Elixir, functional programming and the Lambda calculus.

Programming II - Elixir Version

Johan Montelius

Spring Term 2018

Introduction

In this tutorial you're going to explore lambda calculus and how it relates to functional programming. We're going to look at some examples using Elixir to see how a functional programming language can be expressed in lambda calculus, but before we begin you need a bit of historical background.

Lambda calculus, or λ-calculus, was introduced by Alonzo Church in 1932. It was a formal description of mathematics and used *function abstractions* and *function applications* as the basis. The calculus was used in mathematics to study computability and can be shown to be Turing complete i.e. anything computable can be computed using λ-calculus.

The beauty of λ-calculus is that it is so simple, it consist of very few constructs and rules, yet it is as powerful as the most powerful programming language. No one in their right mind would program anything in λ-calculus, but it has served as the inspiration of a whole family of languages.

In the dawn of computers the only programming language was the language of the hardware i.e. assembler. In the late fifties computers were however powerful enough to allow programs, written in high-level languages, to be compiled to machine code. One of the first high-level languages that were defined was Lisp. It was developed by John McCarthy in 1958 and was directly based on λ-calculus. It introduced many features of programming languages that we now take for granted.

Lisp has been followed by a number of functional programming languages: *ML*, *Scheme*, *Clojure*, *F#*, *Haskell*, *Scala* and many more. The functional programming paradigm has also influenced traditional imperative languages so we now have so called *lambda expressions* even in C++.

Since the λ-calculus has been so influential in the development of programming languages, it's fun to know what it is all about.

1 λ-calculus

We will start by explaining the rules of λ-calculus by using the basic arithmetic operations as an example. This is a bit of cheating since we then don't really explain how the basic arithmetic operations are defined but it's a good start in understanding λ-calculus.

1.1 λ-abstraction

You all understand what I mean if I write a mathematical expression like the following:

$$f(x) = 2x + 5$$

We have defined a function f that takes one argument x and the function is defined as $2x + 5$. You also take it for granted that $f(2)$ is equal to 9 and the λ-calculus is not much more complicated than this. What λ-calculus does, is that it formally describes the rules that we have to apply in order to determine that $f(2)$ evaluates to 9.

The first thing we have to get used to is that functions do not have names. Any sane programming language will of course add names to functions but if we want to describe the bare minimum we do away with anything that will complicate things. If we have no names we can of course not write $f(2)$ but you will see that we can express the same thing without names.

The function above, would in λ-calculus be written as follows:

$$\lambda x \to 2x + 5$$

The traditional way of writing this is $\lambda x.2x + 5$ but we will use an arrow instead since it will then map very natural to the Elixir syntax.

The expression $\lambda x \to 2x+5$ is a called a λ-*abstraction*, we're abstracting a parameter x from an *expression.* As we in regular arithmetic can write $f(2)$ we can in λ-calculus *apply* an abstraction to an expression. We write:

$$(\lambda x \to 2x + 5)2$$

We have here used parentheses to make it clear that the abstraction is one expression and we apply it to the the expression 2. Parentheses can often be omitted (application is left associative and expressions extends as far right as possible) but we keep them here to make it clear what we mean.

1.2 α-conversion

When we reason about λ-calculus we will often refer to the *free variables* of an expression. Free variables are variables that are not *bound* by a λ-abstraction. In the expression $2x + y$, x ad y are free variables but in the expression $\lambda x \rightarrow 2x + y$ only y is free. The variable x is then said to be *bound*.

A λ-abstraction can be rewritten by what is called *α-conversion*. We will then simply rename a variable that is bound in an abstraction, but the meaning will remain the same. The abstraction $\lambda x \rightarrow x + 2$ is of course identical to $\lambda z \rightarrow z + 2$ - the name of the variable does not matter. However, when we do this conversion we have to be careful if we replace x for z in the abstraction below, the result is not the same (try):

$$\lambda x \rightarrow x + z$$

The rules for α-conversion are not trivial since we can easily do the wrong thing. For example, what would it mean to replace x for z in the abstraction below?

$$\lambda x \rightarrow (\lambda z \rightarrow x + z)$$

In general we are not allowed to do a conversion that binds an otherwise free variable. We can define this in a more formal way but this is just an overview of λ-calculus.

