FROM SML TO OCAML

Brandon Wu

February 5th, 2024

Lesson Plan

1 - Overview

Language Learning

Learning a programming language is usually an arduous task, requiring understanding of a brand new toolbox of tools and tricks to program with.

One might struggle for quite a while to even conceptualize the basics of a new programming language, particularly one with a novel structure.

Luckily, as students of 15-150 at Carnegie Mellon University, you already know this one. You just haven’t realized it yet.

Languages by Comparison

λ

Standard ML is the language of choice taught in Carnegie Mellon's 15-150 course. It is a formally specified, mostly **pure** language which adheres to a functional style, and supports algebraic datatypes, full type inference, and a powerful module system.

OCaml (short for **O**bjective **CAML**[1]) is an industrial-strength language used in static analysis, compilers, proof assistants, and at least one quantitative trading firm. It is a somewhat[2] specified, mostly **pure** language which adheres to a functional style, and supports algebraic datatypes, full type inference[3], and a powerful module system.

Go figure.

[1]Which itself stands for "Categorical Abstract Machine Language". Funnily enough, this means that this ML does not mean the same thing as SML's ML, which stands for "Meta Language".

[2]This is another way of saying "not".

[3]Pedantry requires I insert an asterisk here. I refuse to elaborate on why, though.

Why OCaml?

There are some things we need to clear up before proceeding, chief of which is why it is valuable to learn OCaml in the first place. If you've seen my prior lectures[4], you know that I am a firm believer that it is the concepts behind the language that matter most. Why learn a similar language?

The truth is that Standard ML is a more academically oriented language, and while a language should be able to be measured solely by its own virtues, in practice that is not how programming languages work. There are considerations of ecosystem, culture, and community, and OCaml has far better **tooling** and developer friendliness.

In short, 15-150 was to teach you how to program better. Now, this lecture is to teach you how to program better, better.[5]

[4] Graciously hosted at https://brandonspark.github.io/150/

[5] Once you finish reading this lecture slide, you can even go so far as to put OCaml under your list of programming languages on your resume. I believe in you.

Why OCaml?

Otherwise, the reasons for learning OCaml are going to be very similar to the reasons for why you should learn Standard ML.

For scripting and small experiments, dynamically typed languages can be perfectly fine. When maintaining and contributing to a project which will last for years and years, the health and overall tech debt of the project will matter far, far more than whether or not you could get out a feature out a day or two faster (and possibly with bugs).

OCaml, like many other functional languages, is **safe**, first and foremost. It is incredibly important that you learn to use a language which prevents you from shooting yourself in the foot. I'm here to translate your skills from 150 into the real world[6].

[6]Not to be confused with the real world.

OCaml Fun Facts

I can’t resist waxing on about some OCaml fun facts, so I’ll give you some bullet points.

- OCaml is a French language. As in, the French pretty much invented it.[7]
- OCaml was the original implementation language of the Rust compiler. That means that by learning OCaml, you’re one step ahead of the zeitgeist.
- OCaml is used by a few formal methods/static analysis projects, one of them being my workplace, Semgrep! This means that all the advice I am giving you is as a professional[8] OCaml programmer.
- Once, Facebook literally reskinned OCaml and presented it to the world as "ReasonML", a definitely not scary language for web developers, point right point left emojis. It worked distressingly well.

[7] I’m not joking.

[8] As in, they pay me money. Too much, actually.

The Road Ahead

The hope is that at the end of this lecture, you will be not only approximately as proficient in OCaml as you were in Standard ML, but you will be aware of all of the cool new things that OCaml adds, that can make your programming experience a lot more convenient.

If you ever want to write a useful, programming language-based personal project, such as a compiler, pretty printer, formatter, debugger, or static analyzer, I believe that there is no better language than OCaml. We'll also talk about where it stands with respect to some other languages.

2 - Concrete Syntax

Declarations

First off, let's start with declarations.

