A modular foreign function interface

Jeremy Yallop, David Sheets and Anil Madhavapeddy
University of Cambridge Computer Laboratory
first.last@cl.cam.ac.uk

Abstract

Foreign function interfaces (FFIs) between high-level languages and system libraries typically intertwine the actions of *describing* the interface of a system library and *selecting* a binding strategy for linking to it. This tight coupling makes it difficult for programmers to switch between different binding strategies, and discourages the development of new approaches to binding, since more exotic approaches are unlikely to attract sufficient users to justify the cost of development.

We present Cmeleon, a replacement for the standard OCaml FFI that exposes typed constructors that correspond to the operations of the type algebra of C, and binding strategies that interpret this type structure as separate program stages. Cmeleon parameterises external calls across binding strategies, isolating interface descriptions from choices relating to call construction (code generation vs dynamic call frames), concurrency style (blocking, cooperatively or preemptively threaded), and separation (in-process, address space or a network connection).

This flexibility enables significant code reuse of bindings in many different contexts, from rapid interactive development in a REPL to production deployments with generated code and privilege separation. Cmeleon has been used for the past two years to bind to a broad variety of real-world OCaml libraries, and entirely supplants the need for the low-level C FFI for the vast majority of applications.

Categories and Subject Descriptors D3.2 [*Language Classifications*]: Applicative (functional) languages

Keywords functional programming, foreign function interfaces, staged programming

1. Introduction

Practical implementations of high-level languages must support interoperability with low-level code, and there is a multitude of approaches available even for the seemingly simple task of gluing together a single pair of languages. This diversity is largely driven by the many contexts in which these high-level programming languages may be used – in a Unix or Windows system with shared libraries, as statically linked and cross-compiled embedded systems, for interactive prototyping in IDEs, or even phone applications.

Every mainstream programming language exposes a foreign function interface (FFI) to C that permits calls from the language to the underlying system. Unfortunately, safe use of these FFIs requires the programmer to carefully respect many invariants across invocations, or else risk silent memory corruption. FFIs often feature hundreds of API calls and usage rules that make this difficult to get right manually [17]. A static analysis of Python bindings revealed over 150 errors in a representative set of modules [18], with similar results for OCaml [12] and Java [16].

For instance, consider the relatively simple case of calling the `gettimeofday(3)` library function from OCaml to retrieve the time. An implementation using the OCaml FFI follows:

```
#include <caml/mlvalues.h>
#include <sys/time.h>

CAMLprim value
caml_gettimeofday(value u)
{
  CAMLparam1(u);
  CAMLlocal1(res);
  struct timeval tv;
  if (gettimeofday(&tv, NULL) == 0)
    unix_uerror("gettimeofday");
  res = Val_long(tv.tv_sec);
  CAMLreturn(res);
}
```

This snippet reveals numerous FFI calls that take care of converting values between OCaml and C value representations (`Val_long`) or ensure that the garbage collector (GC) does not move OCaml values around during the execution of the foreign call (`CAMLparam1`, `CAMLlocal1`, `CAMLreturn`). This FFI style – while prevalent in popular languages – should be discouraged except for experts.

Dynamic binding Recognising this, a common alternative approach available in many language implementations such as Python, Ruby, OpenJDK, and the Glasgow Haskell Compiler is to describe the C functions from within the high-level language and use the `libffi` library to call foreign functions at runtime by constructing stack frames for the library ABI dynamically. Here is a typical example, which uses Python's `ctypes` library to bind and call the `gettimeofday` function:

```
libc = ctypes.CDLL("libc.dylib", use_errno=True)
tv = struct_tv ()
libc.gettimeofday(ctypes.pointer(tv), None)
```

The dynamic approach is especially convenient for interactive development, but the difficulty of determining the types of C functions at runtime and the lack of access to compile-time features such as enum constants and macro definitions make it less suitable for use in production systems. There is also a steep performance penalty versus writing C bindings since call frames have to be dynamically constructed (§5).

A programmer who wishes to call a C library from their high-level language must currently weigh up the benefits of each approach and commit to one system. A shift in requirements later – for example, relinquishing interactivity for performance – typically requires a rewrite of the bindings.

Abstracting binding strategies This paper presents Cmeleon, a replacement for the standard OCaml FFI that separates the activity of describing foreign types and functions from the decision of which binding approach to use. Here is a binding to `gettimeofday` using Cmeleon:

```
module Bindings(F : FOREIGN) = struct
 open F
 let gettimeofday = foreign "gettimeofday"
   (ptr timeval @→ ptr timezone @→ returning int)
end
```

We call the reader's attention to two salient features of this code, leaving the details for later in the paper. First, the binding to `gettimeofday` is described using high-level functions for constructing representations of C function types (`@→` and `returning`) and for resolving external names at particular types (`foreign`). Second, the binding is parameterised by the *interpretation* of these functions, i.e. by the module `F` of type `FOREIGN`. This separation between description and interpretation is key to our approach, and will allow us to reuse this single foreign interface description with a wide variety of binding strategies, such as static stub generation, dynamic call construction, and several more exotic strategies including inverted (C-to-OCaml) and cross-process calls.

The design of Cmeleon The remainder of this paper presents the design of Cmeleon, which comprehensively glues together OCaml and C code, using OCaml's module system to abstract over the details of how exactly the gluing takes place. Cmeleon decomposes foreign function bindings into reusable constituent parts that can be assembled in a variety of ways to balance interactivity, performance and flexibility without the need to rewrite the code that describes a particular foreign library.

Cmeleon supports the complete spectrum of C types and is structured as: *(i)* a common library for describing type structure, with constructors corresponding to the various operations in the type algebra of the foreign language; *(ii)* various ways to interpret type structure as program stages. Cmeleon provides numerous binding strategies that can interpret the type structure, for:

- choosing between interactive development in a REPL (§3.2), and subsequently statically evaluating them into stub code optimised for deployment (§3.4)
- interfacing OCaml code to C function calls, or inverting the FFI to permit OCaml libraries to easily expose a C ABI using the same core type definitions (§4.1)
- selecting concurrency models for function calls to foreign libraries, such as cooperatively batching requests to one library and launching preemptive threads for those that do not support asynchronous interfaces (§4.2)
- enforcing separate address spaces between foreign libraries and the language runtime for privilege separation [23], or using unconventional linking strategies such as unikernels [19] or hardware isolation features [2] (§4.3)

The paper is structured as follows. We use inductive data types to represent the types of a foreign language in the host language, starting with object types (§2) and moving on to functions (§3). We then explain how advanced interpretations such as asynchronous FFIs and an inverted FFI from OCaml to C operate in the same modular framework (§4).

```
struct timeval {
   unsigned long tv_sec;
   unsigned long tv_usec;
};
```

Figure 1: The timeval struct in C

```
module TvTypes(T: TYPE) = struct
  open T
  let timeval = structure "timeval"
  let sec = field timeval "tv_sec" ulong
  let usec = field timeval "tv_usec" ulong
  let () = seal timeval
end
```

Figure 2: The timeval struct in Cmeleon

Describing foreign bindings within a high level language by programming against a typed abstract interface that can be instantiated with different binding strategies offers flexibility, extensibility and type safety that are not possible with an external tool. The ability to move seamlessly and safely between (e.g.) dynamic, staged and out-of-process bindings – and even reuse the same bindings description for inverted calls – without rewriting the binding description has been invaluable in our own uses of Cmeleon in a variety of OCaml projects, and has seen rapid adoption in the wider OCaml community in a number of real-world commercial and free software projects (§5). We conclude by discussing the prior influences on our work (§6) and the broader implications of our approach (§7).

