# Parametric Polymorphism and Structural Induction

## Polymorphism - parametric and ad-hoc

So far, we mostly dealt with functions that operate on numbers, or larger structures built from numbers. There were a few occasions where we almost stumbled across a higher level of abstraction which is provided by all the languages of the ML family (and some more), and indeed, all the examples have been chosen in such a way as to navigate us around it. Not because the underlying concepts would be overly complex or difficult to master, but rather because they deserve being discussed separately.

While it is practically never necessary, one may explicitly specify the type of a variable rather than having it derived by OCaml's type inference. That is, one could define:

**Definitions with types specified explicitly**

```
# let square (x:int) = x*x;;
val square : int -> int = <fun>

(* Of course, this does not allow us to
   override the types derived by inference: *)

# let square (x:float) = x*x;;
Characters 24-25:
  let square (x:float) = x*x;;
                         ^
This expression has type float but is here used with type int
```

Sometimes, this is useful for providing more documentation in the code (what is it actually that I am passing around?), and sometimes, it helps discovering errors. Some cryptic error message may be resolved by putting extra type specifiers into the code to discover where the system's opinions on a type differ from ours. This may be especially useful in conjunction with higher order functions and forgotten arguments.

Actually, OCaml's close relative SML does not know separate sets of operators for integer and floatingpoint arithmetics. Rather than "*" and "*.", we only have "*", which defaults to integer addition. Should we want floatingpoint addition, we have to tell the system so by explicitly providing that type, unless this can be derived by using `*` in a context where one of the arguments is known to be floatingpoint anyway:

**Explicitly specified types in SML**

```
$ sml
Standard ML of New Jersey v110.42 [FLINT v1.5], October 16, 2002
- fun square_int x = x*x;;
val square_int = fn : int -> int
```

```
- fun square_float (x:real) = x*x;;
val square_float = fn : real -> real

- fun circle_area r = r*r*3.1415926;;
val circle_area = fn : real -> real
```

So, let us just define a function that swaps the entries of a pair of integers:

**Swapping a pair of integers**

```
# let swap_int_pair_v1 ((a:int),(b:int)) = (b,a);;
val swap_int_pair_v1 : int * int -> int * int = <fun>

# let swap_int_pair_v2 ((a,b):(int*int)) = (b,a);;
val swap_int_pair_v2 : int * int -> int * int = <fun>

# let swap_int_pair_v3 (a,b) = ((b:int),(a:int));;
val swap_int_pair_v3 : int * int -> int * int = <fun>

(* All of these definitions above are equivalent *)

# swap_pair_v1 (2,3);;
- : int * int = (3, 2)
```

How do we define a function that swaps a pair of a string and a boolean value?

**Swapping a `string * bool` pair**

```
# let swap_sb_pair ((a,b):(string * bool)) = (b,a);;
val swap_sb_pair : string * bool -> bool * string = <fun>

# swap_sb_pair ("hello",true);;
- : bool * string = (true, "hello")
```

Now, as the types tell us, we can use our integer pair swapping functions only on pairs of integers, and our string/bool pair swapping functions only on string/bool pairs. But what would we get if we just made a definition like:

**Swapping as a polymorphic function**

```
# let swap_pair (a,b) = (b,a);;
val swap_pair : 'a * 'b -> 'b * 'a = <fun>

# swap_pair (2,3);;
- : int * int = (3, 2)

# swap_pair (true, 5);;
- : int * bool = (5, true)

# swap_pair ((1,2),"three");;
- : string * (int * int) = ("three", (1, 2))
```

If we ignore the type of this `swap_pair` function for now, we at least note that if we make the definition without explicitly referring to a type, then we get a variant that can be used on all sorts of pairs. Actually, while there are pairs of many different kinds, all we have to know in order to swap the entries is that we are dealing with a pair, *any* pair. We do not really have to look into the contents of that "container" data structure for this particular operation: this indeed also holds from the machine perspective: the very same sequence of machine instructions will swap any pair. This idea of "I do not have to look into it, but know how to do it just from the structure" has far-reaching consequences and goes by the name of *parametric polymorphism*.

