Safely Mixing OCaml and Rust

Stephen Dolan
University of Cambridge

OCaml and Rust are safe languages with different strengths, but one cannot call the other without resorting to unsafe C code. This talk introduces `caml-oxide`*, a prototype implementation of a safe interface between these two languages.*

Unlike previous approaches (e.g. `ctypes`*,* `SWIG`*),* `caml-oxide` *allows safe direct sharing of data structures without copying, by encoding the invariants of OCaml's garbage collector into the rules of Rust's borrow checker.*

Safety

OCaml is a *safe* language: all programs, even buggy ones, have well-specified behaviour. Buggy programs might not produce the intended answer, but they will not segfault, corrupt memory or start executing attacker-supplied strings as code.

Like all safe languages, OCaml achieves safety with a mixture of compile-time and runtime techniques. Compatibility of datatypes is checked at compile-time by the type system, but ensuring that memory is not accessed after it is freed is done at runtime by the *garbage collector*, which delays freeing memory until it is no longer accessible.

Rust

Rust [2] is a relatively new language with an emphasis on performance. Rust is a safe language, but acheives this quite differently to OCaml. Instead of using a runtime garbage collector, Rust's type system uses affine linear types and region variables (called *lifetimes*) to statically ensure that memory is not accessed after it is freed.

For instance, suppose that an array is created and bound to a local variable, but also made accessible via a global variable, and then the original local variable goes out of scope. In an unsafe language like C, using the global variable to access the array after it is freed can segfault or worse. In OCaml, the garbage collector will notice the global variable and ensure that the array remains accessible. In Rust, the reference to the array persisting beyond the array's lifetime is a compile-time error.

Rust and OCaml have quite different strengths. Rust's fine-grained control over memory management and data layout mean that better performance can often be attained than with idiomatic OCaml. However, explicit lifetime tracking is tedious, and the lifetime types (which are not generally inferred) can be awkward, particularly for higher-order code. For instance, writing some code that takes two functions and returns their composition is surprisingly tricky.

So, it seems like it could be useful to combine both of these safe languages in a single project.

OCaml's FFI and C stubs

Sadly, OCaml's foreign function interface is decidedly unsafe. The FFI is designed to call C functions from OCaml, but these functions (known as *C stubs*) being called must be written in an extremely stylised manner [1].

The central issue is that the OCaml garbage collector might run during any allocation, and when it runs it will free some blocks to which it sees no pointers, and will move other blocks and update the pointers it sees. Therefore, C stubs that manipulate values on the OCaml heap must ensure that whenever the GC might run, it will be able to see all pointers to the OCaml heap that are in use.

Writing C stubs which conform to these requirements is hard. Function parameters and local variables holding OCaml values cannot be declared using normal declarations, but must use special `CAMLparamN` and `CAMLlocalN` macros, so that the GC is aware of them. Even when using these macros, it is still easy to make mistakes. Figure 1 shows three attempts at implementing the function `strtail : string -> string option`, which returns all but the first character of its input, or `None` if the string is empty. Can you tell which implementations are correct? (Answer in the appendix)

Generating C stubs

One way to avoid the difficulty of writing correct C stubs by hand is to generate them. The most advanced system for doing so in OCaml is `ctypes` [3], which reflects C types into the OCaml type system. This works well for calling C functions from OCaml, but does not allow OCaml data structures to be easily manipulated by, allocated from, or shared with C.

```
1  CAMLprim value caml_strtail(value str)
2  {
3    CAMLparam1 (str);
4    CAMLlocal1 (v);
5    if (caml_string_length(str) == 0) {
6      v = Val_int(0);
7    } else {
8      ...
9    }
10   CAMLreturn (v);
11 }
```

(a) Implementation of `strtail`, omitting code to allocate `Some` on line 8.

```
v = caml_alloc(1, 0);
caml_modify(&Field(v, 0),
  caml_copy_string(String_val(str) + 1));

v = caml_alloc_small(1, 0);
Store_field(v, 0,
  caml_copy_string(String_val(str) + 1));

v = caml_alloc_small(1, 0);
Field(v, 0) =
  caml_copy_string(String_val(str) + 1);
```

(b) Three possible implementations of line 8

Figure 1: Which of these is correct?

Borrowing the collector

Instead, `caml-oxide` connects OCaml and Rust by encoding the rules necessary to cooperate with garbage collection into the Rust type system. The main novelty of Rust's type system is the *borrow checker*, which ensures that no value is ever mutably aliased. Mutation is allowed, as long as there is a unique mutable reference. Aliasing is allowed, as long as all references are immutable. Both are allowed for the same value, as long as the lifetime of the single mutable reference is disjoint from the lifetime of the immutable references.