2. Describing C types and values

The main purpose of Cmeleon is binding and calling C functions. However, function types are built from value types (called "object types" in C), and calling C functions involves passing and retrieving C values, so we first describe how Cmeleon represents C values and how it determines their layout.

2.1 Describing C types

We start by describing the representation of C types, which appear as first-class values in Cmeleon.

The binding to `gettimeofday` in the introduction involves a type `timeval`. Figures 1 and 2 show the C definition of `timeval` and the corresponding Cmeleon definition. The binding to the `gettimeofday` function was parameterised by the definitions of the function-binding operations `@→`, `returning` and `foreign`. Similarly, the definition of `timeval` in Cmeleon is parameterised by the definitions of the type-building operations `structure`, `field` and `seal`. We shall see shortly how this parameterisation supports different strategies for determining object layout, just as the parameterisation in the `gettimeofday` binding supports different strategies for binding functions.

Excluding the parameterisation, the C and Cmeleon definitions correspond line for line. The first line of the Cmeleon code creates a value representing the type `timeval`; the second and third lines add `unsigned long` fields with the names `tv_sec` and `tv_usec`, and the final line "seals" the structure, turning it from an incomplete type into a fully-fledged object type with known properties such as size and alignment.

Figure 3 shows the `TYPE` interface with which we have parameterised the definition of `timeval`. The first member of the signature is a parameterised type, `ty`. A value of type `t ty` represents a C type which manifests as the type `t` in OCaml. Besides `ty` and the type-building operations already named there are operations for representing the primitive types `void`, `char` and `int`, an operation

```
module type TYPE = sig
 type α ty
 val void: unit ty
 val char: char ty
 val int: int ty   (* ...etc *)
 val ptr: α ty → α ptr ty
 val structure: string → σ structure ty
 val seal: σ structure ty → unit
 val field: σ structure ty → string → α ty → (α, σ)
    field
 val view: read:(α → β) → write:(β → α) → α ty →
    β ty
end
```

Figure 3: A signature for constructing C object types

`ptr` for constructing pointer types, and a additional function `view` which acts as a kind of map over type representations. In each case the parameter of the result type `ty` indicates the OCaml type used to read and write values of the underlying C type. For example, values of the C type `char` appear in OCaml as values of the `char` type, so the `char` operation has type `char ty`. Similarly, C values of type `void **` appear in OCaml as values of type `(unit ptr) ptr`, and so building the corresponding type representation by applying `ptr` twice to `void` produces something of type `((unit ptr) ptr) ty`.

The full open-source implementation also supports the other C primitives and types – arrays, unions, and additional arithmetic types – but we omit the details for brevity. We defer discussion of function pointers to §3.3.

A universal view of types The `ty` constructor may be viewed as a type of *codes* for C type representations, and the operations of the `TYPE` interface as code constructors corresponding to each element of the C type algebra. Codes are inductively defined: just as the C type constructor for pointers builds types from types, our `ptr` builds type representations from type representations. Viewed this way, our approach is reminiscent of the idea of a *universe* [3, 22] from the dependently-typed programming community, where codes are used to delineate some subset of types of interest – in this case those OCaml types which represent C object types. In a dependently-typed language the codes of a universe come equipped with an interpretation function which maps codes to types, but since OCaml does not support type-level functions we instead index codes by the result of the interpretation – a trick well-known to the generic programming community [8, 28].

2.2 C types, concretely

We have shown how parameterising by the `TYPE` signature gives us access to the operations we need to construct type representations. In order for us to use those representations to build functions and access C values we need a more concrete representation of types.

We now describe a concrete implementation of codes, which will enable us to define functions which work on all values of a C object type. Our concrete representation uses Generalised Algebraic Data Types (GADTs) [9] to precisely capture the relationship between the representation of C types and the OCaml types we use to access C objects. The types of the operations in the `TYPE` interface of Figure 3 ensures that those operations are *used* correctly; the constraints represented by GADTs give us additional confidence that they are also *implemented* correctly.

We define our C type representation as one of a mutually-recursive group of four definitions, for representing C types, pointer values, type isomorphisms and structure values. Values of the `ctype` type represent the C types `void`, `char`, or `int`, pointers to C types, structure types, or type isomorphisms called *views*, which allow us to give alternative interpretations to a particular representation. For example, we might view a `char *` as either an OCaml `string` or as a byte buffer; the underlying C type is the same, but we access objects of the type in different ways.

```
type _ ctype =
    Void    : unit ctype
  | Char    : char ctype
  | Int     : int ctype
  | Pointer : α ctype → α ptr ctype
  | Struct  : struct_type → α structure ctype
  | View    : (α, β) view → α ctype
```

A `ptr` value stores a typed C pointer object. The `reftyp`, `addr` and `managed` fields respectively store the type of the pointed-to object, the raw C address, and (optionally) an OCaml object to which we can attach finalisers for releasing resources managed by Cmeleon.

```
and α ptr =
  { reftyp  : α ctype;
    addr : address;
    managed : Obj.t option; }
```

The `address` type is an alias for the OCaml type `nativeint` that denotes an integer suitably sized for representing machine addresses:

```
type address = nativeint
```

A `view` has two fields containing functions for converting back and forth between the external type and the underlying representation, plus a third field to hold the viewed type.

```
and (α, β) view =
  { read : β → α; write : α → β; ty: β ctype }
```

All structure values managed by Cmeleon are heap-allocated and represented directly as pointers.

```
and α structure =
  { structure : α structure ptr }
```

The `structure` type is parameterised by a type that is instantiated differently for each separate structure type; it distinguishes incompatible structures in a similar fashion to a struct tag in C. Instantiating the parameter appropriately is left to the user, and is typically accomplished by an ascription, as in the following example:

```
# let t : [‘t] structure typ = structure "t";;
val t : [ ‘t ] structure typ = struct t
```

There are two further types associated with structures. The first type, `struct_type`, holds information associated with a C struct type: its *tag* (e.g. `timeval`), a flag indicating whether it is complete or incomplete and, if it is complete, its size and alignment requirements. Structs in C can be initially declared as incomplete and completed later, as captured by the mutable fields in our OCaml representation of struct types:

```
type struct_type =
  { tag: string;
    mutable complete: bool;
    mutable size: int;
    mutable align: int; }
```

The second type, `field` holds the type, name and offset associated with a struct field:

```
type (α, σ) field =
 { ftype: α ctype; fname: string; foffset: int }
```

The two type parameters of `field` represent the type of the field and the type of the enclosing structure type. The second type parameter is phantom – it does not appear in the definition. Only the type of the `field` operation in the `TYPE` interface (Figure 3) ensures that a field is associated with the structure type used to create it.

Operations on types Cmeleon provides a number of operations on C types, some of which can be defined in pure OCaml. For example, assuming we have information about the storage requirements of primitive types, it is straightforward to define functions that determine the size and alignment of arbitrary types, or that pretty-print types and values.

```
val sizeof : α ctype → int
val alignment : α ctype → int
val string_of_ctype : α ctype → string
val string_of : α ctype → α → string
```

Here are `sizeof`, `alignment`, `string_of_typ` and `string_of` in action at the OCaml top level:

```
# sizeof int
- : int = 4
# alignment (ptr int)
- : int = 8
# string_of_typ (ptr (ptr int));;
- : string = "int**"
# string_of (ptr int) (allocate int 10);;
- : string = "0x1732cd0"
```

Some other Cmeleon functions on `ctype` involve new primitives. For example, `allocate` is a typed analogue to `malloc`, that allocates and initialises a value of a specified type:

```
val allocate : α ctype → α → α ptr
```

The implementation of `allocate` uses a low-level primitive, `raw_allocate`, which returns an untyped buffer. The memory associated with a value of the `managed_buffer` type is freed automatically when the value becomes unreachable from OCaml code:

```
type managed_buffer
val raw_allocate : int → managed_buffer
```

Operations on values Besides these (and other) operations on types, Cmeleon provides a number of operations on C values. Cmeleon's interface supports accessing memory at a full range of C types, but all memory access are implemented in terms of a small number of low-level operations and the low-level types `address` and `managed_buffer`.

