Pontifícia Universidade Católica
do Rio de Janeiro

Gabriel Coutinho de Paula

A Foreign Function Interface For Pallene

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Roberto Ierusalimschy

Rio de Janeiro
September 2021

Pontifícia Universidade Católica
do Rio de Janeiro

Gabriel Coutinho de Paula

A Foreign Function Interface For Pallene

Dissertation presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the Examination Committee:

Prof. Roberto Ierusalimschy
Advisor
Departamento de Informática – PUC-Rio

Prof. Francisco Figueiredo Goytacaz Sant'Anna
UERJ

Profa. Noemi de La Rocque Rodriguez
Departamento de Informática – PUC-Rio

Rio de Janeiro, September 23rd, 2021

All rights reserved.

Gabriel Coutinho de Paula

Graduated in computer engineering by the Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio).

Bibliographic data

Coutinho de Paula, Gabriel

A Foreign Function Interface For Pallene / Gabriel Coutinho de Paula; advisor: Roberto Ierusalimschy. – 2021.

79 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2021.

Inclui bibliografia

1. Informática – Teses. 2. Interface de Função Externa. 3. Lua. 4. Compiladores. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my parents, for providing nurture in a hostile world.

To my grandmother, Regina Cœli. A spirit of intellect who instilled in me unending yearning for knowledge.

To my mentors. If I have seen further it is by standing on the shoulders of Giants.

Acknowledgments

I'd like to express my gratitude to my advisor Professor Roberto Ierusalimschy for all his care, wisdom and patience. Through his counsel I created projects beyond what I thought I was capable of, and accomplished things above what I thought I could. For that I'm eternally grateful, and will forever cherish our relationship. Thank you, professor.

I'd also like to thank CNPq and PUC-Rio for the aids granted, without which this work couldn't have been accomplished.

This study was financed in part by the Coordenação de Aperfeiçoamento Pessoal de Nível Superior (CAPES) — Finance Code 001

Abstract

Coutinho de Paula, Gabriel; Ierusalimschy, Roberto (Advisor). **A Foreign Function Interface For Pallene**. Rio de Janeiro, 2021. 79p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Pallene is a statically typed subset of the Lua programming language, designed to act as a system-language counterpart to Lua's scripting, and used to write lower-level libraries and extension modules for Lua. In this sense, Pallene is a companion language, always intended to be used side-by-side with Lua, sharing its runtime.

Pallene, both as a system-language counterpart to Lua and as a bridge between Lua and foreign languages, must provide a mechanism to interact with libraries and system services written in a low-level language like C. To this end, we present a Foreign Function Interface (FFI) design and implementation for Pallene, which allows for calling external functions that follow C calling conventions and manipulating C data representations directly.

Our design balances flexibility and safety following an empirical approach, wherein we prefer to sacrifice hypothetical flexibility unless we see a real use case that forces us to compromise on safety. Our implementation aims to be as portable as Pallene, as well as performant and simple. Since Pallene's implementation already depends on a C compiler, the FFI aggressively uses the already available C compiler to its advantage. This way, we can directly use foreign libraries' header files, enabling the FFI to verify the correctness of function bindings and to use macro definitions.

Keywords

Foreign Function Interface; Lua; Compilers.

Resumo

Coutinho de Paula, Gabriel; Ierusalimschy, Roberto. **Uma Interface de Funções Externas para Pallene**. Rio de Janeiro, 2021. 79p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Pallene é um subconjunto estaticamente tipado da linguagem de programação Lua, projetada para atuar como uma linguagem de sistemas em contraparte ao scripting de Lua, e usada para escrever bibliotecas de baixo nível e módulos de extensão para Lua. Nesse sentido, Pallene é uma linguagem companheira, usada sempre lado a lado com Lua, compartilhando seu runtime.

Pallene, tanto como uma linguagem de sistema em contraparte a Lua e como uma ponte entre Lua e outras linguagens, deve fornecer um mecanismo de interação com bibliotecas e serviços de sistema escritos em uma linguagem de baixo nível como C. Para este fim, apresentamos um design e implementação de Interface de Função Estrangeira (IFE) para Pallene, que permite chamar funções externas que seguem as convenções de chamada C, e manipular representações de dados C diretamente.

Nosso design equilibra flexibilidade e segurança seguindo uma abordagem empírica, em que preferimos sacrificar flexibilidade hipotética a menos que vejamos um caso de uso que nos faça comprometer em segurança. Nossa implementação tem como objetivo ser tão portátil quanto o Pallene, além de ter um bom desempenho e ser simples. Uma vez que a implementação de Pallene já depende de um compilador de C, a nossa IFE usa agressivamente o compilador de C já disponível para sua vantagem. Dessa forma, podemos usar diretamente os arquivos de cabeçalho de bibliotecas externas, permitindo que a IFE verifique a exatidão das ligações de função externa e que use macros de C.

Palavras-chave

Interface de Função Externa; Lua; Compiladores.

Table of contents

List of tables

List of codes

List of Abreviations

ABI – Application Binary Interface

AOT – Ahead-Of-Time

API – Application Programming Interface

FFI – Foreign Function Interface

IDL – Interface Description Language

LLC – Last-Level-Cache

JIT – Just-In-Time

A computer lets you make more mistakes faster than any other invention in human history, with the possible exceptions of handguns and tequila.

Mitch Ratcliffe, *Technology Review, April 1992.*

1
Introduction

Extending programming languages with functionalities written in a different programming language is often necessary in software development. In particular, practical scripting languages—while carrying out the role of glueing other software components together (OUSTERHOUT, 1998)—must be able to interoperate with other languages.

External code comes in many different flavours. Nonetheless, the popular 'least common denominator' is plain C libraries. Furthermore, many languages implement ways to expose their code as a C interface. In this sense, C calling conventions can be seen as a *lingua franca* of sorts. As such, for the purpose of interfacing with foreign code, we are mostly interested in C.

1.1
Foreign Function Interface

Foreign function interface (FFI) is a mechanism for a program written in one programming language—the host language—to call functions and manipulate objects from another programming language—the guest language. The interface has to account for differences in calling conventions and data representation between the host and the guest languages, as well as differences in the management of memory and other resources.

In particular, we're interested in the FFIs where the host language is a high-level language (such as scripting languages) and the guest language is a low-level language like C. This raises a non-trivial impedance mismatch in data representation between the two languages that any FFI design has to address.

At a high level, the process of invoking a foreign function involves three steps. First, the FFI must convert the arguments of the call from the host to the guest language representation. Next, the FFI must handle the transfer of control from host language to guest language, and then back. Finally, the FFI has to convert the guest language representation of the results into their corresponding host language representation (FISHER; PUCELLA; REPPY, 2000; FISHER; PUCELLA; REPPY, 2001). Note that this back and forth conversion is entirely dependant on each language's *data representation*. If the representations are the same, the conversion is trivial.

Although called "foreign *function* interfaces", we are not just interested in functions. Representing foreign types, manipulating foreign objects and their lifetimes, as well as accessing macro procedures and constants are also part of

the foreign interface. Here are some examples of external things we may want to do:

- Represent and manipulate foreign types;
- Support abstract foreign objects like handles;
- Call foreign functions as native host-language functions;
- Access constants, which may include enumerations, function pointers, and preprocessor definitions;
- Invoke C preprocessor macro procedures;
- Pass callbacks.

1.2 The Pallene Programming Language

Pallene is a statically typed subset of the Lua programming language, designed to act as a system-language counterpart to Lua's scripting, and used to write lower-level libraries and extension modules for Lua. As such, Pallene is a companion language, always intended to be used side-by-side with Lua, sharing its runtime (GUALANDI; IERUSALIMSCHY, 2018; GUALANDI; IERUSALIMSCHY, 2020; GUALANDI, 2020).

The language was also designed for performance. Pallene's ahead-of-time compiler uses type annotations to generate type specialised code that is more performant that its Lua equivalent. The compiler generates C code, which is then handed to a C compiler.

Pallene provides the gradual guarantee (SIEK et al., 2015). If we take a Pallene program and remove all type annotations, it becomes valid Lua code. And if we transform a Lua program into a Pallene program by adding type annotations, the type annotations cannot affect the evaluation of the program, except perhaps by introducing runtime type errors. As such, the combination of Lua and Pallene can be seen as a gradually typed language.

Since Pallene is a companion language to Lua, the two languages must be seamlessly integrated. Consequently, types in Pallene mirror Lua; every value in Pallene must exist in Lua, and we must be able to pass arbitrary Lua values to Pallene. However, there's a typing mismatch between the two languages: Lua is dynamically typed and Pallene is statically typed. This discrepancy is addressed by Pallene's runtime tag-checker; once a Lua value crosses language boundaries, its tag is checked at run-time. If there's a tag mismatch, Pallene raises an error.

The relevant Pallene types for this work are the following:

integer A 64-bit signed integer, being the same as Lua's own integer values. Although signed, a careful programmer may use these to store unsigned integers, being particularly attentive with comparisons. Tag-checking is straightforward: Pallene just examines whether the value is a Lua integer.

float A double-precision floating-point number, being the same as Lua's own floating point values. Tag-checking is straightforward: Pallene just examines whether the value is a Lua floating-point number.

arrays A collection containing contiguous values of the same type, starting at one. Underneath, this is a regular Lua table interpreted as an array. When tag-checking, Pallene first checks whether it is a table, and lazily checks whether the elements have the correct tag when they are indexed. This means an array with mismatched element types will fool Pallene's tag-checker if the offending entries are never accessed.

Lua records A record type, consisting of a collection of fields, each typed as well. Underneath, Lua records are regular tables interpreted as a record. This means accessing fields is done by indexing a table with a string value containing the field's name. When tag-checking, Pallene first checks whether it is a table, and lazily checks whether the fields have the correct tag when they are accessed. This means a record with mismatched fields will fool Pallene's tag-checker if we never access them.

Pallene records A record type, consisting of a collection of fields, each typed as well. Underneath, this is a Lua userdata containing a normal structure. This means accessing fields is done directly, and does not require indexing a Lua table. Pallene records are a more performant alternative to Lua records, since they avoid indexing the hash part of a table. For tag-checking, Pallene creates a unique metatable for each record type. When a Lua value crosses into Pallene, the tag-checker first examines whether it is a userdata, and then checks whether its metatable matches the unique table for that type. This approach to add new types to Pallene is one we'll use later.

There are other types like booleans, functions and `any`. However, they aren't of particular importance for this work.

1.3 Objective

On this work, we analyse some examples of FFIs and present our own design and implementation for Pallene. Our goal is to investigate an all-rounded solution to this problem of interfacing with foreign code.

There are two related traits of Pallene that are relevant for an FFI. The first is static typing, which yields a lower impedance for C interoperability when compared with a dynamically typed language. The second is its ahead-of-time compiler that targets the C programming language. This means a C compiler is already a dependency for Pallene during compilation, which gives us many opportunities to improve the design and greatly simplifies the implementation of the foreign interface. Although we created the FFI for Pallene, the proposed design should not be restricted to Pallene, and can be adopted by languages with a similar profile.

We are interested in using FFIs to extend Pallene programs with statically typed external libraries. The foreign code we want to interact with is writ in stone: it's known by the programmer before execution, and the programmer has a fixed definition of the external libraries at hand. It's worth keeping in mind that Pallene is statically typed. Consequently, this design choice is quite natural in the context of the language.

There are other kinds of foreign function interfaces which do not have this restriction, such as `libffi` (GREEN, 1996). They are more generic in the sense that the foreign interface doesn't need to be static, enabling dynamic interactions with external libraries. On these FFIs, the interface is determined at run-time and not fixed on the source code, allowing for arbitrary functions to be added during execution. This kind of interaction is inherently platform dependant, and cannot be done purely in C.

Many FFIs are built on top of `libffi`, and for good reason: it's very mature, a lot of work has been put into it, and it supports a wide range of platforms. However, we are not interested in dynamic FFIs; we are interested in interacting with fixed, static external libraries. The dynamism of `libffi` comes with issues related to portability and performance; it's fairly portable but not as much as C, and it's not particularly fast. Since Pallene is a statically typed language that generates C code, and a C compiler is already a dependency, we can do better than building on top of `libffi`.

The rest of this document is structured as follows. In Chapter 2 we contextualize our work: we discuss our motivations for building an FFI for Pallene, propose a set of criteria for analysing FFIs, and introduce other important concepts we'll need later on. In Chapter 3 we examine the design

and implementation of other relevant FFIs. In Chapter 4 we propose an FFI design for Pallene, and evaluate it under our criteria. In Chapter 5 we discuss the implementation of our proposed design, and also evaluate it under our criteria. Finally, in Chapter 6 we give our final remarks and summarize our contributions.

1.4 Motivating example

Before we move on, we'll show an example of a external library one may wish to interact with, and a sneak peek of our design for calling this library. We believe that showing a concrete example now may better anchor and contextualize the discussions in the following chapters.

The example we chose is from `zlib`. We chose `zlib` because it involves dealing with pointers and buffers, which are interesting features the FFI must support. Specifically, our example is a few convenience compression functions. Code 1 shows these function declarations, simplified for didactic purposes (we've expanded by hand the macros and `typedefs` contained in these function signatures).

The standard use of this API involves calling two different functions. The first function, `compressBound`, takes the size of the input we wish to compress, and returns the maximum size this input may have when compressed. We need this to allocate a buffer which is large enough to fit the compressed input. The second function, `compress2`, does the actual compression. It takes a buffer where it will write the compressed input, an integer pointer containing the size of this buffer (which the function will then write the real size of the result), the input, the size of the input, and the level of compression. It will return a number, containing the status of the computation (in this case, we're only interested in the `Z_OK` return code).

Code 1: zlib_compress.h

```
1 #define Z_OK 0
2
3 unsigned long compressBound(unsigned long sourceLen);
4
5 int compress2(
6   char *dest,
7   unsigned long *destLen,
8   const uint8_t *source,
9   unsigned long sourceLen,
10  int level
11 );
```

Every FFI interaction begins with a description of the foreign code, which is fixed and known by the programmer, as discussed before. Code 2 shows this interface description in Pallene. We start by including the header file provided by the library. Then, we declare a constant with the name `Z_OK`, which will contain the value of the constant named `Z_OK`, defined in the library's header file. Finally, we declare the functions, providing their name and their type. Of note, there's the `C{cuchar}` parameter in line 8, which is a CArray (a new type we added to Pallene, here acting as a buffer) containing elements of C type `unsigned char`, and the `retinout` modifier on a parameter of C type `unsigned long` in line 9, which says the function will receive an integer and return an extra value also of type integer (under the hood, the C function receives a pointer containing data and writes to it).

Code 2: zlib_compress_decl.pln

```
1 foreign
2   include <zlib.h>
3
4   constant Z_OK: integer
5
6   function compressBound(culong): culong
7   function compress2(
8     C{cuchar},
9     retinout culong,
10    string,
11    culong,
12    cint
13  ): cint
14 end
```

Code 3 shows Pallene using this API. We start by querying the required buffer size on line 2. Then, on line 3, we instantiate a buffer with this size, where all the elements are initialized with zero. Finally, we call the actual compression function. Note that it returns two values; the second comes from the `retinout` parameter we described on the function's signature.

