23 Foreign Function Interface

This chapter includes contributions from Jeremy Yallop.

OCaml has several options available to interact with non-OCaml code. The compiler can link with external system libraries via C code and also can produce standalone native object files that can be embedded within other non-OCaml applications.

The mechanism by which code in one programming language can invoke routines in a different programming language is called a *foreign function interface*. This chapter will:

- Show how to call routines in C libraries directly from your OCaml code
- Teach you how to build higher-level abstractions in OCaml from the low-level C bindings
- Work through some full examples for binding a terminal interface and UNIX date/-time functions

The simplest foreign function interface in OCaml doesn't even require you to write any C code at all! The Ctypes library lets you define the C interface in pure OCaml, and the library then takes care of loading the C symbols and invoking the foreign function call.

Let's dive straight into a realistic example to show you how the library looks. We'll create a binding to the Ncurses terminal toolkit, as it's widely available on most systems and doesn't have any complex dependencies.

Installing the Ctypes Library

If you want to use Ctypes interactively, you'll also need to install the `libffi` library as a prerequisite to using Ctypes. It's a fairly popular library and should be available in your OS package manager. If you're using opam 2.1 or higher, it will prompt you to install it automatically when you install `ctypes-foreign`.

```
$ opam install ctypes ctypes-foreign
$ utop
# require "ctypes-foreign" ;;
```

You'll also need the Ncurses library for the first example. This comes preinstalled on many operating systems such as macOS, and Debian Linux provides it as the `libncurses5-dev` package.

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

23.1 Example: A Terminal Interface

Ncurses is a library to help build terminal-independent text interfaces in a reasonably efficient way. It's used in console mail clients like Mutt and Pine, and console web browsers such as Lynx.

The full C interface is quite large and is explained in the online documentation[1]. We'll just use the small excerpt, since we just want to demonstrate Ctypes in action:

```
typedef struct _win_st WINDOW;
typedef unsigned int chtype;

WINDOW *initscr   (void);
WINDOW *newwin    (int, int, int, int);
void    endwin    (void);
void    refresh   (void);
void    wrefresh  (WINDOW *);
void    addstr (const char *);
int     mvwaddch  (WINDOW *, int, int, const chtype);
void    mvwaddstr (WINDOW *, int, int, char *);
void    box (WINDOW *, chtype, chtype);
int     cbreak (void);
```

The Ncurses functions either operate on the current pseudoterminal or on a window that has been created via `newwin`. The `WINDOW` structure holds the internal library state and is considered abstract outside of Ncurses. Ncurses clients just need to store the pointer somewhere and pass it back to Ncurses library calls, which in turn dereference its contents.

Note that there are over 200 library calls in Ncurses, so we're only binding a select few for this example. The `initscr` and `newwin` create `WINDOW` pointers for the global and subwindows, respectively. The `mvwaddrstr` takes a window, x/y offsets, and a string and writes to the screen at that location. The terminal is only updated after `refresh` or `wrefresh` are called.

Ctypes provides an OCaml interface that lets you map these C functions to equivalent OCaml functions. The library takes care of converting OCaml function calls and arguments into the C calling convention, invoking the foreign call within the C library and finally returning the result as an OCaml value.

Let's begin by defining the basic values we need, starting with the `WINDOW` state pointer:

```
open Ctypes

type window = unit ptr

let window : window typ = ptr void
```

We don't know the internal representation of the window pointer, so we treat it as a C void pointer. We'll improve on this later on in the chapter, but it's good enough for now. The second statement defines an OCaml value that represents the `WINDOW` C pointer. This value is used later in the Ctypes function definitions:

[1] http://www.gnu.org/software/ncurses/

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

```
open Foreign

let initscr = foreign "initscr" (void @-> returning window)
```

That's all we need to invoke our first function call to `initscr` to initialize the terminal. The `foreign` function accepts two parameters:

- The C function call name, which is looked up using the `dlsym` POSIX function.
- A value that defines the complete set of C function arguments and its return type. The `@->` operator adds an argument to the C parameter list, and `returning` terminates the parameter list with the return type.

The remainder of the Ncurses binding simply expands on these definitions:

```
let newwin =
  foreign "newwin" (int @-> int @-> int @-> int @-> returning window)

let endwin = foreign "endwin" (void @-> returning void)
let refresh = foreign "refresh" (void @-> returning void)
let wrefresh = foreign "wrefresh" (window @-> returning void)
let addstr = foreign "addstr" (string @-> returning void)

let mvwaddch =
  foreign
    "mvwaddch"
    (window @-> int @-> int @-> char @-> returning void)

let mvwaddstr =
  foreign
    "mvwaddstr"
    (window @-> int @-> int @-> string @-> returning void)

let box = foreign "box" (window @-> char @-> char @-> returning void)
let cbreak = foreign "cbreak" (void @-> returning int)
```

These definitions are all straightforward mappings from the C declarations in the Ncurses header file. Note that the `string` and `int` values here are nothing to do with OCaml type declarations; instead, they are values that come from opening the `Ctypes` module at the top of the file.

