Declarative foreign function binding through generic programming

Jeremy Yallop, David Sheets and Anil Madhavapeddy

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 inflexibile 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.
In contrast, approaching the design of a foreign function interface as a *generic programming* problem allows foreign function specifications to be written declaratively, with easy switching between a wide variety of binding mechanisms — static and dynamic, synchronous and asynchronous, etc. — with no changes to the specifications.

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 `gettimeofday` to retrieve the current time:

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

Before calling `gettimeofday`, the programmer must write a binding that exposes the C function as an OCaml function. Writing bindings presents many opportunities to introduce subtle errors [8, 12, 14], although it is a conceptually straightforward task: the programmer must convert the arguments of the bound function from OCaml values to C values, pass them to `gettimeofday`, and convert the result back to an OCaml value.

In fact, bindings for functions such as `gettimeofday` can be produced mechanically from their type definitions, and tools that can generate bindings (e.g. [2]) 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, based on generic programming (e.g. [9]), a collection of techniques for defining functions such as equality, serialisation, and traversal that can be applied at a wide variety of types. Generic programming involves introducing a representation of some collection of types, then writing generic functions, parameterised by that representation, that can operate across all of the corresponding types.

The starting point of generic programming is typically a representation of host language types. However, as this paper shows, generic programming techniques can also be applied to binding foreign functions, where the types of interest are the types of the foreign language, and the generic functions are binding strategies that turn the names and types of foreign-language functions into functions that can be called from the host language. In this way it is possible to eliminate the boilerplate needed to bind foreign functions — not by generating it with an external tool, but by using the abstraction mechanisms of the language to parameterise over the common type structure. The result is type-safe, flexible, and tightly integrated into the host language.

For concreteness, this paper focuses on *ocaml-ctypes*, a widely-used library for calling C functions from OCaml based on the generic programming approach, and assumes some knowledge of OCaml language features such as functors and generalized algebraic data types (GADTs) [13]. However, the techniques described in the following pages can be used to build a declarative foreign function library in any language that supports generic programming.

1.1 Outline

The generic programming approach presented here involves two key ingredients.

The first ingredient is an interpretation-independent representation of foreign language types as host language values (§2).

The second ingredient is an abstract binding interface that can be implemented in different ways to support different binding mechanisms. Section §3 develops various such mechanisms, including an evaluator for binding foreign functions dynamically (§3.1), a code generator for generating bindings statically (§3.2), an inverted approach for exposing host language functions to the foreign language, and some more exotic approaches, for asynchronous calls and out-of-process calls with improved memory safety (§3.3).

The techniques used for declarative binding of foreign-language functions can also be applied to determining the layout of foreign-language objects (§4).

The extended version of this paper offers more complete code listings of some of the generic functions and generated code from this edition and additional evidence for the practicality of the generic programming approach to foreign function interface design, including a description of a number of real-world uses of *ocaml-ctypes* in commercial and free software, and measurements that show that the performance of bindings generated by *ocaml-ctypes* is comparable to that of hand-written code.

2 Representing foreign types as native values

The first step in building a generic foreign function library is constructing a representation of foreign language types as host language values.

```
type _ ctype =
  Void    : unit ctype
| Char    : char ctype
| Int     : int ctype
| Pointer : α ctype → α ptr ctype
| View    : { read : β → α; write : α → β; ty: β ctype } → α ctype
| Struct  : struct_type → α structure ctype
| Funptr : α cfn → α funptr ctype
and _ cfn =
  Returns : α ctype → α cfn
| Fn : α ctype * β cfn → (α → β) cfn
and α structure = (* elided *) and struct_type = (* elided *)
and α ptr = (* elided *)
```

Fig. 1: C type representations, concretely

```
module type TYPE = sig
 type α cty
 val void: unit cty

 (* Scalar types *)
 val char: char cty
 val int: int cty
 val ptr: α cty → α ptr cty
 val view: (α → β) → (β → α) → α cty → β cty

 (* Aggregate types *)
 type τ structure and (α, τ) field
 module type STRUCTURE = sig
   type t
   val t : t structure cty
   val field: string → α cty → (α, t) field
   val seal: unit → unit
 end
 val structure: string → (module STRUCTURE)

 (* Functions and function pointers *)
 type α fn
 val returning: α cty → α fn
 val (@→): α cty → β fn → (α → β) fn

 type α funptr
 val funptr: α fn → α funptr cty
end
```

Fig. 2: C type representations, abstractly

Figure 1 defines generalized algebraic datatypes (GADTs) `ctype` and `cfn` for representing a variety of C object and function types. Each C type is mapped to a corresponding OCaml type, which is represented by the type parameter of `ctype`; for example, a value of type `int ctype` represents a C type that appears in OCaml as a value of type `int`.

