Boxroot, fast movable GC roots for a better FFI

Guillaume Munch-Maccagnoni, Gabriel Scherer

Boxroot, fast movable GC roots for a better FFI

Guillaume Munch-Maccagnoni
INRIA

Gabriel Scherer
INRIA

September 13, 2022

We propose a new interface and implementation for managing garbage collector (GC) roots for the OCaml foreign-function interface (FFI), which offers:

- better performance than existing root-registration interfaces;
- efficient support for multicore OCaml, thanks to a multicore-friendly design;
- a reasoning based on resource-management idioms, enabling an easier OCaml FFI in Rust.

Contents

1 Introduction

The manipulation of managed values (of a program written in a garbage-collected language) inside foreign (non-managed) code is a matter for performant interoperability. In terms of implementation, one has to deal with 1) keeping values alive while they are referenced from the foreign language, and 2) updating pointers into the heap as objects are moved (in the case of moving GCs). This is typically achieved by giving programmers opaque pointers into a table of referenced objects, manipulated via safe primitives (Jones et al., 2011, §11.4). The elements of this table are treated as roots for the GC. For instance, the Java Native Interface (Liang, 1999) proposes such opaque pointers, the *local* and *global references*.

The OCaml language also aims for performant interoperability with its C FFI, however using a representation of values which 1) is transparent, 2) avoids the indirection.

Some discipline must be followed by the programmer in all cases (e.g. one must not let local references escape in the case of the Java Native Interface; one must declare local parameters as roots in the case of the OCaml-C FFI). Inevitably, this means that buggy programs can be found in the wild, which misuse these interfaces (Furr and Foster, 2005; Kondoh and Onodera, 2008). Also, OCaml programmers seek to achieve greater performance in practice by following a more expert (and undocumented) discipline.

We propose a new interface and implementation for handling managed values for the OCaml FFI, which offers:

- better performance than OCaml's current root-registration interfaces (local and global roots);
- efficient support for OCaml 5 with a more multicore-friendly design, using per-domain[1] data structures;
- a reasoning fitting resource-management idioms, enabling an easier OCaml FFI for Rust, leveraging the latter's advanced type system.

Our contributions include a C library called Boxroot[2] (for *boxed root*), a new root registration interface for OCaml. This library is already in use in several OCaml-Rust interfacing libraries.

[1] A *domain* is OCaml's unit of parallelism, roughly a collection of threads mapping to one CPU core.
[2] https://gitlab.com/ocaml-rust/ocaml-boxroot

1.1 Context

The initial motivation for more efficient and flexible rooting schemes was our ongoing investigations into mixing *linear allocation with re-use* (Lafont, 1988; Baker, 1992) with garbage collection. However, our motivation for investigating the Rust discipline and building the Boxroot prototype was an industrial user interested in building a safe FFI between Rust and OCaml.

This industrial user started the development of the ocaml-interop Rust library (main author: Bruno Deferrari), a library aiming to implement a low-level but safe interface between OCaml and Rust. This library was initially inspired by an earlier experiment (Dolan, 2018), but it quickly moved to using our approach when the first experimental version of Boxroot became available.[3]

The current status of our prototype is a library that hooks into the stock OCaml runtime to propose our alternative root-registration API. We have carefully optimised the library for both OCaml 4 and 5 (multicore). The development of Boxroot has led to various improvements to the C API of the multicore OCaml runtime to make this possible.

One possibility for the future is to have this root-registration interface integrated in the OCaml distribution, to make it directly available to all users and to more easily fix remaining limitations (Section 2.3). We started pushing in this direction by authoring an RFC for the OCaml language.[4]

1.2 Our contributions in three layers

The ideas in our approach is best described in three layers: runtime, language abstractions, typing.

1. In term of runtime implementation, we see the creation of roots manipulable from foreign code as a resource allocation problem. Unlike the current notion of global root in the OCaml API, which has to deal with arbitrary addresses supplied by the user of values, and like the references in the Java Native Interface, our root-registration interface Boxroot chooses the location of the roots in memory. This allows for more performant and flexible roots.

   We formulate the observation that it is solved using standard allocation techniques. Our root-registration interface is essentially an allocator (for only one-word elements) based on modern concurrent allocation techniques (along with some generational GC techniques). Likewise (and we believe that this implementation strategy was previously undocumented in the literature), local and global references in Hotspot's implementation of the Java Native Interface implement arena (bump-pointer) allocation with checkpoints and concurrent bitmap allocation, respectively.

2. At the level of language abstractions, we move from the *callee-roots* protocol which the OCaml-C FFI is based on (functions receive GC-allocated 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 pointers that guarantee that the values they contain is registered with the GC). This caller-roots protocol is supported by the possibility to manipulate roots as resources: like generic heap-allocated memory, our roots can be moved

[3] We thank Bruno Deferrari for his initial contribution.

[4] https://github.com/ocaml/RFCs/pull/22

(their ownership transferred) or borrowed (received without the responsibility to dispose of them).

Like the OCaml FFI, and unlike the Java Native Interface, our roots are transparent. Borrowing a root exposes a reference to its value; these values can be accessed directly. Furthermore, when no rooting is necessary (if a function does not allocate, typically), plain values can still be passed around, just like in current expert OCaml FFI practices. By enjoying a more expressive discipline, programs can allocate fewer roots, and so we hope to avoid the need for separate implementations of global references and local references as in the Java Native Interface (presumably useful for efficiency).

3. To enforce the resource-management discipline of roots (prompt release and correct use), we want to treat individual roots as *smart pointers* (resource containers), obeying similar rules as (for instance) the reference-counting pointers from C++ and Rust.[5] Our roots can therefore be passed and returned by ownership, borrowed, and stored in data structures obeying ownership semantics. They can also be sent or shared across threads.

   (This means that C users would pay the development costs of explicit resource management, such as freeing by hand and manually handling errors. Our focus is not on the C language, but in the Java Native Interface, local references have the advantage of being released automatically when the function returns to Java.)

   In Rust, we further draw from previous experiments for a safe GC interface (Goregaokar, 2015; Jeffrey, 2018; Dolan, 2018) to implement the rules for living in harmony with the GC in Rust, including the relaxed discipline in which one can manipulate non-rooted values.[6]

   Furr and Foster (2005); Kondoh and Onodera (2008) previously proposed to approach the problem of the GC discipline in foreign code from the angle of static analysis. Unlike these works, our approach does not need external tooling, and fits the language and API design, because the problem is reduced to one the Rust language was meant to solve (modulo some limitations of its type system, see Section 4.9). Despite the previous works, no tool for finding bugs in this area currently exists for the OCaml-C interface; in addition we are not aware in general of a tool ensuring that a discipline is entirely respected.

We believe that these ideas can be taken separately, and generalise beyond OCaml, to other FFI situations where a language with GC interacts with a language without pervasive GC, especially those that implement move semantics or similar techniques for resource handling (typically C++ or Rust).

Furthermore, our results go beyond the matter of safe and performant FFI. Indeed, our experiment gives an idea of theoretical costs of embedding (owning) a GC-allocated value inside a linearly-allocated value. As a rule of thumb:

[5] This perspective comes naturally if one sees reference-counting smart pointers as implementing a form of garbage collection in C++ and Rust; this also brings to mind Bacon et al. (2004).

[6] Opaque roots are a priori simpler to deal with: GC values are never handled directly, everything is rooted. Yet this is not so simple: 1) in the case of OCaml, a part of the work must still be done in order to ensure that the thread holds its *domain lock*, and more generally there has to be some way to control GC safe points; 2) the functions that access "opaque" roots must be written at some point.

- the performance of allocating and deallocating a boxed root should be seen as similar to that of allocating and deallocating linearly-allocated values with a modern concurrent allocator,
- the performance of scanning boxed roots during GC should be seen as similar or better to that of scanning the same values if they were reachable from the GC heap, thanks to good memory locality.

In qualitative terms of performance, this notion of root is "zero-cost" (it does not create costs that are not already there), making it a suitable device for consideration in future language explorations in mixing linear allocation and GCs.

2 Root-registration interfaces in the OCaml FFI

2.1 Local roots

When the OCaml runtime performs garbage collection, OCaml blocks can be moved around in memory. The GC needs to know about all pointers to these blocks, to be able to update them to point 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. *Local roots* implement a shadow stack which the programmer has to manage by hand. 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: either an immediate (a tagged, signed integer) or a pointer into the heap. The `CAMLparam2(f, x)` macro will register `f` and `x` with the GC by storing their address in a linked list on the stack, which gets visited by the GC when it is looking for roots. `CAMLlocal1` declares a local variable of type `value` and stores its address in the same way. When the GC scans the list, `f`, `x` and `y` get updated if the OCaml blocks they point to are moved by the GC—for instance here during `caml_callback(f,x)`, which evaluates the application `f x` in OCaml. Eventually, `CAMLreturn` un-registers those three local roots, by unlinking from the linked list.

A `value` in C may or may not be registered as a root. It is up to the user to ensure that all values kept alive across a collection point are rooted. For this, the OCaml-C FFI promotes a 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). It proposes a simple discipline which is easy for a C programmer to apply consistently and review, but it still allows escape hatches for experts who know when rooting can be avoided.

Two issues with this approach are:

1. Those values are not *movable*, as only their current address is registered with the GC. If one calls 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 or to store values in a data structure. 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 a caller-roots protocol, 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. It is possible to pass local roots by reference, but this is not the style which is taught.

