A Foreign Function Interface for Pallene

Gabriel Coutinho de Paula
gpaula@inf.puc-rio.br
PUC-Rio
Rio de Janeiro, Rio de Janeiro, Brazil

Roberto Ierusalimschy
roberto@inf.puc-rio.br
PUC-Rio
Rio de Janeiro, Rio de Janeiro, Brazil

ABSTRACT

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. Furthermore, 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, we avail ourselves of the assistance of this compiler to gain a rudimentary understanding of C. This enables the FFI to verify the correctness of function bindings and to use C macro definitions.

CCS CONCEPTS

• **Software and its engineering → Source code generation**; **Extensible languages**.

KEYWORDS

Foreign Function Interface, Lua, Compilers

ACM Reference Format:
Gabriel Coutinho de Paula and Roberto Ierusalimschy. 2022. A Foreign Function Interface for Pallene. In *XXVI Brazilian Symposium on Programming Languages (SBLP 2022), October 6–7, 2022, Virtual Event, Brazil.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3561320.3561321

1 INTRODUCTION

Lua is often described as an extensible extension language, since one can easily extend Lua applications with external components and embed Lua in other host applications [13]. This characteristic of easy bridging between Lua and system languages is required for scripting [14]; Lua was designed to interoperate well with C, which is then used as a bridge between Lua and other components. This integration between Lua and C is achieved through Lua's C API [12].

In practice, however, there's an impedance mismatch between Lua and C that makes interaction between the two languages costly. This cost is manifested both conceptually and in performance, at the boundaries of scripting and system programming. In particular, the transition from dynamic language to static language and back gives rise to a fundamental shift in programming paradigm and style. Furthermore, the programmer must conciliate different typing schemas, requiring manual runtime type-checking and error handling in case of type mismatches.

The Pallene programming language [4, 9–11] was introduced partly to address this issue. Pallene is a system-language counterpart to Lua's scripting, always intended to be used side-by-side with Lua, sharing its runtime. It's a statically typed subset of the Lua programming language, used to write lower-level libraries and extension modules for Lua. The Pallene implementation was designed for performance, with an ahead-of-time compiler using type annotations to generate type specialized code that is more performant that its Lua equivalent. The compiler generates C code to be then handed to a C compiler; this means a C compiler is already a dependency.

To substitute C and fulfill the role of a system-language for Lua, Pallene must have a mechanism like a foreign function interface (FFI) to interact with other components, since one of the fundamental roles of C in the context of Lua is to bridge Lua and external libraries. We expect this FFI to be safe to use and rich enough to interact with arbitrary external code.

On this work, we present an FFI design and implementation for the Pallene programming language, targeting C. Our focus has been to create an FFI which is both *flexible* and *safe*. On flexibility, we've analyzed it by studying commonly used C libraries, and checking whether they can be directly used through the FFI. On safety, we've analyzed the ways one can cause arbitrary crashes or cause undefined behaviours from inside Pallene. These two goals are often at odds. On the one hand, if the FFI exposes too much of C we end up bringing unsafeness into our language. On the other hand, if we expose too little we end up restricting the number of libraries the FFI can support.

We've addressed this dilemma in three different ways. First, our design balanced flexibility and safety following an empirical approach, wherein we preferred to sacrifice hypothetical flexibility unless we see a real use case that forces us to compromise on safety. Second, we designed an FFI that *mostly* preserves safety, playing nice on the well-behaved cases, while still offering dangerous features for edge cases as a fallback. Third, since the flexible FFI must

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SBLP 2022, October 6–7, 2022, Virtual Event, Brazil
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9744-5/22/10…$15.00
https://doi.org/10.1145/3561320.3561321

allow unsafe interactions, we clearly marked the dangerous parts of the FFI.

We've also addressed how to improve the interface with external libraries. Several FFI designs [1] rely on user-supplied definitions of external libraries [1, 5, 16–19], instead of using the C header files with the original definitions. To use these header files would require the FFI to understand C in some level, which is no trivial matter. These user-supplied definitions, however, introduces two issues: we cannot be sure these definitions are correct—which makes the FFI less safe; and we cannot make use of C macro definitions—which makes the FFI less flexible.

