S1.1: Introduction to OCaml

CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2018

After the principles...

- We've said that many of the principles in which we are interested are more directly elucidated in a language like OCaml.

- So we should learn a bit of OCaml before we do much else.

Reading

Reading from "Introduction to OCaml" by Jason Hickey.

- Chapter 1: Introduction
- Chapter 2: Simple Expressions
- Chapter 3: Variables and Functions
- Chapter 4: Basic pattern matching
- Chapter 5: Tuples, lists, and polymorphism
- Chapter 6: Unions

You do not need to digest every detail, but be aware of main points and user as a reference later.

OCaml: primary characteristics

- expression evaluation - no assignment statements (in pure functional core of the language)
- strong, static type system: (types are inferred!)
- support for structured data - lists, tuples, records, inductive types (a.k.a. algebraic data types)
- pattern matching of data
- higher-order, curried functions
- automatic memory management
- sophisticated module system

OCaml: expression evaluation

- Expression evaluation is the primary means of computation in OCaml, and other functional languages.
- Roughly, expressions are what appear on the right hand side of assignment statements.
- If we give up assignments, while-loops, for-loops, etc. what can we do?
- What do functional programming languages provide that makes such a thing at all reasonable?

  Recursion, higher-order functions, for starters.

Working with the OCaml interpreter in lecture

- I use `utop` as a "toplevel" REPL for OCaml.
  (REPL = read-evaluate-print-loop)
- The history files contain all commands and expressions typed in.
- Terminal captures from lecture demos are (will be) in the public class repository.
- Go back and review these if you have questions.

Simple expression evaluation

- OCaml's treatment of integers, Booleans, strings, etc. is not very different from other languages.
- Let's consider some examples.
- One exception: integer operators

  `+ - * /`

  are different from floating point operators

  `+.   -.   *.   /.`
- Note that OCaml also reports the type of values it computes.

OCaml types

- In our demos we saw the following basic types: `int`, `float`, `bool`, `string`, `char`.
- These have operators and functions that are not unexpected.
- Type errors are reported for expressions such as `1 + "Hello"`.
- Exceptions arise from some errors in values (of the correct type), such as division by 0.

OCaml modules and utop

- OCaml modules organize code into useful components.
- The `String` and `Char` modules contains functions you might expect.
- utop shows what names are valid as you type in expressions and this can be useful in looking for helpful functions in the libraries.
- Try typing `Char.u` and you'll see four possible function names at the bottom of your screen.
- Continue typing pp, hit 'tab' then `;;` and press return. utop shows you the type of that function. This should give you some idea what it does.
- Use this for the `String`, `List`, and other modules.

Name bindings, let expressions

- We can bind values to names using let-expressions.
  - `let` *var* `=` *expr*
    e.g.`let x = 7 ;;`
  - `let` *var* `=` *expr* `in` *expr*
    e.g.`let y = 8 in y * y ;;`
- Let's look at some examples.

Name bindings, let expressions

- OCaml *infers* the types for these variables.
- The term "variables" is a bit of a misnomer in that we cannot change (or vary) their associated value.
- We can nest let-expressions.
- A variable declaration's *scope*: program text over which the variable can be used.
  This is a static property.
- A variable declaration's *extent*: program execution time in which reference may occur (and the value must be accessible in memory).
- These will get more interesting with closures.

Functions, over simple data

- Functions are just values
- `let inc = fun x -> x + 1`
- We can write function literals, that is, lambda-expressions
  Historically, these are written as $\lambda x \rightarrow x + 1$
- OCaml provides more intuitive declarations:
  `let` *name parameters* `=` *expr*
  e.g.`let inc x = x + 1`
- Examples ...

Function types

- Consider the type of our increment function

```
# let inc x = x + 1 ;;
val inc :  int -> int = <fun>
```

- Now, what about add?

```
# let add x y = x + y ;;
val add :  int -> int -> int = <fun>
```

- We might expect (incorrectly) something like
  `int, int -> int`

Function types

- In fact, `let add x y = x + y` is just short-hand for

```
# let add = fun x ->
                  fun y -> x + y ;;
val add :  int -> int -> int = <fun>
```

- The function type operator `->` is right associative.

- `int -> (int -> int)`
  and
  `int -> int -> int`
  are equivalent.

