A modular foreign function interface

Jeremy Yallop, David Sheets and Anil Madhavapeddy

Docker, Inc, and
University of Cambridge Computer Laboratory

Abstract

Foreign function interfaces are typically organised monolithically, tying together the specification of each foreign function with the mechanism used to make the function available in the host language. This leads to inflexible systems, where switching from one binding mechanism to another (say from dynamic binding to static code generation) often requires changing tools and rewriting large portions of code.

We show that ML-style module systems support exactly the kind of abstraction needed to separate these two aspects of a foreign function binding, leading to declarative foreign function bindings that support switching between a wide variety of binding mechanisms — static and dynamic, synchronous and asynchronous, etc. — with no changes to the function specifications.

Note. This is a revised and expanded version of an earlier paper, Declarative Foreign Function Binding Through Generic Programming (Yallop et al., 2016). This paper brings a greater focus on modularity, and adds new sections on error handling, and on the practicality of the approach we describe.

Keywords: foreign functions, functional programming, modularity

1. Introduction

The need to bind and call functions written in another language arises frequently in programming. For example, an OCaml programmer might call the C function puts to display a string to standard output\(^1\):

\(^1\) For the sake of exposition the example is simple, but it captures the issues that arise when writing more realistic bindings.
Before calling `puts`, the programmer must write a binding that exposes the C function as an OCaml function. Writing bindings presents many opportunities to introduce subtle errors (Furr and Foster, 2005; Kondoh and Onodera, 2008; Li and Tan, 2014), although it is a conceptually straightforward task: the programmer must convert the argument of the bound function from an OCaml value to a C value, pass it to `puts`, and convert the result back to an OCaml value.

In fact, bindings for functions such as `puts` can be produced mechanically from their type definitions, and tools that can generate bindings, such as SWIG (Beazley, 1996), are widely available. However, using an external tool — i.e. operating on rather than in the language — can be damaging to program cohesiveness, since there is no connection between the types used within the tool and the types of the resulting code, and since tools introduce types and values into a program that are not apparent in its source code.

This paper advocates a different approach, in which foreign functions such as `puts` are described using the values and types of the host language. More concretely, each C type constructor (int, *, char, and so on) becomes a value in OCaml, and each value that describes a function type can be interpreted to bind a function of that type. For example, here is a binding to the `puts` function, constructed from its name and a value representing its type:

```ocaml
let puts = foreign "puts" (str → returning int)
```

(Later sections expound this example in greater detail.)

Describing foreign language types using host language values results in a much closer integration between the two languages than using external tools. For example, the interface to SWIG is a C++ executable that generates OCaml code, and there is no connection between the C++ types used in the implementation of SWIG and the types of the generated OCaml code. In contrast, the type of the `foreign` function (which is expounded in detail in Section 2.2) is directly tied to the type of the OCaml function that `foreign` returns, since calls to `foreign` are part of the same program as the resulting foreign function bindings.

This improved integration has motivated implementations in a number of languages, including Common Lisp\(^2\), Python\(^3\) and Standard ML (Blume,

---

\(^2\)CFFI [https://common-lisp.net/project/cffi/manual/index.html](https://common-lisp.net/project/cffi/manual/index.html)

\(^3\)Ctypes [https://docs.python.org/2/library/ctypes.html](https://docs.python.org/2/library/ctypes.html)
However, although these existing designs enjoy improved integration, they do not significantly improve flexibility, since in each case the mechanism used to bind foreign functions is fixed. For example, Python’s `ctypes` module binds C functions using the `libffi` library\(^4\), which constructs C calls entirely dynamically. The programmer who would like more performance or safety than `libffi` can offer can no longer use `ctypes`, since there is no way to change the binding mechanism.

This paper describes a design that extends the types-as-values approach using modular abstraction to support multiple binding mechanisms, including (i) a dynamic approach, backed by `libffi`, (ii) a static approach based on code generation, (iii) an inverted approach, which exposes host language functions to C, and several more interpretations, including (iv) bindings that handle `errno`, and (v) bindings with special support for concurrency or cross-process calls. The key is using parameterised modules to abstract the definition of a group of bindings from the interpretation of those bindings, making it possible to supply various interpretations at a later stage. Each binding mechanism (i.e. each interpretation) is then made available as a module implementing three functions: the \( \rightarrow \) and \( \text{returning} \) functions, which construct representations of types, and the `foreign` function, which turns type representations into bindings.

For concreteness this paper focuses on a slightly simplified variant of `ocaml-ctypes` (abbreviated `ctypes`), a widely-used library for calling C functions from OCaml that implements our design. As we shall see, the OCaml module system, with its support for abstracting over groups of bindings, and for higher-kindled polymorphism, provides an ideal setting.

1.1. Outline

This paper presents the `ctypes` library as a series of interpretations for a simple binding description, introduced in Section 2. Each interpretation is presented as an implementation of the same signature, `FOREIGN`, which exposes operations for describing C function types. We gradually refine `FOREIGN` throughout the paper as new requirements becomes apparent.

Section 3 introduces the simplest implementation of `FOREIGN`, an interpreter which resolves names and builds calls to foreign functions dynamically.

Section 4 describes a second implementation of `FOREIGN` that generates

---

\(^4\) `libffi` [https://sourceware.org/libffi/](https://sourceware.org/libffi/)
OCaml and C code, improving performance and static type checking of foreign function bindings.

Section 5 shows how support for higher-order functions in foreign bindings extends straightforwardly to supporting inverted bindings, using the FOREIGN signature to expose OCaml functions to C.

Section 6 describes some additional interpretations of FOREIGN that support error handling and concurrency.

