the start of our search; here, we cannot say for sure which of `cCall1`, `cCall2`, or `cCall3` mucked with `p.x`. However, we can generate a modified program which can isolate the faulty C call. Consider the snippet on the right. If `cCall1` was the culprit of the failure, then the access immediately following it will fail. If not, and `cCall2` was at fault, then the access immediately after `cCall2` will fail. If neither of these are true, then `cCall3` is at fault, causing the final access to `p.x` to fail. This amounts to fault localization: When we are uncertain about which of a number of unsafe operations are at fault for a runtime failure, we can generate a new program which isolates the faulty operation.

## 5 Poseidon Lua: Implementation

As a demonstration of the practicality of these semantics, they have been implemented as modifications to Lua 5.3.3 [13] and Typed Lua [14]. Lua is extended to provide low-level interfaces, and Typed Lua is extended to make use of them with C types. The extensions to

Lua have no guarantees of safety or correctness on their own, and are treated as an internal implementation language for the modifications of Typed Lua. Typed Lua is extended with C types, through the addition of a C pointer in Lua which refers to C data (as explained in Section 4.1).

Typed Lua's grammar is extended as follows:

```
T ::= (all existing Typed Lua types) | PtrType

PtrType ::= ptr ptr* PtrTargetType

PtrTargetType ::= CVoidType | CPrimitiveType | Name

CType ::= CPrimitiveType | PtrType

CVoidType ::= void

CPrimitiveType ::= char | int | double

Statement ::= (all existing Typed Lua statements) | StructDeclaration

StructDeclaration ::= struct Name StructIdDecList end

StructIdDecList ::= StructIdDec StructIdDec*

StructIdDec ::= Id : CType

Expression ::= (all existing Typed Lua expressions) | CallocExpr

CallocExpr ::= calloc ( PtrTargetType )
```

`T`, in particular, is the existing Typed Lua non-terminal for types. As a consequence, any variable, parameter or field in Poseidon Lua may contain a *pointer* to a C value, but may not contain a C value directly. All other types are unmodified, and behave as they do in Typed Lua. As in C, the Poseidon Lua compiler assures that every type named in a C pointer type has a corresponding struct declaration, and that no name corresponds to multiple structure declarations, and as in C, the struct declaration defines the memory layout of objects of that type. Unlike in C, declarations are not required to precede uses of the type they declare. A simple wrapper for `calloc` is provided to assure that allocations are always of the correct size. For this prototype, we implemented only `char`s, `int`s and `double`s, but there is no conceptual limitation on implementing any other primitive type. For convenience, Poseidon Lua also provides syntax and semantics for C arrays, but they are not discussed in this work.

This modified Typed Lua compiles to Lua, extended with intrinsics to manipulate memory directly. Typed Lua code which doesn't use C features is unchanged: That is, if C `ptr`s are not used, `calloc` is not used, and the code passes type checking, then it compiles into identical Lua code without type annotations or declarations (i.e. the types are erased). Lua already provides a datatype, "light user data", intended for storing pointers to C data, and this datatype is used for all `ptr`-typed variables and fields. This is why Lua was used for this prototype. However, Lua's light user data is completely opaque to Lua code: In order to use it, one must implement a C interface, from which the underlying pointers are

```
struct House
    num_rooms : int
end
local house_1 : ptr House = calloc(House)
house_1.num_rooms = 6
```

**Figure 5** Simple Poseidon Lua code example

```
local house_1 = CS_calloc(4)
CS_storeInt(house_1, 0, 6)
```

**Figure 6** Simple Lua code example compiled from Poseidon Lua

exposed. Our principle extensions to Lua are low-level operators to directly manipulate memory through these pointers: `CS_loadChar`, `CS_storeChar`, and similar for ints, doubles and pointers. In addition, `CS_calloc` and `CS_free` are provided to give direct access to C's `calloc` and `free`, a literal `CS_NULL` corresponding to C's `NULL` is provided to check for errors, and `CS_loadString` and `CS_storeString` are provided to convert between C strings (0-terminated `char` arrays) and Lua strings. "CS" in this context is an abbreviation of "C Semantics".

