Review of OCaml-Rust bindings

Frédéric Bour
Jacques-Henri Jourdan
Guillaume Munch-Maccagnoni
Gabriel Scherer

September 25, 2020

Contents

1 Purpose and results

1.1 Context

Nomadic Labs maintains a Tezos Node implemented in OCaml. They want to add new features that rely on cryptographic libraries implemented in Rust, so they need to create bindings to Rust libraries for an OCaml program.

Nomadic Labs is experienced with writing bindings to cryptographic libraries in C. When writing C FFI bindings for OCaml, strict rules must be followed when manipulating values owned by the OCaml Garbage collector; there is no static checking that the rules are followed, and errors result in hard-to-track memory corruption later in the program lifetime, but Nomadic engineers are experienced with C-FFI usage and trust themselves to follow the rules (we call this the *memory safety* requirement). In addition to these memory-safety rules, cryptographic bindings have additional requirements: authors wish to ensure that key material is never copied in uncontrolled ways in memory, to ensure that they can correctly erase it on demand.

The OCaml GC assumes that values on the OCaml heap can be copied at will; cryptographic bindings are thus written carefully to store sensitive material outside the OCaml heap, by using external-backed Bigarrays or Custom blocks (we call this the *zero-copy* requirement).

When writing library bindings using Rust directly instead of C, are those two requirements (memory safety and zero-copy) met as expected? This is the question that Nomadic Labs engineers asked us. They presently do not trust existing Rust-OCaml FFI libraries (in particular `ocaml-rs`) enough to use it in their products; they go through C as an intermediate glue language, or try to write simpler, easier-to-reason-about libraries.

Candidates for the "right approach for trustable OCaml-Rust bindings in the current ecosystem" are as follows:

- Using the Rust-C FFI and then the C-OCaml FFI, writing the glue code in C using the C-FFI API that Nomadic Labs feel comfortable with. This is what their engineers are currently considering the safe practice.

  Is there a better way to write Rust-OCaml bindings, without going through C for glue code?

- Using the Rust-OCaml binding library `ocaml-rs` [Shipko, 2019], which is developed outside Nomadic Labs are relied upon by various other projects in the OCaml ecosystem. This would be the standard choice, but Nomadic Labs engineer do not trust this library to respect their zero-copy requirement.

  Can `ocaml-rs`, or a subset of its API, meet the requirements?

- Edward Tate, engineer at Nomadic Labs, wrote an alternative OCaml-Rust binding library as an experiment, `alien_ffi` [Tate, 2019a], which is simpler and smaller. Edward Tate wrote a cryptographic binding on top of it, `ocaml_threshold`.

  Can `alien_ffi` meet the requirements?

It should be noted that none of the code that we have reviewed written by Nomadic Labs has been used in production.

1.2 Our purpose

The main purpose of our survey was to analyze the proposed ways to write OCaml-Rust bindings without glue code, and provide our expert advice on which approach would meet the memory-safety and zero-copy requirements (if at all possible given the existing software ecosystem). Our recommended approach should be explained as a set of clear recommendations that Nomadic Labs engineers can refer to in order to write or review OCaml-Rust bindings.

1.3 Actions

To achieve this demand, we started by comparing the design of `ocaml-rs` and `alien_ffi`, to understand whether they met Nomadic Labs requirement for writing Rust-OCaml crypto bindings in the general case. We found a set of design or implementation defects in either libraries. We followed by carefully reviewing parts of their implementation and bindings using them (to check whether they were affected by these defects), doing small changes to the implementations if necessary, so that we could build a set of recommendations that meet Nomadic Labs' requirements. We also forged a consensus opinion on how the existing library ecosystem should evolve, in both the short and long term, to provide a better basis for safe OCaml-Rust bindings.

1.4 Timeframe

We initially estimated the workload at 2 days for each of us, and in fact ended up working 3 days.

The entirety of our work (which is slightly more than announced in the mid-way summary we sent after the first day) is described in details in this report.

1.5 Results

Yes, we believe that it is possible to write OCaml-Rust bindings without C glue code that meets Nomadic Labs' requirements. The best approach to do this today, in our opinion, is to use a specific subset of `ocaml-rs` that we reviewed to be safe (described in the report below), with additions for custom-blocks handling that we propose.

In the short term, we suggest several important improvements to fix various shortcomings of `ocaml-rs`, in particular:

- We found various bugs that endanger the memory-safety property.

  (We shall report them to the library maintainer, so hopefully they will get fixed shortly.)

- `ocaml-rs` (and `alien_ffi`) rely on version-specific details of the OCaml runtime in fragile ways, and will break under newer OCaml versions. (The current implementation of `ocaml-rs` is already incompatible with OCaml 4.10, which is about to be released.)

- The design of `ocaml-rs`'s API mixes low-level parts that are close to the C FFI, and a higher-level interface that provides convenience in exchange for loss of control of zero-copy properties. The library should be refactored to move the low-level part into a separate library, for everyone to use with clear control.

Recommendation for the future: In the long term, we recommend using Rust lifetimes to statically capture some of the GC invariants, as proposed by the tiny proof-of-concept `caml-oxide` [Dolan, 2018] for Rust/OCaml and in Josephine [Jeffrey, 2018] for Rust/Javascript. Such an API could be build on top of the low-level part of `ocaml-rs`, providing a safer interface over its implementation.

Note: we proposed to work on version-fragility and perform the `ocaml-rs` refactoring as part of our work, provided three extra days of work. Nomadic Labs prefers to do this work internally as a way to build internal knowledge of the problem – possibly requiring some supervision as separate expert work in the future.

1.6 Our contributions: full details

- A comparative design analysis of `ocaml-rs` and `alien_ffi` that was considered by Nomadic Labs for usage in OCaml/Rust bindings.

Section 2.

- A partial code review for `ocaml-rs`, surfacing various implementation issues for a robust binding library.

  Section 3.

- A code review of the concrete binding `ocaml_threshold` [Tate, 2019b] for memory safety. (This was suggested by our design analysis of the `alien_ffi` binding library, which is used in `ocaml_threshold`.)

  Section 4.

- A set of recommendations for binding authors with the current binding-library ecosystem, that suffices to meet Nomadic Labs' requirements (memory-safety and zero-copy).