OCaml has a funny way to express the type of a polymorphic function: it will use so-called `type jokers`. In mathematics, one conventionally uses Greek letters for type variables, and this funny apostrophe-letter notation is supposed to be read in that way, "`'a`" denoting "alpha", "`'b`" denoting "beta", and so on. But actually, OCaml sticks to using the Latin alphabet `'a` to `'z`, then `'a1` to `'z1`, and so on, if there is need for really really many jokers. Usually, there is not. Should we ever have to specify a type joker of our own, we can use an arbitrary name that is formed like a variable name, and prepend a "`'`" character. So, we may use meaningful joker names such as `'position` if we want to.

Whenever an expression whose type is polymorphic (that is, contains one or more type jokers) is used in some given context, OCaml checks whether it is possible to specialize the jokers in an appropriate way to match that context. If this succeeds, the original expression behaves as if it had the specialized type right from the beginning. Note that this check already occurs at compile time, not only at run time, and this is the reason why the OCaml language is called "statically typed".

There are some slight over-simplifications in that description of type inference, and actually, it is a bit difficult to describe it in a completely correct way without resorting to some mathematical formalism. However, one should know that it is easy to get used to its behaviour and develop an intuition for it by just playing around with it for some time. In a certain sense, this is very "natural".

We presumably just need a few more examples:

**More examples for parametric polymorphism**

```
# let make_twin_pair x = (x,x);;
val make_twin_pair : 'a -> 'a * 'a = <fun>

# let make_triplet_aab a b = (a,a,b);;
val make_triplet_aab : 'a -> 'b -> 'a * 'a * 'b = <fun>

# let rotate_triplet (a,b,c) = (c,a,b);;
val rotate_triplet : 'a * 'b * 'c -> 'c * 'a * 'b = <fun>

# let make_triplet_baa a b = rotate_triplet (make_triplet_aab a b);;
val make_triplet_baa : 'a -> 'b -> 'b * 'a * 'a = <fun>

# let left_of_pair (p,q) = p;;
val left_of_pair : 'a * 'b -> 'a = <fun>

# let identity x = x;;
val identity : 'a -> 'a = <fun>
```

```
# let apply_twice f x = f(f x);;
val apply_twice : ('a -> 'a) -> 'a -> 'a = <fun>

# apply_twice (fun x -> x+2) 6;;
- : int = 10

# apply_twice (fun x -> x*.8.0) 6.0;;
- : float = 384.

# apply_twice (fun (p,q) -> (p,q+10)) (true,5);;
- : bool * int = (true, 25)

# let compose f g = fun x -> f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# let constantly x = fun y -> x;;
val constantly : 'a -> 'b -> 'a = <fun>

# let c3 = constantly 3 in c3 100;;
- : int = 3

(* Clearly, the empty list has to be polymorphic as well *)

# [];;
- : 'a list = []

(* Many of the functions from the standard library are polymorphic *)

# List.length;;
- : 'a list -> int = <fun>

# List.length [1;10;100];;
- : int = 3

# List.length [0.5;0.7];;
- : int = 2

# Array.make;;
- : int -> 'a -> 'a array = <fun>

# Array.make 5 true;;
- : bool array = [|true; true; true; true; true|]

# Array.make 7 "dwarf";;
- : string array =
[|"dwarf"; "dwarf"; "dwarf"; "dwarf"; "dwarf"; "dwarf"; "dwarf"|]

(* This especially holds for most of the really useful tools:
   They are so useful because they express an abstract idea that
   can be applied in a broad class of situations.
*)

# Array.map;;
- : ('a -> 'b) -> 'a array -> 'b array = <fun>

# Array.map (fun x -> x*2) [|1;2;3;4;5|];;
- : int array = [|2; 4; 6; 8; 10|]
```

```
# Array.map (fun x -> (x,x)) [|1;2;3;4;5|];;
- : (int * int) array = [|(1, 1); (2, 2); (3, 3); (4, 4); (5, 5)|]

# let array_add_2 = Array.map (fun x -> x+2) in
   (array_add_2,array_add_2 [|1;2;3;4;5|]);;
- : (int array -> int array) * int array = (<fun>, [|3; 4; 5; 6; 7|])
```

To make things a bit more complicated, there are some functions and operators in OCaml which appear in the guise of being parametrically polymorphic, but in fact are not - far from it. The most prominent examples are the comparison operators: evidently, just comparing two integers should be in some sense much simpler than lexicographically comparing complex composite values. So, OCaml is cheating a bit here. Even worse, it pretends that something like equality comparisons existed for functions. It is important to know about these cases of mundane ad-hoc polymorphism in the guise of parametric polymorphism. Actually, the examples are quite few, and the distinction does not matter too much in most applications, but it is important to know it's there.