Luckily, since Pallene's implementation already depends on a C compiler, we availed ourselves of the assistance of this compiler to understand C. This understanding comes in two ways. First, the Pallene FFI generates C code in such a way that C compilation fails when bindings are incorrect, catching incorrect bindings at compile time. And second, the Pallene FFI generates code that directly uses C macro definitions, without the FFI itself expanding them. Both cases rely on the C compiler puzzling out later down the pipeline code generated by the FFI. This enabled our FFI to verify the correctness of function bindings and to use C macro definitions, without itself having to understand C.

2 EXTENDING PALLENE WITH C

Guest languages come 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; C calling conventions can be seen as a *lingua franca* of sorts. These are the reasons why we are interested in C for the purpose of interfacing with foreign code [1]. However, this choice comes with issues of its own.

As discussed, there are two goals we wish to achieve with our FFI design, each with their own challenges: *flexibility* and *safety*. These challenges arise from the fact we are interested in targeting C. To understand the challenges we must first outline what we want from an FFI.

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 [6, 7]. 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. Here are some examples of external interactions 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.

Manipulating foreign types directly from the host language is particularly important when dealing with large volumes of data, since the cost of marshaling and unmarshaling becomes prohibitive, and can be avoided when using the same data representation. Consequently, the FFI has to provide some mechanism for data-level interoperability [2, 6, 7].

To safely and fully interact with external libraries we require more than just their binary. The reason is that there is information in the C source code which is erased during compilation. For example, the types of the functions we wish to invoke are erased. Several FFIs rely on programmer-supplied knowledge about types at runtime, such as `libffi` [8] and LuaJIT's FFI extension [15, 16]. However, this is not safe, since wrong type definitions will yield wrong bindings. Additionally, C macro procedures do not exist in compiled objects; there are no symbols to link against since they don't generate functions in the application binary interface (ABI). Consequently, if the only connection between host and guest codes happens through a linker, there's no way an FFI can provide safe and complete interactions with external libraries. We'll discuss in Section 3 how our FFI ensures correct bindings and can interact with macros.

2.1 On Flexibility and Safety

With these requirements in mind, we can outline our goals of *flexibility* and *safety*.

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 (although in Pallene this issue is mitigated by the fact we already have a C compiler at hand; writing a bit of C code is not the end of the world). 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 and Pallene, for example, are safe languages. 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 Pallene with external libraries, we can break its safety: arbitrary foreign code can cause arbitrary crashes. As such, Pallene 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 [1]. On one side we have safe Pallene, 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 Pallene, such as raw pointers, may carry over harmful behavior

[1] A more comprehensive analysis of other FFI design and implementation can be seen on [3].

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.

It is impossible to keep the yellow tape strictly outside Pallene without sacrificing flexibility; our two goals are often at odds and the FFI design has to conciliate between them in some way. This issue manifests itself when dealing with data-level interoperability [2, 6, 7]. In particular, accessing memory that is not owned by the Pallene runtime is a fundamentally unsafe operation. The FFI cannot be fully safe if we allow dereferencing arbitrary pointers, and it cannot be fully flexible otherwise. As discussed, our design balances this 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. 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. 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. 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.

Second, we can design an FFI that *mostly* preserves safety, playing nice on the well-behaved cases, while still offering dangerous features for edge cases as a fallback. 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 can break 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 can provide 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 can clearly mark the dangerous parts of the FFI. All unsafe use of the FFI should be explicitly named as such. Ideally, we'd want the yellow caution tape to be strictly outside Pallene. Since this is not possible, we address 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.

3 LEVERAGING HEADER FILES

In C, the canonical way libraries specify their interface is through C header files. FFIs can generate their bindings using these same header files as a description of external libraries. One issue with this approach, however, is that this description is low level. For instance, a parameter of type `char*` can be a buffer, a null-terminated string, an array of `char`, a pointer to a single value, a generic pointer like `void*`, or not point to valid memory at all. Languages that are not C may benefit from a more precise description than what C provides. Additionally, an FFI that uses header files needs to understand C in some capacity, and extracting information from C header files can be quite complex; it requires implementing parts of a C compiler front-end and a C preprocessor.

Another approach to specify interfaces is through the host language itself. This can be accomplished by adding a static description sub-language inside the host language, which is responsible for describing the API of external libraries. A description sub-language specially crafted for a specific FFI enables richer specifications of external libraries. Additionally, it doesn't incur the complexity of needing to understand C, which makes the implementation much simpler.