The `TYPE` signature (Figure 2) provides an abstract interface for building type representations, with an abstract type `cty` in place of the concrete type `ctype` and a function for each constructor. Using `cty` rather than using `ctype` directly introduces additional flexibility in mapping types as described by the user to concrete representations of types, as Section 4 will show.

Representing C scalar types The constructors `Void`, `Char` and `Int` represent the C types with corresponding names, which are mapped to the OCaml types `unit`, `char` and `int`. (The full implementation supports the other scalar types — `float`, `short`, etc.) The `Pointer` constructor builds a C type representation from another C type representation, much as the C type constructor `*` builds a type from a type. (In the full implementation the parameterised type `ptr` comes with various operations for reading and writing values, but they are not needed in the exposition here.) The `View` constructor uses an isomorphism to vary the mapping between C types and OCaml types; for example, given functions for converting between `char ptr` and `string`

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

the following expression builds a value of type `string ctype` to represent values that appear in C as `char *` and in OCaml as `string`:

```
View {read = string_of_ptr; write = ptr_of_string; ty = Pointer Char}
```

Representing C aggregate types Besides scalar types such as integers and pointers, C supports a number of aggregate types. The `TYPE` signature (Figure 2) exports types `structure` and `field` for representing `struct`s and `struct` fields, with a function `structure` for creating new `struct` types, and with a signature `STRUCTURE` that exposes a value `t` representing a `struct`, a function `field` that adds a field to an existing `struct` type, and a function `seal` that converts an incomplete type to a complete type that cannot be further extended. The two type parameters of `field` represent the type of the field and the type of the structure to which the field belongs. The type `t` in the `STRUCTURE` signature operates as a static tag: each call to `structure` generates an instance of `STRUCTURE` whose `t` is distinct from all other types in the program; this prevents `struct` representations from being used interchangeably, which would violate type safety.

Figure 3 shows the `STRUCTURE` machinery in action. Each line of OCaml code (on the right) corresponds to the corresponding line of the C code (on the left), which declares a `struct timeval` with two fields.

The first line creates a module `Tv` representing an initially empty struct type `timeval`. The actual representation of the struct type, based on the `Struct` constructor of Figure 1, is internal to the `Tv` module; only the `field` and `seal` functions and the type representation `t` are exposed through the interface.

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: that is, 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; violation of either of these constraints results in an exception.

There are multiple possible implementations of the `STRUCTURE` interface and its operations `field` and `seal`, which are explored further in Section 4.

```
struct timeval {               module Tv = (val structure "timeval")
   unsigned long tv_sec;       let sec  = Tv.field "tv_sec" ulong
   unsigned long tv_usec;      let usec = Tv.field "tv_usec" ulong
};                             let () = Tv.seal ()
```

Fig. 3: The `timeval` struct in C and OCaml

As with `ptr`, `structure` comes with various operations for reading and writing fields, allocating new structures, and so on, but they are again not needed in this exposition. Additionally, the full implementation supports `union` and array types.

Representing C function types Finally, besides object (i.e. value) types, C supports function and function pointer types. The `TYPE` interface (Figure 2) exports a type `fn` for representing C function types, along with constructors `returning` and `@→`, and a type `funptr` for representing C function pointer values, along with a value `funptr` for constructing function pointer type representations. The following expression constructs a representation of the type of `gettimeofday` from the introduction:

```
ptr Tv.t @→ ptr Tz.t @→ returning int
```

which has the following type, writing `tv` for `Tv.t structure`, and similarly for `tz`:

```
(tv ptr → tz ptr → int) fn
```

As the type parameter `tv ptr → tv ptr → int` indicates, the `@→` builds curried OCaml functions to represent C functions of multiple arguments. However, `returning` and `@→` carefully distinguish object types, which are represented with `cty`, from function types, which are represented with `fn`. 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 the coding represents them differently. More precisely, `returning` builds a representation of a function type from the object type that the function returns, and `@→` adds an object type as an additional argument to an existing function type. The `funptr` function supports the inverse conversion, turning object types into function types.