Basic top-level declarations in SML and OCaml look quite similar, the only difference being the usage of the `let` keyword instead of `val`.

```sml
val x = (2, 3 + 5)
```

SML

```ocaml
let x = (2, 3 + 5)
```

OCaml

We also replace the `fun` keyword here, in OCaml. Instead, we use the keywords `let rec`, which signifies a `let` binding which is recursive.

```sml
fun f x y = f x (y - 1)
```

SML

```ocaml
let rec f x y = f x (y - 1)
```

OCaml

Incredible.

Nested Declarations

SML and OCaml both have `let` bindings, but the SML one allows for many declarations at once, whereas an OCaml let binding only allows a single declaration. They have the form of:

```
let
    ⟨declarations⟩
in
    ⟨expression⟩
end
```

SML

```
let ⟨pattern⟩ = ⟨expression⟩ in
⟨expression⟩
```

OCaml

Note that the OCaml version does not have an `end`.

Nested Declarations

These do not look so different in the case of a single local declaration, but let's look at an example with more than one:

```
let
  val x = 1
  val y = 2
in
  x + y
end
```

SML

```
let x = 1 in
let y = 2 in
x + y
```

OCaml

In OCaml, it looks almost as if the `in` serves like a semicolon in other languages, which delimits the `let` binding from the rest of the expression.

Lists and Tuples

Lists and tuples also look a little bit different.

Lists delimit their items with semicolons instead of commas:

```
val x = [1, 2, 3]
```
SML

```
let x = [1; 2; 3]
```
OCaml

Tuples also do not *always* require parentheses around them, as they do in SML. This means that lists of tuples can look quite strange, in OCaml:

```
val x = [(1, 2), (3, 4)]
```
SML

```
let x = [1, 2; 3, 4]
```
OCaml

Pattern Matching

Pattern matching also has an alternative syntax.

```
case x of
  0 => 1
| _ => 2
```

SML

```
match x with
| 0 -> 1
| _ -> 2
```

OCaml

Not only have a few words and symbols been swapped around, but you can now put a bar before the very first case in a `match` expression.

Feature OCaml also has first-class syntax for "when clauses", which combine pattern matching with conditionals. So for instance:

```
case x of
   [] => 1
| x::xs =>
     if f x then 2 else 3
```

SML

```
match x with
| [] -> 1
| x::xs when f x -> 2
| x::xs -> 3
```

OCaml

Functions

OCaml does not have a notion of function clauses. Functions which case upon their arguments are usually written as explicitly naming their arguments, then cased upon with a `match` expression.

```
fun fact 0 = 1
  | fact n = n * fact (n - 1)
```

SML

```
let rec fact n =
  match n with
  | 0 -> 1
  | n -> n * fact (n - 1)
```

OCaml

Note that it is possible to write an OCaml function without the `rec`! Thus, it is possible to write a function with just a single `let`, meaning `let rec` is not a perfect analogue to `fun`. For instance, we could write:

```
let f x y = x + y
```

Lambdas

For lambdas, we actually get more functionality out of OCaml.

Feature Whereas SML has a single kind of lambda expression (or `fn` expression), OCaml has two! One uses the `fun` keyword[9], and the other uses the slighter longer `function` keyword. For instance, the following expressions are exactly the same:

```
fun x -> x + 1
```
SML

```
function x -> x + 1
```
OCaml

[9]Confusingly.

OCaml: Kinds of Lambdas

They have different strengths, however! A `fun` lambda expression allows shorthand for multiple curried arguments, and a `function` lambda expression allows pattern matching. For instance:

```
let f : int -> int -> int -> int =
  fun a b c -> a + b + c
```

```
let rec fact = function
  | 0 -> 1
  | n -> n * fact (n - 1)
```

It is idiomatic that functions which case upon their last arguments will use a `function` expression, instead of naming the argument explicitly!

Datatypes

In OCaml, the `type` keyword is overloaded for both declaring a datatype and declaring a type alias.

```
datatype tree =
    Empty
  | Node of tree * int * tree
```
SML

```
type tree =
  | Empty
  | Node of tree * int * tree
```
OCaml

However, the following code is valid in both languages:

```
type t = int
```

Remark It's also worth noting that constructors in OCaml *must* begin with a capital letter. Correspondingly, identifiers must begin with a lowercase letter[10].