2.2 (Generational) global roots

For FFI values whose lifetime does not follow the call-return structure of the program, the OCaml runtime has also been providing a *global roots* interface.

```
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 inteface, the user passes an arbitrary `value *` address for registration; the runtime 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 OCaml with global skip-lists. Registering and removing a global root therefore has a logarithmic time complexity in the number of roots.

Generational global roots discriminate between young and old values at creation (whether they are allocated in OCaml's minor heap). By keeping track of which roots point to young values, it is possible to avoid scanning most roots at the start of the minor GC.

Like with local roots, values registered using this interface are not movable either: passing a `value` results in a `value` with a different address. A common usage pattern to get a movable root on top of this interface is as follows: the user `mallocs` a `value`-sized block and registers its address as a global root with the OCaml GC, before passing the block around or storing it in a foreign data structure.

This interface has other limitations:

- In OCaml 4, it is necessary to hold the OCaml master lock while unregistering a root. This is unsuitable for situations where a root is stored in a foreign structure, in a thread that does not always hold the master lock. Indeed, the program should always be able to clean-up its resources, for instance, during unwinding following a panic (exception) in Rust. Obviating other obstacles, acquiring OCaml's runtime lock can take up to 50ms, which is not contemplable in a destructor.
- In multicore OCaml, the global structure is protected by a mutex. This choice supports deallocating a root in a thread that does not belong to the domain in which it has been allocated (for instance for sending or sharing OCaml values between domains). But it also makes it more expensive. It is unclear that there is a simple way to make it efficient while still supporting the sending and the sharing of roots between threads (and avoiding the previous issue).
- In OCaml's current implementation of global roots, scanning the roots involves the traversal of a linked list. While this is optimal in terms of complexity, this is more costly than scanning local roots, which have a better cache locality.

2.3 Scanning hook

The OCaml runtime supports (almost) arbitrary ways of scanning GC roots, thanks to the *scanning hook*. With it, users[7] can register scanning functions, which are called at the beginning of minor and major GC, receiving a *scanning action* as argument which in turn must be applied to every root and its value.

A custom scanning function is registered by mutating the variable `caml_scan_roots_hook` from the header file `caml/roots.h`:

```
typedef void (*scanning_action) (value, value *);
extern void (*caml_scan_roots_hook) (scanning_action);
```

This scanning hook has so far been mainly used by the virtual machine of the Coq proof assistant.

We use scanning hooks, and various other OCaml runtime hooks, to implement Boxroot. We had to implement various improvements to the hooks of OCaml 5 to make our multicore implementation possible.

(We also inherit their main limitation, that they do not provide for the sake of latency a way of scanning roots incrementally: the roots have to be scanned all at once, in parallel in each domain, at the start of major GC cycles. However this limitation is not fundamental to our approach.)

3 Boxroot

3.1 The Boxroot interface

The C interface of our Boxroot library is as follows:

[7] It should be noted that the scanning hook is an internal interface. Its use requires familiarity with the OCaml runtime, and it is not subject to normal backwards-compatibility expectations.

```
boxroot boxroot_create(value);
```

`boxroot_create(v)` allocates a new root 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. The OCaml domain lock must be held before calling `boxroot_create`.

```
inline value boxroot_get(boxroot r) { return *((value *) r); };
inline value const * boxroot_get_ref(boxroot r) { return (value *)r; };
```

`boxroot_get(r)` returns the contained value, subject to the usual discipline for non-rooted values. `boxroot_get_ref(r)` returns a pointer to a memory cell containing the value kept alive by `r`, that gets updated whenever its block is moved by the OCaml GC. The pointer becomes invalid after any call to `boxroot_delete(r)` or `boxroot_modify(&r,v)`. The OCaml domain lock must be held before calling `boxroot_get` or before deferencing the result of `boxroot_get_ref`.

```
void boxroot_delete(boxroot);
```

`boxroot_delete(r)` deallocates the root `r`. The value is no longer considered as a root by the OCaml GC afterwards. One does not need to hold the OCaml domain lock before calling `boxroot_delete`.

```
void boxroot_modify(boxroot *, value);
```

`boxroot_modify(&r,v)` changes the value kept alive by the root `r` to `v`. The OCaml domain lock must be held before calling `boxroot_modify`. It is essentially equivalent to the following:

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

However, `boxroot_modify` is more efficient: it avoids reallocating the boxroot if possible. The reallocation, if needed, occurs at most once between two minor collections.

3.1.1 Boxroot vs. global roots

Boxroot can be implemented on top of global roots in OCaml 5 by using `malloc` as we mentioned, which combines an OCaml-agnostic allocator with an address-agnostic GC-registration data-structure, protected by a global lock. But we show that large efficiency gains are possible by integrating the allocator with the GC-registration logic:

- operations can have constant time complexity,
- the implementation can be more multicore-friendly,
- scanning can be done in all domains in parallel,
- data structures can have better temporal and spatial cache locality, benefitting both scanning and the operations.

In fact, the good performance lets us propose it as a single rooting mechanism replacing both global and local roots.

In the context of the OCaml-C FFI, Bour (2018) has also previously advocated the passing of roots instead of values, born out of the experience of writing OCaml bindings to the Qt library. One argument was that it is more amenable to dynamic checks (an aspect which we did not investigate, since our goal regarding safety is to perform static checks using Rust's type system). It also proposed a dynamic allocation of roots in *regions*, which would be similar to local references in the Java Native Interface.

3.1.2 Alternative: a doubly-linked list

The reader might have wondered whether it is possible to give a simpler implementation of the boxroot interface with a doubly-linked list (or two, with an obvious generational optimisation). This is essentially our own boxroot implementation, but for memory pools of size 1. We have implemented a naive version of it (without thread-safety), for comparison purposes. (This control was suggested to us by Frédéric Bour.)

3.2 Implementation of Boxroot

Boxroot is a custom multi-threaded allocator that manages fairly standard freelist-based local memory pools (see Berger et al., 2000, among others). Our allocator manages a unique size, the size of OCaml values (64 bits on 64-bit platforms).[8] But we arrange to scan these pools efficiently. In standard fashion, the pools are aligned in such a way that the most significant bits can be used to identify the pool from the address of their elements, the roots, and thus access the pool metadata.

Since elements of the freelist are guaranteed to point only inside the memory pool, and non-immediate OCaml values are guaranteed to point only outside of the memory pool, we can identify allocated slots as follows:

$$\texttt{allocated(slot, pool)} \stackrel{\text{def}}{=} \texttt{(pool != (slot \& \~{}(1<<}N\texttt{ - 2)))}$$

N is a parameter determining the size of the pools, 2^N. (In our case, $2^N = 16\,\text{KB}$, corresponding to 2032 roots per pool after subtracting the size of the pool header.) The bitmask is chosen to preserve the least significant bit, such that immediate OCaml values (those with their least significant bit set) are correctly classified.

The scanning function is registered via the scanning hook and dispatches to different functions according to whether we are scanning for the minor GC or not.

3.2.1 Generational optimisations

We take advantage of GC generations in several ways.

The memory pools are managed in one of several rings, depending on their *class*. The class distinguishes pools according to OCaml generations, as well as pools that are free (which need

[8] We could in fact manage arbitrary unboxed memory layouts, such as mixing GCed and non-GCed values.

not be scanned). A pool is *young* if it is allowed to contain pointers to the minor heap. During minor collection, we only need to scan young pools. At the end of the minor collection, the young pools, now guaranteed to no longer point to any young value, are promoted into the ring of *old* pools.

When allocating a new root, we unconditionally allocate it in a young pool, without testing at allocation-time whether the initial value is young or old. Indeed, we have found that this test is better done during scanning, for several reasons:

- in some situations, there may be many more roots allocated than living through collections (for instance in the use-case of local roots);
- it is possible to discriminate between old and young values more efficiently in a tight loop, such as when scanning for the minor GC;
- we noticed the benefits of inlining the fast paths of all the operations (create, get, get_ref, delete and modify). So we must be careful to avoid negative real-world effects of inlining (which would be hard to notice in our current benchmarks), such as increased code size and introducing branches that are not statically predictible.

Inlining is interesting in use-cases with short-lived roots, where cache misses are not expected to be the dominant cost, such as when used in replacement of local roots (Section 5.2).

When the current pool is full, we check if a young pool is available. If not, then we demote an old pool into a young pool, but only one which is less than half-full. Otherwise we prefer to allocate in a free pool or to allocate a new pool. This ensures that we only allocate into pools that do have most of their slots available for young allocation, which reduces situations where a lot of old values has to be needlessly scanned by the minor GC.

The rings are managed in such a manner that pools that are less than half-full are moved to the start of the ring. This ensures that it is easy to find an old pool for allocation: we only have to look at the first one.

Care is taken so that programs that do not have any root allocated, do not pay any of the costs. At each collection, the currently-allocating pool is reset, so that a domain that does not allocate any root between two minor collections does not have to scan anything. In addition, pools that become empty are immediately moved to the free ring, which is not considered for scanning.

Alternative generational optimizations We initially tried several other approaches to make box-roots "generational", in particular:

1. allocating old values directly into old pools, by classifying the values during allocation;
2. adding young roots to OCaml's remembered set (a data structure used by OCaml's write barrier), instead of scanning them ourselves during minor collections. To avoid repeatedly adding the same root to the remembered set in situations like function-local roots, an auxiliary freelist is added to each pool, as a cache of already-added young roots. (This has been suggested by Stephen Dolan.)

We realised that in both cases, performing instead the classification during scanning results in fewer classifications performed overall, for a significantly reduced implementation complexity. We have mentioned above other benefits of doing the classification during scanning.

3.2.2 Multicore implementation

In OCaml 4 and earlier, all threads run sequentially by holding a master lock. OCaml 5 relaxes this constraint by allowing several domains to run in parallel. An OCaml domain is a collection of (green or system) threads, running sequentially by holding the *domain lock*, OCaml 5's per-domain equivalent of the master lock. The different threads on the same domain share many data structures, most notably the same minor heap. Programmer are given the advice to allocate roughly as many domains as CPU cores to use.

OCaml 5.0 introduces a new garbage collector implementation (Sivaramakrishnan et al., 2020), with a stop-the-world phase to collect minor heaps in parallel (similar in this aspect to previous efforts by Bourgoin et al., 2020), and an implementation of a mostly-concurrent mark & sweep collector for the major heap. The communication between domains is defined by a new memory model (Dolan et al., 2018).

Boxroot for OCaml 5.0 allocates, for each domain, a set of pool rings. Each domain-local set of pool rings is protected by the local domain lock. Every allocation is performed in the domain-local pools. As for deallocations, each one is classified into: *local*, *remote domain*, *purely remote*. (We have implemented improvements to the OCaml 5.0 C API to let us perform this classification efficiently.)

- *Local deallocations* are done on the same domain, while holding the domain lock. They are performed immediately without any synchronisation being necessary.
- *Remote-domain deallocations* are those done from a different domain than the one that has allocated the boxroot, while holding the remote domain's lock. The typical use-case is sending OCaml values between threads, or foreign data structures containing such values. Since *some* domain lock is held, we know that no interference with scanning is possible. The deallocated root is pushed on a remote free-list using two RMW atomic operations (no CAS). The remote free list is pushed back on top of the domain-local free list at the start of scanning, which takes place during a stop-the-world section, when no other remote deallocation can take place.
- *Purely-remote deallocations* are done without holding the domain lock. The typical use-case is for the clean-up of foreign data structures that store OCaml values when the domain lock is released (e.g. during a panic in Rust). This is rarer, so its performance is secondary. To avoid the interference with scanning (without making scanning very slow), each pool has a mutex that needs to be locked during purely-remote deallocations. This mutex is also locked before scanning the pool.[9] In all other aspects, the purely remote deallocation is treated like a remote-domain deallocation.

4 Boxroots in Rust

Boxroots have been introduced in the service of a caller-roots resource-safety protocol, which we describe alongside the Rust data type for boxroots.

[9] As an alternative implementation, one could avoid locking during purely-remote deallocations by storing the metadata about delayed deallocations in a bitmap (taking 32 additional words in the header for our 16KB pools). By storing this metadata separately, one can register remote deallocations in parallel with scanning.

4.1 GCs in Rust — Related works

This work can also be seen as safely adding a GC to Rust. As far as the safe treatment of GCs in Rust is concerned, various prototypes have inspired this work:

- Goregaokar (2015) has proposed GC values as resources, where 1) being a root is tracked as a property of the value using trait-based mechanisms, 2) the GC is non-moving (whereas many of our issues arise when allocating inside the GC heap can invalidate pointers).
- Dolan (2018) has proposed a proof of concept emulating the callee-roots OCaml FFI discipline with Rust macros. The initial version of `ocaml-interop` extended this proof of concept into a library. The macro-heavy approach made its soundness more difficult to reason about, and less idiomatic for the user, than the current version using Boxroot.
- Jeffrey (2018) has proposed the safe manipulation of a JavaScript GC in Rust with a caller-roots protocol. The difficulty of applying this work directly made us notice the importance of the conceptual distinction between callee-roots and caller-roots. Here, roots are registered with stack-allocated linked lists.

4.2 Background about Rust

The Rust language (Matsakis and Klock II, 2014; Anderson et al., 2016) is a modern systems programming language, that eliminates a large class of bugs typically found in C and C++ programs using various techniques. Rust achieves this without the need for a GC. Most notably, a notion of ownership and borrowing is used for resource-management and control of aliasing.

For instance, Rust offers a type of *vectors* (dynamic arrays) similar to that of C++11, but its type system prevents writing programs subject to iterator invalidation. Iterator invalidation describes a class of bugs that happen when a reference into a data structure is used, but the data structure has been changed via an alias, causing its internal reorganisation. Iterator invalidation can be hard to spot because it results from some action at a distance.

The following Rust program attempts to perform an iterator invalidation:

```
fn main() {
    let mut vec = vec!['a', 'b', 'c'];
    let first = &vec[0];
    vec.push('e');
    let mut _vec2 = vec!['f', 'g', 'h'];
    println!("vec[0] contains {}", first);
}
```

In words, a vector `vec` of three elements is allocated. A reference to the first element of the vector is given the name `first`. Then a fourth element is added at the end. A second vector `_vec2` of three elements distinct from the first ones is allocated. Then one shows the contents of the reference `first`.

Writing the same program in a language such as C++ can (and does) result in showing the contents of the first element of `_vec2` instead of `vec`, which has no relationship whatsoever with the

meaning of the program! This happens when adding the fourth element reallocates `vec` elsewhere in memory, and the recycled memory happens to be used for allocating `_vec2`.

In Rust, this does not happen. The compiler outputs the following error message:

```
error[E0502]: cannot borrow `vec` as mutable because it is also borrowed as immutable
 --> src/main.rs:4:5
  |
3 |     let first = &vec[0];
  |                  --- immutable borrow occurs here
4 |     vec.push('e');
  |     ^^^^^^^^^^^^^ mutable borrow occurs here
5 |     let mut _vec2 = vec!['f', 'g', 'h'];
6 |     println!("vec[0] contains {}", first);
  |                                    ----- immutable borrow later used here
```

In words, Rust's type system detects that the reference `first` might be invalid, following the modification of `vec` using its original alias. The Rust type system has a notion of *lifetimes* (or regions) to control aliasing (among others). `first` borrows immutably from `vec`, and Rust sees that this borrow lives until the end of the function. It then sees that a mutable borrow of `vec` is required for adding an element to it (by definition of the vector interface), while the first borrow still lives, which is incompatible with the borrowing discipline: *at any point, one can have either one mutable borrow, or any number of immutable borrows.*

4.2.1 Garbage collection as iterator invalidation

Let us briefly recapitulate the basic common idea in Jeffrey (2018) and Dolan (2018) for dealing safely with a moving GC in Rust. The problem with a moving GC for a language such as Rust is that the GC can invalidate pointers through some action at a distance.

As we have seen, this idea is not foreign to Rust: indeed, something similar happens when appending to a vector requires to reallocate its backing memory. The approach for dealing with a moving GC in Rust can therefore be understood with the help of an analogy sketched in the following table.

Vector	GC heap	Role
`v : Vec`	`gc : GC`	Owns the elements
iterator	GC values	Borrows direct access to the elements
appending	allocating	Can invalidate pointers
index	**root**	Invariant name for the elements

In words, we pass around a *linear capability* (Fluet et al., 2006), `gc`, representing the permission to access the GC heap (in our case in the form of a mutable or immutable borrow of a token, denoting the permission to access the runtime data-structures of the current OCaml domain). This capability lets us to borrow direct pointers into the GC heap, in the form of *non-rooted* GC values understood as copiable borrows of `gc`.

Allocation is registered as a mutation of `gc`, and due to this, the Rust type system prevents us from using the values seen as borrowing from `gc` after an allocation. In order to access these values after allocation, we must refer to them through an invariant name, one preserved across pointer invalidation, like an index in the case of vectors. A *root* is a Rust value that gives an invariant name to a chosen GC value. The notion of root is usually a low-level GC implementation detail, but now we make it an abstraction in the language.

4.2.2 Alternative: GC types as Copy types

A different and natural idea is that GC-allocated values should be manipulated in Rust as plain copiable values, at least as far as variables are concerned. In theory this can be achieved either with conservative scanning of the stack, or by modifying the Rust compiler so that it would produce the information the GC needs to find GC roots in the stack. Coblenz et al. (2022) proposes an evaluation for such a language design, which would seem to remove the need for our GC roots as an abstraction. In addition, Rust's LLVM back-end is in theory able to emit precise stack maps, which would make the scanning of roots on the stack more efficient than Boxroot for this usage (we are not aware yet of a modified Rust compiler which is able to emit such stack maps).

However, the two approaches are complementary:

- Boxroot also solves the questions of how to store a GC-allocated value inside an arbitrary foreign value (not owned by the OCaml GC), and the two mechanisms to register roots can be combined;
- the approach in Coblenz et al. (2022) requires a modified compiler, whereas Boxroot is more readily applicable to arbitrary combinations of languages without having to build support into the compiler.

4.3 The runtime capability

As in previous works (Jeffrey, 2018; Dolan, 2018), we pass around a linear capability in the form of mutable or immutable borrows of a resource of type `Runtime` defined below. This linear capability represents the ownership of a domain lock, and therefore must remain local to the thread.

```
pub struct Runtime {
    _private: (),
}

impl !Send for Runtime {}
impl !Sync for Runtime {}
```

The last lines are pseudo-code for disabling the `Send` and `Sync` traits for the type `Runtime`, which prevents sending and sharing this capability between threads.

4.4 OCaml non-rooted values (the C `value` type)

The next type represents OCaml values (compatible with the type `value` in C) that are always safe for reading transitively. In particular they immutably borrow from the capability if needed to safely read the data (i.e. if they transitively point to the OCaml heap).

```
#[derive(Copy, Clone)]
#[repr(C)]
pub struct Value<'a> {
    raw: isize,
    _marker: PhantomData<&'a Runtime>
}
```

Since they must always be safe to read, these values must be invalidated when an allocation takes place if necessary (this can move or deallocate the value if it resides on the OCaml heap). This is the role of the lifetime annotation.

Accessing a field of a value gives a value with the same access permission:

```
pub fn Value::get<'a>(self: Value<'a>, offset: usize) -> Value<'a>
```

We do not know *a priori* whether the value is a block and if so, what its length is. Let us just decide that the function panics (raises an exception) if the value does not have the right memory layout (let it test whether it is a block, and read the length from the block header). In this paper, we do not address the question of matching OCaml types to Rust types in terms of memory layout: this is a different question from that of co-existing with the GC, the one that interests us here. This question aside, we can implement this function safely thanks to the lifetime discipline.

4.4.1 Examples

```
pub fn val_long(n: isize) -> Value<'static>
```

This function signature can be for converting a Rust integer into an immediate OCaml value (the macro `Val_long` in C). Since it is always safe to read from an immediate value, the lifetime is unconstrained (`'static`).