Section 7 explores a second application of the multiple-interpretation approach, using an abstract signature TYPE to describe C object layout, and giving static and dynamic interpretations of the signature.

Section 8 presents evidence for the practicality of the ctypes approach, touching on adoption, performance and some brief case studies.

Finally, Section 9 contextualizes our work in the existing literature.

2. Representing types

C types are divided into three kinds: object types describe the layout of values in memory, function types describe the arguments and return values of functions, and incomplete types give partial information about objects. Bindings descriptions, as for puts in the introduction, involve representations of both object types, such as int, and function types, such as int(const char *).

2.1. Representing object types

Figure 1 gives a signature for the abstract type typ, which represents C object types, including a number of constructors (int, str, ptr, and so on) which build OCaml values that represent particular C types. The complete definition of typ includes constructors for other primitive types, and for arrays, unions and structs, which are omitted for brevity here. Structs are considered later, in Section 7.

The parameter of each typ value tracks the OCaml type that correspond to the C type it describes. For example, the value str, which describes the C type char *, has the following type:

val str : string typ

reflecting the fact that C values described by the str value have the OCaml type string.

Figure 2 gives an implementation of typ and its constructors as a generalized algebraic data type, or GADT (Cheney and Hinze, 2003) — that is,
Figure 1: C object type descriptions

```
type α typ
val void : unit typ
val str : string typ
val int : int typ
val ptr : α typ → α ptr typ
```

Figure 2: An implementation of Figure 1

```
type α typ =
| Void : unit typ
| Int : int typ
| Char : char typ
| Ptr : α typ → α ptr typ
| View : (β → α) * (α → β) * β typ → α typ
and α ptr = address * α typ

val string_of_ptr : char ptr → string
val ptr_of_string : string → char ptr

let void = Void and int = Int and char = Char
let ptr t = Ptr t
let str = View (string_of_ptr, ptr_of_string, Ptr Char)
```

as a datatype whose parameters may vary in the return type of each constructor. The signatures of the constructors closely match the signatures in Figure 1 except that str is built using a constructor View, which builds a new type representation from an existing representation and a pair of functions that convert between the corresponding types. For str, the type representation is built from Ptr Char and from a pair of functions string_of_ptr and ptr_of_string that convert between the C and OCaml representations of strings.

### 2.2. Representing function types

Binding C functions also requires a representation of C function types. Here is the binding from Section 1 again, which we will use throughout the paper to illustrate our different interpretations.
module type FOREIGN = sig
  type α fn
  val (@ : α typ → β fn → (α → β) fn)
  val returning : α typ → α fn

  val foreign : string → α fn → α
end

Figure 3: The FOREIGN signature for describing foreign functions (version 1)

module Puts(F: FOREIGN) = struct
  open F
  let puts = foreign "puts" (str @ → returning int)
end

Besides the object type representations already considered, three OCaml functions are used to define puts: foreign, @→ and returning. In order to support different implementations of these functions we have placed the definition of puts within a functor, parameterised by a module F that will supply their implementations when the functor is applied.

Figure 3 shows the FOREIGN signature. There is a parameterised type fn, for representations of C function types, two functions @→ and returning, for building values of type fn, and a function foreign that accepts a name and a C function type representation and returns an OCaml value.

As with typ, the parameters of fn track the OCaml types that correspond to the C types they describe. The types of @→ and returning combine the types of their arguments, building up the types of OCaml functions that correspond to the types of the C functions they will expose, so that the value describing puts, written str @→ returning int, has the OCaml type string → int.

3. Interpreting bindings

Our first implementation of FOREIGN is a simple interpreter, Foreign_dyn, that turns bindings description into callable OCaml functions. For example, here is the effect of passing Foreign_dyn to the binding description for puts at the OCaml top level:

# include Puts(Foreign_dyn);;
val puts : string → int = <fun>
module Foreign_dyn = struct
  type _ fn =
    Fn : α typ * β fn → (α → β) fn
  | Returns : α typ → α fn

  let (@→) s t = Fn (s, t) and returning t = Returns t
  let foreign = foreign_dyn
end

Figure 4: The Foreign_dyn implementation of FOREIGN (Figure 3)

val dlopen : string → dl_flag list → library
val dlsym : library:library → string → address

Figure 5: The dlopen and dlsym functions

# puts "Hello, C!";;
Hello, C!
- : int = 10

Figure 4 gives the implementation of Foreign_dyn. There are two parts: a
definition of the type fn as a datatype whose two constructors Fn and Returns
 correspond directly to @→ and returning, and a definition of the foreign
function as foreign_dyn, which we now proceed to define.

The purpose of foreign is to turn the name of a C function and a de-
scription of its type into a callable OCaml function. The two parameters
of foreign correspond to the two actions necessary to complete this task:
resolving the name and interpreting the type description.

The foreign_dyn implementation of the foreign function dynamically re-
solves the name "puts" and dynamically synthesises a call description of the
appropriate type. Dynamic name resolution is implemented by the POSIX
functions dlopen and dlsym (Figure 5) and call frame synthesis uses the libffi
library to handle the low-level details.

The dlopen and dlsym functions each take two arguments. The two arguments
to dlopen represent the name of a library to load and a list of flags,
specified by POSIX, that specify the resolution strategy. The two arguments
to dlsym represent the library to search and the symbol to search for; if the
optional library argument is not passed, then the symbol table of the exe-
cutable of the calling process is searched instead.