[10]In practice, this actually turns out to be remarkably useful when reading OCaml code, as it lets you easily differentiate them.

Small Remarks

And finally, here are some miscellaneous notes:

- Values of the `'a option` type are now `None` and `Some e`, rather than `NONE` and `SOME e`.
- Tuples evaluate right to left, rather than left to right. This will almost never matter, but the more you know.
- OCaml has `float`s instead of `real`s.
- `+` and other arithmetic operators are not overloaded to work on both `int`s and `real`s (`float`s)
- Constructors are not truly identifiers, and can't be passed in as functions. For instance, you cannot write `map Some [1; 2]`, it would need to be `map (fun x -> Some x) [1; 2]`

3 - Augmented Arguments

Extra Power

Now, we will talk about features that OCaml has that are *not* present in Standard ML, that provide a direct level of power and expressiveness to the language.

Consider the `foldl` function.

```
fun foldl f z [] = []
  | foldl f z (x::xs) =
      foldl f (f (x, z)) xs
```

The Order of Arguments

One very common point of friction with using the `foldl` function is remembering the order of the arguments. Is it the accumulating function first, the accumulator, or the list?

This gets annoying fast, and can also produce more unwieldy code, especially if you're trying to use pipes. For instance, you cannot pipe an expression into `foldl` as the accumulator.

A nice feature from other languages is the idea of **named arguments**, which permit passing in arguments by an explicit name, rather than positionally. This is a feature that OCaml supports.

Named Arguments

So, in OCaml, we could instead write the `foldl` function like this:[11]

```
let rec foldl ~f ~acc l =
  match l with
  | [] -> []
  | x::xs ->
      foldl ~f:f ~acc:(f (x, acc)) xs
```

The tildes denote that the arguments are not positional arguments that are then bound to the names `f` or `acc`, but named arguments with the names `f` and `acc`.

Note that we use the syntax of `~argname:expr` to denote that we are passing in the expression `expr` as the argument with name `argname`.

[11]The `~f:f` written above can actually be shortened to just `~f`, which implicitly is the same. It just uses the existing binding of `f`, instead of explicitly having to name it twice.

Named Arguments: Usage

What's the point? It means that now, when using the `foldl` function, we do not need to remember which argument comes first, but instead just the names of the arguments that it takes in.

```
foldl ~acc:0 ~f:(fun (x, acc) -> x + acc) [1; 2]
```

In fact, we could also put the named arguments behind the list, if we wanted. This grants us a great deal of flexibility when it comes to structuring our code.

When working with functions with many arguments, or functions whose call-sites are given undescriptive arguments, named arguments can be very helpful for writing clearer code.

Having Patients

Consider the problem of trying to create records of patients in a hospital.

```
type patient =
  string (* name *)
  * int  (* patient id *)
  * string option (* insurance *)
```

To that end, we might want a function which can construct values of this type, in case we ever modify it in the future.

```
fun mk_patient name insurance_opt =
  (name, new_id (), insurance_opt)
```

(assuming that we had a `new_id : unit -> int` that just used and updated a global ref of patient IDs)

Having Patients

But, this can be inconvenient to use! This means that any time we have a patient whose insurance is unknown, we will need to explicitly pass in a `NONE`:

```
mk_patient name NONE
```

Even worse, sometimes patients do not have names – for instance, when an unknown person is treated. So the call-site for the function would look like:

```
mk_patient "John Doe" NONE
```

Optional Arguments

This is kind of gross, especially if we wanted to change the default, or change what arguments the function takes in. Put in another way, the information having to do with the function's defaults is distributed across all of its call-sites, instead of at the function itself.

Fortunately, OCaml has **optional arguments**, which allow functions to specify arguments that do not *have* to be given to the function.

So instead, we could write[12]:

```
fun mk_patient ?(name = "John Doe") ?(insurance = None) () =
  (name, new_id (), insurance)
```

[12] The extra `()` argument is because optional arguments must be **erasable**, meaning that it must be clear which arguments a partial application is treating as optional. This makes it explicitly so optional args must be given before the unit.

