Inductively Defined Types

OCaml supports the definition of more interesting data types than just those defined by cases.

For example, we can define a data type called `nat` that corresponds to *natural numerals in unary representation* (Peano style). The two constructors are `Z` (for zero) and `S` (for successor. [Note: in OCaml, constructors have to start with an upper-case letter).
We define `nat` as a recursively-defined type.

```
type nat = Z | S of nat;;
```

Note that the constructor `S` takes a `nat` value as an argument, whereas the base case `Z` takes no argument. The canonical values of the type `nat` are `Z, (S Z), S (S Z), S(S (S Z))` and so on. The type `nat` thus represents the inductively-defined (i.e., the smallest) set *closed* under the constructors applications. Mathematically, one should consider this type to be a solution to the equation $N = \mathbf{1}\ {}_Z{+}_S\ N$, the least solution of which is a denumerable set.

Giving names to values is useful when you program. So OCaml's *definition mechanism* is handy.

```
let zero = Z;;
let one = S Z;;
let two = S (S Z);;
let three = S (S (S Z));;
```

[Note: You should also get used to writing constructor and function applications as e.g., `(f x)` and `(S Z)` instead of `f(x)` and `S(Z)`.]

Let us now define a function from the inductively defined type `nat`. The main language features we shall use are case analysis, which is supported via *pattern-matching* with respect to the constructors, and *recursion*, which is the flip side of induction. First, just case analysis (as before), but note that we can use `_` to indicate "don't-care" as the argument in the pattern `S _`.

```
let nonzero n = match n with
    Z -> false
  | S _ -> true
;;
```

The earlier definitions come in handy when testing out the program: a good *definition mechanism* for a language allows one to use the defined name instead of the expression to which it has been defined equal in all contexts.

```
nonzero zero;;
nonzero three;;
```

Now let us see more concretely how *recursion* is the flip side of induction. We present a recursively defined function `nat2int` from `nat` to the OCaml built-in type `int`. The keyword **`rec`** following **`let`** indicates that the function being defined is recursive. In the definition, (in the inductive case) there are patterns of the form `S x`, which contain a variable `x` (which in OCaml must begin with a lower-case letter) that will feature in the argument to a recursive call to the defined function `nat2int`. The pattern-matching mechanism will match the pattern `S x` to the argument and *bind* the variable `x` to a component of the argument value, and use this bound value in the right-hand side expression. This binding of `x` is temporary and *local* to only this case.

```
let rec nat2int n = match n with
    Z -> 0
  | S x -> 1+(nat2int x)
;;

nat2int zero;;
nat2int one;;
nat2int two;;
nat2int three;;
```

Note that the function `nat2int` acts as a *semantic* function that provides meanings in the OCaml type `int` to "syntactic" elements which are our unary numerals in the type `nat`.

While OCaml provides support for very general recursion, a particularly useful form is the notion of "*primitive recursion*" which you will encounter in Theory of Computation. Roughly speaking, in primitive recursion, the recursive calls are of a restricted format, and are on strictly smaller terms, and so one can easily show that the program will terminate.

```
let rec addnat m n = match m with
    Z -> n   (* 0+n = n *)
  | S x -> S (addnat x n)   (*  (1+x) + n = 1 + (x+n)  *)
;;

addnat zero three;;
addnat three zero;;
addnat one two;;
addnat two one;;
```

Proving Correctness of (Primitive) Recursive Programs

Let us prove that function `addnat` is correct with respect to the semantics expressed by function `nat2int`.

```
forall m: nat, forall n: nat,
   nat2int (add m n) = (nat2int m) + (nat2int n)
```

Proof: By induction on the structure of `m: nat`

Base case `(m=Z)`

```
nat2int (addnat Z n)
```

```
   = nat2int n   // defn of addnat
   = 0 + (nat2int n)  // 0 is the left identity of +
   = (nat2int Z) + (nat2int n)  // defn of nat2int <-
```

Induction Hypothesis: Suppose that for `m = k we have`

```
    forall n: nat,
        nat2int (addnat k n) = (nat2int k) + (nat2int n)
```