```
pub fn caml_alloc<'a>(gc: &'a mut Runtime, size: usize, tag: u8) -> Value<'a>
```

This function signature can be for allocating a block of size `size` (in words) and tag `tag` in the OCaml heap. It requires a mutable borrow of the capability, and returns a value with the same lifetime. Therefore calling `caml_alloc` invalidates existing non-rooted pointers into the OCaml heap, and calling it a second time invalidates the value resulting from the first call (which should better have been stored in a root in the meanwhile).

Note The type `Value<'a>` is intended to be copiable. However there is currently a well-known limitation of the Rust type system which makes it believe that the value returned from `caml_alloc` mutably borrows from the capability, rather than immutably. This can force introducing more roots than necessary, and unfortunately this pattern is easy to come across in our context (Section 4.9).

4.4.2 Non-allocating functions

We can now write and call functions that receive OCaml values as arguments, but that do not allocate on the OCaml heap:

```
fn f<'a>(x: Value<'a>, y: Value<'a>) -> Value<'a>
fn g<'a,'b,'c>(gc: &'a Runtime, x: Value<'b>, y: Value<'c>) -> Value<'a>
```

- The first function signature is for accessing the OCaml heap but only through the values `x` and `y`.
- The second function signature is for accessing `x` and `y`, and also runtime functions that do not allocate (for instance `caml_named_value`, which returns a named global value allocated in an OCaml program).

It is now possible to call allocating functions that do not receive GC-allocated values as arguments, as well as non-allocating functions that do receive arguments. It is tempting to consider the following signature for a function that can allocate on the heap while accepting values as arguments (for instance, a function that would allocate the pair of `x` and `y`):

```
/* Discouraged */
fn g<'a,'b,'c>(gc: &'a mut Runtime, x: Value<'b>, y: Value<'c>) -> Value<'a>
```

However, while this signature can be written, it might not mean what you think. The only way for passing values `x` and `y` to this function is if their lifetimes `'b` and `'c` do not overlap with `'a`. In our case, this means that `x` and `y` must be allocated outside of the OCaml heap, that is, they must be immediate or static.

4.5 Rooting with Boxroot

Moving to solve the last problem, we now define a type of boxroots:

```
#[repr(C)]
pub struct Root {
    cell: std::ptr::NonNull<std::cell::UnsafeCell<Value<'static>>>
}
```

In words, a `Root` is a non-null pointer to an `UnsafeCell` containing a `value`. The memory layout of `Root` is the same as the C type `value *`.

The Rust type `UnsafeCell` tells the compiler that this memory location is possibly aliased and could be changed by someone else (in our case, by the OCaml GC, when promoting young values). This is necessary to avoid over-eager optimisations Rust might do based on its deep understanding of (non-)aliasing of variables.

```
pub fn Root::create<'a,'b>(gc: &'a Runtime, val: Value<'b>) -> Root
pub fn Root::get<'a,'b>(self: &'b Root, gc: &'a Runtime) -> Value<'a>
pub fn Root::drop(&mut self)
```

```
unsafe impl Send for Root {}
unsafe impl Sync for Root {}
```

- `create` wraps `boxroot_create`. It requires the domain lock being held, and a readable value. It returns a boxroot. (In case `boxroot_create` returns `NULL`, it panics.)
- `get` wraps `boxroot_get`. It requires the domain lock to be held, and returns a value that borrows the capability passed in argument.
- `drop` is used to implement the trait `Drop` for Root (to implement the RAII pattern). Its use has no constraint apart from ownership of the argument, not even the domain lock being held.
- The `Send` and `Sync` traits let boxroots be sent and shared accross threads. The ownership of a boxroot can for instance be shared between threads by inserting it inside an atomic reference-counted pointer.

4.5.1 The caller roots the arguments

This leads us to a first solution for writing a function that can allocate on the OCaml heap while accepting additional values as arguments.

```
fn h<'a>(gc: &'a mut Runtime, x: Root, y: Root) -> Value<'a>
```

Notice that a `Root` does not carry a lifetime. Being independent from the `gc` capability is its main characteristic. The above signature can therefore accept any `Root` as argument, and we have seen that we can create a root from any `Value`. This is a proper solution to our problem.

However, by taking ownership of boxroots, the callee deprives the caller from its roots. The latter might for instance have to allocate duplicate roots if it needs the values later. An obvious solution is to borrow the roots instead.

```
/* Discouraged */
fn h2<'a,'b,'c>(gc: &'a mut Runtime, x: &'b Root, y: &'c Root) -> Value<'a>
```

This works but has several drawbacks:

- The memory layout of `&Root` is essentially `value**`: we pay the cost of a double indirection.
- This function only accepts values rooted with `Root`. In practice, though, `h2` does not care how the value is kept alive. Nothing should prevent it from accepting values that are static or immediate (without forcing those to be rooted needlessly), or values rooted by other means.

4.6 `ValueRef`, a type of OCaml values with reference semantics

Alongside the `Value` type which has value semantics, we introduce the type `ValueRef` which represents OCaml values with reference semantics. It is essentially a `volatile value *` in C.[10]

[10] In the current implementation of the OCaml 5 runtime, `volatile value` is a proxy for an `_Atomic(value)` accessed via relaxed loads and stores.

```
#[repr(C)]
pub struct ValueCell {
    atomic: AtomicIsize
}

type ValueRef<'a> = &'a ValueCell;
```

In words, a `ValueRef` is a reference to a `ValueCell`. A `ValueCell` is an atomic `value`. The atomicity plays two roles:

- it lets us define `ValueRef`s that point into the OCaml heap. To satisfy the OCaml memory model, the loads from the OCaml heap must be relaxed atomic.[11]
- it replaces `UnsafeCell` in its role, letting the compiler know that the contents can change at the whim of the GC.

A `ValueCell` has no lifetime annotation. Instead, the lifetime of the reference determines a `ValueRef`'s validity.

A `ValueCell` is meant as a polymorphic container, which can represent values that are rooted or not, allocated the OCaml heap or not (static), etc. Examples:

```
pub unsafe fn Value::get_ref<'a>(self: Value<'a>, offset: usize) -> ValueRef<'a>
pub fn Root::get_ref<'a>(self: &'a Root) -> ValueRef<'a>
```

The two functions compile merely to identity/pointer arithmetic.

- In the first case, a pointer to the field `offset` is returned, possibly inside the OCaml heap. Its lifetime is the same as that of the original value. Thus, if the reference points inside the heap, it will not be possible to access it after an allocation.
- The second one is `boxroot_get_ref`. The `ValueRef` points to a rooted location, and its lifetime is that of the boxroot. If an allocation happens, the value is kept alive by the GC, and the location kept up-to-date, so the `ValueRef` keeps on living.

A `ValueCell` can be dereferenced only when holding the domain lock.

```
pub fn as_val<'a,'b>(gc: &'a Runtime, val: ValueRef<'b>) -> Value<'a> {
```

Reading the value pointed-to by the `ValueRef` produces a non-rooted value. Its lifetime is thus constrained by the capability. Indeed, we have just seen that `'b` can outlive `'a` in reasonable situations (case where the `val` is a root), and therefore cannot be used as the lifetime of value.

[11] A bit stronger than a relaxed load actually, but this is a different question.

4.6.1 Expanding the caller-roots protocol

We implement the `Deref` trait for roots, in such a way that references to roots are convertible into `ValueRef`.

```
impl Deref for Root {
    type Target = ValueCell;
    fn deref(&self) -> ValueRef { .../* boxroot_get_ref */ }
}
```

This trait is treated specially by Rust, and used to automatically convert `&root` to the type `ValueRef` when `root : Root`.

It is now possible to rewrite `g` and `h2` above as follows:

```
fn g2<'a,'b,'c>(gc: &'a Runtime, x: ValueRef<'b>, y: ValueRef<'c>) -> T
fn h3<'a,'b,'c>(gc: &'a mut Runtime, x: ValueRef<'b>, y: ValueRef<'c>) -> T
```

where `T` can be `Value<'a>` and `ValueRef<'a>`.

The function signature `g2` is functionally similar to `g`; this aims to show that one could dispense of `Values` altogether for simplicity.

Like `h2`, a function with the signature of `h3` can allocate on the OCaml heap while receiving borrowed roots as arguments. It is better than `h2`:

- it avoids the double indirection,
- it accepts a greater variety of arguments.

Indeed, we implement for instance the `Deref` trait for values, in such a way that references to non-rooted values are automatically converted into `ValueRef`.

