
CS 421 – Spring 2007

Lecture Notes Set 10:

Variants

Elsa L. Gunter¹

Transcribed by: Pooja Mathur

Corresponding to Slides: 06-userdef (slides 10 - 11, 16 - 22)

Made available: February 7, 2007

Revision 1.0

1 Change Log

1.0 Initial Release.

2 Variants

2.1 Syntax (slide 10)

The syntax of variants allows you to have type arguments in your enumeration types, which we will use to create a disjoint union. It allows you to have the same type argument occurring for different constructors, which allows you to have a "tag" union. With this tag, you can basically use integers for one purpose in one case, while another case you can use the integers for something else, and the tag is what differentiates between the two.

This also allows for recursion, because there is no restriction that the types of a variant cannot have itself as one of its member types. Using this, you can recreate the list type.

2.2 Enumeration Types (slide 11)

This gives us a very rich collection of type defining properties all in one package that capture things that are found in various other languages.

This is where you can give a collection of names and that collection of names makes a new type.

2.3 Disjoint Unions (slides 16-17)

Going beyond that, we can build disjoint Unions. We can take, what were previously different kinds of information and roll them up into one single type. This really comes in handy when you want to make something like a list, where the elements can be heterogeneous. That is, you can have elements, for example, of type double, or integer, or float. We have already learned that this is not directly possible in OCaml.

So, these type constructors, in OCaml allow you to roll your own type. It allows you to specify how much heterogeneity you want in your type. Then you can start putting these elements of different types into your own list. This method of type construction will allow you to take, for example, a copy of the integers and tag them (which in the slides are blue) and a copy of the floats (tagged yellow) and another copy of the integers (tagged red) and then you can add some singletons (single elements).

For example, we can create a type called id.

```
# type id = DriversLicense of int |  
        SocialSecurity of int | Name of string;;
```

¹© 2007, Share and Enjoy

```
type id = DriversLicense of int | SocialSecurity of int | Name of string
```

Now, this may be useful if you're working with something like databases. When you are entering a record into this database you may be allowed to put in any of the fields that are part of it. So you can enter a string, an integer, or another integer. And depending on how you tag it, it will accept the information as being a unique identifier for this person. The intuition is given by the tags. You can enter a drivers license, a social security number, or a name. Doing this allows you to create an id.

So if you were to enter:

```
let jim = Name "James Watson";;  
val jim = Name "James Watson"
```

This would give you an object that, under regular circumstances might simply be a string, now instead has been coerced to be of type id. So here, you can take an integer or a string and apply it to the appropriate constructor. Note that no values are assigned to the object jim for either the SocialSecurity or the DriversLicense. This is not a record, because it does not have all the fields, it is simply a variant so it only has one of the tags. This is where the disjoint part comes in. If an object is of type id, then it can only have one of the three tags. If it had more, it wouldn't be disjoint.

Then, you can also do checks on your values to make sure they fit what ever criteria you may have. So in this id example, we can check to see if the drivers license number is outside of that range, or that one's social security number is less than 900,000,000, and you can also check that the given name is not that of "John Doe", since that is used for people of unknown identity.

```
# let check_id ident = match ident with  
  DriversLicense num →  
    not (List.mem num [13570; 99999])  
  | SocialSecurity num → num < 900000000  
  | Name str → not (str = "John Doe");;  
val check_id : id → bool = <fun>
```

To do these checks, you pass an identifier in and do a match with it. During the matching, for driver's license, you do the match with the tag and then a variable name for the actual value. You can't just say match id with int → ...etc... Nor can you say match id with DriversLicense → ...etc because you are just giving the match statement a tag, but no underlying value there. You need the variable name to get at the actual information.

If you were to use the tag, Name, again, it would cause the previous use of Name to be hidden. Also, note that you can use the actual type names when doing functions on them. That is, in the example for pattern matching, the function took an argument of type id and it is called ident. However, like in the power point slides for this lecture, you can use id as the argument name as well. OCaml won't get confused.

As a final note, List.mem is another internal function in the OCaml list library. It tests for membership of an element in a list. So in the example, we are testing to see if DriversLicense num is inside the range of integers 13570 to 99999. And it can only be used when looking at some type that can be checked for equality.

3 Polymorphism in Variants

One thing that is allowed when writing types is that the types you write can be polymorphic. To see that, you have to give it a name for the type variable that it is going to be polymorphic over, or rather the type variables is going to be polymorphic over.

3.1 Option type (slide 18)

Here is an example of a polymorphic type. It is already written into OCaml, but you can very easily write it yourself. This type, 'a option, is incredibly useful.

```
# type 'a option = Some of 'a | None;;  
type 'a option = Some of 'a | None
```

'a option has two constructors and those constructors create a disjoint union of something of type 'a and one single element called None. So if you had 'a be int, then you could have a something like an int option that would be Some of (some integer, like 5 for example) 5 or it would be None.

When would this be useful? Well, let's say you were writing a function that needed something of an option type be returned. For example, like the mp problem for finding the smallest element in the list. If the list was empty, 0 was returned. And, now if you are returning an integer the function has to be over a list of integers. If the function was polymorphic, we could have had a list of strings and returned a the string with the smallest number of characters. But with the option type, None could be returned, signifying there was no element in the list that evaluated to being the smallest.

Another example is if you wanted to return the head of a list. If the list was empty, it would be better to return None so to keep the polymorphism of the function. If you were to instead return the empty string, or 0. Your function is now bound to a specific type. So if your function ran on a list of integers and you gave it a list of strings an exception would be thrown. And in fact, some of the exceptions thrown when programming OCaml can be diverted if option types were used instead.