Optional Arguments: Usage

This ends up being far more convenient, because this means that optional arguments can be omitted at their call-sites, if the information is not available.

Thus, we could instead have our call-sites as, in the case where we have no insurance, and no name and no insurance, respectively:

```
mk_patient name ()
```

```
mk_patient ()
```

4 - Records

On Tuples and Product Types

The next most salient way that OCaml diverges from SML is in its treatment of records. First, we will require an introduction to records, which is not a topic that is often covered in 15-150.

In Standard ML and OCaml, we can use tuples, or **product types**, to organize multiple values into one. We see that the type of a tuple is merely a summary of the constituent types it contains – a tuple of two integers is simply given the self-explanatory type `int * int`.

This can become quite inconvenient. For instance, take the type signature of a function which computes the number of days between two dates:

```
daysBetweenDates : int * int * int -> int * int * int -> int
```

An Ambiguous Type

This is clearly an incredibly undescriptive type. There are seven instances of integers, and depending on where the user originates from, it might be unclear which integer denotes the month, day, or year!

For instance, the American convention is MM/DD/YYYY, but the European convention is DD/MM/YYYY. That can make using this API confusing at best, and dangerous at worst.

So, what can we do? The problem is that tuples are not descriptive as to what each component of the tuple means. Records will solve this issue.

SML: Record Types

Def A **record** is a value that contains multiple values, like a tuple, but each value corresponds to a name, called a **field** .

So for instance, we might define a `date` type as a record, which gives specific names to each of its fields:

```
type date = {
  year  : int,
  month : int,
  date  : int
}
```

SML

We can simply construct a record using similar notation, with curly braces:

```
val new_year : date = { year: 2024, month: 1, date: 1 }
```

OCaml: Record Types

In OCaml, the notation is similar, but with the semicolons for delimiters that OCaml favors:

```
type date = {
  year  : int;
  month : int;
  date  : int
}
```

OCaml

With values being constructed analogously:

```
let new_year : date = { year = 2024; month = 1; date = 1 }
```

Anonymous and Declared Records

Given these small changes, it may not seem like OCaml records deserve special note. The main difference between SML and OCaml records is that SML records are **anonymous**, whereas OCaml records must be declared.

By anonymous, I mean that SML records are anonymous in the same way as tuples. For example, the tuple type `int * int` is anonymous, as it doesn't need to be explicitly given a name, to be a type that can be used. It just exists.

Thus, with no prior type declarations, only the following SML code compiles, and infers the type of `x` to be `{a : int, b : string}`:

```sml
val x = {a = 1, b = "hi"}
```

SML

```ocaml
let x = {a = 1; b = "hi"}
```

OCaml

OCaml: Records and Type Inference

In the presence of a type declaration, OCaml will be able to infer the type of the record safely. In the below example, the type of `x` is correctly inferred to be `t`:

```
type t = {a : int, b : string}
let x = {a = 1; b = "hi"}
```

OCaml

This means that, in order for OCaml records to be used effectively, it must always be clear which type that a given record should be inferred to.

Remark In the above example, since there was a type with field `a` in scope, `x` was inferred correctly to be of that type, with no annotations. If `t` were instead in a separate module, however, we would need to help out the type system a bit and tell it what module the record's type is from.

SML: Record Access

In SML, record fields must be accessed using either the record access operator, or by pattern matching. For instance, the following are both legal ways to extract out the field `a`:

```sml
val x = {a = 1, b = "hi"}
val 1 = #a x
val {a = num, ...} = x
```

SML

The second line conjures a function `#a`, which is inferred to have type `{a : int, b : string} -> int`, in this context.

The third line deconstructs the record `x`, binding its field `a` to the name `num`. It then uses an ellipsis to denote that it wildcards out the rest of the fields of the record.

