Towards better systems programming in OCaml with out-of-heap allocation

Guillaume Munch-Maccagnoni*
Inria, LS2N CNRS

30th May 2020

Abstract. The current multicore OCaml implementation bans so-called "naked pointers", pointers to outside the OCaml heap unless they follow drastic restrictions. A backwards-incompatible change has been proposed to make way for the new multicore GC in OCaml. I argue that out-of-heap pointers are not an anomaly, but are part of a better systems programming future.

1. Introduction

Reserving address space to efficiently recognise in-heap pointers (and do even wilder tricks) is standard in garbage collected languages and allocators, as well as in research about memory management. In addition, allowing foreign pointers for interoperability is a standard feature in common systems programming languages.

As part of my research to make OCaml a better systems programming language, I propose to revisit the importance of naked pointers for 1) interoperability, 2) backwards-compatiblity, and 3) performance, three staples of systems programming. I conclude with a challenge addressed to the ML community.

In the appendix, I show a simple design based on virtual address space reservation that is proposed to make naked pointers compatible with the multicore OCaml GC.

2. Gathered evidence from the practice and the literature

2.1. Interoperability

In his essay *Some were meant for C*, Stephen Kell (2017) places the essential value of systems programming languages in *"communicativity"* rather than performance. Among others, he argues against *"managed"* languages in which *"user code may deal only with memory that the language implementation can account for"*.

Concerning foreign pointers specifically, their allowance is indeed standard in common (self-described) systems programming languages beyond C and C++: Rust, Go, and Swift. For instance:

- Rust's ownership-and-borrowing allows it to interoperate safely with garbage collected runtimes, and directly manipulate values from the foreign GCed heap, for instance with SpiderMonkey's GC for JavaScript (Jeffrey, 2018).
- In Rust, value interoperability with C++ is instrumental in the migration from one language to the other, since programs that are migrating can remain a long time in a hybrid state.
- *"Swift is a high-performance system programming language. It has a clean and modern syntax, offers seamless access to existing C and Objective-C code and frameworks, and is memory safe by default."* [1]

There is ongoing research to augment OCaml with new kinds of types and values which would allow operating more safely with foreign pointers (among others): Radanne, Saffrich, and Thiemann (2019), my own (2018).

Proposed replacements in OCaml's *no-naked-pointers* mode include introducing an indirection via a special block, or disguising the pointer into an integer (using the lsb as a tag), i.e. as unstructured data. One will be unable, for instance, to create an OCaml-like value on the Rust stack or heap and pass it as an argument to an OCaml function.

Thus, among the systems programming languages I looked at, none requires such wrapping to access foreign objects. In OCaml as well, the discussion at ocaml/ocaml#9534, and the case of LLVM bindings that has been reported, show preliminary empirical evidence of naked foreign pointers in OCaml in the wild.

In addition, a third technique is sometimes mentioned regarding the *no-naked-pointers* mode, which is to prefix the out-of-heap values with a "black" header (meaning it is already marked). This technique is unorthodox because it consists in checking whether one owns a pointer by dereferencing it. One consequence is that such values can never be deallocated, because it is never possible to reason about whether they are still reachable from the GC. For this reason, while this can be appropriate for runtime-owned values that live for the duration of the program, its interest for users is more marginal. Nevertheless, despite its unsuitability for dynamic allocation, it has been advocated by the designer of *no-naked-pointers* notably because *"for some applications, the overhead of having an associated heap block will be too high"*[2].

Lastly, at ocaml/ocaml#9534 and ocaml/ocaml#9535, without questioning the no-naked-pointers approach, library developers reported the convenience of allowing a special `NULL` pointer value for bindings (and other purposes).

*Guillaume.Munch-Maccagnoni@inria.fr, Nantes, France

[1] https://github.com/apple/swift
[2] https://github.com/ocaml/ocaml/pull/9610#discussion_r431007766

2.2. Address space reservation

The technique we propose relies on reservation of large areas in the virtual address space.

