acmqueue

OCaml for the Masses

Why the next language you learn should be functional

Yaron Minsky, Jane Street

Sometimes, the elegant implementation is a function. Not a method. Not a class. Not a framework. Just a function. - John Carmack

Functional programming is an old idea with a distinguished history. Lisp, a functional language inspired by Alonzo Church's lambda calculus, was one of the first programming languages developed at the dawn of the computing age. Statically typed functional languages such as OCaml and Haskell are newer, but their roots go deep—ML, from which they descend, dates back to work by Robin Milner in the early '70s relating to the pioneering LCF (Logic for Computable Functions) theorem prover.

Functional programming has also been enormously influential. Many fundamental advances in programming language design, from garbage collection to generics to type inference, came out of the functional world and were commonplace there decades before they made it to other languages.

Yet functional languages never really made it to the mainstream. They came closest, perhaps, in the days of Symbolics and the Lisp machines, but those days seem quite remote now. Despite a resurgence of functional programming in the past few years, it remains a technology more talked about than used.

It is tempting to conclude from this record that functional languages don't have what it takes. They may make sense for certain limited applications, and contain useful concepts to be imported into other languages; but imperative and object-oriented languages are simply better suited to the vast majority of software engineering tasks.

Tempting as it is, this conclusion is wrong. I've been using OCaml in a production environment for nearly a decade, and over that time I have become convinced that functional languages, and in particular, statically typed ones such as OCaml and Haskell, are excellent general-purpose programming tools—better than any existing mainstream language. They also have an enormous range, being well suited for small scripting tasks as well as large-scale high-performance applications. They are not the right tool for every job, but they come surprisingly close.

THE MOVE TO OCAML

Most of my experience programming in OCaml came through my work at Jane Street, a financial firm founded in 2000. Nine years ago, no one at Jane Street had heard of OCaml. Today, Jane Street is the biggest industrial user of the language, with nearly two million lines of OCaml code and 65 (at last count) employees who use the language on a daily basis. Probably the best way to explain what makes OCaml such an effective tool is to start by explaining how and why that transformation took place. To understand that, you first need to understand something about what Jane Street does.

Jane Street's core business is providing liquidity on the world's electronic markets. It is, essentially,

a middleman. It continually places orders for many different securities on many different exchanges. Each order expresses a willingness either to buy or to sell a given security at a given price, and, collectively, they are an advertisement to the markets of Jane Street's services. Through these orders, the firm buys from people who need to sell and sells to people who need to buy, making money from the gap between the buying and selling prices. All the time it is competing on price with other players trying to do the same thing.

Electronic liquidity provision is technologically intense, not only because of the computational resources that need to be deployed (an enormous amount of data needs to be consumed, analyzed, and responded to in real-time), but also in terms of the complexity of the enterprise—trading can cross multiple exchanges, regulatory regimes, security classes, and time zones. Managing the resulting complexity is a daunting task that requires a significant investment in software.

All this technology carries risk. There is no faster way for a trading firm to destroy itself than to deploy a piece of trading software that makes a bad decision over and over in a tight loop. Part of Jane Street's reaction to these technological risks was to put a very strong focus on building software that was easily understood—software that was readable.

Reading code was part of the firm's approach to risk from before we had written our first line of OCaml. Early on, a couple of the most senior traders (including one of the founders) committed to reading every line of code that went into the core trading systems, before those systems went into production. This was an enormous ongoing time investment and reflected the high level of concern about technology risk.

I started at Jane Street the year after I finished my Ph.D., working there part-time while doing a post-doc. My work at Jane Street was focused on statistical analysis and optimization of trading strategies, and OCaml was the primary tool I used to get the analysis done. Why OCaml? I had learned it in grad school and fell in love with the language then. And OCaml was a great match for this kind of rapid-prototyping work: highly performant, yet faster and less error-prone than coding in C, C++, or Java.

I was convinced that my stint at Jane Street would be short and the code I was writing was all throw-away, so I made a choice to maximize my own productivity without worrying about whether others could use the code later. Six months and 80,000 lines of code later, I realized I was wrong: I took a full-time position at Jane Street and soon started hiring to create a research group there.