OCaml: Record Access

λ

In OCaml, this looks slightly different. The record access operator is instead achieved by taking the record, and using a dot operator, followed by the name of the field to be accessed[13]. So this is translated to:

```
type t = { a : int , b : string }
let x = {a = 1; b = "hi"}
let 1 = x.a
let {a = num; _} = x
```

OCaml

In addition, instead of an ellipsis, we use the wildcard symbol _ to denote that we don't care about the rest of the fields.

[13]This seems small, but this is actually an incredibly convenient quality of life change for writing OCaml code. A small change like making record access postfix makes it so much easier to use.

Functional Record Updates

Feature OCaml also boasts the ability to create a copy of an existing record, with one or more fields changed. This is called a **functional record update**.

For instance, suppose we have the following record:

```
type t = {a : int, b : string, c : bool}
let x = {a = 1; b = "hi"; c = true}
```

If we wanted to produce a copy of `x` with only the field `a` changed, we could use the following terse syntax:

```
let y = {x with a = 2}
```

OCaml

Functional Record Updates

In SML, you would have no choice but to state every field of the record again:

```
val y = {a = 2, b = #b y, c = #c y}
```

SML

This quickly becomes hugely untenable, especially for records which have many fields.[14]

[14] This is one reason for why, as much as I love SML, I avoid records in many cases, because the convenience of using it is so low. In OCaml, records are very easy to use.

Records in Review

λ

In summary, records are an enhancement of tuples that equips them with named fields. They are extremely useful, especially for writing descriptive code which remains clear to future maintainers, so it is advisable to be familiar with them.

In OCaml, they are much more convenient to use, in many respects, which partly contributes to why OCaml feels better to use, as a developer.

5 - Metaprogramming

Programming Languages and Metaprogramming

λ

Def **Metaprogramming** is the technique of writing programs which themselves modify programs. In programming languages, it usually refers to the practice of being able to generate code from within the language itself.

There are many ways to metaprogram, including methods external to the language itself. For instance, you could keep a script that prepends all relevant files with certain frequently-used declarations, and call that a kind of metaprogramming.[15]

For an example of how metaprogramming is useful, consider the simple problem of trying to print out a list.

In SML, this is a non-trivial problem, because there is the `print` function, but it has type `string -> unit`. This means that, somehow, to print an `int list`, you must be able to convert an `int list` into a `string`.

[15]The C language's famous preprocessor can be used to do this, for instance.

Printing Lists

The code for printing out an `int list` is not so involved.

```
fun show_int_list l = "[" ^ aux l ^ "]"
and aux [] = ""
  | aux [x] = Int.toString x
  | aux (x::xs) = Int.toString x ^ ", " ^ aux xs
```

In fact, we can even HOF-ize this, to print out arbitrary `'a list`s:

```
fun show_list f l = "[" ^ aux f l ^ "]"
and aux f [] = ""
  | aux f [x] = f x
  | aux f (x::xs) = f x ^ ", " ^ aux f xs
```

Printing Trees

This code was annoying to write, though. It's nothing more than boilerplate, simple code which just wastes time to write. It's not even in the standard library.

What if we want to print out a tree? Well, now we need to write a new function:

```
fun show_tree f Empty = "Empty"
  | show_tree f (Node (L, x, R)) =
      "Node("
        ^ show_tree f L
        ^ ", "
        ^ f x
        ^ ", "
        ^ show_tree f R
        ^ ")"
```

This function is just plain ugly.

An Excess of Boilerplate

Now imagine doing this for every single type that you declare, because none of them come with their own printing functions.

A similar problem comes with if you want to declare a type that is used as a key in some kind of dictionary.

There are two ways of doing this – you can either hash the key values and use a hash dictionary, or come up with a total ordering function for the type, and use a search tree of some kind. Both of these ways are rather involved, and involve an immense amount of boilerplate.

Type-Directed Code Generation

The key insight into solving the previously described issues is to see that it comes right out of the type.

```
datatype tree =
  Empty
| Node of tree * int * tree
```

Just by looking at the type definition of a `tree`, it is very obvious how to print it – case on the constructors, and recurse on the components until you reach the end.

Thus, a sophisticated enough program should be able to read the type definition of a `tree`, and generate the corresponding code, a function of type `tree -> string` which pretty-prints a value of type `tree`.