Each of these low-level operators converts data between Lua's native data types and C's, given a C pointer stored in a Lua light user data, and an offset. The conversions themselves are trivial. None of these operators are intended for direct use by end users. Instead, Poseidon Lua's Typed Lua implementation compiles code which uses C types—that is, code which accesses members of `ptr`-typed variables or fields—to Lua which uses the correct operators. Internally, each low-level operator is compiled to its own opcode in Lua's bytecode.

As a simple example, the Poseidon Lua in Figure 5 compiles to the Lua in Figure 6.

As the changes in our semantics are concerned principally with C data, rather than C functions, we use a modified luaffifb for the function component of the interface. Poseidon Lua's modified luaffifb is changed only by replacing their wrapper objects with Lua's light user data, which can then be handled by Typed Lua types. The jump between C and Lua code incurs much less overhead than wrapping C data for use in Lua, so no further modifications are necessary.

## 5.1 Performance

Poseidon Lua code which doesn't use C types is just regular Typed Lua: when compiled into Lua code this will be identical to the equivalent Typed Lua program being compiled into Lua, and so will not display any performance difference. Thus, to compare the performance of Poseidon Lua against luaffifb, we need benchmarks which particularly measure the access to structured data. Unfortunately, we know of no benchmark suite intended specifically for this purpose, so instead we ported four benchmarks from the Computer Language Benchmarks Game [5]. The subset of benchmarks from CLBG were selected because they had Lua versions which used structured data types. In each case, they were rewritten so that every structured datatype used a C struct, the shape of which was taken from the C version of each benchmark. In Poseidon Lua, these structs were represented as `struct` declarations, and in luaffifb, as their dynamic declarations. In both cases, no actual C calls are made: The data is stored in C-compatible structures, and accessed through them, but the benchmark code is entirely

| | Poseidon Lua | | luaffifb | | Lua | |
|---|---|---|---|---|---|---|
| **Benchmark** | Time (s) | Std. Dev. | Time (s) | Std. Dev. | Time (s) | Std. Dev. |
| binary-trees | 18.8 | 0.447 | 202.4 | 2.97 | 22.0 | 0.707 |
| n-body | 4.0 | 0 | 40.6 | 1.14 | 4.0 | 0.707 |
| spectral-norm | 108.2 | 0 | 270.8 | 2.59 | 105.6 | 0.894 |
| fannkuch-redux | 66.8 | 2.95 | 528.8 | 9.68 | 55.0 | 0 |

**Figure 7** Comparison of performance results over various benchmarks

Lua. We compare the performance of luaffifb, which uses wrappers, to Poseidon Lua, which does not. We also include the original Lua benchmark, which does not use C structured data, for reference, although we expect no significant performance difference with respect to it. The results and standard deviations are shown in Figure 7. As expected, Poseidon Lua shows a substantial speedup over luaffifb, due to the absence of allocated wrappers at runtime. Our performance is close to original Lua, though in some benchmarks the cost of converting between C's primitive types and Lua's overwhelms other benefits.

The benchmarks were performed on Lua 5.3.3 as well as our modified version thereof, on a quad-core 1.8GHz 64-bit Intel desktop PC running Ubuntu 14.04.3LTS.

## 6 Conclusions

In this paper, we presented a framework for reasoning about C FFIs without fully modelling the guest language. This framework relies on making the data interface of the FFI static by combining the type systems of the host and guest languages, and doesn't require a model of the guest language beyond its direct interactions with the host. We also saw how making the data interface static eliminates the need for burdensome wrappers in FFI implementations, as the host language can statically check its own use of the FFI instead of needing to rely on the dynamic checks in the wrappers.

To showcase our framework, we presented Poseidon Lua, a Typed Lua C FFI. We gave the formal semantics of the C FFI in Poseidon Lua, and even without modelling C were able to guarantee some level of soundness of the host language, as well as prove that well-typed host language code is not to blame for errors that occur. We also presented an implementation of Poseidon Lua, and confirmed that making the data interface static does indeed improve the performance of the FFI.