Call synthesis involves two basic types. The first, \texttt{ffitype}, represents C
types; there is a value of \texttt{ffitype} for each scalar type:

\begin{verbatim}
type ffitype
val int_ffitype : ffitype
val pointer_ffitype : ffitype
(* etc. *)
\end{verbatim}

and a function that returns the \texttt{ffitype} corresponding to each \texttt{typ} value:

\begin{verbatim}
val ffitype_of_typ : \alpha \ typ \rightarrow ffitype
\end{verbatim}

The second type, \texttt{callspec}, describes a call frame structure as a list of
argument types and a result type, which can be used to calculate the
appropriate size of a buffer for storing argument and return values. There are
primitive operations for creating a new \texttt{callspec}, for adding arguments, and
for marking the \texttt{callspec} as complete and specifying the return type:

\begin{verbatim}
type callspec
val alloc_callspec : unit \rightarrow callspec
val add_argument : callspec \rightarrow ffitype \rightarrow int
val prepare_call : callspec \rightarrow ffitype \rightarrow unit
\end{verbatim}

The return value of the \texttt{add_argument} function is an integer representing an
offset into the buffer that is used for storing arguments when making a call.

Finally, the \texttt{call} function takes a function address, a completed \texttt{callspec},
and two callback functions, and performs a call.

\begin{verbatim}
val call : address \rightarrow callspec \rightarrow (address \rightarrow \alpha) \rightarrow (address \rightarrow \alpha)
\end{verbatim}

In more detail, \texttt{call addr cs w r} allocates a buffer large enough to hold the
arguments described by \texttt{cs}, populates the buffer with arguments by passing
its address to the function \texttt{w}, invokes the function at \texttt{addr}, and then reads
and returns the return value from the buffer by passing its address to the
function \texttt{r}.

Callbacks suitable for passing to \texttt{call} may be constructed using the \texttt{read}
and \texttt{write} functions that read and write values of specified types to memory:

\begin{verbatim}
val read : \alpha \ typ \rightarrow address \rightarrow \alpha
val write : \alpha \ typ \rightarrow \int \rightarrow \alpha \rightarrow address \rightarrow unit
\end{verbatim}

The call \texttt{read} \texttt{t addr} reads a single value of the type described by \texttt{t} from the
address \texttt{addr}; for example, \texttt{read int} builds a function that can be passed to
\texttt{call} to read an \texttt{int} return value. Similarly, the call \texttt{write} \texttt{t o addr} writes a
single value of the type described by \texttt{t} at offset \texttt{o} from \texttt{addr}. The additional
offset argument reflects the fact that a function may have many arguments, which must be written to different portions of the buffer allocated by call; for example, the following callback function might be passed to call to populate the buffer with an int argument 3 at offset x and a second float argument 4.0 at offset y.

```plaintext
(fun addr →
  write int x 3 addr;
  write float y 4.0 addr)
```

Here is an implementation of foreign_dyn in terms of these operations:

```plaintext
let fn_of_ptr fn (addr,_) =
  let callspec = alloc_callspec () in
  let rec build : type a. a fn → (address → unit) list → a =
    fun fn writers → match fn with
      | Returns t →
        let () = prepare_call callspec (ffitype_of_typ t) in
        call addr callspec
        (fun p → List.iter (fun w → w p) writers)
        (read t)
      | Fn (p, f) →
        fun v →
        let offset = add_argument callspec (ffitype_of_typ p) in
        build f (write p offset v :: writers)
    in build fn

let foreign_dyn name fn =
  fn_of_ptr fn (dlsym name, ptr void)
```

The foreign_dyn function combines a call to dlsym with a call to a second function fn_of_ptr that builds a callable function from a function type representation and an address. The fn_of_ptr function first 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 Returns 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 step: the user can call foreign interactively, as shown above.

3.1. Function pointers

The foreign_dyn implementation 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 uses fn_of_ptr to build a call frame
typedef int (*compar_t)(void *, void *);
int qsort(void *base, size_t nmemb, size_t size, compar_t cmp)

Figure 6: The C qsort function

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

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

Figure 7: Using funptr to bind to qsort

from the address and the function type description. The fn_of_ptr function
is sometimes useful independently, and it is exposed as a separate operation.

Conversions in the other direction are also useful, since an OCaml function
passed to C must be converted to an address:

val ptr_of_fn : α fn → α → unit ptr

The implementation of ptr_of_fn is based on the callspec interface used
to build the call interpreter and uses an additional 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

These conversion functions are rather too low-level to expose directly
to the user. Instead, the following view converts between addresses and
functions automatically:

val funptr : α fn → α typ
let funptr fn = View (fn_of_ptr fn, ptr_of_fn fn, ptr void)

The funptr function builds object type representations from function type
representations, just as C function pointers build object types from function
types. Figure 7 shows funptr in action, describing the callback function for
qsort (Figure 6). The resulting qsort binding takes OCaml functions as
arguments:

qsort arr nmemb sz

10
The `from_voidp` function converts a `void *` value to another 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.

### 4. A staged interpreter for bindings

Interpreting function type descriptions dynamically (Section 3) is convenient for interactive development, but has a number of drawbacks. First, the implementation suffers from significant interpretative overhead (quantified in Section 8). Second, there is no check that the values passed between OCaml and C have appropriate types. The 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, it is impossible to 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 that are not available in the simple reimplementation of the call logic. Similarly, sidestepping the usual symbol resolution process makes it impossible to use tools like `nm` and `objdump` to interrogate object files and executables.

Fortunately, all of these problems share a common cure. Instead of basing the implementation of `FOREIGN` on an evaluation of the type representation, the representation can be used to generate both C code that can be checked against the declared types of the bound functions and OCaml code that links the generated C code into the program.