Code 3: zlib_compress.pln

```
1 local function pln_compress(txt: string): (C{cuchar}, integer)
2   local n = ffi.compressBound(#txt)
3   local buf: C{cuchar} = C{n, 0}
4   local res, buflen = ffi.compress2(buf, n, txt, #txt, 9)
5   assert(res == ffi.Z_OK)
6   return buf, buflen
7 end
```

2
Background

There are two main motivations for calling external libraries from Pallene. The first is Pallene being used by Lua to call foreign functions, acting as a more convenient alternative to Lua's C API. In this sense, Pallene can be seen as bridge between Lua and other languages. The second is for Pallene as a system-language counterpart of Lua. The ability for Pallene to interact with the underlying operating system and other existing libraries is an important feature for the language.

Due to this use as an alternative to the Lua C API, we'll start with a review of this API.

2.1
Extending and Embedding Lua

Lua is often described as an extensible extension language (IERUSALIMSCHY; FIGUEIREDO; FILHO, 1996). This means we can easily extend a Lua program with external code, and that we can embed Lua in other host applications. These two views match up with the two kinds of interaction between C and Lua:

- Lua is the host and has the control, and C is the library. The C code in this kind of interaction is called the *library code*.
- C is the host and has the control, and Lua is the library. The C code in this kind of interaction is called the *application code*.

Lua tackles these interactions through its C API. The C API is the set of functions, constants, and types that allow C code to interact with Lua. Virtually anything that Lua code can do can also be done by C code through the API (IERUSALIMSCHY; FIGUEIREDO; FILHO, 2005; IERUSALIMSCHY; FIGUEIREDO; CELES, 2011; IERUSALIMSCHY, 2016).

To understand the C API, we must first discuss how data is transferred across languages. In C, we interact with the Lua runtime through the *Lua state*, an opaque object (sometimes called a *handle*) that encapsulates the state of the interpreter. Data flows between C and Lua through a data structure contained inside this state; this data structure is a stack of Lua values, which we'll call *the virtual stack*. A key property of the API is that it offers no way for C code to refer directly to Lua objects; any values manipulated by C code must be on the virtual stack.

Calling a Lua function from C starts with pushing it into the stack (remember that functions in Lua are first-class; they are a value like any other), then all its arguments, and finally asking the API to perform the call, specifying the number of arguments pushed. For example, if we wish to call the Lua `print` function from C, we start by pushing it into the virtual stack with `lua_getglobal`, passing the string `"print"` to specify the name of the global variable that stores the function we wish to call. Then, we push the actual string we wish to print, by calling `lua_pushstring`. We finish by calling `lua_call`, indicating the number of arguments we pushed on top of the function (in this case, one argument). This will execute the `print` function, passing it a single string. The return values of this function call will be on the stack.

From the Lua side of things, calling C functions is not special; we invoke them the same way we invoke other Lua functions. To Lua, the boundaries with C are opaque. From the C side of things, however, functions invoked by Lua must follow a special protocol. This protocol starts with fixing the C function's type; it receives the Lua state as its single argument, and returns an integer containing the number of return values. The arguments we passed from Lua are on the virtual stack; we manipulate the stack and the values contained in it through the API. Usually, C functions called by Lua start by extracting the Lua arguments from the stack, type-checking them, and finish by pushing the results back to the stack and returning the number values pushed.

FFIs vs APIs

A Lua FFI would serve the same purpose as the API regarding the first view: using C as library code. With regards to the second view, FFIs are not useful. They are a mechanism for extending a language, rather than embedding it.

The reference Lua implementation does not come with an FFI: extending Lua is done primarily through the C API. One of the reasons for the lack of an FFI is related to the interpreter's commitment to ISO C for greater portability. Since the language is dynamic, bindings can only be known at runtime. However, there is no mechanism in C for dynamically performing function calls; creating an FFI of this kind for the reference Lua implementation would require platform-dependant code, which violates the restriction of using only ISO C.

Nevertheless, there are several Lua FFI implementations available as libraries, such as (MCKASKILL, 2011) and (FACEBOOK, 2015). These two

FFIs follow the design of LuaJIT's FFI extension (PALL, 2005a). We'll go in more details on this design in Chapter 3. There's also the Alien FFI (MASCARENHAS, 2009), which is based on `libffi`. As discussed, they are less portable when compared to the C API.

The differences between the API and FFIs—with regards to extending Lua—are stark. We'll start the comparison with a description of the API. As discussed, from the Lua side, external functions are called like any other Lua function. However, from the C side, functions called by Lua must follow a simple protocol to exchange values: the function's signature is fixed, and Lua arguments are passed and Lua values are returned through the virtual stack.

All data crossing language boundaries are Lua values, with the C programmer in charge of converting them when needed. That is, the task of converting Lua values into C and converting C values back into Lua must be done at C, using the API's functionalities to interact with the virtual stack. This process may have considerable overhead; there's a performance cost in manipulating values in the virtual stack, and all interactions with Lua values must happen through it.

Besides performance, the downside of the C API is related to ease of use. If we want to interact with arbitrary foreign code through the API, we must first write the C glue code by hand. Note that this glue code must be statically compiled ahead-of-time, requiring a C compiler. There's a tool for Lua that generates the C glue code automatically, called the Simplified Wrapper and Interface Generator (SWIG) (BEAZLEY et al., 1996). However, the default behaviour of SWIG is not always enough and requires additional programmer-supplied hints, and the wrappers still have the overhead of the C API.

This is in contrast with FFIs for Lua. Although the overhead depends on the implementation, we have a distinct gain in ease of use: interacting with C code no longer requires writing C glue code by hand, nor a C compiler. This allows us to "stay in the fun world", instead of needing to write C code whenever we want to extend Lua with external libraries.

2.2 Extending Pallene

One way to use the Pallene foreign interface is merely as a bridge between Lua and external libraries. This is a role generally fulfilled by C: Lua has its own C API, which—among other things—enables Lua to call external libraries through C. However, the API has an overhead associated with it, and it can be inconvenient to use.

As an example, suppose we'd like to call the `atan2` foreign function from

Lua, as shown in Code 4. One way to to expose the `atan2` function from C's `math.h` is shown in Code 5. This function is already present in Lua's math library, and it is implemented in a fairly similar way.

Code 4: atan2.lua

```
local lib = require "atan2"
local pi = lib.my_atan2(0, -1)
```

Code 5: atan2_lib.c

```
#include <math.h>
#include "lua.h"
#include "lauxlib.h"

static int l_atan2(lua_State *L) {
  double y = luaL_checknumber(L, 1);
  double x = luaL_checknumber(L, 2);
  double angle = atan2(y, x);
  lua_pushnumber(L, angle);
  return 1;
}
```

Using Pallene as a bridge to call `atan2`, with our FFI design, is a much simpler affair, as shown in Code 6. It is also faster, because function calls from Lua to Pallene bypass the C API (GUALANDI; IERUSALIMSCHY, 2020). We'll discuss later how we will expose foreign functions to Pallene.

Code 6: atan2_lib.pln

```
export function atan2(y: float, x: float): float
  return ffi.atan2(y, x)
end
```

Another way to use the FFI is from the perspective of Pallene as a systems language. The role of writing lower-level libraries and extension modules for Lua is generally fulfilled by C. Pallene was designed to fulfill that same role and, as such, be a substitute of C in this regard.

For this reason, extending Pallene with existing external libraries is essential. Some Lua modules have to be written in C, because of dependencies on external libraries or system calls. With the foreign interface, these modules could be written in Pallene.

Finally, one curious use for a Pallene FFI is unrelated to extending the language. Since we want to achieve data-level interoperability in some capacity, we must give Pallene some mechanism to manipulate external objects.

These objects may be used by Pallene without actually interacting with foreign languages. As these structures tend to be faster, we can use them for performance critical parts of the code. This use is prevalent in LuaJIT (PALL, 2005b). There's a similar work on this, which adds faster data structures to Pallene (LIGNEUL, 2019). However, the goal was strictly making Pallene faster, and had nothing to do with extending the language.

2.3 Data-level interoperability

As we've discussed before, an FFI has to deal with foreign types in some way. This necessity goes both ways: we must pass Pallene values to C, and bring C values back into Pallene. Reiterating, whenever values cross language boundaries, we must convert them. And the way we convert values is entirely dependant on their representation.

The C data representation is fixed. Our goal with an FFI is to extend the host language with external libraries, which cannot be changed for our convenience. As such, the only malleability available to us is at the host language.

One approach to deal with foreign objects is through *marshalling* and *unmarshalling*. We use existing host language types as surrogate for the low-level types, translating between the two when needed: high-level values are marshaled into their low-level equivalent, and low-level values are unmarshaled into their high-level equivalent.

This approach works well with scalars. However, we may run into issues when translating with more complicated types such as C structures, arrays, and pointers. In particular, larger volumes of data cannot be feasibly marshaled and unmarshaled. In such cases, the host language needs to have direct access to these foreign objects, requiring some mechanism for data-level interoperability (FISHER; PUCELLA; REPPY, 2000; FISHER; PUCELLA; REPPY, 2001; BLUME, 2001).

2.4 Criteria

To design and implement an FFI, we need to first define a set of criteria for analysing it. There are five criteria we are interested in evaluating: flexibility, safety, portability, performance and implementation complexity. The first two are related to the design, and the latter three to the implementation. Nevertheless, certain design decisions can make the trade-offs between the implementation criteria more palatable. We'll discuss them in turns.

We'll start with flexibility. Here we define flexibility as the capacity to interact with arbitrary foreign code. Ideally, we wish to stay within the host language as much as possible, bypassing the need to write specialized code in the guest language. This is in contrast with the Lua C API, in which extending the language requires writing C code by hand. We'll measure flexibility by analyzing commonly used C libraries and checking whether they can be directly used through the foreign interface.

On safety, we first must define what is a *safe language* and how safety can be compromised. In a safe language, the behaviour of a program can always be understood in terms of the language itself. In particular, errors are always detected and explained in terms of the language. No matter how buggy your code may be, crashes cannot be attributed to the language itself. Lua, for example, is a safe language. This is in contrast with C, in which many buggy programs can only be explained in terms of the underlying hardware. In other words, code in Lua can never crash the interpreter, whereas code in C can cause a segmentation fault.

When we extend Lua with external libraries, we can break its safety: arbitrary foreign code can cause arbitrary crashes. As such, Lua code that interacts with arbitrary external libraries loses its safety guarantees. Crashes are no longer restricted to bugs in the language implementation, they can also be caused by the foreign extension.

Therefore, there's a yellow caution tape at the boundary between the host and the foreign language. On one side we have safe Lua, on the other dangerous foreign code. Nonetheless, unsafe behaviour can still bleed into the safe parts of the language if we are not careful. For example, manipulating foreign objects inside Lua, such as raw pointers, may carry over harmful behavior even if this code does not interact with C directly.

The dangers of external libraries are outside the scope of a foreign interface. Nevertheless, while inside the host language, it is the responsibility of the foreign interface to provide safety. The goal is to keep the yellow caution tape at the gates, as restricted to foreign code as possible.

The Lua C API is safe in this sense. The yellow caution tape is strictly outside the host language, wherein only C code can cause a crash. To interact with arbitrary foreign code, the application programmer must write adapter glue code in C. Note that the crashes caused by C code, in the context of extending Lua with a foreign library, can either be at the adapter glue code or at the library itself.

This is in contrast with LuaJIT's FFI, in which interacting with foreign code does not require hand writing adapter glue code. However, LuaJIT's FFI

is not safe. Incorrect use of the interface may cause a crash at the host language. Parts of the host language are now within the yellow caution tape. Exposing more features of the guest language to the host language may give rise to safety issues. If we expose the entire foreign language we sacrifice too much safety, and if we expose too little we sacrifice flexibility.

This issue manifests itself when dealing with data-level interoperability. In particular, accessing memory that is not owned by the Lua run-time is a fundamentally unsafe operation. The FFI cannot be fully safe if we allow dereferencing arbitrary pointers, and it cannot be fully flexible otherwise. We'll see in Chapter 4 how our design balances this conflict.

Moving to the three implementation criteria, we'll start with portability. We're interested specifically in the FFI's *implementation* portability, in comparison with the language implementation. In other words, the FFI has to be as portable as Pallene. Since Pallene targets C and makes no use of platform-dependent code, the FFI has to follow the same restrictions.

Performance has to be considered carefully. It is hard to compare foreign interfaces implementations of different host languages. As such, for this work, we'll keep performance analysis restricted to the Lua world, comparing it with native C code. In any case, we are interested in measuring the overhead of invoking foreign functions, which involves the conversions from Pallene values to C and back, as well as the function call itself. We are also interested in measuring data-level interoperability performance.

Finally, implementation complexity can be a bit subjective to judge. Nonetheless, assertions can be made on this dimension without controversy when the difference of complexity spans an order of magnitude or more.

2.5 Specifying interfaces

Every FFI needs to specify, in some way, the interface of the external library we wish to interact with. There are multiple ways this can be done. Essentially, this specification is usually done through some kind of language, which describes the API of the external library. This description generally consists of foreign function declarations and foreign types, but can also include constant definitions and macro procedures. Note that a mismatch between the description and the concrete interface is one way safety may be compromised.

C header files

In C, the canonical way of specifying APIs is through header files. Curiously, there's nothing special about the language used in these files: it is just plain C. In practice, the distinction between the contents of `.h` files and `.c` files is merely convention: there are no rules enforced by the compiler.

The conventional use of C header files is for declaring external symbols like functions and global variables, and for defining macros and types we wish to make public. The actual definition of external symbols are done at some `.c` file. This way, header files can then be imported by other `.c` files, carrying over the interface's description. The task of joining these declarations to their implementation is done later by a linker. Note that any library that provides a binary interface following C calling conventions can be described by a C header file.

Library's header files are not strictly necessary in C. The application programmer could rely solely on the documentation of the library to interact with it; they could write external symbol declarations by hand. This is, however, tedious and prone to errors. It is much more convenient to use header files supplied by the library. The conventional use of header files is practical because we get compiler errors in the case of incompatible descriptions. This incompatibility of descriptions may happen both at the library level and at the application level. The former happens when the declaration does not match its implementation, and the latter when the declaration does not match its use. As such, header files act as a contract of sorts between libraries and applications. We'll assume this contract is sound from here on.

Back to FFIs, we aren't interested in specifying C libraries for other C applications: we want to specify C libraries to be used by other programming languages. In any case, header files can still be used to this end. Not only that, it is very convenient to use header files for this purpose: under our soundness assumption, we are sure description mismatches won't happen.

The Swift programming language (LATTNER, 2004) achieves C interoperability through the use of header files. The foreign interface is described through them, supplied by the library developer, which is used to generate all the necessary bindings. Similarly, SWIG for Lua creates glue code by reading header files. There are two issues with this approach, however.

The first is complexity. We need an entire C compiler front-end implementation, as well as a C preprocessor, to extract all useful information from this description. This is quite an undertaking. The second is ambiguity. Although enough for a C compiler, specifying an interface at a higher-level using

C is imprecise. Languages that are not C may need more information than what C provides. This is quite an important issue when our goal is to generate a safe and flexible FFI.

Interface description languages

Interface Description Languages (IDLs), sometimes called Interface Definition Languages, is a generic term for a family of small languages used to specify the interface of external libraries. They sit between the host and guest languages, and as such can be seen as being *language-independent* (except when we consider the IDL itself).