Most of the parameters in the Ncurses example represent fairly simple scalar C types, except for `window` (a pointer to the library state) and `string`, which maps from OCaml strings that have a specific length onto C character buffers whose length is defined by a terminating null character that immediately follows the string data.

The module signature for `ncurses.mli` looks much like a normal OCaml signature. You can infer it directly from the `ncurses.ml` by running a command called `ocaml-print-intf`, which you can install with opam.

```
$ ocaml-print-intf ncurses.ml
type window = unit Ctypes.ptr
val window : window Ctypes.typ
val initscr : unit -> window
val newwin : int -> int -> int -> int -> window
val endwin : unit -> unit
val refresh : unit -> unit
```

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

```
val wrefresh : window -> unit
val addstr : string -> unit
val mvwaddch : window -> int -> int -> char -> unit
val mvwaddstr : window -> int -> int -> string -> unit
val box : window -> char -> char -> unit
val cbreak : unit -> int
```

The `ocaml-print-intf` tool examines the default signature inferred by the compiler for a module file and prints it out as human-readable output. You can copy this into a corresponding `mli` file and customize it to improve its safety for external callers by making some of its internals more abstract.

Here's the customized `ncurses.mli` interface that we can safely use from other libraries:

```
type window

val window : window Ctypes.typ
val initscr : unit -> window
val endwin : unit -> unit
val refresh : unit -> unit
val wrefresh : window -> unit
val newwin : int -> int -> int -> int -> window
val mvwaddch : window -> int -> int -> char -> unit
val addstr : string -> unit
val mvwaddstr : window -> int -> int -> string -> unit
val box : window -> char -> char -> unit
val cbreak : unit -> int
```

Note that the `window` type is now abstract in the signature, to ensure that window pointers can only be constructed via the `Ncurses.initscr` function. This prevents void pointers obtained from other sources from being mistakenly passed to an Ncurses library call.

Now compile a "hello world" terminal drawing program to tie this all together:

```
open Ncurses

let () =
  let main_window = initscr () in
  ignore (cbreak ());
  let small_window = newwin 10 10 5 5 in
  mvwaddstr main_window 1 2 "Hello";
  mvwaddstr small_window 2 2 "World";
  box small_window '\000' '\000';
  refresh ();
  Unix.sleep 1;
  wrefresh small_window;
  Unix.sleep 5;
  endwin ()
```

The `hello` executable is compiled by linking with the `ctypes-foreign` package. We also add in a `(flags)` directive to instruct the compiler to link in the system `ncurses` C library to the executable. If you do not specify the C library in the dune file, then the program may build successfully, but attempting to invoke the executable will fail as not all of the dependencies will be available.

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

```
(executable
  (name      hello)
  (libraries ctypes-foreign)
  (flags     :standard -cclib -lncurses))
```

And now we can build it with Dune.

```
$ dune build hello.exe
```

Running `hello.exe` should now display a Hello World in your terminal!

Ctypes wouldn't be very useful if it were limited to only defining simple C types, of course. It provides full support for C pointer arithmetic, pointer conversions, and reading and writing through pointers, using OCaml functions as function pointers to C code, as well as struct and union definitions.

We'll go over some of these features in more detail for the remainder of the chapter by using some POSIX date functions as running examples.

Linking Modes: libffi and Stub Generation

The core of ctypes is a set of OCaml combinators for describing the structure of C types (numeric types, arrays, pointers, structs, unions and functions). You can then use these combinators to describe the types of the C functions that you want to call. There are two entirely distinct ways to actually link to the system libraries that contain the function definitions: *dynamic linking* and *stub generation*.

The `ctypes-foreign` package used in this chapter uses the low-level `libffi` library to dynamically open C libraries, search for the relevant symbols for the function call being invoked, and marshal the function parameters according to the operating system's application binary interface (ABI). While much of this happens behind-the-scenes and permits convenient interactive programming while developing bindings, it is not always the solution you want to use in production.

The `ctypes-cstubs` package provides an alternative mechanism to shift much of the linking work to be done once at build time, instead of doing it on every invocation of the function. It does this by taking the *same* OCaml binding descriptions, and generating intermediate C source files that contain the corresponding C/OCaml glue code. When these are compiled with a normal dune build, the generated C code is treated just as any handwritten code might be, and compiled against the system header files. This allows certain C values to be used that cannot be dynamically probed (e.g. preprocessor macro definitions), and can also catch definition errors if there is a C header mismatch at compile time.