Curried functions

- Curried functions "take their arguments one at a time."

- ```
  # let add x y = x + y ;;
  val add :  int -> int -> int = <fun>
  ```

- We can define increment using add.

```
# let inc = add 1 ;;
val inc :  int -> int = <fun>
```
```

Curried functions

- This means we may need to use parenthesis to group each individual argument.

```
# add (1 + 3) 7 ;;
- :  int = 10
```

- The "function application operator" is implicit.
  We just write the arguments after the function.
- This "operator" is left associative.
  Can we add parenthesis to make this explicit?
- Consider some examples and possible errors.

Different varieties of phrases

There are *different kinds* of phrases in OCaml and other languages.

1. "expressions" - that evaluate to a value
2. "statements" - in C, Java, etc.
   These perform some action, perhaps changing a value in memory.
3. "types" - a sub-language for "type expressions"
4. "declarations" - declaring new names/variables
5. "patterns" - (coming soon)...

The language of types

- Types are an important "sub-language" in OCaml and other languages.
- There are constants: `int`, `float`
- Variables: `'a`, `'b`
- Operators: `list`, `->`
- These form a proper language of types.

Types as an organizing principle

- Types (or type expressions) provide the first approximation of understanding what a function does.
- Without a proper language of types, it is difficult to even think in these terms.
- This is missing in dynamically typed languages like Scheme, Clojure, Python, etc.
- In this regard, *static* checking is not the point. Being able to think properly about types - if they are checked at compile time, at runtime, or never(!) - is what matters now.
- Thinking in terms of types shapes our thinking and helps us design programs.

Using files in utop

- `#use "samples.ml" ;;`
- This will load the declarations in a file just as if you'd typed them in.
- You do not, however, need the `;;` to terminate declarations.
- Let's write a GCD function, stored in `gcd.ml`.

Recursive functions

- "let-rec" expressions
- *e.g.*
```
let rec fact n =
      if n = 0 then 1 else n * fact (n-1)
```
- The scope of the variable in a let-rec includes the defining expression.
- This is not the case if a non-recursive let.

GCD

- How can we compute the greatest common divisor of two positive integers?
- Our strategy,
  1. pick a number that must be greater than or equal to the GCD
  2. decrement it by one until it is a common divisor
- How can we design such a function?

Designing GCD

1. What is its type?
2. Any useful helper functions it should call?
3. "decrement" sounds like an assignment statement, but we don't have those.

Lists and tuples

We now turn to some more traditional data structures.

- lists — of elements of the same types
- tuples — of elements of different types

Lists

- Lists are ubiquitous in functional languages and sometimes serve the same purpose as arrays in imperative languages.
- Literals
  `[ ]`, `[1; 2; 3; 4]`
- Constructors
  `[ ]`, `::` (read "cons") ⇐ very important !
  `1 :: []` = `[1]`
  `1 :: 2 :: 3 :: []` = `[1; 2; 3]`
- Operators
  `[1; 2; 3] @ [4; 5; 6]` = `1; 2; 3; 4; 5; 6]`

Pattern matching and lists

- Pattern matching - "a much better `switch` statement"
- ```
  let rec sum xs =
     match xs with
     | [ ] -> 0
     | x::rest -> x + sum rest
  ```
- `match` ... `with` *clauses*
- A *clause* has the form `|` *pattern* `->` *expr*
- Let's write some list functions.

List functions

- OK, what is your solution?
- Do your patterns match those in this solution?
- Good style: use _ in patterns when the value matched is not used in the corresponding expression.
  ```

Parametric polymorphism

- `sum: int list -> int`
- `is_empty: 'a list -> bool`
- `length: 'a list -> int`
- What is different (and interesting) about these types?
- `is_empty` takes lists of any type, while `sum` takes only lists integers
- This notion is called *generics* in Java, *templates* in C++.

Incomplete patterns

- OK, please share your solution.
- Let's consider different variations.
- OCaml warns you if your patterns are *"non-exhaustive"*
  It even tells you what your missing.
- What *should* we do for our function?
- What does OCaml do?

Recursive list functions

- Many functions over lists use a `match` with a clause for each of the 2 constructors: `[ ]` and `::`
- Lists are *inductive data*,
  processed by *recursive functions*.
- That previous point is important.
- The OCaml `function` syntax is a useful shortcut.

```
let rec sum_v2 = function
   | [ ] -> 0
   | x::rest -> x + sum_v2 rest
```

Tuples

- At first glance, tuples are like records but elements are identified by position, not by field name.

- These are *product* types. The values of a tuple type of 2 elements are those in the Cartesian product of the element types.