As the introduction promised, binding descriptions written using `FOREIGN` can be reused for code generation without any changes to the descriptions themselves. However, switching from `Foreign_dyn` to an approach based on code generation does require changes in the way that programs are organised and built. The `Foreign_dyn` module makes bindings available immediately via a single functor application:

```
Puts(Foreign_dyn)
```

In contrast, the approach described in this section involves applications of the
module type FOREIGN = sig
  type α fn

  val (@→) : α typ → β fn → (α → β) fn
  val returning : α typ → α fn

  type α result
  val foreign : string → α fn → α result
end

Figure 8: The FOREIGN signature for describing foreign functions (version 2)

Puts functor to three different implementations of FOREIGN. The Puts functor is first applied to modules Foreign_GenC and Foreign_GenML that respectively generate a C file and an OCaml module.

Puts(Foreign_GenC)
Puts(Foreign_GenML)

The generated files are then compiled and linked into the final program by means of a second application of Puts:
Puts(Foreign_GeneratedML)

The remainder of this section describes these various implementations of FOREIGN in more detail.

Transforming the evaluator of Section 3 into a code generator can be seen as a form of staging, i.e. specializing the dynamic foreign function based on static information (the type description) in order to improve its performance when the time comes to supply the remaining arguments (the arguments to the bound function). As we shall see, the principles and techniques used in the staging and partial evaluation literature will be helpful in implementing the code-generating foreign.

In order to support the staged interpretation, a small adjustment is needed to the FOREIGN signature. In the dynamic implementation of FOREIGN, the foreign function returns an OCaml value of type α matching the index of the fn argument that represents the type of the bound function. In the staged

---

5 As a reviewer notes, two of these implementations, Foreign_GenC and Foreign_GenML, which are described separately for expository purposes, could be combined into a single module, eliminating one application of Puts and simplifying the build process for the user.
interpretation, the foreign function generates code rather than returning a value directly. In order to support both the dynamic and the staged implementations, we give the foreign function an abstract return type (Figure 8) that can be instantiated appropriately in each implementation. Although the signature of FOREIGN changes, no change to the bindings description (for puts) is needed.

4.1. Generating C

The first FOREIGN implementation, Foreign_GenC (Figure 9), uses the name and the type representation passed to foreign to generate C code. The functor application Puts(Foreign_GenC) passes the name and type representation for puts to Foreign_GenC.foreign, which generates a C wrapper for puts.

The generated C code, shown below, converts OCaml representations of values to C representations, calls puts and translates the return value representation back from C to OCaml⁶. If the user-specified type of puts is incompatible with the type declared in the C API then the C compiler will complain when building the generated source.

```ocaml
value ctypes_puts(value x0) {
  char *x1 = ADDR_OF_PTR(x0);
  int x2 = puts(x1);
  return Val_int(x2);
}
```

⁶There are no calls to protect local variables from the GC because the code generator can statically determine that the GC cannot run during the execution of this function. However, it is not generally possible to determine whether the bound C function can call back into OCaml, and so the user must inform the code generator if such callbacks may occur by passing a flag to foreign.
module Foreign_GeneratedML = struct
  (* ... *)
  external ctypes_puts : address → int = "ctypes_puts"

  type α result = α

  let foreign : type a. string → a fn → a =
    fun name t → match name, t with
    | "puts", Fn (View (_, write, Ptr _), Returns Int) →
      (fun x1 → ctypes_puts (fst (write x1)))
end

Figure 10: The generated module, Foreign_GeneratedML, which matches FOREIGN (Figure 8)

4.2. Generating OCaml

The second new FOREIGN implementation, Foreign_GenML (Figure 9), generates an OCaml wrapper for ctypes_puts. The ctypes_puts function deals with low-level representations of OCaml values; the OCaml wrapper exposes the arguments and return types as typed values. The functor application Puts(Foreign_GenML) passes the name and type representation of puts to Foreign_GenML.foreign, which generates an OCaml module Foreign_GeneratedML that wraps ctypes_puts.

The OCaml module generated by Foreign_GenML (Figure 10) also matches the FOREIGN signature. The central feature of the generated code is the foreign implementation that scrutinises the type representation passed as argument in order to build a function that extracts raw addresses from the pointer arguments to pass through to C.

The type variable a is initially abstract but, since the type of t is a GADT, examining t using pattern matching reveals information about a. In particular, since the type parameter of fn is instantiated to a function type in the definition of the Fn constructor (Figure 4), the right-hand side of the first case of the definition of foreign above is also expected to have function type. Similar reasoning about the Ptr, Int and Returns constructors reveals that the right-hand side should be a function of type σ ptr → int for some type σ, and this condition is met by the function expression in the generated code.
4.3. Linking the generated code

The generated OCaml module Foreign_GeneratedML serves as the third FOREIGN implementation; it has the following type:

\[
\text{FOREIGN with type } \alpha \text{ result } = \alpha
\]

The application Puts(Foreign_GeneratedML) supplies Foreign_GeneratedML as the argument F of the Puts functor (Section 2.2). The generated foreign function above becomes F.foreign in the body of Puts, and receives the name and type representation for puts as arguments. The inspection of the type representation in foreign serves as a form of type-safe linking, checking that the type specified by the user matches the known type of the bound function. In the general case, the type refinement in the pattern match within foreign allows the same generated implementation to serve for all the foreign function bindings in the Puts functor, even if they have different types.

4.4. The Trick

The pattern match in the Foreign_GeneratedML.foreign function can be seen as an instance of a binding-time improvement known in the partial evaluation community as The Trick (Danvy et al., 1996). The Trick transforms a program to introduce new opportunities for specialization by replacing a variable whose value is unknown with a branch over all its possible values. In the present case, the Foreign_GeneratedML.foreign function will only ever be called with those function names and type representations used in the generation of the Foreign_GeneratedML module. Enumerating all these possibilities as match cases results in simple non-recursive code that may easily be inlined when the functor is applied.