```
impl<'a> Deref for Value<'a> {
    type Target = ValueCell;
    fn deref(&self) -> ValueRef { .../* identity */ }
}
```

With this it is possible to pass, by reference, immediate and static OCaml values to `g2` and `h3`.

4.7 Summary

- `Value<'a>` has value semantics, it can represent static and immediate values, or pointers to blocks allocated on the heap (in which case the lifetime `'a` is bounded by the next allocation).
- `ValueRef<'a>` has reference semantics. It is a polymorphic type that can abstract a variety of pointers to OCaml values: immediates, static values, non-rooted values, values inside the OCaml heap, roots (boxed, local, global), maybe more. The lifetime `'a` may or may not be bounded by the next allocation.

To more accurately represent the OCaml memory model, we can further define:

- other types with value semantics (e.g. a type for values that are are still being initialized, `ValueUnshared`, and one for the encoded raised exceptions returned by some of OCaml's runtime functions, `ValueExn`)
- other types with reference semantics (e.g. types for mutable and atomic locations in the OCaml heap, `&MutCell` and `&AtomicCell`).

We have seen 3 kinds of functions that manipulate OCaml values, distinguished according to how they depend on the capability capability:

1. **Ø**: No runtime access (`fn f(...) -> T`). The function can access the heap via supplied arguments, but it cannot call runtime functions nor dereference `ValueRef`s.
2. **R**: Read-only runtime access (`fn f(gc: &Runtime, ...) -> T`). The function can call runtime functions which do not trigger a GC, and it can dereference `ValueRef`s.
3. **RW**: Read-write runtime access (`fn f(gc: &mut Runtime, ...) -> T`). The function can call allocation functions as well as release and re-acquire the runtime lock.

In each case, the return type `T` can contain `Value`s, `ValueRef`s, and `Root`s.

The following table summarises which function signature accepts which kinds of arguments.

	Ø	R	RW
Types with value semantics			
`Value<'a>`	Yes	Yes	Yes (if static or immediate)
`ValueExn<'a>`	Yes	Yes	No*
`ValueUnshared<'a>`	Yes	Yes	No*
Types with reference semantics			
`ValueRef<'a> = &'a ValueCell`	No*	Yes	Yes (if rooted, static, or immediate)
`&'a MutCell`	Yes	Yes	No*
`&'a AtomicCell`	Yes	Yes	No*
Rooted values			
`Root`	No*	Yes	Yes

*: Not in a useful way

4.8 Example

One can easily pass immediates to an allocating function:

```
let val : Value<'static> = val_long(42);
h3(gc, &val, &val);
```

If you forget that one must pass references, you can count on Rust error messages to help you:

```
745 |     h3(gc, val, val);
    |            ^^^
    |            |
    |            expected `&ValueCell`, found struct `Value`
    |            help: consider borrowing here: `&val`
```

One can also easily pass non-rooted values to non-allocating functions.

```
let val2 = caml_alloc(gc, 1, 0);
f(val, val2); // by value
g2(gc, &val, &val2); // by reference
```

It is not possible to pass non-rooted values to allocating functions

```
//h3(gc, &val, &val2); // Compilation error: good
```

The Rust compiler complains along the following lines:

```
769 |     let val2 = caml_alloc(gc, 1, 0);
    |                           -- immutable borrow occurs here
...
775 |     h3(gc, &val, &val2); // Compilation error: good
    |     --^^^^^^^^^^^^^^^^
    |     |
    |     mutable borrow occurs here
    |     immutable borrow later used by call
```

This is the same error message that we had with iterator invalidation. The programmer gets around this by introducing a root:

```
let val2_root = Root::create(gc, val2);
let val3 = h3(gc, &val, &val2_root);
g2(gc, &val3, &val2_root);
```

Lastly, at this point, `val2` is not longer live:

```
//g2(gc, &val, &val2); // Compilation error: good
```

4.9 Rust limitations

4.9.1 Return-value pollution

The attentive reader will have noticed that there is still something wrong with the previous example. As we have mentioned in 4.4, the Rust type system has a vexing limitation. `val2`, the result of `caml_alloc`, is (erroneously) understood by Rust as borrowing from `gc` *mutably*. Consequently, it should not even be possible to call `Root::create` with both `gc` and `val2` as arguments.

There are various ways around this, but there is nothing fundamental about it.

1. It is possible to break the lifetime dependency by casting the lifetime away and re-borrowing the capability immutably. Using domain knowledge, we know that this is safe. Then, using the expressive macro system of Rust, we can try to automate this trick.

2. Alternatively, it makes sense not to require the `gc` capability for the boxroot operations. It can be replaced with a dynamic check that the domain lock is held, which panics (raises an exception). Indeed, this can be convenient because the situation where one needs to introduce a root and the one where the value does not already denote that the domain lock is held (i.e. it is static or immediate) tend to be mutually exclusive. Alternatively, such false roots can be `malloc`ed separately and never scanned.

This is not the only limitation of Rust that we have encountered during our research.

4.9.2 Container pollution

Another limitation which we have encountered is with implementing the destructor for boxroots. We are treating specially inside the implementation of Boxroot the case where the domain lock is not held during deallocation. This is treated as a remote deletion. One reason, the good one, is that one purpose of Boxroot is to store OCaml values inside foreign data structures. In this use-case, one might legitimately want to be able to clean-up the structure anytime. This means deleting the root without necessarily holding the domain lock.

There was also another reason for this choice. Rust does not seem to offer a good way of ensuring that the destructor can only be called while the capability is still live (if necessary by imposing constraints on where the roots can live). Not having any functional constraint on the deallocation of boxroots sidesteps this issue.

However this hides a more general limitation: it does not seem possible to obtain the capability inside of a destructor, even when this destructor runs while the capability is in scope and borrowable. Thus, this causes some difficulty if the release of a resource requires using the OCaml runtime (for instance, calling some OCaml code).

Again there are several workarounds, none ideal:

1. Avoiding the automatical destruction of resources whose clean-up needs the capability. By manually destroying them instead, it is possible to pass the capability as an argument to the manual destructor.

2. Providing a function to create a new capability out of nothing inside the destructor. This is highly unsafe, but destructors are currently a dark corner of Rust where other unsafe manipulations take place currently.[12] A refinement of this idea is to dynamically check whether the domain lock is held, and wrap the new capability in an option type (this is still unsafe).

Ideally, one should be able to hold a borrow of the capability inside the resource that is going to be destroyed, for later use. But holding such a borrow in the Rust type system will prevent any allocation until this resource is destroyed.

[12] Cf. `#[may_dangle]`.

This limitation is related to a failure by Rust to distinguish the borrowing of a container, from the borrowing of its contents. For instance:

```
let mut x = Box::new(2);
let x_ref : &Box<u8> = &x;
*(x.as_mut()) = 3;
println!("x contents: {}", x_ref.as_ref());
```

In words, a memory cell is allocated containing the integer 2. A reference to this memory cell is taken, then the *contents* are modified to hold the integer 3. One then tries to print the contents of the memory cell through its reference.

Rust fails with the following error message during compilation:

```
error[E0502]: cannot borrow `x` as mutable because it is also borrowed as immutable
 --> src/main.rs:4:6
  |
3 |     let x_ref : &Box<u8> = &x; // (1)
  |                            -- immutable borrow occurs here
4 |     *(x.as_mut()) = 3; // (2)
  |      ^^^^^^^^^^^^ mutable borrow occurs here
5 |     println!("x contents: {}", x_ref.as_ref());
  |                                -------------- immutable borrow later used here
```

Rust understands that the mutation of the contents invalidated the reference to the container. Arguably, though, it would make sense for the program to print the new contents, 3.

In our problem, we replace `x_ref` with the creation of a resource that borrows from the `gc` capability, and we replace `println` with the call to the destructor of the resource (inside which we try to access the capability via its borrow).

It should be noted that both this limitation and the previous one have been noticed separately in the Rust community, and several proposals have been drafted to attempt to relax the discipline. However, no convincing solutions have been proposed yet.

4.10 Local roots in Rust

Previously, Dolan (2018) has demonstrated a safe lifetime discipline for OCaml's local roots. We worked on improving this approach by making it less macro-heavy and more idiomatic to Rust; our takeaway is that it can be done, but it is hard and the result is unpleasant to use.

One has to model the fact that at the beginning of the function one can access either the GC (e.g. allocate), or access the `value` arguments, but not both at the same time. This involves passing to every function a capability which is built from several ingredients:

- a dependent ownership type (`OwningHandle` from the crate `owning_ref`, a sort of linear Σ-type) to express that the permission to access the values is dependent on the permission to access the GC: one can recover a capability to access the GC only by relinquishing access to the values (in other words the user is forced to root the values if they want to keep using them after recovering the GC capability);

- a `LCell` from the crate `qcell`, a container that allows to track a permission to access values using a capability separate from the values themselves, with a phantom type.

It requires a lot of boilerplate when calling and entering functions, which is mostly erased at compilation. But it lets us use the Rust type system to prevent making mistakes.

With Boxroot we have found a simpler solution to the problem of passing the GC capability along with value, which does not need such elaborate concepts. But this is nevertheless interesting since it makes it possible to have a zero-overhead safety layer over the existing OCaml runtime C API, which is to remain callee-roots.

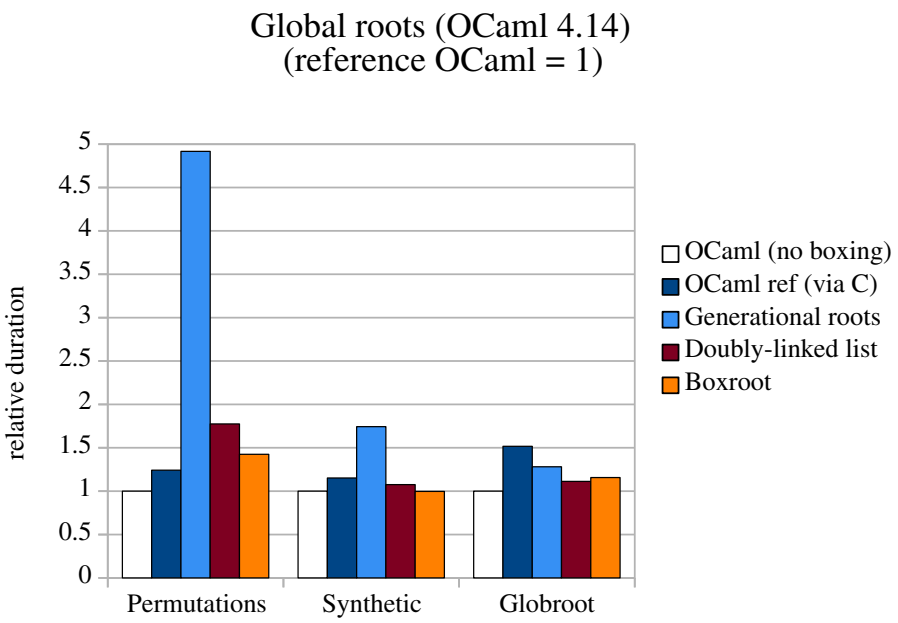
5 Benchmarks for Boxroot

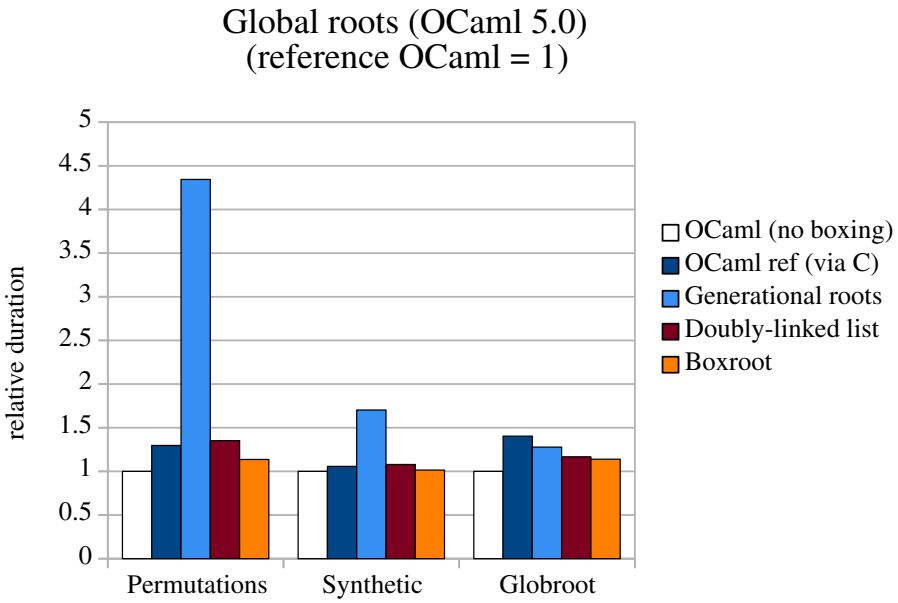
In most cases in current usage, FFI code is not very root-intensive and the performance impact of rooting is likely to be minor; but for root-heavy benchmarks, boxroot performs similarly or better than other mechanisms.

We have devised a few benchmarks to guide us in the implementation of Boxroot. Three benchmarks compare boxroots to global roots, one benchmark compares them to local roots. At the time of writing, our benchmarks are single-threaded, so what is being tested is the implementation of a concurrent allocator, but without any contention. We will mention other limitations for each benchmark.

The figures below are obtained with OCaml 4.14 and OCaml 5.0 (development version), on a common commercial laptop with CPU AMD Ryzen 5850U.

5.1 Global roots benchmarks





Our benchmarks compare various implementation of an OCaml type representing a memory cell containing a single value, that has the same interface as boxroot (`create`, `get`, `delete`, `modify`):

- **OCaml (no boxing)**: a pure OCaml implementation where `create` is the identity and nothing happens on deletion. No C call is performed.
- **OCaml ref (via C)**: a C implementation of a mutable record (using `caml_alloc_small(1,0)`). Deletion is implemented by assigning the OCaml integer `0` to the contents. Scanning is done via the value being reachable from the OCaml heap.
- **Generational root**: idem, but using malloc and a generational global root.
- **Boxroot**: a boxroot, using the implementation we described in Section 3.2.
- **Doubly-linked list**: a variant of `boxroot`, but using a simpler implementation with doubly-linked lists (Section 3.1.2). This implementation is not thread-safe: it does not support multiple domains nor deletion when the runtime lock is released.

The various implementations except "OCaml" have similar memory representation, some being reachable from the OCaml heap and some outside of the OCaml heap. The ones outside of the OCaml heap are disguised as immediate values for the OCaml GC, by using a standard pointer-tagging trick to cast them as values (`(value)p | 1`).

We can expect *a priori* that "OCaml" is always faster, since it has none of the overheads of other methods. This is meant as a baseline, but its existence also highlights a limitation of our global roots benchmarks: "Boxroot" is intended for applications where values are *not* easily reachable from the OCaml heap and thus where neither "OCaml" nor "OCaml ref" are available.

5.1.1 *Permutations* benchmark

The small program used in this benchmark computes the set of all permutations of the list `[0; ..; n-1]`, using a non-determinism monad represented using (strict) lists. (This is an expensive way of computing $n!$ with lots of allocations.)

In our non-determinism monad, each list element is wrapped as described above.

This benchmark creates a lot of roots alive at the same time. It allocates 1860 boxroot pools, and performs 1077 minor and 18 major collections. Roughly 22M boxroots are allocated in total.

time in seconds	OCaml 4.14	OCaml 5.0
OCaml	1.20	1.25
OCaml ref	1.49	1.69
Generational roots	5.90	5.43
Doubly-linked list	2.13	1.69
Boxroot	1.71	1.42

We see that generational roots add a large overhead. In OCaml 4.14, Boxroots out-perform generational global roots, and give slightly slower time than equivalent pure-GC implementations.

The performance of the simple implementation with doubly-linked lists is already very good compared to generational roots. However, keep in mind that this does not take into account the costs of making it thread-safe. We also believe that the cache-locality-awareness of the system allocator plays a role in its performance, given that it is the sole consumer for its allocation size-class (essentially the nodes are allocated contiguously in memory, using the same memory pool technique we use for Boxroot). This might not reflect the behaviour of real workloads.

In OCaml 5.0, the performance gives us a hint of considerations entering into boxroot's performance. Here Boxroot unexpectedly outperforms "OCaml ref". Two hypotheses can explain this speedup:

1. Boxroot puts less pressure on the GC. It indeed performs 14% fewer minor collections than "OCaml ref".
2. Boxroot has good cache locality during root scanning. Indeed one major difference with OCaml 4.14 is the absence of prefetching during the major GC. The sensitivity to this effect is confirmed by running the benchmark with OCaml 4.12.0, which also lacks prefetching and shows Boxroot also outperforming its pure-GC counterparts.

5.1.2 *Synthetic* benchmark

In this benchmark, we allocate and deallocate values and roots according to probabilities determined by parameters:

- `N=8`: $\log_2$ of the number of minor generations,
- `SMALL_ROOTS=10_000`: the number of small roots allocated (in the minor heap) per minor collection,

- `LARGE_ROOTS=20`: the number of large roots allocated (in the major heap) per minor collection,
- `SMALL_ROOT_PROMOTION_RATE=0.2`: the survival rate for small roots allocated in the current minor heap,
- `LARGE_ROOT_PROMOTION_RATE=1`: the survival rate for large roots allocated in the current minor heap,
- `ROOT_SURVIVAL_RATE=0.99`: the survival rate for roots that survived a first minor collection,
- `GC_PROMOTION_RATE=0.1`: promotion rate of GC-tracked values,
- `GC_SURVIVAL_RATE=0.5`: survival rate of GC-tracked values.

These settings favour the creation of a lot of roots, most of which are short-lived. Roots that survive are few, but they are very long-lived. The parameters are meant to exaggerate the impact of Boxroot performance. (In contrast, we are not aware of settings for which Boxroot would exhibit pathological behaviour.)

time in seconds	OCaml 4.14	OCaml 5.0
OCaml	6.12	2.82
OCaml ref	7.15	2.98
Generational roots	10.83	4.80
Doubly-linked list	6.68	3.04
Boxroot	6.19	2.86

In OCaml 4.14, this benchmark allocates 772 boxroot pools, and performs 2,619 minor and 141 major collections. Roughly 16M roots are allocated.

Boxroot performs better than other root implementations, but it is unexpected again that it outperforms "OCaml" and "OCaml ref". This is not well-understood. First, unlike the previous benchmarks, "OCaml ref" does actually fewer minor and major collection. Second, running with or without a prefetching GC shows comparable performance between Boxroot and "OCaml".

In OCaml 5.0, this benchmark allocates 408 boxroot pools, and performs 2,615 minor and 55 major collections. 8.7M roots are allocated. Therefore results are not comparable between major the old and the new GC; the absolute speedup is an artifact of our benchmark. The relative results are similar between OCaml 4.14 and OCaml 5.0, as seen in the figure.

This synthetic benchmark lets us conclude that there is a large improvement over generational roots. We do not draw conclusions regarding its good performance compared to the pure-OCaml version, in the absence of an interpretation of this result.

5.1.3 *Globroot* benchmark

This benchmark is adapted from the OCaml testsuite. It exercises the case where there are about 1024 concurrently-live roots, but only a couple of young roots are created between two minor collections.

This benchmark tests the case where there are few concurrently-live roots and little root creation and modification between two collections. This benchmark does not perform any OCaml

computations or allocations. It forces collections to occur very often, despite low GC work. So the costs of root handling and scanning are magnified, they would normally be amortized by OCaml computations.

In this benchmark there are about 67000 minor collections and 40000 major collections. 217k boxroots are allocated using only 1 pool.

time in seconds	OCaml 4.14	OCaml 5.0
OCaml	0.89	0.72
OCaml ref	1.27	1.02
Generational roots	1.14	0.92
Doubly-linked list	0.99	0.84
Boxroot	1.03	0.82

Since there are few roots created between collections, list-based implementations are expected to perform well (their scanning is quick provided that they implement some form of generational optimisation). In contrast, Boxroot has to scan on the order of a full memory pool at every minor collection even if there are only a few young roots, while our pool size is chosen large (2032 slots).

There used to be a noticeable overhead in this benchmark compared to list-based implementations (generational roots, doubly-linked lists, and remembered-set-based boxroots). The overhead *per minor collection* was negligible (~2µs). But we have also reduced it by optimizing the scanning during minor collection: if we know that only young values must be scanned, then we can test by ourselves whether values are young, before applying the scanning action to them. There is a very efficient way of doing this test in a loop. This optimization brought a speed-up of up to 3.8× to the scanning of pools that contain very few young values (2.6× in this benchmark). The overhead per minor collection is now difficult to distinguish from fluctuations.

The relative results are similar between OCaml 4.14 and OCaml 5.0.

5.2 Local roots benchmark

We designed this benchmark to test the idea of replacing local roots altogether with boxroots.

Currently, OCaml FFI code uses a "callee-roots" discipline where each function has to root each local OCaml value in use, using the efficient `CAMLparam`, `CAMLlocal`, `CAMLreturn` macros. These macros manage a shadow stack of roots pointing to function arguments and local variables. (Section 2.1)

Boxroot proposes a "caller-roots" approach where callers package their OCaml values in boxroots, following an ownership discipline (they can be passed to the callee owned or borrowed). Creating boxroots is slower than registering local roots, but the caller-roots discipline can avoid re-rooting each value when moving up and down the call chain, so it is expected to have a performance advantage for deep call chains.

This benchmark performs a (recursive) fixpoint computation on OCaml floating-point values, with a parameter N that decides the number of fixpoint iterations needed, and thus the length of the C-side call chain. Essentially:

```
let rec fixpoint f x =
  let y = f x in
  if 0 = Float.compare x y then y
  else ocaml_fixpoint f y

let () =
  for _i = 1 to num_iter do
    ignore (fixpoint (fun x -> if truncate x >= n then x else x +. 1.) 1.)
  done;
```

The time is then normalised to show the average duration of a single recursive call.

- **OCaml**: the purely-OCaml function `fixpoint` above, no C calls.
- **OCaml ref**: same, but adding one level of indirection for each argument, simulating Boxroot's memory layout.
- **Local roots**: calls the `local_fixpoint` function given in Section 2.1. It is the C equivalent of `fixpoint`, following the documented OCaml-C style using local roots. In addition, `compare_val` compares the values of `x` and `y`, but introduces local roots in order to simulate a more complex computation.
- **Boxroot**: the Boxroot version is as follows:

  ```
  int compare_refs(value const *x, value const *y);

  boxroot boxroot_fixpoint_rooted(value const *f, boxroot x)
  {
    boxroot y = boxroot_create(caml_callback(*f, boxroot_get(x)));
    if (compare_refs(boxroot_get_ref(x), boxroot_get_ref(y))) {
      boxroot_delete(x);
      return y;
    } else {
      return boxroot_fixpoint_rooted(f, y);
    }
  }
  ```

  where `compare_refs` does the same work as `compare_val` but expects its values already rooted. The function `boxroot_fixpoint_rooted` takes two roots as argument: the first one is borrowed, whereas the second one is taken ownership of.

  The work is done by `boxroot_fixpoint_rooted`, but we need a `boxroot_fixpoint` wrapper to go from the callee-roots convention expected by the OCaml-C FFI, to a caller-root convention.