## Examples of "Fake" parametric polymorphism in OCaml

```
# let is_smaller a b = a < b;;
val is_smaller : 'a -> 'a -> bool = <fun>

# let is_equal a b =  a=b;;
val is_equal : 'a -> 'a -> bool = <fun>

# is_equal (fun x -> x*2+2) (fun x -> (x+1)*2);;
Exception: Invalid_argument "equal: functional value".

(* Note that this raises an exception at run time while actually,
   the notion of "equality" cannot naturally be extended to functions,
   and hence, equality cannot be parametrically polymorphic. Still,
   OCaml claims so *)

(* Some other exotic functions are even worse sinners, such as: *)

# output_value;;
- : out_channel -> 'a -> unit = <fun>
# stdout;;
- : out_channel =
# output_value stdout (2,3);;
\204\225¦¾^@^@^@^C^@^@^@^A^@^@^@^C^@^@^@^C BC- : unit = ()

(* This is for the serialization of values.
   We may even crash OCaml if we are careless with that: *)

# Marshal.to_string (2,3) [];;
- : string =
"\132\149\166\190\000\000\000\003\000\000\000\001\000\000\000\003\000\000\000\003\160BC"

# let x = Marshal.to_string (2,3) [] in
   let (y:(bool*string)) = Marshal.from_string x 0
   in y;;
```

```
Process inferior-caml segmentation fault
```

It is interesting to see how polymorphism interacts with abstracting out some sub-functionality from a given function:

**Why parametric polymorphism is important - an Example**

```
let pointwise_add f g = fun x -> (f x) + (g x);;

let pointwise_multiply f g = fun x -> (f x) * (g x);;

let pointwise_max f g = fun x -> max (f x) (g x);;

let f0 = pointwise_max (fun x -> 0) (fun x -> 10-x*x);;

val f0 : int -> int = <fun>

# f0 3;;
- : int = 1

# f0 2;;
- : int = 6

# f0 5;;
- : int = 0

(* Now, pointwise this and pointwise that is always almost the same,
so we may just extract that out: *)

# let pointwise c f g x = c (f x) (g x);;
val pointwise :
 ('a -> 'b -> 'c) -> ('d -> 'a) -> ('d -> 'b) -> 'd -> 'c = <fun>

(* That type actually is quite a bit complicated. Let us re-write it
   in an equivalent form. We may always re-name type jokers, so
   we do it like this:

   'a => 'fx,  'b => 'gx, 'c => 'fx_c_gx, 'd => 'x.

   Then, this reads:

   ('fx -> 'gx -> 'fx_c_gx) -> ('x -> 'fx) -> ('x -> 'gx) -> 'x -> fx_c_gx

*)

# let test_pointwise = pointwise (fun x y -> x+y) (fun x -> x*1000) (fun x -> x*10);;
val test_pointwise : int -> int = <fun>

# test_pointwise 3;;
- : int = 3030

# let pointwise_add_v2 = pointwise (fun x y -> x+y);;
val pointwise_add_v2 : ('_a -> int) -> ('_a -> int) -> '_a -> int = <fun>

(* Pay close attention to the strange '_a type joker.
   Whenever something like this shows up, bad things are about to happen.
   This usually occurs when we globally define a function by partial
   application of a polymorphic function. If we give that function
   its required extra parameters, which we just pass on to the right
   hand side, everything is fine:
*)

# let pointwise_add_v3 f g x= pointwise (fun x y -> x+y) f g x;;
val pointwise_add_v3 : ('a -> int) -> ('a -> int) -> 'a -> int = <fun>
```

