

CS 320: Concepts of Programming Languages

Lecture 2: OCaml Basics

Ankush Das

Nathan Mull

Sep 5, 2024

- ▶ Please read and follow the instructions on the course repository
- ▶ Some instructions coming up for installing OCaml
- ▶ Using Windows? Having trouble with installing OCaml?
Please use WSL2
- ▶ Having trouble cloning the `cs320-fall-2024` repo using ssh?
Try the https link for cloning the repo

- ▶ *HW0 is due on Friday at 11:59pm.* Please submit and reminder that this is not graded.
- ▶ *HW1 will be released today, Thu, Sep 5; due next Thursday, Sep 12*
- ▶ Submissions are to be made on *Gradescope*
- ▶ In Gradescope, when you select “*Upload Submission*”, you can choose *GitHub* as an option.
- ▶ And then select your *cs320-fall2024-private* repository

Goals for Today's Lecture

4

- ▶ **Installing OCaml**
- ▶ **Interacting with OCaml using UTop**
- ▶ **Some basic features of OCaml**
- ▶ **Basic Types in OCaml**
- ▶ **Abstraction for Today: Let Expressions**
- ▶ **Formal Syntax, Type System, and Semantics of Let Expressions**

- ▶ We will be following the online book called *OCaml Programming: Correct + Efficient + Beautiful* by Michael R. Clarkson et al.
- ▶ Available at: <https://cs3110.github.io/textbook/cover.html>
- ▶ More details on the course webpage
- ▶ Book also contains detailed instructions for installing OCaml at: <https://cs3110.github.io/textbook/chapters/preface/install.html>
- ▶ Please go through them step by step!

Installing OCaml on MacOS (Highlights)

6

- ▶ Install a Unix package manager: [Homebrew](#) or [Macports](#)
- ▶ Install OPAM (OCaml Package Manager) using Homebrew:
`brew install opam`
- ▶ Initialize OPAM:
`opam init --bare -a -y`
`opam update`
- ▶ Install OCaml:
`cd cs320-fall-2024-private`
`opam switch create ./ 4.13.1` (creates a local switch*)
`eval $(opam env)`

*to create a global switch, use `opam switch create 4.13.1`

Interacting with OCaml

7

- ▶ There are many main ways of writing and executing OCaml programs: `utop`, `ocamlc`, `dune`, etc.
- ▶ Today, we will use `utop`
- ▶ Installing `utop` and some other standard tools and a standard library that you will use in this course:
`opam install dune utop ounit2 ocaml-lsp-server`
`opam install stdlib320/.`
- ▶ We will cover the other ways in the future (and in labs)

What is UTop?

8

- ▶ It's an interactive mode for OCaml, also called REPL (Read-Eval-Print-Loop)
- ▶ How do you run `utop`? Just run the command `utop` in your terminal and start writing OCaml code
- ▶ Anything that can be written in OCaml files can also be written in UTop and vice-versa

What to write in UTop?

9

- ▶ UTop accepts two kinds of input:
 - ▶ *Expressions*: Write OCaml functions, entire programs, UTop will evaluate and return the output
 - ▶ *Directives*: Commands that tell UTop to perform some action, e.g., load a file, exit, etc. Always prefixed by #. We will cover some directives in the lecture

- ▶ OCaml is a language of expressions. Most of the programming abstractions we will encounter are expressions. Today, we will study two main expressions: let expressions and functions.
- ▶ Like any abstraction we study, we will study 3 main aspects of it:
 - ▶ *Syntax*: “**let** <varname> = <expression>” (e.g., **let** **x** = 3 + 4)
 - ▶ *Type System*: For expression “**let** **x** = **e**”, compute the type **T** of **e** and assign type **T** to **x**
 - ▶ *Semantics*: Compute the value **v** of **e**, then assign value **v** to **x**
- ▶ Let's focus on syntax. Types and semantics will come next!

