PhD topic: Low Level OCaml

Gabriel Scherer, Parsifal, INRIA Saclay

First version: July 19, 2019. Current version: August 8, 2019

Abstract

We propose to study extensions of the OCaml language adding features for low-level programming.

The goal would be to introduce new programming constructs that are less general and expressive, but as a result allow constant-factor speedups due to more efficient implementation strategies: control over allocation, memory representation, local control flow, memory reuse, etc.

On the other hand, those extensions should preserve the good properties of the language: they should preserve safety, enable reasoning about programs, avoid whenever possible to make unportable assumptions about the underlying machine, and strive to remain elegant.

1 General approach

OCaml is a high-level programming language: each feature (functions, datatypes, exceptions...) was designed to be as expressive as possible, sometimes at the cost of a higher runtime cost to support this expressiveness or in exchange for added simplicity. Functions can be returned instead of just called, values can escape their lexical declaration site, exceptions have a dynamically-determined rather than statically-determined handling site, datatypes have an imposed uniform representation that enables a simple compilation strategy for polymorphism, etc.

On the other hand, sometimes programmers would like to write lower-level code, giving up expressiveness or simplicity in exchange for extra performance. In the critical sections of their program, they would be willing to use less expressive or more complex constructs.

Common approaches to this desire include

1. Making the compiler recognize that an expressive feature is used in a less-general way, and optimize this pattern accordingly using lower-level features of the compiler intermediate representation. (One example in OCaml is the implicit transformation of local (non-escaping) references into stack-allocated mutable variables.)

2. Writing the critical code in a lower-level language, often at a loss of safety guarantees, interacting through a foreign-function interface. (C is certainly terrible for safety; Rust is proposed as a FFI choice these days, and may improve the safety aspect)

Instead we propose to extend the *source* language with lower-level features that are less expressive, in exchange for better performance guarantees, and may require a more complex typing discipline, in exchange for preserved safety and modularity/composability. Here are two reasons why having low-level features at the source level is important:

1. It lets experts use them directly in their programs. Expert users writing performance-sensitive code already write their critical sections in a very different style (for example, trying to target the subset of OCaml that never allocate, or at least never boxes floating-point values), but they don't have proper language support, navigating in the dark through hard-learned lessons and profiling information.

2. Optimizations related to "lowering" a general construct used in a less-general way into a lower-level construct can be expressed as source-to-source transformations. We believe that this is key to enabling a good interface for the user and the compiler to discuss optimization choices: the user can visualize the result of optimization in a well-specified language, and this approach should also make it easy to express optimization expectations at the source level.

We would like those exceptions to remain *principled*, according to the following principles:

- Those extensions should not destroy the good properties of ML programming languages: memory safety, typing guarantees, a form of declarativeness, type abstraction and separate compilation. For example, coercing integers into pointers and back is not on the table, nor is forcing ourselves to use whole-program compilation in all circumstances.

- The addition of those extensions should remain transparent to novice users that are not aware of them. In particular, a library whose interface does not expose any low-level types (even if it may internally use some of those low-level features) should behave like a library implemented in high-level OCaml. The reader may be interested in our previous work on formalizing this intuition in Scherer, New, Rioux, and Ahmed (2018).

- We should strive to present these low-level features in a way that is as decoupled as possible from the unportable architectural details of the underlying machine on which the program run. Often, expressing things in machine-specific terms is the easy/direct way, and finding a machine-independent expression of the same idea can be hard work or require original thinking: we want that work to happen.

  One typical example of it is the presentation of tail-recursion. A bad way to explain tail-recursion in ML programs is to explain the stack frames and calling convention, and how tail-call optimization can make recursive programs run in constant stack space. A good way to explain tail-recursion is to study the trace of program reduction in a small-step operational semantics, and remark that the size of the reduction sequence remains constant – and that there is a precise relation between the size of the term and the size of the machine realization of the program.

One can also think of this proposal as a project to move the boundary between the components of a system that one wants to write in OCaml, and those that we need to write in a lower-level language. If we manage to bring more low-level concern to OCaml, without losing the safety and convenience of the language, how far down can we go without wanting to move to another language?

Warning We do not consider it a strict requirement of this PhD topic to upstream all the features explored/prototyped in the OCaml language, but it is certainly possible for the interested student to invest effort into making that possible. We have to be modest with upstreaming hopes, however. Any language change takes a while to be discussed, justified, evaluated, reviewed, and convince stakeholders; it is likely that some of our explorations, even after a fair amount of polishing work, would feel to specialized for upstream inclusion, and instead justify future work on another iteration of a more general feature; or on the contrary, a feature could be judged too invasive for upstream inclusion, and be replaced by a low-tech approach that solves the most pressing user needs with less code change and less generality.

