Natively run OCaml from Rust

"How to get segfaults with two safe languages"

Mathias Sablé-Meyer Lucas E. Morales

June 2018

Contents

Introduction . . . 2
Expose the relevant OCaml functions . . . 3
The static way . . . 3
  Compiling the OCaml libraries . . . 3
    Using ocamlopt . . . 3
    Using dune — formally known as "jbuilder" . . . 3
  Targeting C . . . 4
    The C wrapper . . . 4
    The main.c file . . . 5
    A remark about OCaml libraries . . . 7
  Targeting Rust . . . 7
    The main Rust file . . . 7
    The build process . . . 9
The dynamic way . . . 11
  Compiling the OCaml libraries . . . 11
  Targeting C . . . 12
  Targeting Rust . . . 13
    With rustc . . . 13
    With cargo . . . 14
Conclusion . . . 14
Edits . . . 16

Introduction

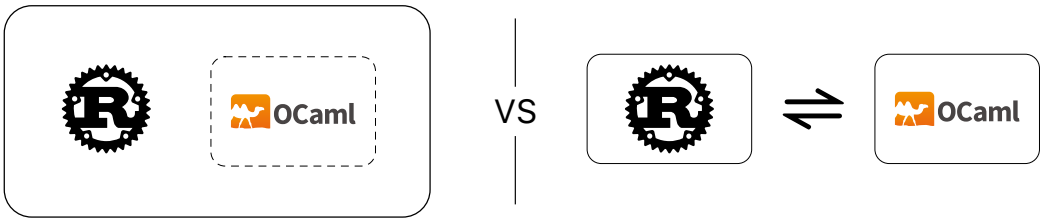

Figure 1: Conceptual difference between calling and FFI.

- **Theorem**: There are tasks such that there are languages such that there are people who prefer said language for said task.
- **Proof**: *left as an exercise for the reader.*
- **Problem**: how to interface separate pieces of code at low cost?

Here come FFIs, or how to compile pieces of code from different languages so that you don't have to rely on forking, subprocesses, `JSON`/`MessagePack`/`Cap'n Proto`, etc.

One language will declare the availability of some functionality to the external world, another will declare its use of some external functionality, both will be compiled together and the resulting binary will not distinguish who's who and just run fast. Some manual fiddling may be required to get the memory representation to match, but often it's just a question of shifting by a few bytes one way or another.

In the present document we focus on interfacing `OCaml` [1], `Rust` [2] and `C` [3] but more languages support this out of the box [4–6]. More specifically, we'll see how we can make `OCaml` functions available to either `C` or `Rust`, in two different ways.

Disclaimer: this has only been tested on GNU/Linux, with

- OCaml >= 4.06
  - dune >= 1.0
- Rust 1.25.0 to 1.29.1
- gcc 8.1

We hope it generalizes!

Expose the relevant OCaml functions

OCaml needs to know what function the external world is allowed to call. Adding this line does exactly this:

```
let _ = Callback.register "twice" twice
```

This tells the world that there is a function named "twice" that it can try to use, and when used it will call the OCaml function `twice`.

The first argument, the string, will be crucial down the road since the callback syntactically matches on this. Mismatched names lead to segfaults and there are no protections — all lies in the programmer's hands.

Next step is to compile this in such a way that it can be linked in other projects. Both static and dynamic linking will be covered below.

The static way

Compiling the OCaml libraries

Using ocamlopt

Ocamlopt has a flag for static libraries, `-output-obj`. However this will *not* export OCaml's runtime, leading to more linking mess. There is another[1] flag that does the job, namely `-output-complete-obj`.

```
$ ocamlopt -output-complete-obj math.ml -o libmath.o
$ file libmath.o
[...] ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), \
      with debug_info, not stripped
```

This creates a static library *that contains OCaml's runtime.*

Using dune — formally known as "jbuilder"

> Dune is a build system designed for OCaml/Reason projects only. It focuses on providing the user with a consistent experience and takes care of most of the low-level details of OCaml compilation.

[1] It's not in the man page, but `ocamlopt --help` yields "output-complete-obj: Output an object file, including runtime, instead of an executable"

> All you have to do is provide a description of your project and Dune will do the rest.
>
> dune repository

Using dune can prove useful for big project with several dependencies.

It took us a while to understand why we couldn't get it to produce the same thing as above. The trick is that it relies on the output *name* to know what to put in the file — having a correct `dune` file is not enough.

