Why learn OCaml?

Brett Gilio

<2020-08-29 Sat 15:28>

Contents

This article is an overview of the features of OCaml and the rich type system it uses. The target audience of the article is for people who have familiarity with programming in a statically-typed environment, and understand at least conceptually what a type system is used for. This is *not* an introduction to functional programming, nor the core syntax/libraries of OCaml. Knowledge of type theory is not assumed, but you will be well-served to approach some of the type-theoretic concepts used in this article with an open mindset.

1 What is OCaml?

In formal terms, OCaml[1] is an implementation of the Caml dialect, started by Xavier Leroy with the INRIA Gallium[2] research team. Influenced primarily by Standard ML[3] and LISP (as well as C and Pascal for their imperative qualities), OCaml was initiated as a functional programming language designed to offer stellar performance, strong static typing, Hindley-Milner (and later bidirectional HM-successor-style) type inference. Eventually the Caml dialect, in the style of Standard ML, took the extension of an impure and effectful I/O system and implemented a class-based object system which still conformed to the strong type system.

OCaml is most widely recognized, as with most ML/Haskell dialects, for their exceptional use in implementing compilers and type systems. Many successful proof assistants, compiler toolchains, static analyzers, and SMT-style solvers have been implemented in OCaml. For example, the Coq[4] proof assistant is well known for implementing MLTT-influenced Cartesian closed[5] dependent types in OCaml, as well as being one of the more prominent examples of an interactive proof assistant used by formal mathematicians and verification-focused programmers today. Additionally, OCaml has found prominent success in commercial areas, such as Jane Street[6] and their in-house trading system which manages upwards of millions of stock trades every day.

While OCaml does not feature full formal specification at the type level like Standard ML, it does offer a high degree of both determinism, inferencing, and expression at the type-system level. Many who work on OCaml may feel that while formal specification is nice, as it unifies and codifies an implementation's behavior in a mathematical and logically-sound way (guaranteeing soundness of the type system), the result can slow progress of the language and prevent it from keeping up-to-date with the needs of current programmers. For example, while the Standard ML '97 revision[7] does not feature Unicode support, and neither does the Standard ML Basis Library[8] this has not been an issue for OCaml. What OCaml lacks in a formal specification, it readily makes up for in having a quality type-

[1]`https://ocaml.org/`
[2]`http://gallium.inria.fr/`
[3]`https://smlfamily.github.io/`
[4]`https://coq.inria.fr/`
[5]`https://ncatlab.org/nlab/show/cartesian+closed+category`
[6]`https://www.janestreet.com/`
[7]`https://smlfamily.github.io/sml97-defn.pdf`
[8]`https://smlfamily.github.io/Basis/`

system, pseudo/partial-specification, and set of libraries to implement `UTF-8`, `UTF-16`, `UTF-32` strings and other encodings.

Where there is negative reaction toward OCaml, particularly from people first approaching the language (typically from environments like Racket[9], Haskell, or even Python) is that the standard library for OCaml is quite small. This, however, is intentional! OCaml has taken inspiration from C in this regard, the core components, syntax, and implementation of OCaml is highly portable and readily extensible. There are many visible APIs for working with OCaml from other languages (with C-runtimes) precisely because the implementation is succinct. Admittedly, however, the standard library of OCaml has shown its age in recent years and many components have been implemented or reworked to bring the OCaml standard library up-to-date (but this still seems to be a work in progress[10]).

In my opinion, when it comes to OCaml and its object-oriented system (and perhaps contrary to other people, like Scala[11] fans), I feel the object/class system is well designed and considered. The consistent rebuke I hear to this is that it feels like an after-thought, given that the progression was from Caml -> OCaml with the addition of the object/class system. However, I personally disagree with this assessment. To me, the Java (and subsequently Scala) style of objects, classes, constructors, inheritance, overloading, and polymorphism is not as precise or elegant as what is offered by OCaml. Additionally, the type safety (while greatly improved in Scala) is not as implicit, nor as secure in Java where object constructions routinely lead to errors from downcasting (which OCaml prohibits!).