  In particular, we indicate which parts of the `ocaml-rs` API we reviewed to be zero-copy-correct, and which parts should be avoided.

  Section 5.

- General and longer-term recommendations for a safer basis for binding libraries, such as using lifetimes to capture ownership of the OCaml runtime and following best practices for resource management.

  Section 6.

- Support for `custom` blocks in `ocaml-rs`.

  Section 7.

2 OCaml/Rust binding libraries: A comparative design analysis

2.1 Purpose

`ocaml-rs` is a full-fledged binding library that tries to provide various levels of abstractions to write OCaml-Rust binding. It is not doing any metaprogramming (automatic glue generation from binding declarations; there is a separate Rust crate, `derive-ocaml`, built on top of `ocaml-rs` to do this), but it tries to provide both:

- a low-level interface that corresponds closely to the C-FFI API (in particular, all OCaml values are at an opaque `Value` type), and
- a higher-level interface for conversion between structured Rust values and structured OCaml values, with more precise type information; this is provided mainly by data-conversion traits, that know how to transport certain types and type-formers between the Rust and OCaml world.

`alien_ffi` is an experimental library designed to remain small and simple, easy to reason about, supporting exactly the needs of Nomadic Labs for cryptographic code.

2.2 Memory safety

Context The main difficulty when writing C FFI code, or Rust FFI code with existing libraries, is to correctly handle values owned by the OCaml Garbage Collector. The GC may be called in the middle of the FFI code (typically by allocation functions), and it needs to have full view of which OCaml values are live at this point. Binding authors thus have to declare parameters and temporaries as GC roots, and ensure that non-registered values are never used after a GC call which may have moved and invalidated them.

Bugs arising from wrong handling of GC-owned values are somewhat frequent in binding libraries. They are also very hard to find and diagnose with confidence, because dangerous moves/invalidation only happens rarely, if the GC "runs at the wrong time", in a way that is extremely dependent on the application workload (typically unit tests will not put enough pressure on the GC for this to happen), and result in memory-corruption issues that only reveal themselves after the corresponding data is accessed (typically at the next GC compaction phase), later in the program lifetime. For users writing bindings manually, we believe that the only robust approach in the long term is to follow a very strict discipline (systematically registering all values) and review it carefully – unfortunately, despite old experiments [Furr and Foster, 2005], there currently are no tools in production that can statically analyse bindings for safety.

Libraries `ocaml-rs` provides root-management operations as Rust macros, some that roughly follow the C-FFI style, and some that are slightly higher-level. It provides no additional safety over the C-FFI runtime, but users experienced with writing memory-safe C-FFI code should be comfortable using `ocaml-rs` in the same style.

Defect (`alien_ffi`, important): `alien_ffi` does not provide any root-management primitive. We believe that this is a design defect of this binding library. It does not imply that it is impossible to write memory-safe code: in very restricted scenarios, when there is a single call that may involve the OCaml GC, no root registration is required. (The `alien_ffi` library provides explicit "arenas" whose purpose is to avoid GC-allocations by storing memory in Rust-owned regions instead, which somewhat reduces the need to allocate through the OCaml GC.) This severly limits the expressivity of binding code; for example, we believe that there is no memory-safe way to allocate and return an OCaml value of type `int * (bool * bool)` using this API.

It *is* possible in certain domains to write bindings that never need to register GC roots, because the datatypes exchanged are so simple that a single allocation with non-OCaml value arguments is enough. (We explain this in more details in Section 4.) But we believe that this cannot be a long-term solution, even when restricting one's attention to a given problem domain.

When we realized this, we decided to review `ocaml_threshold`, the experimental binding written using `alien_ffi`, to check whether it is memory-safe despite the absence of root handling. We explain our review process and results in Section 4.

2.3 Support for zero-copy runtime features

Both `ocaml-rs` and `alien_ffi` support bigarrays, that is byte/int arrays which can store their data outside the OCaml heap. (One needs to review the implementation to ensure that this outside-heap support is used, as a bigarray may also allocate its data on the GC heap.)

The `alien_ffi` library has good support for `custom` blocks, whereas such support is very limited in existing releases of `ocaml-rs` – there is only support for abstract values to which a finaliser is attached.

Defect (`ocaml-rs`, medium): `ocaml-rs` needs to gain `custom`-block support to meet Nomadic Labs requirements.

2.4 Fragility with respect to OCaml runtime changes

The C FFI for the OCaml runtime exports a mix of functions (typically to allocate values) and macros (typically to deconstruct OCaml values from C). There are two reasons for using macros:

Performance. The macros correspond to extremely cheap operations for which you don't want to pay the cost of a function call. (For these macros, static inline functions may work equally for C users, but cross-language inlining between Rust and C is currently not robust enough to expect inlining to happen for Rust users.)

Metaprogramming. The root-declaration macros `CAMLparam*`, `CAMLlocal` and `CAMLreturn` cannot be defined as functions because they extend the lexical scope of the code in which they are used with internal variables, and communicate with each other through those variables.

The `ocaml-rs` library re-implements those macros as Rust macros. This is good for performance, but it creates a strong implicit dependency between the `ocaml-rs` implementation and only certain versions of the OCaml runtime: if you compile an `ocaml-rs` program against a runtime version with a different value representation, memory will be silently corrupted at runtime.

Defect (`ocaml-rs`, important): There is currently no check in the `ocaml-rs` configure-and-build pipeline that the current OCaml version corresponds to the Rust macros' assumptions. This is a problematic design flaw that should be fixed as soon as possible. This is a pernicious mistake because those runtime representations used to be fairly stable in the past, and are changing now with the gradual integration of the Multicore-OCaml runtime; this fragility may have not been an issue for all versions of OCaml tested by `ocaml-rs` maintainers, but we can already say that the current definitions are incompatible with the newly-released OCaml 4.10.

The `alien_ffi` library made the different choice of writing some C support code (`libalien_-ffi_c`) that re-exports these macros as C functions, which are then exported and used as functions on the Rust side. The macro implementation is not duplicated, and thus there is no fragility with respect to the OCaml runtime version. There should be a performance overhead, but note that performance of the binding layer is not critical for cryptographic libraries, where we expect the C-side or Rust-side computation to be long and costly, and thus a few extra function calls should not have a noticeable impact. (We did not benchmark this.) In the context of a library specialized for cryptographic bindings, this is an excellent design choice made by the `alien_ffi` author.