Let Expressions in UTop

11

- ▶ We can define a series of let-expressions in UTop!
- ▶ To start: we simply type “`utop`” on our terminal
- ▶ To exit: either type “`#quit;;`” or press Control-D
- ▶ We will define expressions using “`let x = e;;`” syntax in UTop.
- ▶ Recall some examples from last lecture

- ▶ Functions are also defined in OCaml using 'let-expressions'
- ▶ Why? Because there is no difference between a variable definition and function definition
- ▶ For OCaml, “`let x = 5`” and “`let f x = x + 5`” are quite similar.
- ▶ The first defines a variable `x` of type `int`.
- ▶ The second defines a variable `f` of type `int -> int`
- ▶ Let's do some more examples of functions in `utop`

- ▶ Suppose we want to define $f(x) = x^3 + x^2$
- ▶ We can define “`let f x = x*x*x + x*x`”
- ▶ But this would do redundant multiplications
- ▶ Can we define $y = x * x$ and then define “`let f x = x*y + y`”
- ▶ Yes! We can define local variables also using... let expressions!

Local Variables in OCaml

14

```
let f x =  
  let y = x * x in  
  x * y + y
```

```
let f x =  
  let y = x * x in  
  x * y + y
```

- ▶ You can define multiple local variables!

```
let f x =  
  let y = x * x in  
  let z = x * y in  
  z + y
```

2 Kinds of Let Expressions

15

- ▶ Global Definitions at the Top Level:
 - ▶ *Syntax*: “`let <varname> = <expression>`” (e.g., `let x = 3 + 4`)
- ▶ Local Definitions within a Function Body:
 - ▶ *Syntax*: “`let <varname> = <expression> in <expression>`” (e.g., `let x = 3 + 4 in x + x`)
- ▶ Both global and local definitions can be function definitions

- ▶ A powerful feature of OCaml is that it allows recursive definitions!
- ▶ To define a recursive function, we can again use the let-expression with *one small change in syntax*
- ▶ *Syntax*: “**let rec** <fname> <arglist> = <expression>”
- ▶ We need to add an extra **rec** to mean recursive
- ▶ Demo Time!
 - ▶ Define function for adding first n natural numbers.
 - ▶ Define the factorial function.

Mutually Recursive Functions

17

- ▶ Functions can also be mutually recursive, i.e., definition of **f** refers to **g** and definition of **g** refers to **f**
- ▶ Mutually recursive functions can be defined using “**and**”
- ▶ *Syntax:*
“**let rec** <fname1> <arglist1> = <expression1>
and
<fname2> <arglist2> = <expression2>
and
.....

- ▶ So far, we have looked at syntax. We know how to *write* programs, but we don't know if they are *valid* and we don't know how they *evaluate*
- ▶ Let's look at both of these intuitively first, starting with types

- ▶ *Every* variable and expression in OCaml has a type
- ▶ Types define how expressions can be constructed and how variables can be used
- ▶ For e.g. the following expression is **not** valid

```
let x = true in
let y = x + 1 in
y
```
- ▶ **x** has type **bool** while it is used as an integer.
- ▶ Try this in **utop** to confirm the type error

Primitive Types in OCaml

20

Type	Values	Operators
int	2, 3, -101	+, -, *, /, mod
float	3., -1.01	+. , -. , *. , /.
bool	true, false	&&, , not
char	'b', 'c'	
string	"word", "@#\$#"	^
unit	()	

- ▶ For integer and floating-point arithmetic, there is a different set of operators:
- ▶ Integers: $+$, $-$, $*$, $/$, `mod`
- ▶ Floats: $+.$, $-.$, $*.$, $/.$
- ▶ There is *no operator overloading*
- ▶ To compare two expressions of the same type, we use the standard $>$, $>=$, $<$, $<=$ operators
- ▶ Equality is checked using $=$ operator (and not $==$). Inequality is checked using $<>$ operator (and not $!=$)

Function Types

22

- ▶ Function types are expressed as $A_1 \rightarrow A_2 \rightarrow \dots \rightarrow A_n \rightarrow B$ where A_1, A_2, \dots, A_n are argument types and B is the result type
- ▶ For example, `let f x y z = x * y + z`
- ▶ The type of `f` is `int -> int -> int -> int`
- ▶ Why not `float -> float -> float -> float`?
- ▶ Because of the operators `*` and `+`. They only apply to integers

- ▶ OCaml has a great type inference algorithm! So, you (almost) never have to specify types of any variables. They are automatically computed.
- ▶ But it's still important to understand how the type system works
- ▶ Let's start with the let-expression: `let x = e1 in e2`
- ▶ Intuitively, this is how the type system works.
 - ▶ First, infer the type of `e1`, say it is τ
 - ▶ Second, assign the type τ to `x` and then infer the type of `e2`
- ▶ This is still just words though. How can we make this formal?

