
CS 421 – Spring 2007

Lecture Notes Set 9: HOFs and Records

Elsa L. Gunter¹

Transcribed by: Pooja Mathur

Corresponding to Slides: 04-05-hof (slides 28 - end) 06-userdef (slides 1 - 15)

Made available: February 5, 2007

Revision 1.0

1 Change Log

1.0 Initial Release.

2 Upcoming exam (04-05-hof slides 36-end)

Here are *some* things that you should have an understanding of for the upcoming exam.

- Writing functions with recursion
 - Recursion over integers
 - Recursion over lists
 - Tail recursion vs. Forward recursion
- Using higher order functions
- Writing the environment at different points in the code
- Writing the closure for a function (only for non-recursive functions)
- Writing enumeration types and functions over those types (which we will get to today)

There are problems on in the lecture slides that are likened to something you may see on an exam. Also, more information on the exam can be found on the website where there is a sample exam as well as a much more detailed study guide.

3 Other Higher Order Functions - continued

3.1 Mapping (04-05-hof slides 28-31)

Recall from previous lectures that we have `fold_left`, which is an example of tail recursion or iteration. And we have `fold_right`, which is an example of forward recursion or primitive recursion. An instance of primitive recursion that is often used is `map`.

Last time, we started to talk about the example `inclist`.

```
# let rec inclist list = match list with [] → []  
  | x :: xs → (1 + x) :: inclist xs;;
```

¹© 2007, Share and Enjoy

```
val inclist : int list → int list = <fun>
```

This is a function that takes a list and adds 1 to each element of the list, transforming it into a new list. That is basically what map does, it maps one list to another one. Each element in the new list was directly derived from the corresponding element in the original list.

Since map is basically doing the exact function we want to do in inclist, we can rewrite the function as:

```
# let inclist = map ((+) 1);;
val inclist : int list → int list = <fun>
```

The newly rewritten function does exactly what we were doing before, but now, there is much less code. Inclist will take a list and perform the addition operation on each element with the value 1.

Earlier, it was mentioned that map is a form of primitive recursion. That means that we can also write map using fold_right. To do this we need to know the usual three things, the base case, the operation being performed on the list and how it is being recursed. Well, the recursion is obvious; it is primitive recursion so we know that means fold_right. We know the base case is the empty list. The operation is that we are performing some function on each element and we are cons-ing that new value to the rest of the list. With that information we can get map to look like:

```
# let map f list =
  fold_right (fun x y → f x :: y) list [];;
val map : ('a → 'b) → 'a list → 'b list = <fun>
```

3.2 Possible applications of HOFs

One possible use of an HOF would be to see if given a list which of the elements after being run under a test were true or false. You could do this easily with map (or as we have just seen, fold_right as well).

Now, if you wanted to know if all the elements of the list passed the test, this could be done with either of the fold functions. Fold_left would be optimal because you can recurse over the list as long as the test is returning true. Once you get a false, you can immediately stop, even if the very first element of a million element list. All you need is one false to enable you to stop recursing. With fold_right, you would have to recurse to the very end no matter what and then check the test. So with the million element list, if only the very first element was false, you wouldn't know that until you returned all the way to the head of the list. You would not use map because map returns a list and we are looking for a single value.

3.3 Zipping (slides 32-35)

Another useful function is the zip function. It does exactly what it sounds, it 'zips' two lists together. But you only want to do this when there are elements in both lists to do this with. So the zip function only does work when there is something in both lists. That is, we cut off the longer list if there are two lists of different lengths.

```
# let rec zip list1 list2 =
  match (list1,list2) with ([], _) → []
  | (_, []) → []
  | (x::xs, y::ys) → (x,y)::zip xs ys;;
val zip : 'a list → 'b list → ('a * 'b) list = <fun>
```