While we focus on a C FFI, in principle our approach also works for other choices of guest language, as we deliberately avoid modelling C. That said, our model of C's memory and C's types in the host language make languages with similar memory behavior to C's most suitable, though one could plug in any type system and model memory differently if they are so inclined. We focused on a C FFI because they are very common, and prove particularly challenging to reason about with traditional methods.

## References


1 M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 213–227, New York, NY, USA, 1989. ACM. URL: `http://doi.acm.org/10.1145/75277.75296`, `doi:10.1145/75277.75296`.

2 CompCert. CompCert Main Page. `http://compcert.inria.fr`. Accessed: 2018-07-23.

3 Facebook. luaffifb. `https://github.com/facebookarchive/luaffifb`. Accessed: 2019-01-10.

4 Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex*. Mit Press, 2009.

5 Isaac Gouy. The Computer Language Benchmarks Game. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`. Accessed: 2019-01-10.

6 Kathryn E Gray. Safe cross-language inheritance. In *European Conference on Object-Oriented Programming*, pages 52–75. Springer, 2008.

7 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming*, pages 126–150. Springer, 2010.

8 Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

9 Julia. Calling C and Fortran Code. `https://docs.julialang.org/en/stable/manual/calling-c-and-fortran-code/index.html`. Accessed: 2018-07-14.

10 Hanshu Lin. Operational semantics for Featherweight Lua. *Master's Projects*, page 387, 2015.

11 Lisp. CFFI The Common Foreign Function Interface. `https://common-lisp.net/project/cffi/`. Accessed: 2018-07-25.

12 Tidal Lock. Tidal Lock Gradual Static Typing for Lua. `https://github.com/fab13n/metalua/tree/tilo/src/tilo`. Accessed: 2018-06-20.

13 Lua. Lua 5.3 documentation. `https://www.lua.org/manual/5.3/`. Accessed: 2018-06-20.

14 André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. Typed Lua: An optional type system for lua. In *Proceedings of the Workshop on Dynamic Languages and Applications*, pages 1–10. ACM, 2014.

15 MathWorks. Matlab Calling C Shared Libraries. `https://www.mathworks.com/help/matlab/using-c-shared-library-functions-in-matlab-.html`. Accessed: 2018-07-04.

16 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(3):12, 2009.

17 Microsoft. Typescript – language specification version 1.8. Technical report, Microsoft, January 2016.

18 James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

19 Graham Ollis. Perl Perl Foreign Function Interface based on GNU ffcall. `https://metacpan.org/pod/FFI`. Accessed: 2018-07-06.

20 Mike Pall. The LuaJIT project. *Web site: http://luajit. org*, 2008.

21 Daniel Patterson and Amal Ahmed. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:15, 2017. `doi:10.4230/LIPIcs.SNAPL.2017.12`.

22 Python. CFFI Documentation. `https://cffi.readthedocs.io/en/latest/`. Accessed: 2018-07-06.

23 Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

24 Mallku Soldevila, Beta Ziliani, Bruno Silvestre, Daniel Fridlender, and Fabio Mascarenhas. Decoding Lua: Formal semantics for the developer and the semanticist. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2017,

pages 75–86, New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3133841.3133848, doi:10.1145/3133841.3133848.

**25** Sam Tobin-Hochstadt, Vincent St-Amour, Eric Dobson, and Asumu Takikawa. Typed Racket Documentation. https://docs.racket-lang.org/ts-guide/. Accessed: 2018-08-01.

**26** Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, pages 1–16. Springer, 2009.

**27** Alon Zakai. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011.

# A Appendix

## A.1 Full Typing Rules

$$\frac{\mathit{validType}(T_C) \qquad \beta\ \mathit{unused}}{\Gamma, K \vdash \textbf{calloc}\ T_C\ : \textbf{ptr}_\textbf{L}\ T_C \leadsto \textbf{calloc}\ T_C\ \beta} \quad (\text{TT\_CALLOC})$$

$$\frac{\begin{array}{c} n < \mathit{length}(\Sigma_C) \\ \mathit{goodLayout}(n, T, \Sigma_C) \end{array}}{\Gamma, (\Sigma_T, \Sigma_C, \Sigma_V) \vdash \textbf{ptr}_\textbf{L}\, n\, T_C : \textbf{ptr}_\textbf{L}\, T_C \leadsto \textbf{ptr}_\textbf{L}\, n\, T_C} \quad (\text{TT\_LUA\_PTR})$$