However, we run into the issue of correctness. We have no verification that the bindings are correct, since there's no guarantee 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, since we can only interact with things that generate symbols, we completely lose the ability to use macros, which are frequently used in C libraries.

In Pallene we adopted a third alternative to describe interfaces: a hybrid approach that requires both header files and host language declarations. This has the downside of being somewhat redundant; some information is replicated on both descriptions. Nevertheless, equipped with both, the FFI is able to check that bindings are correct, it gets richer descriptions of interfaces, and it allows the use macros. We believe this trade-off is well worth it.

It seems, however, that by using header files we incur the complexity of understanding C. But Pallene already has a dependency that understands C: an entire C compiler! If we get clever, the FFI can use this compiler to indirectly understand C, without itself needing to understand C.

To describe the interface of external libraries in Pallene, we added the concept of a foreign block to its syntax. Listing 1 shows the declaration of the function `sin`, from the `libc` mathematical library. This declaration is checked against the declaration in `math.h`, yielding a compilation error if they don't match. If they match, this function will be available for the rest of the Pallene program.

Listing 1: `numeric.pln`

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

One rule Pallene follows is that the C code it generates cannot fail to compile. Any C compilation errors are a bug in the Pallene implementation. We slightly weakened this rule for this specific use: the FFI generates C code in a way that wrong bindings result

in invalid C, which will raise a C compilation error. The Pallene compiler will output this error to the user, catching them at compile-time. This eliminates an entire class of errors in the FFI, making it safer. We lose nice error messages, but we greatly reduce the complexity of the FFI implementation.

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 directly includes the external library's headers, and is compiled separately and later linked with the main Pallene object. The bridge is where we've weakened Pallene's compilation error rule: it is the only artifact that may fail at the C compilation stage. This failure can only happen due to an incompatible binding declaration at the foreign block, which will be caught by the C compiler if we try to use C definitions incorrectly.

The reason for doing foreign bindings through a bridge is twofold. First, to understand C with this approach, we need to include foreign libraries' header files directly in the Pallene compiler output. If we were to include them on the main Pallene file, we'd pollute it; there would be 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.

As an example, Listing 2 shows the generated bridge for the foreign block in Listing 1. The type of `pln_foreign_function_01` is entirely defined by the programmer in Pallene. The body of this bridging function is trivial: it calls `sin` passing all the arguments through. If the programmer declared the type of `sin` incorrectly in Pallene, (*e.g.* saying it receives a pointer, or that it has two parameters), the code in the bridge would be invalid C: it wouldn't match with the declaration in `math.h`, thus raising a C compilation error. The same approach is used by the FFI for its other features like pointers, strings and arrays.

Listing 2: `numeric_bridge.c`

```c
#include <math.h>

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

It doesn't matter whether `sin` is actually a function or a macro procedure. Since both share the same syntax, the body of the bridging function will invoke the function or macro procedure normally. The linking is done against the bridge, bypassing the issue of macros not generating symbols. A final consideration is that macros aren't typed, but there are certain types of parameters that are incompatible. These incompatibilities are also caught at compile-time.

The main Pallene object has to see these bridging functions in some way. Common use of C delegates this to header files; both library code and application code include the external C declarations contained in the headers, and the linker will later bind their use to their definitions. Since in Pallene the compiler is generating both the main and the bridge C files, we can elide the use of header files, instead manually including the declarations. In Listing 3, we show a snippet of the generated main Pallene C file, with a external declaration of `pln_foreign_function_01`. Note that if there's a mismatch in the description of external libraries, the C compilation error will only happen on the bridge, and never on the main file.

Listing 3: `numeric_main.c`

```c
extern double pln_foreign_function_01(
  double arg_0
);
```

Besides binding against functions, we can also bind against global constants, in such a way that we get the same safety guarantees that we get for functions. Listing 4 shows a foreign block description with the constant `M_PI` contained in `math.h`, and Listing 5 shows the generated bridge. Note that as long as `M_PI` is a compile-time constant, it doesn't matter how exactly it is defined; it can be a macro or a global variable. If the Pallene program supplied a type incompatible with the value of `M_PI`, the C compiler would raise an error when compiling the bridge code. Listing 6 shows a snippet of the code generated for constants in the main Pallene C file.

Listing 4: `constants.pln`

