Reasoning About Foreign Function Interfaces

Blame and Nondeterministic Formal Semantics

by

Alexi Turcotte

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Foreign function interfaces (FFIs) are commonly used as a way to mix programming languages. In such systems, a program written in a *host language* calls functions written in a *guest language* from within the same program. Perhaps the most popular language to interface with is C, due in no small part to its performance (often gained through unsafe operations), and programmers often write performance-critical code in C and call it from other languages. But while C is a very performant language, it is far from being memory-safe, and one might expect C to introduce unsoundness into host language systems.

This host/guest language relationship echoes that of typed and untyped code in gradual type systems. In such systems, untyped values flowing into typed code must be cast at the boundary between typed and untyped code, and this introduces the possibility for runtime type errors in otherwise statically guaranteed code. Similarly, when a host language calls a function written in a guest language, this introduces any unsoundness in the guest language to the host language, and new errors become possible at runtime. And when an FFI is being used to call C functions, anything is possible.

In this thesis, we explore the effects of C on languages using a C FFI. To demonstrate, we give a formalization of Poseidon Lua, an environment wherein Typed Lua code may call C functions, cast C values, and allocate C data. To showcase the *interaction* between Lua and C, we choose to formalize a core calculus for Lua, and do not model C per se; instead, we reason about C as if C calls were a black-box, remaining general with respect to C's semantics, while carefully quantifying the effects that C can have on Lua by leveraging the concept of *blame* from gradual typing. We present a *nondeterministic* operational semantics for Poseidon Lua, and use blame to assure that C is always at fault for runtime errors in Lua.

Acknowledgements

I would like to thank all the people who made this thesis possible.

First and foremost, Gregor: You've been the finest advisor any student could ever ask for, and it's been an great pleasure to work with you these past years. I am especially grateful for all of your helpful comments, our insightful discussions, and our hilarious conversations. That's Mama Pepperoni™!

Also, a huge thank you to Patrick and Prabhakar for reading and picking this thesis apart, I'm looking forward to your feedback and comments.

Of course, Ellen: your energy is truly infectious, and I can't imagine what this grad school journey would have been like without you, you're an absolute unit. Let's finally see all those movies we've been meaning to watch, yeah?

Also, shout-out to my boi Ifaz, Marianna, Abel, Thierry, Aaron, and the PLG at Waterloo. I was always in good company, and y'all are pretty sick.

Last but not least, I'd like to thank my friends and family for being supportive, encouraging, and understanding; grad school life can be a bit of a roller coaster, and I know that I can always count on you.

Table of Contents

Chapter 1

Introduction

Foreign function interfaces (FFIs) allow programs written in one language to call functions from another language within the same program. FFIs are used frequently, with applications ranging from database queries to programs relying on other, faster languages to efficiently execute performance-critical sections of the program. In particular, using FFIs to call C functions is quite common: Dynamic languages such as Python [24] and Perl [21] offer an easy-to-use C FFI, as do scientific computing languages and environments such as Matlab [18] and Julia [11]. Many of these languages are relatively memory-safe, restricting programmers from possibly dangerous direct manipulation of memory addresses, which is decidedly not the case with C. In these systems, C FFIs introduce some unsoundness, and might cause code in the host language (i.e. the calling language) to crash in unexpected ways. Programs calling foreign functions might expect to lose some formal guarantees, and this is often the case, but this loss is typically offset by a significant performance gain.

Although far predating it, this style of mixing languages shares many similarities with gradual and optional typing. When the scope of programs written in dynamic languages increases, so too does the burden on the programmer to ensure that their code is manageable and maintainable. Without type annotations and the resulting compile-time error checking, these projects quickly become unwieldy, and an increasingly common practice is to *gradually* add type annotations to dynamic code, taking small steps towards fully statically-typed code. Optional and gradual typing differ in their take on runtime soundness: In optionally-typed languages, only fully-typed code is guaranteed sound, and no typechecks are performed in the presence of the dynamic type, whereas gradual type systems ensure runtime soundness in the typed component by inserting type checks at the boundary between typed and untyped code. Naturally, these checks slow down execution of the program, but they allow us to make some statement of runtime type soundness.

From this perspective, untyped code can be seen as introducing runtime type errors into otherwise sound fully statically typed code, thereby at best weakening guarantees about well-typed code sections. This is not dissimilar to FFIs, as calling code written in a less strict guest language introduces new ways in which host language programs can fail, incidentally weakening any guarantees about the host language. Formal analyses of gradually typed languages employ the notion of *blame*, which is a way to track dynamic type errors and concretize their otherwise nebulous origin [29]. At a high level, untyped values crossing into typed code are tagged with blame, and in the event that they are involved in a runtime type violation, the source of the type error can be traced back to untyped code.

To the best of our knowledge, the parallel between FFIs and gradual typing remains unexploited. The guest/host language relationship in FFIs is akin to the untyped/typed language relationship in gradual typing: Just as untyped code introduces runtime type errors in typed code, guest language calls introduce "guest language errors" into the host language. To get a handle on the effect of the dynamic `any` type, gradual typing uses blame to isolate the source of type violations, and we aim to use the concept of blame to formally reason about the effects of C foreign functions in the host language.

In this thesis, we present a formalization of Poseidon Lua, a language runtime and accompanying formalization of Typed Lua [17] interoperating with C. In this language, Typed Lua plays the part of the host language, and may call C functions (identifying it as the guest language) and directly reference C data. We formalize Poseidon Lua by developing a small calculus for Lua and extending it with types and C functionality such as function calls, downcasts, and data read and write. To draw attention to the *interoperation* of C and Lua, we chose a core calculus for Lua, based on Featherweight Lua (FWLua) [12], and do not model C's semantics per se: We do not model the specifics of the inner workings of C calls, which allows any semantics for C to be "plugged in", and instead focus on quantifying the worst-case *effect* that C can have on the Lua runtime environment, accomplished with a nondeterministic semantics for C which accounts for any choice of C semantics. Using techniques from the analysis of gradual typing, we prove the likes of conditional type soundness, C fault isolation, and we show that we can without a doubt blame C for runtime failures that occur in Lua.

The primary contributions of this work are:

- a formalization of Poseidon Lua, an efficient implementation of Lua interoperating with C;
- several proofs about Poseidon Lua, including a proof of conditional type soundness, C fault isolation, as well a proof that C is always to blame for runtime faults;

- more generally, we present a nondeterministic formal semantics (of which our semantics for Poseidon Lua is an example) which is more broadly applicable;
- a treatment of FFIs with techniques adapted from gradual typing, showing how to generalize the notion of blame to other systems.

Chapter 2

Background and Motivation

In this thesis, we tackle FFIs with formal techniques used in reasoning about gradual typing. In this chapter, we discuss important background information necessary to the understanding of our system. We start by delving into language mixing.

2.1 Foreign Function Interfaces

FFIs are prevalent in modern programming, and they date back to Common Lisp [13], which first introduced the concept of calling functions written in another language. Many dynamic languages, such as Python [24] and Perl [21], have easy-to-use C FFIs, allowing programmers to quickly and easily call functions written in C, a language made famous by its speed (and infamous for its frequently undefined behavior). In fact, C FFIs are very common in systems where performance is critical: Scientific computing environments, such as Matlab [18] and Julia [11], carry out intensive numeric computations and simulations, and often programmers turn to C to speed up the running time of their often computationally intensive programs.

The semantics of FFIs and language composition are not unknown to the research community. Early work by M. Abadi and coauthors [1] explores dynamic typing in a statically typed language, a mixing of two very different language paradigms. Other work by K. Gray [7] tackles the problem of multi-language object extension, and presents a sound calculus modeling the language interoperability and the semantics of objects written in one language being extended in another. More recently, some work by M. Grimmer and coauthors [8] has delved into the design of a virtual machine to facilitate multi-language

development, noting that different programming languages are well-suited for different tasks and so being able to use them interchangeably within the same program would be optimal depending on the tasks required.

These forays into the semantics of language mixing are fine-grained in their treatment of the technicalities of the composition; for example, J. Matthews and R. B. Findler [19]'s *boundaries* explicitly regulate value conversions. We are more interested in a higher-level view of the language composition, emphasizing the effect that C activity can have on a host language with more guarantees. To avoid being bogged down in a full formalization of C, our aim is to present a more general formal semantics in which any semantics for C can be chosen.

Multi-language environments, such as TruffleVM [8], are often implemented on virtual machines, which we discuss next.

2.2 Virtual Machines

One way to allow languages to interact is through a multi-language virtual machine (VM). VMs centralize many of the fundamental design decisions of programming languages, providing virtual environments in which programs can run. There are many examples of modern languages which run on VMs. Java is the primary example, as the portability of the JVM (Java VM) is what makes this language so easy to run on almost any platform. There are in principle no incompatibilities between programs written on different underlying hardware, as they all run on the JVM; any low-level differences are handled by the JVM itself.

Besides being able to run on multiple platforms, VMs can offer a portable target architecture for languages. For example, Java programs compile into Java bytecode which is run by the JVM, but other languages may also be compiled to Java bytecode. If another language were to compile into bytecode, then interoperation between that language and Java would be greatly simplified, since both languages are equivalent at the bytecode level. In general, when multiple languages are implemented on top of the same VM, calls between these languages are facilitated by a common underlying architecture. In this vein, TruffleVM achieves easy language composition by compiling higher-level languages (such as JavaScript, Ruby, and C) down to its underlying architecture, and guarantees efficient interoperation through its own efficient mechanisms to share data between languages.

This approach creates semantic mismatch, however. If the VM is high-level enough to easily target languages like Java, then it is awkwardly low-level to target languages like

Lua, and far too high-level to easily target languages like C. Interaction between C code compiled by a standard C compiler and dynamic-language code run by a standard VM can't be addressed without substantial modification to C. Indeed, this problem arises even in the absence of VMs per se, as many C++ libraries provide C interfaces for C-only FFIs.

Since Poseidon Lua generalizes to any implementation of C, it does not demand such a common underlying layer. However, it could be argued that C *is* that layer, as most languages, including Lua, are implemented in C or C++.

2.3 Lua, Featherweight Lua, and Typed Lua

Lua is a lightweight 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, and the functionality of tables can be further augmented through metamethods, which are essentially hooks for the Lua compiler. Lua is used in many applications, ranging from embedded code in automobiles to scripts in "AAA" game titles [15].

Idiomatic Lua programs are slightly difficult to quantify due to how easy the language is to extend, and Lua programmers often augment the functionality of their tables through the metaprogramming mechanisms of metatables and metamethods. As we will see in later examples, classic object-oriented programming patterns, such as methods and constructors, can be easily encoded in Lua with these mechanisms.

To gain a deeper understanding of Lua programs, a semantics was developed by M. Soldevila and coauthors [26], mechanized in PLT Redex [5] using reduction semantics with evaluation contexts. Another semantics, not unlike Featherweight Java [10] and LambdaJS [9], proposes a core calculus for Lua. Called Featherweight Lua (FWLua) [12], 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 [12] also provides a reference interpreter written in Haskell.

To capture Lua idioms, authors of this core calculus focused on the essential building-blocks of Lua table functionality, **rawset** and **rawget**, and, together with other basic semantics constructs like functions and binary operations, propose functions which mimic the semantics of full-fledged Lua. For example, to capture Lua's scoping rules, FWLua

reserve certain tables to be so-called "scope tables": for example, the `_local` table 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.

Lua is a dynamic language, and as is often the case with these languages (see TypeScript [20] and Typed Racket [28]), there have been a few attempts at adding some form of type information. One such example with Lua specifically is Tidal Lock [14], a static analyzer relying on simple type annotations. Another is Typed Lua, an optional type system for Lua [17].

Lua is an extensible language, and is in some sense driven by best practice. In their design of Typed Lua, authors performed an automated analysis of existing Lua programs to obtain a clear picture of how programmers use the language. The authors paid close attention to idiomatic Lua code, and ensured that their design aligned with conventional language use. Typed Lua is optionally typed, which means that the types have no effect on performance since all type information is removed when code is compiled. Authors of Typed Lua accounted for a large subset of Lua, but omitted a few parts, namely polymorphic functions and table types, and certain uses of the `setmetatable` function.

Poseidon Lua can be essentially described as Typed Lua interoperating with C, and we will discuss this further near the end of this chapter, and we will describe its type system in detail later in Chapter 4. For now, to get a better handle on these languages, we will show several code snippets from all three, starting with tables.

The following 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
```

In Lua, tables can be accessed in a variety of ways, 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. In FWLua, the above directly 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. Note again that our adaptation of FWLua allows for variables.

FWLua additionally desugars statement sequencing into function application ([23], pp. 119-120). The statement $e_1; e_2$ desugars into $(\lambda x.e_2)e_1$ where x is not a free variable of e_2. That said, for simplicity's sake, we will write FWLua in sequence. Note that our formalization of Poseidon Lua includes sequencing, which greatly simplified one of our proofs.

Typed Lua disallows most incremental construction of tables, as it is more strict and requires that table member names be explicitly declared. The syntax for table construction follows:

```
local p : {string: number, string: number} = {x = 3, y = 4}
p.x -- 3
p.z = 5 -- runtime error
```

Here, it is no longer possible to incrementally construct a table, and full initialization of tables is required. One may also declare interfaces in this language, which are just names for a structural type. Consider:

```
local interface Point
  x : number
  y : number
end
...
p : Point = {x = 3. y = 4}
```

Typically, gradual and optional type systems that are developed based on an existing dynamic language are required to easily (i.e., with minimal annotations) type code idiomatic to the language. In Lua, incremental table construction is very typical, so Typed

Lua allows this in a limited manner. Keeping in mind the `Point` interface from above, consider:

```
local q = {}
q.x = 3
q.y = 4
local p : Point = q
```

Typed Lua has some type inference, and here the type of `q` at the assignment site can be inferred to be {`string: number, string: number`}, which matches the type of the `Point` interface. We do not model this inference or these casts, as we feel that they are more of a convenience than a necessity, since one could just construct a table as-is without this incremental construction.

Another important facet of any language is functions. The following illustrates syntax for declaring a function in Lua:

```
function addTwo(a, b)
  return a+b
end
```

In Lua, functions are first-class values, and you can store functions as table members. As such, we can define a similar function in this alternate syntax:

```
local subTwo = function (a, b)
  return a - b
end
local calc = {}
calc.sub = subTwo
```

The function definition enables the call `subTwo(4, 2)`. Additionally, we stored the function as a table member in the table `calc`, so it may be called with `calc.sub(2, 1)`.

Functions in FWLua are always restricted to accept a single argument, and multi-argument functions achieved through currying of single-argument functions. To illustrate, consider the function `subTwo`, formulated in FWLua below.

```
rawset(_local, "subTwo",
       function a return
         function b return
           a - b
         end
       end)
```

Note that for simplicity, we write `function a return e end` to mean $\lambda a.e$ (all functions return a value in FWLua). To call `subTwo`, one would write:

```
(rawget(_local, "subTwo")(5))(3)
```

Above, the expression `rawget(_local, "subTwo")(5)` return a function which takes 1 argument (in fact, the function produced is exactly `function b return 5 - b end`), so we apply it to `3` in order to perform the inner computation. This is a standard technique, and you can write a similar expression in Lua (and indeed in any language with first-class functions):

```
function subTwo (a)
  subFromA = function (b) return a - b end
  return subFromA
end
subTwo(5)(3)
```

Again, here the expression `subTwo(5)` returns a single-argument function, and we apply that function to `3` to obtain the desired result.

As for Typed Lua, typing functions is unsurprising. Consider below the `addTwo` function from earlier, written in Typed Lua:

```
function addTwo(a : number, b : number) : number
  return a + b
end
```

Things get a little more complicated with the composition of single argument functions, but it is nonetheless possible to type them. Consider the Typed Lua version of `subTwo`:

```
function subTwo (a : number) : number -> number
  subFromA = function (b : number) : number
    return a - b
  end
  return subFromA
end
```

Here, it is clear that `subTwo(_)` produces a function, and must be applied to another argument to fully evaluate.

A big part of Lua's flexibility is derived from its metatables and metamethods. A metatable is a table which is gained through the `setmetatable` function: it contains

metamethods (such as `__add` and `__eq`) which overload the appropriate operators to instead call the metamethod code. These constructs allow programmers to add functionality to tables and customize their behaviour, and is markedly similar to operator overloading in other languages. To illustrate, consider the following:

```
Point = {} -- initialize Point prototype
Point.mt = {} -- set metatable for Point
function Point.new(x, y) -- contructor for a Point
  local p = {}
  setmetatable(p, Point.mt) -- give p Point's metatable
  p.x = x
  p.y = y
  return p
end
```

Now, when we create `Point`s using the `Point.new` constructor, we will receive a new table with `x` and `y` fields with `Point.mt` as a metatable. We can populate the metatable as follows:

```
function Point.add(p1, p2)
  local p = Point.new(p1.x + p2.x, p1.y + p2.y)
  return p
end
Point.mt.__add = Point.add
```

`Point.add` takes two points and adds them together element-wise. When we set `Point.mt`'s `__add` member to `Point.add`, we are stating that when adding two `Point`s, instead of using the arithmetic add operator, we should call `Point.add`. In other words, we set the add metamethod on `Point.mt` to be `Point.add`. Now, we can write:

```
local p1 = Point.new(1, 2)
local p2 = Point.new(3, 4)
local p3 = p1 + p2
p3 -- p3.x is 4, p3.y is 6
```

Tables with `Point.mt` as a metatable have the addition operator overloaded for them, and now addition over these tables will look for `x` and `y` members to add together. The same can be done with all arithmetic and relational operators in Lua.

Two more useful metamethods are `__index` and `__newindex`. If a table's metatable has an `__index` metamethod, accessing nonexistent members in the table will redirect

to `__index`. Similarly, if a metatable has an `__newindex` metamethod, assignments to uninitialized fields will redirect to `__newindex`. This is useful for inheritance, but also for prototyping. Consider:

```
Racer = {}
Racer.DefaultInfo = {startKm = 0, curKm = 0, endKm = 42}
Racer.mt = {}
function Racer.new()
  local r = {}
  setmetatable(r, Racer.mt)
  return r
end
```

The above defines a prototype for someone running a marathon. We can populate the `Racer.mt` metatable with an `__index` member, which will redirect accesses to uninitialized table members:

```
Racer.mt.__index = function (racer, key)
  return Racer.DefaultInfo[key]
end
```

Now, it is clear that we don't set any of the `Racer`'s fields in the constructor. So, if we try to access one of those fields, we will get:

```
local r = Racer.new()
r.startKm -- produces 0
r.endKm -- produces 42
```

When Lua discovers that `r` has no member `startKm`, it will look at `r`'s metatable to see if it has an `__index` member. If it does, it will call it with two arguments: the table being accessed (`r`) and the member that "missed" (`startKm`). We may also define a `__newindex` metamethod, which works similarly but for writing to nonexistent members.

In Lua, tables can also play the part of objects. That said, Lua doesn't support objects per se, and instead programmers need to extend tables and construct object functionality with metatables and metamethods. Consider:

```
local iterator = {}
iterator.cur = 0
```

```
iterator.inc =
  function (iter)
    iter.cur = iter.cur + 1
  end
iterator.inc(iterator) -- cur is now 1
```

Here, we created an iterator-style object, with a `cur` field for the current index and an `inc` method to increment that index. Unfortunately, the above example isn't really object-oriented in the traditional sense, when we might expect the `inc` method to be able to somehow reference the calling iterator's field (indeed, in object oriented languages one would expect to write `iterator.inc()`, without passing the iterator itself). Lua's class functionality is reminiscent of object-orientation in prototype-based languages such as JavaScript. Consider:

```
Iterator = {cur = 0} -- the Iterator "class"
function Iterator:new (t)
  local iter = t or {}
  setmetatable(iter, self)
  self.__index = self
  return iter
end
function Iterator:inc ()
  self.cur = self.cur + 1
end
```

The constructor `Iterator:new` uses some Lua language features which are worth mentioning in detail. First, either an empty table is allocated and stored in local variable `iter`, or the argument `t` is placed there if such a `t` was given. Then, we call `setmetatable(iter, self)`, which sets `iter`'s metatable to `self`, which we know to be `Iterator` thanks to the colon in the function name. As before, this has the effect of redirecting field accesses on fields which do not exist in `iter` to `Iterator`; this way, writing `iter.cur` before it has been initialized will redirect the lookup to `Iterator`, which will produce `0`. In setting `self`'s (i.e. Iterator's) `__index` field to itself, we ensure that field lookups do not go beyond `Iterator`. Once all that bookkeeping has been completed, `iter` is returned as a fresh `Iterator`.

This is a typical setup for classes and object-orientation in Lua. The `Iterator` class is effectively a prototype, and a call to `Iterator:new()` creates a new instance of the prototype. The colon in `Iterator:new()` is merely syntactic sugar for having a "self" argument,

i.e. `Iterator:new()` is equivalent to `Iterator.new(self)`. To create an `Iterator`, one writes:

```
i = Iterator:new()
ii = Iterator:new()
i.cur -- 0
i.inc() -- increments i's cur field to 1
ii.cur -- still 0, as expected
```

Inheritance is also straightforward in Lua. Imagine that we wanted to define a `MaxedIterator`, which gives a maximum to the iterator's `cur` value. Consider:

```
MaxedIterator = Iterator:new()
```

This defines `MaxedIterator` as an instance of `Iterator`. To create one (with, say, a `max` field), we can write:

```
miter = MaxedIterator:new({max = 4})
```

In passing a value for `t`, the `Iterator` constructor is now modifying a table with a `max` field (instead of the default empty table). At this point, we can call the `inc` method in `miter`:

```
miter:inc() -- increments miter.cur to 1
```

The `MaxedIterator` prototype does not have an `inc` method, but the metatable set in the `Iterator` constructor ensures that when the method lookup fails, it is redirected to `Iterator`, where it is found. Now, we can modify the functionality of `inc` to take the `max` into account:

```
function MaxedIterator:inc()
  if self.cur < self.max then
    self.cur = self.cur + 1
  end
end
```

Now, `miter:inc()` will not exceed the specified `max`. Under the hood, this function definition causes `MaxedIterator` to have an `inc` field, so when `inc` is called on a `MaxedIterator` (say `miter`), the interpreter won't find `inc` on `miter`, but will take advantage of `miter`'s metatable to look in `MaxedIterator`, finding the refined definition of `inc`.

We will again turn our attention to Typed Lua. Here, we will redefine our `Iterator` example with types. The following code snippet shows an interface declaration in Typed Lua.

```
interface Iterator
  cur : number
  const inc : () => ()
end
```

In Typed Lua, the `=>` syntax is syntactic sugar for defining a function with a first parameter named `self` with the type of the enclosing structure (here, `Iterator`). An equivalent formulation of `inc` is `const inc : (self : Interator) -> ()`. To construct the `inc` method, we write:

```
const function Iterator:inc()
  self.cur = self.cur + 1
end
```

The `const` annotation both here and in the `Iterator` interface state that the method cannot be changed. Finally, the constructor for `Iterator` is not unusual, and Typed Lua's type inference takes care of determining the return type:

```
const function Iterator:new()
  local iter = setmetatable({}, {__index = self})
  iter.cur = 0
  return iter
end
```

The above code looks slightly different than the constructor code in plain Lua, but it is equally valid (in that you could write either in both Typed Lua and Lua).

Typed Lua also supports inheritance, in a similar manner to Lua. Consider:

```
MaxedIterator = Iterator:new()
MaxedIterator.max = 0

const function MaxedIterator:new(max: value)
  local miter = setmetatable(Iterator:new(),
                             {__index = self})
  miter.max = tonumber(max)
  return miter
end
```

Unfortunately, the overridden constructor (here, `MaxedIterator:new`) must be a subtype of its supertype's constructor, so the type of passed values must be very permissive. In order to consolidate all of the types in constructors, Typed Lua treats `setmetatable` with some care: In `setmetatable(t1, t2)`, if the type T_1 of `t1` is a supertype of the type T_i of `t2`'s `__index` field, then it changes the type of `t1` to T_i, which (in the above constructor) has the effect of changing the type of `miter` to `MaxedIterator`, thereby allowing the assignment to `miter.max`.

Note that Typed Lua does not have a polymorphic type system, so programmers must explicitly call `setmetatable` in constructors—some existing Lua libraries do this, but Typed Lua has not yet managed to hide these calls behind more pleasant abstractions. Further, Typed Lua cannot type the use of metatables for operator overloading; for example, if two `Points` `p1` and `p2` are added like `p1 + p2`, Typed Lua is incapable of typing the result, and will not know that a `Point` should result from this operation.

One important thing to note here is that none of these fancy language features are impossible to replicate with other, simpler ones. In addition to colon in function declarations being syntactic sugar, the `__index` table member's lookup redirection is nothing more than a sequence of accesses:

```
local i = Iterator:new()
i.cur -- 0, since Iterator.cur = 0
getmetatable(i).cur -- also 0 as i's metatable is Iterator
```

In FWLua, provided that we store the metatable in a `_metatable` field, one could write:

```
rawget(rawget(rawget(_local, "i"), _metatable), "cur")
```

Metatables merely provide information to the Lua compiler about to handle certain operations when the first interpretation fails, and in fact in old versions of Lua metatables were known as *fallbacks*. With proper awareness by a programmer, metatables become more of a convenience than a necessity, and so are a form of syntactic sugar. While greatly increasing the usability of the language, and sometimes easing language analysis, they are in no way core to the language itself.

2.3.1 Peculiarities and Semantics

In developing our formal specification of Poseidon Lua, we base our formalization of Lua on the minimal calculus of FWLua, which allows us to focus on the the C FFI without getting bogged down in the semantic details of Lua. The semantics of FWLua is a *big-step*

semantics, as opposed to a more traditional *small-step semantics*. In small-step semantics, expressions take one small reduction step at a time, and in big-step semantics, an entire expression (sub-expressions included) is reduced all in one big step.

The following illustrates the differences:

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2 \qquad numeric(v_1) \qquad numeric(v_2) \qquad v = v_1 + v_2}{e_1 + e_2 \Rightarrow v} \text{ (BS\_ADD)}$$

In this big-step rule, the expression $e_1 \oplus e_2$ takes one big step to v, provided that both e_1 and e_2 step to numbers. In small-step, we would write:

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2} \text{ (SS\_ADD\_1)} \qquad \frac{value(v_1) \qquad e_2 \to e_2'}{v_1 + e_2 \to v_1 + e_2'} \text{ (SS\_ADD\_2)}$$

$$\frac{numeric(v_1) \qquad numeric(v_2) \qquad v_3 = v_1 + v_2}{v_1 + v_2 \to v_3} \text{ (SS\_ADD\_DO)}$$

Here, both sub-expressions step through to values before the expression steps as a whole. Also, we can encode an evaluation order with small-step, as it's clear here that e_1 steps through to a value before e_2.

Big-step semantics is convenient from a human point-of-view, as a reader can glean most important information from the rule. The granular transitions of small-step semantics are largely uninformative for most readers, as the details of the mechanisms of the small reductions are not very interesting, and yet those details are absolutely crucial when mechanizing a formal semantics for such a system. As such, FWLua's big-step semantics proved inadequate, as mechanization demands the utmost attention to detail, and we developed a small-step semantics based on it.

FWLua also defines a number of constructs to achieve normal Lua behaviour. For example, it desugars variables into table read and write to special "scope tables" (which approximate scope at runtime), and defines functions for variable access and update, which leverage the runtime table store. This is perfectly reasonable, but these constructs are particularly complicated to reason about in mechanized proof assistants.

Following in the footsteps of gradual type systems, we set out to specify both a *typed* and *untyped* language, and have the typed language "compile" into the untyped language

through our typing judgment. It was important to us to have variables be fully-fledged members of our typed language, and we faced a number of challenges when trying to compile these variables into the untyped language (where they did not exist). For example, a simple variable access in the typed language transformed into a complex set of functions in the untyped language, relying on the presence of the aforementioned scope tables in the runtime environment. This was immensely complicated, especially given that we aren't particularly interested in the untyped-typed language relationship, and want to focus on interoperation with C. As such, we decided to promote variables to be first-class members of the runtime language as well as the typed language, and did away with a large amount of unnecessary complexity.

In summary, Poseidon Lua uses Typed Lua's type system, with the addition of C types through the Lua pointer type, discussed in more detail in Section 2.5. We will be formalizing a modification of FWLua, extending it with Typed Lua's type system, and adding C calls, casts, data allocation, and C store read and write. Our intention is to prove conditional type soundness of Poseidon Lua, as well as show that C code is always at fault when runtime errors occur in otherwise sound Typed Lua. This shares some notable similarities with results from gradual typing, which is discussed next.

2.4 Optional and Gradual Typing

The desire to transition from dynamic to static typing has led to the development of optional and gradual typing, with which users can choose to add type annotations if, when, and where they please. This means that the translation from untyped to typed code becomes as simple as adding type annotations to untyped code, and removes any sources of error due to low-level language incompatibilities [2]. It also means that partially typed code can still run, which allows the code to be used during the entire translation process (it is no longer "all-or-nothing" in terms of typing).

The distinction between optional and gradual typing lies in their runtime soundness guarantees. In optional typing, fully typed sections of the code are typechecked at compile time; but, if the dynamic type `any` is used, then no typechecks are performed. The result is that the execution is as fast as the plain dynamic code, but there are no runtime type guarantees. In contrast, gradual type systems ensure runtime soundness by inserting type checks at the boundaries between typed and untyped code. This means that any type errors not caused by the static compile-time type checking (i.e. type errors where there were no type annotations) will be caught at runtime as ill-typed values enter the typed sections of code. The downside of the soundness guarantee is that the added runtime

checks add significant overhead to the program execution, resulting in a slowdown which can be prohibitive [27].

A formal representation of the typing guarantees of gradual typing defines it as a fully qualified typing paradigm. The *gradual guarantee*, formalized by J. Siek and coauthors [25], lists a number of requirements for a gradual type system to satisfy. Essentially, this guarantee states that "changes to the annotations of a gradually typed program should not change the static or dynamic behavior of the program". In other words, a well-typed program should evaluate to the same result independent of the number of (correct) type annotations present in the program. This is fundamental to the idea of gradual typing: the actual functionality of the program remains unchanged, with the only effects being on the type checking, in the event of a program without any static or dynamic type errors.

Another crucial result in gradual typing is the statement that "well-typed programs can't be blamed" [29]. P. Wadler and R. B. Findler adapt the concept of *blame* for contracts for higher-order functions [6]: A contract is merely a requirement that must be fulfilled, and authors define a framework in which contract violations could be *blamed* on some code [6]. For example, a function `fun` might have a contract asserting that its argument is a `string` with a particular length, say `n`, and returns a single-digit integer. If `fun` was called with a `string` of length $\neq$ `n`, then the caller would be blamed for incorrectly using `fun`, whereas if `fun` was found not to produce a single-digit integer, it would itself be blamed for not fulfilling its own contract.

In showing that well-typed programs can't be blamed, P. Wadler and R. B. Findler apply blame from contracts to the casts in gradual typing [29]. An important result from their work is a simple proof that, if a gradually-typed program goes awry, then blame always lies with the untyped code. This result allows us to make a meaningful statement of soundness for gradual type systems, even in the presence of dynamic code.

Our ultimate goal is to apply the tools and techniques of the formal analysis of gradual typing to FFIs. The gradual guarantee doesn't quite apply to FFIs, since it deals with removing annotations from annotated code (thus making it unannotated), and there is a direct and easy transformation between these two languages, which is not the case for FFIs (as there is no clear transformation between C and Typed Lua). C is a more expressive language than Lua, and it is unclear how one would convert arbitrary C code into Lua code.

We might not have an equivalent to the gradual guarantee, but we may certainly adapt some concepts from gradual typing for our formal treatment. The concept of blame, for instance, is particularly interesting to us. In the realm of gradual typing, we guarantee that untyped code is always to blame for runtime type errors; while in the realm of FFIs,

when a language with stronger guarantees calls function written in a less safe language, we can argue that errors exclusive to the guest language can always be blamed on guest language code. Specifically in Poseidon Lua, we show that C code is always to blame for errors in well-typed Typed Lua, in an adaptation of the statement "well-typed programs can't be blamed". This is taken up in Chapter 5.

2.5 Poseidon Lua

Poseidon Lua is implemented as a set of extensions to Lua 5.3.3 [15] and Typed Lua [17]. Lua is extended with:

- a dynamic type to hold an opaque, unmanaged (non-garbage-collected) pointer,
- intrinsics to read and write scalar values from offsets through such opaque pointers, converting them to or from Lua scalar types in the process, and
- modifications to Lua's existing FFI to call C functions using these opaque pointers rather than its existing wrapped types.

These extensions are not intended to be used directly, and provide no correctness or safety guarantees if used without types. Instead, Typed Lua is also extended, with syntax for declaring C `struct`s, and a type constructor `ptr` which declares a variable, parameter or field as holding an unmanaged pointer of a given C type. Accesses to members of such a value are compiled by Typed Lua into the added Lua intrinsics. The goal was not to allow all C-like behavior in Lua, but to allow faster interoperability between unsafe C code and safe Lua code, so many unsafe behaviors are not supported; for instance, arrays must be explicitly declared as such and allocated to the correct size. Casting, null pointers and deallocation are allowed, as eliding them would too severely limit the combined behavior. Dereferencing a null pointer can be configured to throw a Lua exception—checking for null before dereferencing—or to raise a traditional operating-system-level error such as a segmentation fault. Dereferencing bad pointers with values other than null will also raise a fault, but such bad accesses are always the fault of C code or an incorrect downcast or deallocation. That is, in the absence of incorrect downcasts and deallocations, the Poseidon Lua code itself accesses C data correctly; type safety errors can be blamed on C.

This implementation style allows Poseidon Lua to gain performance over standard FFIs in two ways. First, standard FFIs only allow access to objects through *wrapper objects*, which are objects allocated in the host language—in this case, Lua—which wrap

the foreign language—in this case, C—objects and mimic a host-language interface. While this technique is extremely flexible, it has a major downside: Each object *accessed* in C requires an object to be *allocated* in Lua. Without extremely careful optimization, multiple wrapper objects can be allocated for a single C object. Poseidon Lua avoids any wrapper objects by holding C pointers directly.

This has the immediate consequence of needing to include at least some part of C's type system as a subset of Lua's. Usually, FFIs don't include guest language types, and the wrappers manage the translation from guest language code (and thus from the guest language's type system into the host language's). In fact, the inclusion of guest language types in the host language's type system goes hand-in-hand with the aforementioned optimization, and without it, Poseidon Lua's performance gain would not be possible: By directly holding C pointers, the system is faster *and* the type systems merge to some degree.

Second, by avoiding this allocation, Poseidon Lua's garbage collector avoids tracing these wrappers. In principle, this is also an advantage over plain Lua or Typed Lua, but in practice, the first benefit is far more significant.

The implementation's performance was tested on a small selection of benchmarks (binary-trees, n-body, spectral-norm and fannkuch-redux) chosen because they access data in structures; that is, they're not purely numerical benchmarks and have structures allocated at runtime. In each benchmark, the core data types were adapted to C types, and the code was adapted to use either Lua's existing FFI to access those data types or Poseidon Lua. That is, no actual C code was added, so our comparison is Lua against Lua, but C data types were added. The performance difference of Poseidon Lua compared to plain Lua with no C interfacing at all was not statistically significant, as Lua's existing optimizations for accessing its own data types had largely the same performance benefits as using direct memory access, and the benchmarks were likely not long enough to see a significant benefit from reduced tracing. On the other hand, Poseidon Lua had an average speedup of 7.8× over Lua's FFI, by avoiding needless run-time allocation.

This thesis focuses on the design and soundness guarantees of the programming language. The formal guarantees do not depend on this particular implementation, which will be presented separately. Were Poseidon Lua adapted to compile to Lua's existing FFI instead of its own intrinsics, the behavior and correctness would be identical, albeit slower. Although we focus our formal semantics on Lua, we believe that the concepts generalize to foreign function interfaces in many other programming languages.

Chapter 3

Nondeterminism and C at a Glance

In this thesis, we aim to demonstrate that the tools and techniques used to reason about gradual typing can be applied to reasoning about FFIs. While we will present the formal semantics for Poseidon Lua, our results hold for *any* choice of language combination. In fact, in the spirit of being general, we do not model C's semantics. While possible, we feel that a full model of C would detract from our purpose, as the semantics would be enormously complicated. Instead, we will account for all possible semantics for C, which leads us to a nondeterministic semantics, discussed further in Section 3.1. We focus less on the mechanics of each language, and more on qualifying the *effect* that the guest language has on the system. This effect is not intrinsic to the choice of languages, but rather to the safety differential between them.

The nondeterministic semantics that we are left with may be a byproduct of our modeling (or lack thereof) of C, but we believe that it is itself interesting. For our purposes, the *interaction* of C and Lua is interesting, and not either language independent of the other; if a full model of C was important for a particular formal spec, one could plug in their choice of C semantics in for our black box version and have a more precise picture of its execution. That said, whatever one chooses as a C semantics will simply represent choosing *one* of the infinite semantics encompassed by our nondeterministic model. In carefully accounting for all semantics, one can reason simultaneously about *all* such semantics, provided that the level of abstraction is satisfactory.

Another advantage of nondeterminism is that it allows one to be implementation-agnostic: Many languages have defined operational semantics which describe some manner of execution, but at the end of the day it's the implementation that actually drives the program. For example, CompCert [3] is a formal description of the semantics of C as

realized by *its compiler*, and one should not expect all of CompCert's assertions to hold for C programs compiled with a different compiler. In short, when the effect of an execution is the subject of interest, rather than the execution itself, a nondeterministic semantics such as the one presented in this thesis allows one to reason independently of language implementations.

In order to showcase blame and nondeterminism as tools for analyzing FFIs, formalizing Poseidon Lua is a good choice for a few reasons. First, calling C functions from other code (including Lua) occurs frequently, and so having C as a guest language is realistic. Second, Typed Lua is a far safer language than C, which throws the unsafe behaviour of C into the spotlight: In Typed Lua (and indeed Lua), very few runtime errors are possible, in stark contrast to C. Finally, C can be a pretty destructive language, which requires us to account for a wide variety of behaviour. Poseidon Lua also has a number of interesting properties besides making C calls, as it allows explicit downcasting of pointers as well as allocation of C data.

C's direct memory manipulation and lack of memory management have given it a reputation for being both lightning fast and notoriously unsafe—for example, C pointer arithmetic is unsafe, and one way that C can gain performance over other languages. Its power and speed lead many programmers to write performance-critical code in C, and find ways to incorporate C into their other environments. As one would expect, calling a C function from any other language invites C's undefined behavior into the environment, and certainly has ramifications on any existing guarantees. This is not dissimilar to gradually typed languages, where `any`-typed values flowing into typed code reintroduce the possibility of runtime type errors. We wish to work with and adapt the notion of blame to FFIs, and make a similar statement to that in [29], showing that C is always to blame for runtime failures, no matter where they occur.

In gradual type systems, blame is introduced at cast sites, and carried with the untyped value as it moves through statically-typed code. In a sense, the act of accepting an untyped value into typed code introduces the possibility of runtime type errors, a fact mirrored in systems using a C FFI, where all bets are off as soon as a C function is called. If we could assign blame at these call sites and propagate it appropriately, we would be able to track their effects and trace any issues that arise because of them back to the offending call.

To illustrate, consider a C function call: Without knowledge of the function's code, we cannot guarantee that the function will even succeed, let alone consider what value is returned or its influence on the heap. And even if the function succeeds, we should be skeptical of any future references to memory that C might have had access to, as it's possible that the call freed or corrupted all memory. With a full model of C, we could get

a handle on this behaviour, but fully modelling C is both challenging (see CompCert [3]) and distracting from our purpose to focus on its effects. Without such a model, we are left with a nondeterministic semantics, which we discuss in more detail next.

3.1 Nondeterminism

Our choice not to model C's semantics is a double-edged sword: On the one hand, our formalism and statements are more general, but on the other hand we can't say exactly what a C function actually does, and have no choice but to account for all possibilities. This amounts to a *nondeterministic* semantics; performing a C call steps nondeterministically to every state that the C call could have produced, including every possible change to the heap and, of course, failure. It is possible for a C function to *never* return; if this is the case, then we have nothing to say about Lua anymore, since no Lua code will be executing.

While we have said that we account for all possible executions of C, our model does make some assumptions. For one, we assume that C does not touch Lua's memory, and that its effects are contained to an explicitly defined "C store". This mirrors reality in most other FFIs, where guest code and data is not aware of host code and data. 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 or mutate 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 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 too complex for far too little gain. 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.

Ultimately, we wish to encompass both C function executions which altogether fail and crash the entire program, for instance by dereferencing a null pointer, as well as executions which succeed. To represent this in our formalization, calling a C function steps nondeterministically to either failure (representing an execution of C that crashed) or to some value (representing a successful call). That said, C can do a lot more damage than just dereferencing an invalid pointer: With irresponsible address arithmetic, it's possible that a C function can tamper with some amount of memory that it shares with the host program, thereby introducing an insidious source of uncertainty into later superficially safe memory accesses. To account for this in our formalization, we make C function calls tag

the entirety of shared memory with *blame*, indicating that a value held at an address may have been altered by the call. This shared memory is known as the *C store* (also known as σ_C), which is a list of C values, C types, and optional blame information. Though unusual, the storage of a type at a particular location in σ_C is used to signify what type that the location is *intended* to have; this is important for downcasting, which we will explore later. Locations in σ_C are address-flavoured, and our semantics perform some amount of "pointer arithmetic" under the hood, which will also be discussed later. Note that the blame tagging and tracking described herein would never be done at runtime, as the cost would be enormous.

The presence of blame at a location complicates the next access to that location, and the idea is to capture that accessing that location should be nondeterministic. To represent that an address is essentially a wildcard after a C call, C store reading and writing exhibits nondeterministic behaviour if the location is tagged with blame. As with C calls, accessing blamed locations in the C store steps nondeterministically, though there aren't quite as many states to quantify. First, the access might fail (for example, we may be dereferencing a null pointer), and so failure is one option. Next, an access may succeed, and if it does, we are guaranteed that subsequent accesses will also succeed (of course, until the next C call): All of Lua's effects are accounted for in its semantics, and C activities which might affect the location will tag it with blame. To model this so-called reclamation of determinism, we "erase" blame from C store locations which are successfully accessed, thereby causing them to behave deterministically from then on, until they are newly tagged with blame. Of course, this hinges on our assumption of no multi-threading.

To illustrate, consider the following code snippet:

```
void something(Point p) {...}
...
p : Point = calloc Point
something(p) -- a C function call
print(p.x)
```

Without semantics for the `something` function, we enter a state of nondeterminism. The call to `something` can either fail and crash our entire program, or it can succeed and return, whereby we can only assume that the entire C store is tainted with blame from the call. Then, when accessing `p.x`, we are faced with two possibilities. First, the access may succeed, thus erasing blame on that heap location; the value has been observed, and so we are sure of its behavior from then on. Alternatively, the access may fail for whatever reason, at which point program execution terminates.

Now, imagine that we had something of a "micro-semantics" for C, in which we allowed

simple memory-management operations. In this situation, we would be able to model the execution of `something`. Imagine that `something` was defined as:

```
void something(Point p) {
  free(p);
}
```

In this case, the execution of the previous code snippet would fail with a segmentation fault when attempting to access `p.x`. Another possible definition of `something` might look like:

```
void something(Point p) {
  p.x = 2;
}
```

In this case, the code snippet would execute without issue. The crucial point to note, here, is that our choice of nondeterministic semantics accounts for both of these possibilities. The situation where `p` is freed corresponds to the C call working, and the access failing, while the situation where `p` is properly updated corresponds to both the C call and the access succeeding.

In choosing to leave C as a black box, we have no choice but to account for all possible semantics for the inner workings of C (within our assumptions). A direct consequence of this is that our proofs are more general than equivalents in a system with a full-fledged C semantics, and our results would not be much stronger by fully realizing C. Blame gives us all we could possibly want from C, as it allows us to quantify the effects of the language without being caught up in the minutia of its execution.

Our system is nondeterministic by design, and many operational semantics, at least those which do not model concurrency, rely at least in part on some form of determinism, but make no attempt to prove that their system is in fact deterministic. As we will see in the next chapter, it's enough to have two reduction rules with premises which can possibly match for a semantics to be nondeterministic. In these situations, proofs of type soundness and other results are in some sense "weakened", since it's possible that one of the reduction rules has a more easily satisfied condition and thus may always be selected over the more restrictive rule.

Besides C calls, Poseidon Lua allows programmers to explicitly downcast C values, which we discuss next.

3.1.1 Downcasts

In C, the following is valid, though it is undefined behaviour:

```
char *q_ptr, q;
q = 'q';
q_ptr = &q;
printf("%d that's no q...\n", *(int *)q_ptr);
```

For those unfamiliar with C, this print statement displays garbage. Basically, space for a character pointer `p_ptr` and a character `q` is reserved. Then, we write the character `'q'` into `q`, and make `q_ptr` point to the address of `q`. Then, in the print statement, we dereference `q_ptr` *as an integer pointer*, thereby indicating that we intend to read the bits `q_ptr` refers to as an integer. This type checks in C: As long as you're not dereferencing protected or freed memory, you can interpret the bits as you please with casts such as the above. A type in C merely indicates to the compiler how the bits at the memory location should be treated, and generally speaking there are no type guarantees beyond that "the value will have that type". The above is undefined behaviour in C, and this for good reason.

Unlike other language compositions, Poseidon Lua allows for direct referencing of C data through the Lua pointer, and additionally allows for the downcasting of C values. Thus, our formalization must support this downcasting, but this is another source of nondeterminism, since all bets are off as to what the value contained at the memory location actually is following a cast. To achieve this, the C store in our formal specification also contains type information for each of its locations. With this information, the semantics of a cast are rather straightforward: Simply change the type in the store (indicating what type the value is intended to have), and tag the location with blame.

As before, the immediately succeeding access to the blamed location is nondeterministic. If a subsequent access of a cast value succeeds, the newly observed value is stored at that location and the blame is erased.

3.1.2 Allocating C Data

The following code snippet may crash in C, depending on the memory model:

```
int ** p;
p = (int **)calloc(1, sizeof(int *));
printf("%d\n", **p);
```

Here, a pointer-to-a-pointer `p` is allocated, with a default value of `0`. In the print statement, the initial dereference of `p` yields a pointer to an int, and unfortunately the allocation cannot guarantee that `*p` points to an int. In fact, the default value for `*p` (as determined by calloc) is the address `0`, an address which causes a segmentation fault once dereferenced.

In Poseidon Lua, all C values are behind a Lua pointer, so the use of the pointer-to-a-pointer `**p` isn't all that unrealistic an analogy. `calloc` initializes allocated memory with default values. For integers or other numeric types, this is perfectly reasonable and well-defined. However, when allocating pointers, we tread in dangerous waters: The allocated location is initialized with `0`, and dereferencing that location will immediately trigger a segmentation fault.

Consider instead the following:

```
int ** p;
int * a_ptr;
int a;
p = (int **) calloc(1, sizeof(int *));
a = 5;
a_ptr = & a;
p = & a_ptr;
printf("%d\n", **p);
```

Here, we assign a known valid value to the allocated location, causing the dereference in the print statement to succeed. In allocating an `int`, saving a reference to it in `a_ptr`, and saving a reference *to that reference* in `p`, we fully initialize the double pointer. This way, we are sure of the contents of each of `p`, `*p`, and `**p`, and shouldn't run into any errors (indeed, the code snippet executes and prints `5`).

Poseidon Lua allows the allocation of C values, and allocating pointers must be yet another source of nondeterminism. In the language, every C value is wrapped in a Lua pointer, so a statement allocating a C value is really allocataing a *pointer* to a C value. Since C pointers (in addition to Lua pointers) are allowed, we must pay close attention to the allocation of C pointers, since such an allocation describes a situation much like the above snippet.

In summary, we have outlined how we plan to quantify C's behaviour while maintaining a minimal model of its semantics. As we mentioned, we will be dealing with a nondeterministic semantics, where direct C calls, casts, and deallocations introduce the possibility for immediate or future runtime failure. The formal specification of our system will be fully revealed in the next chapter.

Chapter 4

Formalizing Poseidon

In this thesis, we are exploring the theoretical aspects of language mixing with techniques borrowed from the formal treatment of gradual typing, and this chapter will detail our formal specification. Due in part to its semantic unwieldiness, and to our desire to stay as general as possible, we chose not to model C beyond the effect that it has on the host language, Typed Lua. This gives us a nondeterministic semantics, which we found to be atypical (except for models of concurrent systems) in our survey of the literature, and interesting from the perspective of formal reasoning.

Essentially, we are providing the formal specification for a system with black-box function calls with potentially far reaching effects. Further, Poseidon Lua also allows programmers to downcast pointers, allocate C data, and seamlessly integrate C and Lua values. To capture all of this, our formalism carries blame at runtime, stores type information alongside values in the C store, and expressions nondeterministically reduce in the face of blame. In this chapter, we will show how these features translate into a full formal semantics of Poseidon Lua, highlighting interesting and unusual aspects and justifying our choices.

4.1 Overview

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. We will need to formalize our host language, Typed Lua, and also the ways in which it interacts with C.

Our formalization of Lua is based on FWLua [12], and we adapted their big-step semantics to a more standard small-step equivalent. In order to mechanize our formalization, some simplifying additions were required, namely the promotion of variables from syntactic sugar to full-fledged language members. FWLua syntax for table construction is incremental, starting with an empty table and adding desired fields. Unfortunately, this approach is not viable when dealing with *typed* tables; for example, initializing a table with a classic `Point` type to the empty table should not type, since the empty table is not a subtype of a table with any members. To address this, we also include table literals, allowing programmers to properly initialize non-empty tables.

Poseidon Lua requires that FFI calls be made only from well-typed code, and so we adapted the type system of Typed Lua [17], with some modifications made possible by our simplified semantics for Lua. One such omission is their so-called *second-class* types, which do not correspond to actual Lua values. For instance, our function type is greatly simplified by our restriction that functions always consume and produce a single value.

An additional property of our formalization is that we make a distinction between a *typed* and *untyped* language: The typed language is the language that is written by the user, and the untyped language is the language that actually reduces at runtime. This is in line with some gradually and optionally typed languages; for example, in TypeScript, programmers write annotated TypeScript code which is compiled into JavaScript which is then run.

Instead of keeping C and Lua values separate, Poseidon Lua allows for both to exist in Lua code and interact seamlessly. This necessitates the inclusion of C types into the type system, which is accomplished through a "Lua pointer" type (which always points to a C value). For the purpose of our formalization, the C type system is quite small, limited to integer, pointer, function, and structure types; we feel that this minimal type system allows us to describe all of the strange behavior one would expect from C.

To drive this memory functionality in our semantics, we need to carry a number of stores at runtime. On the Lua side, we have a *variable store* which maps variables to values, and variable declarations and reassignments may modify the store's contents. We also have a *table store* to store Lua table structures, and this store is indexed by a *register*, which is nothing more than a label into the table store. In Lua, tables may themselves be indexed by most primitive types, but for simplicity we only allow string indexing of tables. Keeping the variable and table stores separate allows us to achieve table aliasing; for example, in setting a variable `x` to be some table, the variable store at index `x` will contain a register pointing to the table, and that register can be copied to other variables, whereby they will all refer to the same underlying table.

As for the C store, our set of allowed C operations requires us to add a bit more machinery to it to meet our needs. Namely, downcasting necessitates that we store the *expected type* of a C value in the C store, so that casting a location can have a meaningful semantics. Further, we need to be able to tag C values with blame to indicate when nondeterminism is necessary. Taken together, we land on a C store which is a list of (C value, C type, optional blame) triples. Locations in the C store are address-flavored by design: C structs are not first-class inhabitants of the store, instead their members are laid out contiguously, and field access is achieved by computing the offset for the specified member.

As is standard, typing locations in Lua and C stores requires store typings to describe them, which are part of the typing context. To type expressions, we will define a typing judgment, which we sometimes refer to as the *type transformation* on account of its role in transforming the typed language into the untyped runtime language. The type transformation makes use of a typing context made up of store typings, describing the types at every location in each of the runtime stores, and a typing environment, describing the type of each defined variable.

For the remainder of this chapter, we will discuss the component parts of our formalization in more detail. We begin our in-depth discussion of Poseidon Lua with our type system.

4.2 Types

Our type system mostly matches that described in Typed Lua [17], with a few notable alterations. For instance, our function type only has a single argument type; this, of course, is not the case in Lua, but we curry multivariate functions to repeated application of single variable functions, by which a single argument function type suffices. We discuss this in more detail in the following section. In fleshing out this type system for our core calculus, we found no need for Typed Lua's type variables, recursive types, or projection types. Further, to simplify reasoning about Lua, we only allow string indexing in tables. Our types are given below.

$$
\begin{array}{rcll}
T & ::= & \textbf{nil} & \textit{nil type} \\
 & | & \textbf{value} & \textit{top type} \\
 & | & \textbf{ref } T & \textit{reference type} \\
 & | & T_1 \cup T_2 & \textit{union type} \\
 & | & L & \textit{literal type} \\
 & | & B & \textit{base type} \\
 & | & T_1 \rightarrow_L T_2 & \textit{function type} \\
 & | & \{f_1, ..., f_n\} & \textit{table type} \\
 & | & \textbf{ptr}_\textbf{L}\ T_C & \textit{C type} \\
T_C & ::= & \textbf{int} & \textit{C integer type} \\
 & | & T_C^1 \rightarrow_C T_C^2 & \textit{C function type} \\
 & | & \textbf{ptr}_\textbf{C}\ T_C & \textit{C pointer type} \\
 & | & \{s_1 : T_C^1, ..., s_n : T_C^n\} & \textit{C struct type}
\end{array}
$$

$$
\begin{array}{rcll}
f & ::= & s : T & \textit{fields} \\
 & | & \textbf{const } s : T & \textit{const fields} \\
L & ::= & < booleans > & \textit{literals} \\
 & | & < numbers > & \\
 & | & < strings > & \\
B & ::= & \textbf{boolean} & \textit{base types} \\
 & | & \textbf{number} & \\
 & | & \textbf{string} &
\end{array}
$$

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 subtypes of their corresponding base types. As usual, function types are contravariant in their argument types, and covariant in their return types. Regarding tables, their types have width subtyping: A table type T is a supertype of a table type T' which has *at least* all of the fields of T. In other words, adding extra fields preserves the subtyping relationship. Finally, 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' is identical to 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.

Poseidon Lua allows Lua to interface with C through expressions with the "Lua pointer" type $\textbf{ptr}_\textbf{L}\ T_C$; Lua is only ever dealing with *pointers* to C values, and not C values themselves, and 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. In some sense, C is "plugged" in to Lua through the $\textbf{ptr}_\textbf{L}\ T_C$ type.

Even though we don't model C, we still include parameter and return types for C functions to ensure that they are called with correctly-typed arguments (indeed, FFIs typically export function types as part of their API). We also distinguish between Lua and C pointers to allow for pointers to exist in C's type system (without needing to resort to the Lua pointer construct). The C struct type is also as expected, though in our formalization we restrict structs to only having non-struct members (pointers to structs are fine), as a struct with struct members can be easily rewritten to use a single struct.

4.3 Language

For the purpose of our formalization, we present a core calculus akin to FWLua. In this section, we will discuss the language of expressions, both typed and untyped, before moving on to the typing judgement and reduction relation.

The language of *untyped expressions* E, also known as the languge 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. Certain expressions, such as *location update*, are not directly written by the programmer (indeed, you cannot directly reference memory in Lua). To address this, we additionally make the distinction between a *user language* and an *intermediate language*, since being able to type all possible runtime expressions is crucial for any type soundness result. The user language is all of the typed language except location update, Lua dereference, and Lua location.

We will consider typed expressions and their untyped counterparts in tandem, presenting both together and explaining the unusual expressions. We start with values.

$$
\begin{array}{rcll}
v & ::= & \mathbf{nil_L} & \textit{nil value} \\
 & | & r & \textit{register} \\
 & | & c & \textit{constant} \\
 & | & \mathbf{loc}\, n & \textit{Lua store location} \\
 & | & \mathbf{ptr_L}\, n\, T_C & \textit{C store pointer} \\
 & | & \ldots & \\
v_C & ::= & \mathbf{ptr_C}\, n & \textit{C store pointer} \\
 & | & n & \textit{C number literal}
\end{array}
$$

These values are the same in the typed and untyped languages. On the Lua side, we have the following values: $\mathbf{nil}_L$ is Lua's nil value, r is a *register* (i.e. location) in the Lua table store, c is a constant (either numbers, strings, or booleans), $\mathbf{loc}\, n$ is a location in the *variable* store, and Lua pointer $\mathbf{ptr}_L\, n\, T_C$. We included Lua store locations to allow for variable mutation: The variables are essentially stored in the variable store, and updated or retrieved as required. Tables exist in a separate store, the table store, and registers refer to these locations. For example, if we wanted a variable x to refer to a table, x's location in the variable store would hold a register identifying the location of the table in the table store. Functions are also values, and we discuss them in more detail shortly.

The Lua pointer expression $\mathbf{ptr}_L\ n\ T_C$ is essentially a pointer into the C store, pointing to location n, and expecting it to have type T_C. While C values only exist in Lua through

this expression, they do inhabit the C store and must be coerced to Lua (or vice versa, depending on the circumstance). They are the C integer (just a number, nothing special) and C pointer $\mathbf{ptr}_C\, n\, T_C$ expressions. The C pointer expression is notably similar to Lua's, but they differ in that they have different types (in truth, the C pointer cannot even be written in the language, but the flavour of both expressions is similar).

We also formalize binary operations, though there is little interesting to see. For the sake of completion, we give it below:

$$\begin{array}{rcll} te & ::= & te_1\ op\ te_2 & \textit{binary operation} \\ & | & ... & \end{array} \qquad \begin{array}{rcll} e & ::= & e_1\ op\ e_2 & \textit{binary operation} \\ & | & ... & \end{array}$$

In $te_1\ op\ te_2$, the first and second expressions will need to have the correct types for the specified operator. This is echoed in the runtime language, where both expressions in $e_1\ op\ e_2$, once reduced to a value, need to have the correct type for the operator.

The allowed binary operators are:

$$\begin{array}{rcll} op & ::= & +, -, *, / & \textit{arithmetic} \\ & | & \leq, <, \geq, > & \textit{order} \\ & | & \wedge, \vee & \textit{boolean} \\ & | & .. & \textit{concatenation} \\ & | & == & \textit{equality} \end{array}$$

Next, we consider function expressions, which are also values in Poseidon Lua. In the typed language, C and Lua functions are separate expressions, but calling functions is done through one common expression.

$$\begin{array}{rcll} v_t & ::= & \lambda x : T.te & \textit{Lua function} \\ & | & \mathbf{cfun}\ T_C\ n & \textit{C function} \\ & | & ... & \end{array} \qquad \begin{array}{rcll} v & | & \lambda x.e & \textit{Lua function} \\ & | & \mathbf{cfun}\ n & \textit{C function} \\ & | & ... & \end{array}$$

Note that functions are values in Lua (and indeed in Poseidon Lua). Here, we have $\lambda x : T.te$ describing a Lua function abstraction, and $\mathbf{cfun}\ T_C\ n$ describes a C function. In the C function expression, T_C is the function type $T_{Arg} \rightarrow_C T_{Return}$. The type information is required by the type transformation to type these functions, as we cannot leverage the function body (as is the case with traditional functions). Untyped language equivalents are straightforward, with the removal of the now unnecessary type information.

Next, we turn our attention to variables. The location value $\mathbf{loc}\, n$ describes how variable locations can be referred to, but in truth it arises as an *intermediate* expression when

variables are substituted with their locations, a by-product of program execution. Consider the following expressions:

$$
\begin{array}{rcll}
te & ::= & \textbf{let}\, x : T := te_1\, \textbf{in}\, te_2 & \\
& | & x := te & \\
& | & \textbf{loc}\, n := te & \\
& | & \textbf{deref}\, te & \\
& | & ... &
\end{array}
\qquad
\begin{array}{rcll}
e & ::= & \textbf{let}\, x := e_1\, \textbf{in}\, e_2 & \textit{let binding} \\
& | & x := e & \textit{variable update} \\
& | & \textbf{loc}\, n := e & \textit{location update} \\
& | & \textbf{deref}\, e & \textit{Lua dereference} \\
& | & ... &
\end{array}
$$

Variables are declared and initialized using the let binding expression, which has a fairly self-explanatory type annotation in the typed language. Once fully evaluated, e_1 will be stored in the variable store and its location will be substituted in for x in e_2, with suitable dereferences inserted automatically. Variable mutation is also allowed, and the update expressions allow for this: Variable update is the expression in the typed language which is written by the programmers, and substitution transforms it into the location update expression (if a variable appears in left-hand position). When in right-hand position, substitution transforms variables into dereference expressions, which simply access the specified location and return the contents of the variable store.

Unlike FWLua, we allow table literals in our calculus. Consider:

$$
\begin{array}{rcll}
te & ::= & \{s_1 = v_1, ..., s_n = v_n\} & \textit{table} \\
& | & ... &
\end{array}
\qquad
\begin{array}{rcll}
e & ::= & \{s_1 = v_1, ..., s_n = v_n\} & \textit{table} \\
& | & ... &
\end{array}
$$

The expression $\{s_1 = v_1, ..., s_n = v_n\}$ provides a list of name, value pairs. The idea here is that a table will be allocated, and the v_i will be written to the s_i members in the table store, and the expression will step to a register pointing to the newly constructed table. This register can then be saved into a variable, for example in a let binding.

We will digress for a moment to justify the promotion of table literals to first-class members of our calculus. A known issue with formalizations is the so-called "constructor problem", or "initialization problem": Essentially, records and objects of some type T must be constructed from *something*, and it's difficult for that something to have type T. We could initialize the record or object to be empty, but the empty type is surely not a subtype of T, and if we instead write a whole table of type T, then we can no longer represent recursive data structures without reassignment. Specifically regarding core calculi, many usually (and indeed in FWLua) disallow record construction, instead opting for empty record allocation followed by field initialization, which does not work in a typed language.

To illustrate, consider this small example where we initialize a table (with non-empty

type) to the empty table, which shows us how initialization to an empty table is a sure source of type violations:

```
let p : {x: num, y: num} := {} in
p.x := 42
p.y := 10
```

The issue with this code snippet is that the empty table {} is *not* a subtype of {`x: num, y: num`}; in fact, width subtyping tells us that the empty table type is a supertype of all table types. To deal with this, we chose to allow table literals in the language. With that, programmers may write:

```
let p : {x: num, y: num} := {x=0, y=0} in
p.x := 42
p.y := 10
```

Our system checks to see if the table literal expression (here, {`x=0, y=0`}) has each of the appropriate assignments.

With that out of the way, table read and write are treated next. The untyped language for these operations is a little unique, and is essentially the same language that Lua uses to interface with tables.

$$
\begin{array}{rcll}
te & ::= & te_1.te_2 & \textit{dot access} \\
 & | & te_1.te_2 := te_3 & \textit{dot update} \\
 & | & \ldots &
\end{array}
\qquad
\begin{array}{rcll}
e & ::= & \textbf{rawget}\, e_1\, e_2 & \textit{table select} \\
 & | & \textbf{rawset}\, e_1\, e_2\, e_3 & \textit{table update} \\
 & | & \ldots &
\end{array}
$$

In the typed language, programmers write the expected $te_1.te_2 := te_3$ and $te_1.te_2$ expressions for table member write and read. When te_1 is a Lua table (i.e. a register), the type transformation will generate a **rawget** or **rawset** expression. In both of these, e_1 is the table being written to/read from, e_2 is the index in that table, and in **rawset**, e_3 is the value being written.

The dot access and dot update typed expressions may also transform into C equivalents of **rawget** and **rawset**, called **cget** and **cset**. Consider:

$$
\begin{array}{rcll}
e & ::= & \textbf{cget}\, e\, n\, T_C & \textit{C store access} \\
 & | & \textbf{cset}\, e_1\, n\, e_2\, T_C & \textit{C store update} \\
 & | & \ldots &
\end{array}
$$

These expressions are intentionally analogous to **rawget** and **rawset**. 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 **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 in the C store).

Another instance of the type transformation differentiating between C and Lua is with function applications. Consider:

te	::=	$te_1(te_2)$	*function call*	e	::=	$e_1(e_2)$	*Lua function appl.*
	|	...			|	**ccall** $e_1\, e_2\, T_C$	*C function call*
					|	...	

In the typed language, programmers may write a function application as $te_1(te_2)$ as is standard. 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 $e_1(e_2)$ expression is straightforward, so let us focus on the C call: In **ccall** e_1 e_2 T_C, e_1 is the C function being called, e_2 is the argument to that function, and T_C is the function's type. The type is necessary since C calls exhibit nondeterministic behaviour, and we can leverage T_C to reason about the value that is returned from the function.

Poseidon Lua allows C allocation and downcasting. Consider:

te	::=	**calloc** $T_C\, n$	*C allocation*	e	::=	**calloc** $T_C\, n$	*C allocation*
	|	**ccast** $te\, T_C\, n$	*C downcast*		|	**ccast** $e\, T_C\, n$	*C downcast*
	|	...			|	...	

As one might expect, **calloc** T_C n allocates something of type T_C, and n is the identifier (blame) uniquely associated with the allocation, which allows a trace-back if a runtime error occurs. **ccast** te T_C n downcasts the pointer te to type T_C, and again n is a unique blame identifier associated with the cast.

In a departure from FWLua, we allow explicit expression sequencing in our calculus. Consider:

te	::=	$te_1;\ te_2$	*sequence*	e	::=	$e_1;\ e_2$	*sequence*
	|	...			|	...	

This expression was particularly useful in one of our proofs, where we needed to construct an expression to clear blame from the C store, and we highlight this point in Chapter 5. While not strictly necessary, we felt that including expression sequences in our calculus was not overly complicated while making it more expressive.

Finally, it's worthwhile to quickly discuss the error expression. Consider:

$$\begin{array}{rcll} e & ::= & \mathbf{err}\,\beta\,bi & \textit{error} \\ & | & \ldots & \end{array}$$

This expression has two pieces of information: β can be viewed as the *temporal* blame, and corresponds to a particular cast, call, or allocation, and bi is *spatial* blame, corresponding to a particular function handle, cast expression, or C allocation expression. Together, they indicate both which C activity in the history of said activities caused the error (β), as well as which C function, cast, or allocation handle was a part of that call (bi).

Now that we have established the language of our system, we will delve into our main relations: the type transformation and reduction relations.

4.4 Typing and Reduction Relations

Making a distinction between typed and untyped language is not an uncommon practice, and in such systems the typing judgement is often modified to connect the two languages together. We define a *type transformation* relation, a modification of the standard *typing judgment* relation, which transforms a typed expression into an untyped expression. Consider:

$$\Gamma, K \vdash te : T \rightsquigarrow e \tag{4.1}$$

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 Γ and K. Here, Γ 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 (as discussed in Section 4.1), and a variable store for variables. K can thus be broken up into three store typings: Σ_T describing the table store, Σ_C for the C store, and Σ_V for the variable store.

The full relation will be given in Section 4.5.

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

$$e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \rightarrow e' \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \,/\, \beta' \tag{4.2}$$

Here, e and e' are expressions in the untyped language, σ_T and σ'_T are *table stores*, σ_C and σ'_C are *C stores*, σ_V and σ'_V are *variable stores*, and finally β and β' are integers

representing blame information. At a high level, the table store σ_T is a list of Lua tables, the variable store σ_V is a list of values, and finally the C store σ_C is a list of C value, C type pairs which may be tagged with blame. Further, a C store may itself be tagged with blame, which we sometimes refer to as *global blame*. As we mentioned, the unusual inclusion of type information in the runtime C store is required to properly model C downcast semantics. The reduction relation will be fully given and discussed in Section 4.6.

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

$$e \,/\, \mathcal{S} \to e' \,/\, \mathcal{S}' \tag{4.3}$$

We refer to $\mathcal{S}$ and $\mathcal{S}'$ above as the *runtime environment*. It will be necessary to differentiate between C stores based on whether or not they contain blame; for this purpose, we say that a C store is *blame-free* if the store as a whole is not globally blamed, and none of the elements of the store are tagged with blame. To simplify discussion of environments with or without blame, we say that a runtime environment is blame-free if its C store is blame-free.

Before diving into the type transformation, it's worth discussing some of the mechanisms that will come up in our discussion. When a variable is declared in a let binding or bound in a function abstraction, we reserve some space for it in the variable store σ_V and substitute that location in throughout the relevant scope. We achieve automatic dereferencing by substituting a dereference operation in where appropriate, and allow for updates by substituting the location when the variable is in left-hand position. This approach is fairly standard for representing mutable variables.

We make use of a few auxiliary functions in our semantic rules.

- $\mathit{goodLayout}(n, T_C, \Sigma_C)$ checks to see if location n in the C store typing Σ_C represents type T_C. If T_C is a primitive type or a pointer type, this succeeds if $\Sigma_C(n) = T_C$. Recall that structs are not first-class C store members, and 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 Σ_C with the correct type, i.e. for all fields s_i we must have $\Sigma_C(n + i) = T_C^i$.

- $\mathit{update}\,(\sigma, l, v)$ is used in reduction rules, and updates store σ at location l with value v.

- $\mathit{layoutTypeWithBlame}\,(T_C, \beta)$ is used in the reduction of allocation of C values. It lays out type T_C and places blame β on pointer members (as per our discussion of

unsafe pointer allocation in Chapter 3). 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 blame if T_C is a pointer type, and if T_C is a struct, we layout each of its members in a similar fashion.

- *makeValueOfType* (T_C) creates a value of type T_C, and this is used in situations where we want to say that we get *some* value, and know nothing about it. This amounts to a statement such as: "there exists a value v of type T_C".

- *offsetForType* (s, T_C) computes the offset of member s in structure type T_C. Our formalization of the C store does not allow structs to exist in the store per se, and they are instead laid out according to their type, and this function relates their type (T_C) to their layout in the store.

- *blameCStore* (σ_C, β) tags every (as of yet untagged) C store σ_C location with blame β uniquely associated with an FFI call. Untagged locations gain blame, which allows subsequent failing accesses to report which FFI call is suspect. We only tag untagged locations to deal with multiple C calls which may have tampered with some memory, and this will be further elaborated later.

- *coerceToLua* (v_C) is a function which takes a C value v_C 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 $\mathbf{ptr_C}\ n\ T_C$, then it is coerced into a Lua pointer $\mathbf{ptr_{Lua}}\ n\ T_C$ (to the same location). Otherwise, the coercion fails. This function is used in **cget**s to coerce the C value from the C store to Lua to allow the reduction to go through.

- *coerceToC* (v_{Lua}) is similar to the *coerceToLua* function, though it coerces Lua values to C instead. If v_{Lua} is a numeric constant, the function produces a C integer with the same numeric value, and if v_{Lua} is a Lua pointer $\mathbf{ptr_{Lua}}\ n\ T_C$, an equivalent C pointer $\mathbf{ptr_C}\ n\ T_C$ is produced. This function is used in **cset**s to coerce the Lua value to C before putting it in the C store.

- *buildTable* (T_L) is a simple function which takes a table literal T_L and converts it into a format that the table store can handle.

Now that we have a big picture of our formalization, we will present our type transformation and reduction rules.

4.5 Type Transformation

Recall that the type transformation relates typed expressions to runtime expressions. In this section, we will discuss a number of interesting typing rules, paying particular attention to those pertaining to C. Note that we do not discuss every rule, and the full set of rules is available in Appendix A.

As we mentioned in Section 4.1, we promoted table literals to first-class language members. The rule for table allocation follows:

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

Rule TT_Table declares a table of type $\{f_1, ..., f_n\}$ with field values $\{tv_1, ..., tv_n\}$ (which is compiled into $\{v_1, ..., v_n\}$ in the untyped language). For the mechanization, a slightly modified syntax was required, using the classic nil and cons list constructors, and we will discuss this further in Chapter 5.

Of course, we can store these tables in variables, and the type transformation rule for variable allocation is given next. Consider:

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

The typing rule for traditional let bindings is standard, requiring that $te1$ type to T, ensuring that we are indeed assigning to x something of the specified type. The type of the binding is the type of te_2 in an extended environment, where x has type T.

The variables x in the let bindings above are typed according to the following:

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

This rule is unsurprising, wherein we simply find x in the typing environment Γ and produce the type stored there.

Having seen how to type variable and table allocation, let us consider the typing of C allocation.

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

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 structure (itself made up of valid types). This prevents programmers from making nonsensical statements, such as allocating C functions. Similar to C calls, C allocation may be a source of blame if a pointer type is being allocated, so we include a unique blame identifier n to identify it if necessary.

Through our substitution relation, allocated variables are automatically dereferenced where appropriate, and variable locations only exist explicitly on the right-hand side of assignment expressions, which we will go over shortly. First, we will see how to type locations into the table and C stores through the register and Lua pointer expressions.

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

We only type valid registers, and in our formalization a valid register is one which points to a location in the table store (i.e. the location is smaller than the length of the store). **reg** n types according to the type stored in the table store typing at location n. Typing a Lua pointer to C is more involved. Consider:

$$\frac{n < \mathit{length}(\Sigma_C) \qquad \mathit{goodLayout}(n, T, \Sigma_C)}{\Gamma, (\Sigma_T, \Sigma_C, \Sigma_V) \vdash \mathbf{ptr}\ n\ T_C : \mathbf{ptr_L}\ T_C \rightsquigarrow \mathbf{ptr}\ n\ T_C} \qquad \text{(TT\_C\_Ptr)}$$

C values are always 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 $\mathbf{ptr}\ n\ T_C$, n is the location referenced in the C store typing Σ_C (and, by extension, the C store σ_C), 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 (akin to TT_Reg), since $\Sigma_C(n)$ will never have a struct type. With this type information, we 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.

Different operations are required to access the data contained at these locations. Register access is done entirely through dot access (rule TT_Dot_Access), discussed shortly. Variable access is done through a dereference expression, shown below.

$$\frac{\Gamma, K \vdash te : \mathbf{ref}\, T \rightsquigarrow e}{\Gamma, K \vdash \mathbf{deref}\, te : T \rightsquigarrow \mathbf{deref}\, e} \quad \text{(TT_Var_Deref)}$$

In our system, recall that variable declaration triggers a substitution of a newly allocated location into the expression in which the variable is bound. This substitution determines whether or not to dereference the location, based on whether or not it's in right-hand position. Programmers will never write this dereference operation, but it is nonetheless required for our proofs (it is not part of the user language, but rather the intermediate language).

Programmers must explicitly dereference Lua pointers, and this requires a bit more machinery. Consider:

$$\frac{\begin{array}{c}\Gamma, K \vdash te : \mathbf{ptr_L}\, T_C \rightsquigarrow e \\ \mathit{validForCDeref}(T_C) \qquad T_L = \mathit{coerceCType}(T_C)\end{array}}{\Gamma, K \vdash \mathbf{deref_C}\ te : T_L \rightsquigarrow \mathbf{cget}\ e\ 0\ T_C} \quad \text{(TT_Var_C_Deref)}$$

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 other 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.

Poseidon Lua allows variable mutation, and the following rules show how we type that behaviour:

$$\frac{\begin{array}{c}x \in \Gamma \\ \Gamma, K \vdash te : \Gamma(x) \rightsquigarrow e\end{array}}{\Gamma, K \vdash x := te : T \rightsquigarrow x := e} \quad \text{(TT_Var_Assign)}$$

In this rule, we are typing the actual variable assignment. As usual, the variable x must be in the typing environment Γ, and the expression being assigned (here, te) must have

the correct type $\Gamma(x)$ for the assignment. At runtime, each of these should be translated into location updates, which are shown next.

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

When a variable appears in left-hand position, the substitution relation will substitute it with its location, thereby transforming the variable assignment into a location update. Similar to TT_Var_Assign, te must be of the correct type (here, $\Sigma_V(n)$) in order for the update to type, and of course n must be in Σ_V.

Now, we will consider both Lua and C function declarations. As we alluded to in earlier chapters, we deal only with single-variable functions. On the Lua side, multi-variable functions are be curried into chained single-variable function calls (see Section 2.3). On the C side, one could pass a pointer to a structure containing all of the desired function arguments. We first consider Lua function declarations:

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

The Lua function has a standard typing rule, in line with usual variable binding. The function $\lambda x : T.te$ types according to its body te in the context Γ extended with binding $x \mapsto T$.

C functions type as follows:

$$\frac{\mathit{fresh}\ n}{\Gamma, K \vdash \mathbf{cfun}\, (ct_1 \to_C ct_2)\ n : (ct_1 \to_C ct_2) \rightsquigarrow \mathbf{cfun}\, n} \quad (\text{TT\_C\_Function})$$

Here, note that the C function expression contains the type of the function, and the function trivially types. Type information is necessary because we don't model C's semantics: In TT_Function, the return type could 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.

Since C calls are sources of blame, we include n as an identifier uniquely associated with the function handle, taken as "user input" insofar as it is able to be generated at compile-time. In the event of a failure, we can determine which call (and, thus, which function handle) is to blame.

Now, we delve into function application.

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

In rule TT_Lua_Fun_Appl, we have what is essentially a standard typing judgment for functions. The application of a function te_1 with type $T \to_L T'$ to an argument of type T yields something of type T'. Thankfully, we can type C calls in a similar way:

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

In rule TT_C_Fun_Appl, we proceed in a similar fashion as in TT_Lua_Fun_Appl, where we type the function application according to its return type. Note the T' in the compiled (on the right of the $\rightsquigarrow$) C call: As we mentioned in Section 4.3, the untyped call requires the return type for reduction to be possible, and we will discuss this in more detail in the next section.

Note that both function application rules have the same typed expression $(te_1(te_2))$, but compile into different untyped expressions. Depending on the type of te_1, the transformation generates either an Lua function application $e_1(e_2)$ or an FFI call $\mathbf{ccall}\, e_1\, e_2\, T'$. This way, programmers in the typed language do not need to call C and Lua functions in different ways, and can use the same syntax in the typed language.

Similar differentiation occurs with dot accesses and updates:

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

Here, if te_1 types to T_1, T_1 is a table type, and te_2 types to a string literal s which is a field name in T_1, then the table member access types. The table access transforms into a **rawget** in the untyped language. Next, C:

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

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.

As with functions, the types allow for generation of either a C **cget** or a Lua **rawget** when appropriate. Similar logic applies to member update:

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

Here, if te_1 typed to T_1, T_1 is a table type, te_2 types to a string literal s which is a field name in T_1, and te_3 types to the type of field s in T_1, then the table member update may type. The table update transforms to a **rawset** in the untyped language.

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

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 additionally 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.

Next, as we mentioned previously, Poseidon Lua allows C values to be downcast. The typing rule for said downcasts follows.

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

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.

Finally, to simplify all of the rules in the face of subtyping, we include a standard subsumption rule.

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

Subsumption allows us to seamlessly integrate subtype relationships into typing rules.

In this section, we have outlined the type transformation for our formal specification, which roughly corresponds to more classical typing judgments. This relation dictates the relationship between the typed and untyped language, describing in which circumstances typed expressions could "type" and transform into equivalents in the untyped language. However, we have yet to see how the language actually *runs*, and we address this in the next section.

4.6 Reduction Relation

In this section, we will present the reduction relation, describing the reduction of runtime expressions and the execution of programs. Though the Lua language specification doesn't specify an evaluation order for expressions, we mirrored [26] in their choice of a left-to-right evaluation order, justified by both the standard Lua compiler [16] and LuaJIT [22], as well as Poseidon Lua itself.

For the purpose of presentation, we divide the reduction relation into two parts: the Lua reductions are presented first, before diving into the C reductions.

We start in the same order as we viewed the type transformation, with table literals.

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

This is the reduction for table construction. The table literal $\{s_1 = v_1, ..., s_n = v_n\}$ is a list of string, value pairs which are the initial members of a table. The *buildTable* auxiliary function merely transforms the table literal into a format that σ_T can handle. We place the newly created table into the table store, and reduce to the register pointing to that location in the variable store. This register could then be saved into a variable, perhaps through a let binding, which is considered next.

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

Above, $\mathbf{let}\,x := e_1\,\mathbf{in}\,e_2$ is stating that x should be equal to e_1 in e_2, in that we should substitute e_2 in for x in e_1. The reduction rule R_Let reserves space in σ_V for x and substitutes that location in for it in e_2, which will be automatically dereferenced to e_1 where appropriate. Lua function application is similar:

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

Here, the idea is to replace x with e_2 in e. Again, space is reserved in σ_V for x, and the location is substituted in for it in the function body, and substitution takes care of dereferencing.

The semantics of said dereferencing is standard:

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

Variables in right-hand position, after space has been reserved in σ_V, are automatically converted to dereferences. To reduce these dereferences, we locate the value referred to in σ_V and produce it. Of course, v must be a value, but this should always be true according to our other reduction rules which detail how we store things in σ_V.

Next, we show variable mutation. Recall that in the typed language, we had typed expressions for both variable assignment and location update: $x := e$ and $\mathbf{loc}\ l := e$, respectively. In the untyped language, valid (re)assignments will *always* be location updates, as any previously defined variable appearing on the left-hand side of a reassignment will have been substituted with a location. That said, consider the following reduction rule:

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

Here, we update location l in the variable store σ_V with the specified value.

We will now consider rules for interacting with tables. As in FWLua, we desugar all table operations to **rawget**s and **rawset**s, which are the underlying Lua mechanisms which drive table functionality.

$$\frac{\sigma_T(n) = T \qquad T(s) = v}{\mathbf{rawget}\,(\mathbf{reg}\,n)\,s \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to v \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta} \quad \text{(R\_RAWGET)}$$

rawgets access members in tables contained in the table store σ_T. The expression $\mathbf{rawget}\,(\mathbf{reg}\,n)\,s$ looks for a table at location n in σ_T, and looks in that table for member s, producing the value found there. Thanks to the type transformation, this lookup will always work, since the typing judgment rejects invalid accesses.

Next, we survey table member updates.

$$\frac{\begin{array}{c} \mathit{value}\,(e_3) \qquad \sigma_T(n) = T \qquad s \in T \\ T' = \mathit{update}\,(T, s, e_3) \qquad \sigma'_T = \mathit{update}\,(\sigma_T, n, T') \end{array}}{\mathbf{rawset}\,(\mathbf{reg}\,n)\,s\,e_3 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to \mathbf{reg}\,n \,/\, \sigma'_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta} \quad \text{(R\_RAWSET)}$$

rawsets update table members. In the expression $\mathbf{rawset}\,(\mathbf{reg}\,n)\,s\,e_3$, we find the table at location n in σ_T, and replace the contents of member s with the value e_3.

Having seen the reductions which describe the behaviour of Lua, we will turn our attention to those that describe both C's behaviour and its interaction with Lua.

4.7 Adding C

The starring feature of our formalization is the C FFI. Recall that we chose not to model C's semantics, which allows us to keep our proof more general, and focus on the effects of C rather than the minutia of its execution. Indeed, a direct consequence of this is that we need to account for all possible semantics of C, and are thus faced with nondeterminism.

In this section, we will describe the reduction rules associated with C, starting with allocation of C things. For illustrative purposes, consider the following code snippet:

```
cstruct IntNode {
  int data
  IntNode * nextNode
}
a : IntNode = calloc IntNode
```

Here, we have an `IntNode` C structure containing an integer member `data` and a pointer `nextNode` to another IntNode. C allocation reserves space in the C store σ_C and initializes

it with default values based on the types being allocated (in this case, the types of the struct members).

Now, accessing `a`'s `data` member is fine—the integer is reliably initialized to `0`. However, attempting to dereference the `nextNode` member might cause some trouble (in fact, dereferencing newly allocated pointers in C is undefined behaviour), since the default value (typically `0`, as is the case in our system) might not actually refer to a location which contains an `IntNode`, at least not until it is properly initialized.

To fully capture this behaviour, consider the following reduction rule for C allocation:

$$\frac{\begin{array}{c} n = \mathit{length}\,(\sigma_C) \\ \sigma'_C = \sigma_C + \mathit{layoutTypeWithBlame}\,(T_C, (\beta, n)) \end{array}}{\mathbf{calloc}\,T_C\,n \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \rightarrow \mathbf{ptr}\,n\,T_C \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V \,/\, \beta + 1}\ (\text{R\_CAlloc})$$

The $\mathbf{calloc}\,T_C\,n$ expression allocates enough memory in σ_C to accommodate a value of type ct. The function *layoutTypeWithBlame* determines the C store configuration necessary for the value. If T_C either is or contains a C pointer type, then we tag that location with blame, to indicate to our system that its behaviour is undefined until it is successfully observed. Here, the blame information is β, an integer which is carried around in the runtime environment and incremented every time C causes blame, and n, the unique identifier for this **calloc** expression. Following allocation, a C pointer with the location of the beginning of the newly allocated memory is produced.

Only the allocation of pointer types are "dangerous", as only pointers introduce undefined behavior (and therefore blame) into the system. For instance, we could safely allocate structs with (non-pointer) struct members.

In the spirit of discussing blame, we now turn our attention to FFI function calls, which are perhaps the most obvious source of blame in our system. We have no concept of what *exactly* occurs in the call, only of what *can* occur, and must account for all possible scenarios. Consider the following example:

```
p1 : Point = calloc Point
p2 : Point = cFuncCopyPoint(p1) // C function call
```

Say we have access to a C function, called `cFuncCopyPoint`, which takes and returns a `Point`. Now, say we allocate a `Point`, store it in `p1`, and call `cFuncCopyPoint` with it. Without a model of the body of the function, we can't be sure of what the function has done. The API for the function (here, its type) indicates that if execution of the function is successful, it should return something of type `Point`.

To capture all of this behaviour, we make use of two auxiliary functions: *makeValueOfType* and *blameCStore*. In the event that a function successfully executes, producing a value of its intended type, *makeValueOfType* allows us to say that the resulting value is *some* value of the intended type. Of course, the function might fail, perhaps via segmentation fault in the C code, but whether or not the function successfully executes, we capture the possibility of it tampering with σ_C by tagging the entire C store with blame through the *blameCStore* function.

Consider the following two reduction rules for function execution.

$$\frac{\begin{array}{c} \mathit{value}\,(e_2) \\ v = \mathit{makeValueOfType}\,(ct_2) \qquad \sigma'_C = \mathit{blameCStore}\,(\sigma_C, (\beta, n)) \end{array}}{\mathbf{ccall}\ (\mathbf{cfun}\ (ct_1 \to_C ct_2)\ n)\ e_2\ ct_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to v \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V \,/\, \beta + 1} \text{(R\_CCall\_Worked)}$$

Here, **ccall** (**cfun** $(ct_1 \to_C ct_2)\ n)\ e_2\ ct_2$ calls a C function **cfun** $(ct_1 \to_C ct_2)\ n$ with argument e_2 and is expecting to receive something of type ct_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 tag it all with blame (β, n): β corresponds to the βth C activity which caused blame, and n roughly corresponds to the line of code of the call. Now, C function calls can also fail:

$$\frac{\mathit{value}\,(e_2) \qquad \sigma'_C = \mathit{blameCStore}\,(\sigma_C, (\beta, n))}{\mathbf{ccall}\ (\mathbf{cfun}\ (ct_1 \to_C ct_2)\ n)\ e_2\ ct \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to \mathbf{err}\ \beta\ n \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V \,/\, \beta + 1} \text{(R\_CCall\_Failed)}$$

To capture that these are both 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. *blameCStore* accounts for any memory twiddling that the function might have done, tagging untagged locations with blame. The importance of leaving existing blame untouched is best understood with the following example, where more than one C call occurs between accesses to the same location. Consider:

```
p : Point = calloc Point
someCCall(p)
someOtherCCall(p)
print(p.x)
```

Here, if `p.x` fails, we know that the memory location `p` refers to was mucked with, but we can't say for sure if `someCCall` or `someOtherCCall` is to blame. The failure will

report `someCCall`, since it's the earliest FFI call which might have freed `p`; this indicates to the programmer that they should check `someCCall` as well as any subsequent C calls for possible errors. Now, consider:

```
p : Point = calloc Point
someCCall(p)
print(p.x) -- works
someOtherCCall(p)
print(p.x) -- fails
```

Here, the first access of `p` succeeds, erasing the blame from `p` in the C store. Then, the next access of `p` fails, whereby we blame `someOtherCCall`. This idea of isolating guilty calls will come up in Chapter 5, in our statement of Fault Isolation.

Another possible source of blame is downcasting C values, which reduces as follows:

$$\frac{\sigma_C(n) = (v, T_C, \emptyset) \qquad \begin{array}{c} n < \mathit{length}(\sigma_C) \\ \sigma'_C = \mathit{update}\,(\sigma_C, n, (v, T'_C, (\beta, b))) \end{array}}{\mathbf{ccast}\,(\mathbf{ptr}\,n\,T_C)\;T'_C\;b \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to \mathbf{ptr}\,n\,T'_C \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V \,/\, \beta + 1}\;(\text{R\_CCast})$$

Downcasting of C values is allowed in Poseidon Lua. In C, casting a memory location to a certain type (say, T'_C) simply allows those bits to be read as a value of type T'_C. We achieve similar semantics with the inclusion of blame: When attempting to read location n in σ_C after it was cast, the presence of blame indicates that v may not be of type T'_C. To keep our system as general as possible, we don't attempt to model the cast per se, and the read will instead replace v with a new value of type T'_C (or fail). This is discussed next.

Until now, we focused on the introduction of blame and fairly direct sources of nondeterminism, and we will now turn our attention to blame'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 allocation: When we allocate some C pointer type, we must tag the location with blame, as we have no guarantee that the allocated pointer points to something of the correct type. Now, imagine that we take such an freshly allocated pointer, and replace it with a known pointer of the correct type (perhaps as part of an assignment operation). From then on we can be sure of the behavior of that location, as it has been observed to be correct (in the sense that our correct assignment succeeded). Such an operation can be said to *erase* the blame at that location; in our formalization, blame represents uncertainty about a C value, and once we become certain of it we can safely remove the blame.

In more formal terms, the presence of blame at a location in σ_C indicates that accessing that location yields nondeterministic results. To capture this, as before we ensure that access to a blamed location can reduce to more than one expression under the same premises. Recall that elements in σ_C are of the form $(v, T_C, \beta?)$, where v is a C value, T_C is its type, and $\beta?$ is optional blame information (β can either be $\emptyset$, representing no blame, or some id).

Consider the following semantics for accessing σ_C:

$$\frac{\sigma_C(n+o) = (v, T_C, \emptyset) \qquad v_{out} = coerceToLua(v)}{\mathbf{cget}\,(\mathbf{ptr}\,n\,T'_C)\,o\,T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \rightarrow v_{out} \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta} \text{ (R\_CGet\_No\_Blame)}$$

Here, the expression $\mathbf{cget}\,(\mathbf{ptr}\,n\,T'_C)\,o\,T_C$ accesses σ_C at location n with offset o, and is expecting something of type T_C. In reduction R_CGet_No_Blame, there is no blame at location $n+o$ in σ_C, and so the store access cannot fail. The access steps to v_{out}, which is the Lua equivalent of the C value contained in σ_C. Now, consider a situation where there is blame:

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

Here, we access σ_C at location n with offset o, and are expecting something of type T_C as before. However, there is blame at $\sigma_C(n+o)$, and so both success and failure are possible. In this reduction rule, we deal with successful access to blamed locations. In this case, successful access erases the blame and a value of the appropriate type (thanks to the *makeValueOfType* auxiliary function) is returned after being coerced to Lua. The C store is updated with the new value, the expected type of the **cget**, and with no blame, and from this moment on, usage of this location is deterministic; the value was observed to be *something* of type T_C, though not necessarily the value that was there before. The following reduction rule deals with failing access:

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

Here, the access fails, reporting the blame information (the ID b of the suspect C activity and the line of code l).

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

Similar to **cget**, **cset** has nondeterministic semantics in the face of blame. First, consider the case where no blame is present:

$$\frac{\sigma_C(n+o) = (v, T'_C, \emptyset) \qquad \mathit{value}\,(e_2) \qquad v_{put} = \mathit{coerceToC}(e_2) \qquad \sigma'_C = \mathit{update}\,(\sigma_C, n+o, (v_{put}, T_C, \emptyset))}{\mathbf{cset}\,(\mathbf{ptr}\,n\,T''_C)\,o\,e_2\,T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e_2 / \,\sigma_T \,/\, \sigma'_C \,/\, \sigma_V \,/\, \beta} \text{ (R\_CSet\_No\_Blame)}$$

In the expression $\mathbf{cset}\,(\mathbf{ptr}\,n\,T''_C)\,o\,e_2\,T_C$, we write e_2 to location n with offset o in σ_C, and we expect the location to have type T_C. Note that we must first coerce e_2 to a C value v_{put} to store it in σ_C. As before, there is no blame at location $n+o$ in σ_C, and so the store update cannot fail. The rule for **cset**s on blamed locations is given below:

$$\frac{\sigma_C(n+o) = (v, T'_C, (b, l)) \qquad \mathit{value}\,(e_2) \qquad v_{put} = \mathit{coerceToC}(e_2) \qquad \sigma'_C = \mathit{update}\,(\sigma_C, n+o, (v_{put}, T_C, \emptyset))}{\mathbf{cset}\ (\mathbf{ptr}\,n\,T''_C)\ o\ e_2\ T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e_2 / \,\sigma_T \,/\, \sigma'_C \,/\, \sigma_V \,/\, \beta} \text{ (R\_CSet\_Blame\_Works)}$$

Here, again, we coerce e_2 to a C value v_{put} to location n with offset o in σ_C, and we expect the location to have type T_C. However, $\sigma_C(n+o)$ is blamed, and so we are in a nondeterministic state. In rule R_CSet_Blame_Works, the access succeeds: We update $\sigma_C(n+o)$ with the new value v_{put}, the type T that we were expecting, and erase the blame. Note that no call to *makeValueOfType* is required, as we are sure of the value we are writing. Of course, failure is always an option:

$$\frac{\sigma_C(n+o) = (v, T'_C, (b, l))}{\mathbf{cset}\ (\mathbf{ptr}\,n\,T''_C)\ o\ e_2\ T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to \mathbf{err}\ b\ l \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta} \text{ (R\_CSet\_Blame\_Fails)}$$

Here, the write fails, and reports the blame information stored at $\sigma_C(n+o)$.

In this section and the last, we presented the formal semantics for Poseidon Lua. We discussed the *type transformation*, which related typed expressions to runtime expressions, drawing the correspondence between written and executed Poseidon Lua, and the *reduction relation*, which defined the semantics of "running" programs in Poseidon Lua. Next, we

detail the properties of our system, such as type soundness, and dig into some other interesting results.

Chapter 5

Formal Properties

When making statements and proving results in a nondeterministic context, one needs to be careful that their assertions are meaningful. For example, in the statement of preservation, typically one considers *all* possible expression reductions, and the typing judgment restricts the reductions to those which are sensible. However, when certain expressions can fail in well-typed circumstances, preservation falls apart.

In this chapter, we explore the effects of nondeterminism on our language, and show type soundness (via proofs of preservation and progress), conditional on the *absence* of blame from the runtime environment. If there is blame in the environment, we are in a state of nondeterminism, and we can show that in such an environment, there exists a path to a *blame-free* environment, where we reclaim both determinism and type soundness; this result is aptly named *reclamation*. Taken together, this signifies that Poseidon Lua is sound and deterministic until a C call, allocation, or downcast occurs: These unsafe activities introduce blame, which can be erased thanks to reclamation. Finally, from a more practical standpoint, we show *fault isolation*, which states that in the event of a runtime failure, a new program can be generated which can isolate the unsafe reduction which caused the failure.

We take this opportunity to stress that we do *not* present full proofs in this chapter, and instead present only the most salient details. To be assured that our proofs are indeed correct, we have mechanized the entire formal specification and semantics in Coq, a dependently-typed programming language and proof assistant that sees a lot of use in the programming languages community for work of this sort. For the interested reader, our machine-checked proof is available as extra material, and this chapter will focus on a high-level overview of our proofs with some small supporting examples. We start by

considering the type soundness of Poseidon Lua.

5.1 Type Soundness

Preservation and progress are the two ingredients of type soundness, and these statements need to be carefully worded to be meaningful in the face of nondeterminism.

The classic [23] statements of preservation and progress are:

- *Preservation:* if a well-typed term (i.e. a term obtained through the typing judgment) steps, then it steps to a term which is also well-typed.
- *Progress:* well-typed terms are either values, or they step.

In most well-constructed semantics, these statements make sense, and together they make up type soundness. That said, when dealing with a nondeterministic semantics, however well-constructed, we are required to reword the statements of preservation and progress: For example, in our nondeterministic semantics, a well-typed C Call expression can step to the error state *and* to some value that it produced as part of successful execution. This weakens progress for C Calls, as a well-typed C Call can always trivially step to error, and also complicates preservation, as the error expression is not well-typed in our system.

The crucial observation to make is that, while runtime expressions can nondeterministically step to error, they may only do so in the presence of blame in the C store. As such, we restrict our statements to only apply to blameless C stores. This isn't to suggest that we can say nothing about Poseidon Lua when blame is present, and in Section 5.2 we explore how Poseidon Lua behaves in this setting.

We will first turn our attention to the statement of preservation.

5.1.1 Preservation

Generally speaking, preservation states that the validity of a typing judgment is preserved throughout program execution, and that the type of an expression can only get *more precise* as it steps. As we mentioned, we needed to carefully word this statement for it to be meaningful when well-typed expressions can step nondeterministically to the error state, and we achieve this by requiring that no blame be present in the runtime environment.

Theorem 5.1.1 (Preservation) *If typed expression te has type T and type compiles to expression e in environment Γ and typing context K, written as $\Gamma, K \vdash te : T \rightsquigarrow e$, and e in runtime context S reduces to e' in runtime context S', written as $e \,/\, S \rightarrow e' \,/\, S'$, and both S and S' are blame-free, then there exists K', te', and T' such that $\Gamma, K' \vdash te' : T' \rightsquigarrow e'$, where $T' <: T$ and K' extends K.*

Essentially, Theorem 5.1.1 is stating that if an expression types and steps, then the expression resulting from the step also types, but only in the absence of blame. Restricting the runtime environment to be blame-free makes the system deterministic, and prevents reduction to the C error expression (where we would lose Preservation). This is not unlike the classical statement of preservation [23], with the caveat that no blame is present.

Note that we also assert that the resulting typing context K' (which types e') must extend the original typing context K. Since certain expressions, such as let bindings and function applications, can change the runtime environment by extending runtime stores, we must allow the typing context to be able to evolve and change to describe the new runtime environment.

In Coq, the statement of the theorem is:

```
1  Theorem preservation :
2    forall e s p H e' s' p' H' IL IL' K te t,
3    [],, K |- te : t ~~> e ->
4    all_stores_well_typed K s p H [] ->
5    e / s / (p, None) / H / IL ==>
6        e' / s' / (p', None) / H' / IL' ->
7    blame_free (p,  None) ->
8    blame_free (p', None) ->
9    exists te' t' K',
10       extends_all K' K /\
11       [],, K' |- te' : t' ~~> e' /\
12       all_stores_well_typed K' s' p' H' [] /\
13       t' <: t.
```

The above is roughly equivalent to the high-level statement of preservation in Theorem 5.1.1. Mechanically, we showed preservation via induction on the type transformation at line 3, and carefully worked through each of the subcases. In the Coq statement, `s` is the table store, `p` is the C store, `H` is the variable store, and `IL` is the runtime blame information, and together these make up the runtime environment. Notice the placement of the C store in the reduction at lines 5 and 6; the `None` in `(p, None)` and `(p', None)`

represents the optional global blame information on the C store—here it is `None` since the environment should be blame-free. We also see the assertions at lines 7 and 8 that the C stores `p` and `p'` are blame-free, and so we are guaranteed that no blame is present in the proof. Notable also is the statement that all stores be well typed at line 4: This relates the typing context `K` with the runtime environment, and assures that we have type information for all variables, pointers, and tables in the initial runtime environment, and we discuss this in more detail shortly.

Proving preservation required a number of supporting lemmas. First, to show anything relating to locations in stores, we need a statement that the runtime environment be *well-typed* with respect to the typing context; in the Coq code, this is line 4. Each of the variable, C, and table stores requires a slightly different statement.

The variable store is well-typed under two conditions. First, its length must match that of its store typing. Second, and most importantly, an expression at some location l in the variable store must type to a subtype of $\Sigma_V(l)$. The first point allows us to transfer information about the length of the store from the type transformation to the reduction relation, and the second point allows us to type variable access. The Coq statement of this is:

```
1  Definition var_store_well_typed
2   (vST: store_ty) (vst: var_store) (Gamma: typing_env)
3   (tST: store_ty) (cST: c_store_ty) :=
4    List.length vST = List.length vst /\
5    forall l,
6      l < List.length vst ->
7      exists T t te,
8        Gamma,, store_typings tST cST vST |- te : T ~~>
9          (var_store_lookup l vst) /\
10       Some t = (store_Tlookup l vST) /\
11       T <: t.
```

Here, line 4 corresponds to the lengths of the store and its typing matching, and lined 7 through 11 deal with typing the contents of the store.

Similarly, the table store must have the same length as its store typing, and each table element must type. In other words, let T be a table in σ_T, with type T_T in Σ_T, in typing environment Γ with typing context K (itself having table store typing Σ_T): For each field s of T, there must exist a typed expression te such that $\Gamma, K \vdash te : T \rightsquigarrow T(s)$, where $T <: T_T(s)$. As before, this allows us to type table access, and relate the runtime table representation to information in the type transformation.

The C store is markedly different. Since C values are not part of our language, interfacing with C must be done through the Lua pointer expression, and getting things from the C store is not quite as simple as, say, getting things from the variable store: In a variable store σ_V, Lua *expressions* are stored, so a statement such as **deref** (**loc** l), $\mathcal{S} \to \sigma_V(l), \mathcal{S}$ makes sense since $\sigma_V(l)$ is an expression, unlike $\sigma_C(l)$. Consequently, accessing things in the C store requires that they be coerced before being passed to Lua, and so it isn't fair to say that the inhabitants of the C store type; instead, we require that elements of the C store, once *coerced* to Lua equivalents, type. This is in line with all rules for accessing the C store, as we are either writing a coerced Lua value into the store, or producing a coerced C value as part of a read operation. In Coq, this translates to:

```
1  Definition c_store_well_typed
2   (cST: c_store_ty) (cst: c_store) (Gamma: typing_env)
3   (tST: store_ty) (vST: store_ty) :=
4    List.length cST = List.length cst /\
5    forall l,
6      l < List.length cst ->
7      exists te c_exp aBI ct K,
8      Some (actualStuff c_exp ct aBI) = c_store_lookup l cst /\
9      Gamma,, K |- te : (type_of_coercion ct) ~~>
10       (e_value (coerce_to_Lua c_exp)) /\
11     Some ct = (store_Tlookup_c l cST) /\
12     K = store_typings tST cST vST.
```

Here, we see that C store elements are packaged up as `actualStuff` triplets, in line with what we described earlier. Also, line 12 is merely a presentational detail (otherwise we would have no hope of fitting the typing judgment on two lines!). We additionally see that the types in the C store and the C store typing (here, both are `ct`) must match, and this is to relate type information from typing context to the runtime environment. One might wonder how C casting is done: If the store typing must match the store, type-for-type, and preservation requires that the new typing context extend the previous, then how could we type downcasts which change the type in a way that is *not* an extension of the previous context? The answer to this lies in our use of blame on downcast locations, and indeed the runtime environment resulting from a downcast is not blame-free, a condition of our proof, and after all blame has been erased, we will be able to construct a new typing context.

With the notion of a well-typed runtime environment, we also need to show that both allocation and update preserve the well-typedness of our system. To show this, we need to prove that reductions which extend and modify a well-typed runtime environment do

so in a sensible way. First, consider allocations. Say we have a store σ_i with store typing Σ_i, and we are allocating something in σ_i at runtime, extending it somehow to obtain σ_i'. To preserve the well-typedness of our system, we must provide a new store typing Σ_i' for which σ_i' is well-typed, and we can construct this Σ_i' by extending Σ_i with the appropriate types. Next, consider updates. Say we have a store σ_i with store typing Σ_i, and we are updating some location l in σ_i with a new value, obtaining a new store σ_i'. Again, we must provide a new store typing Σ_i' for which σ_i' is well-typed, and we can construct Σ_i' from Σ_i by updating the type information at l.

The lemma to prove this looks like:

```
1  Lemma var_assign_pres_store_typing :
2    forall cST tST vST Gamma H T t te l e,
3    l < List.length H ->
4    var_store_well_typed vST H Gamma tST cST ->
5    Gamma,, store_typings tST cST vST |- te : T ~~> e ->
6    Some t = (store_Tlookup l vST) ->
7    T <: t ->
8    var_store_well_typed vST (var_store_update l H e)
9      Gamma tST cST.
10 Proof.
11 ...
12 Qed.
```

The above is stating that if a location is valid, and the variable store was already well-typed, and the new thing we are writing to location `l` types to a subtype of `vST(l)`, then the resulting store is also well-typed. The proof is relatively straightforward, and can be divided into two parts: First, we show that the updated location `l` is well-typed w.r.t. `vST(l)`, which can be shown with lines 5 through 7, and then we show that all other locations are well-typed, which is true by line 4 (all non-updated locations were well-typed before, and nothing about them has changed).

Modifying the C store follows the same principles: Here, we update some location l in σ_C with some Lua value has been coerced to a C value. Recall that the C store is well typed only if each of its values, once coerced *back to Lua*, type. Showing that this remains true when updating is straightforward: Say we are updating some location l in σ_C with a C value v_C which was coerced from a Lua value v_{Lua}. In order to have this Lua value at all, it needed to pass the typing judgement, and that is exactly all the evidence we need to show that the C store is still well-typed (of course, as long as v_{Lua}'s type is a subtype of $\Sigma_C(l)$). Similar logic applies to extending the C store with new allocations.

We also have a few reduction rules which deal directly with substitution, and recall that in our system we substitute a *location* in for a variable, automatically dereferencing the location unless it appears on the left-hand side of a variable assignment expression. These are easiest to deal with with a proof that substitution preserves typing, stating that substitution does not break the typing judgment, and that we can "hoist" substitutions into the typing environment. Roughly speaking, if l refers to something of type T_l, then this looks like:

$$\Gamma, K \vdash ([x \leftarrow l]\, te) : T \rightsquigarrow ([x \leftarrow l]\, e) \approx \Gamma + \{x \mapsto T_L\}, K \vdash te : T \rightsquigarrow e$$

On the left, we have the type transformation on expressions with substitution, and that is basically equivalent to typing the expressions, sans substitution, in an extended context. One major advantage of "substitution preserves typing" is that we don't need to deal with variables, since they can all be safely substituted: Every binding in Γ is equivalent to a substitution, and this allows us to take Γ to be empty in our proofs.

In C, structs are stored contiguously in memory, with the first member at offset `0` from the struct's location, and other members at appropriate offsets according to the size of the data of previous members. Poseidon Lua has structs, and we formalize them with a C flavour, in that struct members are laid out in the C store as they are in C, though each member has unit size (as such, the second field of a struct will always be at offset 1). These structures are behind a Lua pointer, and in order for such a pointer to type we require that it refer to a location in σ_C which is correctly laid out w.r.t. the type of the struct (recall the *goodLayout* auxiliary function). Practically, updating locations in σ_C occurs as part of a **cset** on some pointer, typically a structure pointer, and so we need to ensure that the update preserves the "good layout" of the that pointer. This particular result required some surprising machinery, since we needed to relate the *goodLayout* auxiliary function (which deals with types and offsets) to the **cset** expression (itself dealing with pointers and offsets), paying close attention to ensuring that the offsets from **cset** and those obtained from the type were compatible. By comparison, dealing with tables and table types was simplified by the field name common to both the type and the table, and we had a direct correspondence between a table's value and type at field s.

To concretize this high-level description, we will walk through an interesting case of preservation: the case for a **cset** on a struct member when there is no blame in the C store. Recall the typing and reduction rules for **cset**:

$$\frac{\sigma_C(n+o) = (v, T_C, \emptyset) \qquad v_{out} = coerceToLua(v)}{\mathbf{cget}\,(\mathbf{ptr}\,n\,T'_C)\,o\,T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \rightarrow v_{out} \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta} \text{ (R\_CGet\_No\_Blame)}$$

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

Say we have $te_1.te_2 := te_3$ which type compiles to $\mathbf{cset}\ (\mathbf{ptr_L}\ l\ T_{Struct})\ n\ e_v\ T_C$, wherein we wish to write e_v to location l in the C store. To get through this case of preservation, we need to show 4 things:

1. The resulting typing context K' extends the previous typing context K;
2. Whatever this expression steps to types in typing context K' to some type T';
3. T' is a subtype of T;
4. The resulting runtime environment is well typed w.r.t. the resulting typing context.

We tackle these one at a time. First, we know that K and K' can only possibly differ in their C store typings Σ_C and Σ'_C respectively. A C store extends another if it is at least the same length and assigns the same types to each common location, and it is easy to see that this still holds, as Σ_C must be equal to Σ'_C, since the only way a **cset** at location l can type in a context with C store typing Σ_C is if it is writing something of type $\Sigma_C(l)$. In Coq, this amounted to relating the type of the struct member to the type at the appropriate offset in the runtime C store.

As for point 2, we know from the well-typedness of the stores that each store location can type. We also know that since $\mathbf{cset}\ (\mathbf{ptr_L}\ l\ T_{Struct})\ n\ e_v\ T_C$ types, then necessarily the sub-expression $\mathbf{ptr_L}\ l\ T_{Struct}$ also types by the inductive hypothesis. This pointer can only type of $\Sigma_C(l)$ has a good layout w.r.t. type T_{Struct} (recall the *goodLayout* auxiliary function), which tells us that any member of T_{Struct} is present with the correct type at its offset in Σ_C, and that all valid offsets are in the bounds of the C store typing. In addition, the type transformation gives us that n is the offset for whatever member te_2 referred to. Taken together, we can establish a clear correspondence between the member being written to and the runtime memory location being written to: Location $l + n$ corresponds to a location in the C store, and the C store being well-typed tells us that whatever is present at $\sigma_C(l + n)$ types, whereby point 2 is proved.

Point 3 is trivial, as **value** is a supertype of all types.

Point 4 is more interesting. We need to show that K (which is the same as K', as per point 1) is well-typed with respect to the newly updated runtime environment. Trivially, as the Lua stores are unchanged, they are still well-typed, and we need only show that that the C store is well-typed w.r.t. its typing context. Essentially, we only need to show that everything in the new C store σ'_C types, and for all locations other than $l + n$ this is trivially true (since they were well-typed, and were not changed by the reduction). That said, it is easy to see how $\sigma'_C(l+n)$ contains something which can type, since we just *wrote* such a thing into that location (recall that we wrote e_v to this location, and it types as per the inductive hypothesis). This requires that coercion be bijective and thus invertible, since we converted Lua to C in order to write, and must convert C to Lua to type, but this is true for all types which are valid struct members (i.e. primitives and C pointer types).

Broadly speaking, many difficulties arose out of our alternate representations at type time and runtime. For example, our definition of the C store was a source of many headaches: Not only does it not contain expressions which can step, requiring contents to be coerced before reasoning about them, it also does not contain struct literals, instead laying them out sequentially. This required a lot of nontrivial "hand-shaking" between struct types, C store layout, and offsets.

That said, we now turn our attention to progress.

5.1.2 Progress

At its core, progress asserts that expressions which type are either values (and thus terminal), or "make progress" in that typable non-value expressions reduce. Progress is not weakened at the same level as preservation when dealing with nondeterminism, but there are a few pitfalls that one must avoid, discussed shortly. First, the statement:

Theorem 5.1.2 (Progress) *If $\Gamma, K \vdash te : T \rightsquigarrow e$, then either e is a value, or there exists S and S', both blame-free, as well as e' such that $e \,/\, S \to e' \,/\, S'$.*

Here, we argue that if an expression types, then we can assure that it is either a value, or it steps. Though not strictly necessary, the absence of blame offers some symmetry with preservation. Had we included the possibility of having blame in the runtime environment, many aspects of progress would be made trivial; for example, access into the C store in the presence of blame could automatically step to error, which is not a particularly interesting transition from the perspective of type safely. The same mechanisms relating C types to

the C store that were used in proving preservation were required to prove progress, as well, and a similar induction on the type transformation was performed.

In Coq, our statement of progress is:

```
Theorem progress : forall tST cST vST te T e s p H L,
 [],, (store_typings tST cST vST) |- te : T ~~> e ->
 all_stores_well_typed (store_typings tST cST vST) s p H [] ->
 blame_free (p, None) ->
 value e \/
 exists e' s' p' mb H' L',
   blame_free (p', mb) ->
   e / s / (p, None) / H / L ==> e' / s' / (p', mb) / H' / L'.
```

This captures the statement of Theorem 5.1.2, though like preservation we require that the initial environment be well-typed w.r.t. the typing context. We don't model an explicit error state (besides C errors, but those only arise when blame is involved, and we are concerned with blame-free environments), so the only danger is that a non-value expression can get stuck and be unable to reduce, which is precluded with a proof of progress.

The proof of preservation essentially required that the reduction relation and type transformation together provide enough information to type the resulting expression. In contrast, proving progress simply requires that the typing judgment give enough information to allow the expression to step, and it proved to be more straightforward to make the jump from type to runtime information than from runtime to type information.

To illustrate, we will walk through a small example involving Lua table access. Recall the typing rule for dot access on tables:

$$\frac{\begin{array}{cc}\Gamma, K \vdash te_1 : T_1 \rightsquigarrow e_1 & \mathit{tableType}\,(T_1) \\ \Gamma, K \vdash te_2 : s \rightsquigarrow e_2 & s \in T_1\end{array}}{\Gamma, K \vdash te_1.te_2 : T_1(s) \rightsquigarrow \mathbf{rawget}\, e_1\, e_2} \qquad (\text{TT\_Dot\_Access})$$

We wish to show that this, along with the runtime environment being well-typed, are enough to allow $\mathbf{rawget}\, e_1\, e_2$ to step. If either e_1 or e_2 step, then $\mathbf{rawget}\, e_1\, e_2$ trivially steps. If not, the typing judgment requires that they look like $\mathbf{rawget}\,(\mathbf{reg}\, r)\, \mathit{str}$, where $\mathbf{reg}\, r$ is a register and str is a string, as e_1 must have a table type (and registers are the only values which have such a type), and e_2 must have a string literal type, and so when e_2 is a value it must be a string.

Since register r types by the inductive hypothesis, we have:

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

Given the above, we know that $\mathbf{reg}\, r$ refers to a location which is known to be well-typed, and so we can take advantage of the well-typed assertion at line 3 of the progress statement. The well-typedness of the table store assures us that if a member s is present in a table store member T at location l, then the table at location l in the runtime table store *also* has a member s. To reduce the **rawget**, we need:

$$\frac{\sigma_T(r) = T \qquad T(\mathit{str}) = v}{\mathbf{rawget}\, (\mathbf{reg}\, r)\, \mathit{str} \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \rightarrow v \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta} \quad (\text{R\_Rawget})$$

We know that $\sigma_T(r) = T$ is true, thanks to the well-typedness of the environment, and $T(\mathit{str}) = v$ is true by the well-typedness of tables in the table store, since we have all the necessary type information for member str thanks to the typing judgment.

The effect of blame on our proofs is most apparent in preservation. Since we induct on the type transformation and are dealing with all possible reductions for an expression, if blame is allowed in the runtime environment we are faced with the possibility of an expression stepping nondeterministically to a C error. This is inadmissible from the point of view of type safety, and we had no choice but to include a blame-free clause. Note that one may still use C, so long as one does not either make an FFI call, cast a C value, or allocate a C pointer (Lua pointers to C values are fine).

In the next section, we explore the extent to which we can reason about our system when blame is present, and argue that our blame information is useful.

5.2 Blame

Recall that, in our system, blame is administered by expressions which can tamper with the C store: When these expressions reduce, the C store (or a relevant part) is blamed with a pair of unique identifiers, one for the line of code which caused the blame, and another isolating the blaming call, cast, or allocation in a "history" of all such operations. Later in execution, if the error state is reached, we produce the relevant blame information: If a C call triggered the failure, we blame that call, otherwise some sort of C store access triggered the fault, and we produce the blame information contained at the accessed location.

Reflecting on our system, one quickly realizes that the only reductions which cause blame involve C, be it a C function call, C pointer allocation, or C downcast. Given that, we see that *C is always at fault for runtime failure*:

Theorem 5.2.1 (Always Blame C) *If the error expression* **err** β b_{id} *is reached, there exists some C expression which is to blame.*

Essentially, the βth such C expression (at line of code c_{id}) caused the error, either by directly failing as a C call or by blaming some location in the C store. The presence of blame is required for runtime errors to occur, and C is the only source of blame, and so Lua cannot be blamed for these errors.

In our version of type soundness, we required that the runtime environment be blame-free. This assured that our semantics behaved deterministically, and disallowed reduction to the error expression. One might argue that we are essentially proving that "Poseidon Lua without C is type sound", but that is not exactly true. As was mentioned previously, programmers are still free to allocate, access, and write C data, and it's C calls, casts, and pointer allocation which introduce the possibility of arbitrary program termination.

In fact, we can go beyond type soundness, and all is not (totally) lost in the presence of blame. Each reduction which can nondeterministically fail can *also* succeed, and in failing or succeeding the program is essentially following one of two possible execution paths. Of course, there might be more than two execution paths (for instance, if we have several reductions with nondeterministic behaviour), but as we saw in Chapter 3, each of these paths represents a possible semantics for the program, and ultimately one will be selected at runtime.

To get a handle on this, let us consider a small example program:

```
local p : calloc int -- allocate a C integer
cCall(p)
print(*p)
print(*p)
```

Here, we allocate a C integer, store it in `p`, and make a C call. Say that the C call succeeds, and tags the C store with blame (namely, `p` is tagged with blame from the call). Now, say the dereference in the first `print` succeeds: In our formalization, this corresponds to a **cget** on a blamed location succeeding. At this point, is there any doubt about the *next* dereference in the following `print` call? The first successful **cget** erased the blame, so `p` should behave deterministically from this point on.

This brings us to an interesting observation: In the presence of blame, it is possible to *reclaim* determinism through a sequence of reductions which erase blame from the C store. This is known as reclamation.

Theorem 5.2.2 (Reclamation) *If there is blame in a runtime environment S, then there exists e, e', and S' such that $e \,/\, S \rightarrow^* e' \,/\, S'$ and S' is blame-free.*

This result relies on the fact that certain reductions erase blame from the C store, as they discover that, for instance, a location in the C store is well-behaved. If enough of these reductions occur, then all blame will be erased from the C store. Roughly speaking, this amounts to a proof that there exists a path from an environment with blame to an environment without blame, which corresponds to a program where the undefined behaviour of C did not cause anything to crash. Once we are blame-free, we regain type safety until another unsafe operation occurs.

Mechanically speaking, proving Theorem 5.2.2 is straightforward given the following two results:

Lemma 5.2.3 (Blame Erasure) *Say we have a C store σ_C in runtime environment $\mathcal{S}$ with blame at location l. There exists e, e', and σ'_C in runtime environment $\mathcal{S}'$ such that $e \,/\, \mathcal{S} \rightarrow e' \,/\, \mathcal{S}'$, where $\sigma'_C(l)$ is blame-free and $\forall\, n \neq l, \sigma_C(n) = \sigma'_C(n)$.*

With blame erasure, we show that there is always a reduction which can erase the blame at a particular location in σ_C. This can be either a **cset**, updating the blamed location, or a **cget**, reading from the store and observing the value contained within, and we proved this using **cget**s. This easily leads us to a small corollary, stating that the amount of blame in a C store can always decrease:

Corollary 5.2.3.1 (Blame Reduction) *If there is blame in a runtime environment $\mathcal{S}$, then there exists e, e', and $\mathcal{S}'$ such that $e \,/\, \mathcal{S} \rightarrow e' \,/\, \mathcal{S}'$, where $\mathcal{S}'$ has strictly less blame than $\mathcal{S}$.*

It is easy to see how one can have blame reduction given blame erasure, as one simply has to pick a blamed location in σ_C and apply blame erasure to it. We are sure that such a location exists, as there is blame in the runtime environment. To prove reclamation, we can repeatedly apply blame reduction to a C store σ_C which isn't blame-free to reduce the amount of blame in it, until no blame is present. Since the C store is finite, and we can

reduce the amount of blame monotonically, we are sure that there will eventually no longer be any blame in the environment.

In Coq, these proofs were surprisingly involved, as a number of small results were required in order to be able to deal with the blame which was now present in the environment; Most of our other results about C stores were specific to C stores which were blame-free, since this was a condition for type soundness. The mechanized statement of reclamation follows.

```
1 Theorem reclamation: forall p s H IL,
2   ~ blame_free (p, None) ->
3   c_store_pair_well_formed (p, None) ->
4   exists e e' s' p' H' IL',
5     e / s / (p, None) / H / IL ==>* e' / s' / p' / H' / IL' /\
6     blame_free p'.
```

Immediately, we see that the global blame accompanying the C store is `None`. This is mostly a convenience, as it halved the number of cases we needed to consider, and it's straightforward to show that if global blame is present, there is a path which gets rid of it. As per line 3, we also require that the C store `p` be well-formed, and this isn't quite the same "well-formedness" as in type soundness: Here, we wish to restrict the contents of the C store to primitives and pointers so as to preclude the appearance of invalid data. This clause renders it impossible for C functions to be in `p`, as C functions can only appear in C calls and shouldn't be stored. To show reclamation, we inducted on the *amount* n of blame in `p`, which we know is nonzero thanks to the fact that `p` is not blame-free. The base case where $n = 0$ is trivial, since n should be nonzero, and the recursive case is more interesting; we have some amount $n + 1$ of blame in the C store, and the inductive hypothesis states that any C store with n blame can be cleared of blame. Since there is blame in a C store, then there must exist an index such that accessing it yields a blamed value:

```
1 Lemma not_blame_free_means_loc_with_blame : forall p,
2   ~ blame_free (p, None) ->
3   exists l v t bi,
4     Some (actualStuff v t (Some bi)) = c_store_lookup l p.
```

Using this, we can obtain a location which is blamed, and then apply blame erasure to it. In Coq, blame erasure looks like:

```
1 Lemma blame_erasure : forall l s p H IL,
2   blamed_loc l p ->
3   c_store_pair_well_formed (p, None) ->
```

