OCaml Advanced Feature: Labels

Labeled and Optional Parameters

Simon Lindblad

February 14, 2017

1 Labeled Arguments

Usually in OCaml, you define parameters positionally. That is, which argument that gets assigned to which parameter is defined by the order in which they are provided. OCaml supports a way for you to define parameters by name, instead of by position.

To see how it works, we can start with an example:

```
let rec pow ~base ~exp =
    if exp = 0
    then 1
    else base * pow ~base:base ~exp:(exp-1)
```

This is a simple function that calculates the power of a base to an exponent. The names of the two parameters are base and exp. You define a label by putting a tilde in front of it. At line 4 in the sample we call the function by giving the parameters name. Here we use them in the order they are defined, but that's not necessary. Let's take a look at the same function, but where the arguments in the call have swapped positions.

```
let rec pow2 ~base ~exp =
    if exp = 0
    then 1
    else base * pow2 ~exp:(exp-1) ~base:base
```

Running these two will, as you'd expect, yield the same result. Note that you don't need provide labels, if you don't they would work in the same way as normal OCaml arguments do.

```
# pow 2 3;;
- : int = 8
# pow2 2 3;;
- : int = 8
```

There's quite a few languages with this property. We can compare it to Python's named parameters:

```
>>> def pow(base, exp):
...     return base**exp
>>> pow(exp=3,base=2)
8
>>> pow(base=2, exp=3)
8
>>> pow(2, 3)
8
```

As you can see it works in Python per default. Any parameter in Python can be assigned by providing a name when you call it. If no name is provided, it'll take the order in which the parameters are written.

OCaml and Python differentiates in certain aspects. One is when it comes to the names of the labels. In Python, they're the same as the variable. However, in OCaml you can have distinct names for the label and variable, as seen below.

```
# let d ~dividend:x ~divisor:y = x/y;;
val d : dividend:int -> divisor:int -> int = <fun>
# d ~dividend:6 ~divisor:2;;
- : int = 3
```

An interesting case of labels in OCaml is when you partially apply a function (something that can't be done in Python). You can call the function without with all arguments without providing any labels. When you partially applying one you need to.

```
# pow 2;;
Error: The function applied to this argument has type
       base:int -> exp:int -> int
This argument cannot be applied without label
# pow ~base:2;;
- : exp:int -> int = <fun>
```

Without providing a label the compiler won't know which one of the parameters you are assigning.

2 Optional Arguments

All of the arguments we've seen so far have still been required. OCaml supports optional arguments. Let's rewrite our pow function to use base 2 as default.

```
let rec pow ?(base=2) ~exp =
    if exp = 0
    then 1
    else base * pow (exp-1)
```

An interesting aspect of this is that you need at least one non-optional argument. To explain the how it works, let's look at another implementation of pow.

```
let rec pow ?(base=2) ~exp =
    if exp = 0
    then 1
    else base * pow ~exp:(exp-1)
```

At first glance it might look like this would work the same way. However, pow exp:(exp-1) does is not evaluated to an integer, it's evaluated as a function expecting an argument called base. Only by omitting the label, you tell the compiler that you don't want to assign any value to the optional argument. It's because of the same reason, partial evaluation, that you can't have an optional argument in the end. The compiler has no way of knowing if you're just partially evaluating a function, or omitting the argument. Hence, if you would put an optional parameter in the end of the function definition, it would be treated as a regular argument. This is shown below.

```
# let test ~x ?(y=1) = x + y;;
Characters 14-17:
Warning 16: this optional argument cannot be erased.
test : x:int -> ?y:int -> int = <fun>
```

You can build up more complex function definitions. Here are some examples to illustrate how they work.

```
# let func ?(x1=1) x2 ?(x3=3) x4 = x1*x2*x3*x4;;
val func : ?x1:int -> int -> ?x3:int -> int -> int = <fun>
# func ~x1:1 2 ~x3:3 4;;
- : int = 24
# func 1 2;;
- : int = 6
# func 1;;
- : ?x3:int -> int -> int = <fun>
```

In the first call, we provide all arguments and the function is executed as expected. In the second call we only provide 2 arguments, and the function returns an integer. That is, we provide the values for the two non-optional arguments x2 and x4 and the compiler knows that we want to omit the two optional ones. In the last call, the compiler rules out x1 since we didn't provide a label. However, since we didn't provide anything for x4 it returns a function where we have x3 as an optional argument and x4 as a regular one.

In Python, you have optional and default arguments as well. There defined like:

```
def pow(exp, base=2):
    return base**exp
```

Note that the order of the arguments were switched in this example. Python doesn't allow you to define arguments with default values before those without. If you could do that the compiler wouldn't know which one you were assigning. I.e the following is not allowed:

```
def pow(base=2, exp):
    return base**exp
```

In the original definition however, the code would work as expected.

```
>>> pow(3)
8
>>> pow(3, 2)
8
```

In Python, you define an argument as optional by providing a default value. In OCaml they are two separate steps. So, if you don't provide a default value the default value is treated as an optional.

```
# let f ?x1 x2 = x1;;
val f : ?x1:'a -> 'b -> 'a option = <fun>
# f 1;;
- : 'a option = None
# f ~x1:2 1;;
- : int option = Some 2
```

Here you can see that if an argument for x1 isn't provided, it'll be set to None. If an argument for x1 is set, it's treated as Some 2. I.e it's still an optional, but it contains a value.

In Python, you don't have this feature. A common solution is to set the optional parameter's default value to None, and then check whether the parameter was assigned a value in the function body.

The labeled and optional parameters in OCaml can cause some troubles when it comes to type inference. They can't be inferred completely. Let look at the following example.

```
# let f1 ~x ~y = x+y;;
val f1 : x:int -> y:int -> int = <fun>
# let f2 ~y ~x = x*y;;
val f2 : y:int -> x:int -> int = <fun>
# let h g = g ~y:2 ~x:2;;
val h : (y:int -> x:int -> 'a) -> 'a = <fun>
# h f2;;
- : int = 4
# h f1;;
Error: This expression has type x:int -> y:int -> int
       but an expression was expected of type y:int -> x:int -> 'a
# f1 ~y:1 ~x:1;;
- : int = 2
```

First, we define two functions, f1 and f2 that both takes two labeled arguments, x and y, but the order in which they take them is not the same. After that we define a function h that takes a function as an argument and calls that function with two labeled arguments, x and y. One would think that both are the same, however when you call h with f1 it does not work. OCaml isn't able to infer the type perfectly. However, in the last call we can see that f1 does work with the arguments as they are in h.

3 When Should Labels and Optionals be Used?

Since variables in OCaml has a tendency the be of shorter, harder to understand, names than in languages like Java, labels and optionals provide a way to make your code more readable. And that's when you should use it, when it makes the code more readable. They don't provide any new revolutionary way to code, but it makes it a bit easier to understand what's going on. That's also where the OCaml standard library uses it. Functions that manipulates files etc frequently have labeled parameters to show which one is the source and which one is the destination, etc. Providing good labels can be an excellent complement to documentation.

4 References

1. https://caml.inria.fr/pub/docs/manual-ocaml/lablexamples.html (Accessed on 2017-02-12)

2. https://realworldocaml.org/v1/en/html/variables-and-functions.html (Accessed on 2017-02-13)