The `!@` and `<-@` functions support reading and writing pointers:

```
val (!@) : α ptr → α
val (<-@) : α ptr → α → unit
```

The `reftyp` member of a `ptr` value is crucial to the implementation of these functions. Accessing a struct in memory is typically a different operation than accessing a `char`, and inspecting the representation of the pointed-to type makes it possible to determine both how many bytes to read (or write) and how to interpret the result (or input) as an OCaml value. For views, `!@` and `<-@` perform the appropriate translation on the OCaml value before passing it to (or after retrieving it from) C. Here is an example, where we define a view type which is represented as an `int` in C, but appears in OCaml as a `bool`:

```
# let bool = view int
   ˜read:(fun i → i <> 0)
   ˜write:(fun b → if b then 1 else 0);;
val bool : bool typ = int
# let p = allocate bool false;;
val p : bool ptr = (int*) 0x1521a10
# !@p;;
- : bool = false
# p <-@ true;;
- : unit = ()
# !@p;;
- : bool = true
```

The `getf` and `setf` serve a similar function for structure values:

```
val getf: σ structure → (α, σ) field → α
val setf: σ structure → (α, σ) field → α → unit
```

If we have a value representing a `struct timeval` then we can use `getf` and `setf` along with the field values `tv_sec` and `tv_usec` to read and write its fields:

```
# setf tv usec (ULong.of_int 10);;
- : unit = ()
# tv;;
- : [ ‘tv ] structure = { tv_sec = 0, tv_usec = 10  }
```

2.3 Determining structure layout

We have presented an abstract interface for building C type descriptions (§2.1) and a concrete representation of C types and values (§2.2). What do we gain from separating the abstract interface from the concrete representation rather than programming directly with the latter?

In fact, the mapping from type descriptions to type representations is not entirely trivial. The key difficulty is determining the layout of structure fields, which involves several considerations. First, the C standard allows compilers to insert padding bytes between structure fields in order to improve performance. Second, the types and members of structs in C APIs sometimes vary across platforms and between library versions. Third, many compilers can be configured to use alternative algorithms for struct layout: GCC's `__attribute__(packed)`, which requests that structs be laid out compactly in memory, is a typical example. It is, of course, crucial for a program that accesses C structs to have a view of their layout that matches the actual layout used by the C library.

In Cmeleon structs are described using the operations of the `TYPE` interface. There are several implementations of `TYPE`, each of which interprets the operations in the interface as functions which determine memory layout details for structs, and then builds a concrete type representation.

2.3.1 Computing structure layout

As we have said, the C standard allows implementations to insert padding when laying out struct members. In practice this typically means that each field begins on an alignment boundary for the field type, that the end of the struct is padded up to the next alignment boundary, and that the alignment for the struct is taken to be the most stringent alignment requirement of the fields.

Our first implementation of the `TYPE` signature implements the operations which build struct types as functions which follow these rules. The `next_offset` function computes the next alignment boundary:

```
let next_offset offset alignment =
  match offset mod alignment with
    0 → offset
  | overhang → offset - overhang + alignment
```

The `structure` function builds an incomplete empty struct with no alignment requirements:

```
let structure tag =
  { tag; size = 0; align = 0; complete = false }
```

The `field` function computes the alignment and offset for the field and updates the struct alignment and size:

```
let field structured fname ftype =
 match structured with
 | Struct { complete = false } as  spec →
   let falign = alignment ftype in
   let foffset = next_offset spec.size falign in
   let field = { ftype; foffset; fname } in
   spec.size <- foffset + sizeof ftype;
   spec.align <- max falign spec.align;
   field
 | Struct {tag} → raise (ModifyingSealedType tag)
 | _ → raise (Unsupported "non-struct")
```

Finally, the `seal` function inserts end padding and marks the struct as complete:

```
let seal = function
 | Struct { size = 0 } →
   raise (Unsupported "struct with no fields")
 | Struct { complete = true; tag } →
   raise (ModifyingSealedType tag)
 | Struct spec →
   spec.size <- next_offset spec.size spec.align;
   spec.complete <- true
 | _ → raise (Unsupported "sealing non-struct")
```

These functions are defined in a module `Computed_structs` which implements the `TYPE` interface. Applying `TvTypes` to the `Computed_structs` module gives us a concrete type representations for `timeval`:

```
# include TvTypes(Computed_structs);;
  val timeval : [‘tv] structure typ =
    struct timeval { unsigned long tv_sec;
                     unsigned long tv_usec;  }
  val sec : (Unsigned.ulong, [‘tv] structure) field =
    {ftype = unsigned long; foffset = 0;
     fname = "tv_sec"}
  val usec : (Unsigned.ulong, [‘tv] structure) field =
    {ftype = unsigned long; foffset = 8;
     fname = "tv_usec"}
# sizeof timeval;;
- : int = 16
# offsetof usec;;
- : int = 8
```

Limitations of computing layout Computing structure layout in this way works for simple cases, but has a number of limitations that make it unsuitable to be the sole approach to laying out data. For bindings to libraries that declare structs whose fields vary from platform to platform, or that specify non-standard layout requirements (e.g. with the `__packed__` direction), attempting to replicate the way that the C compiler lays out structs quickly becomes unmanageable.

2.3.2 Retrieving structure layout

We can avoid the drawbacks of `Computed_structs` with an alternative implementation of `TYPE`. Instead of attempting to replicate the C compiler's structure layout algorithm we will go directly to the source and retrieve layout data from the C compiler itself.

Bringing layout information from the C compiler into our OCaml program involves several stages (Figure 4). First, we provide an implementation of the `TYPE` signature that generates C code each time one of the structure-building operations is called. Second, we run the generated C code to produce an OCaml module which also satisfies the `TYPE` signature. Finally, we link this generated module into the program by passing it as argument to `TvTypes`.

Let's follow the process through step by step. The first step involves applying the `TvTypes` functor to an implementation of the `TYPE` signature that prints out a C function call for each call to `structure`, `field` or `seal`. For example, the following calls to `field`

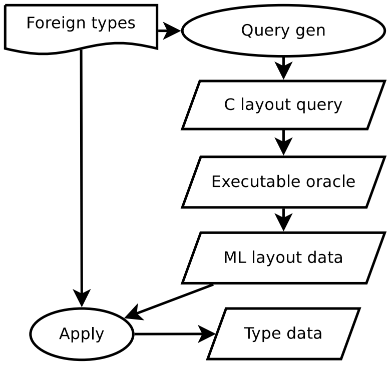


Figure 4: Struct layout detection: a functor that constructs C type representations is applied to a query generator to generate C code. The C code compiles to an executable oracle that produces an ML module containing layout data. The original functor is applied to this layout module to link the type data into the program.

```
field timeval "tv_sec" ulong;
field timeval "tv_usec" ulong;
```

generate the following C:

```
printf("{ftype; fname; foffset = %zu}\n",
        offsetof(struct timeval, tv_sec));
printf("{ftype; fname; foffset = %zu}\n",
        offsetof(struct timeval, tv_usec));
```

Compiling and running the complete generated C program generates an OCaml module which matches the `TYPE` signature, and whose implementations of `structure`, `field` and `seal` return the information obtained by the C program. Here is the generated implementation of `field`, which contains the text generated by the C fragment above:

```
let field s fname ftype = match s, fname with
  | Struct { tag = "timeval"}, "tv_sec" →
    {ftype; fname; foffset = 4}
  | Struct { tag = "timeval"}, "tv_usec" →
    {ftype; fname; foffset = 4}
  ...
```

Applying the `TvTypes` functor to this generated module links the retrieved information directly into the program to build struct representations that are guaranteed to conform to the layout used by the C compiler, even if the order, alignment or number of fields in the OCaml description of `timeval` differ from the details declared in the C library.