One needs to:

- Target an *object*: that's in the config file *and* in the extension.
- Include the *runtime*. For dune this requires an executable target.

Here's a possible `dune` file:

```
(executables
  (names math)
  (modes object)
)
```

This discussion and this part of the documentation got us to understand that to do that a working extension is `.exe.o`, so build it with `dune build math.exe.o`. Inspecting the file gives the following output:

```
$ dune build math.exe.o
$ cp _build/default/math.exe.o libmath.o
$ file libmath.o
[...] ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), \
      with debug_info, not stripped
```

Targeting C

Code: `ocaml_c_static.tar.gz`

The C wrapper

A C file that wraps the external functions into C functions is then required. It has to handle the memory layout of values, plus some other plumbing. This is where the exposed string name — "twice" — will matter, **beware**!

```c
#include <stdio.h>
#include <string.h>
#include <caml/mlvalues.h>
#include <caml/callback.h>

int twice(int n)
{
    static value *twice_closure = NULL;
    if (twice_closure == NULL) twice_closure = caml_named_value("twice");
    return Int_val(caml_callback(*twice_closure, Val_int(n)));
}
```

`Val_int` and `Int_val` are provided by mlvalues.h to interface the memory layout. We will have to reimplement them later on for Rust interfacing.

```c
#define Val_long(x)     ((intnat) (((uintnat)(x) << 1)) + 1)
#define Long_val(x)     ((x) >> 1)
#define Val_int(x) Val_long(x)
#define Int_val(x) ((int) Long_val(x))
```

This needs to be compiled to a static library too. One way to do this is to run `ocamlc -c mathwrap.c` which will produce a `mathwrap.o` static library[2]. `ocamlc` takes care of the caml headers, which by default `gcc` wouldn't — though in the end `gcc` gets called.

```
$ ocamlc -c mathwrap.c
$ file mathwrap.o
[...] ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

The main.c file

The `main.c` has little more to do. It initializes the OCaml runtime and must be told about the functions — they are not `extern` because they are already imported by the wrapper:

```c
#include <stdio.h>

int twice(int n);
int caml_startup(char ** argv);

int main(int argc, char ** argv)
```

[2]This can be taken care of either by ocamlopt or by dune directly.

```
{
  caml_startup(argv);
  printf("twice(2)        = %d\n", twice(2));
  printf("twice(twice(2)) = %d\n", twice(twice(2)));
  return 0;
}
```

It must be compiled against both the C wrapper object and the OCaml object. It also needs to know where OCaml's header files are — and it needs two additional libraries on our machines[3].

```
$ ocamlopt -output-complete-obj math.ml -o libmath.o
$ ocamlc -c mathwrap.c
$ gcc -lm -ldl -I `ocamlc -where` libmath.o mathwrap.o main.c
$ ./a.out
twice(2)        = 4
twice(twice(2)) = 8
```

Below is a complete Makefile to summarize the required steps:

```
ocaml-dune: math.ml mathwrap.c main.c dune
    dune build math.exe.o
    ocamlc -c mathwrap.c
    cp _build/default/math.exe.o libmath.o
    gcc -lm -ldl -I `ocamlc -where` libmath.o mathwrap.o main.c
    ./a.out

ocamlopt: math.ml mathwrap.c main.c
    ocamlopt -output-complete-obj math.ml -o libmath.o
    ocamlc -c mathwrap.c
    gcc -lm -ldl -I `ocamlc -where` libmath.o mathwrap.o main.c
    ./a.out

ocamlc: math.ml mathwrap.c main.c
    ocamlc -output-complete-obj math.ml -o libmath.o
    ocamlc -c mathwrap.c
    gcc -lm -ldl -I `ocamlc -where` libmath.o mathwrap.o main.c -lcurses
    ./a.out

clean:
    dune clean
    rm -f *.o *.cm* a.out dune-project
```

[3]This may change from version to version? I've seen curses used elsewhere too.

Note: we added an ocamlc version for completeness, which requires the libcurses library to be passed on to `gcc`.

A remark about OCaml libraries

Note that adding OCaml libraries to your source, say the `Unix` module, can be taken care of by dune. It will go as far as adding the symbols in the output `.o` file when it can[4]. As far as we can tell, ocamlopt will not.

If one wants to stay with ocamlopt and add `unix.cmxa` to the ocamlopt instruction, then `unix.a` must be passed to gcc. However some functions may be defined twice, both in `unix.a` and in `libmath.o` hence errors will occur. The workaround is to use the `ar` command to add `libmath.o` to `unix.a` and avoid the repetitions of symbols — something like this:

```
$ cp `ocamlc -where`/unix.a .
$ ar rs unix.a libmath.o
```

If you want to add more libraries, think about switching to dune. Otherwise the stack gets worse: `ar x <lib>.a` to get its internal `.o` files, then `ar qs <...>` to archive the whole thing together — it gets messy quickly and we're on shaky grounds.

Targeting Rust

Code: `ocaml_rust_static.tar.gz`

The idea is fundamentally the same: make an object file and have a wrapper around it that handles the memory layout. One has to go back and forth with the OCaml headers to see how the types are encoded in C in order to make things match.

The main Rust file

We'll give the file and then go through it:

```
mod ocamlthings {
    use std::ptr;
```

[4] If it can't shared objects are required, see below

```
    #[link(name = "math")]
    extern "C" {
        // translated from caml/callbacks
        fn caml_startup(argv: *mut *mut u8);
        fn caml_named_value(name: *const u8) -> *const Value;
        fn caml_callback(closure: Value, arg: Value) -> Value;
    }

    // translated from caml/mlvalues
    type Value = usize;
    macro_rules! val_int {
        ($x: expr) => {
            (($x as usize) << 1) + 1
        };
    }
    macro_rules! int_val {
        ($x: expr) => {
            ($x as usize >> 1) as i32
        };
    }

    pub fn initialize_ocaml() {
        let mut ptr = ptr::null_mut();
        let argv: *mut *mut u8 = &mut ptr;
        unsafe {
            caml_startup(argv);
        }
    }

    pub fn twice(n: i32) -> i32 {
        static TWICE_NAME: &str = "twice\0";
        static mut TWICE_CLOSURE: *const Value = ptr::null();
        unsafe {
            if TWICE_CLOSURE.is_null() {
                TWICE_CLOSURE = caml_named_value(TWICE_NAME.as_ptr());
            };
            return int_val!(caml_callback(*TWICE_CLOSURE, val_int!(n)));
        }
    }
}

fn main() {
    ocamlthings::initialize_ocaml();
    println!("twice(2)        = {}", ocamlthings::twice(2));
    println!("twice(twice(2)) = {}", ocamlthings::twice(ocamlthings::twice(2)));
```

```
}
```

The first half is a port of OCaml's header files. For your own project you may either want to check the raml project that already ports a few of the structures or read the relevant OCaml header files.

The `#[link(name = "math")]` line tells the compiler that the symbols are defined in an object named `libmath.o`.

Finally it declares Rust functions that call the OCaml counterparts, with the relevant memory layoutization. This is where the name must match what OCaml exported, **beware**!

The build process

With Rustc directly

Building the object files is the same as the C case. Because the library link name is already given in main.rs, only the `-L` flag is required — not the `-l` one (to specify a library path rather than a specific library name).

For some reason we can't link against the `.o` file with Rust. An `.a` archive that contains only one element (`libmath.o`) makes the following work:[5].

```
$ dune build math.exe.o && cp _build/default/math.exe.o ./libmath.o
$ ar qs libmath.a libmath.o
$ rustc -L . src/main.rs
$ ./main
twice(2)        = 4
twice(twice(2)) = 8
```

You can put that in a Makefile and have either dune or ocamlopt build the initial object.

Note: If you get a segfault, either you're calling a function that does not exist[6], or you have linking problems[7], or you're doomed.

[5] We'd love to understand more about this?

[6] Remember the syntactic matching!

[7] Are you linking against a `.a` and not `.o`?

With Cargo

This is the canonical way of building stuff in Rust. Instead of a Makefile, a `build.rs` handles the building process. The `Cargo.toml` file has nothing special so we'll just give the build file:

```
use std::env;
use std::process::Command;

fn main() {
    let out_dir = env::var("OUT_DIR").unwrap();
    let dune_dir = "_build/default";

    Command::new("dune")
        .args(&["build", "math.exe.o"])
        .status()
        .expect("Couldn't run builder clean. Do you have dune?");
    Command::new("cp")
        .args(&[
            &format!("{}/math.exe.o", dune_dir),
            &format!("{}/libmath.o", out_dir),
        ])
        .status()
        .expect("File copy failed.");
    Command::new("ar")
        .args(&[
            "qs",
            &format!("{}/libmath.a", out_dir),
            &format!("{}/libmath.o", out_dir),
        ])
        .status()
        .expect("ar gave an error");

    println!("cargo:rustc-link-search={}", out_dir);
}
```

It looks a lot like the C Makefile: it calls dune and copies the library. The extra `ar` pass is required here too, we still don't know why. The last line informs rustc to look for static objects in the build directory.