Here is an example of running the zip function. Note that if either of the two lists had more element (and one stayed the same as it is now) then we would still get the same exact return value, because the extra part of one list gets cut off.

```
# zip [1;2;3] [4;5;6];;
- : (int * int) list = [(1, 4); (2, 5); (3, 6)]
```

Now, let's take zip a little further. Here we have the function zipwith. What zipwith does is takes the elements of the two lists and uses them as arguments to a function. Of course the function has to take two arguments. The results of applying the function on the elements of the two lists gets returned as a new list. So the type of the function is saying, we take a function that takes two arguments of type 'a and 'b and returns a value of type 'c. That function is an argument of zipwith, as well as a list of 'a and a list of 'b and that whole thing returns a list of type 'c. Looking at this, like the zip function, our base cases throw away anything extra in either list. Our recursive case pulls off the head of the lists and performs in the input function on them and cons-es that onto the recursive call on the tail lists.

```
# let rec zipwith f list1 list2 =
  match (list1,list2) with ([], _) → []
  | (_, []) → []
  | (x::xs, y::ys) → f x y ::zipwith f xs ys;;
val zipwith : ('a → 'b → 'c) → 'a list → 'b list → 'c list = <fun>
```

The `zipwith` function is similar to `map` in that you are using a function to make a transformation between lists, only here, you are taking two lists instead of one. The transformation involves creating a combination of the two lists.

Note that `zip with` is simply a more general version of `zip`. That is, `zip` can be written with `zipwith`. Basically, the idea when doing this is that your function that you give as input is shifting things from curried to uncurried. The name for which is pairing. This is what it may look like:

```
# let zip = zipwith (fun x y → (x,y));;
val zip : 'a list → 'b list → ('a * 'b) list = <fun>
```

4 User Defined Types

So far we have been using types that come directly from the OCaml library. But what if we wanted some structure/type that didn't have a base in the library. Well, why not write it ourselves?

4.1 Records (userdef slides 1 - 9)

A record is a group of information, all bundled together. The information does not have to be of the same type. To look up components of a record, you look things up by field name, rather than position. One advantage of using records over tuples is documentation. Code becomes more readable with this. Records force you to use names in the hopes that you will use intelligent names (meaning that your code becomes easier to read and understand).

Let's look at an example for creating a type.

```
# type person = {name : string; ss : (int * int * int); age : int};;
type person = {name : string; ss : int * int * int; age : int; }
```

We use the keyword `type` to declare that we are creating a new type. Then we have the name of the type. Inside the we write our labels (a.k.a. field) and those are `name`, `ss`, and `age`. The associated base types for each label is follows the label name and a single colon. So, `name` is a string, `ss` is a triple of integers, and `age` is a single integer. That is what we get as the type of a person.

Now, let's try to use the type person.

```
# let teacher = {name = "ElsaL.Gunter"; age = 102; ss = (119, 73, 6244)};;
val teacher : person = {name = "ElsaL.Gunter"; ss = (119, 73, 6244); age = 102}
```

There are a few things to take note. First, notice that when `teacher` was declared, the fields were inputted in a different order that was noted in the official type. That is, `age` was given before `ss`. OCaml is ok with this. It can figure out what you meant and will give you the official type value the same as in the type declaration. Sometimes, one will see that the labels have also been alphabetized, but that is also internal with the version of OCaml that you are using. Another thing to notice is that OCaml figured out on its own that `teacher` was of type `person`. You don't have to tell it that.

Look at this next example. Here, we again inputted the labels in a different order and OCaml will reorder them and figure out that `student` is also of type `person`.

```
# let student = {ss = (325, 40, 1276); name = "JosephMartins"; age = 22};;
val student : person = {name = "JosephMartins"; ss = (325, 40, 1276); age = 22}
```

You can test these objects for equality if you can test each component for equality. In this case, you can, so if you wanted to see if `teacher` was the same as `student` you would write:

```
# student = teacher;;
- : bool = false
```

And that would lead to `false` since the fields have different values.