1.1 A look at [Objective] Caml

To continue the previous section, and using (probably the best top-level in all of programming) `utop`[12], we can take a look at some of the basics of object-oriented programming using the OCaml environment.

```
class point =
  object
    val mutable x = 0
    method get_x = x
    method move d = x <- x + d
```

[9] https://racket-lang.org/

[10] https://github.com/ocaml/ocaml/issues/7812

[11] https://www.scala-lang.org/

[12] https://github.com/ocaml-community/utop

```
end;;
(* class point :
  object val mutable x : int method get_x : int method move : int -> unit end *)
```

The above source code should be rather self explanatory. We are defining a point class with a mutable object `x`. We can see the type signature below the declared class defining many of the implicit types of each of the objects, as well as the methods. Using this class we can now instantiate the class from the definition above, and then use the methods of the class to perform some computations.

```
let instance = new point;;
(* val instance : point = <obj> *)

instance#get_x;;
(* - : int = 0 *)

instance#move 5;;
(* - : unit = () *)

instance#get_x;;
(* - : int = 5 *)
```

However, classes do not have to be used to create objects. An object can be constructed immediately without going through a class, where inheritance is not preferred, or where references to expliticly bound `self` may be preferred. Additionally, type abbreviations are not permitted in immediate objects, which can lead to problematic type references (which OCaml responsibly dictates and prevents at compile-time).

```
let ints = ref [];;
(* val ints : '_weak1 list ref = {contents = []} *)

class external_reference_self =
  object (self)
    method n = 1
    method register = ints := self :: !ints
end;;
(* Error: This expression has type < n : int; register : 'a; .. >
          but an expression was expected of type 'weak1
          Self type cannot escape its class *)
```

The issue above, as stated, is that when `self` is placed within the external reference to immediately created object `ints`, what is created is an unfavorable and unsound situation where we can not extend `ints` using the `self` reference (as there is a mismatched type). The ability of the OCaml type system to not only recognize, but also understand the implicit types of the above class and (failed) attempt at inheritance prevents a situation where object classes could very well allocate information to objects in (or, outside rather) of the class which would not be properly handled for computation. This could further lead to core dumps in insecure language runtimes where this problem is not identified at compile-time.

OCaml does have a resolution to this situation (as expected if you have ever used C++ or Java), coercions[13, 14, 15].

More detailed examples of object-oriented programming in OCaml can be found here.

1.2 How *strong* is this strong static typing?

In today's world, it is not uncommon to have strong typing with your favorite programming language. The obvious exclusions here are C/C++, but even with the right compiler extensions (or static analyzers), you can get some of the same guarantees of strongly typed languages in traditionally weakly typed environments. Taking the example above of insecure languages, where undefined behavior is routine and type security is missing (or not as strong as LISP/ML/Haskell-family languages); the result of misattributing types, passing data between incompatible objects and methods, or incorrect memory behavior mangling the references to live/dead objects can be disasterous and hard to debug.

People who have not experienced the beauty of a static type system like the one in OCaml will find themselves unaware of the beauty beneath it. It is particularly common (though gradually changing) that people with experience in strong, statically typed programming languages have not gotten a feel for the benefits of Hindley-Milner type inference. The result of this is that programmers in systems like Java feel as though their job can have a cumbersome task akin to bookkeeping and accounting.

While nowadays programming languages like Java and Python have some level of annotation-less type inferencing, it is not nearly as powerful as the

[13]`https://caml.inria.fr/pub/docs/manual-ocaml/objectexamples.html#s%3Ausing-coercions`

[14]`http://docs.cascading.org/cascading/2.6/userguide/html/ch07s02.html`

[15]`https://www.learncpp.com/cpp-tutorial/implicit-type-conversion-coercion/`

level of abstraction one can attain using OCaml.

1.2.1 Damas-Hindley-Milner in a "nutshell"

The DHM (or just HM) type system in slightly technical terms is an abstraction for providing a typed lambda calculus. In other words, you can take the functional (in a mathematical sense) aspects of a particular syntax and make formal rules about their behavior; expressing all forms of computations[16].

The Hindley-Milner implementation offers the logical correctness of parametric typing rules to the structure of the lambda calculus. From this combination of typed logical correctness, and deterministic behavior we can derive the type of any expression (or term) from a specified environment. This feature is called the "principle type". Perhaps the most famous of proven algorithms that offers this level of type inferencing is "Algorithm W".

A lot of the notions of type theory, and type inferencing can be very daunting to people looking to get into more formal, and type correct programming languages. As explained by Martin Grabmüller[17], this business is often inflated with high categorical and type theoretic mathematics that make the task of understanding *what* is happening harder than it is at the core.