One common use of IDLs are in the context of remote procedure calls (RPCs). RPCs face many of the same problems of FFIs: specifying external interfaces, specifying external types, and of marshalling and unmarshalling data.

Description at the host language

Another way to specify APIs is inside the host language itself. This can be accomplished by adding a description sub-language inside the host language, which is responsible for describing the interfaces of external libraries. This approach is somewhat similar to IDLs, except it is not language-independent: it is tied to the host language.

Some languages with metaprogramming capabilities may be able to accomplish this without needing a description sub-language. We can use the host language itself to describe the interfaces of external libraries. An example of this is the Racket programming language foreign interface (BARZILAY; ORLOVSKY, 2004).

With this approach, since we are writing the description to be consumed by a specific host language, we can create richer specifications. As we'll see later, in some ways this allows for a safer FFI design.

However, we run into the issue of correctness. We have no verification that the bindings are correct, as there's no guarantee that the description matches the concrete interface. This pulls the yellow caution tape closer to the host language: its not just foreign code that is unsafe, but also the boundary between the languages. Also, we completely lose the ability to use C constants and preprocessor macros, which are frequently used in C libraries.

2.6 FFI implementations

Generally speaking, FFI implementations—like languages—can be classified as either ahead-of-time (AOT), just-in-time (JIT), or interpreted. A hybrid FFI implementation approach is also possible and not entirely uncommon. Note that the language itself may be implemented differently than the FFI.

The first two approaches, AOT and JIT, rely on code generation. To make interoperability possible in these cases, there are pieces of bridging *glue code* between the host and guest language, responsible for implementing FFI bindings. The difference between AOT and JIT concerns *when* the bindings are created:

- Ahead-of-time glue code is created and compiled statically, before running the program that intends to use them;
- Just-in-time glue code is created at run-time, while the application is running.

The interpreted approach, unlike the first two, does not rely on code generation. It shares the JIT approach property of creating bindings at run-time. However, instead of generating code, it dynamically performs the function call during the execution. This cannot be done purely in C, and requires assembly.

In any case, to create bindings we require low level knowledge of the guest language call conventions, as well as details about the architecture we are running under. Ahead-of-time FFIs greatly reduces complexity: this job is done by a C compiler, which generates the bindings statically, before the application is running.

2.7 External preprocessor macros

One feature rarely seen on foreign interfaces is access to C preprocessor macros. These are very commonly used in APIs, and generally comes in two flavours: constant definitions and procedures. It is generally hard to deal with macros because they don't exactly generate code. They simply do textual substitution at compile time. As such, there's no symbol to link against. They are not even typed.

2.8 Callbacks

Callbacks in C, as values, are function pointers with a signature defined by the library code, passed as an argument during a function call. There's a inversion of control in callbacks, in which the application becomes the client of the library. We call this process of passing a function pointer to the library as *registering* a callback, which may be invoked after the registration call site.

Callbacks are common in APIs and, as such, flexible FFIs must support them in some way. In the context of FFIs, callbacks constitute the opposite of the usual interaction: the guest language becomes the *caller*, and the host language the *callee*.

The flexible FFI must enable application programmers to pass high-level host language functions as callbacks. To this end, the FFI implementation first needs to create glue code to bridge the two languages. This glue code is in the form of low-level stub functions with the expected callback signature, wherein its address is passed as argument to the guest language during a foreign call. When invoked by the guest language, the stub function does the appropriate conversions and calls the correct high-level host language function.

As such, only FFI implementation schemas that generate code (*i.e.* AOT and JIT) are able to provide callbacks. The library `libffi`, for example, does common function calls using the interpreted schema, and when dealing with callbacks generates code just-in-time for stubs.

Callbacks are a particularly difficult type to represent when host language functions depend on some execution context, such as the runtime or captured variables in closures. In C, a callback is a simple function pointer; this extra data required by some languages is not readily available. Specifically in the case of Pallene, functions require a reference to the Lua state to interface with the Lua runtime. However, there's no standard way to pass the Lua state to the callback.

Just-in-time schemas solves this by tying the generated stubs to a data pointer at runtime. Ahead-of-time schemas don't have the benefit of creating code at runtime, and cannot use this approach. We cannot think of a solution for general callbacks in AOT schemas.

However, many callbacks include a user-data field. In C parlance, the callback signature has a parameter of type `void*`. When we register the callback, we pass a pointer to the user-data object along with the callback itself. When the library invokes the callback, it passes this object to the callback. We can use this field to sneak in the Lua state. Reiterating, this is not a general solution but covers many cases.

Garbage collection of closures must be taken into consideration. If the high-level functions are subject to collection, and the callback may be invoked after the registration call site, we may run into dangling pointer issues if we are not careful. Another consideration, more specific for Pallene, is that not all functions return; in case of errors Pallene may call `longjmp` and "hop over" the calling C function. This may cause all sorts of issues, depending on the library.

3
Related Work

On this chapter, we'll go through three relevant foreign function interface designs and implementations: LuaJIT's FFI extension, Swift C interoperability, and OCaml's foreign function interface. These three offer a diverse overview of common FFI patterns; they are quite different from each other, forming a reasonable summary of FFI approaches. We've also investigated the Java Native Access (FAST; WALL; CHEN, 2007) and Racket's foreign interface (BARZILAY; ORLOVSKY, 2004). However, these two don't offer much diversity to our list.

3.1
LuaJIT's FFI Extension

The FFI extension is tightly integrated into LuaJIT (PALL, 2005b). Although its design has been copied over to the Lua reference implementation by third-party libraries, the implementation is restricted to LuaJIT and not available as a separate module. It is provided as a library, which allows calling external C functions and using C data structures from pure LuaJIT code. It is important to highlight this second use: the FFI extension can be used without interacting with foreign code, providing low-level data structures which may be faster than Lua's.

Interactions with external libraries start with a description of their interface, using a special subset of the C language. This description is in the form of C declarations, which may be copied over from the library's header files; the FFI extension contains a tiny C99 parser to understand these definitions. However, the FFI can only interact with things that generate symbols. One must be careful about macros; instead of functions, many libraries' interface is through macros, which expand to an underlying function call. This isn't a problem when one includes the header files themselves, but certainly is when the interaction is mediated by a dynamic linking loader. As such, before calling foreign functions, we must manually expand macros procedure invocations to their final form in the host language. Likewise, inline functions are ignored by the FFI.

This approach has many advantages, particularly in using C itself to describe a C interface; there's no need to learn a new language, and the definitions may be mostly copied over from the library's header files (being careful about macros). However, there are two disadvantages to this approach.

First, there's nothing ensuring the bindings are correct; an error in the description will introduce serious (and sometimes subtle) bugs. Second, since the descriptions are pure C, there's some ambiguity to the declarations. A pointer, for example, could be many things (*e.g.* an array, a null terminated string, a handle, a pointer to a single value, an uninitialized pointer), and ownership of its content not well-defined. This forces the FFI to support generic use of pointers, which can lose information about the particularities of each use.

For simple types, the extension relies on translating Lua's preexisting values. For example, when crossing language boundaries, Lua numbers get translated to the C numeric type specified in the foreign interface description. For more complex types, such as arrays and records, the FFI relies on new types; besides the standard Lua objects, LuaJIT added the `cdata` type. They are garbage collected as normal.

In Code 7, we show an example of a foreign interface description for `zlib` and its use, adapted from LuaJIT's documentation. First, we declare the functions we wish to interact with and dynamically load the external library. Of interest, line 19 creates a buffer with type `uint8_t*` and size `n`, which will be used later. This buffer is a `cdata`, owned by LuaJIT. We could have created the buffer through a `malloc` call instead; the difference is that it would be no longer owned by LuaJIT, and would have to be manually managed. On line 20, we allocate a buffer of size one, using the same function as before, to emulate a standard pointer to a single integer. It is also a `cdata`, owned by LuaJIT.

Now we can call `compress2` on line 21. On line 23, we transform the result of the function call, stored on the buffer we allocated, into a Lua string. To do so, we pass its address and the size of the string, written on the `buflen` pointer by `zlib`. The expression `buflen[0]` is analogous to dereferencing a pointer.

Note that `cdata` storing buffers, created by `ffi.new` or otherwise, can be indexed normally (except for `void*` pointers and pointers to incomplete types), which will do the appropriate pointer arithmetic and dereferencing. This is what gives LuaJIT data-level interoperability with C; we can have values in Lua which store C data, and perform the same operations we could in C. This also allows the extension to be used for speeding up code; instead of using Lua tables, we can directly interact with the underlying memory. Since the implementation contains a JIT compiler, it can inline these memory accesses without any indirection overhead.

This comes with important safety considerations, however. Accessing this memory is unsafe in many ways. Since we didn't necessarily create these

Code 7: luajit_bindings.lua

```
local ffi = require "ffi"

ffi.cdef[[
unsigned long compressBound(unsigned long sourceLen);

int compress2(
  uint8_t *dest,
  unsigned long *destLen,
  const uint8_t *source,
  unsigned long sourceLen,
  int level
);
]]

local zlib = ffi.load "z"

local function compress(txt)
  local n = zlib.compressBound(#txt)
  local buf = ffi.new("uint8_t[?]", n)
  local buflen = ffi.new("unsigned long[1]", n)
  local res = zlib.compress2(buf, buflen, txt, #txt, 9)
  assert(res == 0)
  return ffi.string(buf, buflen[0])
end
```

pointers (they may be a return value from a C function), there's no way to know if they contain valid memory. There's also no way to do bound-checking; a bug may overrun the buffer without detection. We also have to consider the garbage collector. Although the memory created by the `ffi.new` function is traced, its content is opaque. As such, if we compose pointers (for example, an array of them), we must anchor them elsewhere, otherwise we may create dangling pointers.

In conclusion, LuaJIT's FFI extension is quite flexible (with the exception of macros). However, it brings unsafeness to Lua; the yellow caution tape has almost entirely been pulled inside the language. The boundaries between Lua and C are not guaranteed to be safe, since the description may be wrong, and most operations are dangerous in some way.

3.2 Swift C Interoperability

Swift, although a general purpose language, was developed with the goal of replacing Objective-C at Apple. Since Objective-C is a strict superset of C, interoperability with C libraries is built-in, a well appreciated and much used feature of the language. Consequently, Swift—being the Objective-C killer—was designed to be tightly integrated with Objective-C and, by transitivity, with C.

We will investigate how Swift deals with C interoperability. It offers a nice contrast to LuaJIT: Swift is a statically typed language compiled ahead-of-time, and the language itself was designed with C interoperability in mind. LuaJIT on the other hand is dynamically typed, interpreted and compiled just-in-time, and has to contend with Lua's design, which conceptualized C interoperability in an entirely different way.

The differences start with describing foreign interfaces. Swift uses the library's header files directly, instead of relying on declarations at the host language. Because of this, the compiler needs to understand C in some capacity. This is achieved through the use of Clang (LATTNER, 2007; LATTNER, 2008); it is shipped with Swift, which already uses the LLVM infrastructure (LATTNER, 2003). Directly using header files eliminates an entire class of bugs that may happen because of wrong foreign interface descriptions. After parsing the header files, the Swift compiler generates bindings for Swift, mapping them into native Swift functions.

The next step is understanding how C types are mapped into Swift. To this end, the standard library contains many types of C, provided as first-class types in Swift; the compiler maps each individual C type on a foreign function's signature to their corresponding type of C in Swift. For example, the standard library has the type `UnsafePointer<Pointee>`, where `Pointee` is a generic type parameter; constant pointers in signatures are mapped into this type. There are also the unsafe mutable pointer type for when the content is not constant, the opaque pointer type for incomplete pointer types, the unsafe raw pointer type for constant `void` pointers, and the unsafe mutable raw pointer for non-constant `void` pointers. These together cover all use cases for pointers, giving us a complete mapping of C pointers into Swift. We can create values of these types from Swift, or they can come directly from C.

Note that they don't necessarily point to a single element; they may point to a larger block of memory containing multiple elements, just like C. We can manipulate these unsafe pointers normally, doing pointer arithmetic and dereferencing them freely. There's an additional unsafe buffer pointer, which doesn't appear in the interface mappings, but is useful nonetheless. Most of the pointer types are dangerous, and have been appropriately marked as such in their name.

There are a few features that makes C interoperability more ergonomic and safer. One of them is Swift's implicit bridging. For example, in a function call, an unsafe pointer can be implicitly created from a standard array if their element types match; we can call a function that takes an `UnsafePointer<Int>` passing an array of integers. Since we never actually have the unsafe pointer

at hand, and its value is transient, safety is not compromised. Another feature is the `&` operator. For example, consider a C function that receives an `int*` pointer, which is mapped to an `UnsafeMutablePointer<Int>` Swift type; we use the `&` operator to create a pointer from a normal `Int` variable in scope, just as we would in C. Likewise, we avoid using unsafe parts of the language. Consequently, interoperating with C is not always entirely dangerous. Also, many of these pointer types are compatible between themselves, and are subjected to Swift's implicit bridging; we could call a function which receives an `UnsafePointer<Int>` passing an `UnsafeBufferPointer<Int>`.

In Code 8, we show the same use of `zlib`. There are many ways to implement this function; we'll do it in a way to show more features of the language. In comments, we show the mapped types of the C functions. The weird type names (*i.e.* `Bytef`, `uLong`, `uLongf`) are `typedefs` from `zlib`, handled automatically by the language. Another peculiarity are the "`!`" tokens on the pointer types. Swift has optionals, which are analogous to the *maybe monad* (WADLER, 1990; PETRICEK, 2018). These "`!`" indicate that the types are nillable.

In line 16 we create a buffer of the appropriate size, where the results of the compression will be stored. We store this buffer in a variable of type `UnsafeMutablePointer<Bytef>`. Since memory we allocate this way has to be manually managed, we must deallocate it after use. Note the `defer` on the following line; it will be called when the scope ends, freeing the memory. We could have used an array instead, avoiding the need to use unsafe parts of Swift. In line 19 we call the `compress2` function, making use of Swift's implicit bridging for arrays. We finally convert the result into an array, which copies the underlying memory.

Since Swift pointers can have types, there's a class of subtle programming bugs that may be introduced if we violate the strict-aliasing rules: just as in C, type punning may yield undefined behavior. Also, since we can do arithmetic and dereferencing freely, all sorts of issues arise from that. The silver lining is that one doesn't always need to use the unsafe bits; opaque pointers are safe, and so are the other common Swift types that are implicitly bridged.

In conclusion, Swift's C interoperability is extremely flexible. It even supports macro constants, although not macro procedures. It has dangerous parts, but we can often avoid them; parts of the yellow caution tape are inside Swift, but not entirely. The tape is made evident in Swift: we know exactly where the unsafe parts are.