Retrieving the layout information from a generated C program rather than attempting to compute it ourselves ensures that the layout used in the program matches the layout used by the C compiler. The full Cmeleon library uses the same approach to offer additional operations for retrieving other static data, such as the values of enum constants or macros.

An alternative approach to retrieving structure layout The workflow shown in Figure 4 is not suitable for every situation. In particular, when cross-compiling it may be impractical to run generated C code in the execution environment during the build process. We plan to support an alternative workflow in which the generated C code is linked directly into the program rather than executed to produce an ML module in order to support this case.

3. Describing functions

We now turn to the question of binding foreign functions, where a broadly similar approach allows us to separate binding descriptions

```
module type FUNCTION = sig
  type _ fn
  val returning : α ctype → α fn
  val ( @→ ) : α ctype → β fn → (α → β) fn
end

module type FOREIGN = sig
 include FUNCTION
 type _ res
 val foreign: string → (α → β) fn → (α → β) res
end

module Bindings(F : FOREIGN) = struct
 open F
 let gettimeofday = foreign "gettimeofday"
  (ptr timeval @→ ptr timezone @→ returning int)
end
```

Figure 5: Module types for C function types and foreign bindings, followed by a foreign binding that uses them.

from binding strategies. We first introduce codes for function types (§3.1), as we did for object types, and once again index the codes by their interpretation into the OCaml type space. Abstracting over the object type language allowed us to apply different strategies for determining object layout; a similar approach allows us to interpret our function descriptions in a variety of situations, starting with an interpreter for foreign calls (§3.2), which we extend to cover the situation of calling back into OCaml from C (§3.3). We then stage our interpreter to produce a heterogeneous code generator (§3.4) and use functors and GADTs to support linking the generated code into our program without compromising type safety. Finally, we bolster our claim to generality by exhibiting a number of alternative interpretations. Starting from the same binding descriptions we interpret the codes to obtain an exporter that turns a host language module into a foreign language library (§4.1), asynchronous foreign function bindings (§4.2), and bindings with greater safety by running in a separate address space (§4.3).

3.1 Coding C function types

Figure 5 shows an abstract interface `FUNCTION` for building function types. There are three components: values of the type `fn` represent C function types, which are constructed using `returning`, which specifies the return type, and `@→`, which prepends an argument to an existing function type.

In keeping with the prevailing style in OCaml, we use currying to represent C functions of multiple arguments. However, `returning` and `@→` carefully distinguish object types and function types; a C function that takes one argument and returns a function pointer that accepts another argument is quite different from a function of two arguments, and our coding represents them differently.

It will be useful to have a concrete representation that implements `FUNCTION`. Translating the types of `returning` and `@→` into constructor types gives us an inductive data type `cfn` which can be used to implement the abstract interface `FUNCTION`:

```
type _ cfn =
  Returning : α ctype → α cfn
| Function : α ctype → β cfn → (α → β) cfn
```

For object types the question of representation arose twice: we need to represent both the types and the values of C. For functions the situation is simpler. The only operation we need to perform on a C function is invocation, so we can represent C functions directly as OCaml functions.

3.2 Interpreting calls

Now that we can represent C function types the next step is to add an interpretation function for binding to C functions. Figure 5 shows the `FOREIGN` interface which we used in the introduction to build the binding to `gettimeofday`. The `FOREIGN` signature extends `FUNCTION` with an operation `foreign` for constructing a C function binding from a name and a representation of its type. The return type of `foreign` uses the abstract parameterised type `res` (short for `result`); we are initially interested in situations where α `fn` becomes α `cfn` and α `res` becomes α, so the type of `foreign` is:

```
val foreign : string → (α → β) cfn → (α → β)
```

That is, `foreign` turns a C function type description and a name into an OCaml function.

In order to build a function of this type we will implement `foreign` as an interpreter that resolves names and synthesises call frames dynamically. Dynamic name resolution is implemented by the POSIX function `dlsym`. Call frame synthesis uses the `libffi` library to handle the low-level details, and we build a typed interface on top of its primitive operations.

Call synthesis using the `libffi` library involves two basic basic types. The first, `ffi_type`, represents C types; we introduce a corresponding OCaml type and expose inhabitants for various primitive types:

```
type ffitype
val int_ffitype : ffitype
val char_ffitype : ffitype
val pointer_ffitype : ffitype
```

The second `libffi` type, `ffi_cif`, describes a call frame structure. We again introduce a corresponding OCaml type `callspec` and expose primitives for creating a new `callspec`, for adding arguments to the `callspec`, and for "sealing" the callspec to mark it as completed and specify the return type:

```
type callspec
val alloc_callspec : unit → callspec
val add_argument : callspec → ffitype → int
val prepare_call : callspec → ffitype → unit
```

(The return type of `add_argument` represents an offset which can be used for writing each arguments into the appropriate place in a buffer when performing a call.)

Finally, we need an operation for actually invoking functions. The `call` function takes a function address, a completed `callspec`, and two callbacks which write arguments and read the return value from buffers.

```
val call : address → callspec →
  (address → unit) → (address → α) → α
```

Building a typed interface to these `libffi` primitives – that is, using them to implement `foreign` – is straightforward. Each call to `foreign` uses `alloc_callspec` to create a fresh `callspec` ; each argument in the function representation results in a call to `add_argument` with the appropriate `ffitype` value. The `Returning` constructor results in a call to `prepare_call`; when the arguments of the function are supplied the `call` function is called to invoke the resolved C function. There is no compilation stage: the user can call `foreign` interactively (Figure 8a). Here is a simple example, using the `isdigit` function, which returns non-zero when the argument represents a digit character:

```
# let isdigit =
    foreign "isdigit" (int @→ returning int);;
val isdigit : int → int = <fun>
# isdigit (Char.code '3');;
- : int = 2048
# isdigit (Char.code 'x');;
- : int = 0
```

```
typedef int (*compar_t)(void *, void *);
int qsort(void *, size_t, size_t, compar_t)
```

Figure 6: The `qsort` function

```
let compar_t =
 funptr (ptr void @→ ptr void @→ returning int)

module Bindings(F : FOREIGN) = struct
 open F
 let qsort = foreign "qsort"
   (ptr void @→ size_t @→ size_t @→ compar_t @→
    returning void)
end
```

Figure 7: Using `funptr` to bind to `qsort`

3.3 Interpreting callbacks from C to OCaml

The interpreter of §3.2 turns a function name and a function type description into a callable function in two stages: first, it resolves the name into a C function address; next, it builds a call frame from the address and the function type description. In circumstances where we have an address rather than a name available for the function this second stage is useful independently, and so Cmeleon supports it as a separate operation:

```
val function_of_pointer :
  (α → β) cfn → unit ptr → (α → β)
```

Conversions in the other direction are also useful: to pass an OCaml function to C, we must convert it to an address:

```
val pointer_of_function :
  (α → β) cfn → (α → β) → unit ptr
```

The implementation of `pointer_of_function` is based on the `callspec` interface that we used to build the call interpreter. We need one just extra primitive operation, which accepts a `callspec` and an OCaml function, then uses `libffi` to dynamically construct and return a "trampoline" function which calls back into OCaml:

```
val make_function_pointer : callspec → (α → β) →
    address
```