Unfortunately, this solution cannot scale to the "metaprogramming" macros, which cannot be exported as functions; in particular, we do not presently see how to use this approach for root-registration macros. (This is not a problem for `alien_ffi` which does not handle root registration, but as explained in Section 2.2 we believe that all binding libraries should support root registration.)

Open Question: How could OCaml-Rust binding library authors reconcile performance requirements (if they want a general-purpose binding that can provide low overhead, as the C FFI) and robustness with respect to OCaml runtime changes?

Recommendation: A minimal first step is to ensure that any assumption made on the OCaml runtime version is checked explicitly during the configuration or build step of the binding library.

2.5 Falsely safe interfaces

We remarked during our design comparison that neither binding libraries mark their FFI functions as `unsafe` in their interfaces.

Defect (`ocaml-rs` and `alien_ffi`, medium): If using a function may break memory safety (and hence possibly result in, e.g., a segfault), then the declaration of this function should be marked as `unsafe`. Both `ocaml-rs` and `alien_ffi` export the unsafe interface of the C FFI to Rust users, which may very well result in memory corruption or segfaults. Not marking those as `unsafe` subverts and weakens the Rust safety model.

Some of the functions/macros in the interface are safe and need not be marked `unsafe`. In the future we could hope for better static guarantees (see Section 6) which would make more functions safe again.

2.6 Preliminary conclusions

The conclusion of our comparative design analysis is that while both `ocaml-rs` and `alien_ffi` lack features that matter to write memory-safe, zero-copy bindings, it seems easier to improve `ocaml-rs` implementation to meet the requirements (by integrating support for `custom` blocks) than to improve `alien_ffi` to meet the requirements (by adding root-handling primitives).

The `ocaml-rs` library also has the disadvantage of size: not all features are useful to meet Nomadic Labs' requirements, and there seem to be some implementation defect lurking (in the useful parts and in the other parts). We decided to do a code review of the parts of the API that suffice to meet Nomadic Labs requirement.

Recommendation for the future: In the medium term, we believe that a "core" library, that would export just the Rust equivalent of the C-FFI interface, should be provided as a separate crate, to clarify the separation of concerns between robust full-FFI support (the "core" library) and convenient features that provide less control (and may endanger the zero-copy expectations).

3 `ocaml-rs`: partial code review

Our design analysis of binding libraries (Section 2) suggested that the interface exposed by `ocaml-rs` is a reasonable basis on which to build OCaml/Rust bindings. We thus decided to review the implementation (the parts useful to meet the requirements of having a low-level FFI layer) to check that we could recommend using the current implementation.

In general, we found that ocaml-rs does a good job of exposing FFI features to Rust. This is reasonable basis to build upon for OCaml FFI. Still, several subtle bugs and counter-intuitive behaviors made their way to the library. This is probably due to the lack of familiarity with internals of OCaml runtime.

The review focused on the "core" sub-directory of OCaml-rs. This directory provides a relatively faithful translation of the OCaml C FFI. The rest of the library adds higher-level features that are convenient but not necessary:

- anything provided by the higher-level layer can be implemented on top of the `core` FFI,
- some control is lost making, for instance, the *zero-copy* requirement harder to guarantee.

We would thus recommend to avoid the higher-level features and consider the core layer only. The following issues should be addressed before using the library in production.

Note that our review was non-exhaustive, further issues may remain.

Remark: we are planning to communicate this list of issues to the `ocaml-rs` maintainer.

3.1 Zero-copy requirements

In general the low-level layers of `ocaml-rs` faithfully represent the C-FFI interface, and provide the same zero-copy guarantees. We believe that `ocaml-rs` is usable for cryptographic code with zero-copy requirements, as long as one restricts oneself to this "core" API.

Specifically in the case of bigarrays/bigstrings, we reviewed the low-level `of_slice` function and verified that the input slice is not copied into the bigarray. On the other hand, the high-level `from` function from the trait

```
impl<T: Copy + BigarrayKind> From<&[T]> for Array1<T>
```

does copy its input data into the OCaml heap, and should not be used in zero-copy code.

Recommendation: Avoid using the `To`,`From` traits in zero-copy code, in particular when bigarrays are involved. On the other hand, `of_slice` is the expected zero-copy primitive to create bigarrays.

3.2 Implementation defects

3.2.1 Evaluation order in `store_field` macro

The `store_field` macro is defined in Rust as:

```
macro_rules! store_field {
    ($block:expr, $offset:expr, $val:expr) => {
        $crate::core::memory::caml_modify(field!($block, $offset), $val);
    };
}
```

It corresponds to the following C-macro:

```
#define Store_field(block, offset, val) do{ \
    mlsize_t caml__temp_offset = (offset); \
    value caml__temp_val = (val); \
    caml_modify (&Field ((block), caml__temp_offset), caml__temp_val); \
  }while(0)
```

However, in the C code, the evaluation order is made explicit: `val` is evaluated before `block`. The reason is that potential side-effects in the evaluation of `val` could trigger a garbage-collection that can invalidate the result of `block` evaluation.

This implementation is designed to allow code like this:

```
Store_field(result, 0, f());
Store_field(result, 1, g());
...
```

The Rust version should offer the same guarantees.

Defect (`ocaml-rs`, medium): The current `Store_field` implementation will lead to undefined behavior for code of this form.

Remark: the C-macro is not safe for side-effects in the `block` argument. If we wanted a completely safe Rust interface, `store_field` could be a function that requires its value arguments to be registered as roots.

This issue is an instance of problems described in sections Too much macros and Hidden unsafety.

3.2.2 Buffer overflow in local root management

Local functions are expected to register the local OCaml values they manipulate so that the GC is aware of them.