Reserving virtual address space is standard in several common garbage-collected languages: Java, GHC, and Go. GHC reserves 1 TB by default[3]. The new multi-threaded runtime for Julia reserves 100 GB for thread stacks[4]. Go used to reserve 512 GB up-front at some point, and now reserves more progressively. The concurrent OCaml multicore GC reserves 4 GB for the minor heap.

Go further uses virtual addressing tricks to support interior pointers: pointers to the inside of a block[5]. The bitpattern is used to encode the size of the block and thus where it starts.

Making use of large areas of reserved address space is a common technique used in research about memory allocators. For instance, Lvin, Novark, Berger, and Zorn (2008) reads: *"On modern architectures, especially 64-bit systems, virtual address space is a plentiful resource."* In Powers, Tench, Berger, and McGregor (2019), a large virtual address space consumption is traded for the ability to merge physical pages, in order to perform compaction without relocation.

2.3. Backwards compatibility

For simplicity or conclusion in the lack of an efficient solution, the multicore OCaml GC does not support OCaml's naked pointers. Replacing the page table by a technique based on virtual address space reservation could offer the possibility to avoid code breakage while preserving efficiency, as suggested in an experiment from 2013 by Jacques-Henri Jourdan[6].

In the recent years, the perils of breaking backwards compatibility in a programming language has been demonstrated by Python 3, which introduced breaking changes that were expected to be minor, including a change to the semantics of strings whose consequences were miscalculated. While little documentation about the role of backwards-compatibility in programming language evolution in general is found in the literature, Malloy and Power (2019) investigate the Python 2 vs. Python 3 split after ten years and conclude that instead of favouring modernisation, the breaking changes in Python 3 slowed the language evolution: *"[...] Python software developers are not exploiting the new features and advantages of Python 3, but rather are choosing to retain backward compatibility with Python 2. Moreover, Python developers are confining themselves to a language subset, governed by the diminishing intersection of Python 2 [...] and Python 3."* We do not know how to predict the impact of breaking changes in programming languages, even if they are believed to be minor. Thus, a non-breaking solution could in fact quicken the multicore transition.

It has been argued that code using foreign pointers could be adapted in various ways:

- With an indirection: use blocks with special tag *Custom* or *Abstract* or boxed native integers (`nativeint`).
- Tagging pointers: disguise pointers as integers by setting the lsb.

In addition to explicitly breaking existing code, as we have seen in the case of foreign pointers, none of the propositions is ideal: in theory both have a linear overhead when traversing a structure compared to the same traversal using foreign pointers for instance, in which case the code adaptation that is required could be non-systematic. Yet, no empirical evaluation of the impact of the proposed changes has been proposed so far—a (non-precise) tool to detect naked pointers has recently appeared[7] and will be a good tool to perform such an evaluation.

In addition, the first solution incurs an allocation, whereas the second solution only requires boilerplate, but confuses tools like debuggers, static analysers and such, in ways that are unlikely to be fixable over time. Moreover, they do not bring additional safety in terms of resource-management: what they do is to deal in a rather radical manner with the problem of growing the heap while remembering who was outside, which is mostly required by the reliance on the system malloc by the GC to request chunks of memory. This issue too can also be solved by virtual addressing techniques.[8]

The latter issue indeed happens in practice; for instance one industrial user explains that they solve it by forcing a GC to remove out-of-heap pointers after a large out-of-heap structure has been deallocated, when they use the LLVM bindings. However they write: *"Short of rewriting the llvm bindings, I am not aware of a better / more future-proof implementation strategy. Even ignoring any performance concerns about allocating a block for every one of the very many pointers passed from libllvm to OCaml."* In that case, the technique we propose would avoid a rewrite, but also fix the bug.