Rather than expose the conversions between functions and pointers directly to the user, we build a view that converts between addresses and pointers automatically:

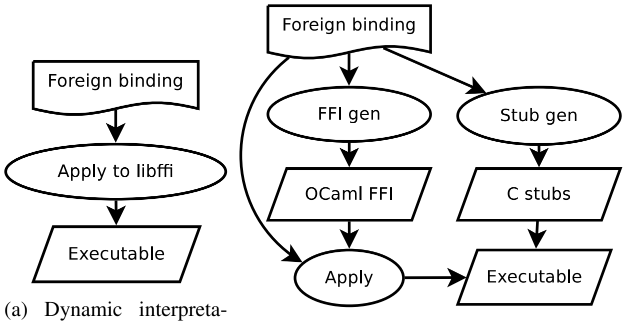
```
let funptr fn =
  view (ptr void)
  ~read:(function_of_pointer fn)
  ~write:(pointer_of_function fn)
val funptr : (α → β) cfn → (α → β) ctype
```

`funptr` builds object type representations from function type representations, just as function pointers build object types from function types in C. Figure 7 shows `funptr` in action, describing the callback function for `qsort` (Figure 6). We can pass OCaml functions to the resulting `qsort` binding directly:

```
qsort arr nmemb sz
  (fun l r → compare (from_voidp int !@l)
                     (from_voidp int !@r))
```

(The `from_voidp` function converts from a `void *` value to another object pointer type.)

This scheme naturally supports even higher-order functions: function pointers which accept function pointer as arguments, and so on, allowing callbacks into OCaml to call back into C. However, such situations appear rare in practice.



(a) Dynamic interpretation of a binding

(b) Static staging of a binding

Figure 8: Basic binding interpretations

3.4 Staging the call interpreter

Interpreting function type descriptions as calls is convenient for interactive development, but has a number of drawbacks. First, the implementation suffers from significant interpretative overhead, which we quantify in §5. Second, there is no check that the values we pass between OCaml and C have appropriate types. Our implementation resolves symbols to function addresses at runtime, so there is no checking of calls against the declared types of the functions that are invoked. Finally, we cannot make use of the many conveniences provided by the C language and typical toolchains. When compiling a function call a C compiler performs various promotions and conversions, which are not available in our simple reimplementation of the call logic. By sidestepping the usual symbol resolution process we also lose the ability to use tools like `nm` and `objdump` to determine how functions are used.

The second of these problems is reminiscent of the difficulties with the function that computes structure layout (§2.3.1), which also suggests the cure. Instead of basing our implementation of `foreign` on an *interpretation* of the type provided by the user we will use the type description to *generate* both C code which can be checked against the API and OCaml code which we will link into the program. The details of the workflow are a little different for binding functions than for retrieving details about object layout: we are dealing with link-time function addresses rather than compile-time struct offsets, so we cannot inline the results into the program.

These differences aside, the broad pattern is similar. We first instantiate the `Bindings` functor (Figure 5) with implementations of `FOREIGN` that generate code, then link the code into the program with a further instantiation of `Bindings` (Figure 8b). Let us trace through the details of the staging. The `Bindings` functor in Figure 5 contains a binding to the `gettimeofday` function. The first instantiations of `Bindings` generate C and OCaml code.

The generated C code (the `gettimeofday_C` implementation) converts OCaml representations of values to C representations, calls `gettimeofday` and translates the return value representation back from C to OCaml[1]. If the user-specified type of `gettimeofday` is incompatible with the type declared in the C API then the C compiler will complain when building the generated source.

```
value cmeleon_gettimeofday(value a, value b)
{
   struct timeval  *c = ADDR_OF_PTR(a);
   struct timezone *d = ADDR_OF_PTR(b);
   int e = gettimeofday(c, d);
   return Val_int(e);
}
```

[1] There are no calls to protect local variables from the GC because Cmeleon was able to statically determine that the GC cannot run during the execution of this function.

The generated OCaml module matches the FOREIGN signature. The central feature is a generated foreign function which scrutinises the type representation passed as argument and extracts raw addresses to pass to C:

```
let foreign : type a. string → a cfn → a =
 fun name t →
  match name, t with
 | "gettimeofday",
    Function (Pointer _, Function (Pointer _, Returning
     Int)) →
  (fun x1 x2 → gettimeofday_C x1.addr x2.addr)
```

Readers familiar with GADTs will recall the type refinement that takes place during the pattern match. Although the result type a is initially abstract, matching on the type representation reveals information about the type, so that the right-hand side of the first case is expected to be a function of type σ ptr → τ ptr → int for some types σ and τ.

More precisely, the generated OCaml module has type:

```
FOREIGN with type α fn = α
```

and so passing it as argument to the Bindings functor builds a module containing a callable gettimeofday function.

4. Advanced Interpretations

We now briefly consider several more exotic interpretations of FOREIGN. We start with an inversion of the model to support exporting a C ABI from an OCaml interface description (§4.1), then describe how to support an cooperative asynchronous monadic interface (§4.2), and how to separate the address spaces of the OCaml and C runtime behind a multi-process interface (§4.3).

4.1 An inversion: exporting C ABIs from OCaml code

Now that we've seen how to invert the call interpreter to support callbacks (§3.3) and how to stage the call interpreter to improve safety and speed (§3.4), the question naturally arises: Is there a use for an inverted, staged interpreter? It turns out that there is.

The main use of Cmeleon is making C libraries available to OCaml programs. However, as the discoveries of disastrous bugs in widely-used C libraries continue to accumulate, the need for safer implementations of those libraries written in high-level languages such as OCaml becomes increasingly pressing. As we shall see, Cmeleon supports exposing OCaml code to C via an interpretation of FOREIGN that interprets the parameter of the res type as a value to consume rather than a value to produce.

Specialising the res type of the FOREIGN signature (Fig 5) with a type that consumes α values gives the following type for foreign:

```
val foreign : string → (α → β) fn →
   ((α → β) → unit)
```

that is, a function which takes a name and a function description and consumes a function. This is just what we need in order to turn the tables: rather than a function which resolves and binds foreign functions, we now have a function which exports functions under specified names.

Continuing our running example, suppose that we want to export a function whose interface matches gettimeofday. Just as before, we can reuse the binding from Figure 5, but this time we will instantiate result to produce a function exporter. As with the structure layout retriever (§2.3.2) and the staged call interpreter (§3.4) we will apply the functor multiple times – first to generate a C header and a corresponding implementation which forwards calls to OCaml callbacks, and then to produce an exporter which connects the C implementation with our OCaml functions.

We saw in §2.2 that Cmeleon includes a pretty-printer that formats C type representations using the C declaration syntax. Applying the pretty-printer to the gettimeofday binding produces a declaration suitable for a header:

```
int
gettimeofday(struct timeval *, struct timezone *);
```

The generation of the corresponding C implementation proceeds similarly to the staged call interpreter, except that the roles of OCaml and C are reversed: the generated code converts arguments from C to OCaml representations, calls back into OCaml and converts the result back into a C value before returning it. The addresses of the OCaml functions exposed to C are stored in an array in the generated C code. The size of the array is determined by the number of calls to foreign in the functor – one, in this case.