$$\frac{\begin{array}{c} \Gamma, K \vdash \mathit{te} : \textbf{ptr}_\textbf{L}\, T_C \leadsto e \\ \mathit{validForCDeref}(T_C) \qquad T_L = \mathit{coerceCType}(T_C) \end{array}}{\Gamma, K \vdash \textbf{deref}_\textbf{C}\ \mathit{te} : T_L \leadsto \textbf{cget}\ e\ 0\ T_C} \quad (\text{TT\_VAR\_C\_DEREF})$$

$$\frac{}{\Gamma, K \vdash \textbf{cfun}\,(ct_1 \to_C ct_2) : (ct_1 \to_C ct_2) \leadsto \textbf{cfun}} \quad (\text{TT\_C\_FUNCTION})$$

$$\frac{\begin{array}{c} \Gamma, K \vdash \mathit{te}_1 : (T \to_C T') \leadsto e_1 \\ \Gamma, K \vdash \mathit{te}_2 : T \leadsto e_2 \qquad \beta\ \mathit{unused} \end{array}}{\Gamma, K \vdash \mathit{te}_1(\mathit{te}_2) : T' \leadsto \textbf{ccall}\, e_1\, e_2\, T'\, \beta} \quad (\text{TT\_C\_FUN\_APPL})$$

$$\frac{\begin{array}{c} \Gamma, K \vdash \mathit{te}_1 : \textbf{ptr}_\textbf{L}\, T_1 \leadsto e_1 \qquad \mathit{structType}(T_1) \\ \Gamma, K \vdash \mathit{te}_2 : s \leadsto e_2 \qquad s \in T_1 \qquad n = \mathit{offsetForType}\,(s, T_1) \end{array}}{\Gamma, K \vdash \mathit{te}_1.\mathit{te}_2 : T_1(s) \leadsto \textbf{cget}\ e_1\ n\ T_1(s)} \quad (\text{TT\_C\_DOT\_ACCESS})$$

$$\frac{\begin{array}{c} \Gamma, K \vdash \mathit{te}_1 : \textbf{ptr}_\textbf{L}\, T_1 \leadsto e_1 \qquad \mathit{structType}\,(T_1) \\ \Gamma, K \vdash \mathit{te}_2 : s \leadsto e_2 \qquad \Gamma, K \vdash \mathit{te}_3 : T_1(s) \leadsto e_3 \\ s \in T_1 \qquad n = \mathit{offsetForType}\,(s, T_1) \end{array}}{\Gamma, K \vdash \mathit{te}_1.\mathit{te}_2 := \mathit{te}_3 : \textbf{value} \leadsto \textbf{cset}\ e_1\ n\ e_2\ T_1(s)} \quad (\text{TT\_C\_DOT\_UPDATE})$$

$$\frac{\Gamma, K \vdash \mathit{te} : \textbf{ptr}_\textbf{L}\, T'_C \leadsto e \qquad \beta\ \mathit{unused}}{\Gamma, K \vdash \textbf{ccast}\, \mathit{te}\, T_C\ : T_C \leadsto \textbf{ccast}\, e\, T_C\, \beta} \quad (\text{TT\_C\_CAST})$$

$$\frac{\forall\, i,\ \ f_i = s_i : T_i \vee f_i = \textbf{const}\, s_i : T_i \qquad \forall\, i,\ \ \Gamma, K \vdash tv_i : T_i \leadsto v_i}{\Gamma, K \vdash \{s_1 = tv_1, ..., s_n = tv_n\} : \{f_1, ..., f_n\} \leadsto \{s_1 = v_1, ..., s_n = v_n\}} \quad (\text{TT\_TABLE})$$

$$\frac{\Gamma, K \vdash te_1 : T \rightsquigarrow e_1 \qquad \Gamma + \{x \mapsto T\}, K \vdash te_2 : T' \rightsquigarrow e_2}{\Gamma, K \vdash \textbf{let}\, x : T := te_1 \,\textbf{in}\, te_2 : T' \rightsquigarrow \textbf{let}\, x := e_1 \,\textbf{in}\, e_2} \quad (\text{TT\_Let})$$

