Lecture: Non-Imperative Programming

You need a programming language to build a new programming language–isn't that odd? In this course, we will learn how common language features such as objects, exceptions, loops, etc. are built. But, we wouldn't learn much if we simply used these features to implement themselves. Therefore, we are going to restrict ourselves to (mostly) functional programming to build these features from the ground up. There are several partisan reasons to use functional programming, object-oriented programming, or some other programming fad. The technical reason we start with functional programming in this course is that functional programming is mathematically simple. In particular, it will be straightforward and to extend a simple functional language with other common language features.

Before we start building language features, we'll see how to write code in a simple functional style, without additional features such as objects, loops, and so on. We are not going to introduce any new OCaml features in this lecture, but we will introduce several key functional programming ideas.

1 Functions are Values

Functions can be passed as arguments to functions. The last lecture introduced several higher-order functions, which show that functions can be passed as arguments to other functions. For example, `filter f alist` produces a list of all the items $x \in alist$, where $f(x)$ is true.

```
let rec filter (f : 'a -> bool) (alist: 'a list) : 'a list =
  match a list
  | [] -> []
  | hd :: tl -> if f x then hd :: (filter f tl) else filter f tl
```

This allowed us to write several functions very easily, for example:

```
let is_even (n: int): bool = n % 2 = 0

let only_evens (alist: int list) = filter is_even alist

let non_zero (n : int): bool = n != 0

let only_non_zero (alist: int list) = filter non_zero alist
```

Functions can be nested within other functions. If our goal was to write the list-processing functions, then the functions `is_even` and `non_zero` are cluttering our code. Another programmer may think that they are signficant, even though they are just trivial helper functions. We can eliminate this top-level clutter by nesting them:

```
let only_evens (alist: int list) =
  let is_even (n: bool): bool = n % 2 = 0
  in filter is_even alist

let only_non_zero (alist: int list) =
  let non_zero (n : int): bool = n != 0
  in filter non_zero alist
```

In particular, the names `is_even` and `non_zero` are *not defined* outside their respective functions.

Functions can produce functions. Functions can also produce other functions. For example:

```
let make_adder (x: int): int -> int =
  let add_x (y: int): int = x + y in
  add_x x

let add_three = make_adder 3

assert(add_three 10 = 13)
```

Functions do not need to be named. Unlike other values, functions seem to have the following special property: every function has a name, but the other kinds of values do not. For example, we can simply write `[1; 2; 3]` and don't need to give this list a name. So far, all of the functions we've seen have been named with `let`.

It turns out that this is just a convention. As with other values, functions don't need names. For example, here is a function that adds two numbers:

```
fun (x: int) (y: int) => x + y
```

This function does not have a name, but it can be applied just like any other function:

```
(fun (x: int) (y: int) -> x + y) 10 20
```

The code above is not easy to read. It will be a lot clearer of we give the function a name, which we can do using `let`:

```
let adder = fun (x: int) (y: int) -> x + y
```

We can write the same definition in this way:

```
let adder (x: int) (y: int) = x + y
```

You should think of the latter form of function definition as a convenient shorthand.

In general, it is a good idea to name your functions. But, there are certain situations where a short, anonymous function can make your code easier to read and write.

For example, we earlier defined the `only_evens` function, using a helper function to check for evenness. Here is simple, one-line definition using an anonymous function:

```
let only_evens (alist: int list) = filter (fun (n: int) -> n % 2 = 0) alist
```

Functions can be stored in data structures. The following type holds two functions:

```
type foo = { m1 : int -> int; m2 : int -> int }
```

We can create values of type `Foo` in the following way:

```
let my_foo = { m1 = fun (x: int) -> x + 1; m2 = fun (y: int) -> y * 20 }

my_foo.m1 10
my_foo.m2 10
```

Admittedly, this isn't very useful, but notice that the function applications look a lot like method calls.