At this time, the firm was casting around for a new approach to building software. The systems that powered the company in its first years were primarily written in VBA and C#. Indeed, the core trading systems themselves were Excel spreadsheets with a great deal of custom VBA code. This was a great way to get up and running quickly, but it was clear from the start that this was not a sustainable approach.

In 2003, Jane Street began a rewrite of its core trading systems in Java. The rewrite was eventually abandoned, in part because the resulting code was too difficult to read and reason about—far more difficult, indeed, than the VBA that was being replaced. A big part of this was Java's verbosity, but it was more than that. The VBA code was written in a terse, straight-ahead style that was fairly easy to follow. But somehow when coding in Java we built up a nest of classes that left people scratching their heads when they wanted to understand just what piece of code was actually being invoked when a given method was called. Code that made heavy use of inheritance was particularly difficult to think about, in part because of the way that inheritance ducks under abstraction boundaries.

In 2005, emboldened by the success of the research group, Jane Street initiated another rewrite of its core trading systems, this time in OCaml. The first prototype was done in three months, and was up and trading three months after that. The use of OCaml in the company has only expanded since then. Today it is used to solve problems in every part of the company, from accounting to systems administration, and that effort continues to grow. In recent years, the trading side of the firm has increased its use of the language, and OCaml training is now a standard part of the curriculum for new trading hires. Overall, the transition to OCaml has been a huge success, resulting in far stronger technology than we could have achieved otherwise.

WHY OCAML?

What is it about the language that makes it work so well? Here's a short summary of what I perceive as OCaml's key strengths.

- **Concision.** Our experience with OCaml on the research side convinced us that we could build smaller, simpler, easier-to-understand systems in OCaml than we could in languages such as Java or C#. For an organization that valued readability, this was a huge win.
- **Bug detection.** Programmers who are new to OCaml are often taken aback by the degree to which the type system catches bugs. The impression you get is that once you manage to get the typechecker to approve of your code, there are no bugs left. This isn't really true, of course; OCaml's type system is helpless against many bugs. There is, however, a surprisingly wide swath of bugs against which the type system is effective, including many bugs that are quite hard to get at through testing.
- **Performance.** We found that OCaml's performance was on par with or better than Java's, and within spitting distance of languages such as C or C++. In addition to having a high-quality code generator, OCaml has an incremental GC (garbage collector). This means the GC can be tuned to do small chunks of work at a time, making it more suitable for soft real-time applications such as electronic trading.
- **Pure, mostly.** Despite how functional programmers often talk about it, mutable state is a fundamental part of programming, and one that cannot and should not be done away with. Sending a network packet or writing to disk are examples of mutability. A complete commitment to immutability is a commitment to never building anything real.

Mutable state has its costs, however. Mutation-free code is generally easier to reason about, making interactions and dependencies between different parts of your codebase explicit and easier to manage. OCaml strikes a good balance here, making mutation easy, but making immutable data structures the default. A well-written OCaml system almost always has mutable state, but that state is carefully limited.

Perhaps the easiest of these advantages to demonstrate concretely is that of concision. The importance of concision is clear: other things being equal, shorter code is easier to read, easier to write, and easier to maintain. There are, of course, limits: no good is done by reducing all your function names to single characters, but brevity is nonetheless important, and OCaml does a lot to help keep the codebase small.

One advantage OCaml brings to the table is type inference, which obviates the need for many type declarations. This leaves you with code that is roughly as compact as code written in dynamic languages such as Python and Ruby. At the same time, you get the performance and correctness benefits of static types.

Consider the following OCaml function map for transforming the elements of a tuple.

```
let map f(x,y,z)=
   (f x, f y, f z)
```

Here, map is defined as a function with two arguments: a function f and a triple (x,y,z). Note that f x is the syntax for applying the function f to x.

Now consider what this would look like in C# 4.0. The C# code, while functionally equivalent, looks cluttered, with the real structure obscured by syntactic noise.

```
Tuple<U,U,U> Map<T,U>(Func <T,U> f, Tuple<T,T,T> t)
{
    return new Tuple<U,U,U>(f(t.item1), f(t.item2), f(t.item3));
}
```

Another source of concision is OCaml's notation for describing types. At the heart of that notation is the notion of an *algebraic datatype*. Algebraic datatypes are what you get when you have a system that includes two ways of building up new types: products and sums.