Now, anytime you create a new type in OCaml, you want to ask the question, how can I learn from this type? That is, how can I extract information from this type?

With records, to extract information, you use pattern matching. So, for example, say you wanted to get the name, age, and third component of the triple from the `ss` of `teacher`. You would pattern match each of those elements with a name, to bind them by. So the right side of the equal sign is the name/pattern you want to bind with a label (which is located on the left). After any label, you can simply use the wild card `_` to say you don't care about that field, there is no reason to pattern match for it.

```
# let {name = elsa; age = age; ss = (_, _, s3)} = teacher;;
val elsa : string = "Elsa L. Gunter"
val age : int = 102
val s3 : int = 6244
```

Another way to access a field in a record is the dot identifier. Similar to its use in languages like C++ and Java, you can say `teacher.age` to extract the age out of the `teacher` object.

There are other things you can do with records as well. One thing is making a new record from changing a little information from a record that already exists. Note that when you do this, you are not updating the old record, you are actually making a new record with the changes applied.

```
# let birthday person = {personwithage = person.age + 1};;
val birthday : person → person = <fun>
```

So here in the example, we made a function called `birthday` that took a person as input. Then the function would take the age of that person and simply add one to it. So when applied to `teacher`, the new age would be 103.

```
# birthday teacher;;
- : person = {name = "ElsaL.Gunter"; ss = (119, 73, 6244); age = 103}
```

Take notice though, we didn't bind this new record of `teacher` to anything. So even though we made a new record with the altered age, the record disappears after the function is finished running and returns it. And if you were to check the record `teacher` again, it would still have the age of 102.

We can do this same function of creating new records from old one with also altering more than one field. All you have to do is pattern match the new information to the labels of the old record and then give the name bound the the record you want to alter.

4.2 Variants (slides 10-15)

Another way to make types in OCaml is by using Variants. Now, the basic syntax for making a new type in OCaml is to first use the keyword `type`, and then the name of the type, the equal sign, and then you give a sequence of constructors. If a constructor takes arguments you give the types of the arguments.

Doing this creates a type called *name*. It creates a collection of constructors that have the property of when being applied to arguments that are the correct input type, you get returned something of type *name*. The constructors affect pattern matching. They are the basis of almost all pattern matching.

The simplest type that can be constructed by this is called an enumeration type. So, basically this turns out to be a bag full of stuff. The stuff has some information associated with it. OCaml makes for a little organization to enable the extraction of that data.

So, let's say you wanted to make a type for the days of the week. You would get:

```
# type weekday = Monday | Tuesday | Wednesday
| Thursday | Friday | Saturday | Sunday;;
type weekday =
  Monday
| Tuesday
| Wednesday
| Thursday
| Friday
| Saturday
| Sunday
```

Now, these days are ordered. So you can use that idea for pattern matching. Say you wanted to write a function that told you what the next day was. You would write something like this:

```
# let day_after day = match day with
```

```

Monday → Tuesday
| Tuesday → Wednesday
| Wednesday → Thursday
| Thursday → Friday
| Friday → Saturday
| Saturday → Sunday
| Sunday → Monday;;
val day_after : weekday → weekday = <fun>

```

What is happening here is that we are saying, if you see day x match it with day y . We are taking something of type `weekday` to something of type `weekday`.

You can take this a little farther and use the pattern matching idea to write other, more interesting functions. So if you wanted to know what weekday it was after n many days, you could write a function that looks like this:

```

# let rec days_later n day =
  match n with 0 → day
  | _ → if n > 0
        then day_after (days_later (n - 1) day)
        else days_later (n + 7) day;;
val days_later : int → weekday → weekday = <fun>

```

So, if n is 0, then 0 days later is the day we give as input. Otherwise, if n is greater than 0, we continue to subtract 1 from n , all the while changing day to the next day. If n is less than 0, we are looking for days before. And in that case, you simply add 7 to n until you get a positive number, where that will be the day you looking for.