$$\frac{x \in \Gamma}{\Gamma, K \vdash x : \Gamma(x) \rightsquigarrow x} \quad (\text{TT\_Var})$$

$$\frac{n < length(\Sigma_T)}{\Gamma, (\Sigma_T, \Sigma_C, \Sigma_V) \vdash \textbf{reg}\, n : \Sigma_T(n) \rightsquigarrow \textbf{reg}\, n} \quad (\text{TT\_Reg})$$

$$\frac{\Gamma, K \vdash te : \textbf{ref}\, T \rightsquigarrow e}{\Gamma, K \vdash \textbf{deref}\, te : T \rightsquigarrow \textbf{deref}\, e} \quad (\text{TT\_Var\_Deref})$$

$$\frac{x \in \Gamma \qquad \Gamma, K \vdash te : \Gamma(x) \rightsquigarrow e}{\Gamma, K \vdash x := te : T \rightsquigarrow x := e} \quad (\text{TT\_Var\_Assign})$$

$$\frac{\Gamma, (\Sigma_T, \Sigma_C, \Sigma_V) \vdash te : \Sigma_V(n) \rightsquigarrow e}{\Gamma, (\Sigma_T, \Sigma_C, \Sigma_V) \vdash \textbf{loc}\, n := te : T \rightsquigarrow \textbf{loc}\, n := e} \quad (\text{TT\_Loc\_Update})$$

$$\frac{\Gamma + \{x \mapsto T\}, K \vdash te : T' \rightsquigarrow e}{\Gamma, K \vdash \lambda x : T.te : (T \rightarrow_L T') \rightsquigarrow \lambda x.e} \quad (\text{TT\_Function})$$

$$\frac{\Gamma, K \vdash te_1 : (T \rightarrow_L T') \rightsquigarrow e_1 \qquad \Gamma, K \vdash te_2 : T \rightsquigarrow e_2}{\Gamma, K \vdash te_1(te_2) : T' \rightsquigarrow e_1(e_2)} \quad (\text{TT\_Lua\_Fun\_Appl})$$

$$\frac{\Gamma, K \vdash te_1 : T_1 \rightsquigarrow e_1 \qquad tableType(T_1) \qquad \Gamma, K \vdash te_2 : s \rightsquigarrow e_2 \qquad s \in T_1}{\Gamma, K \vdash te_1.te_2 : T_1(s) \rightsquigarrow \textbf{rawget}\, e_1\, e_2} \quad (\text{TT\_Dot\_Access})$$

$$\frac{\Gamma, K \vdash te_1 : T_1 \rightsquigarrow e_1 \qquad tableType(T_1) \qquad \Gamma, K \vdash te_2 : s \rightsquigarrow e_2 \qquad s \in T_1 \qquad \Gamma, K \vdash te_3 : T_1(s) \rightsquigarrow e_3}{\Gamma, K \vdash te_1.te_2 := te_3 : \textbf{value} \rightsquigarrow \textbf{rawset}\, e_1\, e_2\, e_3} \quad (\text{TT\_Dot\_Update})$$

$$\frac{\Gamma, K \vdash te : T \rightsquigarrow e \qquad T <: T'}{\Gamma, K \vdash te : T' \rightsquigarrow e} \quad (\text{TT\_Subsumption})$$

$$\frac{c \text{ constant}}{\Gamma, K \vdash c : c \rightsquigarrow c} \quad (\text{TT\_Const})$$

$$\frac{\Gamma, K \vdash te_1 : \textbf{number} \rightsquigarrow e_1 \qquad \Gamma, K \vdash te_2 : \textbf{number} \rightsquigarrow e_2 \qquad op \in \{+, -, *, /\}}{\Gamma, K \vdash te_1\, op\, te_2 : \textbf{number} \rightsquigarrow e_1\, op\, e_2} \quad (\text{TT\_Binop\_Arith})$$