2 Exploration areas

We will list below a few examples of language design problems that could be explored from the general perspective described above. This is not a list of task to be performed, but rather a set of open-ended scenarios that can give an idea of the breadth of the topic. There is more to do than can be hoped in a PhD thesis, so the student would have the freedom to make choices, restrict focus, and propose their own directions.

A first important step would be to find programs in the wild that we believe would benefit from some of the lower-level features we have in mind – this can be started by collecting examples of performance-critical code from OCaml users. One could either start from a program of interest and attempt to understand which change would be most useful for this specific program, or start with a low-level feature of interest and find programs that might benefit from it. In any case, we expect an evaluation of a proposed design to come with a positive impact on a real-world program. This impact could be an improvement in performance, but also in robustness of the code, ease of maintenance, readability or modularity.

2.1 Designing the non-allocating fragment of OCaml

A few OCaml users are actively writing low-latency programs in OCaml, by carefully controlling the amount of allocation that happen in their critical loops – allocation in OCaml is not expensive in itself, but it periodically triggers the garbage-collector which makes performance in general, and latency in particular, harder to reason about.

Currently it is a very painful experience to try to write non-allocating OCaml code. You have to know how the compiler will handle language constructs (for example, which function declarations will allocate a closure and which won't), and using any sort of abstraction is possible but tricky. It's easy to re-introduce allocations without noticing, and the only way to avoid it is to carefully profile your code to measure its allocation rate.

How can we make this experience more pleasant? Are there elegant lower-level language features that could be introduced, so that developers can express their *intent* to avoid allocations, and have a static discipline helping them meet that requirement?

A first step in that direction would be to get in contact with people writing that kind of code, obtaining code samples, and thinking about domain-specific annotations (to express intent), program analyses and warnings/errors that could be added to help. On the longer term, we suspect there is much that could be done to not only discipline users and punish mistakes, but also design language features to improve the expressivity and modularity of that fragment of the language.

Collaboration Jane Street is a natural interlocutor to have for that project, given that they are the main producer of this kind of OCaml code. We have already had discussions with them about it – Thomas Refis, Brian Nigito, Yaron Minsky and Stephen Weeks.

Regions! Besides ensuring that code does *not* allocate, there is interest in controlling *how* code allocates and *where*.

A very simple first step is to give users a mechanism to let them ask for a specific allocation to happen directly in the major heap (because they know it will be long-lived, etc.). The natural next steps are to introduce explicit on-demand region-based memory management, letting expert users allocate specific values directly in separate regions of the OCaml memory with specific memory lifetime and collection rules. (Is "the stack" just a region?)

The functional-programming community remembers regions as a failed experiment from the nineties, but those were implicit/inferred regions; explicit region control for expert may be worth revisiting.

2.2 From unboxed numerics to data representation control

Unboxed numerics In OCaml, most numeric types are "boxed", represented by a pointer to the numeric value stored on the heap – because the memory value itself would not directly fit within the OCaml data-representation and memory model. Boxing introduces extra allocations and indirections, which has a performance cost. The compiler of course performs unboxing optimization whenever possible, and does a decent job at eliminating boxing/unboxing pairs in complex expressions. (Machine integers are not boxed, just "tagged" using bit-shifting operations, and the overhead this introduces is much smaller, negligible in most cases.)

However, letting the optimizer perform unboxing is not always satisfying. It is difficult for users to reason about where boxing will happen or not, requiring expertise in that part of the compiler behavior. Again, there is no clear way for users to express their *intent*, and be warned when a computation boxes when they assumed it wouldn't. (The reliable way to check that is to read the program's intermediate representation.) Finally, one cannot demand unboxing across function boundaries and in general abstraction boundaries, which makes it more difficult than it should be to write generic code and refactor those parts of the programs.