This registration is done with the `caml_local_roots` variable, a linked list of type:

```
struct caml__roots_block {
  struct caml__roots_block *next;
  intnat ntables;
  intnat nitems;
  value *tables [5];
};
```

This constant 5 is why the C root macros are specialized up to size 5:

```
CAMLparam1..5
CAMLlocal1..5
CAMLxparam1..5
```

It offers a good trade-off between engineering simplicity and overhead (e.g. if `CAMLlocal1` is used, a block with 4 empty cells is allocated, this is not a major waste).

In `ocaml-rs`, these macros were made variadic with unbounded recursion ("caml_param" and "caml_local" in src/macros.rs). The problem is that the block allocated is still of fixed size 5, thus having more than 5 parameters or locals will let the GC access and corrupt arbitrary parts of the stack.

Defect (`ocaml-rs`, important): Variadic parameter-registration macros are wrong above 5 parameters.

In the short-term, the Rust macros should fail if passed more than 5 arguments.

We believe that the `tables` array was intended to be a flexible array member `ntables`, although the constant 5 was used to maximize C compatibility. A possible encoding in Rust would be to make it a flexible array member. This would have to be discussed further with runtime authors.

3.2.3 Performance bug in local root registration & wrapping

In the `caml_body` macro, function parameters are first registered as local roots and then wrapped to access them with a "friendlier" type:

```
$($crate::caml_param!($param);
  let $param = $crate::Value::new($param);)*
```

The performance issue is that `caml_param` is called inside the recursive expansion, for each parameter. Thus we create a new, almost empty, local root block for each parameter.

Defect (`ocaml-rs`, medium): Performance bug in the `caml_body` macro.

What are the benefits of the `$crate::Value::new` wrapping? The intention is maybe to give a more convenient semantics to the wrapped type, however this is probably misleading at this level:

- we are mixing low and high-level aspects here,
- this wrapping complexifies the structure of the code, and indeed it hides one major bug (buffer overflow) and introduces a minor one (performance bug), but only for parameters: the wrapping is done correctly for local variables, with the buffer overflow and no performance issues.
- if a user chooses to use the lower-level macros, they might get different and surprising behaviors because the wrapping won't have happened. This is not necessary for OCaml external functions because the existing macros are sufficient, but this is not the case for auxiliary wrappers (internal functions taking or returning a mix of OCaml values and rust native arguments).

In short, the implementation of root management is unsatisfactory (even if the bugs were fixed) because of the complexities it introduces without a clear gain in safety.

Also, it is done in the `macros.rs` file, outside of the `core` directory. As we suggest a more radical separation between a low-level layer (that interfaces with OCaml runtime internals) and a high-level one (that provide convenience and idioms familiar to Rust developers), it would be nice to move the management of local roots to this core library.

3.3 API design and code organization issues

3.3.1 `caml_alloc_final` is suboptimal

The OCaml runtime has a notion of custom blocks for embedding foreign values in the OCaml heap, along with a pointer to a `custom_operations` structure providing runtime operations such as serialization, hashing, comparison and finalization. The intention is that many custom values of a similar "class" will use the same operations.

In the `ocaml-rs` FFI, the general interface to custom blocks is not exposed. See the Section 7 of our report for a proposal.

Note: the binding of `custom_operations` in `alien_ffi` but a few mistakes made their way. It is also very sensitive to OCaml version.

On the other hand, `ocaml-rs` does expose the `caml_alloc_final` helper function of the OCaml runtime. However this function was provided for backward compatibility and using it in the long term is not satisfying. It can have surprising performance behavior: a custom operation block has to be initialized for each finalization function. This custom block is dynamically allocated, stored and later looked up from a linked list. Thus, the complexity of each custom allocation is $O(\#finalization\_fun)$.

3.3.2 `caml_callback` is inherently unsafe, and so are failure functions

Defect (`ocaml-rs`, medium): The `caml_callback*` family of functions does not catch exceptions that happen in OCaml code. If such an exception happens, the Rust-managed stack frames will simply be discarded. This will break soundness of Rust code. (Note: because of asynchronous exceptions any code that calls `caml_callback*` function should be prepared for such an event).

Recommendation: `ocaml-rs` should not expose the `caml_callback*` functions which reraise exceptions to the OCaml runtime. It should only expose the `caml_callback*_exn` variants, that return a result type to be checked by the binding with `Is_exception_result`.

If an exception result is detected, the FFI author may turn it into a specific return value (for example when returning at an `option` or `result` type on the OCaml side), which is perfectly safe.

On the other hand, there is currently no satisfying way to re-raise an exception from the Rust side. Using the exception-raising functions from `src/core/fail.rs` is only safe if there are no resources to release in the current scope. To ensure that this is the case, raising functions should only be used at the exit boundary of the FFI call; we see two approaches to do this:

- One could this operation inside a wrapper on the Rust side, for example by providing a sort of `CAMLextern` macro to indicate entry points of the FFI. The code argument of this macro would return a result value, and the macro would be in charge of raising the exception to the OCaml side after all resources went out of scope. This assumes that the macro body is put in a separate function which is not inlined in the handler; this is a plausible short-term solution but it makes fragile assumptions on the Rust compiler behavior.

- Or we could extend the OCaml FFI with a notion of external that return a result value, and have the OCaml code be in charge of re-raising the exception directly. This is more robust but requires a new OCaml feature.

3.3.3 Too much macros

In the OCaml FFI, C macros are used for two purposes:

1. optimization (without using `static inline` functions)
2. expressing low-level operations that cannot be expressed with plain C functions:
   a) making evaluation order explicit (as in `Store_field`)
   b) messing with the lexical scope (as in `CAMLparam,local,return`)

For 1. we should favor simplicity and use plain functions as Rust handle inter-crates inlining well.

For 2. the use of macros is justified. Although, as we have seen, the idioms exposed to by ocaml-rs macros are not entirely satisfying.

3.3.4 Hidden unsafety

Defect (`ocaml-rs`, medium): Most of the functions from ocaml-rs are not marked `unsafe` yet they are.

This is probably intended for convenience, as it would be cumbersome to wrap every FFI block with `unsafe`. However, Rust has a well defined notion of safe code and lying about that can mislead the programmer and cause the compiler to generate incorrect code.

Recommendation: Core functions should be marked `unsafe` if their incorrect usage by the programmer may lead to memory- or ressource-unsafety. Safety could be provided by a separate dynamic-checking or static-discipline layer (for instance, enforcing safe idioms using encodings such as caml-oxide).