This is precisely what `ppx` does. Let's see how it works.

ppx and Printing

OCaml's ppx is a preprocessor framework, which allows for preprocessor libraries to be written, which each implement a different kind of code generation. In this case, we are interested in ppx_deriving, which includes the show plugin.

```ocaml
type tree =
  Empty
| Node of tree * int * tree [@@deriving show]
```

OCaml

The source text only needs an [@@deriving show] annotation to be added to the type definition, which then tells the OCaml compiler to invoke the plugin[16]. The plugin will then generate two functions, named show_tree and tree_show, both of which of type tree -> string.

It's just that easy.

[16] This also requires that the file being compiled is actively built with the ppx_deriving.show preprocessor, or the annotation will mean nothing.

More on ppx

λ

If this type definition were in a module ascribing to a given signature, you would also attach the `[@@deriving show]` annotation to the signature, as well.

```
type tree [@@deriving show]
```

OCaml

In this case, suppose that the signature were to leave the `tree` type abstract. Here, we are just saying that because we have derived the `show` function, the signature also exposes the `show_tree` and `tree_show` functions.

It's worth noting that deriving dependencies are transitive. If you are deriving `show` on a type which uses another type, it will rely on having previously derived `show` for the used type, as well.

Deriving Deeper

We can do the same for a few other plugins, as well.

```ocaml
type t = A | B of int | C of string * int list
  [@@deriving show, ord, hash]
```

OCaml

This demonstrates an arbitrary type `t`, which derives `show` for pretty-printing, but also `ord`, which produces a function `compare` [17], which is a total comparison function on values of type `t`. This makes it suitable for usage for using values of type `t` as keys into a binary search tree, for instance.

We also derive `hash`, which derives a hash function for values of type `t`. This is useful if we wanted to use these values as keys into a hash table.

[17] It's `compare` and not `compare_t`, because this is how it is hard-coded to behave on types named `t`.

Conclusion

λ

`ppx` is a very powerful framework for writing cleaner, simpler code through preprocessor libraries, and in the case of `ppx_deriving`, it is an incredibly useful labor-saving device.

Other languages like Rust and Haskell have similar mechanisms for deriving traits and typeclasses, respectively, in a similar way to `ppx_deriving`.

You can also use `ppx` to write custom preprocessing libraries, so the sky is the limit. This ends up providing a great deal of power in writing more expressive programs.

6 - Binding Operators

An Introduction to Monads

Functional languages always end up making a big hubbub about monads [18].

Recall the signature of monads, a type class of a parametric type:

```
signature MONAD =
  sig
    type 'a t
    val return : 'a -> 'a t
    val bind : 'a t -> ('a -> 'b t) -> 'b t
  end
```

We say that a structure implementing `MONAD` is a monad if it satisfies some laws:

- `bind (return x) f` ≅ `f x`
- `bind m return` ≅ `m`
- `bind (bind m f) g` ≅ `bind m (fn x => bind (f x) g)`

[18] This is not unfair, given that monads are extremely useful. However, commonly it is made out to be far more complicated than it is.

The Option Monad

Take the simplest useful monad, which is the `option` monad. We might implement it in SML as:

```
structure OptionMonad : MONAD =
  struct
    type 'a t = 'a option

    val return = SOME
    fun bind opt f =
      case opt of
        NONE    => NONE
      | SOME x => f x
  end
```

This monad is incredibly useful when dealing with several functions which might return an `option`.

Piping Functions

For our example, let's just assume we have functions `f : t1 -> t2 option`, `g : t2 -> t3 option`, and `h : t3 -> t4 option`.

Then, we could write the following code:

```
fun pipeline (x : t1) : t4 option =
  bind (bind (bind x f) g) h
```

Or, if we define an infix function `>>=` that has type `'a t * ('a -> 'b t) -> 'b t`, essentially a tupled version of `bind`:

```
fun pipeline (x : t1) : t4 option =
  x >>= f >>= g >>= h
```

which looks much cleaner.