A natural idea is to introduce a new type, `float#`, of unboxed floating-point numbers, and adapt the compiler to support it whenever possible (for example, in the arguments and return values of a function). The first difficulty is the interaction with polymorphism: unboxed floats do not fit within the OCaml memory model, and polymorphic functions of type (for example) `'a -> 'a` are compiled with the expectation that they will receive and return a valid OCaml one-word value; it would be incorrect (and lead to crashes) to allow to call such a function with a `float#` argument. We know how to handle this in the type theory: use *kinds* to classify the types in several categories, in particular the standard kind $\star$, "types whose inhabitants respect the usual OCaml memory model" (all existing OCaml types), and "weird unboxed values" (for `float#` and, similarly, `int32#`, `int64#`, `nativeint#` etc.). A polymorphic variable `'a` would be understood to quantify over the first category only, so that trying to instantiate `forall 'a. 'a -> 'a` with `float#` would fail at compile-time.

The Haskell programming language has already done that work of adding unboxed numeric types, and correctly restricts polymorphism in the way outlined above, so it would be reasonable to start by reusing their design.

Preventing to pass unboxed floats to polymorphic functions means that using unboxed floats will remain painful for users wishing to make use of existing generic libraries, and may still force duplication of code – this is a less expressive, less generic language feature, which comes with downsides. But at least, users would now be able to explicitly ask for floats to be unboxed in some part of the program, and exchange them, including across functions and module boundaries and within datatypes, which makes it possible to use better software engineering practices.

General data representation Besides numerics, there are several situations in OCaml where users would like to have a better control of data-representation choices. Consider the following datatypes:

```
type bar = { b: blah; foo: foo }
 and foo = { f1: t1; f2: t2 }

(* or *)
type bar = B of blah | Foo of foo
 and foo = F1 of t2 | F2 of t2
```

In either cases, the representation of a `foo` value within a `bar` value will be a pointer to an independent block of memory, adding an extra indirection. If there is no other type using `foo` – for example they may be both internal to a module that only exports `bar` as an abstract type – we could hope for a more compact representation, equivalent to the following:

```
type bar = { b: blah; f1: t1; f2: t2 }

(* or *)
type bar = B of blah | F1 of t2 | F2 of t2
```

That representation is more compact, consuming less words in memory to represent a value of type `bar`. On the other hand, in many cases it leads to writing less elegant code, because it fails to group related values together in their own type. Is there a way to write the former, yet get a representation corresponding to the later?

The question is more difficult than it first seems. In particular, in some cases changing from the separate representation to the inline presentation may *reduce* performance (depending on the runtime assumptions and code-generation strategy), for example on the following programs, which would previously just extract a pointer and return it, and may now have to allocate a new value:

```
let extract bar = bar.foo

let extract (Foo foo) = foo
```

Many designs can be considered for this question and have various upsides and downsides. (For example, do we allow to manipulate sub-values of a values as first-class OCaml objects? Is there a way for the runtime and the garbage collector to safely support this?)

One vexing aspect of this problem is that, in many cases, there is no observable performance difference in using the inlined representation: again, the OCaml compiler is good at allocating fast, can combine several allocations together (so `Foo (F1 v1)` will result in a single allocation). Clear gains may be obtained by reducing memory usage of large in-memory workloads, but users can do this manually today (by storing long-lived data in an explicitly compressed form). Again, we should think of those features not as something that will change the life of most OCaml users, but as a tool to enable nicer critical-section code for a few expert users of specialized programs and libraries.

One concrete proposal in this direction would be to expose *unboxed tuples* and possibly even *unboxed sums*, as non-standard OCaml types (separated by kind, just like unboxed floats), and view the standard record and sum types of OCaml as the combination of an unboxed type and a boxing operator that allow to embed those unboxed values into the generic kind $\star$. With that design, finer data-representation choices can be expressed:

```
type (bar : ⋆) = { b: blah; f: foo }
 and (foo : #2) = #{ f1: t1; f2: t2; }
```

In this mockup, `... #` is syntax for unboxed products, and `#2` is the kind of unboxed values that are represented by two consecutive memory words. Counting word size may not be the right choice for unboxed kinds – other aspects relevant to the runtime may have to be made known statically, such as whether the words should be traced by the garbage collector or represent opaque data – and the right kind may not be `#2` if `t1` or `t2` are themselves unboxed product.

Having unboxed datatypes at non-standard kinds is not necessarily the best interface to offer to users who "just" want to express "dear compiler, please inline this part of the value". But it could be a good framework to design a general approach, with more restricted but easier-to-use representation-control devices being expressed on top of it.

Collaboration Leo White and Stephen Dolan (both at Jane Street) have reportedly thinking about these issues for the past year. They would be natural collaborators to get in touch with for this part of the project.