3.3.5 Low-level representation of tags

`core/tag.rs` is a bit surprising: it provides two encodings of tags, a low-level and a high-level ones. However the high-level part seems even less safe and the benefits are unclear (non constant constructor tags went away).

Maybe the enumeration should be marked as `#[non_exhaustive]` (although, given our understanding of `non_exhaustive` semantics, this might still be unsatisfactory; further study is needed).

3.3.6 Rethinking the "types" modules

The `types.rs` and `core/types.rs` modules aim to provide a typed-bridge between OCaml and Rust by constructing isomorphism between OCaml and Rust representations.

`core/types.rs` is probably here by mistake: it seems to have been superseded by `types.rs` and should be removed. Also, some manipulations in it are clearly wrong or misleading (for instance the `!is_block(v)` in the `From::from` implementation for `Array`).

While such a high-level bridge is convenient, the current implementation has several shortcomings:

- it is "half-typed" (for example, a single type `Tuple` for tuples of any length) and does not offer much safety,

- the automatic conversions encourage "deep embeddings", where the whole graph of an OCaml (resp. Rust) value is converted to Rust (resp. OCaml). This make it harder to reason about performance and copy behavior.

For instance, the API for plain OCaml Bigarrays will never do unexpected copy but that can happen silently with the Rust layer because of the pervasive use of ad-hoc polymorphism (the `From` trait). It seems unlikely that someone dealing with Bigarrays and FFI has the intention to do deep copies often: the convenience brought by this encoding is maybe optimizing for the wrong use case.

Recommendation for the future: Our general feeling is that one should split the two levels of abstraction: have a library that only deals with exposing the Rust equivalent of the C FFI as predictably as possible, and move this sort of convenience features to a separate library on top.

3.3.7 Const in `global_roots`, and more generally compilation assumptions

```
pub fn caml_register_global_root(value: *const Value);
pub fn caml_remove_global_root(value: *const Value);
```

Defect (`ocaml-rs`, minor): The pointers passed to the global root functions is marked `const`, yet the GC is free to move these values and rewrite the pointed area as desired.

The Rust compiler could do optimizations based on this flawed constness assumption, for instance by delaying a store or eliminating a reload when it has determined that this is locally safe.

Fortunately, this is unlikely to be the case yet because the Rust compilation model is still close C and the assumptions made by the binding of the FFI still (mostly) applies. However this is certainly not future-proof as Rust developers have expressed the intent to work on type-directed optimizations soon.

The current situation is unsatisfactory and, in the long term, we should really work out the correct unsafe annotations and safe embeddings.

3.3.8 Incompatible with future versions of OCaml: safety checks and versioning scheme

`ocaml-rs` is largely a translation of the C headers of OCaml FFI. However, these headers can change as new versions of OCaml are released.

For instance, references to `caml_local_roots` are no longer correct as of OCaml 4.10 as this has been replaced by a macro using `Caml_state`. Without reproducing the same logic (with some form of versioning), `ocaml-rs` is not usable with the future versions of OCaml.

Fortunately, this specific case will lead to a linker error. Other errors can be more subtle: for instance, the `custom_operations` structure evolved in the past. If both the C and Rust implementations are not kept in sync, this will directly translate to out-of-bounds access (currently only `alien_ffi` is affected, as upstream `ocaml-rs` does not offer general support for custom blocks).

Recommendation: In the short term, any Rust library that makes assumptions on version-dependent aspects of the OCaml runtime should come with an explicit version-compatibility check at configure- or build-time. No such check currently exists in `ocaml-rs`.

In the long-term, we could think of more advanced versioning mechanisms:

- compile-time conditional tests could be used to expose only the features available in the local version of OCaml, and

- the OCaml compiler could also expose a version-specific global symbol (e.g. `caml_version_4_10`) that FFI-using libraries (either C or Rust) would depend on to prevent linking code compiled with incompatible versions.

3.3.9 Zero-terminated strings

`named.rs` builds a zero-terminated string by using the format macro, for compatibility with the C strings used by OCaml.

Recommendation: The recommended way of building C-compatible strings in Rust is rather to use the `Cstr` module.

3.3.10 Minor comment about local roots

The `ocaml-rs` documentation currently includes the following remark:

> However, when using the type wrappers provided by `ocaml-rs` (`Array`, `List`, `Tuple`, `Str`, `Array1`, ...), `caml_local!` is already called internally. This means that the following is valid without having to declare a local value for the result:
>
> ```
> caml!(example1(a, b, c){
>     List::from(&[a, b, c])
> );
> ```

The comment is slightly misleading and incorrect in the general case. This has more to do with evaluation order, the limited use of side-effects by intermediate functions and a strict style were no intermediate ocaml value is used. A different nesting of function calls could lead to a temporary value to be allocated and not registered as a root.

For instance this slightly different version is incorrect:

```
caml!(example2(a, b){
  List::from(&[List::from(&[a]), List::from(&[b])])
);
```

Note: `example1` and `example2` do not compile anymore with recent versions of `ocaml-rs`.

4 `ocaml-threshold`: review for memory safety

During our comparative design analysis of `ocaml-rs` and `alien_ffi`, we realized that the lack of root-management support in `alien_ffi` makes it memory-unsafe for all FFI use-cases that require root registration, which we expect to be the common case among FFI bindings. We thus decided to review `ocaml_threshold`, the experimental binding written using `alien_ffi`, to check its memory safety.

Our conclusion is that the version of `ocaml_threshold` we reviewed [Tate, 2019b] *is* memory safe: each of its function does fit within the narrow fragment that does not require any root management for safety.

In the rest of this section, we explain the reasoning rules we used to check the code for safety, and we mention question or small issues that we found during our review.

4.1 Root-free bindings?

Let us now explain relatively simple criterions for when a piece of FFI code is safe without root registration. Note that we do *not* recommend writing bindings in this style:

- It is easy to get it wrong and to end up with memory errors; systematically following the root-registration rules is easier to write and to trust, and could even be enforced by the Rust type system in a long-term future (Section 6.1).

