Boxroot, movable GC roots for better FFI

Guillaume Munch-Maccagnoni and Gabriel Scherer

2022

Introduction

We propose a new root-registration API and implementation for the OCaml FFI, which offers:

- better performance than existing APIs (local or global roots)
- a more multicore-friendly design
- simpler reasoning, enabling an easier FFI for Rust

Our approach combines two main ideas.

1. We move from a callee-roots protocol (functions receive GCed values as arguments and have to ensure that they are registered with the GC at their own allocation points) to a caller-roots protocol (functions are passed smart pointers that guarantee that the values they contain is registed with the GC).
2. As a result of this change, the root-registration API is in charge of allocating memory for the roots (smart pointers), instead of having to register arbitrary user-allocated addresses. Having control over memory placement enables a more efficient implementation. In essence, it is a very simple allocator (only 1-word elements),

We believe that this approach generalizes beyond OCaml, to other FFI situations where a GCed language (typically: functional programming languages) interacts with a non-GCed language, especially those that emphasize move semantics for resource handling (typically C++ or Rust) and static safety.

Context

Our initial motivation for this work was an industrial user interested in building a safe FFI between Rust and OCaml.

The current status of our work is a library that hooks into the OCaml runtime to propose our alternative root-registration API. We carefully optimized the library for Sequential OCaml (OCaml 4), and have a non-optimized prototype for Multicore OCaml (OCaml 5). Our code is used in production by said industrial user.

Our wish for the future is to have this root-registration API integrated in the upstream OCaml runtime, so that it is directly available to more users. We started pushing in this direction by authoring an RFC for the OCaml compiler distribution: https://github.com/ocaml/RFCs/pull/22.

Root registration APIs

Root-registration in the current OCaml FFI

When the OCaml runtime performs Garbage Collection (GC), OCaml values may be moved around in memory. The GC needs to know about all pointers to those OCaml values, to be able to update the pointer to the new location. The OCaml compiler cooperates with the runtime to efficiently declare all OCaml values it knows about, but FFI users (interacting with the OCaml runtime from a different language, typically C) need to do the work themselves. A typical example:

```
value local_fixpoint(value f, value x)
{
  CAMLparam2(f, x);
  CAMLlocal1(y);
  y = caml_callback(f,x);
  if (compare_val(x,y)) {
    CAMLreturn(y);
  } else {
    CAMLreturn(local_fixpoint(f, y));
  }
}
```

The `value` type in C represents OCaml values. The `CAMLparam2(f, x)` macro will register the address of `f` and `x` with the GC, so that they get updated if the OCaml blocks they point to get moved by the GC – typically when evaluating `caml_callback(f,x)`, which is the C-side equivalent of an application `f x`. `CAMLlocal1` defines a `value` parameter and registers it as well. `CAMLreturn` un-registers those three local roots.

This FFI API follows the callee-roots protocol: the function `local_fixpoint` is in charge of registering its value parameters with the GC. This is a natural approach as FFI functions receive `value` directly, the type of OCaml values, and not some indirection, and correspondingly it has a very low overhead in common cases (nanoseconds).

Two issues with this approach are:

1. Those values are not "movable", as only their current address is registered with the GC. If you call another FFI function on `f` or `x`, then that function will get the same values at a different address and needs to re-register them. There is also no way to return values that remain registered. This creates an impedance mismatch with systems programming language that rely on move semantics.

2. As an arguably less important consequence, deep FFI call stacks typically lead to several local-root registrations for OCaml values. In this example `local_fixpoint`, each new recursive iteration of the fixpoint will re-register local variables corresponding to the same OCaml values. With caller-roots API, a function receives roots that are already registered, which it can in turn pass to its own callees (or as return values) without any re-registration.

We worked on providing a safe typing (and lifetime) discipline for this API in Rust; our takeaway is that it can be done, but it is hard and the result is unpleasant to use. Intuitively, callee-roots API implies that function parameters, which are omnipresent in FFIs, must always be handled as possibly-not-registered temporaries that get invalidate by the GC, which is the hardest aspect of the FFI to model safely.

(Generational) global roots For FFI values whose lifetime does not follow the call-return struccture of the program, the OCaml runtime also provides a "global roots" API.

```
void caml_register_generational_global_root (value *);
void caml_remove_generational_global_root (value *);
void caml_modify_generational_global_root(value *r, value newval);
```

Notice that with this API again, the user passes an arbitrary `value *` address for registration; the user has to store a set of arbitrary addresses with fast traversal (for GC root scanning) and also fast access to a given element (for modification or removal). This is implemented in the OCaml runtime as a global skip-list.

Values registered using this API are not "movable": moving the `value` around changes its address. A common usage pattern to get a "movable" root on top of this API is as follows: the user first `malloc` a `value *`-sized block, registers its content with the OCaml GC, and passes the malloced address around.