A *product type* is the more familiar of the two. Tuples, records, structs, and objects are all examples of product types. A product type combines multiple values of different types into a single value. These are called product types because they correspond mathematically to Cartesian products of the constituent types.

A *sum type* corresponds to a disjoint union of the constituent types, and it is used to express multiple possibilities. Where product types are used when you have multiple things at the same time (a and b and c), sum types are used when you want to enumerate different possibilities (a or b or c). Sum types can be simulated (albeit somewhat clumsily) in object-oriented languages such as Java using subclasses, and they show up as union types in C. But the support in the type systems of most languages for interacting with sum types in a safe way is surprisingly weak.

Figure 1 provides an example of algebraic datatypes at work. The code defines a type for representing Boolean expressions over a set of base predicates and a function for evaluating those expressions. The code is generic over the set of base predicates, so the subject of these expressions could be anything from integer inequalities to the settings of compiler flags.

The sum type expr is indicated by the pipes separating the different arms of the declaration. Some of those arms, such as True and False, are simple tags, not materially different from the elements of an enumeration in Java or C. Others, such as And and Not, have associated data, and that data varies between the cases. This type actually contains both sums and products, with the And and Or branches containing tuples. Types consisting of layered combinations of products and sums are a common and powerful idiom in OCaml.

One notable bit of syntax is the type variable 'a. A type variable can be instantiated with any type, and this is what allows the code to be generic over the set of base predicates. This is similar to how generic types are handled in Java or C#. Thus, Java's <A>List would be rendered as 'a list in OCaml.

The function eval takes two arguments: expr, the expression to be evaluated; and eval_base, a function for evaluating base predicates. The code is generic in the sense that eval could be used for

FIGURE 1. EXPRESSION TYPE AND EVALUATOR IN OCAML

```
type 'a expr =  | True
                | False
                | And  of 'a expr * 'a expr
                | Or   of 'a expr * 'a expr
                | Not  of 'a expr
                | Base of 'a

let rec eval eval_base expr =
   let eval' x = eval eval_base x in
   match expr with
   | True  -> true
   | False -> false
   | Base base -> eval_base base
   | And (x,y) -> eval' x && eval' y
   | Or (x,y)  -> eval' x || eval' y
   | Not x     -> not (eval' x)
```

expressions over any type of base predicate, but eval_base must be provided in order to evaluate the truth or falsehood of those base predicates. The function eval' is defined as shorthand for invoking recursive calls to eval with eval_base as an argument. Finally, the match statement is used for doing a case analysis of the possible structures of the expression, calling out to eval_base when evaluating a base predicate, and otherwise acting as a straightforward recursion over the structure of the datatypes.

Figure 2 shows how the same code might be rendered in Java. The verbosity is immediately striking. Adding a single case such as And takes two (short) lines in OCaml and eight in Java—and the Java code is actually pretty minimal as these things go. If you want to allow the creation of other algorithms around this expression type that are not baked into the class definition, then you probably want to use the visitor pattern, which will inflate the line count considerably.

Another facet of the language that demands some further explanation is the ability of the type system to catch bugs. People who are not familiar with OCaml and related languages (and some who are) often make the mistake of underestimating the power of the type system. It's easy to conclude that all that the type system does for you is ensure that you passed in your parameters correctly (e.g., that you provided a float where you were supposed to provide a float).

But there's more to it than that. Even naive use of the type system is eerily good at catching bugs. Consider the following Python code for destuttering a list (i.e., removing sequential duplicates).

```
# Removes sequential duplicates, e.g.,
# destutter([1,1,4,3,3,2]) = [1,4,3,2]

def destutter(list):
    l = []
    for i in range(len(list)):
        if list[i] != list[i+1]:
            l.append(list[i])
    return l
```