Back on the OCaml side we generate code to populate the array when the OCaml module is loaded, and index it by an enumeration data type callback whose type parameter specifies the types of the functions that we will store:

```
type _ callback = Gettimeofday :
  (address → address → int) callback

val register_callback : α callback → α → unit
```

The generated foreign function pattern matches on the type to produce a function consumer, which passes the consumed function to register_callback:

```
let foreign name t : type a. string → a cfn → (a →
     unit) =
 match name, t with
 |"gettimeofday", Function (Pointer x2,
    Function (Pointer x4, Returning Int)) →
   (fun f → register_callback Gettimeofday
    (fun x1 x2 →
      f {reftyp=timeval; addr=x1; managed=None}
        {reftyp=timezone; addr=x2; managed=None}))
```

Applying the Bindings module to this generated module results in a gettimeofday function which can be used to register an OCaml function to be called via the generated C interface:

```
val gettimeofday : (tv structure ptr →
  tz structure ptr → int) → unit
```

4.2 Asynchronous calls

Abstracting over the binding strategy allows us to reuse the same library binding descriptions with a variety of strategies. We've used the FOREIGN signature to abstract over interpreted and staged approaches, and to vary whether we're exposing C functions to OCaml or OCaml functions to C. However, there are still more possibilities to explore.

Since the standard OCaml runtime has limited support for concurrency, many modern OCaml programs make use of cooperative concurrency libraries such as Lwt [27]. Cooperative concurrency requires taking care with potentially blocking calls, since a single blocking call can cause suspension of all threads. To help mitigate the problem, Lwt supports a primitive

```
val detach : (α → β) → α → β Lwt.t
```

which associates a potentially blocking computations with one of a pool of system threads. It is easy to see why we might want to wrap detach around all calls to certain foreign functions. As the signature of detach indicates, Lwt has a monadic interface: potentially blocking computations run in the Lwt.t monad. In order to support marking foreign calls as blocking we must adjust the FOREIGN signature to turn foreign calls into monadic computations:

```
module type FOREIGN = sig
  type α comp
```

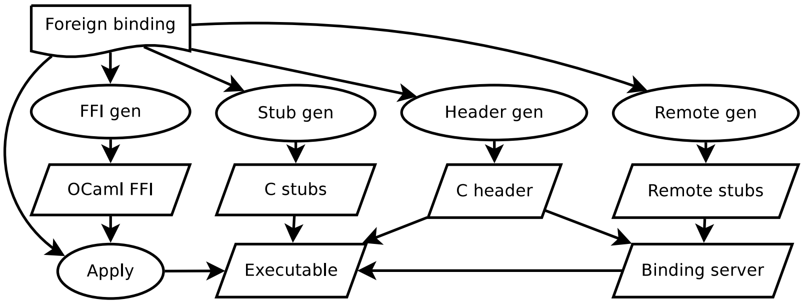


Figure 9: Static interprocess staging of a binding; a foreign binding functor is applied to an FFI generator which produces an OCaml FFI module which itself can be applied to the foreign binding and combined with generated C stubs (which satisfy a generated C header) to produce an executable. This executable communicates with a generated binding server process over a protocol enforced by the generated C header.

```
  val returning : α ctype → α comp fn
  (* otherwise the same as FOREIGN *)
end
```

The original `FOREIGN` signature (Figure 5) can be recovered from this one by substituting `type` α `comp =` α. The implementation is a variation on the scheme described in §3.4: we insert a call to `detach` around each foreign call, and insert code to release OCaml's runtime lock in the generated C.

With the new `FOREIGN` signature in place, we can abstract over the monad which foreign function calls inhabit. Applying `Bindings` to our Lwt-specialised implementation of `FOREIGN`, where α `comp` is instantiated with α `Lwt.t`, gives a binding to `gettimeofday` implementation that runs in the Lwt monad:

```
val gettimeofday :
 tv structure ptr → tz structure ptr → int Lwt.t
```

4.3 Out-of-process calls

High-level languages often make strong guarantees about type safety that are compromised by binding to foreign functions. Languages like Python, Haskell, OCaml and Java preclude memory corruption by isolating the programmer from the low-level details of memory access; however, a single call to a misbehaving C function can result in corruption of arbitrary parts of the program memory.

One way to protect the calling program from the corrupting influence of a C library is to allow the latter no access to the program's address space. Cmeleon supports this approach using a variant of the staged call interpreter (§3.4); instead of invoking bound C functions directly, the generated stubs marshall the arguments into a shared memory buffer where they are retrieved by an entirely separate process which contains the C library.

Once again, this cross-process approach is straightforward to build from existing components. Our data representation is based on C structs: for each foreign function Cmeleon outputs a struct with fields for function identifier, arguments and return value. The struct is built using the type representation constructors introduced earlier (§2.1) and printed using the generic Cmeleon pretty printer.

```
struct gettimeofday_frame {
  enum function_id id;
  struct timeval *tp;
  struct timezone *tz;
  int return_value;
};
```

These structs are then read and written by the generated C code in the two processes. Figure 9 shows the generation of components: besides the C and ML code generated for the staged interpreter, the cross-process interpretation also generates C code that runs in the remote process and a header file to ensure that the two communicants have a consistent view of the frame structs.

Perhaps unsurprisingly, these cross-process calls involve some overhead, which we quantify in §5; nevertheless, having the option is very useful in circumstances where it is essential to prevent memory corruption [13].

5. Evaluation

We evaluate Cmeleon both quantitatively via benchmarks of the various backends (§5.1) as well as qualitatively through our experiences with using it over the last two years both for ourselves (§5.2) and through open-source (§5.3).

5.1 Call Latency

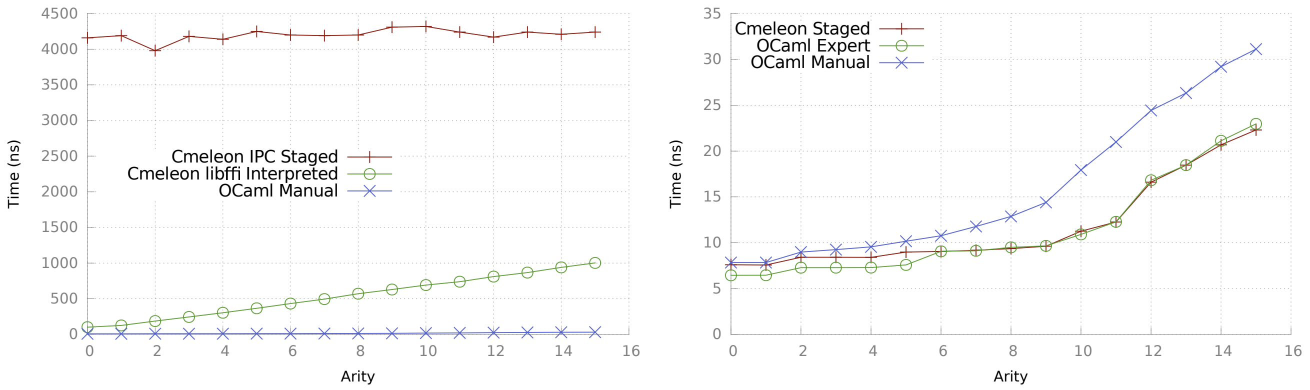
To evaluate the overhead of Cmeleon, we wrote bindings for ten simple machine integer functions of arity 0 to 9 which return their last argument. Then, we interpreted these bindings both dynamically with `libffi` (Figure 10a) and statically through a staged compilation (Figure 10b). We wrote two other modules satisfying the same signature with implementations using the traditional manual OCaml binding technique of manipulating OCaml values in C with preprocessor macros. The *manual* variation followed exactly the FFI directions in Chapter 19 of the OCaml 4.02.1 manual. The *expert* variation took advantage of various omissions, shortcuts, and undocumented annotations which preserve memory management invariants and are known to be safe but difficult to use correctly.