Code 8: swift_bindings.swift

```
/*
func compressBound(sourceLen: uLong) -> uLong

func compress2(
  dest: UnsafeMutablePointer<Bytef>!,
  destLen: UnsafeMutablePointer<uLongf>!,
  source: UnsafePointer<Bytef>!,
  sourceLen: uLong,
  level: Int32
) -> Int
*/

func compress(txt: [Bytef]) -> [Bytef] {
  let n = compressBound(UInt(txt.count))
  var buflen: uLongf = 0
  let buf = UnsafeMutablePointer<Bytef>.allocate(capacity: Int(n))
  defer { buf.deallocate() }

  let ret = compress2(buf, &buflen, txt, UInt(txt.count), 9)
  assert(ret == Z_OK)

  return Array(
    UnsafeBufferPointer(start: buf, count: Int(buflen))
  )
}
```

3.3 OCaml's Foreign Function Interface

The last language we'll analyse is OCaml. It has a built-in mechanism to interface with C, which is somewhat similar to how the Java Native Interface works (LIANG, 1999). However, this is not the kind of mechanism we're interested in, since the hop from OCaml to C is quite restricted; the C functions we are allowed to call can only receive OCaml values, and C is responsible for converting them to useful C values. In this sense, it is closer to something like the Lua C API than what we've been calling an FFI so far. Consequently, this mechanism requires the programmer to write C glue code to interact with arbitrary foreign code, as well as a C compiler.

What we are interested in is a mechanism to call arbitrary foreign functions; we want to "stay in the fun world", and not have to drop down to C. OCaml has a foreign function interface library of the kind we want, called `Ctypes`. There's nothing special about it, it's a library like any other. The `Ctypes` library is a modular foreign function interface for OCaml, which achieves declarative foreign function bindings through generic programming, and is built on top of `libffi` (YALLOP; SHEETS; MADHAVAPEDDY, 2014; YALLOP; SHEETS; MADHAVAPEDDY, 2016; YALLOP; SHEETS; MADHAVAPEDDY, 2018). It is, in many ways, analogous to the Java Native

Access.

Comparing OCaml with the others, it offers a third approach to describe foreign interfaces: we use OCaml itself to this end, writing host language code to describe them. This is in contrast with LuaJIT, which parses a string with C declarations at runtime—and Swift, which uses a C compiler to parse header files and generete bindings at compile-time. OCaml's approach has the downside of not guaranteeing the interface description is correct (just like LuaJIT), which introduces an entire class of bugs to the FFI. It also can only interact with things that generate symbols, not supporting macros.

The library offers a pointer type, which is one of the ways OCaml provides data-level interoperability with C. Pointers can be manipulated freely, allowing arithmetic and dereferencing. We can allocate them in OCaml, specifying the size we want, as well as receiving them from C. Memory allocated in OCaml is subjected to the garbage collector; in fact, holding any pointer derived from the original allocated pointer will anchor the entire block of memory. Pointers received from C, however, are not managed.

The library also offers the `CArray` type, which is another way OCaml provides data-level interoperability with C. This type has the same underlying representation as arrays in C, offering a mostly safe data structure with a nice interface in OCaml, which we can transform into a pointer and pass to C. One can instantiate these `CArrays` on the host language, in which case they are safe; since we know the underlying memory is valid and its size, we can read and write to it without fear. One can also create `CArrays` from a pointer, explicitly specifying its size, in which case they are not necessarily safe; since there's no way to know if the memory is valid or its size, `CArrays` are potentially dangerous.

In Code 9, we show the same use of `zlib`. We start with the interface description. The function named `foreign` is a higher-order function, which takes the name of the foreign function we wish to call and its type, and returns a function of the correct type. This function, once called, invokes foreign code. The type is built using the `@->` infix constructor and the `returning` constructor. Both `txt` in line 18 and `buf` in line 21 are `CArrays`. In line 22, we create a new pointer to a single value, and in line 26 we dereference this pointer through the `!@` operator.

In conclusion, the OCaml foreign function interface is quite flexible, with the exception of macros. It has dangerous parts, but we can avoid them; parts of the yellow caution tape are inside OCaml, but not entirely. The `zlib` example is safe; the only operation that can go wrong is the pointer dereferencing, but since the pointer was allocated and initialized in OCaml, and no additional

Code 9: ocaml_bindings.swift

```
open Ctypes
open Foreign

(* foreign description *)
let compress_bound =
  foreign "compressBound" (ulong @-> returning int)

let compress2 = foreign "compress2" (
    ptr uint8_t @->
    ptr ulong @->
    ptr uint8_t @->
    ulong @->
    int @->
    returning int
  )

(* compress wrapper *)
let compress txt =
  let len = Unsigned.ULong.of_int (CArray.length txt) in
  let n = compress_bound len in
  let buf = CArray.make uint8_t n in
  let buflen = allocate ulong (Unsigned.ULong.of_int n) in
  let res =
    compress2 (CArray.start buf) buflen (CArray.start txt) len 9 in
  assert (res == 0) ;
  CArray.sub buf ~pos:0 ~length:(Unsigned.ULong.to_int !@buflen)
```

arithmetic was performed on it, it is not dangerous. The dangerous parts of it are not explicitly named, one must be familiar with the FFI to know where safety may be compromised. Also, the boundaries between OCaml and C are not guaranteed to be safe, since the description may be wrong.

4
Design

Reiterating, our goal with this design is to create a well-rounded FFI, with a good balance between our five criteria: flexibility, safety, portability, performance, and implementation complexity. This chapter will focus more on the first two, as the last three are closer to the implementation. Nevertheless, the design influences the implementation, so some decisions we make here affect the latter criteria.

We start this chapter with an overview, in which we discuss global design decisions, introduce some concepts we'll need throughout, and state some overarching principles the design follows. Then, we'll present the Pallene FFI through real cases, starting with simpler cases and progressing to more complex ones.

4.1
Design approach

During the design process, we came up with two contrasting approaches to the Pallene FFI, which we've named the *ad-hoc* approach and *completeness* approach:

- The *ad-hoc* approach is empirical. It relies on observing C libraries and extracting from this observation knowledge about which features we should add to the design. A corollary of this approach is that if we cannot see a real use of a construction, any support the FFI may offer to it is incidental.
- The *completeness* approach is rational. It relies on analysing the C language itself, regardless of the way it is used, and building an FFI that strives to support everything that may appear on an API.

Remember that in some cases there's a trade-off between flexibility and safety. The decision between the ad-hoc and completeness approaches heavily impacts these two criteria, as well as implementation complexity (completeness is harder).

The completeness approach is, *ipso facto*, more flexible. However, as C is an unsafe language, we must drag unsafeness into Pallene if we go for completeness. That is, as we must be able to freely manipulate C types from Pallene in all conceivable ways, the things that a programmer can do in C that may go wrong must also be available in Pallene. For example, completeness

requires a mechanism for type punning, which enables breaking C strict-aliasing rules, bringing undefined behavior into Pallene. Another example is writing arbitrary memory, which is inherently unsafe but must be allowed in Pallene if we want completeness.

The ad-hoc approach is less flexible. Since we only add features if we see a real use case of it, one may always manufacture a hypothetical case in which the FFI would be unable to support. However, it allows for a more informed compromise between safety and flexibility: we only weaken safety in favour of flexibility when needed.

We've chosen the ad-hoc approach, as we believe it gives us a better balance between safety and flexibility. Unless we see a concrete example that forces us to compromise on safety, we prefer to sacrifice hypothetical flexibility. It is also much simpler to implement: it has a more modest scope, and Pallene is not very amenable to adding lots of C features, particularly regarding pointers.

We've compiled a list of five C libraries to lead our empirical approach, using them as a canon of sorts. We believe they are a good representation of C interfaces since they require a wide variety of features. Their interface will guide our decision-making around what features a flexible FFI must support, and when to sacrifice safety in favour of flexibility. The libraries we chose are:

- `libc`, the standard C library;
- `liblua`, the Lua C API;
- `zlib`, a data compression library;
- `sqlite3`, the C interface of SQLite;
- `libgit2`, a C implementation of Git provided as a library;
- `libcurl`, a client-side data transfer library.

Additionally, a design goal we have with the design is to try to keep changes to the Pallene language minimal. This means avoid adding new types to Pallene unless we absolutely need to. This need may arise both from the flexibility criteria (we wouldn't be able to use certain libraries otherwise) and from a performance criteria (to enable data-level interoperability when marshalling and unmarshalling is too expensive).

Another consideration is which schema we will use for the implementation. This decision has many ramifications on portability, performance, and implementation complexity. In any case, it is not merely an implementation detail: it has impacts on the design as well. There are things that we cannot do depending on this choice, such as generic callbacks in ahead-of-time or preprocessor procedure macros in just-in-time and interpreted schemas. We've chosen

to implement our Pallene FFI in an ahead-of-time schema. We'll discuss this choice in more details in Chapter 5.

4.2 Our FFI by example

Since we are guiding our design through an empirical approach, we'll explain it through examples. These examples offer both a description of our design, as well as the concrete cases which guided our decision-making; we only add features if we observe them in the five canonical libraries, and we only sacrifice safety if we see a use case which requires it.

The first step in interacting with external libraries is describing their interface. Previously, we discussed a few ways this can be done. Of relevance, we can use the library's header files (which are correct, but may lack important information), and we can describe them at the host language (which are richer in information, but can be incorrect). On the latter, safety may be compromised if the programmer provides incorrect bindings declarations. For example, they may specify symbols that do not exist, or specify symbols with the wrong type. On the former, we may lose important informations about the foreign function, as well as increasing the implementation complexity, since it would require an entire C compiler front-end.

Our FFI uses a hybrid approach, requiring both header files and host language declarations. This has the downside of being somewhat redundant. However, we can both check for correctness of the bindings and offer richer descriptions. We believe this trade-off is worth it. Additionally, offering support for macros can only be accomplished through the use of header files. In Chapter 5, we'll get into more details on how we understand C without increasing the implementation complexity.

Since we must describe foreign interfaces in the host language, Pallene programs that uses the FFI have to start with this description. Also, one of our design goals is to restrict changes to the core Pallene language as much as possible. With these two requirements in mind, we've created a special description language inside Pallene, where the programmer describes the foreign interface separate from the core language. This description language is wrapped inside a *foreign block*, which is used to separate this language from the rest of Pallene. Bindings declared inside this block will be available inside Pallene as native variables, types, and functions.

There are five types of statements allowed inside this block: `include`, `constant`, `type`, `function`, and `callback`. We'll go through them in the following sections.

Includes

Include statements exist because of our hybrid approach to foreign interface descriptions. Since we must provide header files, we need some way to tell the FFI what header files we'll be using. Then, all further declarations inside the foreign block will be checked against the declarations in C. The goal is to make sure an incorrect declaration in Pallene will cause a compile-time error, thus eliminating an entire class of bugs in the FFI.

Because of this verification, includes tend to be the first statements in foreign bindings. They are used to specify the header files which contain the external definitions we wish to bind against, used by the FFI to check the correctness of bindings. As such, foreign block without an include will not be particularly useful: any further binding will not have a matching C declaration, which are enforced by the compiler, raising an error.

Following how the C preprocessor works, includes can be either enclosed by angle brackets or quotation marks, with the same semantics. The Pallane compiler, when invoked, accepts an include path in the same way the C compiler does.

Constants

Constants are commonly found in APIs (*e.g.* return codes), but uncommonly supported by FFIs. Although it's quite possible to use an FFI that does not support constants, by manually copying them over to the host language, it's rather laborious and error prone.

In this context, constants means compile-time constants in C, which generally take the shape of macro defines, constant variables, and `enum` cases. One of the challenges related to supporting constants is that they may not even have a type (in the case of macro definitions), or may have implementation-specific size (in the case of enumerations).

Seeing that constants don't necessarily generate code, there isn't always a symbol to bind against. In these cases, we'd need to parse their C definitions to extract their values. This is one place our hybrid approach pays off. Since we do have the C definitions of constants at hand—contained in the library's header files—and a C compiler, we have the ability to support compile-time constants. In Chapter 5, we'll go into more details on how we implemented this feature.

As discussed, constants don't necessarily have a type in C. However, since we want to expose them to Pallene, their Pallene type must be specified in some

way. We've delegated this task to the programmer, making external constant declarations include their Pallene type. If the supplied type is incompatible with the underlying value, the FFI will raise an error at compile-time.

To declare a constant in Pallene, we need its external C name, its internal Pallene name, and its Pallene type. The example we chose is contained in `libc`. In line 3 of Code 10, we show the C constant `M_PI` exposed to Pallene with name `C_M_PI`, with Pallene type `float`. In the cases where we want the constant with the same internal and external name, line 4 shows a shorthand way to write constants' definition.

Code 10: constants.pln

```
1 foreign
2   include <math.h>
3   constant C_M_PI: float = M_PI
4   constant M_PI_2: float
5 end
```

Numeric functions

The first class of function we'll tackle are functions over numeric types. The goal is to map these functions into Pallene functions, to be called like any other. To this end, we first need to tell the FFI what is the function's C signature: its name and its type. Then, we need to specify in some way the name this function will take inside Pallene, and its Pallene type.

Considering the function's Pallene type, we've reached a dilemma. The parameters and return value types of the foreign function are types of C, which do not exist in Pallene. The Swift language approach consists of adding all these types to the language. We've decided to minimize modifications to the core language, and went for a different approach: using Pallene's preexisting numeric types to represent C types, translating when needed. This also eases integration with Lua.

To declare external functions in the foreign block we use the `function` statement, which specifies explicitly its external name, external type and its internal name, and implicitly its internal type (integer types are mapped to Pallene's `integer`, and floating-point types are mapped to Pallene's `float`).

We also chose an example from `libc`. In Code 11, we start by including the C mathematical library. Removing this include will yield a compilation error, because the FFI will not find a matching C declaration. Consider line 5 in Code 11, wherein we declare a external function named `sin`, from `libc`'s

mathematical library. Its foreign type is `(cdouble) -> cdouble`, explicitly specified in the statement. Its internal type is derived from the foreign type. In this case, the FFI creates a Pallene function with type `(float) -> float`. Finally, we've chosen an internal name for this function: `external_sin`.

Outside the foreign block, this function becomes available like a normal Pallene function. When called, the argument values are marshalled into their corresponding C types, and the return value is unmarshaled into its corresponding Pallene type. In the case of `sin`, the conversion isn't particularly interesting since the two types have the same underlying representation.

Line 8 in Code 11 declares the external function `sinf`, using a shorthand syntax for using the same internal and external names, much like we did with constants. Although the external function `sinf` has a different type than `sin`, the function exposed to Pallene has the same type as the `sin` function. When we invoke `sinf`, however, the marshalling and unmarshalling applied by the FFI actually does something: it converts a double to float and back. Line 11 in Code 11 is a function on integers, having the Pallene type `(integer) -> integer`. The FFI converts between the C types and Pallene types when we invoke this function.

One restriction of C functions is that they cannot return multiple values. Fortunately, there's a workaround for restriction, through the use of references. This is a commonly found pattern in C functions: the caller passes a memory address to the function, where the result will be written to. This is a well behaved use of pointers, as the memory it points to belongs to the caller, which can be supported by our FFI without compromising on safety (general use of pointers require compromising on safety).

At line 14 in Code 11, we show an example of a function (`frexp`) with this pattern. It's a C function that takes a double and a pointer to an integer. The address contained in the pointer will be written to by `frexp`. These functions are, in their essence, functions which return multiple values. As such, our design maps this intent into Pallene: the FFI uses Pallene's multiple return feature to implement this pattern. We make use of parameter modifiers to indicate this intent. In this case, we use the `retout` modifier, which alters the meaning of parameter: it isn't used to pass data to the callee, and is instead used to return data to the caller.