```
$ cargo run  # this is all we need!
twice(2)        = 4
twice(twice(2)) = 8
```

Phew. That may not look like much, but there are many hidden traps waiting all over: static vs shared library, including the OCaml runtime, handling OCaml's "Value" type, gluing all this together…

The dynamic way

Let's now assume that the project uses some OCaml stuff that relies on shared libraries. We'll use cairo (OCaml binding) as an example since that's how we ended up learning about this.

Compiling the OCaml libraries

Let's start with the simplest example provided in the cairo repository and modify it minimally so that it receives a string and write to that string:

```
(* Writes a simple PNG to the given file (by name). *)
let draw s =
  let surface = Cairo.Image.create Cairo.Image.ARGB32 120 120 in
  let cr = Cairo.create surface in
  Cairo.scale cr 120. 120. ;
  Cairo.set_line_width cr 0.1 ;
  Cairo.set_source_rgb cr 0. 0. 0. ;
  Cairo.rectangle cr 0.25 0.25 0.5 0.5 ;
  Cairo.stroke cr ;
  Cairo.PNG.write surface s

(* Don't forget to export it! *)
let _ = Callback.register "draw" draw
```

Upon adding `cairo2` to the `dune` file and assuming OCaml's cairo2 is installed[8], dune should figure out how to build it.

However, most systems don't have the static library for cairo — and trying to build them from source leads to a rabbit hole of building more and more required static libraries. Therefore trying to build a complete obj leads to a linking error along the line of "ld can't find -lcairo".

Ideally, what one wants is a binary that has the OCaml part as a static library and relies dynamically on the shared cairo library.

[8] `opam install cairo2`

This doesn't "Just Work™ We haven't been able to do that[9] so we moved everything to shared libraries. Changing the"modes" field in `dune` to `shared_object` and running `dune build draw.so` does that.

Sanity check:

```
$ file libdraw.so
libdraw.so: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), \
            dynamically linked, with debug_info, not stripped
```

Cool, it's a indeed a shared library.

Targeting C

Code: `ocaml_c_shared.tar.gz`

The wrapper and main files are modified accordingly — see the attached archive below for the details of implementation. Upon trying to run the program you'll be faced with the following message:

```
$ gcc -I `ocamlc -where` -lm -ldl libdraw.so drawwrap.o main.c
$ ./a.out
./a.out:
  error while loading shared libraries:
    libdraw.so:
      cannot open shared object file: No such file or directory
```

Looking at the dynamically linked binaries shows the missing library:

```
$ ldd a.out
linux-vdso.so.1 (0x00007ffe76f56000)
  libc.so.6                   => /usr/lib/libc.so.6 (0x00007f60a2c85000)
  [...]
  libdraw.so                  => not found
```

OK, but *we the programmers* know where it is. A few directions:

- Moving the `.so` to the relevant `/usr/local/lib` folder, but then that should probably wait for a `make install` part.

[9] dune can't build a static object without -lcairo and otherwise makes *everything* dynamic.

- Running the program with `LD_LIBRARY_PATH=. ./a.out` to tell it to look in the current folder.
- Compiling it with `-Wl,-rpath,'$ORIGIN'`[10] so that the binary, whenever run, looks in its own current folder for relevant shared libraries.

We opted for the last solution in the code but here's what happens with the second one:

```
$ gcc -I `ocamlc -where` -lm -ldl libdraw.so drawwrap.o main.c
$ LD_LIBRARY_PATH=. ./a.out
$ file output.png
output.png: PNG image data, 120 x 120, 8-bit/color RGBA, non-interlaced
```

Yay!

That was not so bad was it?

Targeting Rust

Code: `ocaml_rust_shared.tar.gz`

With rustc

Change are needed the `main.rs` similar to what was needed for `main.c` — details in the source archive — and then the previous `dune -> cp libraries -> Rust` sequence should build the binary.

As expected, `./main` fails with a shared library loading error, but as above the `LD_LIBRARY_PATH` should solve that[11]:

```
$ dune build draw.so && cp _build/default/draw.so .
$ rustc -L . src/main.rs
$ LD_LIBRARY_PATH=. ./main
$ file output.png
output.png: PNG image data, 120 x 120, 8-bit/color RGBA, non-interlaced
```

Success.

[10] $ORIGIN means wherever the binary *currently* is. Don't forget to double the $ sign if it's in a Makefile!

[11] setting it to the shared library's path

With cargo

The build.rs file is pretty straightforward once the previous case is solved:

```
use std::env;
use std::process::Command;

fn main() {
    let out_dir = env::var("OUT_DIR").unwrap();
    let dune_dir = "_build/default";

    Command::new("dune")
        .args(&["build", "draw.so"])
        .status()
        .expect("Couldn't run builder clean. Do you have dune?");
    Command::new("cp")
        .args(&[
            &format!("{}/draw.so", dune_dir),
            &format!("{}/libdraw.so", out_dir),
        ])
        .status()
        .expect("File copy failed.");

    println!("cargo:rustc-link-search={}", out_dir);
}
```

And indeed if we try:

```
$ cargo run  # this is all we need!
$ file output.png
output.png: PNG image data, 120 x 120, 8-bit/color RGBA, non-interlaced
```

For the time being we don't know how to tell `rustc`/`cargo` to change the rpath of the binary so we don't know how you can move it around with its friend the shared library and we have to rely on `LD_LIBRARY_PATH`.

Conclusion

We designed a silly benchmark roughly inspired by a arithmetical operation of computing a Linear Congruential Generator, or `rdm`, many times.

We ran this a bunch of times to make the test non trivial, and then used the benchmark crate in rust to compute the time per operation.

OCaml:

```
let rdm = fun x -> (1103515245 * x + 12345)

let looped_rdm = fun _ ->
  let x = ref 0 in
  for j = 0 to 4096 do
    x := !x + rdm(!x)
  done;
  !x
```

Rust:

```
pub fn rdm(a: i64) -> i64 {
    (1103515245 * a + 12345)
}

pub fn looped_rdm() -> i64 {
  let mut i = 0;
  for _ in 0..4096 {
      i += rdm(i);  // either this or ocaml::rdm
  }
  i
}
```

Results of the benchmark:

Call Type	Time (ns)
Rust::`looped_rdm`	7,365
OCaml::`looped_rdm`	9,791
Rust::`ocaml::looped_rdm`	10,025
Rust::`looped_rdm` (calling `ocaml::rdm`)	29,593

Remarks:

- OCaml is a bit slower than Rust for this type of program even before any kind of FFI.
- Rust repeatedly calling an expensive OCaml function had low overhead, but as the last benchmark shows, Rust repeatedly calling a cheap function slows things down a lot.
- Surprisingly enough the last one is much slower than we expected.

> Upon looking at the ASM, we noticed that `caml_start_program` is called for each function call and moves stuff around in the stack — to handle possible exceptions[12]. This is not an issue for OCaml alone. We tried to look at the generated assembly code on the OCaml side: despite some efforts toward removing optimizations we couldn't get it to not inline calls or optimize stack manipulation.

To whoever made it this far:

> I hope this may help some other lonely OCaml developer [...]
>
> Jeffrey

So do we. Reach for us maybe?

- Mathias Sablé-Meyer
- Lucas E. Morales

Edits

- Jun 07, 2018 : Changed link to mlvalues.h to reflect repository change.
- Oct 08, 2018 : Tested with latest version of the tools.

---

[1] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D. and Vouillon, J. 2013. The ocaml system release 4.01 documentation and user's manual. (2013).

[2] Matsakis, N.D. and Klock II, F.S. 2014. The rust language. *ACM sigada ada letters* (2014), 103–104.

[3] Ritchie, D.M., Kernighan, B.W. and Lesk, M.E. 1988. *The c programming language*. Prentice Hall Englewood Cliffs.

[4] Jones, S.P. 2003. *Haskell 98 language and libraries: The revised report*. Cambridge University Press.

[5] Chakravarty, M.M. 2003. *The haskell foreign function interface 1.0: An addendum to the haskell 98 report*.

[12] more input on this is welcome

[6] Bielman, J. and Oliveira, L. 2010. CFFI–the common foreign function interface. *CL package version 0.10*. 6, (2010).