- For example ...

- Pattern matching is used here as well.
  But these patterns are "irrefutable".

Pulling the pieces together

- A *partial mapping* from, say, strings to integers could be represented by a value of the type `(string * int) list`.

- ```
  let m = [ ("dog", 1); ("chicken", 2);
            ("dog", 3); ("cat", 5 ]
  ```

- `let str_eq s1 s2 = s1 = s2`

- `lookup_all "cat" m` evaluates to `[5]`

- `lookup_all "moose" m` evaluates to `[]`

- `lookup_all "dog" m` evaluates to `[1; 3]`

- Can we write `lookup_all`?

What goes wrong with programs?

- Programs, of course, may have errors.

- We may detect these statically - before the program runs.
  Typically this is when the compiler runs or the program is loaded into the interpreter.

- Or dynamically - when the program runs.

- For example, syntax errors are statically detected by a compiler.
  Division by 0 is detected at run-time.
  ```

Errors in programs - static errors

Some errors will be detected automatically, others will not be.

Static errors - the best kind!

We are told of a problem without needing to run the program.

- syntax errors
- static type errors

Errors in programs - dynamic errors

Dynamic errors - these are detected during the execution of the program.

- program terminates abnormally
- OK, at least we know something was wrong
- division by 0 - the processor sends an interrupt
- memory accesses outside of process' memory space
- type errors in dynamically-typed languages (Python, Clojure)
- exception that detect programmer specified errors.

Errors in programs - not detected

The program just keeps going, with bogus data - Oh no...

- Invalid operations that do not fail.
- Invalid memory accesses *inside* of process' memory spaces
- Adding a string to an integer in an untyped language (machine code).

Strong static type systems

- OCaml has a *strong*, *static* type system
- It is a *safe* language.
- What does “safe” mean?
- *strong* = program never execute type-incorrect operation or invalid memory access
- *static* = this is checked before the program runs.
- *safe* = strong, static type system

Expressiveness of types

- OCaml doesn’t detect division by 0, but the hardware will.
- Since the hardware doesn’t detect type-incorrect operations or invalid memory accesses (within the users allotted memory space), OCaml must prevent these.
- So OCaml can detect all invalid operations
  - some through the static type system
  - some through dynamic (run-time) checks

Static vs Dynamic Typing

- A *static* type system works at compile time, before the program runs, to detect type errors.
  - Java, C, OCaml, Haskell have static type systems.
- A *dynamic* type system works at program run time, as the program executes.
  - Python, Scheme, Ruby, Clojure have dynamic type systems.
- Static type systems are preferred
  - Error are detected when the programmer can fix them.
  - Statically-typed languages are more efficient since run-time checking of types is avoided.

Type systems

The challenge - design strong static type systems that are

1. expressive, and
    - It is difficult to have a static type for non-zero integers.
    - So the question becomes

      "What properties can types express?"

2. easy to use.
    - Type inference can help with this as we don't need to write down all the types. But it is recommended to write types for some parts to provide machine-checked documentation.

Different varieties of phrases

There are *different kinds* of phrases in OCaml and other languages.

1. "expressions" - that evaluate to a value
2. "statements" - in C, Java, etc.
   These perform some action, perhaps changing a value in memory.
3. "types" - a sub-language for "type expressions"
4. "declarations" - declaring new names/variables
5. "patterns" - that match values

The language of types

- Types are an important "sub-language" in OCaml and other languages.
- There are constants: `int`, `float`
- Variables: `'a`, `'b`
- Operators: `list`, `->`, `*`

These form a proper language of types.

Wait, is $*$ an operator?

Try:

- `let x :  int * int * int = (1,2,3) ;;`
- `let x :  (int * int) * int = (1,2,3) ;;`
- `let x :  int * (int * int) = (1,2,3) ;;`

We really have several “mix-fix” operators.

$*$ and $*...*$ and $*...*...*$

Types as an organizing principle

- Types (or type expressions) provide the first approximation of understanding what a function does.
- Without a proper language of types, it is difficult to even think in these terms.
- This is missing in dynamically typed languages like Scheme, Clojure, Python, etc.
- In this regard, *static* checking is not the point. Being able to think properly about types - if they are checked at compile time, at runtime, or never(!) - is what matters now.
- Thinking in terms of types shapes our thinking and helps us design programs.

Recap

- Parametric polymorphism.
- Lists as inductive data structures.
- Language of types.
- Types as an organizing principle.