- The root-free fragment is much less expressive, as it cannot perform several OCaml allocations in sequence. Returning an OCaml value of type `int * (bool * bool)` is not possible. In particular, a library with a binding using a root-free library may have a later API evolution that now requires root-handling for proper OCaml coverage.

Recommendation: We recommend using binding libraries that correctly support root registration, instead of aiming to stay in the root-free fragment, unless a static discipline is used to automatically check the correctness of unique tail allocation (a long-term possibility, see Section 6.1).

Unique tail allocation The reason the OCaml FFI requires registering OCaml values as roots is that the garbage collectors may move values around, and many of the FFI functions may invoke the garbage collector. When this happens, OCaml values in FFI code may become invalid. If they were registered as roots (more precisely: if their addresses were registered as roots), then the GC is aware of them and updates them when the value is moved. The `CAMLparam`, `CAMLlocal` and `CAMLreturn` macros cooperate to register (adresses of) C variables as local roots.

In absence of root registration, any runtime function that may invoke the OCaml GC (in particular allocation functions) may invalidate any OCaml value in scope. (This invalidation may be difficult to notice in test scenarios as most allocations do not need to trigger a minor collection and move values around. When it happens, memory silently becomes invalid and undefined behavior will happen later in the application lifecycle, when a program or the runtime tries to access the invalid value.)

Root-free bindings are correct when they fall in the subset that we call "unique tail allocation": there is a single FFI call that may invoke the OCaml GC, at the end of the function binding, and no OCaml-value temporaries are used after this point – the result of this allocation is returned directly. More precisely:

- Along any execution path of the binding there is a single FFI call that may invoke the OCaml GC, and no OCaml value is used afterwards (except returning this function result to the caller).

- Any non-immediate OCaml value argument of this FFI call is used, *within the call*, before the allocation happens.

Remark: one has to be careful about OCaml value arguments to the unique allocating call. For example, the convenience function `value caml_alloc_boxed(value arg)` uses its `arg` argument *after* having called the GC, so this argument has to be registered as a root – it is not a *tail allocation*. In other words, being a *tail allocation* is not a local criterion (one has to inspect the behavior of the FFI functions involved), but a safe local approximation is that the unique allocation takes no OCaml value parameters except immediate (integer) values, which never require rooting.

Remark: for example, there is no reasonable way to allocate (from the FFI) an OCaml value of type `int * (bool * bool)` that respects the unique tail allocation principle, as one would have to allocate the two pairs in sequence.

Our experience with C binding authors is that they often under-estimate which parts of the FFI interface may call the GC. We believe that trying to generalize this "unique tail allocation" principle as a code discipline would run into bugs caused by similar erroneous assumptions.

The `ocaml_threshold` binding `ocaml_threshold` sits in an interesting corner of the FFI design space where the OCaml values returned by FFI functions are always plain data, rather than structured types. This plain data is allocated at the end of each binding, with an FFI call that only takes immediate values, and returned immediately.

Consider for example the following function from `ocaml_threshold`:

```
pub extern fn bivar_poly_row(bivar_poly: Value, i: Value) -> Value {
    let bivar_poly = caml_opaque::acquire::<BivarPoly>(bivar_poly);
    let i = caml_to_int(i);
    caml_opaque::alloc::<Poly>(ops(&POLY_OPS), bivar_poly.row(i)) }
```

In this function, `caml_opaque::acquire` and `ops` are data-access functions that may not invoke the GC, `caml_to_int` is an immediate conversion function that may not invoke the GC: `caml_opaque::alloc` is the only call that may invoke the GC, and its result is returned immediately – OCaml `value` arguments are not used between the allocation and the return. Finally, its arguments are not themselves OCaml value temporaries.

We reviewed the implementation of `ocaml_threshold` [Tate, 2019b] and believe that all its exposed functions respect this "unique tail allocation" principle: despite the absence of root handling in `alien_ffi`, the library is memory-safe.

We however have the following concern: so-called `opaque` data is freed with a finalizer, but the function `Gc.full_major` is called explicitly to ensure that deallocation happens at a certain point. While we do not discourage the use of finalizers, we discourage the use of `Gc.full_major` in favour of deterministic resource management (see our recommendations in Section 6.2).

4.2 Other review remarks

Arenas `alien_ffi` uses Rust arenas to store the backing data of OCaml bigarrays. In `ocaml_threshold`, arena lifetimes are made explicit on the OCaml side thanks to a `with_arena` function that takes an arena-expecting continuation, and is respecting for allocating and releasing the arena.

This design approach has the following advantages:

- Besides issues with the zero-copy requirement, allocating this backing data on the OCaml heap would violate the "unique tail allocation" principle and require root registration.
- The `with_arena` calls clearly delimite the lifetime of this sensitive data, with a deterministic release strategy.

Defect (`ocaml_threshold`, medium): The `with_arena` function does not properly handle exceptions raised by the continuation. The implementation should be fixed before this pattern is used with bindings that may raise non-fatal exceptions.

As a general rule in OCaml, one should protect against unexpected exceptions, even if the library calls inside `with_arena` blocks are not supposed to raise exceptions. Unexpected exceptions can include serious failures such as `Out_of_memory` and interruptions such as `Sys.Break` (Control-C), or bugs, including exceptions arising inside finalizers that happen to run at that time...

Recommendation: The implementation of arena-using code should have a protection against trying to access or release backing data that would already have been released. This can be achieved by attaching a shared boolean flag, managed by the OCaml GC, to the arena value and to its derived pointers.

One potential mistake that could lead to accessing the backing data after release would be to leak an OCaml bigarray outside the `with_arena` continuation (this is a programming mistake that should not result in undefined behaviors).

Static lifetimes Several functions of the `alien_ffi` API claim to return Rust values at the `&'static` lifetime, when in fact the lifetime of these values is more restricted (for example they are in a bounded-life arena). This pragmatic choice comes from the difficulty of statically tracking the lifetime of Rust values transferred through OCaml wrappers; but it is unsafe and should be marked as such in the API – see Section 2.5 for the general lack of `unsafe` markers in binding library interfaces.

Recommendation: Use explicit `unsafe` markers in the API of functions that can break Rust safety guarantees – in this case by unsafely extending a Rust lifetime.