4.5. Cross-stage persistence

The scheme above, with its three implementations of FOREIGN, may appear unnecessarily complicated. It is perhaps not immediately obvious why we should not generate C code and a standalone OCaml module, eliminating the need to apply the Puts functor to the generated code.

One advantage of the three-implementation scheme is that the generated code does not introduce new types or bindings into the program, since the generated module always has the same known type (i.e. FOREIGN). However, there is also a more compelling reason for the third implementation.

The Foreign_GeneratedML.foreign function converts between typed arguments and return values and low-level untyped values which are passed to C.
In the case where the type of an argument is a view, converting the argument involves applying the write function of the view representation. For example, the binding to puts uses the str view of Section 2 to support an argument that appears in OCaml as a string and in C as a char *:

```ocaml
let puts = foreign "puts" (str @ returning int)
```

Calling puts with an argument s involves applying ptr_of_string to s to obtain a char*. However, there is no way of inserting ptr_of_string into the generated code. In the representation of a view the write function is simply a higher-order value, which cannot be converted into an external representation. This is analogous to the problem of cross-stage persistence in multi-stage languages: the generated code refers to a value in the heap of the generating program.

The three-implementation approach neatly sidesteps the difficulty. There is no need to externalise the write function; instead, the generated foreign implementation simply extracts write from the value representation at the point when Puts is applied:

```ocaml
let foreign : type a. string \to a fn \to a =
  fun name t \to
    match name, t with
    | "puts", Fn (View {write}, Returns Int) \to
      (fun x1 \to ctypes_puts (write x1).addr)
    | (*) ...
```

Thus, the third implementation of FOREIGN makes it possible to use views and other higher-order features in the type representation.

5. Inverted bindings

Section 3.1 showed how to invert the call interpreter to support callbacks; Section 4 showed how to stage the call interpreter to improve safety and speed. The question naturally arises: Is there a use for an inverted, staged interpreter? It turns out that there is.

The primary use of ctypes 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 this section shows, it is possible to expose OCaml code to C via an interpretation of FOREIGN that interprets the parameter of the result type as a value to consume rather than a value to produce.
Specialising the result type of the FOREIGN signature (Fig 8) with a type that consumes $\alpha$ values gives the following type for foreign:

```ocaml
val foreign : string → $\alpha$ fn → ($\alpha$ → unit)
```

that is, a function that takes a name and a function description and consumes a function. This consumer of functions is just what is needed to turn the tables: rather than resolving and binding foreign functions, this implementation of foreign exports host language functions under specified names.

Continuing the running example, this foreign implementation can export a function whose interface matches puts. Once again, it suffices to apply the Puts functor from Section 2.2 to a suitable module. As with the staged call interpreter (Section 4), Puts is applied multiple times – once to generate a C header and a corresponding implementation that forwards calls to OCaml callbacks, and again to produce an exporter that connects the C implementation with our OCaml functions.

The ctypes library includes a generic pretty-printing function that formats C type representations using the C declaration syntax. Applying the pretty-printer to the puts binding produces a declaration suitable for a header:

```c
int puts(char *);
```

The generation of the corresponding C implementation proceeds similarly to the staged 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.

The generated OCaml module Foreign_GeneratedInvML populates the array when the module is loaded by calling a function register_callback with a value of type $t$ callback.

```ocaml
val register_callback : $\alpha$ callback → $\alpha$ → unit
```

The type parameter of the callback value passed to register_callback is the type of the registered function:

```ocaml
type _ callback = Puts : (address → int) callback
```

Finally, the generated foreign function is reminiscent of the staged implementation of Section 4; it scrutinises the type representation to produce a function consumer that passes the consumed function to register_callback:
let foreign : type a. string → a fn → (a → unit) =
  fun name t → match name, t with
  | "puts", Fn (View (read, _, Ptr _), Returns Int) →
    (fun f → register_callback Puts
     (fun x1 → f (read x1)))

The applied module Puts(Foreign_GeneratedInvML) exports a single function, puts, which consumes an OCaml function to be exported to C:

val puts : (string → int) → unit

Decorne et al. (2016) give a more detailed study of the use of ctypes inverted bindings, which they use to wrap ocaml-tls (Kaloper-Mersinjak et al., 2015) to build an OCaml-based replacement for the libtls library.

6. More interpretations

6.1. errno

Several standard C functions, and many POSIX functions, store information about errors in a global integer variable, errno. For example, the following C snippet resets errno to clear any earlier errors, attempts to call chdir to change the working directory and, if the attempt fails, uses errno to display details about the problem before exiting the program:

errno = 0;
if (chdir("/tmp") < 0) {
    fprintf(stderr, "chdir failed: %s\n", strerror(errno));
    exit(1);
}

The errno variable can also be used as the basis of more sophisticated error-handling strategies — retrying the failed call, transferring control elsewhere in the program, and so on. For maximum flexibility in handling errors that arise when calling C functions from OCaml it is convenient to return errno alongside the return value of each bound function, allowing the code that calls errno to determine how errors should be handled.

Supporting returning errno from C to OCaml alongside the return value of each function requires one final modification to the FOREIGN signature. Figure 11 shows the final version of FOREIGN, which extends the type of returning with an abstract type return. For our existing implementations of FOREIGN, the type return is defined as the identity:

type α return = α
module type FOREIGN = sig
  type α fn

  type α return
  val (@→) : α typ → β fn → (α → β) fn
  val returning : α typ → α return fn

  type α result
  val foreign : string → α fn → α result
end

Figure 11: The FOREIGN signature for describing foreign functions (version 3)

while for a new implementation Foreign_GenML_errno, return may be defined as the pair of its parameter and an integer representing errno:

  type α return = { rv: α; errno: int }

(Errno support also requires a companion implementation Foreign_GenC_errno to generate C code that captures and returns errno.)