C rarely makes life easier though. There are some definitions that cannot be entirely expressed as static C code (e.g. dynamic function pointers), and those require the use of `ctypes-foreign` (and `libffi`). Using ctypes does make it possible to share the majority of definitions across both linking modes, all while avoiding writing C code directly.

While we do not cover the details of C stub generation further in this chapter, you can read more about how to use this mode in the "Dealing with foreign libraries" chapter in the dune manual.

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

23.2 Basic Scalar C Types

First, let's look at how to define basic scalar C types. Every C type is represented by an OCaml equivalent via the single type definition:

```
type 'a typ
```

`Ctypes.typ` is the type of values that represents C types to OCaml. There are two types associated with each instance of `typ`:

- The C type used to store and pass values to the foreign library.
- The corresponding OCaml type. The `'a` type parameter contains the OCaml type such that a value of type `t typ` is used to read and write OCaml values of type `t`.

There are various other uses of `typ` values within Ctypes, such as:

- Constructing function types for binding native functions
- Constructing pointers for reading and writing locations in C-managed storage
- Describing component fields of structures, unions, and arrays

Here are the definitions for most of the standard C99 scalar types, including some platform-dependent ones:

```
val void      : unit typ
val char      : char typ
val schar     : int typ
val short     : int typ
val int       : int typ
val long      : long typ
val llong     : llong typ
val nativeint : nativeint typ

val int8_t    : int typ
val int16_t   : int typ
val int32_t   : int32 typ
val int64_t   : int64 typ
val uchar     : uchar typ
val uint8_t   : uint8 typ
val uint16_t  : uint16 typ
val uint32_t  : uint32 typ
val uint64_t  : uint64 typ
val size_t    : size_t typ
val ushort    : ushort typ
val uint      : uint typ
val ulong     : ulong typ
val ullong    : ullong typ

val float     : float typ
val double    : float typ

val complex32 : Complex.t typ
val complex64 : Complex.t typ
```

These values are all of type `'a typ`, where the value name (e.g., `void`) tells you the C type and the `'a` component (e.g., `unit`) is the OCaml representation of that C

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

type. Most of the mappings are straightforward, but some of them need a bit more explanation:

- Void values appear in OCaml as the `unit` type. Using `void` in an argument or result type specification produces an OCaml function that accepts or returns `unit`. Dereferencing a pointer to `void` is an error, as in C, and will raise the `IncompleteType` exception.
- The C `size_t` type is an alias for one of the unsigned integer types. The actual size and alignment requirements for `size_t` varies between platforms. Ctypes provides an OCaml `size_t` type that is aliased to the appropriate integer type.
- OCaml only supports double-precision floating-point numbers, and so the C `float` and `double` types both map onto the OCaml `float` type, and the C `float complex` and `double complex` types both map onto the OCaml double-precision `Complex.t` type.

23.3 Pointers and Arrays

Pointers are at the heart of C, so they are necessarily part of Ctypes, which provides support for pointer arithmetic, pointer conversions, reading and writing through pointers, and passing and returning pointers to and from functions.

We've already seen a simple use of pointers in the Ncurses example. Let's start a new example by binding the following POSIX functions:

```
time_t time(time_t *);
double difftime(time_t, time_t);
char *ctime(const time_t *timep);
```

The `time` function returns the current calendar time and is a simple start. The first step is to open some of the Ctypes modules:

`Ctypes` The `Ctypes` module provides functions for describing C types in OCaml.

`PosixTypes` The `PosixTypes` module includes some extra POSIX-specific types (such as `time_t`).

`Foreign` The `Foreign` module exposes the `foreign` function that makes it possible to invoke C functions.

With these opens in place, we can now create a binding to `time` directly from the toplevel.

```
# #require "ctypes-foreign";;
# #require "ctypes.top";;
# open Core;;
# open Ctypes;;
# open PosixTypes ;;
# open Foreign;;
# let time = foreign "time" (ptr time_t @-> returning time_t);;
val time : time_t Ctypes_static.ptr -> time_t = <fun>
```

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

The `foreign` function is the main link between OCaml and C. It takes two arguments: the name of the C function to bind, and a value describing the type of the bound function. In the `time` binding, the function type specifies one argument of type `ptr time_t` and a return type of `time_t`.