1.3 β-reduction

A β-reduction is what we do when we apply a λ-abstraction to an argument. We will *substitute* the variable of the abstraction for the parameter. We write substitution using the following notation

$$(2x + 5)[x/3]$$

We can write down some formal rules for what it means to do substitution but it is all quite simple. The only thing we have to be careful about is when we want to substitute one variable by another. This could of course lead to very strange situations.

If we have the abstraction $\lambda x \rightarrow (\lambda y \rightarrow y + x)$ we can substitute $[x/3]$ and receive $\lambda y \rightarrow y + 3$ but what happens if we substitute $[x/y]$? This would give us $\lambda y \rightarrow y + y$ which is probably not what we mean. This is were we use α-conversion to first transform the expression to avoid name clashes. If we first transform the abstraction to $\lambda x \rightarrow (\lambda z \rightarrow z + x)$ we have no problem substituting $[x/y]$.

1.4 η-conversion

There is only one more rule that we need and this is η-conversion (eta-conversion). This rules states that if we have an abstraction, $\lambda x \rightarrow fx$, where f is an expression in which x does not occur free, then we can convert this to simply f. This is a quite simple rule that allows us to reduce something that is not needed. If we have the abstraction $\lambda x \rightarrow (\lambda y \rightarrow y + 3)x$ this is of course the same thing as $\lambda y \rightarrow y + 3$. Applying the first abstraction to 5 will result in the expression $(\lambda y \rightarrow y + 3)5$ so we might as well apply $\lambda y \rightarrow y + 3$ to 5 in the first place.

1.5 that's all

The three rules: α-conversion, β-reduction and η-conversion, are the only rules we need to define λ-calculus. We can of course spend some more pages on the formal description of the rules and the syntax, but the key point is that it's all very simple. You might rightly ask two questions; if it's so simple can we actually use it for anything and, what about these arithmetic operations, where are those defined?

The answer to the first question is that the language that we have is as powerful, in a theoretical sense, as any programming language that you know or will ever learn. The answer to the second question is - magic, we don't really need numbers nor primitive arithmetic operations; we only used them to introduce the rules of λ-calculus(more on this later).

2 Functional programming

To see how functional programming relates to λ-calculus we will show how we can express an Elixir program in terms of λ-expressions. We will see how different functional programming languages have made different choices but that they all have their roots in λ-calculus.

The connection between Elixir and λ-calculus is easy to see if we show how Elixir writes anonymous functions. The λ-abstraction $\lambda x \rightarrow x + 3$ is in Elixir written:

```
fn(x) ->  x  + 3 end
```

The λ notation is replaced by `fn`, the parameters are enclosed by parenthesis and the expression is terminated by `end`. When we apply a function to an argument the only difference from the λ-calculus is that we enclose the arguments in parenthesis.

```
fn(x) ->  x  + 3 end.(5)
```

If you have not done so all ready, start up a Elixir shell and experiment with functional expressions. Try this:

```
> add3 = fn(x) -> x + 3 end
  ...
> add3.(5)
  ...
```

There is however more to it than just different syntax. The most important difference is the order in which things are evaluated.

2.1 Order of evaluation

The λ-calculus does not define in what order the rules should be applied. The interesting thing is that it, with some exceptions, does not matter. If we have the expression

$$(\lambda x \to (\lambda y \to x + y)2)3$$

it does not matter if we do the innermost reduction first

$$(\lambda x \to x + 2)3$$

or the outermost

$$(\lambda y \to 3 + y)2$$

in the end the result will be the same, 5.

Is this always the case you might wonder and unfortunately it is not. Sometimes the order of evaluation matters. Look a the mysterious looking expression below:

$$(\lambda r \to rr)(\lambda r \to rr)$$

If we apply the leftmost abstraction to the argument we will duplicate this and apply it to itself. The result is

$$(\lambda r \to rr)(\lambda r \to rr)$$

but this has led us nowhere, we're stuck in an infinite loop. Make a note of this: it is possible that an evaluation will never return an answer.