The `retout` modifier excludes its respective parameter from the internal Pallene function type, and increases the number of return values by one. As such, the Pallene type of `frexp` is `(float) -> (float, integer)`, and whatever the caller writes to the memory stored in the pointer argument is translated by the FFI as an extra return value. There's also the `retinout`

Code 11: numeric.pln

```
1 foreign
2   include <math.h>
3
4   -- double sin(double arg);
5   function external_sin = sin(cdouble): cdouble
6
7   -- float sinf(float arg);
8   function sinf(cfloat): cfloat
9
10  -- long labs(long n);
11  function labs(clong): clong
12
13  -- double frexp(double arg, int* exp);
14  function frexp(cdouble, retout cint): cdouble
15 end
```

modifier, for when the parameter is also used to pass values to the callee (in this case, the parameter is not excluded from the function's type). And finally, just the `in` modifier, which does not increase the number of returned values and is used to only pass values to the callee.

Remember that the FFI enforces correct bindings. As such, incompatible binding declarations won't silently introduce bugs: the FFI will try to match the foreign block declarations with the header file declarations, and failing to do so will raise a compile-time error.

String parameters

Strings in C are a contiguous block of memory terminated by a null character, whose contents cannot be altered. As such, they're generally stored in a value of type `const char*`. They are commonly found in APIs, and must be supported in some way by the flexible FFI.

Here we've decided to use a similar approach to functions over numeric types: using Pallene existing `string` type to represent C strings. There is, however, an important safety consideration: general use of C strings cannot be safely handled by the FFI. Pointers returned by C may contain an invalid address, the string may be malformed, and we cannot trace it using our garbage collector. As such, in this section, we're considering only strings as parameters. This restriction makes using strings safe: since string parameters are owned by the Pallene runtime, we can use them without fear. We'll show how the FFI handles general strings later.

Code 12, also from `libc`, shows how we declare a function with string parameters. We call this function passing a Pallene string, which the FFI

translates to a C string without copying. The C function may hold a reference to this string past the call site, but the Pallene programmer must be mindful of its lifetime: since the Pallene string may be collected, the programmer must anchor it in some way if the called function stores the string past the call site.

Code 12: string_parameters.pln

```
1 foreign
2   include <stdio.h>
3
4   -- int puts(const char *str);
5   function puts(string): cint
6 end
```

Opaque pointers

Pointers are vital for programming in C, and their use in APIs is pervasive. There are, however, important safety considerations with pointers. Allowing free use of them, such as dereferencing and pointer arithmetic, is a recipe for bugs. We have no guarantees on pointers that came from C: they may point to invalid memory or a wrong type (dereferencing them will introduce serious bugs), and are not bound checked (accessing memory out of bounds will cause a buffer overrun).

However, a crucial use of pointers in APIs are handles. In C, they are generally represented by a pointer to some incomplete type. Handles are pointers to some underlying value that libraries expect in future function calls. As such, the content of handles are not to be used by application code; they are truly opaque.

With this in mind, we've created a new Pallene type: opaque pointers, which are a restriction on general pointers. The fundamental insight behind opaque pointers is that—although accessing arbitrary pointers is an unsafe operation—as long as they remain opaque we can preserve safety. Opaque pointers are perhaps the most versatile tool in the FFI. We'll first show its basic use, and later on show how we've extended them for more general use.

The first step in using pointers is declaring their type. Here, our example is from `liblua`, shown at line 5 in Code 13. This example creates a new first-class type inside Pallene with name `LuaState`, whose underlying value is a pointer to an incomplete `struct` type called `lua_State`. This type has value semantics, just like normal C pointers: assigning it will copy the underlying address value. Line 6 in Code 13 shows how we can get null pointer of that type, by reusing the same syntax we've shown before for constants. Now, we

can declare foreign functions which take and return values of this type, as shown at lines 9 and 12 in Code 13.

Code 13: opaque_pointer.pln

```
1 foreign
2   include "lua.h"
3   include "lauxlib.h"
4
5   type LuaState = pointer struct lua_State
6   constant NullLuaState: LuaState = NULL
7
8   -- lua_State *luaL_newstate(void);
9   function luaL_newstate(): LuaState
10
11  -- void lua_close(lua_State *L);
12  function lua_close(LuaState)
13 end
```

Opaque pointer parameters can be altered using the same modifiers as numeric parameters, since they also have value semantics. A concrete use of this is seen in `sqlite3`, shown in Code 14: the handle created by SQLite3 is returned by writing to a pointer. By modifying it with an `retout`, the internal Pallene function will return the pointer as an extra return value.

Code 14: opaque_pointer_inout.pln

```
1 foreign
2   include <sqlite3.h>
3   type SQLiteHandle = pointer struct sqlite3
4
5   -- int sqlite3_open(
6   --   const char *filename,
7   --   sqlite3 **ppDb
8   -- );
9   function sqlite3_open(string, retout SQLiteHandle): cint
10 end
```

This feature also allows for declaring function pointers. A curious use of constants is taking a pointer to a declared external function. We could, for example, declare an opaque pointer type called `MallocPointer` with the same type as the function `malloc` in `libc`, and also create a constant of this type, assigning `malloc` itself to it. We can then pass this constant to other function through the FFI. A use of this kind of feature can be seen on `libcurl`, where we can configure the memory allocation functions of the library.

An important use of opaque pointers is when dealing with file streams, as seen in the `IO` parts of `libc`. A `FILE*` type in C can be represented as an opaque pointer normally. This way, when we open a file in Pallene we get

a handle, and whenever we interact with this file we pass this handle to C functions.

On Chapter 5, we'll go into more details on how we implemented opaque pointers in Pallene: they have to be well integrated with Lua (since Pallene is a companion language), and Pallene must be able to tag-check them in some way. We also want them to have value semantics in Pallene.

Returned strings

We've shown how the FFI deals with string parameters. Since they're owned by the Pallene runtime, there's no safety issues with their use. Returned strings, on the other hand, have none of these guarantees. The returned pointer may not point to valid memory (in particular, it can be a null pointer), its content may be invalid (missing the terminator character, or not being a string at all), and its ownership is ambiguous (some APIs are responsible for freeing the memory, others transfer this responsibility to the caller).

In this section, we'll see the first unsafe feature of our FFI. A flexible FFI cannot be fully safe: there are dangerous parts of C that must be exposed to Pallene if we want to interact with certain libraries. As discussed, we are approaching this dilemma through an empirical approach, wherein we only sacrifice safety if we see a concrete example that requires it. For returned strings, we've selected the `setlocale` function from `libc`.

Another way we've addressed the flexibility/safety dilemma is to explicitly mark unsafe features as such, indicating exactly where the yellow caution tape is. If we don't use unsafe features, the tape is strictly outside Pallene. Otherwise, the parts that do use unsafe features may introduce errors which cannot be explained in Pallene terms, such as segmentation faults.

With this in mind, we've added an operation on opaque `char` pointers called `unsafe_to_string`, accessed through a method syntax as shown at line 14 in Code 15. This will treat the content of the opaque pointer as a null terminated string, converting it to a Pallene string by copying the pointer's content. The result is a normal Pallene string, that will be traced by the garbage collector normally.

The Pallene programmer is in charge of guaranteeing the memory is valid, that the string is well formed, and for the lifetime of that memory. If the external API has transferred the ownership to the caller, it is advised the programmer call the deallocation function on that pointer immediately after the conversion. A careless programmer may write code that causes segmentation faults, reading unowned memory, and leaking memory.

Code 15: unsafe_string.pln

```
1 foreign
2   include <locale.h>
3
4   type CharPointer = pointer char
5   constant NullCharPointer: CharPointer = NULL
6   constant LC_ALL: integer
7
8   -- char *setlocale(int category, const char *locale)
9   function setlocale(cint, string): CharPointer
10 end
11
12 local function getlocale(): string
13   local str_pointer = ffi.setlocale(LC_ALL, NullCharPointer)
14   return str_pointer:unsafe_to_string()
15 end
```

Structs

To represent C structures in Pallene we have a choice similar to numeric types: we can use Pallene's existing records to represent `structs`, marshalling and unmarshalling when needed, or we can create a new type in Pallene with the same underlying representation as C `structs`. We've decided to use the first approach, similar to how we dealt with numeric types. Note that these two options are not mutually exclusive: we may add both of them, and have the programmer use the one more appropriate for the task.

With this in mind, we've represented C `structs` in Pallene using normal Lua tables, interpreted as records. These tables in Pallene are called *Lua records*. There's one crucial difference between the previously discussed numeric types and tables: the latter has reference semantics, which changes how we deal with pointers and modifiers.

When declaring foreign C `structs`, we need to know their fields' names and their respective types. Then, we need to know the internal Lua record type. To this end, we employed the same approach used to specify the internal type of foreign functions: since the C types are explicitly described by the programmer, we can implicitly derive the Pallene type by translating them. Additionally, because we want the FFI do check if the host language definitions are correct, we also need to provide the name of the C `struct` defined in the library's header files. Finally, we also need to give it an internal Pallene name.

When declaring a function that takes a C `struct`, we must indicate to the FFI if we are passing a pointer to the `struct` (in which case we also need to indicate if the FFI should marshal and/or unmarshal it), or the `struct` itself. We decided to indicate this intent through modifiers, somewhat similar

to the ones used on numeric types and opaque pointers, but not exactly the same because Lua records have reference semantics.

At lines 14 through 34 in Code 16, we show a foreign `struct` declaration from `zlib`. We start by its Pallene name (in this case `ZStreamS`), which becomes available at Pallene as a new Lua record type with the appropriate fields. Then, we put its C name (in this case `z_stream_s`). This is necessary for our FFI to ensure the definitions match. If they don't, it will generate a compile-time error. Finally, we declare its fields, with both their name and types. The ordering of the fields do not matter. However, the names must match their C definitions, and their respective types must be compatible with their C definitions. The type of the generated Lua record is implicitly derived from the explicit `struct` type declaration, with the Pallene type of each field being converted from its C counterpart.

There are four ways of passing a `struct` to a C function. These are indicated through modifiers in the foreign function declaration. Regardless of the modifier, in Pallene we must call this function passing a Lua record.

- We can pass the `struct`'s address with useful data, but not care about what C writes to it. As such, the FFI will marshal the Lua record into a transient C `struct`, passing its address, but will not unmarshal the `struct` after the function returns. This corresponds to the `in` modifier.
- We can pass the `struct`'s address with no useful data, but care about what C writes to it. As such, the FFI will not marshal the Lua record, instead passing the address of a uninitialized `struct` to the foreign function, and will unmarshal the resulting C `struct` back after the function returns. This corresponds to the `out` modifier.
- We can pass the `struct`'s address with useful data, and care about what C writes to it. As such, the FFI will both marshal the Lua record into the C `struct`, passing its address, and will unmarshal the resulting C `struct` back after the function returns. This corresponds to the `inout` modifier.
- Finally, we can pass the `struct` itself instead of its address. As such, the FFI will just marshal the Lua record into the C `struct`, passing its value rather than its address. This corresponds to no modifier.

In all cases, the FFI creates a transient `struct`. The differences are whether the FFI copies the passed Lua record to this `struct` first, if it passes the address or the `struct` itself, and if it copies the `struct` back. In no cases the FFI returns extra values, as it does in types with value semantics, such as

numeric types and opaque pointer types. Since the `struct` is short lived, C may not hold a reference to this `struct` past the call site. This is a downside of the marshalling/unmarshalling approach.

Code 16: structs.pln

```
1 foreign
2   include <zlib.h>
3   type UCharPointer = pointer uchar
4
5   type ZStreamS = struct z_stream_s
6     next_in: UCharPointer,
7     avail_in: cuint,
8     total_in: culong,
9
10    --[[ ... ]]
11  end
12
13  -- int deflateInit(z_streamp strm, int level);
14  function deflateInit(inout ZStreamS, cint);
15
16  -- int deflate(z_streamp strm, int flush);
17  function deflate(inout ZStreamS, cint);
18 end
```

Dereferencing pointers

We've discussed how to deal with pointers that are opaque, a restriction which makes their use safe. However, sometimes APIs return pointers which we need to access, such as `libgit2`. This library returns a pointer to the structure `git_config_entry`, which is owned by the library. This is in contrast with what we've discussed before, where the memory was owned by Pallene.

Since Pallene does not own these pointers, we cannot safely dereference them. They have all the same issues related to returned strings: the returned pointer may not point to valid memory (in particular, it can be a null pointer), its content may be invalid, and its ownership is ambiguous (some APIs are responsible for freeing the memory, others transfer this responsibility to the caller).

We need some way to access the content of pointers if we wish to interact with libraries like `libgit2`. As such, we've added another unsafe feature for dereferencing pointers, using a method call syntax on opaque pointers. This feature breaks the opaqueness of pointers, and cannot be used on void pointers or pointers to incomplete types.

At line 12 in Code 17, we have a Pallene function that takes an opaque pointer to a complete `struct` type. At line 13, we call the method

`unsafe_deref` on this opaque pointer, which accesses the memory stored in the pointer and copies them to a new Lua record. Note that we call the appropriate free function on the original pointer right after copying its content.

Code 17: deref_structs.pln

```
foreign
  include <git2.h>
  type ConfigEntry = struct git_config_entry
    -- [[ ... ]]
  end
  type ConfigEntryPtr = pointer ConfigEntry

  -- void git_config_entry_free(git_config_entry *);
  function git_config_entry_free(ConfigEntryPtr)
end

function deref_example(entry_ptr: ConfigEntryPtr): ConfigEntry
  local entry: ConfigEntry = entry_ptr:unsafe_deref()
  ffi.git_config_entry_free(entry_ptr)
  return entry
end
```

Arrays parameters

Opaque array parameters can already be represented with opaque pointers. We can call `malloc` normally, and pass the pointer around. We can even use `libc`'s buffer manipulation functions to interact with it. However, this can be unwieldy at times, as well as unsafe since it lacks bound checking and requires manual memory management. Also, there are use cases where we need transparent arrays: elements need to be accessed and manipulated, and as such cannot use an array that is opaque. The example we've selected is from `zlib`.

Note that arrays returned from C are unsafe to use in Pallene. Unless we treat them as fully opaque, accessing them yields the same issues as general use of pointers. As such, in this section we are restricting the use of arrays as parameters. Since Pallene owns these arrays, we can be sure the underlying memory is valid, we know its size and can do bound checking, and we can trace it normally with our garbage collector.

The first way to support array parameters is the same way we deal with `structs`: we use Pallene's preexisting objects to represent them. Pallene arrays are Lua tables, where the keys are contiguous integers starting at one. We'll use them to represent arrays, converting when needed. The syntax for declaring a foreign function which takes an array is the same as Pallene's: we wrap the parameter type in curly braces like so `{ctype}`. The FFI will verify if the

Pallene declaration matches the external declaration normally. For example, if we declare a function that takes an array of `longs`, the FFI will check if the C declaration receives a pointer of type `long*`.

When we call a function with an array parameter, passing a Pallene array as argument, the FFI will first acquire a block of memory with the appropriate size. If the parameter has an `in` modifier, the FFI will copy the contents of the Lua table to it, applying the appropriate translation function to each element before copying. After the function returns, if we've marked it with an `out` modifier, the FFI will copy the elements back to the Lua table. The caller may not hold the array past the call site, since it may be collected by the garbage collector at some point.

This works well for small, bounded arrays. However, the marshalling and unmarshalling approach does not work well for larger volumes of data. As such, we need a mechanism for data-level interoperability. To this end, we've added a new type to Pallene: CArrays.