Induction Step

Let `m = (S k)`

```
     nat2int (addnat (S k) n)
   = nat2int (S (addnat k n)   // defn of addnat
   = 1 + (nat2int (addnat k n)  // defn of nat2int
   = 1 + ((nat2int k) + (nat2int n) )  // IH on m=k
   = (1 + (nat2int k)) + (nat2int n)   // assoc of +
   = (nat2int (S k)) + (nat2int n)   // defn of nat2int <-
```

Therefore by Simple structural induction on `m`

```
  forall m: nat, forall n: nat,
        nat2int (addnat m n) = (nat2int m) + (nat2int n)
```

Exercise: State and prove that `Z` is the (i) left identity of `addnat;` (ii) right identity of `addnat.`

Exercise: State and prove that `addnat` is commutative.

Exercise: State and prove that `addnat` is associative.

Let us now define another primitive recursive function. i.e., multiplication, assuming that `addnat` is correct.

```
let rec multnat m n = match m with
    Z -> Z  (* 0 * n = 0 *)
  | S x -> addnat n (multnat x n)  (* (1+x) * n = n + (x*n)
*)
;;

multnat zero two;;
multnat two zero;;
multnat one three;;
multnat three one;;
multnat three two;;
```

Assuming that `addnat` is correct, prove that `multnat` is correct.

```
forall m: nat, forall n: nat
   nat2int (multnat m n) = (nat2int m) * (nat2int n)
```

Proof: By induction on the structure of `m: nat.`

Base case (m=Z)

```
    nat2int (multnat Z n)
  = nat2int Z   (* defn of multnat *)
  = 0  (* defn of nat2int *)
```

Induction Hypothesis: Assume that for m=*k*, *forall* n:nat,

nat2int (multnat *k* n) = (nat2int *k*) * (nat2int n)

Induction Step (m = S *k*)

```
     nat2int (multnat (S k) n)
  =  nat2int (addnat n (multnat k n) // defn of multnat
  =  (nat2int n) + (nat2int (multnat k n)) // correctness of addnat
  = (nat2int n) + ((nat2int k) * (nat2int n)) // IH on m=k
  = (1 + (nat2int k)) * (nat2int n)   // right distribution of * over + <-
  = (nat2int (S k)) * (nat2int n) // defn of nat2int <-
```

Exercise: State and prove that Z is a (i) left annihilator for multnat; (ii) right annihilator for multnat.

Exercise: State and prove that (S Z) is (i) a left identity for multnat; (ii) a right identity for multnat.

Exercise: State and prove that multnat is commutative.

Exercise: State and prove that multnat is associative.

Exercise: State and prove that multnat distributes left and right over addnat.

If we make a convenient assumption that $0^0 = 1$ rather than undefined, we can define exponentiation as a primitive recursive function

```
let rec expnat m n = match n with
    Z -> (S Z)    (* m^0 = 1 *)
  | S x -> multnat m (expnat m x)    (* m^(1+x) = m * (m^x) *)
;;

expnat zero one;;
expnat zero zero;;
expnat zero three;;
expnat three zero;;
expnat two three;;
expnat two one;;
expnat three two;;
```

Exercise: State and prove the correctness of expnat.

Exercise: State and prove that `(S Z)` is the right identity for `expnat`.

Exercise: Prove that *forall* `m: nat,` *forall* `n: nat,` *forall* `x: nat,`

```
expnat x (addnat m n) = multnat (expnat x m) (expnat x n)
```

Lists

OCaml supports the definition of a generic type constructions such as lists over any type. That is, for any type, we have a uniform way of building lists with elements of that type. Note however, that all elements of a given list must have the *same* type, that is one cannot have a mixed list with say integers and booleans.

A lot of reasoning about lists does not concern itself with the type of the list elements. This kind of genericity is called *"Parametric Polymorphism"*.

Lists are a built-in polymorphic type in OCaml. However, one can imagine that someone must have made a parametric type definition of the form

```
type 'a list = Nil | Cons of 'a * ('a list)
```

for two constructors traditionally called `Nil` and `Cons`.
The polymorphic type is `'a list`, where `'a` stands for *any* type. *Type variables* are written by putting a quote mark before an identifier beginning with a lower-case letter. It is customary to read the "quote-a" as "alpha", "quote-b" as "beta", etc. to highlight that these are type variables.