Now take a look at the abstraction below:

$$\lambda x \to (\lambda y \to x)$$

This is an abstraction that when applied to an argument would give us an abstraction. This abstraction takes an argument but simply returns the original argument. If you carry out the required β-reduction for the expression below, I think the result will be 3. It does not really matter what we have instead of 2, the result will always be 3.

$$(\lambda x \to (\lambda y \to x)2)3$$

The problem is if we instead of 2 write the abstraction that will only result in a looping computation.

$$(\lambda x \to (\lambda y \to x)((\lambda r \to rr)(\lambda r \to rr)))3$$

If we're smart we will of course apply $(\lambda y \to x)$ first and get away with the looping expression and then apply the abstraction $(\lambda x \to x)$ to 3. If we're not so smart we will spend the rest of our lives evaluating the loop.

Any programming language would have to define the order of evaluation so that a programmer could avoid infinite loops and be able to estimate the run-time complexity of a program. In Elixir, as in most languages, the rule is that the that the arguments are evaluated first, before the function is applied. This is called *eager evaluation*, *applicative order* or *call-by-value.* Some languages, most noticeably Haskell, take the opposite approach and applies the function first and evaluate the arguments only if needed. This is called *lazy evaluation*, *normal order* or *call-by-name.*

There are pros and cons with either strategy so one should not take one for granted. This small tutorial on the subject is too short to look at the different strategies. The important thing is that different strategies exist and that they are equal modulo infinite computations (and exceptions).

> *If one strategy produces a result then any strategy will, if it terminates, produce the same result.*

This property is something that a compiler or run-time system can make use of. Without fear of doing the *wrong* thing, it can choose to modify the evaluation order or do evaluation for example in parallel. The behavior must of course be predictable in time and space requirements. Changes made by the system should only improve things i.e. never go into an infinite loop if the evaluation order of the programming language would not go into a loop.

2.2 More than one argument

One limitation with the λ-calculus syntax is that it only allows one parameter to a function. This might seem like a serious deficiency but computation-wise it does not make a difference. If we extended the syntax of the language we could of course write something like this:

$$\lambda xy \to x + y$$

This is a harmless extension to the language since we can always rewrite it in terms of λ-abstraction that only take one argument. The above expression would be written:

$$\lambda x \to (\lambda y \to x + y)$$

Applying one argument after the other is rather complicated and therefor any functional programming language have a syntax that allows more arguments. The important thing to note is, that we do not have to change or add any rules to the calculus. It is all just syntactic sugar that is it makes things easier to read and write.

While using several arguments to a function make life easier the we sometimes want to do the opposite. We want to turn a function of several parameters to a sequence of functions of one parameter each. This is so important that it has been given a name, *currying*, after the mathematician Haskell B. Curry (who also gave name to the language Haskell).

In languages where currying is provided by the compiler, one can define a function with for example two arguments and then apply it to its first argument to get a specialized function in return. This is not possible in Elixir, but if it was, one would be able to write something like the following:

```
f = fn(x, y) ->  x  + y end; f3 = f.(3); f3.(5)
```

Here, `f` is function that takes two arguments but `f3` is a specialization that will add `3` when it is applied to a second argument. If we want to do something like this in Elixir we would have to do it by hand. Try this in an Elixir shell:

```
f = fn(x) -> fn(y) ->  x  + y end end; f3 = f.(3); f3.(5)
```

2.3 Let-expressions

If you have done some Elixir programming you have of course seen how an Elixir function consists of a *head* and a *body*, where the body is a sequence of expressions. We can for example write the following function in Elixir.

```
fn(x) ->  y = x+4; y + y end
```

How is this translated into λ-calculus? There is nothing called a sequence in λ-calculus so we have to rewrite it in terms of regular language constructs. We rewrite the abstraction as follows:

```
fn(x) -> (fn(y) -> y + y end).(x+4) end
```

We now have an function of one argument that will apply the function `fn(y) -> y + y end` to the argument `x+4`. We now have a form that correspond to the regular λ-calculus syntax.

$$\lambda x \rightarrow (\lambda y \rightarrow y + y) x + 4$$

The shorter form is often referred to as a *let-expression* and is read as *let the variable ... hold the value ... in the expression ...*. We find this construct in many functional programming languages, this is for example how it is written in Lisp:

```
(lambda (x) (let (y (+ x 4))  (+ y y) ))
```

and in Haskell we would write it as follows:

```
(\ x  -> let  y = x + 4  in  y + y)
```