CArrays are not Lua tables, instead having its underlying structure match C in a way that we don't need to convert anything. As such, if we want to pass CArrays to C, its elements must already be C values. With this in mind, the FFI restricts all uses of CArrays to elements with C types. The reasoning for this is two-fold. First, the common use case for CArrays is for interfacing with C, and allowing elements be arbitrary Pallene objects is not going to be useful most of the time. Second, storing Pallene objects introduces issues with tracing the lifetime of this memory.

Since the internal structure of the CArray is opaque to the Lua runtime, the data we store in them cannot be traced by the garbage collector. If we allowed elements of arbitrary types, we introduce an entire category of dangling pointer bugs, since the garbage collector may inadvertently collect an object still reachable through the CArray. There's a possible workaround, wherein we store these Lua objects in a way that they can be traced. However, we do not see the use of such a feature: it doubles the amount of memory used, and increases the implementation complexity, without substantial gain. Arrays that store Lua objects have no use in the FFI, and Pallene already has its own array that is capable of storing objects.

Note that the use of CArrays is not restricted to the FFI: it can be used as a faster array (on simple types only) for Pallene. They can be particularly useful for dealing with buffers. And since they aren't a transient object like marshalled Lua tables, CArrays can be held by external code past a foreign call site.

In Code 18, we show three possible ways to declare a function that

takes an array. The first is using an opaque pointer, which can be instantiated through a `malloc` call. The second is using normal Pallene arrays, which will be marshalled if marked with an `in` modifier, and unmarshaled if marked with an `out` modifier (and both if marked with `inout`). The third is using CArrays.

Code 18: array_param.pln

```
foreign
  include <zlib.h>

  -- int compress2(
  --   char *dest,
  --   unsigned long *destLen,
  --   const uint8_t *source,
  --   unsigned long sourceLen,
  --   int level
  -- );

  function compress_arr = compress2(
    out {cuchar},
    retinout culong,
    in {cuchar},
    culong,
    cint
  ): cint

  function compress_carr = compress2(
    C{cuchar},
    retinout culong,
    C{cuchar},
    culong,
    cint
  ): cint
end
```

Callbacks

Callbacks are quite tricky to implement in Pallene. The issue is that Pallene functions need a handle to the Lua runtime. In C, a callback is a simple function pointer, and the Lua state is not readily available. There's no standard way to pass the Lua state to the callback. Consequently, there's no simple way to implement generic callbacks in Pallene; we'd need something like a JIT compiler for this.

However, there's a very useful subset of callbacks we may support: those that include a user-data field. In C parlance, the signature of callbacks of this kind has a parameter of type `void*`. When we register the callback, we pass a pointer to the user-data object along with the callback itself. When the library invokes the callback, it passes this object to the callback. We can use this field to sneak in the Lua state. Reiterating, this is not a general solution.

For example, `libc`'s quick sort implementation has a callback parameter with no user data field, and consequently cannot be used through our FFI: the programmer would need to write C code by hand to address this. However, this solution covers many cases, such as the one seen in `sqlite3`.

The FFI needs to provide some mechanism to pass the Lua state to callbacks, as well as map high-level Pallene functions into C function pointers. We'll go into more implementation details in Chapter 5. Nevertheless, there are details of the implementation that affects the design. Of relevance, since the compiler will have to generate extra code for callback functions, we need to tell the FFI which functions are callback-able, otherwise we'd be adding some overhead to all of Pallene even if the code never actually uses the FFI.

With this in mind, we have to declare a callback type in the foreign block. This process is shown in lines 8 and 9 of Code 19. First, we must specify the name of the callback type. In this example, we've called it `sqlite_callback`. Then, we specify which Pallene functions implement this callback. In this example, there's a single Pallene function called `pln_callback`, implemented later on outside the foreign block, shown in Code 20. Finally, we must specify the type of the C function pointer.

The user-data object has its own type, intrinsic to the FFI: an opaque pointer of type `void*`, called `CallbackObject`. The FFI has a built-in function, `ffi.pln_callback_object`, which returns a value of this type when called. Note that the callback in line 9 of Code 19 has a parameter of type `CallbackObject`; we indicate which parameter of the callback is the user-data object through this intrinsic type.

The type of `pln_callback` matches the type in the callback declaration, minus the `CallbackObject` parameter. The compiler will generate extra code for this function, which will be discussed in Chapter 5. The only special thing about this function is that it may only contain arguments which may appear at the return position of foreign functions. Having done all this, we may pass this function to a foreign function that takes a callback normally.

After the callback is declared in lines 8 and 9 of Code 19, it becomes a valid type inside the foreign block; other external functions may specify it as the type of one of its parameters. In Code 21, we show a function declaration that takes a callback of this type.

Now we are ready to use this callback. In Code 22, we show how this callback may be registered. Note that we pass a Pallene function to it, as well as the callback object.

One issue with this approach is that the Pallene callback function is limited in the things it can access; it is rather sandboxed in its execution,

Code 19: callback.pln

```
foreign
  include <sqlite3.h>
  type SQLiteHandle = pointer struct sqlite3
  type CharPointer = pointer cchar
  type StringArray = pointer CharPointer

  -- int (*callback)(void*,int,char**,char**)
  callback sqlite_callback{pln_callback} =
    (CallbackObject, cint, StringArray, StringArray) -> cint
end
```

Code 20: callback_fn.pln

```
local function pln_callback(
  argc: integer,
  argv: StringArray,
  azColName: StringArray
): integer
  -- ...
end
```

Code 21: callback_foreign.pln

```
foreign
  -- ...

  -- int sqlite3_exec(
  --   sqlite3*,
  --   const char *sql,
  --   int (*callback)(void*,int,char**,char**),
  --   void *,
  --   char **errmsg
  -- );
  function sqlite3_exec(
    SQLiteHandle,
    string,
    sqlite_callback,
    CallbackObject,
    retout CharPointer
  ): cint
end
```

Code 22: callback_reg.pln

```
local function exec(handle: SQLiteHandle sql: string)
  local ret, msg = ffi.sqlite3_exec(
    handle, sql, pln_callback, ffi.callback_object()
  )
  -- ...
end
```

except for accessing global variables. We cannot pass extra values to it so it can have side effects. Generally these extra values would be part of the user-data object, but we've already highjacked it for the execution context. When Pallene has closures, we may use the them to address this lack of execution context. Closures can capture the variables around them, and consequently we may use these captured variables for side effects. Nevertheless, we'll have to revisit this design and implementation.

Returned arrays

We've discussed how to deal with array parameters. Since they are owned by Pallene, we know that the underlying memory is valid and its size, consequently the FFI can ensure safe memory access. However, this is not always the case. Some external libraries return arrays which were not created by Pallene, such as `sqlite3`. Consequently, the flexible FFI has to support them in some way.

To this end, we'll have to add another unsafe feature. One approach is extending pointers with some sort of pointer arithmetic, which we could then dereference using the previously discussed `unsafe_deref` method. However, in the context of arrays, we believe this isn't the best idea. Usually they are accessed many times at different indices; with this approach, each individual memory access would be unsafe, and the programmer would have to ascertain they are indeed reading a valid index. Memory ownership is another concern; if the programmer holds this pointer for longer, they have to make sure the memory will remain valid, and (if needed) to free it later.

We believe the better approach is using `CArrays` to support returned arrays; we represent them first as normal opaque pointers, and add a method to `CArrays` that takes pointers and copies their content over. Since we know the size of the `CArray`, once we copied over the content of the pointer returned from C, any access to the array is safe; the memory is sure to be valid and remain so, accesses are bound checked, and it will be freed automatically. The programmer no longer has to worry about unsafeness after a single dangerous call.

In Code 23, we show the body of the `sqlite3` callback. Although the arrays are technically parameters to the callback, this function may only receive types that can appear in the return position of a foreign function. After instantiating `CArrays` with the proper size and initial value, we are ready to copy the contents of the pointers over. On lines 7 and 10, we show the new unsafe method: `unsafe_copy`. It takes the size to be copied (Pallene will raise

an error if it is greater than the `CArray`) and the pointer. After this, we can access their contents without issues.

Code 23: array_ret.pln

```
local function callback(
  argc: integer,
  argv: StringArray,
  azColName: StringArray
): integer
  local lines: C{CharPointer} = C{argc, NullCharPointer}
  lines:unsafe_copy(argc, argv)

  local cols: C{CharPointer} = C{argc, NullCharPointer}
  cols:unsafe_copy(argc, azColName)

  for i=1,argc do
    print(cols[i]:unsafe_to_string(), lines[i]:unsafe_to_string())
  end

  return 0
end
```

4.3 Feature list

In this section, we summarize the design discussed in the examples section, briefly listing the features supported by our FFI. Additionally, we discuss a few features purposefully not supported.

First, we created two new types in Pallene programming language: Pointers and CArrays. These were added for the FFI, although they can be used independently of it; they are part of the language now.

Pointers The pointer type was created in Pallene to represent C pointers. They may be pointers to incomplete types, pointers to basic C numeric types, pointers to functions, pointers to `structs`, void pointers, and pointers to other pointers. Pointers to complete types may be unsafely dereferenced, yielding a value of the type the pointer points to.

CArrays CArrays are a new collection type created in Pallene, with their internal representation matching arrays of C. They may hold elements of numeric types or of pointer types, have a fixed size, and are bound-checked by Pallene. The reasoning for them is that marshalling Lua tables may be too expensive. Since CArrays match arrays of C, the FFI doesn't need to marshal their contents.

Invoking foreign functions is the main purpose of an FFI. In our design, these functions are mapped into Pallene native functions, and the ability to call

foreign functions is dependent on their type. For safety reasons, it is important to distinguish between values at the parameter position of function signatures, and values at the return position of function signatures. The following list enumerates which types are supported by the FFI.

Numeric types The FFI supports all basic C numeric types, which may appear at the parameter position and at the return position of a foreign function.

Pointer types The FFI supports pointers through the newly added pointer type. They may appear at the parameter position and at the return position of a foreign function.

References to numbers and pointers The FFI supports references to numbers and pointers at the parameter position of foreign functions, using the modifiers `in`, `retout`, and `retinout`.

References to references The FFI supports references to references at any position of a foreign function, through the use of the newly added pointer type. We may unsafely dereference these pointers to get the underlying value.

Strings The FFI supports strings at the parameter position of a foreign function through the `string` type. Returned strings are supported through the pointer type, which may be unsafely converted into a Pallene string.

Structs The FFI supports external C `struct` definitions. When external C `structs` are declared in the foreign block, the FFI automatically creates a new Lua record of the appropriate type; this new Lua record is marshalled to and unmarshalled from the external C `struct` when values cross language boundaries.

At the parameter position of a foreign function, `structs` are available as references when accompanied by the appropriate modifier like `in`, `out` and `inout`, or as values when without modifiers. At the return position, they may only safely appear as values; returned `struct` references are supported only as pointer types, which may be unsafely dereferenced later.

Arrays The FFI supports arrays at the parameter position in two different ways. The first is through Lua arrays, which are marshalled into the appropriate C array, and the second is through the newly added CArray type. At the return position, arrays are supported only through the

pointer type. If we wish to access their contents, there is an unsafe function to copy the contents of a pointer into a CArray.

Restricted callbacks Finally, the FFI supports a restricted kind of callbacks, where there's a user data field as part of the callback signature. Currently, callbacks are limited in what they can access; the only way they can have side effects is through global variables.

In addition to functions, our design supports numeric compile-time constants. These include macro definitions, global constants and `enums`, or constant pointers like `NULL` and pointers to external functions. This feature is not commonly seen in FFIs.

As discussed, unless we found a concrete use of an unsafe feature, we didn't add it to the FFI. There are two features we consciously excluded for this reason: they are unsafe and we couldn't observe a library that requires them.

Typecasting pointers In Pallene, we may not use the FFI to create a pointer from another pointer of a different type. Allowing arbitrary typecasting of pointers is dangerous all kinds of ways. Nevertheless, the FFI allows for limited, safe conversion of pointers when they cross language boundaries: functions that return `void*` may be declared as returning a pointer of any type. However, once in Pallene, there's no mechanism for changing the type of a pointer. One of the flexibility reasons for adding this feature is related to type punning, but we could observe no library that requires this.

Writing to arbitrary memory The FFI design does not have a mechanism to write to arbitrary memory. We've added the feature to read from pointers (which is an unsafe feature) since we found libraries that required it, but we couldn't find a library that requires writing to pointers. Although cumbersome, the programmer may use `libc`'s `memcopy` to this end if so inclined.

In the future, if there's empirical evidence these two features are needed, we may expand the design to include them. Until then, if required, the programmer may circumvent their absence by writing C code directly. Since a C compiler is already a dependency, this isn't the end of the world.

4.4 Evaluation

The design presented on this chapter has safety at the forefront, while also trying to conciliate it with flexibility. The flexible FFI cannot be fully safe; there's a trade-off between flexibility and safety that any design has to contend with. To guide us on this dilemma, we've used an empirical approach, wherein we studied a sample of mature, well used libraries, using them as a canon of sorts. This way, unless we see a concrete example that forces us to compromise on safety, our design favors sacrificing hypothetical flexibility.

All interactions with foreign code start with a description of its interface. To this end, we require that the programmer writes the library's interface in Pallene, using a special language for this purpose, and that the programmer supplies the library's header files. Of relevance for the design, the declarations in Pallene gives us richer information, which enables us to provide safety guarantees in certain cases.

Although the FFI cannot be fully safe, we can certainly design an FFI that *mostly* preserves safety, playing nice on the well-behaved cases, while still being able to fall back to dangerous features if needed. General use of certain patterns, such as pointers, are certainly unsafe. However, these can be broken into separate sub-patterns that might be safe: we may be able to avoid dangerous behaviour by restricting certain uses.

This restriction is made possible by our special interface description language inside Pallene. For example, while a function declaration in C may indicate it receives a pointer, in Pallene we can specialize this type: it may be an `retout` number, an opaque pointer, an array, a string, and so on. This enables us to break pointers in various sub-cases, some of which are safe. Instead of providing an unrestricted pointer type that works in all situations but is dangerous, we provide multiple ways of using pointers; some are safe and enough for well-behaved interactions, and some are unsafe but can deal with the more ill-behaved interactions. This way, we can design a safer but still flexible FFI, by reducing the number of dangerous operations the programmer has to do, while still allowing them if needed.

Given that the flexible FFI must allow unsafe interactions, it is important to know exactly where these are. Ideally, we'd want the yellow caution tape to be strictly outside Pallene. Since this is not possible, we the design addresses this by making the yellow tape evident; all unsafe use of the FFI is explicitly named as such. If the programmer does not use the unsafe parts of the FFI, dangerous code is strictly outside Pallene.

5
Implementation

The FFI implementation and design are intimately related; they were developed together incrementally, influencing each other along the way. Nevertheless, in this chapter we'll go into the implementation details of the FFI.

We'll discuss the code generated by the FFI, showing how it performs foreign calls, supports macro procedures and constants, and enforces correctness of bindings, while balancing implementation complexity. We'll also discuss the implementation of new types, and how they are integrated with Lua. Finally, we'll analyse the performance of our implementation, both in the context of using external libraries—either using Pallene as a system's language or as a bridge to Lua—and of speeding up Pallene with faster data structures.