FIGURE 2. EXPRESSION TYPE AND EVALUATOR IN JAVA

```
public abstract class Expr<T> {

  public interface Evaluator<T> { boolean evaluate(T value); }
  public abstract boolean eval(Evaluator<T> evaluator);

  public class True<T> extends Expr<T> {
    public boolean eval(Evaluator<T> evaluator) { return true; }
  }
  public class False<T> extends Expr<T> {
    public boolean eval(Evaluator<T> evaluator) { return false; }
  }
  public class Base<T> extends Expr<T> {
    public final T value;
    public Base(T value) { this.value = value; }
    public boolean eval(Evaluator<T> evaluator)
    { return evaluator.evaluate(value); }
  }
  public class And<T> extends Expr<T> {
    public final Expr<T> expr1;
    public final Expr<T> expr2;
    public And(Expr<T> expr1, Expr<T> expr2) {
      this.expr1 = expr1;
      this.expr2 = expr2;
    }
    public boolean eval(Evaluator<T> evaluator) {
      return expr1.eval(evaluator) && expr2.eval(evaluator);
    }
  }
  public class Or<T> extends Expr<T> {
    public final Expr<T> expr1;
    public final Expr<T> expr2;
    public Or(Expr<T> expr1, Expr<T> expr2) {
      this.expr1 = expr1;
      this.expr2 = expr2;
    }
    public boolean eval(Evaluator<T> evaluator) {
      return expr1.eval(evaluator) || expr2.eval(evaluator);
    }
  }
  public class Not<T> extends Expr<T> {
    public final Expr<T> expr;
    public Not(Expr<T> expr) { this.expr = expr; }
    public boolean eval(Evaluator<T> evaluator)
    { return !expr.eval(evaluator); }
  }
}
```

This code looks pretty straightforward, but it has a bug: it doesn't properly handle the end of the list. Here's one way of fixing it:

```
def destutter(list):
    l = []
    for i in range(len(list)):
        if i + 1 >= len(list) or list[i] != list[i+1]:
            l.append(list[i])
    return l
```

Now let's see what happens when writing more or less the same function in OCaml, with more or less the same bug:

```
let rec destutter l =
  match l with
  | []              -> []
  | x :: y :: rest ->
    if x = y then destutter (y :: rest)
    else x :: destutter (y :: rest)
```

This uses OCaml's pattern-matching syntax to get access to the elements of the list. Here :: is the list constructor, and `[]` indicates an empty list. Thus, the **[]** case matches the empty list, and the **x::y::rest** case matches lists that have at least two elements, **x** and **y**. The variable **rest** refers to the (potentially empty) remainder of the list.

Like the Python example, this code fails to contemplate what happens when you get to the end of the list and have only one element left. In this case, however, you find out about the problem not at runtime but at compile time. The compiler gives the following error:

```
File "destutter.ml", line 2, characters 2-125:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
_::[]
```

The missing case, **_::[]**, is a list with a single element.

You can fix the code (and satisfy the compiler) by adding a handler for the missing case:

```
let rec destutter l =
  match l with
  | []              -> []
  | x :: []         -> x :: []
  | x :: y :: rest ->
     if x = y then destutter (y :: rest)
     else x :: destutter (y :: rest)
```

The error here is a trivial one that would be found easily by testing. But the type system does just as well in exposing errors that are hard to test, either because they show up only in odd corner cases

that are easy to miss in testing, or because they show up in complex systems that are hard to mock up and exercise exhaustively.

Straight out of the box, OCaml is pretty good at catching bugs, but it can do even more if you design your types carefully. Consider as an example the following types for representing the state of a network connection:

```
type connection_state =
| Connecting
| Connected
| Disconnected

type connection_info = {
  state:              connection_state;
  server:             inet_addr;
  last_ping_time:     time   option;
  last_ping_id:       int    option;
  session_id:         string option;
  when_initiated:     time   option;
  when_disconnected: time   option;
}
```

The connection_state type is a simple enumeration of three named states that the connection can be in; connection_info is a record type containing a number of fields describing different aspects of a connection. Note that the fields that have option at the end of the type are essentially nullable fields. (By default, values in OCaml are guaranteed to be non-null). Other than that, there's nothing about this code that's all that different from what you might write in Java or C#.