Lastly, another well-known use of out-of-heap pointers, which is known to break with the no-naked-pointers mode, is symbolised by the Ancient library and the Netmulticore[9] library, which propose to allocate OCaml values outside of the garbage-collected heap to offer heaps that can be shared or large (such as larger than available RAM). Neither an indirection nor tagged pointers can be used to replace this use-case, a fact that has been known from the start[10]. This is the subject of next section.

3. Interest of out-of-heap allocation in current and future OCaml

I have mentioned that beyond foreign pointers, whose support already benefits backwards-compatibility and interoperability, there are libraries that allocate OCaml values outside of the heap. Here I replace this usage in the context of evolving OCaml to make it a better systems programming language.

3.1. The thesis of complementary allocation methods

The Ancient and Netmulticore libraries perform out-of-heap allocation for performance (large data, parallel computing). This usage of a different memory management technique is better understood through the lens of Bacon, Cheng and Rajan's classic paper *A unified theory of garbage collection* (2004), which

[3] https://gitlab.haskell.org/ghc/ghc/issues/9706
[4] https://julialang.org/blog/2019/07/multithreading/
[5] https://blog.golang.org/ismmkeynote
[6] https://github.com/ocaml/ocaml/issues/6101
[7] ocaml/ocaml#9534
[8] More details on this issue and its solution are given in the appendix.
[9] http://blog.camlcity.org/blog/multicore1.html
[10] https://sympa.inria.fr/sympa/arc/caml-list/2015-05/msg00088.html

places memory reclamation techniques on a spectrum. Tracing, operating on live values, is on one end and reference-counting, operating on dead values, is on the other end. Furthermore, the location on the spectrum explains the advantages and drawbacks of each technique.

The use of out-of-heap allocation, not traversed by the GC and explicitly freed when the programmer requests it, is on the opposite end of the spectrum, and thus complements the OCaml GC with a different set of advantages and drawbacks.

The emergence of safe and expressive abstractions to manage the latter with *unique pointers* in C++11 and Rust (which avoid the reference-count overhead, and prevent cycles that leak) has inspired a proposal to develop out-of-heap allocation, in particular to improve the capabilities of OCaml as a systems programming language (Munch-Maccagnoni, 2018).

Rust-like ownership and borrowing would allow two techniques that are complementary to the OCaml GC, as both benefit from a low latency and the absence of repeated traversal by the GC for long-lived values:

- Fixed malloc-like allocation, which is highly tuned in modern allocators for small sizes. Compared to the generational GC, the user pays a bit more up-front to allocate, but also benefits from automatic cell re-use which statically eliminates some allocations. Furthermore, deallocation is prompt, which means that, whether it is done blocking or concurrently, the amount of memory recovered is proportional to the work done. In particular there is no *space overhead* like with the tracing GC.
- Arena (bump pointer) allocation, for values that tend to live longer than a minor collection but are deallocated all at once. Allocation and deallocation are very cheap, but this comes with expressiveness restrictions.

Both would further benefit from a proposal to improve OCaml's control on memory layout[11], if implemented.

3.2. Low latency

A minor allocation risks incurring a stop roughly proportional to the amount of promoted objects if it triggers a minor collection or a major slice, frequently between 2 ms and 10 ms. The low latency of the other allocation methods would drastically augment the expressiveness of the special low-latency dialect of OCaml currently in use, where one tries to promote very little.

In 1998, Mark Hayden's PhD thesis about the Ensemble system (1998) found that bump-pointer allocation using reusable off-heap arenas managed with reference-counting was a solution to get (otherwise impossible) reliable low latencies for buffer management in networking in OCaml. One goal of new resource-management capabilities would be to make such usage first-class in OCaml for structured data.

Thus, augmenting the low-latency capabilities of OCaml increases suitability as a systems programming language.