$$\frac{\Gamma, K \vdash te_1 : \textbf{number} \rightsquigarrow e_1 \qquad \Gamma, K \vdash te_2 : \textbf{number} \rightsquigarrow e_2 \qquad op \in \{<, \leq, >, \geq\}}{\Gamma, K \vdash te_1 \, op \, te_2 : \textbf{boolean} \rightsquigarrow e_1 \, op \, e_2} \text{(TT\_BINOP\_ORDER)}$$

$$\frac{\Gamma, K \vdash te_1 : \textbf{boolean} \rightsquigarrow e_1 \qquad \Gamma, K \vdash te_2 : \textbf{boolean} \rightsquigarrow e_2 \qquad op \in \{\wedge, \vee\}}{\Gamma, K \vdash te_1 \, op \, te_2 : \textbf{boolean} \rightsquigarrow e_1 \, op \, e_2} \text{(TT\_BINOP\_BOOLS)}$$

$$\frac{\Gamma, K \vdash te_1 : \textbf{string} \rightsquigarrow e_1 \qquad \Gamma, K \vdash te_2 : T_2 \rightsquigarrow e_2 \qquad T_2 \in \{\textbf{string}, \textbf{number}\}}{\Gamma, K \vdash te_1 \, .. \, te_2 : \textbf{string} \rightsquigarrow e_1 \, .. \, e_2} \text{(TT\_BINOP\_STRING)}$$

$$\frac{\Gamma, K \vdash te_1 : T_1 \rightsquigarrow e_1 \qquad \Gamma, K \vdash te_2 : T_2 \rightsquigarrow e_2}{\Gamma, K \vdash te_1 \; == \; te_2 : \textbf{boolean} \rightsquigarrow e_1 \; == \; e_2} \text{(TT\_BINOP\_EQ)}$$

$$\frac{\Gamma, K \vdash te_1 : T_1 \rightsquigarrow e_1 \qquad \Gamma, K \vdash te_2 : T_2 \rightsquigarrow e_2}{\Gamma, K \vdash te_1; \, te_2 : T_2 \rightsquigarrow e_1; \, e_2} \text{(TT\_SEQUENCE)}$$

1040 ## A.2 Full Reduction Rules

$$\frac{n = \mathit{length}\,(\sigma_C) \qquad \sigma'_C = \sigma_C + \mathit{layoutTypeAndTaint}\,(T_C, \beta)}{\textbf{calloc}\, T_C \, \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \; \to \textbf{ptr}\, n \, T_C \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \text{(R\_CALLOC)}$$

$$\frac{\mathit{value}\,(v_2) \qquad v = \mathit{makeValueOfType}\,(ct_2) \qquad \sigma'_C = \mathit{taintCStore}\,(\sigma_C, \beta)}{\textbf{ccall cfun}\; v_2 \; ct_2 \; \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \; \to v \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \text{(R\_CCALL\_WORKED)}$$

$$\frac{\mathit{value}\,(v_2) \qquad \sigma'_C = \mathit{taintCStore}\,(\sigma_C, \beta)}{\textbf{ccall cfun}\; v_2 \; ct_2 \, \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \; \to \textbf{err}\, \beta \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \text{(R\_CCALL\_FAILED)}$$

$$\frac{n < \mathit{length}(\sigma_C) \qquad \sigma_C(n) = (v, T_C, \emptyset) \qquad \sigma'_C = \mathit{update}\,(\sigma_C, n, (v, T'_C, \beta))}{\textbf{ccast}\; (\textbf{ptr}_{\mathbf{L}}\, n \, T_C) \; T'_C \; \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \; \to \textbf{ptr}_{\mathbf{L}}\, n \, T'_C \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \text{(R\_CCAST)}$$

$$\frac{\sigma_C(n + o) = (v_C, T_C, \emptyset) \qquad v_{out} = \mathit{coerceToLua}(v_C)}{\textbf{cget}\; (\textbf{ptr}_{\mathbf{L}}\, n \, T'_C) \; o \; T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \; \to v_{out} \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V} \text{(R\_CGET\_NO\_TAINT)}$$