FFI? When designing mechanisms for data representation control, it is tempting to start thinking: if my proposal made it possible to represent *any* possible data encoding for these values, I can use those lower-level types to describe the boundary between OCaml and *any* other language I want to interact with – which sounds much nicer than auto-generated representation-conversion middleware – so let's do this! I think we must be careful to avoid feature creep for these ideas: universal FFI sounds nice, but then it brings us into a tarpit of architecture-dependent choices, and makes it much harder to attain the clean designs we want to impose on these projects. Something working on this would have to thread carefully in terms of generality; looking at what users out there need and what kind of programs they actually write which could benefit from those features could be a good guiding principle.

2.3 Unique ownership and resource safety

Unique ownership, as ensured statically by linear types and separation logic, is a key concept allowing to write efficient code and resource safety. Some examples:

- Unique ownership allows to manipulate mutable datatypes in a declarative way, while preserving referential transparency and all the nice properties of functional code. For example, single-owner references can enable data-race-free mutable-state concurrency. More pragmatically, any hope of extending unboxed representations as discussed in 2.2 to unboxed *mutable* state must rely on unique ownership to make any sense.
- Unique ownership allows in-place reuse of memory also for code that is not necessarily written in an imperative style, but can be compiled as a form of in-place update. This is a key aspect of the related work on Cogent (O'Connor, Chen, Rizkallah, Amani, Lim, Murray, Nagashima, Sewell, and Klein, 2016), and was also discussed in our own work (Scherer, New, Rioux, and Ahmed, 2018).

- Unique ownership is the underlying principle justifying the "unsafe" coercion between the mutable type `Bytes.t` and the immutable type `String.t`, and in general to reason about the correctness of freezing a mutable structure, transitioning from a single-writer single-reader to no-writer many-readers mode.
- Unique ownership is useful to reason about resources in general, and in particular to safely use control abstractions that may capture resources (function closures, multi-shot continuations, etc).

On the more negative side, linearity and ownership are extremely well-trodden problems in programming language design, that have already received and continue to receive a lot of attention. In our experience, designing a static enforcement mechanism for unique ownership requires a lot of hard work, and is not in any way guarantee to lead to actual performance gains, unless it is coupled with other low-level aspects as discussed in the other sections – dynamic enforcement of unicity, or hybrid schemes such as copy-on-write, may have more low-hanging fruits. It is not clear to us that this is the most effective direction to focus on for this project.

Collaboration Natural points of contact on this topic are François Pottier, who worked on type systems built on top of separation logic in the Mezzo project, and Guillaume Munch-Maccagnoni, who has a proposal for linearity-inspired resource handling in an OCaml dialect, with both static and dynamic safety enforcement mechanisms.

2.4 Richer control-transfer primitives

The two first-class control-flow mechanisms of OCaml are unary functions and dynamic exceptions. It would be interesting to explore lower-level forms of functions (closer to the free-form transfer as typically present in jump-based assembly languages), and lower-level forms of exceptions (more restricted than full dynamic exceptions).

For exceptions, a natural idea is to support local exceptions, whose values cannot escape and whose handler does not extend to callees, which could be compiled into simple jumps instead of handlers on stack. Such a mechanism in fact already exists in the OCaml intermediate languages (static-catch / static-raise), and is used to compile pattern-matching into more primitive control-flow constructs. It could also be generalized into a notion of mutually recursive local functions, which cannot escape and are always called in tail-position.

For functions (local or not), one could consider multi-argument functions, as a more primitive form that both currified and tupled functions desugar to. This may be of importance to design lower-level calling conventions for unboxed types, for example. One could also think of functions returning several values (instead of having to box them in a tuple), which would allow to modularize some critical sections where allocations (of intermediate tuples) are frowned upon.

Maybe more surprising, one could have function-like values with multiple return points. Having several return points is emulated today by either returning a sum type (the caller matches on the sum and routes to the right code; multi-return would eliminate the intermediate sum), or offering a continuation-passing-style interface with several continuations (typically in backtracking code, a success and a failure continuation; multi-return could eliminate a closure allocation). See Shivers and Fisher (2004) for examples using multi-return functions.

References

Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: Bringing down the cost of verification. In *ICFP*, 2016.

Gabriel Scherer, Max S. New, Nicholas Rioux, and Amal Ahmed. FabULous interoperability for ML and a linear language. In *FoSSaCS*, April 2018.

Olin Shivers and David Fisher. Multi-return function call. In *ICFP*, 2004.