CS 320: Concepts of Programming Languages

Lecture 2: OCaml Basics

Ankush Das

Nathan Mull

Sep 5, 2024

Administrivia

- 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

Administrivia

- *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

- 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

OCaml Book

- 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)

- 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
  eval $(opam env)
  ```
  `opam switch create ./ 4.13.1` (creates a local switch*)

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

Interacting with OCaml

- 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?

- 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?

- 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 Let Expressions

- 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

- 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

Last Lecture's Abstraction: Functions

- 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`

Local Variables in OCaml

- 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


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

Local Variables in OCaml


```
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

- 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

Recursive 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

- 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`
  ………………

Type System and Semantics

- 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

OCaml 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

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

A Note on OCaml Operators

- 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

- Function types are expressed as `A1 -> A2 -> … An -> B` where `A1, A2, …, An` 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

Type System (Intuitively)

- 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 =` e_1 `in` e_2
- Intuitively, this is how the type system works.
  - First, infer the type of e_1, say it is τ
  - Second, assign the type τ to `x` and then infer the type of e_2
- This is still just words though. How can we make this formal?

Type System (Formally)

- 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. `Γ = {x : int, y : float, z : 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 e_1 in the context Γ
- Suppose the type of e_1 in the context Γ is τ
- This can be written as $\Gamma \vdash e_1 : \tau$
- Then, we add $x : \tau$ to Γ, and type e_2 in the context $\Gamma, x : \tau$
- Suppose the type of e_2 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: `let x = 3 in let y = 4 in x + y`
- After processing '`let x = 3`', Γ `= {x : int}`
- After processing '`let y = 4`', Γ `= {x : int, y : int}`
- With this context Γ, what is the type of `x + y`?
- Intuitively, `{x : int, y : int}` $\vdash$ `x + y : int`
- Hence, `let x = 3 in let y = 4 in x + y : int`

Let's Make This Fully Formal

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

- 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!

Inference Rule for Let

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

- 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 `let x =` e_1 `in` e_2 has type τ' in context $\Gamma, x : \tau$
- This rule precisely conveys the intuition from prior slides

A Few More Examples

$$\frac{}{\Gamma \vdash e_1 + e_2 :}$$

A Few More Examples

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

A Few More Examples

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

A Few More Examples

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}$$

A Few More Examples

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}$$

- The first premise states that $\mathtt{e}_1$ has type `int` in context Γ
- The second premise states that $\mathtt{e}_2$ has type `int` in context $\Gamma\tau$
- The conclusion states that $\mathtt{e}_1 + \mathtt{e}_2$ has type `int` in context Γ
- All integer operators will be typed the same way

Inference Rules for Integer Operators

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 - e_2 : \mathsf{int}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 \times e_2 : \mathsf{int}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 \,/\, e_2 : \mathsf{int}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 \ \mathsf{mod}\ e_2 : \mathsf{int}}$$

Inference Rules for Float Operators

$$\frac{\Gamma \vdash e_1 : \mathsf{float} \qquad \Gamma \vdash e_2 : \mathsf{float}}{\Gamma \vdash e_1 +. \, e_2 : \mathsf{int}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{float} \qquad \Gamma \vdash e_2 : \mathsf{float}}{\Gamma \vdash e_1 -. \, e_2 : \mathsf{int}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{float} \qquad \Gamma \vdash e_2 : \mathsf{float}}{\Gamma \vdash e_1 \times. \, e_2 : \mathsf{int}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{float} \qquad \Gamma \vdash e_2 : \mathsf{float}}{\Gamma \vdash e_1 \, /. \, e_2 : \mathsf{int}}$$

OCaml Semantics

- 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`.

OCaml Values

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

OCaml Semantics for Addition

- Let's start simple. How do we evaluate $\texttt{e}_1 + \texttt{e}_2$?
- First, we evaluate $\texttt{e}_1$. Suppose $\texttt{e}_1$ evaluates to $\texttt{v}_1$
- Next, we evaluate $\texttt{e}_2$. Suppose $\texttt{e}_2$ evaluates to $\texttt{v}_2$
- Finally, if $\texttt{v} = \texttt{v}_1 + \texttt{v}_2$, then, $\texttt{e}_1 + \texttt{e}_2$ evaluates to $\texttt{v}$
- How do we write this formally?

OCaml Semantics Judgment

- We introduce another semantics judgment: **e⬇v**
- Meaning: Expression **e** evaluates to value **v**
- What will be the inference rule for addition?

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

Semantics For Let Expressions

- 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?

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

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,

  ```
  let x = 3 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

One Subtle Case of Substitution

- See the expression:
  ```
  let x = 4 in
  (let x = 5 in x + 2) + x
  ```
- How should we perform substitution? What is `[4/x] (let x = 5 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 `(let 4 = 5 in 4 + 2) + 4`, which is clearly incorrect!
- Another incorrect substitution: `(let x = 5 in 4 + 2) + 4`

Substitution of Free Occurrences Only

- The actual substitution of `[4/x] (let x = 5 in x + 2) + x` should be
- `(let x = 5 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!

Conclusions: Lessons Learned

- 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!