The `libffi`-interpreted bindings have a large overhead due to writing an ABI-compliant stack frame. Type traversal and directed frame construction for the bound symbol results in a call latency linear in the function's arity. The static bindings are between 10 and 65 times faster than the dynamic bindings. Figure 10a also shows bindings staged to perform interprocess communication (IPC) via semaphores and shared memory in order to isolate the bound library's heap from the main program (§4.3). As expected, the IPC introduces a call latency of several microseconds.

Each test except staged IPC generation ran for 10s on an Intel Core i7-3520M CPU running at 2.9 GHz under Linux 3.14-2 x86_64. Staged IPC generation ran for 45s per test case to collect sufficient samples for a narrow distribution. All tests had a coefficient of determination, R^2, in excess of 0.98 and 95% confidence intervals of less than ±2%.

5.2 Binding Development

Before developing Cmeleon, we found writing OCaml foreign function bindings tedious and punctuated by the frustration and confusion of subtly violated representation invariants [25]. From



(a) Bindings staged for heap isolation and interpreted with libffi

(b) Bindings staged and written manually

Figure 10: Mean FFI call latency by arity

Interface	Topic	Interpretations
libsodium	cryptography	staged
libmacaroons	cryptography	dynamic
OpenSSL	cryptography	staged
nocrypto	cryptography	staged
micro-ecc	cryptography	dynamic
GNU SASL	authentication	staged
glibc passwd	identity	dynamic
GDAL/OGR	geography	dynamic
libmemphis	geography	dynamic
libzbar	barcodes	dynamic
libnl	networking	dynamic
ZeroMQ	messaging	dynamic
nanomsg	messaging	staged
LZ4	compression	staged
OpenGL (ES)	graphics	dynamic
SDL	multimedia	dynamic
unistd.h	POSIX	dynamic
sys/stat.h	POSIX	dynamic
fcntl.h	POSIX	dynamic
dirent.h	POSIX	dynamic
FUSE protocol	file systems	data type
Tokyo Cabinet	database	dynamic
lmdb	database	dynamic
libuv	async I/O	staged

Table 1: Some bindings using Cmeleon

remembering when the GC runs to ensuring that values are appropriately boxed and unboxed to registering GC roots and callbacks, writing and maintaining a binding used to be a lot of work. Today, we regularly use Cmeleon for experiment and recreation through the REPL and convert our experiments into dynamic and staged interpretations for publication.

Where it once took most of a week to correctly bind a 20 function interface, we can now do it in hours. Cmeleon eliminates entire classes of memory representation and GC registration bugs through the encoding of OCaml's runtime invariants in its various interpretations. Table 1 lists some well-known Cmeleon projects and many more are currently in development.

5.3 Community Uses

We released the first version of Cmeleon as an open-source library in June 2013. Since then, the modular style has led to various unanticipated combinations of binding interpretations being used by the wider OCaml community as well as our own projects.

Security-critical bindings Recently, SSL library bindings have garnered considerable interest in the systems community. OCaml developers have used manual bindings to OpenSSL from the Liquidsoap project for many years. The ocaml-ssl library was subsequently wrapped in the Lwt cooperative threading monad to be used asynchronously. The binding and the asynchronous wrapper have both been subject to ongoing issues in language runtime handling arising from the manually written C FFI bindings. Recently, the async_ssl library has used Cmeleon to quickly and correctly bind to OpenSSL and directly map that binding into the Async cooperative threading monad. This library is currently in use commercially at Jane Street Capital. One of the key motivations behind the development of the out-of-process interpretation in Cmeleon (§4.3) has been due to a lack of confidence in OpenSSL's memory safety, which in turn compromises OCaml code that calls into it.

Function call interposition Of the five commercial Cmeleon users, Cryptosense SA is likely the most demanding in their combination of interpretations. Cryptosense uses staged bindings, both inverted and forward, with dynamic callbacks to interpose tracing for the PKCS#11 C API for testing the safety of applications that use hardware security modules (HSMs), smartcards, or other cryptography providers. By using Cmeleon, Cryptosense writes their Cryptosense App Tracer product in type-safe OCaml while operating in a C linking environment [6]. High-level, type-safe code can now be used to build very low-level function call interposition that would otherwise be error-prone and difficult to debug.

Binary protocol implementation The profuse FUSE protocol library uses Cmeleon solely for its ability to represent the types of binary protocols and perform C structure layout queries. A previous library, ocamlfuse, used manual bindings to libfuse, the FUSE library for userspace file systems. Profuse improves on ocamlfuse by directly communicating with the OS kernel via a UNIX domain socket. This gives profuse the flexibility to stack FUSE file systems and manage asynchrony without incurring the overhead of the full parsing of messages and libfuse-managed asynchrony. This use of Cmeleon's type representation and layout query features is only possible due to the modular embedding of the C type system. A DSL-based generator would be much harder to repurpose.

Unikernel compilation Unikernels are a technique to compile specialised applications that run directly on a hypervisor instead of requiring an intervening guest operating system [19, 20]. The

MirageOS unikernel system is written in OCaml, and Cmeleon is used to provide a safe mechanism to link and cross-compile C code into a single-address space Xen virtual machine image. For example, the nocrypto C library provides the cryptographic primitives used by the clean-slate TLS stack that is otherwise written in OCaml. The type safety of a Xen unikernel critically depends on the C trusted computing base being bug-free, and Cmeleon eliminates the need for a significant amount of manually written C FFI code that translates between OCaml and C representations. The bindings currently use the staged C stub generation, but because they are parameterised over interpretations, it is also easily possible to add support for inter-virtual-machine function calls in a similar fashion to inter-process calls within Unix.

6. Influences and related work

We have noted various related work during the exposition. Here we list some additional work which has directly influenced the design of Cmeleon.

The decision to represent foreign types as first-class values in Cmeleon was inspired by several existing FFIs, including Python's ctypes, Common Lisp's Common FFI and Standard ML's NLFFI [5], each of which also takes this approach.

Cmeleon follows NLFFI's approach in indexing representations of C types and values by host language types in order to ensure internal consistency (although OCaml's GADTs, unavailable to the author of NLFFI, make it possible to avoid most of the unsafe aspects of the implementation of that library). However, Cmeleon departs from these libraries in abstracting the declaration of C types from the mechanism used to retrieve information about those types, using OCaml's higher-order module system to perform the abstraction and subsequent selection.

Central to Cmeleon is the use of functors to abstract over interpretations of the `TYPE` and `FOREIGN` signatures. Carette et al [7] use functors in a similar way, first abstracting over the interpretation of an embedded object language (lambda calculus), then developing a variety of increasingly exotic interpretations which perform partial evaluation, CPS translation and staging of terms.

We suggested (§2.1) an analogy between our `ty` type together with its constructors and the use of universes in the dependently-typed programming community. Altenkirch and McBride [1] use universes directly to represent the types of one programming language (Haskell) within another (OLEG) and then to implement generic functions over the corresponding values.

As we have observed (§2.1), mapping codes to types and their interpretations by abstracting over a parameterised type constructor is a well-known technique in the (non-dependently-typed) generic programming community. Hinze [14] describes a library for generic programming in Haskell with a type class that corresponds quite closely to the `TYPE` signature of §2, except that the types described are Haskell's, not the types of a foreign language. There is a close connection between Haskell's type classes and ML's modules, and so Karvonen's implementation of Hinze's approach in ML [15] corresponds even more directly to this aspect of Cmeleon's design.

7. Discussion and Conclusions

The unification of staging with the OCaml foreign function interface has been remarkably successful, with many formerly unstable library bindings now simplified and more reliable and flexible when ported to Cmeleon. The internal complexity of the implementation was well-protected by OCaml's type system (notably GADTs), and the use of OCaml's functors to encode program stages scaled extremely well. We have found the higher-order and first-class aspects of OCaml's module system particularly valuable; although we have not shown the actual applications of the various interpretations, a typical application involves passing a functor containing bindings to a Cmeleon function as a first-class module (package) [11].