So, taking a look at the core principle behind Algorithm W, without the (too much) mathematical jargon or notation we have two components:

- An input to Algorithm W, an expression and an environment. This is simply any and all information about the function/object/parameter and the modules/function-use we are working in.

This results in...

- An output of a type for that given expression, and the return of a principle (overall) type.

The result of this algorithm, detailed using function expressions, should result in the appropriate types for each expression based on the syntax of that expression and its environment. (Note: the expression in the first example below shows a mapping of `int -> 'a`, in OCaml and Standard ML `'a` is used here to designate a generic type.)

[16]`https://plato.stanford.edu/entries/church-turing/`

[17]`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.7733&rep=rep1&type=pdf`

```
fun f -> f 3;;
(* - : (int -> 'a) -> 'a = <fun> *)

fun f -> f (f 3);;
(* - : (int -> int) -> int = <fun> *)

fun f -> f (f "OCaml!");;
(* - : (string -> string) -> string = <fun> *)
```

Combining these elements together we are able to logically derive the type of complex abstractions, such as functors[18], mappings, pattern matching, or simple functions. This example of Algorithm W as offered by Robin Milner has been extended many times. If you are interested in type theory, I suggest the following:

- Pierce-Turner: Bidirectional, Local Type Inference
- Dunfield-Krishnaswami: Bidirectional, Higher-Rank Polymorphic Typing

Parts of these modifications can also be found in programming languages such as Rust which offers a modified HM-algorithm using some of the Dunfield-Krishnaswami (and accordingly Pierce-Turner) research to implement subtyping, local/region inference, and higher-ranked types.

The influence of languages like OCaml has been seen extended to Rust, and despite my personal reservations about many of the claims of that language it is astounding to see progress made in expressive type systems being mainstreamed.

1.3 Unification and Pattern Matching

Many of the impressive advantages of OCaml can be felt in features you will not find in other "strong" statically typed languages. Picking up where we left off, type inference offers many advantages when it comes to creating powerful multifaceted and polymorphic expressions.

Taking the concept of type inference and abstracting up one level we are introduced to type unification. type unificiation takes the result of Algorithm W (the inferred type) and finds a way to create a valid substitution of two expressions where both expressions are of equivalent types. In simpler wording, unification allows two type expressions to match.

[18]`https://www.math3ma.com/blog/what-is-a-functor-part-1`

When we are able to have a result with a matching pair of types we are able to achieve pattern matching.

```
let unification_example x =
  match x with
    1 -> "Foo"
  | 2 -> "Bar"
  | 3 -> "Fizz"
  | 4 -> "Buzz"
  | _ -> "You entered: " ^ string_of_int x;;
(* val unification_example : int -> string = <fun> *)

unification_example 2;;
(* - : string = "Bar" *)

unification_example 4;;
(* - : string = "Buzz" *)

unification_example 9;;
(* - : string = "You entered: 9" *)
```

As in C with the multiway conditional, `switch`, we are able to make any number of possible values (`1...4`, with a wildcard `_`) and have a return based on the value given to `x`. However, do not be fooled, what is happening here is not like `switch` or nested-`if` statements. The result of this pattern matching process is permitted by the type unification of `x` in `unification_example`.

The power of pattern matching as a result of type unification is not limited to just this multiway conditional scenario. You can find pattern matching to be used in data destructuring, variable binding, compiler tokenization, case analysis, and a more elegant solution to exception handling[19].

1.3.1 Note: Type Soundness & Decidability

As type soundness is not constrained to monomorphic types as specified in the Hindley-Milner algorithm and Standard ML formal specification, it is possible to get unsound results where decidability of the expression is not successful. OCaml has relaxed value restrictions (and subtyping rules)[20, 21, 22].

