Lab 10: OCaml sequences, comparators, stable sorting
12:00 PM, Nov 11, 2018

Contents

1 A last OCaml type, and its utility ............................... 1
   1.1 Sequences in OCaml ........................................ 2
   1.2 Using printing in debugging ................................ 3

2 Comparators .................................................................. 3

3 Stable sorting ............................................................. 5

Objectives

By the end of this lab, you will know:

- what comparators are
- how to make a stable merge-sort
- whether insertion sort is stable

By the end of this lab, you will be able to:

- print out debugging information in the middle of an OCaml program
- sort things other than numbers, like BigNums

1 A last OCaml type, and its utility

OCaml has one more type that you haven’t seen, unit.

The usual way to construct a unit is with empty parens: (). Open up an OCaml interpreter and try typing ();; just to see how it looks.

There are other functions in OCaml that produce units. One of these is print_string. When OCaml evaluates print_string "Hello!\n", two things happen:

1. The string “Hello!”, followed by a newline, gets printed out.

2. The value () gets returned
Try typing `print_string "Hello!\n";;` to an OCaml interpreter and see what you get.

Take a look at [https://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html](https://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html) and read about “Output functions on standard output.”

Now think about printing an `int list`. You’d like to write

```ocaml
let rec print_int_list: int list -> unit = function
...
```

This is a little tricky. At the start of the list, you want to print a left bracket. But if you call this recursively, you’ll print another left bracket before the REST of the list, and so on. Here’s a solution: write

```ocaml
let rec print_int_list_contents: int list -> unit = function
...
```

which prints out the items in a list, each item (except the last) being followed by a semicolon and a space.

Now how do you print a whole list? You print an open bracket, then the contents, then a right bracket. The problem is that we don’t have a way to “do a sequence of things in OCaml” yet.

(There’s a very good alternative here: instead of writing a procedure that prints the list contents, you could write one that converted those contents to string form, and then the main list-printing program could put a left bracket at the front of that string, a right bracket at the end, and call `print_string` on the whole result. But we’re walking you through this alternative to illustrate something else, so bear with us.)

### 1.1 Sequences in OCaml

You’ve seen semicolons in OCaml as a way of separating items in a list. There is another use, however: semicolons are used to form **sequences**.

Try typing

```ocaml
4;5;"hello";[1;2];;
```

into OCaml. You’ll get some warnings, but the value of the expression you just typed (yes, it’s an expression!) is `[1;2]`, i.e., the last item in the sequence.

What’s going on here is that `;` is actually an **infix operator** in OCaml, and the value of `a ; b` is just the value of `b`. It seems as if it’d be a waste of time to ever write a sequence, because you could just have written the last item and gotten the same result.

The exception is when the expression `a` has a **side effect**, like printing out some text to your terminal. Try typing

```ocaml
print_string ("Hello!" ^ "\n"); 4;;
```

and see what happens.
There were no warnings this time. Why not? Well, the type of `print_string` is `str -> unit`, and when the semicolon operator gets anything except a unit as its first argument, you get a warning, but when it gets a unit as the first argument, it produces no warning. This is why we are doing this exercise so you are able to print out in the middle of your procedures.

With this in mind, you can write

```ocaml
let rec print_int_list: int list -> unit = function
  aloi ->
  print_string "["; print_int_list_contents aloi; print_string "]" ;;
```

**Task:** Write `print_int_list_contents`, as well as some tests for your procedure. Why is the output type for this procedure `unit`?

### 1.2 Using printing in debugging

Suppose you write a `length` procedure like this:

```ocaml
let rec len: 'a list -> int = function
  | [] -> 0
  | hd::tl -> 1 + len tl;;
```

which happens to be perfectly OK code, but because of a mistake in some check-expect, you think it might be wrong. How can you use our `print_int_list` code from above to help you debug? Well, you’re probably pretty confident about the base case, so let’s do something with the recursive case. Because our printing procedure only works on int lists, we’ll have to change the signature a little. But after that, look at this small change to the recursive case:

```ocaml
let rec len: int list -> int = function
  | [] -> 0
  | hd::tl -> (print_int_list tl); print_string "\n"; 1 + len tl;;
```

Go ahead and type that in, and try computing the length of the list `[1;2;3]`.

Helpful, eh?

**Task:** Now write `subsets` in OCaml, and insert something that prints out the argument given to `subsets` before generating the subsets. Since the procedure we wrote before only prints int lists, you should change the signature of `subsets` to only take int lists. See what happens when you compute the subsets of `[1;2;3]`.

You’ve reached a checkpoint! Please sign up to get a lab TA to review your work.