3.3. Large heaps

The high GC CPU share on some allocation patterns with large data is mentioned on the caml-list as a problem that has Ancient-like allocation as a solution. This is a known problem with automatic memory allocation studied in the context of so-called "big data" where allocations can have an epochal behaviour (long-lived data allocated and deallocated at similar times) where the time share spent doing garbage collection can exceed 50% (see for instance the motivations in Nguyen, Fang, Xu, Demsky, Lu, Alamian, and Mutlu, 2016).

Furthermore, for data larger than RAM, fixed allocation is the only solution to allocate in the swap, since the tracing GC is unsuitable for operating on a disk: this is the original motivation of Ancient, written in a time when RAM was more limited.

A recent experiment by Pierre-Marie Pédrot with Coq to *"Hack a prototype on-disk offloading similar to ocaml-ancient"*, implemented by mashalling to a file, saw a performance improvement of up to 12.3% in benchmarks ($\mu = 3.4\%$, $\sigma = 2.9$)[12].

3.4. Shared heaps

As far as shared memory between threads is concerned, the intent is rather to benefit from the multicore GC, especially for the correct reclamation of lock-free data structures, an area where Rust shows limitations. Nevertheless, sharing memory between parallel threads is a current application for off-heap allocation, for instance with Netmulticore, that breaks under the no-naked-pointers mode.

3.5. Non-moving heap

Fixed allocation is opposed to moving collectors. Not moving is better suited for interoperability again: for instance, the lablgtk bindings for GTK currently have to give the illusion that OCaml's GC is non-moving, by calling `minor_collection` by hand and by disabling compaction.

Not moving is also relevant for security, for instance in the case of crypto APIs which `mlock()` a specific area in memory that contains secrets. (In the particular case of secret keys and other unstructured data, though, OCaml is already well-tooled thanks to bigarrays; but the case of structured data has seen the development of other tools such as cstruct.)

4. Interoperability between GC and fixed allocation

In programming languages, a "no silver bullet" approach which seeks to offer different tools (e.g. different memory allocation techniques) for different jobs can only work if the tools work well in combination (as opposed to leading to dialects that interact poorly: libraries that cannot interoperate, etc.).

When a data structure is allocated in the ancient or netmulticore heaps, it can be manipulated like any other OCaml data structure. Using simplified examples: a statically-allocated list can be traversed using `List.filter` and the resulting list ends up correctly GC-allocated; a large ancient-allocated tree can be traversed with a zipper which uses sharing to allocate only a small portion with the GC during traversal.

This degree of interoperability, sharing and allocation polymorphism is not reachable using the suggested alternatives: indirections or tagged pointers; nor is it with other known alternatives: neither marshalling, nor memory layouts that would determine

[11] https://github.com/ocaml/RFCs/pull/10

[12] https://ci.inria.fr/coq/view/benchmarking/job/benchmark-part-of-the-branch/827/

the allocation method statically à la *"kinds as calling conventions"* (Eisenberg and Peyton Jones, 2017). Thus, regardless of backwards-compatibility, a dynamic *in-heap?* test is desirable and makes the strength of off-heap allocation of structured data.

On the other hand, allocating out of the heap raises many other difficulties:

- How can we ensure the reliable release of out-of-heap memory? (i.e. an ownership discipline)
- How can we ensure the correct use of off-heap data? (i.e. a borrowing discipline)
- How can we deal with "in-heap" pointers from out-side of the heap?

I propose this challenge as an open question to the ML community, since the tools seem mature enough to address it, notably with the ownerhsip and borrowing discipline in Rust (Hoare, 2012; Matsakis and Klock II, 2014) and the ML-compatible linear type system design of Tov and Pucella (2011).

In Munch-Maccagnoni (2018), I propose some answers to these questions[13] inspired by a newly-discovered link between C++11/Rust-style ownership and a categorical semantics of ordered logic (Combette and Munch-Maccagnoni, 2018). Other works from researchers in the OCaml community relate to these goals. Similar goals are given in Gabriel Scherer's research proposal for a *"Low-level Ocaml"*[14]; Radanne et al. (2019) explores some of the complementary machinery necessary to make dynamically-allocated memory safe.