Type System (Formally)

24

- ▶ First step: introduce a typing judgment for expressions
- ▶ For (the subset of) OCaml, the judgment is written as $\Gamma \vdash e : \tau$
- ▶ Γ is just a set of variables with their types
(e.g. $\Gamma = \{x : \text{int}, y : \text{float}, z : \text{bool}\}$)
- ▶ Another name for Γ : *Context*
- ▶ Γ denotes the variables in *scope*
- ▶ Meaning: Expression e has type τ in the presence of context Γ

Typing Let Expressions Formally

- ▶ To type `let x = e1 in e2`, in the context Γ , we first type `e1` in the context Γ
- ▶ Suppose the type of `e1` in the context Γ is τ
- ▶ This can be written as $\Gamma \vdash e_1 : \tau$
- ▶ Then, we add `x : τ` to Γ , and type `e2` in the context $\Gamma, x : \tau$
- ▶ Suppose the type of `e2` in the context $\Gamma, x : \tau$ is τ'
- ▶ This can be written as $\Gamma, x : \tau \vdash e_2 : \tau'$

Let's Do An Example (Addition)

- ▶ Expression: $\text{let } x = 3 \text{ in let } y = 4 \text{ in } x + y$
- ▶ After processing ' $\text{let } x = 3$ ', $\Gamma = \{x : \text{int}\}$
- ▶ After processing ' $\text{let } y = 4$ ', $\Gamma = \{x : \text{int}, y : \text{int}\}$
- ▶ With this context Γ , what is the type of $x + y$?
- ▶ Intuitively, $\{x : \text{int}, y : \text{int}\} \vdash x + y : \text{int}$
- ▶ Hence, $\text{let } x = 3 \text{ in let } y = 4 \text{ in } x + y : \text{int}$

Let's Make This Fully Formal

27

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau \vdash e_2 : \tau'}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau'}$$

- ▶ This is the formal typing rule for let expression
- ▶ What is this fraction-like thing and what are things above it and below it?
- ▶ This is called an *inference rule*! The things above the horizontal line are called *premises*; the thing below the line is called *conclusion*
- ▶ This inference rule is implemented inside the OCaml type inference algorithm. Soon, you'll be implementing them too!

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau \vdash e_2 : \tau'}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau'}$$

- ▶ The first premise states that e_1 has type τ in context Γ
- ▶ The second premise states that e_2 has type τ' in context $\Gamma, x : \tau$
- ▶ The conclusion states that $\text{let } x = e_1 \text{ in } e_2$ has type τ' in context $\Gamma, x : \tau$
- ▶ This rule precisely conveys the intuition from prior slides

A Few More Examples

29

$$\Gamma \vdash e_1 + e_2 :$$

A Few More Examples

29

$$\Gamma \vdash e_1 + e_2 : \text{int}$$

A Few More Examples

29

$$\Gamma \vdash e_1 : \text{int}$$

$$\Gamma \vdash e_1 + e_2 : \text{int}$$

A Few More Examples

29

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

A Few More Examples

29

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

- ▶ The first premise states that e_1 has type int in context Γ
- ▶ The second premise states that e_2 has type int in context Γ
- ▶ The conclusion states that $e_1 + e_2$ has type int in context Γ
- ▶ All integer operators will be typed the same way

Inference Rules for Integer Operators

30

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 - e_2 : \text{int}}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \times e_2 : \text{int}}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 / e_2 : \text{int}}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \bmod e_2 : \text{int}}$$