Note that by looking at what's common among "pointwise sum", "pointwise product", and other similar functions, and abstracting out that which is special for every single case, we were able to find a function that just represents the abstract idea behind "doing something pointwise". The definition of the function `pointwise` is interesting: it does not involve any numbers or tuples or anything else - just functions. All there is to it is which function's value is passed on into which other function, and how values are distributed between functions. Such a function is called a *combinator*. There is quite a lot of mathematical theory on combinators. Two other (much simpler) combinators, which we already have seen in this lecture, are `identity` and `constantly`, and indeed, one can have a lot of fun with those three only.

In a certain sense, one may say that polymorphism just kind of solves the same problem that C++ claims to solve via templates. At least, pretty much everything that is in the C++ Standard Template Library (STL) is just parametrically polymorphic. However, the C++ approach does come with a few serious flaws. To see this, let us look at the following example:

**Parametric Polymorphism and C++**

```
// File pair.cpp

#include <iostream>

using namespace std;

template<class T1,class T2> pair<T2,T1> swap(pair<T1,T2> p)
{
  return pair<T2,T1>::pair(p.second,p.first);
}

int main()
{
  pair<int*,int*> pp1 = pair<int*,int*>::pair(0,0);

  pair<int*,int*> spp1 = swap(pp1);

  cout <<"(" << (unsigned long)(pp1.first) << ", "
             << (unsigned long)(pp1.second) << ")" << "\n";

  cout <<"(" << (unsigned long)(spp1.first) << ", "
             << (unsigned long)(spp1.second) << ")" << "\n";

  pair<char*,char*> pp2 = pair<char*,char*>::pair(0,0);
  pair<char*,char*> spp2 = swap(pp2);

  cout <<"(" << (unsigned long)(pp2.first) << ", "
            << (unsigned long)(pp2.second) << ")" << "\n";

  cout <<"(" << (unsigned long)(spp2.first) << ", "
            << (unsigned long)(spp2.second) << ")" << "\n";


  return 0;
}
```

## Analyzing the binary

```
$ g++ -O3 -o pair pair.cpp

$ ls -la pair
-rwxr-xr-x  1 tf tf 14025 Dec 14 02:26 pair

$ ./pair
(0, 0)
(0, 0)
(0, 0)
(0, 0)

$ nm pair|grep swap
08048920 W _Z4swapIPcS0_ESt4pairIT0_T_ES1_IS3_S2_E
08048950 W _Z4swapIPiS0_ESt4pairIT0_T_ES1_IS3_S2_E

$ objdump -d pair

pair:     file format elf32-i386

(...)

08048920 <_Z4swapIPcS0_ESt4pairIT0_T_ES1_IS3_S2_E>:
 8048920:       55                      push   %ebp
 8048921:       89 e5                   mov    %esp,%ebp
 8048923:       83 ec 08                sub    $0x8,%esp
 8048926:       8b 4d 10                mov    0x10(%ebp),%ecx
 8048929:       8b 45 08                mov    0x8(%ebp),%eax
 804892c:       8b 55 0c                mov    0xc(%ebp),%edx
 804892f:       89 4d fc                mov    %ecx,0xfffffffc(%ebp)
 8048932:       8b 4d fc                mov    0xfffffffc(%ebp),%ecx
 8048935:       89 55 f8                mov    %edx,0xfffffff8(%ebp)
 8048938:       89 08                   mov    %ecx,(%eax)
 804893a:       8b 4d f8                mov    0xfffffff8(%ebp),%ecx
 804893d:       89 48 04                mov    %ecx,0x4(%eax)
 8048940:       89 ec                   mov    %ebp,%esp
 8048942:       5d                      pop    %ebp
 8048943:       c2 04 00                ret    $0x4
 8048946:       90                      nop
 8048947:       90                      nop
 8048948:       90                      nop
 8048949:       90                      nop
 804894a:       90                      nop
 804894b:       90                      nop
 804894c:       90                      nop
 804894d:       90                      nop
 804894e:       90                      nop
 804894f:       90                      nop

08048950 <_Z4swapIPiS0_ESt4pairIT0_T_ES1_IS3_S2_E>:
 8048950:       55                      push   %ebp
 8048951:       89 e5                   mov    %esp,%ebp
 8048953:       83 ec 08                sub    $0x8,%esp
 8048956:       8b 4d 10                mov    0x10(%ebp),%ecx
 8048959:       8b 45 08                mov    0x8(%ebp),%eax
 804895c:       8b 55 0c                mov    0xc(%ebp),%edx
 804895f:       89 4d fc                mov    %ecx,0xfffffffc(%ebp)
 8048962:       8b 4d fc                mov    0xfffffffc(%ebp),%ecx
 8048965:       89 55 f8                mov    %edx,0xfffffff8(%ebp)
 8048968:       89 08                   mov    %ecx,(%eax)
 804896a:       8b 4d f8                mov    0xfffffff8(%ebp),%ecx
 804896d:       89 48 04                mov    %ecx,0x4(%eax)
 8048970:       89 ec                   mov    %ebp,%esp
 8048972:       5d                      pop    %ebp
 8048973:       c2 04 00                ret    $0x4
 8048976:       90                      nop
 8048977:       90                      nop
 8048978:       90                      nop
 8048979:       90                      nop
 804897a:       90                      nop
 804897b:       90                      nop
 804897c:       90                      nop
 804897d:       90                      nop
 804897e:       90                      nop
 804897f:       90                      nop

(...)

$
```