1.1 Some Definitions

Here are some terminology that is often used when discussing programming languages and programming techniques.

- *Higher-order functions* are functions that consume or return other functions as values. You can tell if a function is higher-order by inspecting its type. Does it have any nested uses of `->` in the type? If so, it is a higher-order function.
- *First-class functions* are a feature of a programming language. For a programming language to have first-class functions, it must treat functions as values, with all the rights and privileges that other values have. You must be able to use functions as arguments, produce functions as results, store functions in data-structures, and so on.

2 Scope and Substitution

Global vs. Local Variables Why does the following program raise an error?

```
let f (x: int) =
  let y = x + 10 in
  y

let r1 = f 11
let r2 = y
```

Although the program defines a variable called `y`, the *scope* of the variable is limited to the function `f`. Therefore, we cannot refer to the variable outside the function, which is why we get an error.

Substitution What does the following program produce?

```
let x = 20
let f (x: int): int = x + 5
f 10 + x
```

Within the body of `f`, the the name `x` refers to the argument to the function, which *shadows* the global variable `let x = 20`. Therefore, `f 10 = 15`. However, outside the function `x` refers to the global variable `x`, and `15 + 20 == 35`.

We can make this argument more precise by *substituting* variables with their values:

- Given the orignal program above, we first substitute the global `x` with its value `20`, to get the following program:

  ```
  let f (x: int): int = x + 5
  f 10 + 20
  ```

  Notice that we substituted the `x` on the last line with `20`, but left the `x` within `f` unchanged, since it referred to the argument `x`.

- Next, we can apply the function `f` by substituting its argument `x` with the value `10`:

  ```
  (10 + 5) + 20
  ```

- Finally, we are left with a simple arithmetic expression that is trivial to evaluate.

Here is a more compact way of making the same argument:

	Expression	Reasoning
	`let x = 20 in let f (x: int): int = x + 5 in f 10 + x`	Original expression
=	`let f (x: int): int = x + 5 in f 10 + 20`	Substitute `x` with `20`
=	`(10 + 5) + 20`	Inline `f` and substitute `x` with `10`
=	`15 + 20`	Evaluate `10 + 5`
=	`35`	Evaluate `15 + 20`

Nested Functions Scope can appear trickier when working with nested functions. But, we can use the same substitution principle to reason about them.

For example, we wrote the `make_adder` higher-order function before:

```
let make_adder (x: int): int -> int =
  let add (y: int): int = x + y in
  add

make_adder 10
```

The following calculation starts with the definition above (written in a single line for brevity) and shows that it is equivalent to a more obvious definition of `add10`:

	Expression	Reasoning
	`let make_adder (x: int): int -> int = let add (y: int): int = x + y in add in let add10 = make_adder 10`	
=	`let add10 = let add (y: int): int = 10 + y in add`	Substitute `make_adder` and substitute `x` with `10`
=	`let add10 = let add = fun (y: int) -> 10 + y in add_x`	Rewrite `add` using `fun` notation
=	`let add10 = fun (y: int) -> 10 + y`	Substitute `add` with its definition
=	`let add10 (y: int) = 10 + y`	Rewrite `add` using `let` notation

Notice that every line in this calculation is a valid OCaml program (which you should check!) and all the lines are truly equivalent to each other.

It is always possible to perform these kinds of calculations to simplify expressions with higher-order functions. For large programs, a detailed calculation may be infeasible. But, if you understand how these calculuations work in detail on small programs, you'll be able to reason carefully about larger programs without needing to actually do the calculations in detail.

3 Proper Tail Calls

As you probably know, the factorial function is defined over non-negative integers by the following recurrence:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0 \end{cases}$$

```python
def fac(n):
    if n == 0:
        return 1
    else:
        return n * fac(n - 1)
```

(a) Recursive Factorial in Python.

```python
def fac(n):
    r = 1
    while (n > 0):
        r = r * n
        n = n - 1
    return r
```

(b) Iterative Factorial in Python.