`custom_operations` structure order

Defect (`alien_ffi`, medium): There was a small programming mistake in the placement of the `compare_ext` field in the Rust declaration of the `CustomOperations` structure, which is supposed to use the same representation as the C `custom_operations` struture.

As a result, most of the fields in the structure are wrongly placed: only `identifier`, `finalize` and `compare` can be accessed correctly. As far as we are aware, none of the code using `alien_ffi` currently uses the other fields, so the bug did not affect them.

Note that the `custom_operations` structure has evolved over OCaml releases. The current implementation in `alien_ffi` is only safe for certain OCaml versions.

Recommendation: `alien_ffi` should explicitly check the compatibility of the OCaml version at configure- or compile-time.

5 Short-term recommendations for binding authors

Should one write Rust/OCaml bindings by using a layer of C glue code, or using a Rust/OCaml binding library directly? Currently both approaches have merits; C glue code introduces an extra layer of complexity, but it avoids relying on Rust/OCaml binding libraries that are young and not battle-tested yet. Our review suggests that they are essentially usable today, with a bit of care to remain in the fragment that closely corresponds to the C FFI primitives. In the long run, we believe that Rust/OCaml binding libraries will provide productivity benefits, in particular if they can expose interfaces that are safer (with more static checking) than the C FFI approach; the best way to reach this ideal situation is to engage with OCaml/Rust binding libraries and their ecosystem.

To write pure OCaml/Rust bindings without intermediary C glue, our short-term recommendation is to use the "core" API exposed by `ocaml-rs`, avoiding its convenience features such as the traits `To` and **`From`**. We propose `ocaml-rs` support for custom blocks in Section 7, which completes the feature set required for zero-copy cryptographic binding libraries.

We believe that `alien_ffi` is safely used in `ocaml_threshold`, but its lack of root management makes it less suitable to write OCaml bindings. To any user of `alien_ffi`, we strongly recommend to read and follow the "unique tail allocation" criterion that we used in our review of `ocaml_threshold` (Section 4.1).

In the medium-term, care should be taken to maintain `ocaml-rs` with good support for the low-level api corresponding to the OCaml FFI. We recommend splitting this core fragment into a base crate, that the rest of `ocaml-rs` could be built upon. We strongly recommend marking the API as unsafe, and taking particular care to ensure that the implementation is robust against changes in the OCaml runtime; in particular, explicit version checks should occur at configuration- or build-time for any version-specific assumption in the codebase.

6 Long-term recommendations for safer binding libraries

For medium to long-term sustainability, we recommend to avoid code styles and API approaches that do not fit Rust and OCaml programmers' expectations about safety, which has to be be guaranteed statically or dynamically (i.e., by raising an exception/panic as last resort instead of triggering undefined behaviour or aborting).

The OCaml C FFI expects heavy discipline from the programmer, but this is in a language (C) where the programmer is already expected to have a lot of self-discipline, and it is risky to follow the exact same approach in languages where programmers learn and expect to rely more on their tools.

Of course, it is difficult to design safe yet efficient libraries for FFI binding, and we understand that existing binding libraries focused on the simpler approach of exposing the API directly. However, even in the context of a simpler approach, the current ecosystem is not satisfying: Rust as a language provides explicit support to signal unsafety and let library users adjust their assumptions accordingly, yet those safeguards were not used by binding libraries authors.

This section aims to make recommendations to achieve better safety in the long term. One of our conclusion is that there is no fundamental reason that zero-copy requirements should be at odds with safety.

6.1 Safe handling of OCaml values in Rust

Manipulating bigarrays or custom blocks on the Rust side is a particular case of direct manipulation of pointers into the OCaml heap. The approach offered by OCaml's C FFI and emulated by `alien_ffi` and `ocaml-rs` (except in some of its higher-level functions outside of our recommendation); it requires foreign code (in C or Rust) to respect the OCaml Garbage Collector (GC) discipline, and in particular correctly track GC roots.

The current approach of `ocaml-rs` is to expose the same root-management interface as the C FFI. This requires strong discipline from programmers; they are writing usafe code (the functions should be marked as such) without the memory-safety that they can expect from a high-level language.

Recommendation for the future: In the long term, we strongly recommend to encode this GC displine into static checks, by following the experimental technique using phantom lifetimes that was developed recently for Rust+Javascript [Jeffrey, 2018] and Rust+OCaml [Dolan, 2018]. A Rust/OCaml binding layer with strong safety guarantees would bring the safety guarantees and static-memory-ownership reasoning expected in idiomatic Rust code, and it would be markedly superior to the current approach of the C/OCaml FFI.

The basic idea of GC-representation using phantom lifetimes is idiomatic to Rust programmers. We recall this idea. A `gc` capability is passed around, representing the capability to access the (Javascript or) OCaml runtime. The following invariants are enforced:

- Any allocation (or any other operation that might trigger the GC) requires mutable access to it (`&'a mut`).
- Any unrooted value (such as a temporary) requires shared access to it (`&'a`).

The goal is to forbid the GC to run as long as any unrooted pointer to the GC heap is alive; equivalently, to invalidate unrooted pointers after a GC has run. This is done by the Rust compiler which checks at compilation that no mutable access can coexist with any shared access. Thus, the checks performed by the Rust compiler allows to represent faithfully the constraints for handling OCaml values.

We did not have the time budget to develop this approach further, and in particular we have not studied all limits to applicability – applying it in practice may turn out to be provide less guarantees than we hope. However, we suspect that this approach could scale to real-world OCaml/Rust use-cases, could be engineered into several libraries (including `ocaml-rs` and `alien_ffi`, with fairly small API changes) and even provide safe interoperability between them:

- This technique can probably be adapted for making simple approaches such as `alien_ffi` safe. The `alien_ffi` library avoids rooting, and the lifetime approach is agnostic regarding any choice about rooting: in particular, it can statically verify that the “unique tail allocation” approach (Section 4.1) is followed correctly. This can prevent memory-safety bugs in the future, simplify the code (in some cases there would be no need for the user to introduce `unsafe {}` blocks), and would also serve to highlight the expressiveness limitations of such approaches.