```
foreign
  include <math.h>
  constant M_PI: float
end
```

Listing 5: `constants_bridge.c`

```c
#include <math.h>
const double pln_constant_01 = M_PI;
```

Listing 6: `constants_main.c`

```c
extern const double pln_constant_01;
```

The FFI uses a similar technique for checking the correctness of C `struct` definitions. Listing 7 shows a foreign block with the definition of `git_buf`, a `struct` in the library `libgit2`. (Although some of the details in Listing 7 may seem cryptic, we'll discuss them in more depth later on.) Reiterating, the goal is to allow C compilation failure exclusively on the bridge, if and only if the bindings are incorrect. To this end, we create a transient type, based exclusively on the description in Pallene, which the main Pallene C file also knows about. This transient type isolates the real `struct` from the main Pallene C file, so that this file doesn't need to include foreign declarations. Listing 8 shows the bridging function; it copies the fields from the transient `struct` into the real `struct`, calls the foreign function, and copies them back. Listing 9 shows a snippet of the main Pallene file, and how it sees the transient

Listing 7: `structs.pln`

```
foreign
  include <git2/buffer.h>
  include <git2/deprecated.h>
  type CharPointer = pointer char

  type GitBuf = struct git_buf
    ptr: CharPointer,
    size: size_t,
  end

  function git_buf_grow(
    inout GitBuf,
    size_t
  );
end
```

Listing 8: `structs_bridge.c`

```
#include <git2/buffer.h>
#include <git2/deprecated.h>

struct Pln_utype_01 {
  char* ptr;
  size_t size;
};

int pln_foreign_function_01(
  struct Pln_utype_01* arg_0,
  size_t arg_1
) {
  git_buf arg_0_derived;
  arg_0_derived.ptr = arg_0->ptr;
  arg_0_derived.size = arg_0->size;

  int ret =
    git_buf_grow(&arg_0_derived, arg_1);

  arg_0->ptr = arg_0_derived.ptr;
  arg_0->size = arg_0_derived.size;
  return ret;
}
```

`struct`. In Section 4 we'll discuss how Pallene marshals its values into this transient `struct`.

With this hybrid approach, we avail ourselves of the assistance of a C compiler's front-end to understand C, without depending on a specific C compiler or needing to implement one ourselves.

Listing 9: `structs_main.c`

```
struct Pln_utype_01 {
  char* ptr;
  size_t size;
};

extern int pln_foreign_function_01(
  struct Pln_utype_01*,
  size_t
);
```

4 DESIGN EVALUATION

Since we designed the FFI following an empirical approach, we evaluated it through concrete use cases. These use cases guided our decision-making; we only add features when we observe a concrete case that need them, and we only add unsafeness where we see a library that justifies it. Although we designed the FFI using a larger set of APIs, for this section we selected the more interesting cases that made us add features and unsafeness to the FFI. This section is not a feature list; refer to [3] for a more thorough description.

4.1 `libc`'s mathematical library

Let's begin with the function `sin`, declared in `math.h`, that we showed in Listing 1. We start with an include directive, which tells Pallene the C header file where the declaration of `sin` can be found. As discussed, the header file is used to check whether the bindings are correct in compile time. Then we declare the function `sin`, exposing it to Pallene. This foreign function takes an argument of type C `double` and returns a value of type C `double`.

Note, however, that the type of these values are C types, which do not exist in Pallene. To invoke `sin` from Pallene, we need some way to represent these types inside Pallene. One approach to represent them is by extending the core language. However, to minimize modifications to the core language and offer a more ergonomic interface, the FFI uses a different approach: translation.

The FFI uses Pallene's preexisting numeric types to represent C types, automatically translating when they cross language boundaries. Integer C types like `int` and `long` are mapped into Pallene's `integer` type, and floating point C types like `float` and `double` are mapped into Pallene's `float` type. This approach also eases integration with Lua, since values of these types would need to be representable in Lua in some way. The FFI also uses the translation approach for constants like `M_PI`: inside Pallene they have a Pallene type, not a C type.

Most functions in `math.h` are similar to `sin`: functions over numeric types like `double` and `int`. Additionally, there's an interesting pattern in functions like `frexp` (also from `math.h`) that need to return multiple values to the caller. In C, functions are restricted to returning a single value. A common workaround to this restriction is using references: C functions may pass multiple values back to the caller by receiving pointer arguments and mutating their content. This is a well-behaved use of pointers; the memory they point to belongs to the caller, and the lifetime of the reference is restricted

to the call site. Although general use of pointers is dangerous, the FFI safely supports this use of pointers by specializing them.

Unlike C, Pallene allows functions with multiple return values. We decided to use this feature of Pallene to implement this pattern of C. To indicate the intent of returning multiple values, we introduced parameter modifiers. Listing 10 shows the `frexp` example. Its declaration starts quite similar to `sin`, but it has an extra parameter of type C `int` with a `ret_out` modifier. This modifier alters the meaning of the parameter: it isn't used to pass data to the callee, and is instead used to return data to the caller. As such, the `frexp` function in Pallene takes a `float` argument and returns a `float` and an `integer`. Listing 10 also shows how `frexp` can be used in Pallene after the foreign block.

There's also the `ret_inout` modifier, which is also used to pass values to the callee, and the `in` modifier, which is used only pass values by reference to the callee. In these cases, the function signature in Pallene keeps the parameter.

Listing 10: `numeric_retout.pln`