### 2 Comparators

We call operators like `>`, `<`, and `=` *comparators*, because they compare values. These comparators are built in to OCaml. Moreover, they are *polymorphic* predicates—they consume `int` values, or `float` values, etc.—and they produce `boolean` values.
Here is an example of a comparator that is not built in to OCaml. It compares two strings, based on their respective lengths:

```ocaml
let string_compare: string * string -> bool = function (str1, str2) ->
  (String.length str1) < (String.length str2)
```

As you can see, our string_compare predicate is designed specifically for strings. It consumes two strings, and produces `true` when the first is shorter than the second, and `false` otherwise.

But here’s the thing: it’s hard to remember when string_compare produces `true`: is it when the first string is shorter than the second one, or longer than the second one? Wouldn’t it be preferable if string_compare produced a more informative value, such as `Less` or `Greater`? With OCaml’s powerful variant type system, we can do exactly this.

First, we define a generalized comparison type:

```ocaml
type comparison =
  | Less
  | Equal
  | Greater
```

Second, we rewrite string_compare so that it produces a value of type `comparison` instead of a boolean:

```ocaml
let string_compare: string * string -> comparison = function (str1, str2) ->
  if (String.length str1) < (String.length str2) then Less
  else if (String.length str1) > (String.length str2) then Greater
  else Equal
```

**Task:** Write a procedure `bignum_compare` which compares two bignums. Your procedure should consume a tuple of bignum values and produce the appropriate comparison value.

**Hint:** Remember that the number 123 is represented in reverse order by the bignum `[3; 2; 1]`.

For example:

```ocaml
bignum_compare ([], [1 ;; 2 ;; 3])
=> Less
bignum_compare ([4 ;; 7 ;; 1], [5 ;; 7 ;; 1])
=> Less
bignum_compare ([4 ;; 7 ;; 1], [5 ;; 7 ;; 1 ;; 2])
=> Less
bignum_compare ([1 ;; 2], [2 ;; 1])
=> Greater
bignum_compare ([1 ;; 8 ;; 8 ;; 9], [1 ;; 8 ;; 8 ;; 8])
=> Greater
bignum_compare ([7 ;; 1], [7 ;; 1])
=> Equal
```

---

1We wrote it ourselves!
Hint: Do not use library procedures like `List.length` or `List.reverse`. They will not lead to an efficient solution. Instead, write a procedure that recurs on two lists.

Task: What is the run time of your bignum comparator?

Task: Rewrite `insertion_sort` so that if it uses a comparator like `bignum_compare` as its last argument, so that the type becomes

`insertion_sort: ('a list * (('a * 'a) -> comparison)) -> 'a list`

To get you started, here’s the code for the ordinary insertion sort:

```ocaml
let rec insert: int * int list -> int list = function
| n, [] -> [n]
| n, (hd::tl) -> if n < hd
    then n::alo
    else hd:: (insert n tl);

let rec insertion_sort: int list -> int list = function
| [] -> []
| hd::tl -> insert hd (insertion_sort tl);
```

You’ll need to change all the “int”s to `'a`, so that it can work with arbitrary data.

Now define a new comparison, `pair_compare`, that compares two `int * string` pairs, and says that `(n1, str1)` is less than `(n2, str2)` exactly when `n1 < n2` (i.e., it compares only the “int” part). Use your insertion sort to sort a list that looks like this:

```
[(2, "two"); (1, "first"); (1, "second"); (0, "zero"); (0, "last")]
```

Task: Is the resulting sorted list a stable sort of the data, in the sense that if two items in the input are equal, then their order in the output is unchanged? (i.e., `(1, "first")` should still come before `(1, "second")`). Experiment to find out.

You’ve reached a checkpoint! Please sign up to get a lab TA to review your work.

### 3 Stable sorting

What about building a stable sorting algorithm?

Task: Rewrite `mergesort` (with a comparator argument) and once again check whether it is stable, by applying it to the example above that used `pair_compare`.

Here’s a slightly modified version of the mergesort from class. The two main differences are:

- The code uses the “compute the length of the list to split it in two” approach rather than the card-dealing approach, which helps in making it a stable sort.
- The code tests whether the head of the left list is less than or equal to the head of the right list; this too helps in maintaining stability.
You may need to change the #use line at the top to make it work for you.

**Task:** Finally, make a copy of this mergesort procedure, called `merge_sort_b` (the “b” is for “broken”) that uses a less-than rather than a less-than-or-equal-to comparison between `hd1` and `hd2`, which should make the sort unstable. Then find a list of int-string pairs that are sorted unstably by this revised version. Because the code we gives you takes in int lists, you will need to convert it to
take in 'a lists so that it could work with pairs.

You’ve reached a checkpoint! Please sign up to get a lab TA to review your work.

Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/csci10170/feedback.