We can now call `time` immediately in the same toplevel. The argument is actually optional, so we'll just pass a null pointer that has been coerced into becoming a null pointer to `time_t`:

```
# let cur_time = time (from_voidp time_t null);;
val cur_time : time_t = <abstr>
```

Since we're going to call `time` a few times, let's create a wrapper function that passes the null pointer through:

```
# let time' () = time (from_voidp time_t null);;
val time' : unit -> time_t = <fun>
```

Since `time_t` is an abstract type, we can't actually do anything useful with it directly. We need to bind a second function to do anything useful with the return values from `time`. We'll move on to `difftime`; the second C function in our prototype list:

```
# let difftime = foreign "difftime" (time_t @-> time_t @-> returning
    double);;
val difftime : time_t -> time_t -> float = <fun>
```

Here's the resulting function `difftime` in action.

```
# let delta =
    let t1 = time' () in
    Unix.sleep 2;
    let t2 = time' () in
    difftime t2 t1;;
val delta : float = 2.
```

The binding to `difftime` above is sufficient to compare two `time_t` values.

23.3.1 Allocating Typed Memory for Pointers

Let's look at a slightly less trivial example where we pass a nonnull pointer to a function. Continuing with the theme from earlier, we'll bind to the `ctime` function, which converts a `time_t` value to a human-readable string:

```
# let ctime = foreign "ctime" (ptr time_t @-> returning string);;
val ctime : time_t Ctypes_static.ptr -> string = <fun>
```

The binding is continued in the toplevel to add to our growing collection. However, we can't just pass the result of `time` to `ctime`:

```
# ctime (time' ());;
Line 1, characters 7-17:
Error: This expression has type time_t but an expression was expected
    of type
         time_t Ctypes_static.ptr = (time_t, [ `C ]) pointer
```

This is because `ctime` needs a pointer to the `time_t` rather than passing it by value. We thus need to allocate some memory for the `time_t` and obtain its memory address:

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

```
# let t_ptr = allocate time_t (time' ());;
...
```

The `allocate` function takes the type of the memory to be allocated and the initial value and it returns a suitably typed pointer. We can now call `ctime` passing the pointer as an argument:

```
# ctime t_ptr;;
...
```

23.3.2 Using Views to Map Complex Values

While scalar types typically have a 1:1 representation, other C types require extra work to convert them into OCaml. Views create new C type descriptions that have special behavior when used to read or write C values.

We've already used one view in the definition of `ctime` earlier. The `string` view wraps the C type `char *` (written in OCaml as `ptr char`) and converts between the C and OCaml string representations each time the value is written or read.

Here is the type signature of the `Ctypes.view` function:

```
val view :
  read:('a -> 'b) ->
  write:('b -> 'a) ->
  'a typ -> 'b typ
```

Ctypes has some internal low-level conversion functions that map between an OCaml `string` and a C character buffer by copying the contents into the respective data structure. They have the following type signature:

```
val string_of_char_ptr : char ptr -> string
val char_ptr_of_string : string -> char ptr
```

Given these functions, the definition of the `Ctypes.string` value that uses views is quite simple:

```
let string =
  view (char ptr)
    ~read:string_of_char_ptr
    ~write:char_ptr_of_string
```

The type of this `string` function is a normal `typ` with no external sign of the use of the view function:

```
val string    : string typ
```

OCaml Strings Versus C Character Buffers

Although OCaml strings may look like C character buffers from an interface perspective, they're very different in terms of their memory representations.

OCaml strings are stored in the OCaml heap with a header that explicitly defines their length. C buffers are also fixed-length, but by convention, a C string is terminated

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

by a null (a `\0` byte) character. The C string functions calculate their length by scanning the buffer until the first null character is encountered.

This means that you need to be careful that OCaml strings that you pass to C functions don't contain any null values, since the first occurrence of a null character will be treated as the end of the C string. Ctypes also defaults to a *copying* interface for strings, which means that you shouldn't use them when you want the library to mutate the buffer in-place. In that situation, use the Ctypes `Bigarray` support to pass memory by reference instead.

23.4 Structs and Unions

The C constructs `struct` and `union` make it possible to build new types from existing types. Ctypes contains counterparts that work similarly.

23.4.1 Defining a Structure

Let's improve the timer function that we wrote earlier. The POSIX function `gettimeofday` retrieves the time with microsecond resolution. The signature of `gettimeofday` is as follows, including the structure definitions:

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

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

Using Ctypes, we can describe this type as follows in our toplevel, continuing on from the previous definitions:

```
# type timeval;;
type timeval
# let timeval : timeval structure typ = structure "timeval";;
val timeval : timeval structure typ =
  Ctypes_static.Struct
   {Ctypes_static.tag = "timeval";
    spec = Ctypes_static.Incomplete {Ctypes_static.isize = 0}; fields
    = []}
```

The first command defines a new OCaml type `timeval` that we'll use to instantiate the OCaml version of the struct. This is a *phantom type* that exists only to distinguish the underlying C type from other pointer types. The particular `timeval` structure now has a distinct type from other structures we define elsewhere, which helps to avoid getting them mixed up.