- Mechanisms for rooting can be built on top; they can re-use `ocaml-rs` code but they can also be liberated from following so closely the OCaml C FFI approach. Other approaches exist [Dolan, 2018; Bour, 2018], we consider that this is a research-level topic.

- Once a convention to represent access to the OCaml runtime has been established in this way, and that several FFIs understand the same convention, they become interoperable, so that it becomes possible to mix different FFIs in the same program. For instance it is possible to commit to one FFI, and later transition to another gradually, provided they understand the same convention.

6.2 Resource management

In addition, we recommend to follow best-practices regarding resource-management in OCaml.

- If a custom block handles a resource whose release should be handled by the garbage collector (such as memory), use a finalizer to release the resource. This might exclude values whose erasure must be predictable and guaranteed. The function `caml_alloc_custom_mem` from OCaml 4.08 is to be preferred for memory-only resources, since it properly adjusts the GC reclamation speed by taking into account the size of the contained value.

- In all other cases, the resource has to be released in a reliable and predictable way. In OCaml, this is typically done with higher-order functions based on `Fun.protect`, which guarantees that the resource lives for the duration of a scope and that the release function is called in case of an exception. Even if finalizers might seem more flexible, we discourage the use of `Gc.full_major()` to ensure that a value has been finalized, because it is not scalable and does not guarantee the absence of leaks.

- In addition, if release is done explicitly rather than with a finalizer, a check has to be performed whenever accessing the resource to ensure that it has not already been released. (OCaml, unlike Rust, has no convenient way to ensure this at compilation time.) Explicit release functions can optionally be given to release the resource in advance.

- Lastly, in case asynchronous exceptions are used in the program (e.g., the exception `Sys.Break` following Ctrl-C), one should be wary that resource-safety is only possible in limited cases.

We give as an example the case of `ocaml-librustzcash` [Nomadic Labs, 2020], but these advice apply as much when interfacing with Rust. It should be noted that, after our review, `ocaml-librustzcash` has been migrated to use the safer `ctypes` library. This later version is outside of the scope of this report.

Defect (`ocaml-librustzcash`, medium): The types `proving/verification_ctx` are exposed as `Bytes.t`.

Recommendation: We recommend making the context types abstract.

Defect (`ocaml-librustzcash`, medium): Except in the explicit finalization functions, the uses of values of type `proving/verification_ctx` in the bindings perform an access to the custom data by using `Ctx_val` without checking that its content is non-`NULL`.

Recommendation: If a context that has already been released is passed to a binding call, we recommend throwing an OCaml exception.

On the OCaml side, we would hope for more abstraction to guarantee correct interleaving of initialization/release functions.

Recommendation: In addition or replacement to the functions `{proving,verification}_ctx_{init,free}`, we recommend to structure the resource-handling code using higher-order wrappers `with_{proving,verification}_ctx` that enforce that the release function is called reliably and predictably, including in failure scenarios.

In direct-style OCaml code, such a wrapper would have the type `(ctx -> 'a) -> 'a` and look as follows:

```
let with_proving_ctx f =
  let ctx = proving_ctx_init () in
  Fun.protect ~finally:(fun () -> proving_ctx_free ctx) (fun () -> f ctx)
```

In the Tezos codebase, the context can be threaded through the `Lwt.t` monad for cooperative concurrency (for instance in `verification_ctx`). In this case, the correctness can be less straightforward. Then, a ressource wrapper can also simplify the code, although a more expressive one is needed for this form of use, with the richer type:

```
(ctx -> 'a tzresult Lwt.t) -> 'a tzresult Lwt.t
```

7 Binding library support for custom blocks

We have added support for the OCaml custom blocks API in the `ocaml-rs` library, in the `ocaml::core::custom` module:

```
pub struct CustomFixedLength { ... }

pub struct CustomOperations { ... }

pub unsafe fn data_custom_val(v : Value) -> *mut c_void { ... }
pub unsafe fn custom_ops_val(v : Value)
               -> *const CustomOperations { ... }

extern "C" {
  pub fn caml_alloc_custom(ops : *const CustomOperations,
         size : Size, mem : Size, max : Size) -> Value;
  pub fn caml_alloc_custom_mem(ops : *const CustomOperations,
         size : Size, mem : Size) -> Value;
  pub fn caml_register_custom_operations(ops : *const CustomOperations);
  pub static mut caml_compare_unordered: c_int;
}
```

This is a direct binding of the OCaml runtime API provided in the file `custom.h`. The documentation of these function is provided as part of the OCaml manual.

These functions can be used to provide a zero-copy guarantee for memory blocks containing sensitive data. In order to do so, one should write to the custom block a pointer to a memory block allocated outside of the OCaml heap. For example, one could write:

```
*(data_custom_val(custom_block_value) as *mut *const u32) =
    pointer_to_sensitive_u32_value;
```

Depending on the strategy used for erasing and releasing the memory block allocated outside of the OCaml heap (see Section 6.2), one could implement a finalizer for these custom blocks, or one would use an explicit manual release function to erase sensitive data and then free the block.

These bindings can be used together with the macros defined in `ocaml::macros` in order to properly declare roots used by client code.

References

Frédéric Bour. Camlroot: revisiting the ocaml ffi, 2018. URL https://arxiv.org/pdf/1812.04905. code at https://arxiv.org/pdf/1812.04905. 18

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. 3, 17, 18

Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. *TOPLAS*, June 2005. 5

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. 3, 17

Nomadic Labs. tezos/sapling-integration, 2020. URL https://gitlab.com/nomadic-labs/tezos/-/tree/sapling-integration. HEAD commit 93df694ffe4d42d94a9317981091633fdd20879d. 18

Zack Shipko. ocaml-rs, 2019. URL https://github.com/zshipko/ocaml-rs/. HEAD commit e207ac27f421ab1671c28ec0fb8cd78ec43d8b60. 2

Edward Tate. alien_ffi, 2019a. URL https://gitlab.com/et4te/alien_ffi. HEAD commit 7ba998e9960d741daa6e6d6da67811f2bdc5d6d6. 2

Edward Tate. ocaml_threshold, 2019b. URL https://gitlab.com/et4te/ocaml_threshold. HEAD commit 5c7e979ceb011d206b696fa6b907807e003b297d. 4, 13, 15