```
foreign
  include <math.h>

  function frexp(
    cdouble,
    ret_out cint
  ): cdouble
end

local x: float = 1234.5678
local exponent, significand = ffi.frexp(x)
```

4.2 `libc`'s `I/O` library

Let's now have a look at `libc`'s `I/O` library, described at `stdio.h`. We'll discuss features of the FFI like pointers, strings and buffers, which are required by this library.

Amongst other things, `stdio` deals with file streams, which are represented by pointers. Pointers like the ones found in `stdio` are vital for programming in C, and their use in libraries is pervasive. General use of pointers, such as dereferencing and pointer arithmetic, is dangerous. Nonetheless, `stdio` file streams are well behaved: they are handles, pointing to some unspecified underlying value that the library expects in future function calls. They are truly opaque, in the sense that their content is not to be used by application code.

With this in mind, we've created a new Pallene type constructor to represent this kind of C pointer. 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.

Listing 11 shows, amongst other things, how to use this constructor. We start by importing the C header file that contains the type and the function declarations we'll use. Then we use this new type constructor to create a new first-class type with name `File`. This new type represents pointers of C type `FILE*` from `stdio`. It has value semantics just like normal C pointers and can be compared with other pointers of the same type. Listing 11 also shows how we can get a null pointer of this type by reusing the same syntax we've shown for constants. This feature is needed because it's common in C to return null pointers in case of errors, in which case we need to be able to compare pointers with null.

We can now declare foreign functions which take and return values of this newly created type. Listing 11 shows bindings for `fopen` which returns a pointer of this kind, and `fclose` which receives a pointer of this kind. The FFI is able to catch pointer mismatch at compile-time, using the same mechanism outlined in Section 3. For example, it will raise a compilation error if we mistakenly try to pass a pointer of a different type to `fclose`.

Moving on, as we said, `stdio` also uses strings. Strings don't have first-class support in C. Instead, C offers `char` pointers, which have many other uses besides strings. What we are calling strings is a particular use of `char` pointers: a value of type `const char*` that holds a contiguous block of memory terminated by a null character, and whose contents cannot be altered. Unlike C, Pallene has first-class support for strings. These strings, similarly to numeric types, can be safely mapped into `const char*` pointers. Once again, general use of pointers is dangerous, but the FFI can safely support this specific use by specializing them: arguments of C functions with type `const char*` can be represented by Pallene strings. For example, the functions `fopen` and `fputs`, shown in Listing 11, have string arguments. Note that it's unsafe to map `char` pointers back into Pallene strings, since pointers that come from C have no guarantees. Because of this, the FFI does not allow for Pallene strings in the return position of C functions; they have to be represented by pointers, using the type constructor we discussed before.

The last feature we'll discuss in this section are buffers, which `stdio` uses to read from file streams. For example, the function `fgets` takes a pointer to a block of memory and writes to it from a file stream. This use of `char` pointers is unlike the use in `fputs`, since C writes to it. We cannot use Pallene's strings to support this; it's a more general use of C pointers.

One way to use `fgets` is to first create a buffer by calling `malloc`. The same feature that allowed us to represent file streams can be used to represent opaque buffers. Then we may call `fgets`, passing this newly allocated block of memory with the correct size. As discussed, this is highly dangerous; incorrect programs may trigger undefined behaviour like buffer overruns and memory leaks. Once the call to `fgets` returns, we need a way to turn our opaque pointer into a Pallene string. This operation cannot be done safely. To support this, we've added a method to the FFI called `unsafe_to_string`, explicitly marked as dangerous; this is a case where we brought unsafeness into the Pallene world. If we call this method on a buffer like the one returned by `malloc`, it will interpret the buffer's content as a null terminated string, converting it to a Pallene string by copying the pointer's content over. Calling this method on an invalid pointer or a malformed string will result in undefined behaviour. Finally, the programmer has to remember to call `free` on the buffer. This is quite unwieldy.

Fortunately, `fgets` has a characteristic that allows for safer interactions: the buffer is owned by the caller. For these particular cases, there's a different mechanism offered by the FFI: the `CArray`

type. `CArrays` are a new Pallene type we added for data-level interoperability. They are objects with reference semantics traced by the garbage collector, and their contents can be accessed safely without the risk of buffer overruns, just like a normal Pallene array. Unlike a normal Pallene array, `CArrays` can only store C types like numeric C types and pointers; when we access an element of a `CArray`, Pallene translates it into its corresponding Pallene type, using the mapping we discussed before. This way, the FFI can marshal a `CArray` into a C buffer just by copying a pointer and not its contents. Listing 11 shows the bindings for `fgets` using a `CArray`. Note the syntax "`C{...}`"; it's the same syntax for normal Pallene arrays, but prepended with a '`C`'. Unlike the `malloc` approach, using `CArrays` doesn't bring unsafeness to Pallene (although misuse of `fgets` will still cause undefined behaviour).

Note that `CArrays`, similar to Pallene strings, can only safely be arguments of foreign functions; they can't safely be in the return position since there'd no way to validate bare pointers returned from C.

Listing 11: `stdio_bindings.pln`