There are limitations on what we allow in let-expressions. One thing we cannot write in Elixir is the following:

```
fn(x) -> l = [x | l];  l end
```

If we would try to turn this into a form of a λ-expression we will have a problem with the infinite list `l`. In some functional programming languages however, this is perfectly fine. Both Scheme and Haskell allow let-expressions to have recursive definitions. In Scheme we can write:

```
(lambda (x) (letrec ((l (cons x (delay l)))) l))
```

We then have to explicitly state that we should not evaluate the variable l when declaring l. We can later force the evaluation if we want to go through the list of infinite numbers. In Haskell, it is so common to use infinite structures that it is provided by default.

```
(\ x  -> let  l = x:l  in  l)
```

Haskell is as we mentioned a language that uses lazy-evaluation so it is more natural to work with infinite structures. Working with infinite structures is quite fun and you can write very nice program (that are mind-boggling) but in Elixir it is not allowed (or rather we have to construct them manually).

2.4 Recursion

One thing that we have avoided so-far is how to express recursion. In Elixir, or any programming language, defining a recursive function is as simple as writing `append`.

```
def append(x, y) do
  case x do
    [] ->
      y
    [h | t] ->
      [h | append(t, y)]
  end
end
```

How do we define a recursive function if we don't have named functions? We could do it in Scheme since we there have the `letrec` construct. We define a local variable, `app`, as a lambda expression and then use this local variable in a recursive call.

```
(lambda (a b)
      (letrec ((app (lambda (x y)
                             (if (equal? x '())
                                     y
                                     (cons (car x) (app (cdr x) y))
                             ))))
                (app a b))
```

If we can not do this in plain λ-calculus then we have a problem. We need to be able to express recursive functions or programs would have to be infinite in size which is not very practical. Can we do a trick and define recursive functions without having to extend the λ-calculus? Have a look at this:

```
def append(a, b) do
  app = fn x, y, f ->
    case x do
      [] -> y
      [h | t] -> [h | f.(t, y, f)]
    end
  end
  app.(a, b, app)
end
```

Hmm, we're defining a local function, `app`, that takes three arguments, two lists and a mysterious third argument `f`. We then hope that `f` will save us when it is time for the recursive call. Will this work? Well, look at how we are using `app`, when we call it with the arguments `a` and `b` we pass `app` itself as the third argument. We're not violating the rules of λ-calculus and yet we have managed to express a recursive function.

There is a general way to do this and the trick is to use something called the *Y-combinator*. It can be used to transform any recursive definition to a non recursive definition.

$$\lambda f \to (\lambda x \to f(xx))(\lambda x \to f(xx))$$

If you want to know more about how this works you should learn about *fix points* and how they can be used. Exactly how this is done is outside the scope of this tutorial, but now you have seen the Y-combinator.

To see if you can implement a recursive function without using a recursive definition you can try to implement the Fibonacci function. Take a look at the implementation of `append/2` and try to do something similar.

2.5 Named functions

Even though it is possible to define recursive functions without having to use names it is a whole lot more practical to use names. All functional programming language introduce named functions and Elixir is not an exception.

Even though we have named functions the ability to use nameless functions and pass them as arguments to other functions is a very powerful technique. Since this is the core of λ-calculus we know exactly what it means and there is nothing strange about it. It is of course strange if you approach functional programming from an imperative programming language but now you know the basics of λ-calculus and will after some practice take it for natural.

3 Church numerals

If we only have λ-expressions we could not express very much but we could express something. We could express the following interesting function, a function that ignores its argument and simply returns a function that will return its argument.