Thanks

Thanks to Josh Berdine, Frédéric Bour, Rian Douglas, Jacques-Henri Jourdan, Adrien Guatto, and Gabriel Scherer for discussions on this topic.

References

David F. Bacon, Perry Cheng, and V. T. Rajan. 2004. A unified theory of garbage collection. In *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*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 50–68. https://doi.org/10.1145/1028976.1028982 2

Guillaume Combette and Guillaume Munch-Maccagnoni. 2018. A resource modality for RAII. In *LOLA 2018: Workshop on Syntax and Semantics of Low-Level Languages* (2018-04-16). https://hal.inria.fr/hal-01806634 4

Richard A. Eisenberg and Simon Peyton Jones. 2017. Levity polymorphism. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 525–539. https://doi.org/10.1145/3062341.3062357 4

Mark Hayden. 1998. *The Ensemble System*. Ph.D. Dissertation. 3

Graydon Hoare. 2012. Rust 0.3 released. (2012). https://mail.mozilla.org/pipermail/rust-dev/2012-July/002087.html 4

Alan Jeffrey. 2018. Josephine: Using JavaScript to safely manage the lifetimes of Rust data. (2018). arXiv:cs.PL/1807.00067 1

Stephen Kell. 2017. Some were meant for C: the endurance of an unmanageable language. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2017, Vancouver, BC, Canada, October 23 - 27, 2017*, Emina Torlak, Tijs van der Storm, and Robert Biddle (Eds.). ACM, 229–245. https://doi.org/10.1145/3133850.3133867 1

Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2008. Archipelago: Trading Address Space for Reliability and Security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. Association for Computing Machinery, New York, NY, USA, 115–124. https://doi.org/10.1145/1346281.1346296 2

B.A. Malloy and J.F. Power. 2019. An empirical analysis of the transition from Python 2 to Python 3. *Empirical Software Engineering* (2019). 2

Nicholas D. Matsakis and Felix S. Klock II. 2014. The rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104. 4

Guillaume Munch-Maccagnoni. 2018. Resource Polymorphism. (2018). https://arxiv.org/abs/1803.02796 1, 3, 4

Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 349–365. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/nguyen 3

Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. 2019. Mesh: Compacting Memory Management for C/C++ Applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 333–346. https://doi.org/10.1145/3314221.3314582 2

Gabriel Radanne, Hannes Saffrich, and Peter Thiemann. 2019. Kindly Bent to Free Us. (2019). arXiv:cs.PL/1908.09681 1, 4

Jesse A. Tov and Riccardo Pucella. 2011. Practical affine types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 447–458. https://doi.org/10.1145/1926385.1926436 4

[13] The paper mentions using the lsb as a pointer tag, but this would be impractical as it would require to emit a masking instruction to realign the pointer before every access.

[14] http://gallium.inria.fr/~scherer/topics/low-level-ocaml.pdf

A. Efficient out-of-heap pointer detection

To efficiently recognise in-heap from off-heap pointers, the broad idea is to offer a small *virtual space map* of very large reserved (but not committed) virtual spaces, using the virtual addressing facilities of the OS. The virtual space is reserved as needed, similarly to what is done in the Go language[15].

Among the design requirements, performance has to be comparable to not having the no-naked-pointers mode and it must not have extravagant adverse effect or be too hard to implement. In addition, I have identified 4 challenges which must be convincingly shown solvable to ensure that the result is both usable and scalable in OCaml and OCaml multicore:

1. the "growing heap" issue, whereby out-of-heap pointers are mistaken for in-heap pointers after the heap grows,
2. 32-bit support & large memory support,
3. the cost of synchronisation for multicore,
4. malloc implementation for multicore.

The design is barely more complicated than the current page table, and I expect it to be more efficient than the efficient 32-bit page table in OCaml (thanks to greater cache locality), even in 64-bit and multicore, with further context-specific performance gains compared to the no-naked-pointers mode obtained by not having to wrap pointers when interfacing with C.