A boxroot API

The API of our `boxroot` library is as follows:

```
/* `boxroot_create(v)` allocates a new boxroot initialised to the
   value `v`. This value will be considered as a root by the OCaml GC
   as long as the boxroot lives or until it is modified. A return
   value of `NULL` indicates a failure of allocation of the backing
   store. */
inline boxroot boxroot_create(value);

/* `boxroot_get(r)` returns the contained value, subject to the usual
   discipline for non-rooted values. */
inline value boxroot_get(boxroot r) { return *(value *)r; }

/* `boxroot_delete(r)` deallocates the boxroot `r`. The value is no
   longer considered as a root by the OCaml GC. The argument must be
   non-null. */
inline void boxroot_delete(boxroot);

/* `boxroot_modxbify(&r,v)` changes the value kept alive by the boxroot
   `r` to `v`. It is equivalent to the following:

       boxroot_delete(r);
       r = boxroot_create(v);

   In particular, the root can be reallocated. However, unlike
   `boxroot_create`, `boxroot_modify` never fails, so `r` is
   guaranteed to be non-NULL afterwards. In addition, `boxroot_modify`
   is more efficient. Indeed, the reallocation, if needed, occurs at
   most once between two minor collections. */
void boxroot_modify(boxroot *, value);
```

This API is inspired by the Rust type `Box<T>`, or by C++ `shared_ptr`. It can be implemented on top of OCaml's global roots by using `malloc` as we mentioned, which combines an OCaml-agnostic allocator with an address-agnostic GC-registration datastructure. Large efficiency gains are possible by integrating the allocator the GC-registration logic, and it also makes it easier to use a multicore-friendly implementation (but we have not done this yet).

Boxroot

Our sequential implementation of the boxroot API allocates boxroots in `malloc`-ed "pools" of fixed size – in our experiment, sizes 512 or 1024 work. Pools amortize the cost of `malloc`, and improve memory locality of root scanning. A natural multicore-friendly design is to let each thread/domain have its own pools.

(No) Generational optimizations

We tried several approaches to make boxroots "generational", in particular:

- classifying pools into "old pools", with only old values (not scanned by a minor GC), and "young pools" which may contain young values

- registering young values in the OCaml remembered set, instead of scanning them ourselves (this was suggested by Stephen Dolan)

It turns out that all those generational optimizations are slower than doing nothing at all. They require a young-or-old test in the hot path of allocation, which is costly enough. By handling all boxroot-allocated values as potentially-young, we defer this test to the next minor GC, where some of those roots may in fact have been removed already.

Code and benchmarks

Our code is available at https://gitlab.com/ocaml-rust/ocaml-boxroot/ and we have, in particular, detailed benchmarks and benchmark results at https://gitlab.com/ocaml-rust/ocaml-boxroot/-/blob/main/README.md. In most cases, FFI code is not very root-intensive and all approaches have roughly similar performance; but for root-heavy benchmarks, boxroot performs similarly or better than the other APIs.

In the interest of space we will only detail our `fixpoint` test, which runs a fixpoint computation with various root-registration APIs, varying the fixpoint iteration count to observe the result of repeated registration in callee-roots APIs: with N=1 iterations, local roots are noticeably faster than anything else, with N=1000 iterations, boxroot is faster than anything else. But for an intermediate situation like N=5, on a representative run:

- the fixpoint using local roots takes 57ns
- the fixpoint using boxroot takes 46ns
- the fixpoint using global roots takes 214ns
- a boxroot variant using the remembered set takes 64ns
- a boxroot variant using a doubly-linked list of roots (instead of pools), suggested by Frédéric Bour, takes 91ns
- using our boxroot implementation in callee-roots style, that is with repeated root registration for th same value, takes 66ns

In this root-intensive example, the efficiency gains of boxroots (in addition to the gains in building an easy, safe FFI API) is noticeable.

References

Frédéric Bour. Camlroot: revisiting the ocaml ffi, 2018. URL https://arxiv.org/pdf/1812.04905. code at https://arxiv.org/pdf/1812.04905.

Stephen Dolan. Safely mixing ocaml and rust, 2018. URL https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxtbHdvcmtzaG9wcGV8Z3g6NDNmNDlmNTcxMDk1YTRmNg. proof-of-concept at https://github.com/stedolan/caml-oxide.

Manish Goregaokar. Designing a gc in rust, 2015. URL https://manishearth.github.io/blog/2015/09/01/designing-a-gc-in-rust/.

Alan Jeffrey. Josephine: Using javascript to safely manage the lifetimes of rust data, 2018. URL http://arxiv.org/abs/1807.00067. code at https://github.com/asajeffrey/josephine.