The second command calls `structure` to create a fresh structure type. At this point, the structure type is incomplete: we can add fields but cannot yet use it in `foreign` calls or use it to create values.

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

23.4.2 Adding Fields to Structures

The `timeval` structure definition still doesn't have any fields, so we need to add those next:

```
# let tv_sec  = field timeval "tv_sec" long;;
val tv_sec : (Signed.long, timeval structure) field =
  {Ctypes_static.ftype = Ctypes_static.Primitive
     Ctypes_primitive_types.Long;
   foffset = 0; fname = "tv_sec"}
# let tv_usec = field timeval "tv_usec" long;;
val tv_usec : (Signed.long, timeval structure) field =
  {Ctypes_static.ftype = Ctypes_static.Primitive
     Ctypes_primitive_types.Long;
   foffset = 8; fname = "tv_usec"}
# seal timeval;;
- : unit = ()
```

The `field` function appends a field to the structure, as shown with `tv_sec` and `tv_usec`. Structure fields are typed accessors that are associated with a particular structure, and they correspond to the labels in C.

Every field addition mutates the structure variable and records a new size (the exact value of which depends on the type of the field that was just added). Once we `seal` the structure, we will be able to create values using it, but adding fields to a sealed structure is an error.

23.4.3 Incomplete Structure Definitions

Since `gettimeofday` needs a `struct timezone` pointer for its second argument, we also need to define a second structure type:

```
# type timezone;;
type timezone
# let timezone : timezone structure typ = structure "timezone";;
val timezone : timezone structure typ =
  Ctypes_static.Struct
   {Ctypes_static.tag = "timezone";
    spec = Ctypes_static.Incomplete {Ctypes_static.isize = 0}; fields
    = []}
```

We don't ever need to create `struct timezone` values, so we can leave this struct as incomplete without adding any fields or sealing it. If you ever try to use it in a situation where its concrete size needs to be known, the library will raise an `IncompleteType` exception.

We're finally ready to bind to `gettimeofday` now:

```
# let gettimeofday = foreign "gettimeofday" ~check_errno:true
    (ptr timeval @-> ptr timezone @-> returning int);;
val gettimeofday :
  timeval structure Ctypes_static.ptr ->
  timezone structure Ctypes_static.ptr -> int = <fun>
```

There's one other new feature here: the `returning_checking_errno` function behaves like `returning`, except that it checks whether the bound C function modifies

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

the C error flag. Changes to `errno` are mapped into OCaml exceptions and raise a `Unix.Unix_error` exception just as the standard library functions do.

As before, we can create a wrapper to make `gettimeofday` easier to use. The functions `make`, `addr`, and `getf` create a structure value, retrieve the address of a structure value, and retrieve the value of a field from a structure.

```
# let gettimeofday' () =
    let tv = make timeval in
    ignore (gettimeofday (addr tv) (from_voidp timezone null) : int);
    let secs = Signed.Long.to_int (getf tv tv_sec) in
    let usecs = Signed.Long.to_int (getf tv tv_usec) in
    Float.of_int secs +. Float.of_int usecs /. 1_000_000.0;;
val gettimeofday' : unit -> float = <fun>
```

And we can now call that function to get the current time.

```
# gettimeofday' ();;
- : float = 1650045389.278065
```

Recap: a Time-Printing Command

We built up a lot of bindings in the previous section, so let's recap them with a complete example that ties it together with a command-line frontend:

```
open Core
open Ctypes
open PosixTypes
open Foreign

let time = foreign "time" (ptr time_t @-> returning time_t)

let difftime =
  foreign "difftime" (time_t @-> time_t @-> returning double)

let ctime = foreign "ctime" (ptr time_t @-> returning string)

type timeval

let timeval : timeval structure typ = structure "timeval"
let tv_sec = field timeval "tv_sec" long
let tv_usec = field timeval "tv_usec" long
let () = seal timeval

type timezone

let timezone : timezone structure typ = structure "timezone"

let gettimeofday =
  foreign
    "gettimeofday"
    ~check_errno:true
    (ptr timeval @-> ptr timezone @-> returning int)

let time' () = time (from_voidp time_t null)

let gettimeofday' () =
```