If we compile this and analyze the binary with utilities such as `nm` and `objdump`, we see that the compiler indeed did generate two different swap functions. However, the code is precisely the same, even up to the number of padding NOP instructions in both cases! So, indeed, at least the GNU C++ compiler (maybe others as well) has the potential to generate great code bloat when it comes to expanding templates. Actually, this was even much much worse with earlier versions of C++ compilers, where template instantiation meant that *every* single method and function that was defined for that particular template was expanded into code - whether it was actually used or not. Even nowadays, one can still find references to the claim that all complaints about C++ templates generating code bloat are out-dated, as C++ compilers will no longer compile "dead methods" that never get called into the binary. That may well be so by now, but still, C++ templates have a tendency to introduce quite ugly unnecessary code bloat through repetitive expansion: there really is no difference between swapping a pair of pointers to integers, or swapping a pair of pointers to whatever else, so there is no reason at all why the same code should be generated multiple times. (Actually, having the same code more than once is not especially nice to instruction caches and therefore CPU memory bandwidth, as well as dynamic branch prediction.) Of course, some template authors are aware of this problem and design their templates in such a way that the parametrically polymorphic aspects are only compiled once, and every instantiation of that template is just a front-end to the really parametric code. But this inevitably requires some ugly casts, which templates were designed to prevent. So, if we see it that way, then templates split the C++ programmers into two classes: template users which never are supposed to use casts, and proficient template implementors, who may and must use casts to deal with parametric polymorphism. A very unsatisfactory state of affairs, actually.

To be fair, the C++ template idea is somewhat more general. If we swapped a pair of a pointer and something larger, the code would be different, as the pair container would normally truly contain a copy of its entries, and not just reference them. While C++ templates are used in perhaps the overwhelming majority of cases to deal with parametric polymorphism, they are general enough to also handle ad-hoc polymorphism. Whether this actually is a good thing, in particular as this kind of prevents us to handle parametric polymorphism in a really nice and elegant way in C++, is up to the audience to decide.

## Structural Induction

One of the basic ideas of object-oriented programming is that objects are entities which satisfy certain contracts that are specified through their classes, which themselves may form a hierarchy of abstraction built on more generic contracts, base classes. Looking in the opposite direction, one can always specialize and subclass a given class.

So, from the object-oriented point of view, it would be natural to treat the natural numbers, that is the numbers `1,2,...` without any artificial and arbitrary machine limit - as objects in a class which we may call, say, `NaturalNumbers`.