Our FFI implementation was developed on top of the reference Pallene ahead-of-time compiler, written in Lua[1]. The Pallene compiler is quite conventional. After a standard parsing step, it first converts the program to a high-level intermediate form, and then emits C code. This C code is finally handed to a C compiler like GCC. For the FFI implementation, perhaps the most relevant tool we have at hand is a C compiler, which we've aggressively used to our advantage.

We implemented the FFI in an ahead-of-time schema. It has its own parser, which is invoked by the core Pallene parser to read the foreign block. It is written using LPeg (IERUSALIMSCHY, 2009), a text pattern-matching tool based on parsing expression grammars (FORD, 2004). The rest of the FFI follows a similar pattern: the Pallene compiler does what it has to do, and calls separate FFI modules to deal with foreign stuff. This way, we minimize changes to the Pallene code base, keeping the implementation modular and isolated from the core of Pallene.

5.1
Interacting with Foreign Code

The first step in interacting with external libraries is describing their interface. As discussed before, the FFI uses a hybrid approach, requiring both header files and host language declarations. To use header files, the FFI has to understand C in some capacity; we'd need something like a C compiler front-end.

[1] The source code for this dissertation is available in a git repository, forked from the reference Pallene compiler on GitHub: `https://github.com/GCdePaula/pallene`

One way FFIs understand C is by reusing an already existing front-end, However, since there's no standard for front-ends, such an FFI would depend on a specific implementation. For example, the Swift compiler—which already depends on LLVM—is shipped with Clang (LATTNER, 2003; LATTNER, 2007; LATTNER, 2008). Nevertheless, for portability reasons, we cannot depend on one specific C compiler implementation; what we have available to us is *an* ISO C compiler. Another way FFIs understand C is by reimplementing the required parts of a C compiler front-end. This can be done in a portable way, but is quite complex. We used a differnt approach altogether; the FFI leverages the already required C compiler to this end, avoiding the need to parse and understand C ourselves.

We use the C compiler to check for the correctness of bindings. One rule Pallene follows is that generated C code cannot fail to compile: C compilation errors are a bug in the Pallene implementation. We decided to slightly weaken this rule for this specific case: wrong bindings will raise a C compilation error, and the Pallene compiler will output this error to the user. As such, if the declarations on the Pallene side do not match the ones on the library's header file, we will catch them at compile-time. This eliminates an entire class of errors that may happen in the FFI, making it safer. We lose nice error messages, but we gain portability and a reduction in the FFI's implementation complexity.

Concretely, when we use the FFI, the Pallene compiler generates an extra C file, besides the main C file. This extra file acts as a bridge, which is later compiled and linked with the main Pallene object. The bridge C file is where we've weakened Pallene's compilation error rule: it is the only artifact that can fail at the C compilation stage. This failure can only happen due to an incompatible binding declaration at the foreign block. This way, we can avail ourselves of the assistance of a C compiler's front-end to check for the correctness of bindings, without depending on one specific C compiler implementation or needing to reimplement one ourselves.

The reason for doing foreign bindings through a bridge is twofold. First, we need to include foreign libraries' header files. If we were to do include directly on the main Pallene file, we'd pollute it; there would no way to deal with name collisions. Second, we wanted to create a clear separation between artifacts that could fail to compile due to wrong binding declarations and artifacts that couldn't fail to compile at all.

For example, in Code 24 we show a simple function binding definition in Pallene. In Code 25 we show the generated bridging file. The `include` statement in Pallene becomes a preprocessor include directive in the bridge. The `function` statement in Pallene creates a C function in the bridge, which

is exported and then linked to the main Pallene object. Note that the bridge function has a name generated by the FFI. Its signature is entirely defined by the programmer, through the foreign block's `function` statement. The bridge function's body consists of simply calling `sin`, passing all the arguments through.

If the bindings do not match, then the body of this bridging function is not valid C and will yield a compilation error. For example, if the programmer told the FFI that the function `sin` receives a pointer, then the generated file would try to pass a pointer to `sin` (declared in the `math.h` header), which would raise a compilation error.

Using header files also enables the FFI to support macro procedures. Macro procedure definitions don't generate code and don't export symbols, consequently there's nothing to link against. However, our linking is done against a bridging C function, which generate symbols normally. Since we include the header files where the macro procedures are defined, we can directly call them inside this bridging function's body, thus exposing macros to the FFI. The code in Code 24 naturally handles macros: the invocation does not require `sin` to be a function, a macro would work just as well. A final consideration is that macros aren't typed, but there are certain types of parameters that are incompatible. These incompatibilities are caught at compile-time as normal.

Code 24: fn_binding.pln

```
foreign
  include <math.h>
  function sin(cdouble): cdouble
end
```

Code 25: fn_bridge.c

```
#include<math.h>

double pln_foreign_function_01 (double arg_0)
{
    return sin(arg_0);
}
```

The most relevant operation of any FFI is performing foreign function calls. On our FFI, these go through our C bridge first. As shown in Chapter 4, our foreign calls are mapped to native Pallene function calls. To this end, we've created the *foreign call* primitive. This primitive, which appears as a new opcode in the Pallene intermediate representation, is responsible for

three sequential operations: translating the arguments into C values, calling the bridge function, and translating the return value into Pallene values.

This translation depends entirely on each value's data representation: Pallene numbers use the standard C conversions; records and arrays are marshalled into the appropriate value; and `CArrays`, opaque pointers and strings are trivially translated.

The bridge C file also enables constants. For example, in Code 26 we show a simple constant binding definition in Pallene. In Code 27 we show the generated bridging file: it simply defines a global variable, assigning the compile-time constant (provided at Pallene) to it. The name of this global variable is generated by the compiler. The main Pallene C file only declares this global variable, which is linked to the definition at the bridge C file.

This approach allows for any kind of compile-time constant. Although things like macro definitions do not generate code (and don't even have a type), since our linking is done against a global variable—which generates symbols normally—we can expose constants to the FFI. Note that if the library defined constant is not compatible with the programmer provided type, the compiler will raise an error at the C compilation stage.

Code 26: const_binding.pln

```
1 foreign
2   include <math.h>
3   constant M_PI: float
4 end
```

Code 27: const_bridge.c

```
1 #include<math.h>
2
3 const double pln_foreign_constant_01 = M_PI;
```

5.2 Implementing New Types

Pallene is a companion language to Lua, sharing its runtime. As such, Pallene is tightly integrated with Lua; we can call Pallene functions from Lua as if they were normal functions, passing Lua values to them and receiving Lua values back from them. This imposes an important restriction to new Pallene types, since their values must also exist in Lua.

Another thing to take into account is garbage collection. Pallene objects, also being Lua objects, are subjected to garbage collection as normal. Since new Pallene types must exist in the two languages, we have to consider how the garbage collector will trace these objects. Finally, we must also consider how Pallene will tag-check new types. Values that come from Lua must have their type identifiable in some way at runtime, so that if their type do not match the one expected, Pallene can raise a runtime error.

A technique used by Ligneul (LIGNEUL, 2019) is to use Lua's full userdata to represent new types. In his work, he added Pallene records to the language, a faster alternative to Lua tables while inside Pallene. Before diving into the details, we'll do a brief revision of Lua's userdata. There are two types of userdata: full userdata and light userdata. Although they share the same name, they are quite different.

Full userdata are objects: they have reference semantics, are only equal to themselves, and are traced by the garbage collector like all other objects. They offer a raw memory area with no predefined operations in Lua, which we can use to store anything. This raw memory is opaque to the Lua runtime. The magic of full userdata manifests itself when we add metatables. This gives us both a way to define new operations on userdata through metamethods, and a way to tag userdata. Another feature of userdata is the so called user values. They are a special part of the usardata, consisting of a set of Lua values, which can be traced by the garbage collector.

Light userdata, on the other hand, represents a single C pointer value. They are not buffers, like full userdata, but bare pointers. They have value semantics, are equal to any other userdata which holds the same C pointer, and are not traced by the garbage collector. They do not have any other data like metatables associated to it.

Ligneul used full userdata to implement Pallene records. For each user-defined record, a unique metatable is created, shared across all instances of userdatum of this kind. With this, we're able to tag-check records: when we receive a value from Lua, we check at runtime if its metatable matches the unique metatable for that specific record type.

To store the record's data, first he split the record's fields into object types (like tables and strings, which are traced by the garbage collector), and into value types (like numbers). The value types go into the raw memory part of the userdata, and the object types go into the user values part of the userdata. This way, the record anchors the objects it stores. Were this not the case, the garbage collector could collect objects still in use, creating dangling pointers.

With this revision on userdata, we can dive into how we implemented our

new types for the FFI. There are two new types added: opaque pointers and CArrays. The latter uses a similar technique to Pallene records. The former's approach is a bit more novel.

Opaque pointers posit an interesting dilemma. In essence, they are a single `void*` value. At first glance, it would seem that a light userdata would be the perfect fit. However, this introduces issues with Pallene's tag-checking. Since light userdata have no other data associated to it, we cannot apply the metatable technique used in full userdata for tagging. If we were to receive in Pallene a light userdata from Lua, we'd have no way to check what type this pointer represents.

This implies that in Lua opaque pointers must be full userdata. However, we miss an important feature: that of pointer equality. Since full userdata are only equal to themselves, we could run into a situation of two userdata holding the same pointer, but checking false for equality. Also, it would be very convenient to treat opaque pointers as just pointers in Pallene. In fact, our design depends on value semantics for opaque pointers, for the `retout` modifier.

This is the dilemma, which we've solved using a hybrid approach, using both kinds of userdata. The solution is first dealing with opaque pointers in Pallene as pointers. Then, as soon as we need to transform it into a Lua value (*e.g.* we're returning an opaque pointer to Lua) we wrap it into a full userdata.

To make sure we keep the equality semantics we want in Lua, we've made this wrapper full userdata into a singleton of sorts. To this end, each opaque pointer type declared in the foreign block has its own Lua table. This table is a weak table, and has light userdata keys (corresponding to the opaque pointer itself) and full userdata values (corresponding to the wrapper singleton userdata). When we want to wrap the pointer, we first check the table (indexing it with the pointer) to see if it contains a corresponding full userdata. If it does, we use the already existing full userdata (which is an object and has reference semantics). Otherwise, we create a new wrapper full userdata and store it in the table.

This way, since there will only be a single full userdata in existence with said pointer, we have the equality semantics we wanted in Lua. Since the table storing all pointer instances is weak, if this userdata becomes unreachable it is collected by the garbage collector. When we receive an opaque pointer from Lua, we can tag-check it normally because full userdata has a metatable. If it passes the tag check, we extract the raw pointer from the full userdata, turning it into a type with value semantics.

CArrays have a more standard implementation, matching the techniques

used in Palleen records. We use full userdata as they were intended: a buffer, storing the CArray's elements in contiguous memory. The userdata's metatable is used to verify the userdata is both a CArray and that the elements' type are correct. Since we can only store elements of the correct type, we don't need to tag-check the elements, making the implementation much easier than common Pallene arrays. As discussed, CArrays can only contain simple types. This way, we don't have to worry about garbage collection.

5.3 Performance Analysis

The primary goal of Pallene is to aid programmers that seek better performance in Lua. Consequently, a good Pallene FFI has to be aligned with this goal; indeed one of our criteria is performance. In this section, we'll discuss the cost of calling foreign functions and, where relevant, measure its speed. We will not measure everything; the parts which do not add overhead when compared with pure C will be skipped. Additionally, we'll measure the performance of CArrays in comparison with standard Pallene arrays, and the performance of using Pallene as a bridge between Lua and foreign languages in comparison with the C API as an extension mechanism.

We performed our experiments on a macOS machine with an Intel Core i9–9880H CPU (2.30GHz). The C compiler used was Clang (version 12.0.5). In all of our experiments, we measured the running time of the relevant code snippet five times and present the median of those measurements. The running time was timed within the program itself, using a Lua library called `chronos`[2]. We used the default Pallene optimizations, which compiles C code with the `-O2` optimization flag.

Marshalling and Unmarshalling Performance

Calling a foreign function in Pallene through the FFI involves three general steps: translating the arguments from Pallene, invoking the foreign function, and translating the results from C.

Let's start with the second step: the invocation. The FFI doesn't call the foreign function directly; it first calls the bridge function, which then calls the foreign function itself. The second call, for the simple cases, interestingly, can be easily optimized to a single tail call. Consequently, the indirection overhead can be generally reduced to about a single jump instruction. Inspecting the code generated by both GCC and Clang with the -O2 flag, the bridge is indeed

[2] `https://luarocks.org/modules/ldrumm/chronos`

optimized to a tail call.

To analyse the cost of the tail call, we compared the running time of two C programs calling the `sin` function from `libc`: one directly and the other using the bridge intermediary function. The time difference was of less than half a percent. Since the indirection has a negligible impact on performance, we conclude that the invocation step of the FFI also has negligible overhead when compared to an invocation done by a pure C program.

Therefore, any additional cost of invoking a C function from Pallene cannot be attributed to the second step. The first and third steps, on the other hand, require more analysis. On simple cases like numeric functions, translating arguments from Pallene and the results from C also have no additional cost; since Pallene and C use the same data representation for numbers, the translation is trivial. However, on more complex cases where the translation requires marshalling arguments and unmarshalling results, there's certainly additional cost. There are two instances where this happens: translating Lua arrays and translating Lua records.

Lua arrays are Lua tables. To call a C function that takes an array requires marshalling the underlying table. Recall that we added CArrays to the language to avoid the costs of translating Lua arrays; since CArrays already have the same representation as arrays in C, the FFI doesn't need to marshal and unmarshal them. Consequently, we used CArrays as the baseline of our measurement, comparing it with the cost of marshalling Lua arrays. For this analysis, we measured with a foreign function that is $O(n)$. To this end, we wrote a C library that receives an array of doubles and its size, and returns the sum of its elements.

Marshalling a Lua array requires acquiring a large enough block of memory, and copying each element over, translating them in the process. Under the hood, the FFI creates a CArray for this purpose. An optimization we did was to reuse this CArray for future calls, instead of re-instantiating one each time. To this end, the FFI has a weak table where the keys are Lua arrays, and the values are CArray, tying lifetime of the latter to the former. When we need to marshal a Lua array, we first check this weak table; if it contains a large enough CArray we reuse the memory, otherwise we create a new one, storing it into the weak table. This eases pressure on the garbage collector, particularly if we are calling the foreign function in a loop, passing the same array

We ran this experiment for varying sizes of arrays, starting at 10 and exponentially increasing to 10^6, repeating the foreign call enough times for the benchmark to take a reasonable amount of time. We reused the same Lua array

and CArray for all repetitions, otherwise we'd be adding the cost of garbage collection to our experiment. Due to the previously discussed optimization, the Lua array is always reusing the same block of memory when marshalling, instead of re-instantiating one each time. The results of these experiments are shown in table Table 5.1.

Size/Repetitions	**Structure**	**Time in seconds**	**Time normalized**
$10^1/10^8$	CArray	1.83s	1.00
	Lua array	3.52s	1.92
$10^2/10^7$	CArray	1.98s	1.00
	Lua array	2.62s	1.32
$10^3/10^6$	CArray	2.00s	1.00
	Lua array	2.47s	1.23
$10^4/10^5$	CArray	1.99s	1.00
	Lua array	2.48s	1.25
$10^5/10^4$	CArray	2.00s	1.00
	Lua array	2.53s	1.26
$10^6/10^3$	CArray	2.00s	1.00
	Lua array	3.57s	1.78

Table 5.1: Running time of experiments on marshalling Lua arrays.