[19] https://www.cs.cornell.edu/courses/cs3110/2014fa/lectures/5/lec05.pdf

[20] https://caml.inria.fr/pub/old_caml_site/caml-list/1507.html

[21] https://www.math.nagoya-u.ac.jp/~garrigue/papers/inria080613.pdf

[22] https://caml.inria.fr/pub/docs/oreilly-book/html/book-ora016.html

```
let unsound_example = function x when x = x -> true;;
(* Characters 22-51:
    let unsound_example = function x when x = x -> true;;
                          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
All clauses in this pattern-matching are guarded.
val unsound_example : 'a -> bool = <fun> *)
```

1.4 Higher-Ranked Types and Polymorphism

As explained above the OCaml type checking system is not as decidable as Standard ML, and this is due to the relaxing of value/rank restriction. This is not unique to OCaml, as it is also the case in the type systems of Haskell (at *rank-n*) and Rust. When dealing with functional programming languages it is often desired to be able to use a term in differently-typed contexts. This is different from the above examples of unification/pattern matching where we are able to determine equivalent types.

When using terms of a type in a differently-typed context we are asserting the polymorphic quality for the type of that term. Polymorphic types use something in mathematical logic known as universal quantification, which may ring a bell if you have taken a course on discrete/predicate logic. The meaning of this "universal quantification" is expressed with a proposition where a type rule can be satisfied by any member of a domain. In other words, universal quantification in functional programming languages allow for a type may be given a different binding based on the context they are quantifying.

You have likely been exposed to parametric polymorphism in other languages, such as C++ with the template system or simple generic programming in Java. An extension of the OCaml Hindley-Milner style type system is the inclusion of subtype polymorphism. Which, like generics allows for an object or expression of a certain type to work correctly if passed to an object with a particular compatible subtype. We have seen in the section on object-oriented programming in OCaml an example where an expression is able to return a generic type.

1.4.1 Parametric Polymorphism

If you have experienced OCaml in the past, you will likely be aware that function/operator overloading is not valid in the current implementation. The reason for this is because the Hindley-Milner inferencing type system and overloading operations can often conflict with each other. There have been attempts in the past to workaround this issue, namely with *ad-hoc* modular implicits[23]. However, subtyping in OCaml is as powerful (if not more powerful and performance-efficient) than the template system found in C++.

In Ocaml, when type inference determines that an expression is decidably valid for any type (*a la. generics*), the type system automatically makes the expression polymorphic and parametric with type variables. We have been introduced to type variables in the section covering object-oriented programming in OCaml, with the notation of the generic/arbitrary type `'a`. Expanding on that knowledge, and the concept of multiple type variables, we can introduce parametric polymorphism and observe value restriction.

```
let id x = x;;
(* val id : 'a -> 'a = <fun> *)

let weak_id = id id;;
(* val weak_id : '_weak1 -> '_weak1 = <fun> *)

weak_id 1;;
(* - : int = 1 *)

weak_id "OCaml";;
(* Characters 8-15:
     weak_id "OCaml";;
             ^^^^^^^
Error: This expression has type string but an expression was expected of type int*)
```

As we see above, we are given the arbitrary type `'a` in the first statement, which when assigned to the polymorphic identifier `weak_id` is transformed into `'_weak1`. The `_` wildcard in the type expression simply means of any unknown type, the universal quantifier. This works when we apply the expression `weak_id` to type `int`, but directly afterwards we are now restricted to expressions of type `int -> int`.

The example above is only weakly polymorphic. The ability to retain full polymorphism we use a technique known as "Eta expansion", and return fully generalized types.[24]

[23]https://arxiv.org/pdf/1512.01895.pdf
[24]http://ocamllabs.io/iocamljs/type_inference.html

```
let map_id l = List.map id l;;
(* val map_id : 'a list -> 'a list = <fun> *)
```

1.4.2 Higher-Rank Polymorphism

This section I graciously am parts lifting from /dev/musings, who explained the situtition so well.

Higher-rank polymorphism in OCaml can be a sticky situation, with many different possible implementations. At its core the OCaml type system only allows for rank-1 polymorphism, which we saw in the last section where `let`-binding is used to introduce polymorphic expressions. Higher-ranked polymorphism comes with issues of decidablity with Hindley-Milner type inference. Generally, the higher the rank the more arbitrary categorical arrows are involved.

- *Rank-1* (`let`-binding) polymorphism is always predicative and decidable, because all universal quantifiers do not appear as parameters to type constructions.
- *Rank-k* polymorphism is impredicative and decidable.[25]
- *Rank-n* polymorphism is impredicative and not decidable.