6.2. Asynchronous calls

Besides supporting errno, the abstraction over the return type in the definition of FOREIGN in Figure 11 supports a number of other cases in which bound functions return in unusual ways. In particular, it supports a style of concurrency that is widely used in the OCaml community.

Since the standard OCaml runtime has limited support for concurrency, many modern OCaml programs make use of cooperative concurrency libraries such as Lwt (Vouillon, 2008), which expose monadic interfaces. Cooperative concurrency requires taking care with potentially-blocking calls, since a single blocking call can cause suspension of all threads. In the Lwt framework, a potentially-blocking function that returns a value of type t is given the return type t Lwt.t. Functions of this type may be connected together with a monadic bind operator, which may transfer control to another lightweight thread. For example, a binding to the puts function in the Lwt framework has type

  val puts : string → int Lwt.t

Producing bindings of this type from our sample bindings specification involves generating an implementation of FOREIGN with a suitable definition of return:
type α return = α Lwt.t

There are several ways to actually construct Lwt.t values. A simple approach, provided by the detach function from the Lwt_preemptive module

val detach : (α → β) → α → β Lwt.t

simply runs a potentially blocking function using one of a pool of system threads. Support for concurrency using Lwt_preemptive.detach requires simple modifications to Foreign_GenML, to insert a call to detach around each call to a bound function, and to Foreign_GenC, to release OCaml’s global runtime lock when calling the corresponding C functions.

The Lwt jobs framework offers a second, finer-grained approach to building Lwt.t values. A job is a bound function that can run in a C thread, without interacting with the OCaml runtime. The Lwt jobs interface splits a binding to a C function into several stages: creating a job by converting the arguments from the OCaml to the C representation; running the job, by calling the C function with the converted arguments; cleaning up the job and collecting the results; converting the results back to the OCaml representation. Since these different stages may run in separate threads, there are a number of subtle invariants, that are easy to violate in hand-written code, but that the ctypes implementation maintains automatically.

The Lwt and errno approaches may, of course, be combined, by defining the return type as an application of Lwt.t to a pair of a value and an errno code:

type α return = (α * int) Lwt.t

6.3. Out-of-process calls

High-level languages often make strong guarantees about type safety that are compromised by binding to foreign functions. Safe languages such as OCaml 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. This can be accomplished using a variant of the staged call interpreter (Section 4) in which, instead of invoking bound C functions directly, the generated stubs marshal the arguments into a shared memory buffer where they are retrieved by an entirely separate process which contains the C library.
struct puts_frame {
    enum function_id id;
    const char *p;
    int return_value;
};

Figure 12: A struct for making cross-process calls to puts

Once again, this cross-process approach is straightforward to build from existing components. The data representation is based on C structs: for each foreign function the code generator outputs a struct with fields for function identifier, arguments and return value (Figure 12). The struct is built using the type representation constructors (Section 2.1) and printed using the generic pretty printer. These structs are then read and written by the generated C code in the two processes. 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.

Section 8 describes experiments that quantify the overhead of these cross-process calls.

7. Structure layout

As we have seen, defining foreign function bindings using an abstract FOREIGN interface allows considerable flexibility in the interpretation of those bindings. As we shall now see, a similar approach may be used to address the other principal challenge in interfacing with foreign libraries, namely determining the layout of objects in memory.

Figure 13 gives the signature for the POSIX function gettimeofday, which accepts two arguments, a pointer to a struct timeval (Figure 14) and a pointer to struct timezone. Constructing suitable arguments for gettimeofday requires determining information about the layout of the structs, i.e. the sizes and offsets of the fields, along with any trailing padding.

Just as the OCaml binding to puts was described using the operations of FOREIGN interface, the struct timeval and struct timezone types may be described using the operations of an interface TYPE (Figure 15).

Figure 16 gives a definition of the timeval structure using TYPE. The first line creates a module Tv representing an initially empty struct type timeval.
int gettimeofday(struct timeval *tv, struct timezone *tz);

Figure 13: The gettimeofday function

struct timeval {
    unsigned long tv_sec; /* seconds */
    unsigned long tv_usec; /* microseconds */
};

Figure 14: The timeval struct

module type TYPE = sig
  type τ structure and (α, τ) field
module type STRUCTURE = sig
  type t
  val t : t structure typ
  val field: string → α typ → (α, t) field
  val seal: unit → unit
end
val structure: string → (module STRUCTURE)
end

Figure 15: The TYPE interface

The module creation operation is generative — that is, each call to structure creates a module with a type t that is distinct from every other type in the program.

The second and third lines call the Tv.field function to add unsigned long fields with the names tv_sec and tv_usec. Calling Tv.field performs an effect and returns a value: it extends the struct represented by Tv with an additional field, and it returns a value representing the new field, which may be used later in the program to access struct tv values.

The final line “seals” the struct type representation, turning it from an incomplete type into a fully-fledged object type with known properties such as size and alignment, just as the closing brace in the corresponding C declaration marks the point in the C program at which the struct type is completed. Adding fields to the struct representation is only possible before the call to seal, and creating values of the represented type is only possible afterwards;
module Timeval(T: TYPE) = struct
module Tv = (val T.structure "timeval")
let sec = Tv.field "tv_sec" ulong
let usec = Tv.field "tv_usec" ulong
let () = Tv.seal ()
end

Figure 16: The timeval structure layout, using TYPE

violation of either of these constraints results in an exception.

As with FOREIGN, there are multiple possible implementations of the TYPE interface and its operations structure field and seal.