$$\frac{\sigma_C(n + o) = (v, T_C, \beta) \qquad v' = \mathit{makeValueOfType}\,(T_C) \qquad \sigma'_C = \mathit{update}\,(\sigma_C, n + o, (v', T_C, \emptyset)) \qquad v_{out} = \mathit{coerceToLua}(v')}{\textbf{cget}\; (\textbf{ptr}\, n \, T'_C) \, o \, T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \; \to v_{out} \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V} \text{(R\_CGET\_TAINT\_WORKS)}$$

$$\frac{\sigma_C(n + o) = (v, T_C, \beta)}{\textbf{cget}\; (\textbf{ptr}\, n \, T'_C) \, o \, T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \; \to \textbf{err}\; \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V} \text{(R\_CGET\_TAINT\_FAILS)}$$

$$\frac{\sigma_C(n+o) = (v, T_C, \emptyset) \qquad \mathit{value}\,(v_2) \qquad v_{put} = \mathit{coerceToC}(v_2) \qquad \sigma'_C = \mathit{update}\,(\sigma_C, n+o, (v_{put}, T_C, \emptyset))}{\textbf{cset}\ (\textbf{ptr}_\textbf{L}\, n\, T'_C)\ o\ v_2\ T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow v_2 / \,\sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \text{ (R\_CSet\_No\_Taint)}$$

$$\frac{\sigma_C(n+o) = (v, T_C, \beta) \qquad \mathit{value}\,(v_2) \qquad v_{put} = \mathit{coerceToC}(v_2) \qquad \sigma'_C = \mathit{update}\,(\sigma_C, n+o, (v_{put}, T_C, \emptyset))}{\textbf{cset}\ (\textbf{ptr}_\textbf{L}\, n\, T'_C)\ o\ v_2\ T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow v_2 \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \text{ (R\_CSet\_Taint\_Works)}$$

$$\frac{\sigma_C(n+o) = (v, T_C, \beta)}{\textbf{cset}\ (\textbf{ptr}_\textbf{L}\, n\, T'_C)\ o\ v_2\ T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow \textbf{err}\ \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V} \text{ (R\_CSet\_Taint\_Fails)}$$

$$\frac{n = \mathit{length}\,(\sigma_T) \qquad t_n = \mathit{buildTable}(\{s_1 = v_1, ..., s_n = v_n\})}{\{s_1 = v_1, ..., s_n = v_n\} \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow\ (\textbf{reg}\ n) \,/\, \sigma_T + t_n \,/\, \sigma_C \,/\, \sigma_V} \text{ (R\_Table)}$$

$$\frac{\mathit{value}\,(e_1) \qquad l = \mathit{length}(\sigma_V)}{\textbf{let}\, x := e_1\, \textbf{in}\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow [x \leftarrow l]\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V + e_1} \text{ (R\_Let)}$$

$$\frac{\mathit{value}\,(e_2) \qquad l = \mathit{length}(\sigma_V)}{(\lambda x.e)(e_2) \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow [x \leftarrow l]\, e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V + e_2} \text{ (R\_Fun\_App)}$$

$$\frac{\sigma_V(l) = v \qquad \mathit{value}(v)}{\textbf{deref}\ (\textbf{loc}\, l) \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow v \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V} \text{ (R\_Loc\_Deref)}$$

$$\frac{\mathit{value}\,(e) \qquad \sigma'_V = \mathit{update}\,(\sigma_V, l, e)}{\textbf{loc}\, l := e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma'_V} \text{ (R\_Loc\_Update)}$$

$$\frac{\sigma_T(n) = T \qquad T(s) = v}{\textbf{rawget}\ (\textbf{reg}\, n)\, s \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow v \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V} \text{ (R\_Rawget)}$$

$$\frac{\mathit{value}\,(e_3) \qquad \sigma_T(n) = T \qquad s \in T \qquad T' = \mathit{update}\,(T, s, e_3) \qquad \sigma'_T = \mathit{update}\,(\sigma_T, n, T')}{\textbf{rawset}\ (\textbf{reg}\, n)\, s\, e_3 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow \textbf{reg}\, n \,/\, \sigma'_T \,/\, \sigma_C \,/\, \sigma_V} \text{ (R\_Rawset)}$$