Since OCaml is strictly *prenex* (rank-1) polymorphic we have to create rank-k solutions using type records[26]. While explicit type annotations can exist in many places in OCaml, the place where it is most abundant and reusable to the type system itself is in the type record field. We can simulate a polymorphic type `r` in a record field, as follows:

```
type r = {f : 'a . 'a -> 'a};;
(* type r = { f : 'a. 'a -> 'a; } *)
```

Now, reusing our `let`-polymorphic expressions from earlier, we can now rewrite them using the injective expression for type `r`.

```
let id x = x;;
(* val id : 'a -> 'a = <fun> *)

let map_pair r (p1, p2) = (r.f p1, r.f p2);;
```

[25] https://wiki.haskell.org/Impredicative_types

[26] https://dev.realworldocaml.org/records.html

```
(* val map_pair : r -> 'a * 'b -> 'a * 'b = <fun> *)

let sample = map_pair {f = id} (3, true);;
(* val sample : int * bool = (3, true) *)
```

The particular difference between this example and higher-ranked types in Standard ML is as mentioned earlier Standard ML works with monomorphic types. When dealing with monomorphic types, we are constrained in the kinds of data structures that can be created using higher-ranked polymorphism.

1.5 The Module System

Structure control and modules are a common tradition nowadays in programming languages. However, not many languages come close to the expressive power of the module system in OCaml. The OCaml module system is actually derived from the one started by Standard ML. Modules in these type-inferred languages are first-class, in that a module can come with its own set of interfaces, signatures, and submodules. One of the only languages that approximates the power offered by OCaml and Standard ML in its module abstraction is, yet another ML-dialect, F#.

OCaml's module system is concerned primarily with a module and its respective signature. Typically a module in another non-ML language will be strictly stratified, in that you can not treat modules as values to an expression. However, in OCaml and the ML-dialects this is corrected by the notion of *first-class modules*[27].

For example:

```
let sort (type s) (module Set : Set.S with type elt = s) l =
    Set.elements (List.fold_right Set.add l Set.empty);;
(* val sort : (module Set.S with type elt = 's) -> 's list -> 's list = <fun> *)

let make_set (type s) cmp =
  let module S = Set.Make(struct
      type t = s
      let compare = cmp
  end) in
  (module S : Set.S with type elt = s);;
(* val make_set : ('s -> 's -> int) -> (module Set.S with type elt = 's) = <fun> *)
```

[27]http://www.cs.cornell.edu/courses/cs3110/2020sp/manual-4.8/manual028.html

With the module system, we can expand the module type signature, or functor type which is inferred for a module much in the same way we can infer the types of expressions. This is useful for making module types reusable, or implementing module-selection in pattern-matching, or any number of other possibilities.

For example, using the signature idiom (abbreviated):

```
module type MYHASH = sig
   include module type of struct include Hashtbl end
   val replace: ('a, 'b) t -> 'a -> 'b -> unit
end;;
(* module type MYHASH =
  sig
    type ('a, 'b) t = ('a, 'b) Hashtbl.t
    val create : ?random:bool -> int -> ('a, 'b) t
    val clear : ('a, 'b) t -> unit
    val reset : ('a, 'b) t -> unit
    val copy : ('a, 'b) t -> ('a, 'b) t
    val add : ('a, 'b) t -> 'a -> 'b -> unit
    val find : ('a, 'b) t -> 'a -> 'b
    val find_opt : ('a, 'b) t -> 'a -> 'b option
    val find_all : ('a, 'b) t -> 'a -> 'b list
    val mem : ('a, 'b) t -> 'a -> bool
    val remove : ('a, 'b) t -> 'a -> unit
    val iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit
    val filter_map_inplace : ('a -> 'b -> 'b option) -> ('a, 'b) t -> unit
    val fold : ('a -> 'b -> 'c -> 'c) -> ('a, 'b) t -> 'c -> 'c
    val length : ('a, 'b) t -> int
    val randomize : unit -> unit
    val is_randomized : unit -> bool
    type statistics =
      Hashtbl.statistics = {
      num_bindings : int;
      num_buckets : int;
      max_bucket_length : int;
      bucket_histogram : int array;
    }
    val stats : ('a, 'b) t -> statistics
    val to_seq : ('a, 'b) t -> ('a * 'b) Seq.t
    val to_seq_keys : ('a, 'b) t -> 'a Seq.t
    val to_seq_values : ('a, 'b) t -> 'b Seq.t
    val add_seq : ('a, 'b) t -> ('a * 'b) Seq.t -> unit
    val replace_seq : ('a, 'b) t -> ('a * 'b) Seq.t -> unit
    val of_seq : ('a * 'b) Seq.t -> ('a, 'b) t
...*)
```

The powerful properties of first-class modules allows not only for rigorous applications to an existing code-base, but also creates and aligns with highly expressive code. With first-class modules, the types of abstractions can be permitted to generalize module use with functors, or create data structures out of modules which are functional and pure.