Here is some information on the individual record fields and how they relate to each other:

- server indicates the identity of the server on the other side of the connection.
- last_ping_time and last_ping_id are intended to be used as part of a keep-alive protocol. Note that either both of those fields should be present, or neither of them should. Also, they should be present only when state is Connected.
- The session_id is a unique identifier that is chosen afresh every time the connection is reestablished. It also should be present only when state is Connected.
- when_initiated is for keeping track of when the attempt to start the connection began, which can be used to determine when the attempt to connect should be abandoned. This should be present only when state is Connecting.
- when_disconnected keeps track of when the connection entered the Disconnected state, and should be present only in that state.

As you can see, a number of invariants tie the different record fields together. Maintaining such invariants takes real work. You need to document them carefully so you don't trip over them later; you need to write tests to verify the invariants; and you must exercise continuing caution not to break the invariants as the code evolves.

But we can do better. Consider the following rewrite:

```
type connecting   = { when_initiated: time; }
type connected    = { last_ping : (time * int) option;
                                  session_id: string; }
type disconnected = { when_disconnected: time; }

type connection_state =
| Connecting   of connecting
| Connected    of connected
| Disconnected of disconnected

type connection_info = {
   state:  connection_state;
   server: inet_addr;
}
```

We now have a combination of product and sum types that more precisely represents the set of allowable states of a connection. In particular, there is a different record type for each of the three states, each containing the information that is relevant just to that state. Information that is always relevant (in this case, just the server) is pushed to the top-level record. Also, we've made it explicit that last_ping_time and last_ping_id are either both present or both absent by representing them as last_ping, which is an optional pair.

By doing all of this, we've embedded into the type many of the required invariants. Now that the invariants are part of the types, the compiler can detect and reject code that would violate these invariants. This is both less work and more reliable than maintaining such invariants by hand.

The example uses algebraic datatypes to encode invariants, but OCaml has other tools for doing the same. OCaml's module system is one example, allowing you to specify invariants in the interface of a module. Unlike most object-oriented languages, OCaml makes it possible to express complex joint invariants over multiple different types. More generally, OCaml's modules are a powerful tool for breaking down a codebase into small, understandable pieces, where the interactions between those pieces is under the programmer's explicit control.

The type system's ability to catch bugs is valuable even for small solitary projects, but it truly shines in a collaborative environment where multiple developers work together on a long-lived codebase. In addition to finding bugs, type signatures play a surprisingly valuable role as a kind of guaranteed-to-be-correct documentation. In the context of an evolving codebase, invariants enforced by the type system have the benefit of being more durable than those enforced by convention, in that they are less likely to be broken accidentally by another developer.

LIMITATIONS

None of this is to say that OCaml is without its flaws. There are, of course, all of the problems associated with being a minority language. OCaml has a great community that has generated a rich set of libraries, but that collection of libraries pales in comparison with what's available for Python, C, or Java. Similarly, development tools such as IDEs, profilers, and debuggers are there, but are

considerably less mature and featureful than their cousins in more mainstream languages.

Another limitation of OCaml has to do with parallelism. The OCaml runtime has a single runtime lock, which means that one must use multiple processes to take advantage of multiple cores on a single machine. For the most part, this fits our development model well: we prefer message passing to shared-memory threads as a programming model for parallelism, since it leads to code that is easier to reason about and it scales better to systems that cross multiple physical machines. The tools available in the wider OCaml world for doing this kind of multiprocess programming, however, are still maturing.

But OCaml's limitations are not fundamental in nature. They have more to do with the details of the implementation or the popularity of the language and not with the language itself. In the end, that's what I find most puzzling. I am now quite convinced that the core ideas behind OCaml are enormously valuable, as evidenced by the fact that OCaml itself, whatever its limitations, is a profoundly effective and powerful tool. Yet, those ideas remain stubbornly outside of the mainstream.

Perhaps this is finally on the verge of changing. Languages such as F# and Scala are bringing some of the ideas behind OCaml and Haskell to a wider audience by integrating themselves within the Dotnet and Java ecosystems, respectively. Maybe 10 years from now, we'll no longer need to ask why these ideas have failed to catch on in the wider world. But there's no reason to wait. You can add OCaml to your toolbox now.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

YARON MINSKY obtained his Ph.D. in computer science from Cornell University in 2002, focusing on distributed systems. In 2003, he joined Jane Street where he founded the quantitative research group, and since 2007 he has managed the technology group there.