$$\lambda f \to \lambda x \to x$$

Another function is a function that returns a function that applies the argument to the argument of that function:

$$\lambda f \to \lambda x \to fx$$

We could then describe functions that applies the argument twice or three times and this is the trick used by Church numerals. We represent the natural numbers by functions on the above form. The number 0 does not use the argument at all while 4 applies the argument four times. This is how we would write the number 4:

$$\lambda f \to \lambda x \to f(f(f(fx)))$$

If this is how we represent numbers the question is if we can use them and for example express addition. Addition is a function that takes two arguments, both encoded as above, and returns a function that represents the addition of the two arguments. How about this (we allow the λ-abstractions to take two arguments):

$$\lambda ab \to \lambda fx \to af(bfx)$$

Since this becomes quite messy to write, and since you should learn some Elixir programming in the course, we could try to implement this in Elixir. We first write two functions, one that will generate a Church numeral from an integer and one that will turn a Church numeral into an integer. Create a file `church.ex` and get your Elixir system ready, you will have to do a lot of experimenting to understand what is going on.

```
defmodule Church do

  def to_church(0) do
    fn(_), y -> y end
  end
  def to_church(n) do
    fn(f, x) -> f.(to_church(n - 1).(f, x)) end
  end

  def to_integer(church) do
    church.(fn(x) -> 1 + x end, 0)
  end

end
```

Note how simple it is to implement the function that turns a Church numeral into an integer. We simply ask that numeral to apply the function `fn(x) -> 1 + x end` as many times as it is supposed to do on the number `0`. Compile the file and try this in the Elixir shell:

```
> four = Church.to_church(4)
  ...
> Church.to_integer(four)
  ...
```

Now we can define the function `succ(n)` that will take a numeral and return a numeral that is the *successor* i.e. one more. We can do this simply by returning a function that takes two arguments, a function `f` and some value `x`, and uses `n` to apply `f` `n` times before applying `f` again on the result.

```
def succ(n) do
  fn(f, x) -> f.(n.(f, x)) end
end
```

Compile and see what the successor of `four` is; does it work?

We can write addition as outlined in the λ-expression above and the product is almost as simple. Do some experiment with these functions to see that they actually do what they are supposed to do.

```
def add(n, m) do
  fn(f, x) -> n.(f, m.(f, x)) end
end
```

```
def mul(n, m) do
  fn(f, x) -> n.(fn(y) -> m.(f, y) end, x) end
end
```

If you think this was easy you can try to figure out how to write the *predecessor* function. The predecessor should of course do the opposite of the successor function and the only question is what the predecessor of zero should be. Since we do not have any representation of negative numbers we simply state that the predecessor of zero is zero.

If you give this challenge some thought it will turn out to be a quite tricky problem. If you don't Google it, or continue to read this assignment, it will probably take you three years to figure out how to do it. Even if you read the following and realize that it works, it is very likely that you will not be able to recreate it tomorrow. OK, you're warned - here we go.

When defining `pred(n)` we must of course make use of `n` somehow. The trick is to apply `n` to two functions where the first function is a function that returns a function. When we are done with `n` we will take the result, which is a function, and apply it to the identity function; I know this does not make sense.

```
def pred(n) do
  fn(f, x) ->
    ( n.(  # n is a Church numeral
        fn(g) -> fn(h) -> h.(g.(f)) end end,  # apply this function n times
        fn(_) -> x end)  # to this function
    ).(fn(u) -> u end)  # apply it to thee identity function
  end
end
```

Assume we call `pred(four)` where `four` is the Church numeral for 4. Then we will return a function, that as expected takes two arguments, a function `f` and and argument `x`. To see how this function works let's apply it to `i` and `0`.

First we will call `four`, with a strange looking function and a function that ignores its argument. Let's apply this functions once, what do we get?

```
fn(h) -> h.((fn(_) -> 0 end).(i)) end
```

We have taken the function that ignores its first argument and substituted it for g. We now apply this function to `f` and get the function.

```
fn(h) -> h.(0) end
```

So after one recursion we have replaced the function that ignores its argument with a function that applies it once. We have three more recursions before we're ready. The next recursion will give us something that looks like follows:

```
fn(h) -> h.((fn(h1) -> h1.(0) end).(i)) end
```

This can of course be written:

```
fn(h) -> h.(i.(0)) end
```

Hmmm, let's give it another shot:

```
fn(h) -> h(fn(h1) -> h1.(i.(0)) end).(i) end
```

or

```
fn(h) -> h.(i.(i.(0))) end
```

OK, one last time:

```
fn(h) -> h(fn(h1) -> h1.(i.(i.(0))) end).(i) end
```

or

```
fn(h) -> h.(i.(i.(i.(0)))) end
```

And now we take this function and apply it to the identity function `fn(u) -> u end`. The result is:

```
i(i(i(0)))
```

This is of course exactly what we would like to see from `pred(four).(i,0)`. You should give it a try and see that it works. We try the following:

```
> three = Church.pred(four)
  ...
> Church.to_integer(three)
  ...
```

I you think this was complicated you're right! Church never figured out how to express the predecessor function. It was one of his students, Stephen Kleene, that four years later showed his professor that the λ-calculus could express something as simple as the predecessor function.