```
4    exists e e' s' p' H' IL',
5      e / s / (p, None) / H / IL ==>
6        e' / s' / (p', None) / H' / IL' /\
7      exists v t,
8        p' = c_store_update l p (actualStuff v t None) /\
9        c_store_pair_well_formed (p', None).
```

Clearly, a combination of these last two lemmas reduces the amount of blame in the C store by 1, say from $n+1$ to n, and so we can use the inductive hypothesis. However, things are not quite so simple: Both blame erasure and reclamation state that in a runtime environment $\mathcal{S}$ with blame, there *exist* e, e', and $\mathcal{S}'$ such that $e, \mathcal{S} \to e', \mathcal{S}'$ (or $e, \mathcal{S} \to^* e', \mathcal{S}'$ for reclamation) where $\mathcal{S}'$ less blame (in blame erasure) and no blame (in reclamation). To connect these two results, we see that once we apply blame erasure, say taking us from $e_1, \mathcal{S}$, where $\mathcal{S}$ has $n+1$ blamed locations, to $e_2, \mathcal{S}'$, where $\mathcal{S}'$ has n blamed locations, we can apply reclamation. Unfortunately, reclamation in environment $\mathcal{S}'$ merely states that there are *some* e_3, e_4, and $\mathcal{S}''$ such that $e_3, \mathcal{S}' \to^* e_4, \mathcal{S}''$, and we can see how reclamation and blame erasure don't quite transitively apply: blame erasure gives us $e_1, \mathcal{S} \to e_2, \mathcal{S}'$, reclamation gives us $e_3, \mathcal{S}' \to^* e_4, \mathcal{S}''$, and e_2 is not necessarily equal to e_3.

That said, we are not restricted to starting with e_1, and we need to show that, for runtime environment $\mathcal{S}$ with $n+1$ blamed locations, there exists x, x', and $\mathcal{S}^{\mathcal{R}}$ such that $x, \mathcal{S}' \to^* x', \mathcal{S}^{\mathcal{R}}$ where $\mathcal{S}^{\mathcal{R}}$ is blame-free. Here, we need not take $x = e_1$ (in fact, doing so would be erroneous!), so we must carefully construct an x which will allow us to take advantage of our two earlier results. To do this, we needed one other small corollary:

Corollary 5.2.3.2 (Blame Erase to Value) *If there is blame in a runtime environment $\mathcal{S}$, then there exists e, e', and $\mathcal{S}'$ such that $e \,/\, \mathcal{S} \to e' \,/\, \mathcal{S}'$, where $\mathcal{S}'$ has strictly less blame than $\mathcal{S}$* **and** *e'* **is a value**.

If we use this corollary instead of blame erasure, we have that the intermediate expression e_2 in our chain of reductions $e_1, \mathcal{S} \to e_2, \mathcal{S}'$ and $e_3, \mathcal{S}' \to^* e_4, \mathcal{S}''$ from above is a value. With that, we can construct a *sequence* of expressions to prove reclamation: If we take x to be e_1; e_3, x' to be e_4, and $\mathcal{S}^{\mathcal{R}}$ to be $\mathcal{S}''$, we have the following sequence of reductions:

$$
\begin{array}{ll}
e_1;\, e_3, \mathcal{S} \to e_2; e_3, \mathcal{S}' & \text{since } e_1, \mathcal{S} \to e_2, \mathcal{S}' \\
e_2; e_3, \mathcal{S}' \to e_3, \mathcal{S}' & \text{since } e_2 \text{ is value} \\
e_3, \mathcal{S}' \to^* e_4, \mathcal{S}'' & \text{by inductive hypothesis}
\end{array}
$$

And in the above reduction sequence, $\mathcal{S}''$ is blame-free, thereby completing our proof of reclamation.

When discussing when to tag locations with blame, we stressed that only unblamed location may gain blame, symbolizing that we couldn't know for sure which unsafe operation might be the cause of an error, should it arise. As it happens, with a good choice of operation, we can discern the true source of runtime errors. The next result addresses this.

Theorem 5.2.4 (Fault Localization) *If an expression e in runtime environment $\mathcal{S}$ reduces to the error expression with blame β, then either β identifies the most recent unsafe C operation, or a new program can be generated which isolates the source of the error.*

To show this, consider the following code snippets:

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

```
cCall1(p)
print(p.x)
cCall2(p)
print(p.x)
cCall3(p)
print(p.x)
```

```
p = calloc Point
```

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 guilty 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 guilty operation.

More formally, we can show fault localization with targeted applications of blame erasure. If an access to location n reduces to error with blame β, then there are two cases. First, β might identify the last unsafe operation, in which case we are done. Otherwise, it identifies an earlier operation; in this case, we can apply blame erasure *right before* the last unsafe operation, and generate an access to n. If this access succeeds, then the unsafe

operation following it *must* be the source of the error, as accessing n did not crash the program before it occurred. If this access fails, we repeat the argument, starting at the newly generated call, and work our way up to the operation identified by β.

In a rather extreme example, one could envision a scenario where a programmer accesses *all* of the C store after each C call, to see which locations were corrupted by the call. While unrealistic, this could be viewed as a way to debug these sorts of programs, as if we observe the set of all locations which might have been tampered with by an unsafe C operation, we can discover the effect that it had on the overall system.

To briefly summarize, our statements of preservation and progress together make up type soundness for Poseidon Lua in the absence of blame, and reclamation asserts the existence of a program execution which erases all blame from the runtime environment, reclaiming determinism and type soundness. Essentially, we have determinism and type soundness until a C call, allocation, or downcast occurs, as they introduce blame, but we can erase this blame thanks to reclamation. In the event of a runtime failure, fault localization assures us that we can generate a new program which can isolate the unsafe reduction which caused the failure.

Chapter 6

Conclusions and Future Work

In this thesis, we explored a new technique to reason about language composition, which emphasizes the interoperation of the two languages, rather than the languages themselves. We drew a correspondence between gradual typing and foreign function interfaces, specifically that the typed/untyped relationship shares some similarities with the host language/guest language relationship when the guest language has fewer guarantees than the host. While we gave a formal specification for Poseidon Lua, and proved results about it, this is merely an example of how our technique works, and nothing specific to C or Typed Lua were at play (other than the soundness disparity between both languages).

That said, C is a great candidate for analysis, since it is hugely expressive and has interesting unsafe behaviour. In addition, C FFIs are among the most popular, with several languages including Python, Julia, and Matlab having direct language support for C FFIs. As we emphasized early in this thesis, C is a fast language, and often performance critical code is written in C and called from another language; indeed, this is frequent in scientific computation. We believe that blame is a useful technique to reason about such C FFIs, as the unruly nature of C can be cleanly captured with blame, without needing to resort to a very involved semantics for C.

In addition to blame, we showed how one can use a *nondeterministic* operational semantics to obtain useful results. This was necessary given our decision not to model C, but it is an interesting approach in and of itself, and there are other systems beyond C which might benefit from nondeterminism: For example, JavaScript programs using the jQuery framework cannot be sure of the data that jQuery retrieves, and could use nondeterminism to capture that. In fact, nondeterminism may be useful for reasoning about situations where calls are being made to a library whose code is unavailable, as if you can

quantify the possible behaviours of the function, nondeterminism could cleanly capture those possibilities and open the door for some formal reasoning.

One might be hesitant to use such a nondeterministic semantics, as proof statements need to be carefully constructed to ensure that they actually have meaning. For example, in Chapter 5 we explored how we carefully constructed our statement of type soundness for Poseidon Lua; while nondeterminism does weaken classical statements (such as preservation and progress), that isn't to say that we can't make other interesting assertions. Indeed, we showed that when our formal specification for Poseidon Lua is nondeterministic, there exists a sequence of reductions which reclaims determinism and recovers type soundness. This gives users an understanding of where their system stands with respect to soundness when making FFI calls: Essentially, until they have observed all of the effects of a call, they tread on uncertain ground.

In short, this thesis presented a formal specification of Poseidon Lua, a fast implementation of Typed Lua with a C FFI. The operational semantics are nondeterministic, which proved to be immensely useful for proving things about it, and we believe that this flavour of semantics is applicable well beyond this small example. Additionally, we applied techniques from gradual typing, adapting in particular the notion of blame: Ultimately, blame allowed us to quantify the effects of black-boxes in our system. In the world of gradual typing, untyped values are something of a wildcard, and they introduce the possibility of runtime type errors in statically typed code. And in the realm of FFIs, calls to functions written in a relatively unsafe language introduce new ways in which the overall program can fail. We exploited this correspondence, and showed conditional type soundness of Poseidon Lua and that C is always to blame for runtime errors in Typed Lua. Further, we showed that if our formal semantics is in a state of nondeterminism, it is possible to recover both determinism and type soundness.

We believe that nondeterministic operational semantics are a useful tool for formal reasoning, beyond the scope of Poseidon Lua and perhaps even beyond reasoning about FFIs. In any situation where something's *effects* are important, more so than its operation, nondeterminism allows one to reason at a higher-level without getting bogged down in the minutia of its operation. Reasoning about concurrent systems is one such area, where we might be able to make analogies between blaming calls to blaming threads. While one does need to carefully craft proof statements and other definitions, the payoff is extraordinary: After all, we were able to quantify the effects of C on another language *without modeling it*.

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.

[2] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strnisa, Jan Vitek, and Tobias Wrigstad. Thorn—robust, concurrent, extensible scripting on the JVM. In *OOPSLA*, 2009.

[3] CompCert. CompCert Main Page. http://compcert.inria.fr. Accessed: 2018-07-23.

[4] Facebook. Flow language documentation. http://flowtype.org/docs/.

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

[6] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. *ACM SIGPLAN Notices*, 37(9):48–59, 2002.

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

[8] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Mikel Luján. Cross-language interoperability in a multi-language runtime. *ACM Trans. Program. Lang. Syst.*, 40(2):8:1–8:43, May 2018.

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

[10] 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.

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

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

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

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

[15] Lua. Lua 5.3 documentation. https://www.lua.org/manual/5.3/. Accessed: 2018-06-20.

[16] Lua. Lua Standard Compiler Documentation. https://www.lua.org/manual/5.3/luac.html. Accessed: 2018-08-01.

[17] 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.

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

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

[20] Microsoft. Typescript – language specification version 1.8. Technical report, Microsoft, January 2016.

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

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

[23] Benjamin C Pierce. *Types and programming languages*. MIT Press, 2002.

[24] Python. CFFI Documentation. https://cffi.readthedocs.io/en/latest/. Accessed: 2018-07-06.

[25] 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.

[26] 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.

[27] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, New York, NY, USA, 2016. ACM.

[28] 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.

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

APPENDICES

Appendix A

Grammars, Typing Rules, and Reduction Rules

A.1 Typed Language

te	$::=$	v_t	*value*
	$\mid$	$\{s_1 = v_1, ..., s_n = v_n\}$	*table*
	$\mid$	$\mathbf{let}\, x : T := te_1\, \mathbf{in}\, te_2$	*let binding*
	$\mid$	$x := te$	*variable update*
	$\mid$	$\mathbf{loc}\, n := te$	*location update*
	$\mid$	$\mathbf{deref}\, te$	*Lua dereference*
	$\mid$	$te_1\, op\, te_2$	*binary operation*
	$\mid$	$te_1(te_2)$	*function call*
	$\mid$	x	*variable*
	$\mid$	$te_1.te_2$	*dot access*
	$\mid$	$te_1.te_2 := te_3$	*dot update*
	$\mid$	$\mathbf{cast}\, te\, T_C$	*C downcast*
	$\mid$	$\mathbf{calloc}\, T_C\, n$	*C allocation*
	$\mid$	$\mathbf{deref_C}\, te$	*C deref*
	$\mid$	$te_1;\ te_2$	*sequence*

v_t	$::=$	$\mathbf{nil}$	*nil value*
	$\mid$	r	*register*
	$\mid$	c	*constant*
	$\mid$	$\mathbf{loc}\, n$	*Lua location*
	$\mid$	$\lambda x : T.te$	*Lua function*
	$\mid$	$\mathbf{cfun}\, T_C\, n$	*C function*
	$\mid$	$\mathbf{ptr}\, n\, T_C$	*C pointer*
r	$::=$	$\mathbf{reg}\, n$	*table store loc*
c	$::=$	n	*number*
	$\mid$	b	*boolean*
	$\mid$	s	*string*
op	$::=$	$+, -, *, /$	*arithmetic*
	$\mid$	$\leq, <, \geq, >$	*order*
	$\mid$	$\wedge, \vee$	*boolean*
	$\mid$	$..$	*concatenation*
	$\mid$	$==$	*equality*

A.2 Runtime/Untyped Language

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

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

A.3 Type System

$$
\begin{array}{rcll}
T & ::= & \mathbf{nil} & \textit{nil type} \\
 & | & \mathbf{value} & \textit{top type} \\
 & | & \mathbf{ref}\, T & \textit{reference type} \\
 & | & T_1 \cup T_2 & \textit{union type} \\
 & | & L & \textit{literal type} \\
 & | & B & \textit{base type} \\
 & | & T_1 \to T_2 & \textit{function type} \\
 & | & \{f_1, ..., f_n\} & \textit{table type} \\
 & | & \mathbf{ptr_L}\, T_C & \textit{C type} \\
T_C & ::= & \mathbf{int} & \textit{C integer type} \\
 & | & T_C^1 \to T_C^2 & \textit{C function type} \\
 & | & \mathbf{ptr_C}\, T_C & \textit{C pointer type} \\
 & | & \{s_1 : T_C^1, ..., s_n : T_C^n\} & \textit{C struct type}
\end{array}
$$

$$
\begin{array}{rcll}
f & ::= & s : T & \textit{fields} \\
 & | & \mathbf{const}\ s : T & \textit{const fields} \\
L & ::= & < booleans > & \textit{literals} \\
 & | & < numbers > & \\
 & | & < strings > & \\
B & ::= & \mathbf{boolean} & \textit{base types} \\
 & | & \mathbf{number} & \\
 & | & \mathbf{string} &
\end{array}
$$

A.4 Extra Typing Rules

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

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

$$\frac{\begin{array}{c}\Gamma, K \vdash te_1 : \textbf{number} \rightsquigarrow e_1 \\ \Gamma, K \vdash te_2 : \textbf{number} \rightsquigarrow e_2 \\ op \in \{<, \leq, >, \geq\}\end{array}}{\Gamma, K \vdash te_1\, op\, te_2 : \textbf{boolean} \rightsquigarrow e_1\, op\, e_2} \quad \text{(TT\_Binop\_Order)}$$

$$\frac{\begin{array}{c}\Gamma, K \vdash te_1 : \textbf{boolean} \rightsquigarrow e_1 \\ \Gamma, K \vdash te_2 : \textbf{boolean} \rightsquigarrow e_2 \\ op \in \{\wedge, \vee\}\end{array}}{\Gamma, K \vdash te_1\, op\, te_2 : \textbf{boolean} \rightsquigarrow e_1\, op\, e_2} \quad \text{(TT\_Binop\_Bools)}$$

$$\frac{\begin{array}{c}\Gamma, K \vdash te_1 : \textbf{string} \rightsquigarrow e_1 \\ \Gamma, K \vdash te_2 : T_2 \rightsquigarrow e_2 \\ T_2 \in \{\textbf{string}, \textbf{number}\}\end{array}}{\Gamma, K \vdash te_1 \,..\, te_2 : \textbf{string} \rightsquigarrow e_1 \,..\, e_2} \quad \text{(TT\_Binop\_String)}$$

$$\frac{\begin{array}{c}\Gamma, K \vdash te_1 : T_1 \rightsquigarrow e_1 \\ \Gamma, K \vdash te_2 : T_2 \rightsquigarrow e_2\end{array}}{\Gamma, K \vdash te_1 \;==\; te_2 : \textbf{boolean} \rightsquigarrow e_1 \;==\; e_2} \quad \text{(TT\_Binop\_Eq)}$$

$$\frac{\begin{array}{c}\Gamma, K \vdash te_1 : T_1 \rightsquigarrow e_1 \\ \Gamma, K \vdash te_2 : T_2 \rightsquigarrow e_2\end{array}}{\Gamma, K \vdash te_1;\, te_2 : T_2 \rightsquigarrow e_1;\, e_2} \quad \text{(TT\_Sequence)}$$

A.5 Extra Reduction Rules

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

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

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

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

$$\frac{\begin{array}{c} \mathit{value}\,(e_1) \\ e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e'_2 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \,/\, \beta' \end{array}}{\mathbf{rawget}\, e_1\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to \mathbf{rawget}\, e_1\, e'_2 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \,/\, \beta'} \text{ (R\_Rawget\_Step\_2)}$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e'_1 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \,/\, \beta'}{\mathbf{rawset}\, e_1\, e_2\, e_3 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to \mathbf{rawset}\, e'_1\, e_2\, e_3 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \,/\, \beta'} \text{ (R\_Rawset\_Step\_1)}$$

$$\frac{\begin{array}{c} \mathit{value}\,(e_1) \\ e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e'_2 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \,/\, \beta' \end{array}}{\mathbf{rawset}\, e_1\, e_2\, e_3 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to \mathbf{rawset}\, e_1\, e'_2\, e_3 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \,/\, \beta'} \text{ (R\_Rawset\_Step\_2)}$$

$$\frac{\begin{array}{c} \mathit{value}\,(e_1) \qquad \mathit{value}\,(e_2) \\ e_3 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e'_3 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \,/\, \beta' \end{array}}{\mathbf{rawset}\, e_1\, e_2\, e_3 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to \mathbf{rawset}\, e_1\, e_2\, e'_3 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \,/\, \beta'} \text{ (R\_Rawset\_Step\_3)}$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e'_1 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \,/\, \beta'}{e_1(e_2) \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e'_1(e_2) \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \,/\, \beta'} \text{ (R\_Fun\_App\_Step\_1)}$$

$$\frac{\begin{array}{c} \mathit{value}\,(e_1) \\ e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e'_2 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \,/\, \beta' \end{array}}{e_1(e_2) \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e_1(e'_2) \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \,/\, \beta'} \text{ (R\_Fun\_App\_Step\_2)}$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e'_1 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \,/\, \beta'}{e_1\, op\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e'_1\, op\, e_2 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \,/\, \beta'} \text{ (R\_Binop\_Step\_1)}$$

$$\frac{\begin{array}{c} \mathit{value}\,(e_1) \\ e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e_2' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V' \,/\, \beta' \end{array}}{e_1\, op\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e_1\, op\, e_2' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V' \,/\, \beta'} \text{ (R\_Binop\_Step\_2)}$$

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

$$\frac{e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V' \,/\, \beta'}{\mathbf{cget}\, e\, o\, T \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to \mathbf{cget}\, e'\, o\, T \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V' \,/\, \beta'} \text{ (R\_Cget\_Step)}$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e_1' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V' \,/\, \beta'}{\mathbf{cset}\, e_1\, o\, e_2\, T \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to \mathbf{cset}\, e_1'\, o\, e_2\, T \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V' \,/\, \beta'} \text{ (R\_Cset\_Step\_1)}$$

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

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e_1' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V' \,/\, \beta'}{e_1;\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e_1';\, e_2 \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V' \,/\, \beta'} \text{ (R\_Seq\_Step\_1)}$$

$$\frac{\mathit{value}\,(e_1)}{e_1;\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta \to e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/\, \beta} \text{ (R\_Seq\_Step\_Through)}$$