Figure 2.1: Two versions of Factorial in Python.

Figure 2.1 shows two implementations of factorial in Python. The first uses recursion and is a direct translation of the mathematical definition. The second uses iteration and is usually considered better, because the recursive version will consume $O(n)$ stack space. (The iterative version takes $O(1)$ space.)

Using OCaml's loops, we can write factorial iteratively, but we're going to stick with functional programming for now.

Here are two definitions of factorial in OCaml. Both are recursive, but the first is a direct translation of the recurrence, whereas the second builds the result in the accumulator.

```
let rec fac1 (n : int) : int =
  if n = 0 then 1
  else n * fac1 (n - 1)
```

```
let rec f (n: int) (r: int): int =
  if n = 0 then r
  else f (n - 1) (n * r)
let fac2 (n: int) = f n 1
```

Using these definitions, let's do two simple calculations. Below, we've worked out the first recursive call in great detail to show how the if-expression evaluates. Subsequently, we've omitted showing the if-expression steps:

```
  fac1 5
= if 5 = 0 then 1 else 5 * fac1 (5 - 1)
= if false then 1 else 5 * fac1 (5 - 1)
= 5 * fac1 (5 - 1)
= 5 * fac1 4
= 5 * (4 * fac1 3)
= 5 * (4 * (3 * fac1 2))
= 5 * (4 * (3 * (2 * fac1 1)))
= 5 * (4 * (3 * (2 * (1 * fac1 0))))
= 5 * (4 * (3 * (2 * (1 * 1))))
= 5 * (4 * (3 * (2 * 2)))
= 5 * (4 * (3 * 4))
= 5 * (4 * 12)
= 5 * 48
= 120
```

```
  fac2 5
= f 5 1
= if 5 = 0 then 1 else f (5 - 1) (5 * 1)
= if false then 1 else f (5 - 1) (5 * 1)
= f (5 - 1) (5 * 1)
= f 4 (5 * 1)
= f 4 5
= f 3 (4 * 5)
= f 3 20
= f 2 (3 * 20)
= f 2 60
= f 1 (2 * 60)
= f 1 120
= f 0 (1 * 120)
= f 0 120
= 120
```

Even though we've skipped several steps, you should be able to convince yourself that both calculations involve roughly the same number of steps. i.e., both definitions take $O(n)$ steps to calculate $n!$ because both perform the same number of comparisons, subtractions, and multiplications.

However, notice that the expressions on the left-hand side grow much larger. In the middle of the calculation, we have an expression with five nested multiplication operations to calculate 5!. In general, `fac1` builds an expression of size $O(n)$ to calculate $n!$, whereas expression-size in `fac2` is bounded by a constant.

You can reason about the space utilization of functional programs in this way by simply looking at the size of the expression. If you calculate on paper and find that expression-size is always bounded, then you can be assured that the program consumes a bounded amount of memory when run on a machine. Similarly, if you find that the expression size grows linearly with the size of the input, the program will consume $O(n)$ memory.

However, there is a easier way to reason about space utilization. Notice that in the definition of `f`, the result of the recursive call (`f (n - 1) (n * r)`) is immediately returned and isn't used in any way. When a function applies another function and immediately returns its result, we say the application is in *tail position* or is a *tail call*. In contrast, in `fac1`, the result of `fac (n - 1)` is multiplied with `n`, therefore it is not in tail position. When a recursive call is in tail position, the size of the expression will not grow with each recursive call. If you think about it at a lower-level of abstraction, `fac1` needs to build a stack of multiplication operations but `fac2` does not, because the multiplications are performed in the accumulator.