  ```
  value boxroot_fixpoint(value f, value x)
  {
  ```

```
  boxroot f_root = boxroot_create(f);
  boxroot x_root = boxroot_create(x);
  boxroot y = boxroot_fixpoint_rooted(boxroot_get_ref(f), x_root);
  value v = boxroot_get(y);
  boxroot_delete(y);
  boxroot_delete(f_root);
  return v;
}
```

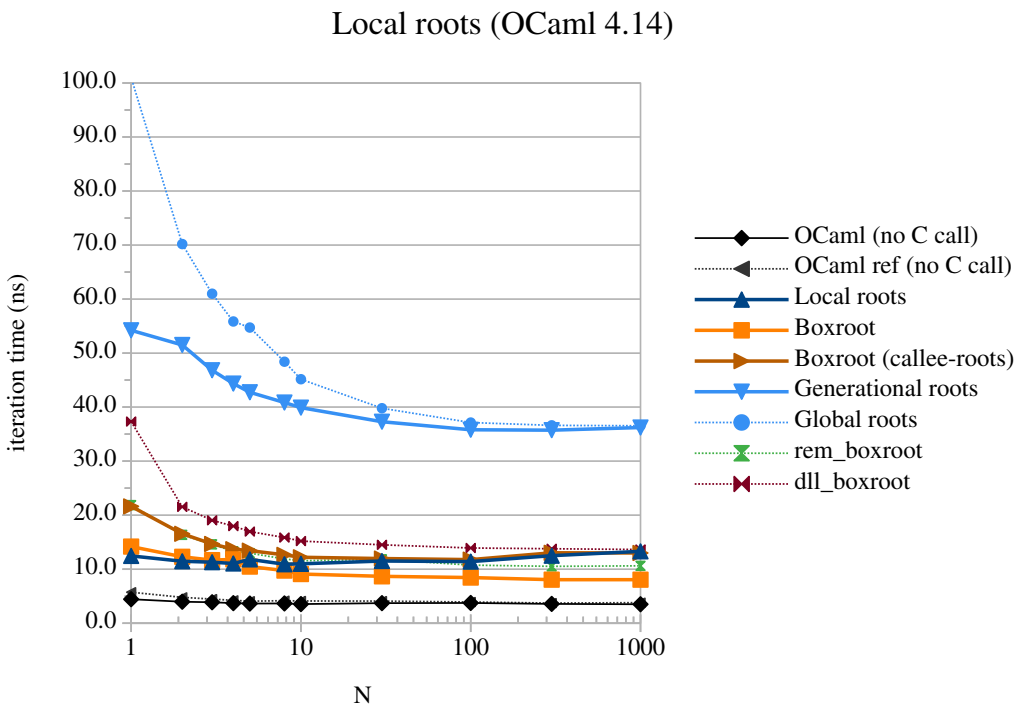
This wrapper adds some overhead for small call depths.

- **Boxroot (callee-roots)**: uses boxroots, but using the same callee-roots discipline as local roots. This control lets us separate the gains due to the differences between rooting protocols from the more measurable performance costs.
- **Generational roots**: same as Boxroot, but using malloc+generatinal roots instead.

We also show earlier experiments that did not implement thread-safety:

- **dll_boxroot**: the implementation of boxroots using doubly-linked lists,
- **rem_boxroot**: a version of `boxroot` which implements a generational strategy based on OCaml's remembered set (Section 3.2.1). It also lacks various optimisations, such as the inlining of the fast paths of `boxroot_create` and `boxroot_delete`.

5.2.1 In OCaml 4.14



We see that, in this test, despite the up-front cost of wrapping the function, boxroots are equivalent to or outperform OCaml's local roots. More precisely, boxroots are slightly more expensive than local roots when following the same callee-roots discipline, and the caller-roots discipline offers huge saves in this benchmark. Analysing the generated assembly of the fixpoint, the saves from the caller-roots discipline come from:

- introducing fewer roots,
- enabling recursion to be done via a tail call,
- enabling better code generation after inlining.

Thus the gains from the caller-roots discipline compared to callee-roots greatly depends on the program and programming style. (We did not take into account local roots optimisations that fall outside of the documented syntactic rules of local roots usage, and require expert knowledge of their implementation. For instance it is possible with local roots, but not documented nor well-known, to rewrite `local_fixpoint` in a tail-recursive manner. In contrast, the optimisations permitted by the caller-roots protocol of Boxroot are intended to be idiomatic to Rust programmers and entirely enabled and made safe by Rust's type system.)

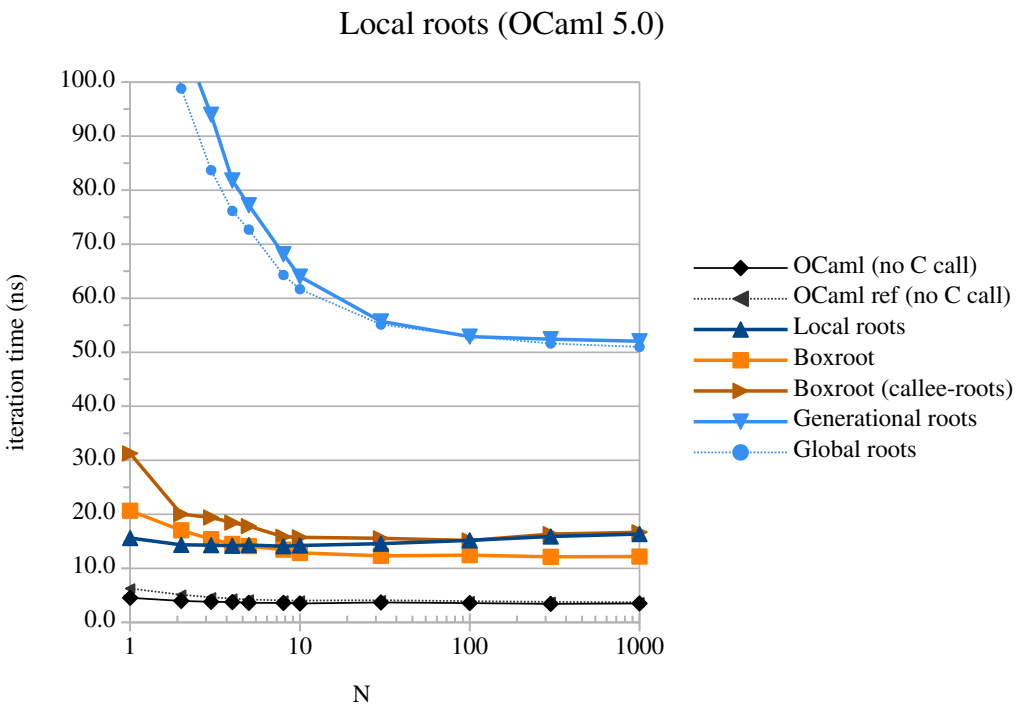
Our conclusions:

- Using boxroots is competitive with local roots.
- It can be beneficial if one leverages the added flexibility of boxroots.

- There could be specific scenarios where it is much more beneficial, for instance when traversing large OCaml structures from a foreign language, with many function calls.

Furthermore, we envision that with support from the OCaml compiler for the caller-roots discipline, the wrapping responsible for the initial overhead could be made unnecessary. The compiler could instead pass roots located on the OCaml stack.

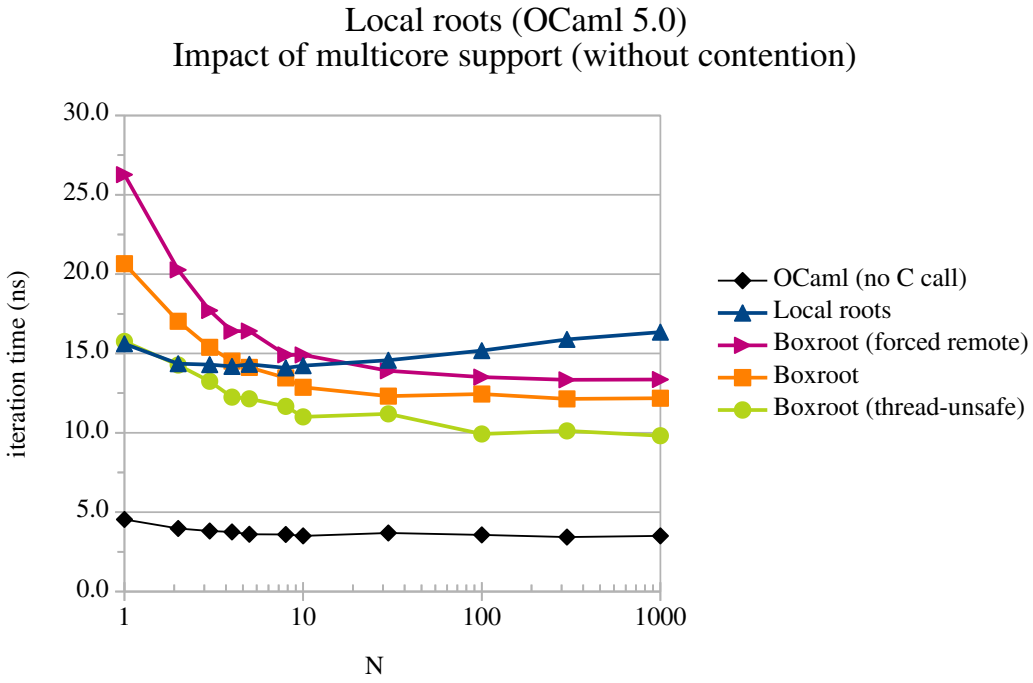
5.2.2 In OCaml 5.0



In multicore OCaml, we observe similar results. However, the overhead of C calls appears to be significantly higher, and generational roots are much more expensive (for two reasons probably: they are now protected by a mutex, and the runtime makes greater use of them internally, which imposes an overhead due to the logarithmic complexity).

Next we have measured the performance of two alternative implementations:

- **Boxroot (thread-unsafe)**: assumes that there is only one thread that never releases the domain lock, and thus avoids related checks in `boxroot_create` and `boxroot_delete`.
- **Boxroot (force remote)**: the opposite, performs all deallocations as if they were done on a different domain, using the lock-free atomic deallocation path.



Our thread-safe implementation of Boxroot for multicore OCaml is slightly slower than a version that does not perform checks necessary for thread-safety. The difference is likely less important in other kinds of situations where more time is spent in cache misses.

The implementation where all deallocations are done remotely is only slightly slower than Boxroot (although with a much higher pool count currently, due to the fact that flushing the remote free-list of each pool is delayed until the next garbage collection). However this single-threaded benchmark does not let us see the costs of cache effects in realistic multithreaded scenarios (cache misses and contention).

Our conclusions for the multicore implementation:

- The overhead of multithreading support is low enough to propose Boxroot as an all-purpose rooting mechanism. (An alternative would be to introduce a specialised version for the case of local roots, which could still be done at some later point. However if micro-optimisations become necessary, the caller-roots discipline already provides plenty of ways of avoiding rooting altogether.)
- The performance of cross-thread deallocation is likely very good, but we need better benchmarks to measure this.

6 Summary

1. We proposed a new data abstraction of the notion of GC root, boxroots. They serve the interface between values managed by a GC and a host language.

Boxroots are resources which follow an ownership discipline. They can be passed to functions by ownership or by borrowing, they can be returned, inserted in foreign data structures, and sent and shared across threads.

2. Boxroots support the safe manipulation of GC values from Rust, extending previous investigations by Jeffrey (2018); Dolan (2018). We presented a caller-roots discipline suitable for dealing with a GC in Rust, which supports passing rooted values both by ownership and by borrow, as well as avoiding unnecessary rooting when a value is static or when a function does not allocate—all thanks to some polymorphism and distinctions made via the lifetime discipline of Rust. A rooting mechanism such as Boxroot is then essential to manipulate GC values under this discipline.

   Our Boxroot library became quickly used in several Rust libraries implementing OCaml-Rust interfaces.

   During our research, we have encountered several limitations of Rust's type system, which are well-known but appear all the more important here. We hope that future improvements of the Rust type system—like it has happened in the past—can make our discipline simpler to use.

3. We presented an implementation of boxroots that performs vastly better than OCaml's generational roots, and significantly better than a naive implementation based on doubly-linked lists. The implementation is also competitive with OCaml's local roots, but more expressive than those. Notably, the gain in expressivity can translate in further performance gains (caller-roots vs. callee-roots protocols).

   Boxroot relies on various implementation techniques, some from modern allocators, some from GCs. It draws expected benefits from modern concurrent allocator technology: cache locality, multicore-friendliness. The cost of making it thread-safe is acceptable. The implementation of cross-thread deallocation is simplified by assumptions made by the garbage collector (regular stop-the-world phases).

   The cost of allocating a boxroot should be expected to be on par with modern concurrent allocators, the cost of scanning the roots on par with modern garbage collectors. This suggests that phase transitions from linearly allocated values to GC-allocated values is zero-overhead.

   The good performance results let us propose Boxroot as a single all-purpose rooting mechanism (at least in the absence of a deeper integration of the GC in the host language: Section 4.2.2).

   Being multi-threading-capable is not only a matter of performance but also of resource safety. Indeed, since one purpose of boxroots is to be stored in foreign data structures, it appears essential that such resources are ready to be cleaned-up anytime. Deleting a boxroot should not require unrealistic constraints, such as acquiring OCaml's runtime lock.

4. We have not yet written benchmarks that exercise the multi-threading capability of Boxroot. All our benchmarks were single-threaded, although measuring an implementation which is thread-safe.

We have also not proposed benchmarks for latency. Indeed we have not implemented an incremental scanning of roots due to limitations of the hooks we use to communicate with the OCaml runtime. This limitation could be lifted after integrating our library with the OCaml runtime, or after making the hooks interface more expressive.

The benefits in terms of alleviating GC pressure should be further investigated. This would require benchmarks that would move significantly more allocations outside of the GC heap, of values that would normally be promoted.

References

Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. Engineering the servo web browser engine using rust. In *ICSE '16*, 2016. doi: 10.1145/2889160.2889229.

David F. Bacon, Perry Cheng, and V. T. Rajan. A unified theory of garbage collection. In John M. Vlissides and Douglas C. Schmidt, editors, *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 50–68. ACM, 2004. doi: 10.1145/1028976.1028982.

Henry G Baker. Lively linear lisp: "look ma, no garbage!". *ACM Sigplan notices*, 27(8):89–98, 1992.

Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *SIGARCH Comput. Archit. News*, 28(5):117–128, nov 2000. ISSN 0163-5964. doi: 10.1145/378995.379232. URL https://doi.org/10.1145/378995.379232.

Frédéric Bour. CAMLroot: revisiting the OCaml FFI, 2018. URL https://arxiv.org/pdf/1812.04905. Proof-of-concept at https://github.com/let-def/camlroot.

Mathias Bourgoin, Benjamin Canou, Emmanuel Chailloux, Adrien Jonquet, and Philippe Wang. Objective caml for multicore architectures. *CoRR*, abs/2006.05862, 2020. URL https://arxiv.org/abs/2006.05862.

Michael Coblenz, Michelle L. Mazurek, and Michael Hicks. Garbage collection makes rust easier to use: A randomized controlled trial of the bronze garbage collector. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1021–1032, 2022.

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.

Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. Bounding data races in space and time. In *Proc. PLDI 2018*, 2018.

Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. Linear regions are all you need. In Peter Sestoft, editor, *Proceedings of the 15th European Symposium on Programming (ESOP 2006)*, volume 3924 of *Lecture Notes in Computer Science*, pages 7–21. Springer, 2006. doi: 10.1007/11693024_2.

Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. 40(6):62–72, 2005. doi: 10.1145/1064978.1065019.

Manish Goregaokar. In pursuit of laziness, 2015. URL https://manishearth.github.io/blog/2015/09/01/designing-a-gc-in-rust/. Blog post entitled: *"Designing a GC in Rust"*.

Manel Grichi, Mouna Abidi, Yann-Gaël Guéhéneuc, and Foutse Khomh. State of practices of java native interface. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, page 274–283, USA, 2019. IBM Corp.

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.

Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook*. Routledge, hardcover edition, 8 2011. ISBN 978-1420082791.

Goh Kondoh and Tamiya Onodera. Finding bugs in java native interface programs. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, page 109–118, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580500. doi: 10.1145/1390630.1390645. URL https://doi.org/10.1145/1390630.1390645.

Yves Lafont. The linear abstract machine. *Theoretical computer science*, 59(1-2):157–180, 1988.

Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, paperback edition, 7 1999. ISBN 978-0201325775.

Nicholas D. Matsakis and Felix S. Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. Retrofitting parallelism onto ocaml. *Proc. ACM Program. Lang.*, 4(ICFP), aug 2020. doi: 10.1145/3408995. URL https://doi.org/10.1145/3408995.