In the concrete representation of Figure 1, the `ctype` datatype supports a additional constructor `Funptr` for representing function pointers. The `Returns` and `Fn` constructors of the datatype `fn` correspond to the `TYPE` functions `returning` and `@→` functions for building `fn` values.

3 Interpreting type representations

```
module type FOREIGN = sig
 type α res
 val foreign: string → α fn → α res
end
```

Fig. 4: The `FOREIGN` interface

The type representations of Section 2 can support a number of generic operations including `sizeof`, allocation, and pretty-printing of types and values. This section focuses on various implementations of an abstract operation `foreign`, which builds a binding to a foreign function from its name and a representation of its type. Figure 4 shows the `FOREIGN` signature, which contains a single function, `foreign`. The return type, `res`, is left abstract so that each binding strategy can instantiate it appropriately.

```
module Bindings(F : FOREIGN) = struct
 let gettimeofday =
   F.foreign "gettimeofday" (ptr Tv.t @→ ptr Tz.t @→ returning int)
end
```

Fig. 5: Binding `gettimeofday`, abstractly

Figure 5 shows a binding for `gettimeofday`, abstracted over the implementation of `FOREIGN`.

3.1 Dynamically interpreting foreign function bindings

Interpreting calls The first implementation of `foreign` evaluates the type representation to build bindings dynamically. The parameterised type `res` in the `FOREIGN` signature is instantiated with the alias α `res =` α, so the type of `foreign` is as follows:

```
val foreign : string → α fn → α
```

That is, `foreign` turns a C function type description and a name into an OCaml function. Applying `foreign` to the name and type representation of `gettimeofday` in the top level returns a function that can be called immediately:

```
# let f = foreign "gettimeofday" (ptr Tv.t @→ ptr Tz.t @→ returning int)
val f : tv ptr → tz ptr → int = <fun>
```

The call to `foreign` resolves the name `"gettimeofday"` and dynamically synthesises a call description of the appropriate type. In the *ocaml-ctypes* implementation, dynamic name resolution is implemented by the POSIX function `dlsym` and call frame synthesis uses the `libffi` library to handle the low-level details.

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

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

The second type, `callspec`, describes a call frame structure. There are primitive operations primitives for creating a new `callspec`, for adding arguments, and for marking the callspec as complete and specifying 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 is used for writing each argument into the appropriate place in a buffer when performing a call.)

Finally, the `call` function takes a function address, a completed `callspec`, and callback functions that write arguments and read return values from buffers.

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

The complete implementation of `foreign` may be found in the extended version of this paper.

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 `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 stage: the user can call `foreign` interactively, as shown above.

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

Fig. 6: The C `qsort` function

Interpreting callbacks The dynamic `foreign` 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 builds a call frame from the address and the function type description. In fact, this second stage is sometimes useful independently, and it is supported as a separate operation:

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

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

Fig. 7: Using `dfunptr` to bind to `qsort`

```
val fn_of_ptr : α fn → unit ptr → α
```

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 pointers automatically:

```
let dfunptr fn = view (funptr fn) (fn_of_ptr fn) (ptr_of_fn fn)
val dfunptr : α fn → α cty
```

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

```
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 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.

3.2 Statically compiling foreign function bindings

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 (quantified in the extended version of this paper). 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.

Transforming the evaluator of Section 3.1 into a code generator can be seen as a form of *staging*, i.e. specializing the dynamic `foreign` function based on static information (i.e. the type description) in order to improve its performance when the time comes to supply the remaining arguments (i.e. 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`.

Generating C In all, three new implementations of the `FOREIGN` signature are needed. The first `FOREIGN` implementation, `GenerateC`, uses the name and the type representation passed to `foreign` to generate C code. The functor application `Bindings(GenerateC)` passes the name and type representation for `gettimeofday` to `GenerateC.foreign`, which generates a C wrapper for `gettimeofday`.

The generated C code, shown below, 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 ctypes_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);
}
```

Generating OCaml The second new `FOREIGN` implementation, `GenerateML`, generates an OCaml wrapper for `ctypes_gettimeofday`. The `ctypes_gettimeofday` function deals with low-level representations of OCaml values; the OCaml

[1] There 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`.

wrapper exposes the arguments and return types as typed values. The functor application `Bindings(GenerateML)` passes the name and type representation of `gettimeofday` to `GenerateML.foreign`, which generates an OCaml module `GeneratedML` that wraps `ctypes_gettimeofday`.