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

```
  let tv = make timeval in
  ignore (gettimeofday (addr tv) (from_voidp timezone null) : int);
  let secs = Signed.Long.to_int (getf tv tv_sec) in
  let usecs = Signed.Long.to_int (getf tv tv_usec) in
  Float.of_int secs +. (Float.of_int usecs /. 1_000_000.)

let float_time () = printf "%f%!\n" (gettimeofday' ())

let ascii_time () =
  let t_ptr = allocate time_t (time' ()) in
  printf "%s%!" (ctime t_ptr)

let () =
  Command.basic
    ~summary:"Display the current time in various formats"
    (let%map_open.Command human =
       flag "-a" no_arg ~doc:" Human-readable output format"
     in
     if human then ascii_time else float_time)
  |> Command.run
```

This can be compiled and run in the usual way:

```
(executable
  (name      datetime)
  (preprocess (pps ppx_jane))
  (libraries core ctypes-foreign))
```

```
$ dune build datetime.exe
$ ./_build/default/datetime.exe
1633964258.014484
$ ./_build/default/datetime.exe -a
Mon Oct 11 15:57:38 2021
```

Why Do We Need to Use returning?

The alert reader may be curious about why all these function definitions have to be terminated by `returning`:

```
(* correct types *)
val time: ptr time_t @-> returning time_t
val difftime: time_t @-> time_t @-> returning double
```

The `returning` function may appear superfluous here. Why couldn't we simply give the types as follows?

```
(* incorrect types *)
val time: ptr time_t @-> time_t
val difftime: time_t @-> time_t @-> double
```

The reason involves higher types and two differences between the way that functions are treated in OCaml and C. Functions are first-class values in OCaml, but not in C. For example, in C it is possible to return a function pointer from a function, but not to return an actual function.

Secondly, OCaml functions are typically defined in a curried style. The signature of a two-argument function is written as follows:

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

```
val curried : int -> int -> int
```

but this really means:

```
val curried : int -> (int -> int)
```

and the arguments can be supplied one at a time to create a closure. In contrast, C functions receive their arguments all at once. The equivalent C function type is the following:

```
int uncurried_C(int, int);
```

and the arguments must always be supplied together:

```
uncurried_C(3, 4);
```

A C function that's written in curried style looks very different:

```
/* A function that accepts an int, and returns a function
   pointer that accepts a second int and returns an int. */
typedef int (function_t)(int);
function_t *curried_C(int);

/* supply both arguments */
curried_C(3)(4);

/* supply one argument at a time */
function_t *f = curried_C(3); f(4);
```

The OCaml type of `uncurried_C` when bound by Ctypes is `int -> int -> int`: a two-argument function. The OCaml type of `curried_C` when bound by `ctypes` is `int -> (int -> int)`: a one-argument function that returns a one-argument function.

In OCaml, of course, these types are absolutely equivalent. Since the OCaml types are the same but the C semantics are quite different, we need some kind of marker to distinguish the cases. This is the purpose of `returning` in function definitions.

23.4.4 Defining Arrays

Arrays in C are contiguous blocks of the same type of value. Any of the basic types defined previously can be allocated as blocks via the `Array` module:

```
module Array : sig
  type 'a t = 'a array

  val get : 'a t -> int -> 'a
  val set : 'a t -> int -> 'a -> unit
  val of_list : 'a typ -> 'a list -> 'a t
  val to_list : 'a t -> 'a list
  val length : 'a t -> int
  val start : 'a t -> 'a ptr
  val from_ptr : 'a ptr -> int -> 'a t
  val make : 'a typ -> ?initial:'a -> int -> 'a t
end
```

The array functions are similar to those in the standard library `Array` module except

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

that they operate on arrays stored using the flat C representation rather than the OCaml representation described in Chapter 24 (Memory Representation of Values).

As with standard OCaml arrays, the conversion between arrays and lists requires copying the values, which can be expensive for large data structures. Notice that you can also convert an array into a `ptr` pointer to the head of the underlying buffer, which can be useful if you need to pass the pointer and size arguments separately to a C function.

Unions in C are named structures that can be mapped onto the same underlying memory. They are also fully supported in Ctypes, but we won't go into more detail here.

Pointer Operators for Dereferencing and Arithmetic

Ctypes defines a number of operators that let you manipulate pointers and arrays just as you would in C. The Ctypes equivalents do have the benefit of being more strongly typed, of course.

- `!@ p` will dereference the pointer `p`.
- `p <-@ v` will write the value `v` to the address `p`.
- `p +@ n` computes the address of the `n`th next element, if `p` points to an array element.
- `p -@ n` computes the address of the `n`th previous element, if `p` points to an array element.

There are also other useful non-operator functions available (see the Ctypes documentation), such as pointer differencing and comparison.