```
foreign
  include <stdio.h>

  type File = pointer FILE
  constant NullFile: File = NULL
  type CPtr = pointer cchar
  constant NullCPtr: CPtr = NULL

  function fopen(string, string): File
  function fclose(File): cint
  function fputs(string, File): cint
  function fgets(C{cchar}, cint, File): CPtr
end
```

4.3 `zlib`

`zlib` is a data compression library. It's an interesting example for the FFI because it makes extensive use of C `structs` to pass values back and forth. To use C functions that take C `structs`, Pallene must be able to represent them. We'll use an approach similar to the ones we described before: translating already existing Pallene types. More concretely, we'll translate *Pallene tables*. Pallene tables are objects, with reference semantics and garbage collection. They are a record type like C `structs`; the approach is to translate between both whenever they cross language boundaries. Note that this translation happens between Pallene tables and the transient C `struct` described in Section 3. It's this transient `struct` that gets sent to the bridge. Because of this, the foreign function may not hold a reference to the `struct` past the call site.

There are a couple of ways C functions can receive `struct` parameters. The first way is by receiving the `struct` value itself instead of a reference. This has no safety considerations; the FFI only has to copy the Pallene table into the transient `struct`. The second way is by receiving a pointer to the `struct`. Once again, general use of pointers is unsafe. However, because Pallene owns this transient `struct`, we can safely support this particular use by specializing pointers. When translating, the difference between references and values is that with references the FFI also has to copy the fields back to the Pallene table after the C function returns, since the function may have mutated the `struct`.

To indicate which way the C function will receive the `struct`, we added modifiers, somewhat similar to the ones used on numeric types, but not exactly the same because Pallene tables have reference semantics. No modifiers mean the C function receives the `struct` itself, and the `inout` modifier means the C function receives a pointer. In the second case, the FFI will copy the fields from the Pallene table into a `struct`, and then copy the fields from this `struct` back into the Pallene table. As an optimization, we also offer the `in` and `out` modifiers, which excludes one of the copying steps.

Listing 12 shows a foreign `struct` type declaration from `zlib`. Note that this `struct` uses `CArrays` as buffers. Then we declare the `deflate` function that receives a value of this type. The `struct` argument has the `inout` modifier.

Listing 12: `zlib_structs.pln`