However, this approach would be conceptually flawed. The reason is that through subclassing, one could always introduce a new class of entities which technically speaking also are `NaruralNumbers`, through a `is-a` inheritance relationship, but which are not in the numbers `1,2,3,...`. Mathematicians like to define the natural numbers as follows:

1. Zero is a natural number.
2. Every natural number has a successor, which again is a natural number.
3. Zero is not the successor of any natural number.
4. If two numbers have the same successor, they are the same.
5. Nothing else is a natural number. So, the natural numbers are the "smallest" structure that satisfies the above rules.

The important point here is the "*nothing else*" statement. In fact, this is equivalent to the principle of induction over natural numbers. So, if we drop that statement, as we inevitably would do by modeling natural numbers in a class, we lose the power to do stringent reasoning over the natural numbers.

This may be even clearer with boolean values: there just are two of them: `true` and `false`. No one would ever think about sub-classing a `Boolean` class in order to implement extensions to boolean values, say `most likely`, `presumably`, `presumably not` and such. All hell would break loose if values like `presumably` crept up in a program expecting a boolean value to be either `true` or `false`.

So, there is a conceptual difference between the idea of a class, which roughly says: "I do not really know in detail what it is that I am dealing with, but it promised me to provide a particular interface, and as long as it does this, I am fine", and the idea of mathematical exclusive definitions which are made for stringent reasoning over all possible cases. Each has its place in this world, and if possible, one should perhaps not try to use the one as a substitute for the other. If a system supports exclusive definitions, it usually can also check automatically whether the case analysis is complete; when using classes for the same task, this certainly cannot be possible, as they are built for extensibility: they are abstract contracts which may be satisfied by a lot of things, and we may not know all the possibilities from the beginning. On the other hand, when using exclusive definitions instead of classes, we may end up in a situation where we eventually find out that actually, we should consider more cases than the ones we provided initially. But then, we would have to make changes all over the place: wherever we go through all the possibilities in our code, we have to add a further one.

So much about the theory. Now, let us see how this actually looks like in OCaml.

**Defining new types**

```
(* A poker example *)

# type suit = Clubs | Diamonds | Hearts | Spades;;
type suit = Clubs | Diamonds | Hearts | Spades

# type color = Red | Black;;
type color = Red | Black

#
let suit_color_v1 s =
  if s = Clubs || s = Spades then Black
  else Red
;;

val suit_color_v1 : suit -> color = <fun>

#
let suit_color_v2 s =
  match s with
  | Clubs -> Black
  | Spades -> Black
  | Diamonds -> Red
  | Hearts -> Red
;;

val suit_color_v2 : suit -> color = <fun>

#
let suit_color_v3 s =
  match s with
  | Clubs -> Black
  | Spades -> Black
  | Diamonds -> Red
;;

          Characters 26-97:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Hearts
  ..match s with
```

```
    | Clubs -> Black
    | Spades -> Black
    | Diamonds -> Red
val suit_color_v3 : suit -> color = <fun>

#
let suit_color_v4 s =
  match s with
  | Clubs -> Black
  | Spades -> Black
  | x -> Red
;;

val suit_color_v4 : suit -> color = <fun>
```

First, we introduce a so-called "variant" type "`suit`", which has four constructors: `Clubs`, `Diamonds`, `Hearts`, `Spades`. One may think of this as a four-valued equivalent of the two-valued `bool` type, which has "constructors" `true` and `false`. As a general rule, the names of all constructors must begin with an uppercase letter.

The `suit` data type is mostly equivalent to a C-style `enum`, as those constructors just represent four different values. The major difference to an `enum` is type safety: a C `enum` value is just a number. Our constructors cannot be accidentally confused with numbers. So, we may then proceed likewise to define a cards' color, which is `Red` or `Black`. There is an obvious function mapping a card's suit to its color. We can either define this the old-fashioned way using `if ... then ... else`, or we can instead use the matching operator, which allows us to just list all the cases. Conceptually, this seems to be similar to a C `switch/case`, but actually, this will turn out to be far more powerful. The syntax may require some getting used to, but is quite suggestive: we just list all the cases, one after another, with the pattern at the left hand side. Note that we may also use variables at the left hand side. Then, the corresponding right hand side case is used when the left hand side structure can be made to match the argument of `match` by binding variables appropriately. Here, this is quite trivial in the `suit_color_v4` example: we just use a variable to match "any remaining cases". Note that `x` does not appear at the right hand side. With more complicated patterns, it may. There is a convention to use the special name "_" (underscore) for such "don't care" variables.