Note that when doing this you don't have to simply input one type. Say you wanted to make some type that took in two types as starting points and were going to call it ty. You have ('a,'b) ty =... C of 'a list | D of ('a → 'b) So you are taking in two types and you have a constructor C that makes a list of 'a and a constructor D that is a function that takes something from type 'a to type 'b. Syntax of OCaml requires you to write the required arguments of your constructors out in front of the whole type itself. So, the "pair" written above can technically be any tuple, so as to supply the arguments needed inside the constructors.

3.2 Functions over the option type (slide 19)

So now, how can we use these disjoint Some/None pairs to write functions? Well, the answer is simply by using pattern matching.

The first is example is one where we create an option. We want to write a function that takes in a function p and a list and return the first element in the list that satisfies that function. We keep recursing down the list until we either find an element that works, and then we are done recursing, or we find the end of the list. If no element satisfies the function, we will return None.

```
# let rec first p list =
  match list with [] → None
  | (x::xs) → if p x then Some x else first p xs;;
val first : ('a → bool) → 'a list → 'a option = <fun>
```

Notice that since we are returning None instead of some value, like 0, our function can run on any type as long as the function takes in an argument of that type and the list is of that type.

So here in the examples our function is going to look for the first element that is greater than the integer 3. And we find 4 and return it. But we can't just return 4, we have to return Some 4 because it needs to have the same type of None, which in this case is an int option.

```
# first (fun x → x > 3) [1;3;4;2;5];;
- : int option = Some 4
```

Now we try the function again, but we are instead looking for the first element that is greater than 5. Well, there is no element in the list greater than 5, so None is returned.

```
# first (fun x → x > 5) [1;3;4;2;5];;
- : int option = None
```

Notice that this is a different problem than finding all the elements in a list that satisfy the function. Here, you wouldn't be returning the None if no elements were found, you would more likely, be returning the empty list. And then instead of simply returning Some x, when an element is found, you would be cons-ing the element onto the return list and continuing the recursion.

Now, let's say instead of just returning the value Some 4, we want to hold it. So we can easily write the code above to be:

```
# let z = first (fun x → x > 3) [1;3;4;2;5];;
- : val z = Some 4
```

And now z holds the value, Some 4. So if we wanted to, say, find the factorial of the returned value, we could take z and do a little pattern matching, like match z with Some n → fact n | None → ... And now the issue is, what to do in the None case. Well, you can leave the None case off, if you are positive the input will never be a None. If it is,

an exception will be thrown. Also, when trying to compile the code, OCaml will send a warning telling you that your pattern matching is not exhaustive, like not including the empty list in list matching.

3.3 Mapping over Variants (slide 20)

Now, let's look at sending in option types into functions as arguments. Here we will also see that the higher order functions used in list functions can also be used in general cases as well.

So, say we want to write a function that takes the value of an option and maps that value to a function, that is, applies a function to that value. So we send into the function an argument function and the option. The option is mapped to either None or Some. In the Some case, we extract the value and mapped that value to the value returned by the argument function after it has been applied to the value. That new value becomes part of an option that can be of the same type or a different type.

```
# let optionMap f opt =  
  match opt with None → None  
  | Some x → Some (f x);;  
val optionMap : ('a → 'b) → 'a option → 'b option = <fun>
```

So the type here is saying, we have a function optionMap that takes in a function that takes a value from type 'a and returns something of type 'b. optionMap also takes something of type 'a option. And then something of type 'b option is returned.

Notice that the same general thing is happening here as when map was performed on a list. In a list we extracted the element and performed some function on the element and put the element back into a list. Here we extract a value from a Some, do the mapping, and then put the element back into a Some. So the outer structure stays the same, but map ends up altering the components of the structure.

3.4 Folding over Variants (slide 21)

When using options, it doesn't really make sense to say you are folding left or folding right, because there is no left or right. There is just one element. Options are not recursive structures.

But, just like with lists, there are a few pieces of information you need to write such a function. First, what is the base case of this function. That is, what will be returned if the option is a None. Then you also want to know what you will do when you have Some, what function will be applied. Basically what will be happening here is you want to go from an option type to some other type completely, not just another option type.

To do this, you will need a function that will return some type and you will need a base case variable that will hold something of that same type that is to be returned during the base case.

```
# let optionFold someFun noneVal opt =  
  match opt with None → noneVal  
  | Some x → someFun x;;  
val optionFold : ('a → 'b) → 'b → 'a option → 'b = <fun>
```

So we are going to write our fold function that takes in three arguments. The first is a function, that takes in something of type 'a and returns something of type 'b, which will be used during the Some case. The next argument is something of type 'b that will be returned if we are in the None case. Finally, there is the option to be sent in. The option is of type 'a option. This makes sense, because during the Some case, we need to extract the value in the option and use it as the argument to the argument function. If the option value and the argument function's argument were of different types and exception would be thrown.

Folding can also be used to define Map. Your argument function for optionMap would be used in the argument function of optionFold. Your base case would be None, since map keeps the same type. And the option argument sent into optionMap would be the same for optionFold.

Another useful function to think about is a function that extracts the value from an option. Here the question is, what do you do if the option is None.

4 Recursive Types (slide 22)

We will discuss next time the creation of recursive types. The idea here is that you can have your component types of the type you are creating actually be the type you are creating.

For example, for lists you can write: `type ('a list) = Nil | Cons of ('a * ('a list))`. So we are using the type we are making as a component of itself.