Now that we have the predecessor function we can implement a function `minus/2` that does subtraction. The implementation is simpler than what you might first think. Here are some hints to get you starting:

- Minus: use the function `pred/1` defined before and implement `minus/2`. Think like this: $m - m$ is simple, if I have a function that subtracts one (hint `pred/1`) I will use n to apply it n times on m.

There are more exercises in Church encoding, how to express Boolean values, conditional expressions, equality etc. The important thing is not exactly how this is done but that it can be done. It proves that λ-calculus is *Turing complete* i.e. any efficiently computable function can be computed using λ-calculus.

To train your skills in Elixir programming and get some more insight into Church numerals you can try the following challenges:

- Boolean value: represent *true* as a function that takes two arguments and returns the first, and *false* as a function that returns the second. Implement the Elixir functions `church_true/0` and `church_false/0` that return these functions.

- Boolean check: implement a function, `to_boolean/2`, that takes a Church Boolean and returns `true` if it is true and `false` if it is false (the solution is very simple).

- Boolean and, or, not: implement the basic Boolean operators. You have to call them something like `church_and/2` etc since `and`, `or` and `not` are the predefined Boolean operators.

- is_zero : implement a function that returns true if the argument is zero, otherwise false. This is tricky but think like this: if n is a numeral, you can apply it to a function and an argument. If n is zero it will not apply the function and will thus return the argument (which should then be?). If n is one it will apply the function once to the argument but return false (so the function should be?).

- Les-than-or-equal: implement the function `leq/2` that returns true or false depending on if the first argument is less than or equal to the second argument. Remember that $3 - 5$ is zero so n is less than or equal to m if $n - m$ is zero.

If you have done the functions above we will take a look at one that looks very simple but that puts its finger on a very interesting property - why eager evaluation might not be the silver bullet we take it for granted to be. Let's begin with a simple solution that we think will work.

- If then else: implement a function `if_then_else/3` that does what it is called. Think like this - a Church Boolean takes two arguments and returns either the first or second depending on if it is true or false. Our first argument to `if_then_else/3` is the Church Boolean that we should use to either return `Then` or `Else`.

If you implemented it as simple as possible it should be a one line function, I told you it was simple. Now let's use `if_then_else/3` to implement subtraction is a new way. Fill in the blanks in this code (let's call it `manos/2`) and give it a try:

```
def manos(m, n) do
  if_then_else(is_zero(n), m, manos(m, pred(n)))
end
```

Could not be easier, give it a try (define the Church numerals `four` and `three` first):

```
> Church.manus(four, three)
  ...
```

Hmm, something is not quite right - what is going on and why? In what order do we evaluate the function? Can we rewrite the `if_then_else/3` in any way to avoid the problem? Do we actually want to be slaves under the eager evaluation strategy? How would this work in Haskell? Maybe we need a construct in Elixir that avoids the eager evaluation order? Should we have special function that does not evaluate all arguments, a special notation that the evaluation should be delayed or a language construct?

As you see, a simple thing as *if-then-else* is quite tricky. You now also know why Elixir and most functional programming languages have a language construct that implements the if-then-else behavior that we are looking for. In Elixir we have *case expressions* and *if expressions* that gives us the functionality that we would not otherwise get.

4 Summary

If you have implemented the Church numerals I think that you have a better understanding of what the λ-calculus is. I also hope that you're a little bit better in working with functions as arguments and return values. This is something that you will not use very much in the beginning but once you get use to it you will realize how powerful it is.

You do not need to understand all the details of λ-calculus, and no one will probably never ask you to implement the Church numerals again, but having a basic understanding of λ-calculus will give you a better understanding of how different functional programming languages are related. The syntax may differ and they have adopted different strategies for order of evaluation but they are all rooted in the same paradigm. Learn the paradigm and you will easily learn any functional programming language.