A.1. Language reference

1. Word-aligned out-of-heap pointers remain valid OCaml values.
2. Out-of-heap pointers that enter the OCaml heap must be guaranteed to not belong to *any future* reserved mapping of the OCaml heap.

A.2. Explanation

The clause 1. preserves current OCaml behaviour. It is also important to note that non-aligned out-of-heap pointers and out-of-heap pointers to char are forbidden, or to explain to which extent they would be allowed inside OCaml values (and operated with external primitives, for instance, for supporting efficient cstruct-like libraries), due to the proliferation of reliance on the `0bxxxxxx10` bit pattern in the runtime to propagate exceptions.

The word of caution from the OCaml manual about out-of-heap pointers that become accidentally in-heap after the heap grows is made normative as the clause 2. In other words, it is left to the user to make sure that this does not happen, by whatever means they have—but let us make sure that realistic context-specific solutions exist. The reason for not mandating a specific solution for 2. is that this is outside of the scope of OCaml: indeed, it concerns the specific interoperation between OCaml and another component. This issue was believed to be unsolvable among core ocaml developers[16]. The mistake, I believe, is to look for a "silver bullet" that solves the problem independently of the context.

[15] https://github.com/golang/go/blob/9acdc705e/src/runtime/malloc.go#L77-L99

[16] See for instance the discussion at https://discuss.ocaml.org/t/ann-a-dynamic-checker-for-detecting-naked-pointers/5805.

A.3. Challenge 1: the "growing heap" issue

Current OCaml and OCaml multicore get memory chunks from the system allocator. Consequently, the risk that out-of-heap pointers allocated by malloc become in-heap after the memory has been freed is real. The GC will be confused and follow these out-of-heap pointers, leading to a crash.

In practice, the "growing heap" issue is less of a problem than it sounds, at least when using address space reservation. One must be "very unlucky" if a full huge mappable address range becomes entirely available (unreserved) after some memory was allocated inside of it, at a location close to where we choose to extend the reserved address space next.

To convert plausibility into certainty, the language designer can use a few tricks, such as asking the programmer, who knows more about the context, to provide this assumption for them. It seems that the programmer can make a lot of deductions from the nature of the pointers they let inside the heap.

For instance, in the case of out-of-heap allocation for systems programming (ancient heap, shared heap, arena…), this is a complete non-issue even on 32-bit: one just needs to use the same standard address space reservation technique, and never give back virtual address space, so that it cannot overlap with any future assignments for OCaml's heap. By courtesy, for symmetric reasons, we will avoid giving back reserved space to the OS.

Also, the constraint 2. is trivially achieved in case one has 1TB of address space reserved up-front and never need to reserve more. To generalise this option, one can offer a runtime parameter where the user can select a fixed virtual address space size if needed (e.g. for 32-bit).

We expect that the programmer might make further deductions and take further appropriate measures by studying the documentation of whichever malloc they use. For instance, jemalloc offers to retain its virtual address space with the option `opt.retain`.

Making 2. part of the norm further offers the possibility of detecting such errors: when one sees a out-of-heap pointer during marking, one can "taint" the corresponding virtual space, and fail on a future virtual space reservation if one cannot obtain an untainted range. Tainting is not exact (an address range can become reserved before an out-of-heap pointer is seen by the GC), but serves to enforce 2. in practice.

A.4. Challenge 2: 32-bit support & large memory support

We treat 32-bit and 64-bit similarly. With challenge 1 solved, we can do similarly to Go and reserve the address space progressively. This also addresses the problem in 64-bit of machines with more than 1 TB of RAM, and offers the choice of virtual spaces smaller than 1 TB if needed.

