exercise09-solution-p

December 14, 2018

0.0.1 Assignment 9.1 (P) (Delayed) evaluation, unit, side-effects, pure functions

We can see let-bindings as functions without any arguments. Therefore they can and will be immediately evaluated. If we want to delay evaluation of the bound expression until application, we can introduce an argument. However, if there are no free variables in our expression, we just need some dummy argument. To indicate this, we use the type `unit` which has only one value `()` (which can be seen as the empty tuple).

Discuss this difference between the following two expressions

```
In [ ]: let x = print_endline "foo" in x, x
```

```
In [ ]: let x () = print_endline "foo" in x (), x ()
```

1. What are side-effects? Give some examples.
2. What are pure functions? What are their benefits?
3. Why does delaying evaluation only make sense in case of side-effects?
4. Why do we want to use `()` instead of some unused variable?

1. Side-effects interact with the environment during evaluation. This means writing data without contributing to the returned value or reading data without depending on the arguments. E.g. writing/reading to/from some channel (e.g. stdout, files, network), changing/reading some data outside the function (mutable data structures, e.g. references, hashtables, arrays), reading time, getting random values etc.
2. Pure functions are functions without observable side-effects (we ignore the effects of computation itself (e.g. used memory, caches, spent CPU cycles)). Like functions in the mathematical sense, their return value will only depend on the values of the arguments and will therefore always give the same result for some input. Reproducible results facilitate testing, parallization (no interaction between different calls), memoization (we can save the result for some input and don't need to compute it again), and on-demand/lazy evaluation (if there are no side-effects, we only need to evaluate some function if we depend on its value).
3. If we have all needed arguments for application and the function has no side-effects, it does not matter when we evaluate it, since the call will always give the same value and not influence anything else. You could argue that even for a pure function it does make a difference when you evaluate if it does not terminate, but this will only make a difference if there are side-effects in the program (i.e. which side-effects are executed before the non-terminating call).
4. The only way to convey in the signature (besides some agreed upon convention) that the evaluation is not influenced by an argument, is by limiting its values to exactly one value. If

we see a signature `'a -> int` it could for example compute some hash of the first argument (only possible because of the built-in polymorphic functions depending on runtime representation); for `unit -> int` we know that it will either be a constant function or get the `int` via side-effect; for `unit -> unit` we know that the function will only do something via side-effects (or do nothing). The choice of `unit` to indicate side-effects for both arguments and results is rather arbitrary. We could also define something like the following to further classify side-effects:

```
In [ ]: type resource = Terminal | File | Network | Mutable
        type action = Read | Write
        type side_effect = action * resource
        let print_endline' x : side_effect = print_endline x; Write, Terminal
        let _ =
            print_endline' "foo" ::
            print_endline' "bar" ::
            []
```

0.0.2 Assignment 9.2 (P) Students in, students out!

Once again, we consider our student records:

```
In [ ]: type student = {
            first_name : string;
            last_name : string;
            id : int;
            semester : int;
            grades : (int * float) list
        }

        type database = student list
```

Now, we define a file format to store students that, for each student, contains a line
first_name;last_name;id;semester;gc
where gc number of lines
course;grade
follow with grades.

1) Implement a function `load_db : string -> database` to load the students from the given file. Throw an exception `Corrupt_database_file` if something is wrong with the file.

2) Implement a function `store_db : string -> database -> unit` to store the students back to the given file.

```
In [ ]: (* find the solution in the file p09_sol.ml *)
```