Constructors play the role of "hooks" that hold data and allow us to define structural patterns on the left hand side of a matching rule. This gets much more interesting with constructors that carry arguments:

**Constructors with arguments**

```
type card =
    Joker
  | Ace   of suit
  | King  of suit
  | Queen of suit
  | Jack  of suit
  | Number of suit * int
;;

type hand = card list;;
```

```
(* This is just a type alias. "hand" and "card list"
   can be used in a fully interchangeable way. *)

let some_hand=[Ace Spades;
               Queen Hearts;
               Number (Hearts,5);
               Number (Clubs,2);
               Joker
             ];;

(* Defining a function on such a type *)

let blackjack_value card =
  match card with
  | Joker -> failwith "Not used in blackjack!"
  | Ace _ -> 11 (* or 1, but let's ignore this for now *)
  | Number (_,n) -> n
  | _ -> 10 (* King, Queen, Jack *)
;;
```

Note how the structured value to the left of a matching rule provides a pattern against which the argument to `match` is checked. When more than one rule could match, the first one is taken. We also see this here with the default rule. If we moved this last line to the first line, every card's value would be "`10`".

Things get even more interesting with recursive type definitions. We may use the type which we are defining right now inside the argument type definition of a constructor to define hierarchically structured values. This looks as follows:

**Recursive variant types**

```
type boolean_term =
 | Bool_Constant of bool
 | Bool_Var of string
 | Bool_And of boolean_term * boolean_term
 | Bool_Or of boolean_term * boolean_term
;;

let some_term =
  Bool_And(Bool_Constant true,
           Bool_Or (Bool_Var "x",
                    Bool_And(Bool_Var "y",Bool_Var "z")));;

(* Defining a function on such terms: *)

let rec term_contains_variable_with_name v bterm =
  match bterm with
  | Bool_Constant _ -> false
  | Bool_Var x -> x=v
  | Bool_And (t1,t2) ->
     term_contains_variable_with_name v t1 ||
     term_contains_variable_with_name v t2
  | Bool_Or (t1,t2) ->
     term_contains_variable_with_name v t1 ||
     term_contains_variable_with_name v t2
```

```
;;

(* Example:
# term_contains_variable_with_name "z" some_term;;
- : bool = true

# term_contains_variable_with_name "w" some_term;;
- : bool = false
*)
```

Furthermore, variant types may be polymorphically parametric. We already have encountered parametric recursive types: the list is the prime example. Let us just re-invent the list with an own definition to see how polymorphic variant types are introduced and how they behave.

**Polymorphic recursive variant types**

```
type 'a mylist =
  | Empty
  | Head_Tail of 'a * ('a mylist)
;;

let some_mylist = Head_Tail (10,Head_Tail(20, Head_Tail(30,Empty)));;

(* A naive definition of a length function: *)

let rec len_mylist li =
  match li with
  | Empty -> 0
  | Head_Tail (_,tl) -> 1 + len_mylist tl
;;

(*
The type is:

- : 'a mylist -> int = <fun>

Example:

# len_mylist some_mylist;;
- : int = 3
*)

(* Variant types also can be polymorphic in more than one way: *)

type ('a,'b) mytree =
    Leaf of 'a
  | Fruit of 'b
  | Branch of ('a,'b) mytree * ('a,'b) mytree
;;

let some_tree = Branch (Leaf 5, Fruit "xyz");;
(* This is:
val some_tree : (int, string) mytree = Branch (Leaf 5, Fruit "xyz")
*)
```

## Records

Another kind of data structure provided by OCaml which will become very important in programs that are less mathematically inclined, but have to deal with a lot of bookkeeping, is the record. This is roughly equivalent to a C struct. Actually, with all the background we have developed so far, there is not too much to be said about records. They are defined and anonymously constructed as follows:

**Records**

```
type file =
    {name: string;
     access_flags: int;
     size: int;
     creation_time: int;
   };;

let some_file=
  {name="/etc/passwd";
   access_flags= 6*8*8 + 4*8 + 4; (* mode 644 *)
   size=1733;
   creation_time=0;
 };;

let s = some_file.size;;

type 'num arithmetics =
    {addition: 'num -> 'num -> 'num;
     multiplication: 'num -> 'num -> 'num;
     zero: 'num;
    }
;;

let bool_arith =
  {addition=(fun a b -> a || b);
   multiplication=(fun a b -> a && b);
   zero=false;
 }
;;

let generic_scalar_product arith v1 v2 =
  let add = arith.addition in
  let mult = arith.multiplication in
  let zero = arith.zero in
  let len = Array.length v1 in
  if Array.length v2 <> len
  then failwith "Scalar product arrays must have same length!"
  else
    let rec sum_up sum_now pos =
      if pos = len then sum_now
      else let contrib = mult v1.(pos) v2.(pos) in
      sum_up (add contrib sum_now) (1+pos)
    in sum_up zero 0
;;
```

```
(*
This has type:

# val generic_scalar_product :
  'a arithmetics -> 'a array -> 'a array -> 'a = <fun>

Actually, one would in practice usually prefer to
use the OCaml module system to get such functionality.
(Not discussed here.)
*)

generic_scalar_product bool_arith [|false;false;true|] [|false;true;false|];;
(* Gives "false" *)
```

One should know about records that it is not permitted to have two records defined in the same OCaml module which both have an entry with the same name. This is quite unfortunate, and considered a pretty stupid design bug of OCaml. Again, the reason is type inference, the idea being that by just mentioning a single slot, the system should be able to derive the type of the record.

This concludes our discussion of data types in OCaml for this crash course. Beginners usually need some practice to get used to programming by pattern matching and case analysis. The exercises will give ample opportunity for this.

## Exercises

1. Define a variant type that represents a binary tree with a value in every node.
2. Define a function on the tree of the previous exercise that returns its maximal depth.
3. Define another function on this tree data structure that collects the data values of all its nodes in a list.
4. Define a function "`scan_list property list what_if_not_found`" that scans a list for the first element on which the function `property` is true, producing the value `what_if_not_found` if no such element was found, test it, and explain its type. Hint: use `ocamlbrowser` to look up the definitions of `List.hd` and `List.tl` to see how to use the list constructor "`::`" in pattern matching.
5. Define a function `eval_bterm term var_occupations` that takes as arguments a `boolean_term` (as defined in the text) as well as a list of pairs `(variable_name,boolean_value)` of variable occupations, and produces the boolean value (either `true` or `false`) represented by this term which one gets by substituting the variables with the values provided and reducing afterwards.
6. Define a function that takes as argument a list and produces a pair of two lists, one containing the elements with even index in the original list, the other one containing the elements with odd index.
7. Look up the documentation of `List.map` and `List.map2` and define a corresponding function `array_map2` on arrays. (Hint: use `Array.init`)
8. Define a function that takes as arguments a comparison function as well as two lists, which can be

assumed to be ordered with respect to this comparison function, and produces a joint list containing all the elements from both lists, also ordered with respect to the comparison function.

9. Use the functions from the last two exercises to define your own list sorting function.

10. Define a function that reverses a list. Note: if the runtime of your function scales quadratically with the length of the list, you missed something obvious. Hint: What do you get if you take a stack of cards and repetitively remove the topmost card and add it to another stack?

11. Define a function mapping a `boolean_term` to a `string list` of all the variable names in it, without repetition.

12. Define a function that tries to find an occupation of variables for which a given `boolean_term` is true. Note: this is not required to be overly efficient. (Again, there would be an entire mathematical theory behind that!)

13. (Difficult!) As briefly mentioned, there is an entire mathematical theory behind combinators. Show that it is possible to complete the following definition in such a way that the result behaves just like the composition function, where the expression that goes in the place of `???` is built exclusively out of the functions `pointwise`, `identity`, and `constantly` (as well as parentheses, of course):

**Some Combinatory Logic**

```
let my_composition f g = ??? f g;;
```

---

*Dr. Thomas Fischbacher*
Last modified: Sat May 13 18:26:01 BST 2006