23.5 Passing Functions to C

It's also straightforward to pass OCaml function values to C. The C standard library function `qsort` sorts arrays of elements using a comparison function passed in as a function pointer. The signature for `qsort` is:

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));
```

C programmers often use `typedef` to make type definitions involving function pointers easier to read. Using a typedef, the type of `qsort` looks a little more palatable:

```
typedef int(compare_t)(const void *, const void *);

void qsort(void *base, size_t nmemb, size_t size, compare_t *);
```

This also happens to be a close mapping to the corresponding Ctypes definition. Since type descriptions are regular values, we can just use `let` in place of `typedef` and end up with working OCaml bindings to `qsort`:

```
# open Core open Ctypes open PosixTypes open Foreign open
    Ctypes_static;;
# let compare_t = ptr void @-> ptr void @-> returning int;;
```

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

```
val compare_t : (unit Ctypes_static.ptr -> unit Ctypes_static.ptr ->
    int) fn =
  Function (Pointer Void,
   Function (Pointer Void, Returns (Primitive
    Ctypes_primitive_types.Int)))
# let qsort =
    foreign "qsort"
      (ptr void @-> size_t @-> size_t @-> funptr compare_t
       @-> returning void);;
val qsort :
  unit Ctypes_static.ptr ->
  size_t ->
  size_t -> (unit Ctypes_static.ptr -> unit Ctypes_static.ptr -> int)
    -> unit =
  <fun>
```

We only use `compare_t` once (in the `qsort` definition), so you can choose to inline it in the OCaml code if you prefer. As the type shows, the resulting `qsort` value is a higher-order function, since the fourth argument is itself a function.

Arrays created using Ctypes have a richer runtime structure than C arrays, so we don't need to pass size information around. Furthermore, we can use OCaml polymorphism in place of the unsafe `void ptr` type.

23.5.1 Example: A Command-Line Quicksort

The following is a command-line tool that uses the `qsort` binding to sort all of the integers supplied on the standard input:

```
open Core
open Ctypes
open PosixTypes
open Foreign

let compare_t = ptr void @-> ptr void @-> returning int

let qsort =
  foreign
    "qsort"
    (ptr void
    @-> size_t
    @-> size_t
    @-> funptr compare_t
    @-> returning void)

let qsort' cmp arr =
  let open Unsigned.Size_t in
  let ty = CArray.element_type arr in
  let len = of_int (CArray.length arr) in
  let elsize = of_int (sizeof ty) in
  let start = to_voidp (CArray.start arr) in
  let compare l r = cmp !@(from_voidp ty l) !@(from_voidp ty r) in
  qsort start len elsize compare

let sort_stdin () =
```

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

```
  let array =
    In_channel.input_line_exn In_channel.stdin
    |> String.split ~on:' '
    |> List.map ~f:int_of_string
    |> CArray.of_list int
  in
  qsort' Int.compare array;
  CArray.to_list array
  |> List.map ~f:Int.to_string
  |> String.concat ~sep:" "
  |> print_endline

let () =
  Command.basic_spec
    ~summary:"Sort integers on standard input"
    Command.Spec.empty
    sort_stdin
  |> Command.run
```

Compile it in the usual way with *dune* and test it against some input data, and also build the inferred interface so we can examine it more closely:

```
(executable
  (name      qsort)
  (libraries core ctypes-foreign))
```

```
$ echo 2 4 1 3 | dune exec ./qsort.exe
1 2 3 4
```

The inferred mli shows us the types of the raw `qsort` binding and also the `qsort'` wrapper function.