`caml-oxide` uses a Rust reference to model the garbage collector, represented by a phantom type. Temporary values have an immutable reference to the GC, while functions that allocate and may invoke the collector expect a mutable reference. By ensuring that the immutable and mutable references have disjoint lifetimes, Rust's borrow checker verifies that temporary values are not referred to across allocation points. If a value must be reused after an allocation, it must be explictly preserved. This works like `CAMLlocalN` in C stubs, but is checked statically.

Example

Below is an attempt at a Rust implementation of the function `triple x = (x, (x, x))` using `caml-oxide`:

```
fn triple(gc, x: AA) -> Pair<AA,Pair<AA, AA>> {  1
  let s = call!{alloc_pair(gc, x, x)};           2
  call!{alloc_pair(gc, x, s)}                    3
}                                                4
```

The `call!` macro is necessary when calling allocation functions, to set up reference boilerplate.

This attempt has a bug: the reference to `x` on line 3 is invalid, since `x` might have been moved during the allocation on the previous line. Unlike when writing a C stub, the Rust compiler gives a compile error on this code, pointing out that we `cannot borrow '*gc' as mutable because it is also borrowed as immutable`— in other words, that we cannot do the allocation on line 2 because a temporary value (`x`) is still live.

The fix is the same as in C: we must create an explicit variable that is registered with the GC, to keep `x` alive across an allocation.

```
fn triple(gc, x: AA) -> Pair<AA, Pair<AA, AA>> {1
  let v = x.var(gc);                             2
  let s = call!{alloc_pair(gc, x, x)};           3
  call!{alloc_pair(gc, v.get(gc), s)}            4
}                                                5
```

Discussion

`caml-oxide` has features not mentioned here, but of interest to the OCaml programmer: OCaml types are reflected into the Rust type system, allowing `.mli` interfaces to be auto-generated from Rust functions like `triple` above. However, the current implementation only understands a limited set of OCaml types. The prototype is available from:

`github.com/stedolan/caml-oxide`

References

[1] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, J. Vouillon. Living in harmony with the garbage collector. OCaml manual, §20.5.

[2] Rust programming language. `https://rust-lang.org`.

[3] J. Yallop, D. Sheets and A. Madhavapeddy. A modular foreign function interface. Science of Computer Programming, 2017.

A Correctness of Figure 1

All three implementations in Figure 1b are wrong.

```
v = caml_alloc(1, 0);
caml_modify(&Field(v, 0),
  caml_copy_string(String_val(str) + 1));
```

C does not specify the order of evaluation of function arguments, so it is possible for `&Field(v, 0)` to be evaluated, yielding a pointer stored in a temporary location, before `caml_copy_string(...)` runs. If `caml_copy_string` invokes the GC, then the value of `v` will be correctly updated as is registered with the GC using `CAMLlocal1`. The temporary location holding `&Field(v,0)`, however, is not registered and will not be updated if `v` moves, causing `caml_modify` to be passed a stale pointer.

The `Store_field` macro carefully controls the order of evaluation (it expands to several statements before calling `caml_modify`), so the following code is correct:

```
v = caml_alloc(1, 0);
Store_field(v, 0,
  caml_copy_string(String_val(str) + 1));
```

However, the second example of Figure 1b is incorrect, because it uses the low-level `caml_alloc_small` which does not initialise the newly-allocated object. It is incorrect to call `Store_field` or `caml_modify` on a field that has not been initialised, since the GC's write barrier inspects and may later dereference the old value of a field being modified.

When using the low-level `caml_alloc_small`, the fields of the new object must be directly assigned using the `Field` macro. The third attempt in Figure 1b does this:

```
v = caml_alloc_small(1, 0);
Field(v, 0) =
  caml_copy_string(String_val(str) + 1);
```

Yet this code is not correct, for much the same reason as the first example: the assignment operator in C does not guarantee order of evaluation, so the address of `Field(v, 0)` may well be computed before the call to `caml_copy_string` invokes the collector, causing the newly-allocated string to be written to a stale address and leaving the new `v` uninitialised.

In order to correctly use `caml_alloc_small`, it is necessary to register the newly-allocated string with the collector, in another variable `s` declared with `CAMLlocalN`:

```
s = caml_copy_string(String_val(str) + 1);
v = caml_alloc_small(1, 0);
Field(v, 0) = s;
```