1.6 Generalized Algebraic Datatypes

Type parameters can be used in a multifaceted way, allowing for additional constratins and abstractions which may change depending on how the value being used is constructed. In earlier sections we addressed the concept of universal quantification, but we can also apply a similar concept to type rules known as the existential quantifier.

Again, back to discrete/prediate logic, when we deal with existential quantification, we are thinking of situations where in a logical system there exists some variable which can be applied to our system from a domain. This is different from universal quantification, where the variable can be anything (or all members in a logical system).

OCaml uses existential quantification to create something known as an equality-qualified type, or a generalized algebraic datatype. While this notion is not unique to OCaml (and can similarly be found in Haskell), GADTs can be thought of as an existential extension to pattern matching. Additionally, you can find similar a similar concept in the OCaml-implemented proof assistant Coq with the introduction of inductive data types[28].

Constraints applied to a value constructor can be applied in a similar manner to pattern matching. However, instead of using a specified value with a specific datatype, we can instead construct a generalized type and us this when applying it to a function which computes similar to pattern matching[29].

```
type _ term =
```

[28]http://adam.chlipala.net/cpdt/html/InductiveTypes.html#:~:text=Coq%20inductive%20types%20generalize%20the,the%20possibility%20for%20type%20dependency

[29]https://caml.inria.fr/pub/docs/manual-ocaml/gadts.html

```
  | Int : int -> int term
  | Add : (int -> int -> int) term
  | App : ('b -> 'a) term * 'b term -> 'a term;;
(* type _ term =
    Int : int -> int term
  | Add : (int -> int -> int) term
  | App : ('b -> 'a) term * 'b term -> 'a term *)

let rec eval : type a. a term -> a = function
  | Int n    -> n
  | Add      -> (fun x y -> x+y)
  | App(f,x) -> (eval f) (eval x);;
(* val eval : 'a term -> 'a = <fun> *)

let two = eval (App (App (Add, Int 1), Int 1));;
(* val two : int = 2 *)
```

1.7 Lisp Macros & Syntax Extension

As OCaml, and the ML-family come from the tradition of LISP, you can often find OCaml to have many of the niceties that come bundled with the expressive power of LISP-dialects. If you have used LISP, you know that the language itself is intrinsically extensible, meaning that you can extend the language itself without recompiling (using macros). This same power is also found in the OCaml language using the preprocessor-pretty-printer of OCaml, `camlp5` (which has replaced and deprecated `camlp4`[30]).

More information about this functionality can be found here.

2 Projects using OCaml

Here is an unorganized list of projects that I enjoy which are using OCaml.

- MirageOS - Unikernels
- CoolTT - Cartestian Cubical Type Theory Proof Assistant
- Frama-C - Extensible Static Software Analyzer
- CompCert - Formally Verified C99 Compiler (Note: This is *non-free* software)

[30]https://whitequark.org/blog/2014/04/16/a-guide-to-extension-points-in-ocaml/

- Coq - Proof Assistant
- F* - An effectful formally verified language
- ReasonML - ML-style/Javascript ecosystem
- Why3 - Deductive Program Verification
- Lablgtk - Stubless GTK Bindings for OCaml
- OWL - OCaml for Scientific Computing
- Camlp5 - Preprocessor-Pretty-Printer of OCaml

3 Tutorials for OCaml

Great! Perhaps you are convinced of some of the functionality of OCaml and you are now looking for some actual tutorials that teach you how to do things.

- Python to OCaml
- Learn OCaml in Y Minutes
- OCaml for the Masses
- Real World OCaml
- CS 3110 - Data Structures and Functional Programming @ Cornell University
- Awesome OCaml

Have a response?

Responses and discussion pertaining to any of the blog entries on my website are welcome! Start a discussion on the mailing list by sending an email to ~brettgilio/blog-discussion@lists.sr.ht.

Errata:

- *<2020-09-03 Thu 18:03>* Correct some typos and ambiguous language.