Explicit Naming

This is not so bad, but this doesn't give an explicit name to any intermediate result of the pipeline. This is not always what we want, since we might want to use it at a later point, for instance:

```
fun computation (file : string) (student: string) =
  parse file
  >>= (fn (_, gradesheet) => lookup student gradesheet)
  >>= (fn grades =>
    SOME ((sum grades) div (List.length grades))
  )
```

(assuming `parse : string -> (attendance * grades) option`, `lookup : string -> int list option`, `sum : int list -> int`)

Haskell and Syntactic Sugar

This is not so complicated to read, but the lambda and extra parentheses are not helping. This can quickly become hugely untenable, given a lot of `binds`.

The Haskell programming language is quite fond of monads, and has its own special syntax for using them in a neater way. The above example would instead be written as:

```
do
  (_, gradesheet) <- parse file
  grades <- lookup student gradesheet
  return ((sum grades) / (length grades))
end
```

This looks quite similar, but avoids a level of parentheses (and indentation) by moving the binding of the `grades` variable to before the computation it is binding, as opposed to after. This is a huge readability win.

OCaml: Custom Binding Operators

In OCaml, you can do something similar, by defining custom **binding operators**.

There are no `do` blocks in OCaml, but a `do` block is really just a spicy list of declarations. A binding operator will just be an enhanced `let`-binding:

```ocaml
let (let*) = OptionMonad.bind

let computeAverage (file : string) (student: string) =
  let* _, gradesheet = parse file in
  let* grades = lookup student gradesheet in
  Some (sum grades / (List.length grades))
```

OCaml

Here, we use the `let*` binding operator, which we have defined to be the same as `OptionMonad.bind`.

Binding Operator Translation

Concretely, if we say that `let*` is defined to be the same as some function `f : 'a -> ('b -> 'c) -> 'd`, then the following are equivalent:

```
let* x = e in
e2
```

and

```
f e (fun x -> e2)
```

Basically, `let*` implicitly calls the equivalent function, and implicitly put the rest of the code after the `in` into a lambda, which is passed to it.

6 - Conclusions

OCaml in Review

Overall, OCaml ends up being a language which is very similar to Standard ML, but with just a few things that make it feel much better to use.

Despite seeming inconsequential, these small improvements add up, from a developer's perspective! Two features, functional record updates and postfix record access, are really all that is necessary to make OCaml records much more usable than SML records.

Aside from language features, OCaml's ecosystem also boasts some other nice accommodations, including a debugger[19], a package manager, a testing framework, and several open-source libraries.

[19] Though there's one now for SML.

Simplicity and OCaml

There are far more differences between the two languages that I neglected to write about (polymorphic variants, first-class modules, objects and classes, variance, applicative functors, GADTs, etc...). This was intentional.

My belief is that OCaml occupies a valuable niche in programming languages where it is broad, but all of the breadth is **opt-in**. That means that, while complex language features exist, they are discouraged, and you don't ever have to deal with them if you don't want to.

I regularly say that when writing OCaml code, I only use around seven distinct language features, or forms of syntax. While naysayers believe functional code to be more complicated, it does not need to be at all. I believe that OCaml is a very easy language to write simple code in.

Languages and Complexity

There are respected languages like Haskell and Rust, which are themselves functional in nature as well. Unfortunately, they occupy a niche where they remain *complicated by default*.

In Haskell, extensive monad usage and excessive abstraction (to the point of convolution) are the norm, and Rust code requires understanding of lifetimes and memory management, making writing code more complicated. These are baseline thresholds which cannot be lowered – they are always present.

Like with mutability, we should leave complexity as an opt-in feature. Complexity is only as complexity is warranted.

More OCaml Resources

Further resources that can help with learning OCaml include:

- the 99 problems provided on the OCaml website, with solutions included.
- the book Real World OCaml, which recently came out with a new edition
- CS3110's online course materials on OCaml

I am also going to be working on a lecture series for OCaml programming, hopefully to debut in 2024. Feel free to follow me on Twitter or check my website to keep up with material as I release it.[20]

[20] I have not yet set up a mailing list.

Thank you!