For very small arrays, Lua arrays are particularly slow. We believe the cost of acquiring memory is more relevant in this case; we are observing not just the marshaling cost, but also the constant cost of resource acquisition. When we increase the size, the cost of acquiring memory is diluted in the cost of marshalling.

This slowdown for small arrays could be mitigated with a modified implementation, which would use a use a small, constant sized buffer allocated in the stack to this end. The FFI would check at runtime whether the size of the array to be marshalled is smaller than this size constant, and use the stack allocated buffer if so. Otherwise, the FFI would acquire a heap buffer as normal. We will consider this change in future releases.

As we increase the size of the Lua array, there's some threshold that makes the marshalling cost go up again. We believe this is due to the size of CPU cache; as the arrays grow they no longer fit on it.

Lua records are also Lua tables. To call a C function that takes a structure requires marshalling the underlying table. For this analysis, we compared the

cost of calling a function that takes a structure with the cost of calling a function that takes the structure's fields unpacked. Concretely, we wrote a second function that, instead of receiving a structure with four fields of type double, receives four doubles. Both of them will return the argument's sum. Since calling this second function from Pallene has no translation overhead, it will do as our baseline.

For this benchmark, we used two different foreign functions: one that takes a `stuct` with four fields, each of type double, and another that takes four doubles. These functions returns the sum of the four doubles. We repeated the foreign call $5 * 10^8$ times, and allocating and initializing the Lua record a single time. The results of these experiments are shown in table Table 5.2. As expected, marshalling Lua records is slower. Whether this extra cost is meaningful depends on the application.

	Time in seconds	**Time normalized**
Unpacked fields	3.51s	1.00
Lua record	5.30s	1.51

Table 5.2: Running time of experiments on marshalling Lua records.

Faster Data Structures

Another use of the FFI is orthogonal to calling foreign functions. Since we added new types to Pallene, they may be used without necessarily interacting with foreign code, as a faster alternative to preexisting Pallene types. Of interest, we measured the performance of CArrays, comparing them with Lua arrays.

Ligneul's work offers some insights on the performance of Lua arrays. The size of the array significantly impacts performance; larger arrays increase last-level-cache (LLC) misses, reducing the speed. CArrays are more compact than Lua arrays, since the latter have to store extra information like the tag. We expect this compactness will reduce LLC misses, increasing performance.

Furthermore, CArrays' elements do not need to be tag-checked. Since we can only assign elements of the correct type, the type of all the elements are entirely encapsulated in the CArray tag. Finally, the size of CArrays is immutable; unlike tables, setting an element out of bounds does not cause the array to increase, and instead triggers an error. We hypothesise that this size immutability makes it easier for the compiler to reason about bound-checking.

Matmul For this benchmark, we multiplied two matrices of size 800, repeating the computation 3 times. The matrices were allocated and initialized once. The speedup of CArrays was impressive, taking only 27% of the time it took Lua arrays, shown in Table 5.3. We believe this speedup is mostly due to the smaller representation of CArrays, which reduces LLC misses, as well as easier to reason about bound-checking.

Spectral Norm In this experiment, we ran the Spectral Norm benchmark with input 5500, repeating the computation 2 times. The CArrays version took 74% of the running time of the Lua array version, shown in Table 5.3. We believe this speedup is due to the smaller representation of CArrays, which reduces LLC misses, but not as pronounced as Matmul.

Binsearch For this benchmark, we ran a binary search on an array of size 10^8, repeating the computation 10^8 times, allocating and initializing the arrays once. The speedup of CArrays was insignificant, shown in Table 5.3. Since memory access is all over the place, the compactness of CArrays is not relevant for reducing LLC misses.

	Type	**Time in seconds**	**Time normalized**
Matmul	Lua array	2.86s	1.00
	CArray	0.76s	0.27
Spectral Norm	Lua array	3.33s	1.00
	CArray	2.46s	0.74
Binsearch	Lua array	7.74s	1.00
	CArray	7.66s	0.99

Table 5.3: Experiments comparing the performance of CArrays and Lua tables.

Pallene as a Bridge

Finally, another use of the FFI is as a bridge between Lua and other languages. The primary advantage of the FFI is related to ease of use; there's no need to write adapter glue code, as one would when using the C API. Nevertheless, the performance of the FFI when compared to the API matters. We've measured this using two functions numeric functions from `libc`: `sin` and `ceil`, shown in Table 5.4.

	Bridge	Time in seconds	Time normalized
sin	C API	4.46s	1.00
	Pallene FFI	4.27s	0.96
ceil	C API	9.28s	1.00
	Pallene FFI	7.92s	0.85

Table 5.4: Result of measurements calling foreign functions from Lua using the C API and using the Pallene FFI

sin For this benchmark, we used the `sin` function from `libc`, calling it from Lua 10^8 times. Using Pallene as a bridge yielded very modest speedup, taking only 96% of the running time of the C API. This speedup is likely due to the Pallene implementation bypassing the C API.

ceil For this benchmark, we used the `ceil` function from `libc`, calling it from Lua $5 * 10^8$ times. Using Pallene as a bridge yielded a larger speed up when compared to `sin`, taking only 85% of the running time of the C API. This difference can be attributed to the running time of `sin` versus `ceil`: the latter is compiled to a single instruction, which highlights the bridge overhead.

5.4 Evaluation

The cornerstone of the proposed implementation is including libraries' header files. To make use of them, the implementation has to understand C at some capacity; it requires a preprocessor, a parser, and to comprehend C types. Implementing these things is reasonably complex, and using an existing C compiler front-end is not portable, since we'd depend on one specific implementation. We went for a different solution altogether; to understand C, our FFI leverages the C compiler we already depend on, eluding the need to reimplement these things ourselves or to depend on a specific front-end. This reduced the implementation complexity and improves portability.

A common source of bugs in FFIs comes from wrong interface description. Since the FFI directly uses header files, the implementation can check for the correctness of bindings at compile-time; code generated by our compiler will be invalid C if the descriptions don't match, and will fail at the C compilation stage. This comes at the cost of nice error messages to users, but eliminates an entire class of bugs in the FFI.

One way FFIs can lack in flexibility is being restricted to things that

generate symbols; macros don't generate symbols, and consequently cannot be used by many FFIs. Through the use of header files again, we can use things that do not generate symbols like macro definitions; our FFI can invoke macro procedures as if they were normal functions, and use macro constants as if they were global variables.

6
Conclusions

This work presented an FFI for Pallene with five main criteria: flexibility, safety, portability, performance and implementation complexity.

The presented design has safety at the forefront, while also trying to conciliate it with flexibility. The issue of safety arises when we interact with foreign code. Unlike Pallene, C is an unsafe language: wrong use of C may yield low-level errors, which cannot be understood in terms of the language itself. Not all uses of C is unsafe. However, restricting the FFI only to uses where we can ensure safeness would significantly impair its usefulness; we wouldn't be able to interact with most libraries. In this sense, there's a trade-off between flexibility and safety that any FFI has to contend with. In the context of a safe host language, if the FFI exposes too much of C, we end up bringing unsafeness into our language. If we expose too little, we end up restricting the number of APIs the FFI can support. Our design addressed this flexibility/safety dilemma in three relevant ways.

First, we used an empirical approach to guide us for when we should compromise on safety in favor of flexibility. We studied a sample of mature, well used libraries, using them as a canon of sorts. This way, unless we saw a concrete example that forced us to compromise on safety, our design favored sacrificing hypothetical flexibility. This empirically informed decision making is one of the basis of our design.

Second, we can design an FFI that *mostly* preserves safety, playing nice on the well-behaved cases, while still being able to fall back to dangerous features if needed. General use of certain patterns are certainly unsafe (*e.g.* pointers). However, these can be broken into separate sub-patterns that might be safe: we may be able to avoid dangerous behaviour by restricting certain uses. For example, we broke up pointers in various sub-cases, some of which are safe. Instead of providing an unrestricted pointer type that works in all situations but is dangerous, we provided multiple ways of using pointers, some of which are safe and are enough for well-behaved interactions, and some of which are unsafe but can deal with the more ill-behaved interactions. As such, we create a safer but still flexible FFI, by reducing the number of dangerous operations the programmer has to do, while still allowing them if needed.

Third, given that the flexible FFI must allow unsafe interactions, we clearly marked the dangerous parts of the FFI. All unsafe use of the FFI is explicitly named as such. Ideally, we'd want the yellow caution tape to be

strictly outside Pallene. Since this is not possible, we addressed this by making the yellow tape evident. If the programmer does not use the unsafe parts of the FFI, dangerous code is strictly outside Pallene.

On the topic of defining interfaces, we've chosen a hybrid approach, using both the library's header files and declarations at the host language. The former is correct, but poor on information; the latter is not guaranteed to be correct, but may carry more details. Using them together ensures that the bindings are correct at compile-time, eliminating an entire class of bugs in the FFI, and have richer declarations, making the FFI safer. Another advantage of importing header files is that the FFI can interact with things that do not generate symbols, such as preprocessor macros. This enabled our FFI to invoke macro procedures and to use macro constants.

In conclusion, the purpose of this work was to create a mechanism for extending Pallene through an FFI. The goals were to enable the use of Pallene as a bridge between Lua and other languages, and to increase the range of Pallene as system-language counterpart of Lua. Additionally, the newly added data structure can be used as a faster alternative to Pallene preexisting types. We hope our design and implementation contributes to these goals.

7
Bibliography

BARZILAY, E.; ORLOVSKY, D. Foreign interface for PLT Scheme. In: **Proceedings of the Fifth ACM SIGPLAN Workshop on Scheme and Functional Programming**. [S.l.: s.n.], 2004. p. 63–74.

BEAZLEY, D. M. et al. SWIG: An easy to use tool for integrating scripting languages with C and C++. In: **Tcl/Tk Workshop**. [S.l.: s.n.], 1996. v. 43, p. 74.

BLUME, M. No-longer-foreign: Teaching an ML compiler to speak C "natively". **Electronic Notes in Theoretical Computer Science**, Elsevier, v. 59, n. 1, p. 36–52, 2001.

FACEBOOK. **luaffifb**. 2015. Disponível em: <https://github.com/facebookarchive/luaffifb>.

FAST, T.; WALL, T.; CHEN, L. **Java Native Access (JNA)**. 2007. Disponível em: <https://github.com/twall/jna>.

FISHER, K.; PUCELLA, R.; REPPY, J. Data-level interoperability. In: **Electronic Notes in Theoretical Computer Science**. [S.l.: s.n.], 2000. p. 2001.

FISHER, K.; PUCELLA, R.; REPPY, J. A framework for interoperability. **Electronic Notes in Theoretical Computer Science**, Elsevier, v. 59, n. 1, p. 3–19, 2001.

FORD, B. Parsing expression grammars: a recognition-based syntactic foundation. In: **Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages**. [S.l.: s.n.], 2004. p. 111–122.

GREEN, A. **The libffi home page**. 1996. Disponível em: <https://www.sourceware.org/libffi/>.

GUALANDI, H. M. **The Pallene Programming Language**. Tese (Doutorado) — PUC-Rio, 2020.

GUALANDI, H. M.; IERUSALIMSCHY, R. Pallene: A statically typed companion language for Lua. In: **Proceedings of the XXII Brazilian Symposium on Programming Languages**. [S.l.: s.n.], 2018. p. 19–26.

GUALANDI, H. M.; IERUSALIMSCHY, R. Pallene: A companion language for Lua. **Science of Computer Programming**, Elsevier, v. 189, p. 102393, 2020.

IERUSALIMSCHY, R. A text pattern-matching tool based on parsing expression grammars. **Software: Practice and Experience**, Wiley Online Library, v. 39, n. 3, p. 221–258, 2009.

IERUSALIMSCHY, R. **Programming in Lua**. 4th ed. ed. [S.l.]: Roberto Ierusalimschy, 2016.

IERUSALIMSCHY, R.; FIGUEIREDO, L. H. D.; CELES, W. Passing a language through the eye of a needle. **Communications of the ACM**, ACM New York, NY, USA, v. 54, n. 7, p. 38–43, 2011.

IERUSALIMSCHY, R.; FIGUEIREDO, L. H. D.; FILHO, W. C. Lua—an extensible extension language. **Software: Practice and Experience**, Wiley Online Library, v. 26, n. 6, p. 635–652, 1996.

IERUSALIMSCHY, R.; FIGUEIREDO, L. H. D.; FILHO, W. C. The implementation of Lua 5.0. **J. UCS**, v. 11, n. 7, p. 1159–1176, 2005.

LATTNER, C. **The LLVM compiler infrastructure**. 2003. Disponível em: <https://llvm.org/>.

LATTNER, C. **The Swift Programming Language**. 2004. Disponível em: <https://swift.org/>.

LATTNER, C. **Clang: a C language family frontend for LLVM**. 2007. Disponível em: <https://clang.llvm.org/>.

LATTNER, C. LLVM and Clang: Next generation compiler technology. In: **The BSD conference**. [S.l.: s.n.], 2008. v. 5.

LIANG, S. **The Java native interface: programmer's guide and specification**. [S.l.]: Addison-Wesley Professional, 1999.

LIGNEUL, G. de Q. **The Implementation of Records in Pallene**. Dissertação (Mestrado) — PUC-Rio, 2019.

MASCARENHAS, F. **Alien**. 2009. Disponível em: <https://github.com/mascarenhas/alien>.

MCKASKILL, J. R. **luaffi**. 2011. Disponível em: <https://github.com/jmckaskill/luaffi>.

OUSTERHOUT, J. Scripting: higher level programming for the 21st century. **Computer**, v. 31, n. 3, p. 23–30, 1998.

PALL, M. **LuaJIT, a just-in-time compiler for Lua**. 2005. Disponível em: <http://luajit.org/luajit.html>.

PALL, M. **LuaJIT FFI extension**. 2005. Disponível em: <https://luajit.org/ext_ffi.html>.

PETRICEK, T. What we talk about when we talk about monads. **The Art, Science, and Engineering of Programming**, AOSA, v. 2, n. 3, p. 12, 2018.

SIEK, J. G. et al. Refined criteria for gradual typing. In: SCHLOSS DAGSTUHL-LEIBNIZ-ZENTRUM FUER INFORMATIK. **1st Summit on Advances in Programming Languages (SNAPL 2015)**. [S.l.], 2015.

WADLER, P. Comprehending monads. In: **Proceedings of the 1990 ACM Conference on LISP and Functional Programming**. [S.l.: s.n.], 1990. p. 61–78.

YALLOP, J.; SHEETS, D.; MADHAVAPEDDY, A. **ctypes**. 2014. Disponível em: <https://github.com/ocamllabs/ocaml-ctypes>.

YALLOP, J.; SHEETS, D.; MADHAVAPEDDY, A. Declarative foreign function binding through generic programming. In: SPRINGER. **International Symposium on Functional and Logic Programming**. [S.l.], 2016. p. 198–214.

YALLOP, J.; SHEETS, D.; MADHAVAPEDDY, A. A modular foreign function interface. **Science of Computer Programming**, Elsevier, v. 164, p. 82–97, 2018.