The OCaml module generated by `GenerateML` also matches the `FOREIGN` signature. The central feature of the generated code is the following `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:

```
external ctypes_gettimeofday : address → address → int
 = "ctypes_gettimeofday"

let foreign : type a. string → a cfn → a =
 fun name t → match name, t with
 | "gettimeofday",
    Fn (Pointer _, Fn (Pointer _, Returns Int)) →
     (fun x1 x2 → ctypes_gettimeofday (rawaddr x1) (rawaddr x2))
```

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 `cfn` is instantiated to a function type in the definition of the `Fn` constructor (Figure 1), 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 `Pointer`, `Int` and `Returns` constructors reveals that the right-hand side should be a function of type σ `ptr` → τ `ptr` → `int` for some types σ and τ, and this condition is met by the function expression in the generated code.

Linking the generated code The generated OCaml module `GeneratedML` serves as the third `FOREIGN` implementation; it has the following type:

```
FOREIGN with type α fn = α
```

The application `Bindings(GeneratedML)` supplies `GeneratedML` as the argument `F` of the `Bindings` functor (Figure 5). The generated `foreign` function above becomes `F.foreign` in the body of `Bindings`, and receives the name and type representation for `gettimeofday` 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 `Bindings` functor, even if they have different types.

The Trick The pattern match in the `GeneratedML.foreign` function can be seen as an instance of a binding-time improvement known in the partial evaluation community as The Trick [7]. 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 `GeneratedML.foreign` function will only ever be called with those function names and type representations used in the generation of the `GeneratedML` module. Enumerating all these possibilities as match cases results in simple non-recursive code that may easily be inlined when the `Bindings` functor is applied.

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 `Bindings` 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 `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 following binding to the standard C function `puts` uses the `string` view of Section 2 to support an argument that appears in OCaml as a `string` and in C as a `char *`:

```
let puts = foreign "puts" (string @→ 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 `Bindings` is applied:

```
let foreign : type a. string → a cfn → a =
 fun name t → match name, t with
 | "puts", Fn (View {write}, Returns Int) →
   (fun x1 → 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.

3.3 Further interpretations

Inverted bindings Section 3.1 showed how to invert the call interpreter to support callbacks; Section 3.2 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 *ocaml-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 `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 fn → (α → unit)
```

that is, a function which 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 `gettimeofday`. Once again, it suffices to apply the `Bindings` functor from Figure 5 to a suitable module. As with the staged call interpreter (Section 3.2), `Bindings` is applied multiple times – once to generate a C header and a corresponding implementation which forwards calls to OCaml callbacks, and again to produce an exporter which connects the C implementation with our OCaml functions.

As mentioned in Section 3, *ocaml-ctypes* includes a generic pretty-printing function 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.

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

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

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

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

Finally, the generated `foreign` function is reminiscent of the staged implementation of Section 3.2; it scrutinises the type representation to produce a function consumer, which passes the consumed function to `register_callback`:

```
let foreign : type a. string → a cfn → (a → unit) =
  fun name t → match name, t with
 |"gettimeofday",
   Fn (Pointer tv, Fn (Pointer tz, Returns Int)) →
   (fun f → register_callback Gettimeofday
    (fun x1 x2 → f (makeptr tv x1) (makeptr tz x2)))
```

The applied module `Bindings(GeneratedInvML)` exports a single function, `gettimeofday`, which consumes an OCaml function to be exported to C:

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

The complete code generated for the inverted binding may be found in the extended version of this paper.

Asynchronous calls Since the standard OCaml runtime has limited support for concurrency, many modern OCaml programs make use of cooperative concurrency libraries such as Lwt [16]. 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 computation with one of a pool of system threads. It is sometimes useful to wrap `detach` around calls to foreign functions.

As the signature of `detach` indicates, Lwt has a monadic interface: potentially blocking computations run in the `Lwt.t` monad. A simple generalization of the `TYPE` signature turns foreign calls into monadic computations:

```
module type TYPE' = sig
  type α comp
  val returning : α ctype → α comp fn
  (* otherwise the same as TYPE *)
end
```

(The original `TYPE` signature of Figure 5 can be recovered from `TYPE'` by substituting `type α comp = α`.) The implementation of Section 3.2 requires corresponding changes: each foreign call in the generated OCaml code is enclosed in a call to `detach`, and each generated C call includes code to release OCaml's runtime lock.

Applying `Bindings` to this Lwt-specialised implementation of `FOREIGN` builds a binding to `gettimeofday` that runs in the Lwt monad:

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

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 3.2) in which, 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. 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 8). The struct is built using the type representation constructors (Section 2) and printed using the generic pretty printer. These structs are then read and written by the generated C code in the two

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

Fig. 8: A `struct` for making cross-process calls to `gettimeofday`

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.

The extended version of this paper describes experiments that quantify the overhead of these cross-process calls.

4 Interpreting type descriptions

As Section 2 showed, the `structure`, `field` and `seal` functions (Figure 2, Section 2) can together be used to describe C `struct` types. The implementation of these operations must determine both the appropriate memory offsets of each field in the struct, and the size and alignment requirements of the whole `struct`; these numbers are determined by the order of the fields, the memory alignment requirements of each field type, and sometimes by additional compilation directives. As with `FOREIGN`, there are a variety of approaches to implementing the `STRUCTURE` interface.

Computing layout information The simplest approach to implementing `STRUCTURE` is to give implementations of `field` and `seal` that simply compute the appropriate layout directly.

```
module Types(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
```

Fig. 9: *timeval* layout, abstractly

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. The extended version of this paper gives the full code.

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__` direction), and attempting to replicate these quickly becomes unmanageable. Second, some libraries, (e.g. `libuv`), define structs with interspersed internal fields which vary both across platforms and across versions. Replicating this variation in the bindings quickly leads to unmaintainable code.

Retrieving layout information These drawbacks can be avoided with an alternative implementation of `STRUCTURE` 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 3.2) 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 which compute the layout into functions which 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 Types structure (Figure 9) is applied to a module `Generate_C` to produce a C program which retrieves layout information with calls to `offsetof` and `sizeof`:

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

Compiling and running the C program produces an OCaml module `Types_impl` which satisfies the `TYPE` signature, and which contains implementations of `field` and `seal` specialized to the `struct`s and fields of the `Types` module:

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

The application `Types(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.

5 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 [4], 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 [5] 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 which 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 [15, 3]. 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.

Mapping codes to types and their interpretations by abstracting over a parameterised type constructor is a well-known technique in the generic programming community [17, 6]. Hinze [10] describes a library for generic programming in Haskell with a type class that corresponds quite closely to the `TYPE` signature of Section 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 [11] corresponds even more directly to this aspect of the design presented here.

References

[1] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 1–20, 2003.

[2] David M. Beazley. SWIG : An easy to use tool for integrating scripting languages with C and C++. In *USENIX Tcl/Tk Workshop*, 1996.

[3] Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. 10(4):265–289, 2003.

[4] Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". *Electronic Notes in Theoretical Computer Science*, 59(1):36–52, 2001.

[5] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, September 2009.

[6] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. Haskell '02, pages 90–104, New York, NY, USA, 2002. ACM.

[7] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does the trick. *ACM Trans. Program. Lang. Syst.*, 18(6):730–751, November 1996.

[8] Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. PLDI '05, pages 62–72, New York, NY, USA, 2005. ACM.

[9] Jeremy Gibbons. Datatype-generic programming. In *Datatype-Generic Programming*, volume 4719, pages 1–71. Springer Berlin Heidelberg, 2007.

[10] Ralf Hinze. Generics for the masses. *J. Funct. Program.*, 16(4-5), July 2006.

[11] Vesa A.J. Karvonen. Generics for the working ML'er. ML '07. ACM, 2007.

[12] Goh Kondoh and Tamiya Onodera. Finding bugs in Java Native Interface programs. ISSTA '08, pages 109–118. ACM, 2008.

[13] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system (release 3.12): Documentation and user's manual.* INRIA, July 2011.

[14] Siliang Li and Gang Tan. Finding reference-counting errors in Python/C programs with affine analysis. In *ECOOP 2014*, pages 80–104. 2014.

[15] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf type theory: an introduction.* Clarendon, 1990.

[16] Jérôme Vouillon. Lwt: A cooperative thread library. ML '08. ACM, 2008.

[17] Zhe Yang. Encoding types in ML-like languages. ICFP '98. ACM, 1998.