We set a prefix size $N \geq 8$ at compile time. We assume the runtime can reserve up to 2^N virtual spaces, of size 2^L bytes aligned at 2^L ($L = 48 - N$ in 64-bit and $L = 32 - N$ in 32-bit). For 64-bit this means virtual spaces ≤ 1 TB (for $N = 16$ this is 4 GB, what the concurrent multicore GC reserves for the minor heap), and ≤ 16 MB in 32-bit. We use a global 1-level or 2-level array of size (at most) 2^N bytes as the virtual space map, so that finding the status of the virtual space a pointer belongs to is done efficiently by shifting and looking up in the array.

We consider 3 possible states for each space in the virtual space map: `Unknown`, `Reserved`, `Tainted`. Thanks to the assumption

2. provided by the programmer and the decision to never give back virtual spaces to the OS, this state is monotonous: entries start with value `Unknown` and may become `Reserved` or `Tainted` at some point, and no longer change.

When a major allocation requires to reserve additional contiguous virtual spaces, such a reservation (aligned at 2^L, of size a multiple of 2^L) is requested to the OS. A hint can be given as to where we would like it to start. The given mapping is checked against the virtual space map: none must be Tainted. In case of success, the corresponding virtual spaces are marked as `Reserved` in the map, and in case of failure then the rule 2. has been violated and the allocation fails.

While not necessary to demonstrate our approach initially, many real-world lessons can be learnt from the detailed sources of the Go allocator. In Go, the size is chosen to be 64 MB or 4 MB depending on the platform[17]. Given the small sizes, special care is taken to reserve space as contiguously as possible to avoid fragmentation. In 64-bit, one hints at a specific place in the middle of the address space unlikely to conflict with other mappings[18]. In 32-bit one asks to start as low as possible and tries to reserve a large chunk initially[19]. There are further details in `malloc.go`[20], concerning more platforms than supported by OCaml.

A.5. Challenge 3: synchronisation

As we explained, when a query to the virtual space map gives `Unknown`, then we know we have an out-of-heap pointer, and we set it the entry to `Tainted`. Thus, tainting virtual spaces has a further interesting consequence: the value of the entry is permanently set to `Reserved` or `Tainted` as early as after the first query against the virtual space map. This gives a simple synchronisation strategy in multicore.

We give each domain a local map that caches the global map. Then, following the same principle as before, each entry of the domain-local map becomes permanently set to `Reserved` or `Tainted` after the first query. If the first query yields `Unknown` in the local map, this time one synchronises with the global map. If it is `Unknown` in the global map, then it is permanently set to `Tainted` globally. Thus any synchronisation when querying the map needs to happen at most once per entry and per domain.

Furthermore, when extending the reserved space during an allocation, one needs to be able to atomically compare and set several adjacent entries. One then can use a mutex, since it only competes with (other extensions of the reserved space and) queries of `Unknown` entries in the global table. In the latter case we need to acquire the mutex as well, which happens at most 2^N times globally.

A.6. Challenge 4: malloc for multicore

An interesting feature of current OCaml multicore is that it defers to the system allocator the management of large blocks (larger than 128 words). This is currently made possible by assuming that code currently relying on naked pointers will be made to break. To avoid reimplementing a high-performance multi-threaded allocator, one can reuse what exists. For instance, jemalloc arenas can be customised to use the memory chunks one provides to it, e.g. carved out of the reserved address space (arena.i.extent_hooks); whereas mimalloc[21] offers a built-in *in-heap?* test. The latter is mildly efficient because it traverses a list of 256 MB blocks, but this presumably could be improved upstream along the current lines.

[17] https://github.com/golang/go/blob/9acdc705e/src/runtime/malloc.go#L219-L231

[18] https://github.com/golang/go/blob/9acdc705e/src/runtime/malloc.go#L489-L498

[19] https://github.com/golang/go/blob/9acdc705e/src/runtime/malloc.go#L549-L564

[20] https://github.com/golang/go/blob/master/src/runtime/malloc.go

[21] https://microsoft.github.io/mimalloc/