[Mathematically, lists are the least fixed-point solution to a recursive type equation $L_\alpha = \mathbf{1}\ {}_{Nil}{+}_{Cons}\ (\alpha \times L_\alpha)$ for any type α.

OCaml interpreters come with a built-in List module which has predefined values and functions over lists. To use a values and functions in a module we refer to them using a dot notation, e.g. `List.append`. However, by "opening" the module so we can use its definitions freely, without qualifying them each time with the module name..

```
open List;;
```

There is a more intuitive way of writing the `Nil` constructor.

```
[ ];;   (* The Nil constructor *)
```

The `Cons` constructor can be thought of taking a pair — an element from a type α and a list of type α list. This constructor is asymmetric in the two arguments, one is an element of type `'a` and the other is a list of elements of that type, an `'a list`. So it is not like a monoid operator. Note also that we can only "`Cons`" an element to the *front* of a list, and this is a constant-time operation.

```
1 :: [ ];;

1 :: (2 :: []);;
```

```
1 :: 2 :: [];;
```

It is more convenient and visually intuitive to write lists using semicolons as separators:

```
[1];;

[1; 2];;
```

List construction works at every type.

```
[T; F];;

[ [1; 2; 3]; [1; 3; 5; 7]; [] ];;
```

We can even have lists of functions (of the same type). However, as noted above, we cannot mix types of elements when forming a list. That is *not* what polymorphism supports.

There are two standard "projection" partial functions that help us in *deconstructing* lists:

```
hd;;
tl;;

(*
One can imagine someone had defined these OCaml functions:
   let hd (Cons (x, xs)) = x;;
   let tl (Cons (x, xs)) = xs

*)
```

These functions apply to lists of all types. However, we get an exception if we apply either of them to an empty list.

```
hd [];;
tl [];;
```

Likewise, the length of a list is a predefined function. It does not depend on the type of the list's elements.

```
(* Length of a list *)
length;;

(* imagine someone had defined this function by recursion as:
   let rec length l = match l with
     [ ] -> 0
   | _ :: xs -> 1 + (length xs)
;;
*)

length [ ];;
length [1;2;3];;
```

Two lists of the same type can be concatenated to return a single list. The original lists are unchanged; a new list is created, and the elements of the first list appear in order before those of the second list.

```
append;;

(* imagine someone had defined a recursive function

  let rec append l1 l2 = match l1 with
      [ ] -> l2
    | x::xs -> x :: (append xs l2)
   ;;
*)

append [ ] [1;2;3];;
append [1;2;3] [ ];;
```

If one worked with the above imagined definition of append, one could do the following (we imagine the implementor of the List library did so).

Exercise: State and prove that `[ ]` is the left and right identity element for `append`

```
append [1] (append [2] [3]);;
append (append [1] [2]) [3];;
```

Exercise: State and prove that `append` is associative.

Exercise: Prove that appending two lists yields a list whose length is the sum of the lengths of the input lists:

```
forall l1: 'a list, forall l2: 'a list,
    length (append l1 l2) = (length l1) + (length l2)
```

It is common to use the operator _ `@` _ as an infix version of `append`.

Note that _ `::` _ ("cons") is a constant time operation, whereas `append` involves a function call. So never write `[1] @ [2;3;4]` but instead write `1 :: [2; 3; 4]`. However, since one can only prepend (cons) an *element* at the front of a list, if we have to place an element at the end of a list, we may have to use append.

Consider the code to reverse a (polymorphic) list (There already is a `List.rev` function).

```
List.rev [3; 2; 1];;


let rec rev s = match s with
        [ ] -> [ ]
   |   x::xs ->  (rev xs) @ [x]
;;

rev [1;2 3];;
```

Cons would not work, since the element is being placed at the *end* of the list.

This code is quite inefficient (you can see that its complexity is quadratic in the size of the list). So one can use a technique that uses an auxiliary tail-recursive definition `rev2` and redefine `rev`. (The `let … in` construct localises the definition of `rev2`.)

```
let rev s =
    let rec rev2 s1 s2 = match s1 with
       [ ] -> s2
     | x::xs ->  rev2 xs (x::s2)
    in
       rev2 s [ ]
;;

rev [1;2; 3];;
```