$$\frac{e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e' \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V}{x := e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow x := e' \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V} \text{ (R\_Var\_Assign\_Step\_1)}$$

$$\frac{e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/ \rightarrow e' \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V}{\textbf{loc}\, l := e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow \textbf{loc}\, l := e' \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V} \text{ (R\_Loc\_Update\_Step\_1)}$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e'_1 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V}{\textbf{let}\, x := e_1\, \textbf{in}\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow \textbf{let}\, x := e'_1\, \textbf{in}\, e_2 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V} \text{ (R\_Let\_Step)}$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e'_1 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V}{\textbf{rawget}\, e_1\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow \textbf{rawget}\, e'_1\, e_2 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V} \text{ (R\_Rawget\_Step\_1)}$$

$$\frac{\begin{array}{c} \mathit{value}\,(e_1) \\ e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e_2' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V' \end{array}}{\mathbf{rawget}\, e_1\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow \mathbf{rawget}\, e_1\, e_2' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \ (\text{R\_RAWGET\_STEP\_2})$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e_1' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'}{\mathbf{rawset}\, e_1\, e_2\, e_3 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow \mathbf{rawset}\, e_1'\, e_2\, e_3 \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \ (\text{R\_RAWSET\_STEP\_1})$$

$$\frac{\begin{array}{c} \mathit{value}\,(e_1) \\ e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e_2' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V' \end{array}}{\mathbf{rawset}\, e_1\, e_2\, e_3 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow \mathbf{rawset}\, e_1\, e_2'\, e_3 \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \ (\text{R\_RAWSET\_STEP\_2})$$

$$\frac{\begin{array}{c} \mathit{value}\,(e_1) \qquad \mathit{value}\,(e_2) \\ e_3 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e_3' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V' \end{array}}{\mathbf{rawset}\, e_1\, e_2\, e_3 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow \mathbf{rawset}\, e_1\, e_2\, e_3' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \ (\text{R\_RAWSET\_STEP\_3})$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e_1' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'}{e_1(e_2) \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e_1'(e_2) \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \ (\text{R\_FUN\_APP\_STEP\_1})$$

$$\frac{\begin{array}{c} \mathit{value}\,(e_1) \\ e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e_2' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V' \end{array}}{e_1(e_2) \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e_1(e_2') \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \ (\text{R\_FUN\_APP\_STEP\_2})$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e_1' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'}{e_1\, op\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e_1'\, op\, e_2 \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \ (\text{R\_BINOP\_STEP\_1})$$

$$\frac{\begin{array}{c} \mathit{value}\,(e_1) \\ e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e_2' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V' \end{array}}{e_1\, op\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e_1\, op\, e_2' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \ (\text{R\_BINOP\_STEP\_2})$$

$$\frac{\begin{array}{cc} \mathit{value}\,(e_1) & \mathit{value}\,(e_2) \\ \mathit{validL}\,(e_1) & \mathit{validR}\,(e_2) \end{array}}{e_1\, op\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow \mathit{evalOp}\,(e_1, e_2, op) \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V} \ (\text{R\_BINOP})$$

$$\frac{e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'}{\mathbf{cget}\, e\, o\, T \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow \mathbf{cget}\, e'\, o\, T \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \ (\text{R\_CGET\_STEP})$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e_1' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'}{\mathbf{cset}\, e_1\, o\, e_2\, T \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow \mathbf{cset}\, e_1'\, o\, e_2\, T \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \ (\text{R\_CSET\_STEP\_1})$$

$$\frac{\begin{array}{c} \mathit{value}\,(e_1) \\ e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e_2' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V' \end{array}}{\mathbf{cset}\, e_1\, o\, e_2\, T \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow \mathbf{cset}\, e_1\, o\, e_2'\, T \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \ (\text{R\_CSET\_STEP\_2})$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e_1' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'}{e_1;\ e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e_1';\ e_2 \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \ (\text{R\_SEQ\_STEP\_1})$$

$$\frac{\mathit{value}\,(e_1)}{e_1;\ e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \ \rightarrow e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V} \ (\text{R\_SEQ\_STEP\_THROUGH})$$