```
foreign
  include <zlib.h>

  type ZStreamS = struct z_stream_s
    next_in: C{cuchar},
    avail_in: cuint,
    total_in: culong,
    next_out: C{cuchar},
    avail_out: cuint,
    total_out: culong,
    --[[ ... ]]
  end

  function deflate(inout ZStreamS, cint);
end
```

Just like `CArrays`, Pallene records can only safely be arguments of C functions. Functions that return a pointer to a `struct` have to use the opaque pointers discussed in Subsection 4.2.

4.4 `libgit2`

The `libgit2` is a C implementation of Git provided as a library. This library has a particular use of pointers that is dangerous, which made us add unsafeness to Pallene. As discussed, the FFI can safely call C functions that use pointers owned by Pallene. Additionally, the FFI can provide safe pointers by specializing them into opaque pointers. Although enough for several cases, these two restricted uses of pointers are not sufficient for the general case. For example, there are pointers in `libgit2` that are neither owned by Pallene nor opaque.

In `libgit2`, the `struct git_config_entry` is instantiated by the library and returned as a pointer, which contains data that the

caller needs to access. The function `git_config_get_entry` yields a pointer to this `struct`. This is in contrast with `zlib` where the object is owned by Pallene instead of C, and in contrast with `stdio` where the pointer is opaque.

We added another unsafe feature to the FFI to support this use of pointers in `libgit2`: the `unsafe_deref` method. This feature allows for dereferencing pointers, invoked through Lua's method syntax on values that are pointers. This will dereference the pointer and translate its content using the mappings we've discussed. To use `libgit2` and dereference `git_config_entry*`, we first define the C `struct` like we did in the `zlib` example. Then we create a pointer type to this `struct` like we did in the `stdio` example, using the type constructor we've discussed. After this is all done, Pallene can invoke `unsafe_deref` on this pointer type. When this method is invoked on a pointer to `git_config_entry`, the FFI will create a Pallene table and copy the fields of this `struct` to the Pallene table. If the pointer is invalid, this will cause undefined behavior. Note that we must remember to call the deallocator function for the `git_config_entry` pointer.

4.5 sqlite3

The `sqlite3` library is the C interface for SQLite. It requires three interesting features from the FFI: returning pointers by reference, callbacks, and reading from arrays not owned by Pallene. We'll discuss all three in turn.

Interaction with `sqlite3` start by creating a handle to the database. The initialization function of the database, in contrast with libraries like `stdio` and `liblua`, does not directly return the handle. Instead, the handle is returned by reference like `frexp` returns extra values. Remember that the opaque pointers we added to Pallene are first-class types, with value semantics just like Pallene numbers. Because of this, the FFI can safely support the same modifiers we'd use for numeric types. In the case of getting a handle for `sqlite3`, we mark the pointer parameter with a `ret_out`; it will behave the same as `frexp`.

Moving on, the `sqlite3` library makes use of callbacks, particularly in the function `sqlite3_exec`. This function takes an SQL query and calls the registered callback for each found row. Callbacks are quite tricky to implement because 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 that.

However, there's a very useful subset of callbacks we support: those that include a user-data field. In C parlance, the signature of callbacks of this kind has a parameter of type `void*`. This parameter is a user defined value, opaque to the library. The library forwards this value to the callback when invoking it. The FFI uses 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 supported through this trick; the programmer would need to write C code by hand to address this. For more details on callbacks, see [3].

Finally, there are uses of `sqlite3` that requires Pallene to read from buffers owned by the library. This is very similar to the need for dereferencing pointers we discussed in the previous section; the FFI cannot safely support buffers that are neither opaque nor owned by Pallene. Consequently, we've had to add another unsafe feature to the FFI.