Inference Rules for Float Operators

31

$$\frac{\Gamma \vdash e_1 : \text{float} \quad \Gamma \vdash e_2 : \text{float}}{\Gamma \vdash e_1 +. e_2 : \text{float}}$$

$$\frac{\Gamma \vdash e_1 : \text{float} \quad \Gamma \vdash e_2 : \text{float}}{\Gamma \vdash e_1 -. e_2 : \text{float}}$$

$$\frac{\Gamma \vdash e_1 : \text{float} \quad \Gamma \vdash e_2 : \text{float}}{\Gamma \vdash e_1 \times. e_2 : \text{float}}$$

$$\frac{\Gamma \vdash e_1 : \text{float} \quad \Gamma \vdash e_2 : \text{float}}{\Gamma \vdash e_1 /. e_2 : \text{float}}$$

- ▶ Semantics describes how to execute programs
- ▶ In OCaml, programs are expressions. So, semantics describes how to evaluate expressions
- ▶ To understand, let's understand what evaluation means.
- ▶ Evaluation means computing the *value* of an expression
- ▶ For e.g., the value of $3 + 4$ is 7 .
- ▶ The value of $x + y$ where $x = 5$ and $y = 3$ is 8 .

Type	Values	Operators
int	2, 3, -101	+, -, *, /, mod
float	3., -1.01	+. , - . , * . , / .
bool	true, false	&&, , not
char	'b', 'c'	
string	"word", "@#\$#"	^
unit	()	

- ▶ Let's start simple. How do we evaluate $e_1 + e_2$?
- ▶ First, we evaluate e_1 . Suppose e_1 evaluates to v_1
- ▶ Next, we evaluate e_2 . Suppose e_2 evaluates to v_2
- ▶ Finally, if $v = v_1 + v_2$, then, $e_1 + e_2$ evaluates to v
- ▶ How do we write this formally?

- ▶ We introduce another semantics judgment: $e \Downarrow v$
- ▶ Meaning: Expression e evaluates to value v
- ▶ What will be the inference rule for addition?

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v = v_1 + v_2}{e_1 + e_2 \Downarrow v}$$

- ▶ How do we evaluate **let** **x** = **e**₁ **in** **e**₂?
- ▶ First, we will evaluate **e**₁. Suppose it evaluates to **v**₁
- ▶ Next, we *substitute* **v**₁ for **x** in **e**₂. And then evaluate it
- ▶ So, what will be the inference rule for let-expressions?

$$\frac{e_1 \Downarrow v_1 \quad [v_1 / x]e_2 \Downarrow v_2}{\text{let } x = e_1 \text{ in } e_2 \Downarrow v_2}$$

What is Substitution?

- ▶ Formally, substitution of v for x in e is written as $[v/x]e$
- ▶ Intuitively, it seems easy! Just replace all occurrences of x by v
- ▶ Example: $[3/x](x+x) = 3 + 3 = 6$
- ▶ Hence,
 $\text{let } x = 3 \text{ in}$
 $x + x$
 evaluates to 6
- ▶ Substitution is *one of the hardest things* to implement and understand in PL; it is very subtle and most PLs out there have made mistakes in implementing it

- ▶ See the expression:
 $\text{let } x = 4 \text{ in}$
 $(\text{let } x = 5 \text{ in } x + 2) + x$
- ▶ How should we perform substitution? What is
 $[4/x] (\text{let } x = 5 \text{ in } x + 2) + x$?
- ▶ Clearly, the last occurrence of x should be replaced by 4 , but what about the rest?
- ▶ If we just blindly replace, we get $(\text{let } 4 = 5 \text{ in } 4 + 2) + 4$, which is clearly incorrect!
- ▶ Another incorrect substitution: $(\text{let } x = 5 \text{ in } 4 + 2) + 4$

Substitution of Free Occurrences Only

39

- ▶ The actual substitution of $[4/x]$ $(\text{let } x = 5 \text{ in } x + 2) + x$ should be
- ▶ $(\text{let } x = 5 \text{ in } x + 2) + 4$
- ▶ Why? The inner occurrence of x is bound by the inner let-expression
- ▶ Next, we do another substitution: $[5/x] (x+2) + 4$
- ▶ Which comes out to $(5 + 2) + 4 = 11$
- ▶ Practice some substitutions at home! Please!

- ▶ How to write syntax formally?
- ▶ How to write type system formally?
- ▶ How to write semantics formally?
- ▶ How to interact with OCaml?
- ▶ How to define local and global variables?
- ▶ How to define local and global functions?
- ▶ Everything there's to know about let-expressions!