7.1. Computing layout information

As with FOREIGN, we first consider a dynamic implementation of TYPE that simply computes the appropriate layout directly (Figure 17).

The structure function builds an incomplete empty struct with no alignment requirements. The field function computes the next alignment boundary in the struct for its field argument, and updates the alignment requirements for the struct. The seal function inserts any padding necessary to align the struct and marks it as complete.

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. First, libraries may specify non-standard layout requirements (e.g. with the __packed__ attribute), and attempting to replicate these quickly becomes unmanageable. Second, some libraries, define structs with interspersed internal fields which vary both across platforms and across versions. (The libuv asynchronous I/O library is a typical example.) Replicating this variation in the bindings quickly leads to unmaintainable code.

7.2. Retrieving layout information

These drawbacks can be avoided with an alternative implementation of TYPE that, instead of attempting to replicate the C compiler’s structure layout algorithm, uses the C compiler itself as the source of layout information, much as the staged foreign (Section 4) generates C code to bind functions rather than using libffi to replicate the calling convention.

As with the staged foreign function, the idea is to use The Trick to transform field and seal from functions that compute the layout into functions
type α typ = (* ... *)
| Struct : { mutable complete: bool; mutable fields : α bfield list; }
    → α strct typ
and σ bfield = Field : (α, σ) fld → σ bfield
and (α, σ) fld = { typ: α typ; offset: int; name: string; }
and α strct = α strct ptr

module Type_dyn = struct
  type τ structure = τ strct and (α, σ) field = (α, σ) field
  module type STRUCTURE = sig
    type t
    val t : t structure typ
    val field: string → α typ → (α, t) field
    val seal: unit → unit
  end

  let structure name =
    (module struct
      type t
      let t = Struct { complete = false; fields = [] }
      let field name typ =
        let offset = compute_offset t in
        let f = { typ; offset; name } in
        t.fields <- f :: t.fields;
        f
      let seal t = compute_padding t; t.complete <- true
    end : STRUCTURE)
end

Figure 17: The Type_dyn implementation of TYPE (Figure 15)

that map particular concrete arguments into previously computed layout information. In order to bring the layout information directly into the OCaml program an additional stage is needed: first, the Timeval functor (Figure 16) is applied to a module Generate_C to produce a C program that retrieves layout information with calls to offsetof and sizeof:

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

Compiling and running the C program produces an OCaml module Types_impl that satisfies the TYPE signature (much as the generated Foreign_GeneratedML module satisfies FOREIGN), and which contains implementations of field and seal specialized to the structs and fields of the Timeval module:
let field s fname ftype = match s, fname with
| Struct { tag = "timeval"}, "tv_sec" → {ftype; fname; foffset = 4}
(* ... *)

The application Timeval(Types_impl) passes the layout information through to the calls to Tv.field and Tv.seal, making it available for use in the program.

This technique extends straightforwardly to retrieving other information that is available statically, such as the values of enum constants or preprocessor macros.

8. Practical aspects

There is evidence that the modular approach to foreign function bindings described here works well in practice. The open-source ctypes library has seen rapid adoption in the OCaml community: at the time of writing there are fifty-three packages with direct dependencies on the latest version of ctypes available via the OCaml package manager OPAM, and many more by dozens of authors on GitHub and in private commercial use. Ctypes is in commercial use at several companies, including Citrix, Cryptosense, Jane Street Capital and Docker. The library has been ported beyond Linux to OpenBSD, FreeBSD, MacOS X, Windows, the MirageOS unikernel (Madhavapeddy et al., 2013), the Android and iPhone mobile phone environments and the Arduino microcontroller. Many of the projects built on ctypes are substantial: for example the tglsl bindings to OpenGL replace earlier OpenGL bindings which comprised over 11000 lines of hand-written C. Table 1 lists a few well-known ctypes projects; libraries written by the authors of this paper are marked with an asterisk.

8.1. Case studies

Multiple interpretations in practice. One common pattern when developing bindings with ctypes is to start with an implementation based on the dynamic implementation of FOREIGN (Section 3), which supports easy interactive development, then switch to the staged interpretation (Section 4) for improved performance and safety once the bindings are mature. A number of libraries have followed this path; one recent example is the ocaml-mariadb library\footnote{https://github.com/andrenth/ocaml-mariadb}, which provides bindings to the MariaDB database. As is commonly the case,
<table>
<thead>
<tr>
<th>Interface</th>
<th>Topic</th>
<th>Interpretations</th>
</tr>
</thead>
<tbody>
<tr>
<td>libsodium*</td>
<td>cryptography</td>
<td>staged</td>
</tr>
<tr>
<td>Nebula</td>
<td>CPU emulation</td>
<td>dynamic</td>
</tr>
<tr>
<td>mariadb</td>
<td>database</td>
<td>staged</td>
</tr>
<tr>
<td>argon2</td>
<td>hashing</td>
<td>dynamic</td>
</tr>
<tr>
<td>GNU SASL</td>
<td>authentication</td>
<td>staged</td>
</tr>
<tr>
<td>glibc passwd</td>
<td>identity</td>
<td>dynamic</td>
</tr>
<tr>
<td>GDAL/OGR</td>
<td>geography</td>
<td>dynamic</td>
</tr>
<tr>
<td>zstd</td>
<td>compression</td>
<td>staged</td>
</tr>
<tr>
<td>libzbar</td>
<td>barcodes</td>
<td>dynamic</td>
</tr>
<tr>
<td>libnl</td>
<td>networking</td>
<td>dynamic</td>
</tr>
<tr>
<td>nanomsg</td>
<td>messaging</td>
<td>staged</td>
</tr>
<tr>
<td>LZ4</td>
<td>compression</td>
<td>staged</td>
</tr>
<tr>
<td>OpenGL (ES)</td>
<td>graphics</td>
<td>dynamic</td>
</tr>
<tr>
<td>SDL</td>
<td>multimedia</td>
<td>dynamic</td>
</tr>
<tr>
<td>gccjit</td>
<td>compilation</td>
<td>staged</td>
</tr>
<tr>
<td>fsevents*</td>
<td>OS X</td>
<td>staged</td>
</tr>
<tr>
<td>sys/stat.h*</td>
<td>Posix</td>
<td>staged, Lwt</td>
</tr>
<tr>
<td>FUSE protocol*</td>
<td>file systems</td>
<td>data type</td>
</tr>
<tr>
<td>Tokyo Cabinet</td>
<td>database</td>
<td>dynamic</td>
</tr>
<tr>
<td>libuv</td>
<td>async I/O</td>
<td>staged</td>
</tr>
<tr>
<td>libudev</td>
<td>OS interface</td>
<td>dynamic</td>
</tr>
<tr>
<td>llibnqsb-tls*</td>
<td>cryptography</td>
<td>inverted</td>
</tr>
<tr>
<td>hardcaml-vpi</td>
<td>hardware simulation</td>
<td>dynamic</td>
</tr>
</tbody>
</table>