One approach would be extending pointers with some sort of pointer arithmetic, which we could then read 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 reading from buffers that are not owned by Pallene. The FFI represents buffers using normal opaque pointers, and add a method on `CArrays` that takes pointers and copies their content over. Since we know the size of the `CArray`, once we copied the contents of the buffer, any access to the array becomes safe; the memory is sure to be valid and remain so, accesses are bound checked, and it will be freed automatically. After a single dangerous call, we no longer has to worry about unsafeness.

5 CONCLUSIONS

This work presented an FFI for Pallene with safety at the forefront, while also trying to conciliate it with flexibility. To measure flexibility, we studied commonly used C libraries, checking whether they can be directly used through the FFI. To measure safety, we analyzed the ways one can cause arbitrary crashes or cause undefined behaviours from inside Pallene.

The first main contributions of our research is the way we've balanced safety and flexibility. This flexibility/safety dilemma exists because C—unlike Pallene—is an unsafe language. Consequently, there's a trade-off between flexibility and safety that any FFI with C has to contend with. At the core of our decision-making rests an empirical approach, which lead our design from beginning to end. Looking at common APIs, we managed to choose a more informed compromise between flexibility and safety. We also managed to understand how we can specialize certain dangerous features and make them safe, while still providing a general unsafe mechanism as a fallback if needed.

The second main contribution of our research is the way we've leveraged the C compiler to understand C. Using a hybrid interface description approach—plus some clever C code generation—we managed to verify the correctness of bindings and use C preprocessor macros without having to understand C ourselves.

The purpose of this work was to create a mechanism for extending Pallene through an FFI. More specifically, 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. We hope our design and implementation contribute to these goals.

REFERENCES

[1] Eli Barzilay and Dmitry Orlovsky. 2004. Foreign interface for PLT Scheme. In *Proceedings of the Fifth ACM SIGPLAN Workshop on Scheme and Functional*

Programming. 63–74.
[2] Matthias Blume. 2001. No-longer-foreign: Teaching an ML compiler to speak C "natively". *Electronic Notes in Theoretical Computer Science* 59, 1 (2001), 36–52.
[3] Gabriel Coutinho de Paula. 2021. *A Foreign Function Interface For Pallene*. Master's thesis. PUC-Rio.
[4] Gabriel de Quadros Ligneul. 2019. *The Implementation of Records in Pallene*. Master's thesis. PUC-Rio.
[5] Todd Fast, Timothy Wall, and Liang Chen. 2007. Java Native Access (JNA). https://github.com/twall/jna
[6] Kathleen Fisher, Riccardo Pucella, and John Reppy. 2000. Data-Level Interoperability. In *Electronic Notes in Theoretical Computer Science*. 2001.
[7] Kathleen Fisher, Riccardo Pucella, and John Reppy. 2001. A framework for interoperability. *Electronic Notes in Theoretical Computer Science* 59, 1 (2001), 3–19.
[8] Anthony Green. 1996. The libffi home page. https://www.sourceware.org/libffi/
[9] Hugo Musso Gualandi. 2020. *The Pallene Programming Language*. Ph.D. Dissertation. PUC-Rio.
[10] Hugo Musso Gualandi and Roberto Ierusalimschy. 2018. Pallene: A statically typed companion language for Lua. In *Proceedings of the XXII Brazilian Symposium on Programming Languages*. 19–26.
[11] Hugo Musso Gualandi and Roberto Ierusalimschy. 2020. Pallene: A companion language for Lua. *Science of Computer Programming* 189 (2020), 102393.
[12] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. 2011. Passing a language through the eye of a needle. *Commun. ACM* 54, 7 (2011), 38–43.
[13] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. 1996. Lua—an extensible extension language. *Software: Practice and Experience* 26, 6 (1996), 635–652.
[14] J.K. Ousterhout. 1998. Scripting: higher level programming for the 21st Century. *Computer* 31, 3 (1998), 23–30. https://doi.org/10.1109/2.660187
[15] Mike Pall. 2005. LuaJIT, a just-in-time compiler for Lua. http://luajit.org/luajit.html
[16] Mike Pall. 2005. LuaJIT FFI extension. https://luajit.org/ext_ffi.html
[17] Jeremy Yallop, David Sheets, and Anil Madhavapeddy. 2014. ctypes. https://github.com/ocamllabs/ocaml-ctypes
[18] Jeremy Yallop, David Sheets, and Anil Madhavapeddy. 2016. Declarative foreign function binding through generic programming. In *International Symposium on Functional and Logic Programming*. Springer, 198–214.
[19] Jeremy Yallop, David Sheets, and Anil Madhavapeddy. 2018. A modular foreign function interface. *Science of Computer Programming* 164 (2018), 82–97.