Although the representation of C types as first class values has been used in previous work (e.g. [5]), the organisation of binding strategies into a cohesive system of staged functors is novel in Cmeleon, and we hope to see it built into other high-level languages in the future by using the abstraction facilities available there.

While OCaml's advanced module system has proved invaluable in the design and implementation of Cmeleon, it is likely that the essential elements of the library can be replicated without too much difficulty in language with support for higher-kinded polymorphism, such as Haskell and Scala, or in untyped languages such as Python and Ruby.

A little further afield, Java's Project Panama is a proposed replacement to the much maligned Java JNI, and could use many of the binding strategies described in this paper. One possible approach is to directly port Cmeleon to Java via the OCamlJava [10] backend. Conversely, many of the software fault isolation strategies proposed in the literature for improving the JNI could also be implemented as Cmeleon stages [26] to improve the performance of our address space separation. Wedge also offers primitives for privilege separation that could be provided by Cmeleon [4], with the additional benefit of not requiring further porting of the bindings.

Cmeleon bindings built by users also benefit from the entire range of binding strategies that we have implemented, most notably the ability to hold suspect foreign libraries in a separate address space. Cmeleon guides binding authors to be explicit about memory ownership for this reason, and we plan to extend the typing of the multiprocess interface to effectively expose a capability system with (what amounts to) typed process identifiers. If the user does not require the multiprocess model, then the staging optimises away any overheads. Industrial users of Cmeleon have commented that it entirely supplants the need for them to write manual C bindings, even for high-performance use cases such as cryptography or financial trading strategies, and our experimental results in this paper confirm this.

Cmeleon bindings are written at a fairly high level of abstraction. However, there is still sufficient overhead involved in writing out all the definitions necessary for binding to a large API that automating the construction of bindings descriptions is an attractive prospect. We have experimented with using the CIL C parser [21] to import C header files directly into Cmeleon and with interrogating DWARF debug information to extract types from compiled objects. Making these work robustly and expose clean OCaml interfaces is the topic of future work, perhaps based on similar work in this space [24].

Cmeleon is building up momentum in the open-source community, and has been ported beyond Linux to OpenBSD, FreeBSD, MacOS X, Windows and the Android and iPhone mobile phone environments. The existing binding strategies are being extended into more exotic environments such as remoting library calls across virtual machine boundaries for use with unikernels. Type definitions are being written over the base C types to cover language runtimes such as Python, leading to the prospect of safe, well-typed FFIs directly between two host languages without writing a single line of C code.

Acknowledgements We thank our colleagues Leo White, Thomas Gazagnaire, Stephen Kell, Mark Florisson, Stephen Dolan and Alan Mycroft for valuable feedback on this paper. Jeremie Dimino, Mark Shinwell and Yaron Minsky (Jane Street), Thomas Braibant and Graham Steel (Cryptosense), Dave Scott, Rob Hoes and Mike McClurg (Citrix), Török Edwin (Skylable), Peter Zotov, David Kaloper, Hannes Mehnert and Daniel Bünzli contributed code and design feedback by adopting the library in their projects. A. Hauptmann and Thomas Leonard ported the library to Windows and Xen.

A full list of contributors and the Cmeleon source code are available at https://github.com/ocamllabs/ocaml-ctypes. The research leading to these results received funding from the European Union's Seventh Framework Programme FP7/2007–2013 under the User Centric Networking project (grant agreement no. 611001) and supported by Horizon Digital Economy Research, RCUK grant EP/G065802/1.

References

[1] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 1–20, Deventer, The Netherlands, The Netherlands, 2003. Kluwer, B.V. ISBN 1-4020-7374-7. URL http://dl.acm.org/citation.cfm?id=647100.717294.

[2] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, 2012. USENIX. ISBN 978-1-931971-96-6.

[3] M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. 10(4):265–289, 2003. ISSN 1236-6064.

[4] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association. ISBN 111-999-5555-22-1.

[5] M. Blume. No-longer-foreign: Teaching an ML compiler to speak c natively. *Electronic Notes in Theoretical Computer Science*, 59(1):36 – 52, 2001. ISSN 1571-0661. . BABEL'01, First International Workshop on Multi-Language Infrastructure and Interoperability 2001).

[6] T. Braibant, J. Protzenko, and G. Scherer. Well-typed generic smart-fuzzing for APIs. In *10th ML Family Workshop, ICFP 2014*, 2014. URL https://sites.google.com/site/mlworkshoppe/fuzzing.pdf.

[7] J. Carette, O. Kiselyov, and C.-c. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, Sept. 2009. ISSN 0956-7968. . URL http://dx.doi.org/10.1017/S0956796809007205.

[8] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 90–104, New York, NY, USA, 2002. ACM. ISBN 1-58113-605-6. .

[9] J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[10] X. Clerc. OCaml-Java: OCaml on the JVM. In H.-W. Loidl and R. Pea, editors, *Trends in Functional Programming*, volume 7829 of *Lecture Notes in Computer Science*, pages 167–181. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40446-7. .

[11] A. Frisch and J. Garrigue. First-class modules and composable signatures in Objective Caml 3.12. ACM SIGPLAN Workshop on ML, September 2010. Baltimore, Maryland, United States.

[12] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 62–72, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. .

[13] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *ACM Conference on Computer and Communications Security*, pages 38–49, 2012.

[14] R. Hinze. Generics for the masses. *J. Funct. Program.*, 16(4-5):451–483, July 2006. ISSN 0956-7968. . URL http://dx.doi.org/10.1017/S0956796806006022.

[15] V. A. Karvonen. Generics for the working ml'er. In *Proceedings of the 2007 Workshop on Workshop on ML*, ML '07, pages 71–82, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-676-9. . URL http://doi.acm.org/10.1145/1292535.1292547.

[16] G. Kondoh and T. Onodera. Finding bugs in Java Native Interface programs. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 109–118, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-050-0. .

[17] B. Lee, B. Wiedermann, M. Hirzel, R. Grimm, and K. S. McKinley. Jinn: Synthesizing dynamic bug detectors for foreign language interfaces. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 36–49, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. .

[18] S. Li and G. Tan. Finding reference-counting errors in Python/C programs with affine analysis. In R. Jones, editor, *ECOOP 2014 Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 80–104. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-44201-2. .

[19] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. .

[20] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. Leslie. Jitsu: Just-in-time summoning of unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA, 2015. USENIX Association.

[21] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43369-4.

[22] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf type theory: an introduction*. Clarendon, 1990.

[23] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[24] T. Ravitch, S. Jackson, E. Aderhold, and B. Liblit. Automatic generation of library bindings using static analysis. In M. Hind and A. Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 352–362. ACM, 2009. ISBN 978-1-60558-392-1. .

[25] D. Scott, R. Sharp, T. Gazagnaire, and A. Madhavapeddy. Using functional programming within an industrial product group: perspectives and perceptions. In *ACM Sigplan Notices*, volume 45, pages 87–92. ACM, 2010.

[26] M. Sun, G. Tan, J. Siefers, B. Zeng, and G. Morrisett. Bringing Java's wild native world under control. *ACM Trans. Inf. Syst. Secur.*, 16(3): 9:1–9:28, Dec. 2013. ISSN 1094-9224. . URL http://doi.acm.org/10.1145/2535505.

[27] J. Vouillon. Lwt: A cooperative thread library. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 3–12, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-062-3. .

[28] Z. Yang. Encoding types in ML-like languages. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 289–300, New York, NY, USA, 1998. ACM. ISBN 1-58113-024-4. .