Table 1: Some bindings using ctypes

switching interpretations required no non-trivial changes to the binding descriptions themselves, but did involve some restructuring of the surrounding code. In particular, one module in ocaml-mariadb initially included both the binding descriptions and some high-level functions that used the bindings. After the switch to the staged approach these high-level functions depended on the combination of the bindings descriptions and the generated OCaml module, and accommodating this new dependency involved moving the high-level functions to a separate module.

In some cases it is convenient to support multiple interpretations simultaneously. For example, the ocaml-sys-stat package, which provides bindings to the types and functions in the Posix `<sys/stat.h>` header, exports both
module Types(T: Cstubs.Types.TYPE) = struct
  open T
  module Version_7_9 = Profuse_types_7_9.Types(F)
  (* ... *)
  module Flags = struct
    let fopen_nonseekable = constant "FOPEN_NONSEEKABLE" t
  end (* ... *)

Figure 18: Part of the FUSE 7.10 bindings in profuse, built atop the FUSE 7.9 bindings

synchronous and asynchronous (Lwt) versions of the same bindings, obtained by applying a single functor to two implementations of FOREIGN.

Finally, it is sometimes useful to build custom implementations of FOREIGN for particular needs, such as instrumenting every bound function to print tracing information; the functor-based approach described here straightforwardly supports this kind of change without any need to modify the binding descriptions.

FUSE. The profuse FUSE protocol library\(^8\) uses ctypes solely for its ability to represent the types of binary protocols and perform C structure layout queries (Section 7). 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 ctypes's type representation and layout query features is only possible due to the modular embedding of the C type system; an external bindings generator tool would be much harder to repurpose.

Additionally, there are several versions of the FUSE protocol in active use: FUSE 7.8 is widely supported, but recent Linux releases support FUSE 7.23, which offers many more features. In order to support several versions simultaneously, profuse binds the structures and values of each version using a functor which imports the binding for the predecessor version in its body, using the OCaml module language to override and extend only the parts that

\(^8\) https://github.com/effuse/profuse/
have changed. Figure 18 gives a typical example: the bindings for FUSE 7.10 are defined as a number of small changes to the bindings for FUSE 7.9, such as the addition of a flag \texttt{FOPEN\_NONSEEKABLE}.

8.2. Overhead: call latency

![Figure 19: FFI call latency by arity: staged-for-isolation and interpreted bindings](image)

To evaluate the overhead of \texttt{ctypes}, we wrote bindings for ten simple machine integer functions of arity 0 to 9 which return their last argument. We then interpreted these bindings both dynamically with \texttt{libffi} (Figure 19) and statically through a staged compilation (Figure 20). 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 \textit{manual} variation followed exactly the FFI directions in \textit{Chapter 19 of the OCaml 4.02.1 manual}. The \textit{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 \texttt{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 19 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 (Section 6.3). 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%.

9. Related work

The approach of representing foreign language types as native language values is inspired by several existing FFIs, including Python’s ctypes, Common Lisp’s Common FFI and Standard ML’s NLFFI (Blume, 2001), each of
which takes this approach.

This paper follows NLFFI’s approach of indexing foreign type representations 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, this paper departs from NLFFI 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.

The use of functors to abstract over interpretations of the TYPE and FOREIGN signatures is a central technique in this paper. Carette et al. (2009) use functors in a similar way, first abstracting over the interpretation of an embedded object language (λ calculus), then developing a variety of increasingly exotic interpretations that perform partial evaluation, CPS translation and staging of terms.

The use of GADTs to represent foreign language types, and their indexes to represent the corresponding native language types (Section 2) can be viewed as an encoding of a universe of the kind used in dependently-typed programming (Nordström et al., 1990; Benke et al., 2003). Altenkirch and McBride (Altenkirch and McBride, 2003) 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.

Mapping codes to types and their interpretations by abstracting over a parameterised type constructor is a well-known technique in the generic programming community (Yang, 1998; Cheney and Hinze, 2002). Hinze (Hinze, 2006) describes a library for generic programming in Haskell with a type class that serves a similar purpose to the TYPE signature of Section 7, 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 (Karvonen, 2007) corresponds even more directly to this aspect of the design presented here.

Availability. The ocaml-ctypes code is available online: https://github.com/ocamllabs/ocaml-ctypes

References


Vouillon, J., 2008. Lwt: A cooperative thread library. ML ’08. ACM.