```
$ ocaml-print-intf qsort.ml
val compare_t :
  (unit Ctypes_static.ptr -> unit Ctypes_static.ptr -> int) Ctypes.fn
val qsort :
  unit Ctypes_static.ptr ->
  PosixTypes.size_t ->
  PosixTypes.size_t ->
  (unit Ctypes_static.ptr -> unit Ctypes_static.ptr -> int) -> unit
val qsort' : ('a -> 'a -> int) -> 'a Ctypes.CArray.t -> unit
val sort_stdin : unit -> unit
```

The `qsort'` wrapper function has a much more canonical OCaml interface than the raw binding. It accepts a comparator function and a Ctypes array, and returns unit.

Using `qsort'` to sort arrays is straightforward. Our example code reads the standard input as a list, converts it to a C array, passes it through qsort, and outputs the result to the standard output. Again, remember to not confuse the `Ctypes.Array` module with the `Core.Array` module: the former is in scope since we opened `Ctypes` at the start of the file.

Lifetime of Allocated Ctypes

Values allocated via Ctypes (i.e., using `allocate`, `Array.make`, and so on) will not be garbage-collected as long as they are reachable from OCaml values. The system

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

memory they occupy is freed when they do become unreachable, via a finalizer function registered with the garbage collector (GC).

The definition of reachability for Ctypes values is a little different from conventional OCaml values, though. The allocation functions return an OCaml-managed pointer to the value, and as long as some derivative pointer is still reachable by the GC, the value won't be collected.

"Derivative" means a pointer that's computed from the original pointer via arithmetic, so a reachable reference to an array element or a structure field protects the whole object from collection.

A corollary of the preceding rule is that pointers written into the C heap don't have any effect on reachability. For example, if you have a C-managed array of pointers to structs, then you'll need some additional way of keeping the structs themselves around to protect them from collection. You could achieve this via a global array of values on the OCaml side that would keep them live until they're no longer needed.

Functions passed to C have similar considerations regarding lifetime. On the OCaml side, functions created at runtime may be collected when they become unreachable. As we've seen, OCaml functions passed to C are converted to function pointers, and function pointers written into the C heap have no effect on the reachability of the OCaml functions they reference. With `qsort` things are straightforward, since the comparison function is only used during the call to `qsort` itself. However, other C libraries may store function pointers in global variables or elsewhere, in which case you'll need to take care that the OCaml functions you pass to them aren't prematurely garbage-collected.

23.6 Learning More About C Bindings

The Ctypes distribution[2] contains a number of larger-scale examples, including:

- Bindings to the POSIX `fts` API, which demonstrates C callbacks more comprehensively
- A more complete Ncurses binding than the example we opened the chapter with
- A comprehensive test suite that covers the complete library, and can provide useful snippets for your own bindings

This chapter hasn't really needed you to understand the innards of OCaml at all. Ctypes does its best to make function bindings easy, but the rest of this part will also fill you in about interactions with OCaml memory layout in Chapter 24 (Memory Representation of Values) and automatic memory management in Chapter 25 (Understanding the Garbage Collector).

Ctypes gives OCaml programs access to the C representation of values, shielding you from the details of the OCaml value representation, and introduces an abstraction layer that hides the details of foreign calls. While this covers a wide variety of situations,

[2] http://github.com/ocamllabs/ocaml-ctypes

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press

it's sometimes necessary to look behind the abstraction to obtain finer control over the details of the interaction between the two languages.

You can find more information about the C interface in several places:

- The standard OCaml foreign function interface allows you to glue OCaml and C together from the other side of the boundary, by writing C functions that operate on the OCaml representation of values. You can find details of the standard interface in the OCaml manual[3] and in the book *Developing Applications with Objective Caml*[4].
- Florent Monnier maintains an excellent online tutorial[5] that provides examples of how to call OCaml functions from C. This covers a wide variety of OCaml data types and also more complex callbacks between C and OCaml.

23.6.1 Struct Memory Layout

The C language gives implementations a certain amount of freedom in choosing how to lay out structs in memory. There may be padding between members and at the end of the struct, in order to satisfy the memory alignment requirements of the host platform. Ctypes uses platform-appropriate size and alignment information to replicate the struct layout process. OCaml and C will have consistent views about the layout of the struct as long as you declare the fields of a struct in the same order and with the same types as the C library you're binding to.

However, this approach can lead to difficulties when the fields of a struct aren't fully specified in the interface of a library. The interface may list the fields of a structure without specifying their order, or make certain fields available only on certain platforms, or insert undocumented fields into struct definitions for performance reasons. For example, the `struct timeval` definition used in this chapter accurately describes the layout of the struct on common platforms, but implementations on some more unusual architectures include additional padding members that will lead to strange behavior in the examples.

The Cstubs subpackage of Ctypes addresses this issue. Rather than simply assuming that struct definitions given by the user accurately reflect the actual definitions of structs used in C libraries, Cstubs generates code that uses the C library headers to discover the layout of the struct. The good news is that the code that you write doesn't need to change much. Cstubs provides alternative implementations of the `field` and `seal` functions that you've already used to describe `struct timeval`; instead of computing member offsets and sizes appropriate for the platform, these implementations obtain them directly from C.

The details of using Cstubs are available in the online documentation[6], along with instructions on integration with `autoconf` platform portability instructions.

[3] https://ocaml.org/manual/intfc.html
[4] http://caml.inria.fr/pub/docs/oreilly-book/ocaml-ora-book.pdf
[5] http://decapode314.free.fr/ocaml/ocaml-wrapping-c.html
[6] https://ocamllabs.github.io/ocaml-ctypes

https://doi.org/10.1017/9781009129220.027 Published online by Cambridge University Press