Here are another pair of functions that consume a bounded amount of memory:

```
let rec even (n: int): bool = if n = 0 then true else odd (n - 1)
```

```
and odd (n: int): bool = if n = 0 then false else even (n - 1)
```

When `even` calls `odd`, it doesn't do anything with its result and simply returns it, i.e., the call to `odd` is in tail position and so is the call to `even` in `odd`. Therefore, these two functions are mutually tail recursive. If you think about it operationally, when the code is compiled and run on a machine, the processor jumps back and forth between the two functions and doesn't need to consume any stack space.

4 Programming without Exceptions

Many functions are not defined on all inputs. For example, if you're reading input from a keyboard (i.e., a string) and want to parse the string as a number, you can apply `string_of_int`:

```
# int_of_string (read_line ())
42
- : int = 42
```

But, if the string is not a numeral, you get an exception:

```
# int_of_string (read_line ())
42
Exception: Failure "int_of_string".
```

You've encountered other ways of signaling errors. For example, if you lookup an unbound key in a Java hashtable, you get the `null` value:

```
import java.util.Hashtable;
Hashtable ht = new Hashtable<Int, String>();
ht.put(10, "hello");
String r = ht.get(20);
assert(r == null);
```

Finally, here is a more insidious example. The following function calculates the point (x, y) where two lines, defined by $y = m_1 \cdot x + b_1$ and $y = m_2 \cdot x + b_2$, intersect.

```
type point = Point of float * float

let inter (m1: float) (b1: float) (m2: float) (b2: float): point =
  let x = (b2 -. b1) /. (m1 -. m2) in
  Point (x, m1 *. x +. b1)
```

However, the function is not defined when the two lines are parallel (i.e., when $m_1 = m_2$). In this case, the the denominator, `m1 - m2` is `0`. So, you might expect a divide-by-zero exception. But, that's *not* how floating point numbers work:

```
utop # 1.0 /. 0.0;;
- : float = infinity
```

So you can't even catch this error with an exception handler, since no exception is raised:

```
utop # inter 5.0 3.0 5.0 4.0;;
- : p = Point (infinity, infinity)
```

All these mechanisms for signalling errors share similar flaws:

1. *Exceptions*: you have to remember to catch them, or your program will crash. You can't tell if a function will throw an exception without carefully reading its code, which may not even be possible if it is in a library.
2. *Producing null*: even worse than exceptions, because your program will not (immediately) crash on an error. When it does crash, you'll spend a lot of time trying to figure out what produced the null-value.
3. *Producing other null-like values*: see above.

The real problem is that the types of these functions are not sufficiently precise.

- The type of `int_of_string` is `string -> int`, but it may throw an exception instead of producing an `int`.
- The type of `ht.get` in Java is `String get(Int)`, but it may produce a `null`.
- The type of `inter` is `float -> float -> float -> float -> point`. However, this function can produce `Point (infinity, infinity)`, which is clearly not what we had in mind.

A Solution Let's use `inter` as an example and modify the function so that its type makes it obvious that it may not always return a `point`. We introduce the following type:

```
type optional_point =
  | SomePoint of point
  | NoPoint
```

And we modify `inter` to produce `NoPoint` instead of a malformed-`Point`:

```
let inter (m1: float) (b1: float) (m2: float) (b2: float): optional_point =
  if m1 = m2 then
    NoPoint
  else
    let x = (b2 -. b1) /. (m1 -. m2) in
    SomePoint (Point (x, m1 *. x +. b1))
```

With this new type, any program that applies `inter` will be forced to check if if a point was produced:

```
match inter m1 b1 m2 m2 with
  | NoPoint => printf "no intersection"
  | SomePoint (Point (x,y)) => printf "intersection at (%f,%f)" x y
```

4.1 The Option Type

Ocaml has a builtin generic type called `option` that abstracts the pattern we discussed above. For example, here is `inter` rewritten to use `option`:

```
let inter (m1: float) (b1: float) (m2: float) (b2: float): point option =
  if m1 = m2 then
    None
  else
    let x = (b2 -. b1) /. (m1 -. m2) in
    Some (Point (x, m1 *. x +. b1))
```