Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 4

Recursive functions and recursive types

Topics covered:

- Kinds of recursion
- Numbers as a recursive type
- New types in ML
- Pattern matching
- More examples: sums, lists and trees.

Recursive functions: factorial

Recursive functions are central to functional programming, so it's as well to be clear about them.

Roughly speaking, a recursive function is one 'defined in terms of itself'. For example, we can define the factorial function in mathematics as

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

This translates directly into ML:

```
- fun fact n =
    if n = 0 then 1
    else n * fact(n - 1);
> val fact = fn : int -> int
- fact 6;
> val it = 720 : int
```

Recursive functions: Fibonacci

Another classic example of a function defined recursively is the n^{th} member of the Fibonacci sequence $1, 1, 2, 3, 5, 8, 13, 21, \ldots$ where each number is the sum of the two previous ones.

$$fib_n = \begin{cases} 1 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib_{n-2} + fib_{n-1} & \text{otherwise} \end{cases}$$

Once again the ML is similar:

```
- fun fib n =
    if n = 0 then 1
    else if n = 1 then 1
    else fib(n - 2) + fib(n - 1);
> val fib = fn : int -> int
- fib 5;
> val it = 8 : int
- fib 6;
> val it = 13 : int
```

Kinds of recursion

How do we know that the evaluation of these functions will terminate?

Trivially `fact 0` terminates, since it doesn't generate a recursive call.

If we evaluate `fact n` for $n > 0$, we need `fact (n - 1)`, then maybe `fact (n - 2)`, `fact (n - 3)` etc., but eventually, after `n` recursive calls, we reach the base case. This is why termination is guaranteed. (Though it loops for $n < 0$.)

This sort of recursion, where the argument to the recursive call(s) decreases by 1 each time is called *primitive* recursion. The function `fib` is different: the recursion is not primitive.

To know that `fib n` terminates, we need to know that `fib (n - 1)` and `fib (n - 2)` terminate. Nevertheless, we are still sure to reach a base case eventually because the argument does become smaller, and can't skip over both 1 and 0.

Proofs of termination

More formally, we can turn the above into a *proof* by mathematical induction than `fact n` terminates for each natural number n. We prove that `fact 0` terminates, then that if `fact n` terminates, so does `fact (n + 1)`.

$$\forall P.\, P(0) \wedge (\forall n.\, P(n) \Rightarrow P(n+1)) \Rightarrow \forall n.\, P(n)$$

The appropriate way to prove `fib n` terminates for all natural numbers `n` is to use the principle of *wellfounded induction*, rather than step-by-step induction.

$$\forall P. (\forall n. (\forall m. m < n \Rightarrow P(m)) \Rightarrow P(n) \Rightarrow \forall n. P(n)$$

There is thus a close parallel between the kind of *recursion* used to define a function and the kind of *induction* used to reason about it, in this case show that it terminates.

The naturals as a recursive type

The principle of mathematical induction says exactly that every natural number is generated by starting with 0 and repeatedly adding one, i.e. applying the successor operation $S(n) = n + 1$.

If we regard the natural numbers as a set or a type, then we may say that it is generated by the *constructors* 0 and S.

Moreover, each natural number can only be generated in one way like this: we can't have $S(n) = 0$, and if

$$\overbrace{S(S(\cdots(S(0))\cdots))}^{p\text{ times}} = \overbrace{S(S(\cdots(S(0))\cdots))}^{q\text{ times}}$$

then $p = q$. The second property is equivalent to saying that S is *injective*.

In such cases the set or type is said to be *free*, because there are no relationships forced on the elements.

New types in ML

ML allows us to define new types in just this way. We write:

```
- datatype num = 0
               | S of num;
> datatype num
  con 0 = 0 : num
  con S = fn : num -> num
```

This declares a completely new type called `num` and the appropriate new constructors.

But in order to define functions like `fact` we need to be able to take numbers apart again, i.e. go from `S(n)` to `n`. We haven't got something like subtraction here.

Properties of type constructors

All type constructors arising from a `datatype` definition have three key properties, which we can illustrate using the above example.

1. They are exhaustive, . every element of the new type is obtainable either by `O` or as `S x` for some `x`.
2. They are injective, i.e. an equality test `S x = S y` is true if and only if `x = y`.
3. They are distinct, i.e. their ranges are disjoint. More concretely this means in the above example that `S(x) = O` is false whatever `x` might be.

Because of these properties, we can define functions, including recursive ones, by *pattern matching*.

Pattern matching — how

We perform pattern matching by using more general expressions called *varstructs* as the arguments in `fn => ...` or `fun => ...` expressions.

Moreover, we can have several different cases to match against, separated by `|`. For example, here is a test for whether something of type `num` is zero:

```
- fun iszero O = true
    | iszero (S n) = false;
> val iszero = fn : num -> bool
- iszero (S(S(O)));
> val it = false : bool
- iszero O;
> val it = true : bool
```

This function has the property, naturally enough, that when applied to `O` it returns `true` and when applied to `S x` it returns `false`.

Pattern matching — why

Why is this valid?

1. The constructors are *distinct*, so we know that there is no ambiguity. The cases for `0` and `S x` don't overlap.

2. The constructors are injective, so we can always recover `x` from `S x` if we want to use `x` in the body of that clause.

3. The constructors are *exhaustive*, so we know that if we have a case for each constructor, the function is defined everywhere on the type.

Non-exhaustive matching

In fact, we can define partial functions that don't cover every case. Here is a 'predecessor' function.

```
- fun pred (S(n)) = n;
! Toplevel input:
! fun pred (S(n)) = n;
!     ^^^^^^^^^^^^^^^^
! Warning: pattern matching is not
           exhaustive

> val pred = fn : num -> num
```

The compiler warns us of this fact. If we try to use the function on an argument not of the form `S x`, then it will not work:

```
- pred O;
! Uncaught exception:
! Match
```

General matching

Moreover, we can perform matching even in other situations, when the matches might not be mutually exclusive. In this case, the first possible match is taken.

```
- (fn true => 1 | false => 0) (4 < 3);
> val it = 0 : int
- (fn true => 1 | false => 0) (2 < 4);
> val it = 1 : int
```

However, in general, constants need special constructor status, or they will be treated just as variables for binding:

```
- let val t = true and f = false
   in (fn t => 1 | f => 0) (4 < 3)
  end;
! .....

> val it = 1 : int
```

Nonrecursive types

New types don't actually need to be recursive. For example, here is a type of disjoint sums.

```
- datatype ('a,'b)sum = inl of 'a
                      | inr of 'b;
> datatype ('a, 'b) sum
  con inl = fn : 'a -> ('a, 'b) sum
  con inr = fn : 'b -> ('a, 'b) sum
```

This creates a new type *constructor* `sum` and two new constructors. Again we can define functions by pattern matching, e.g.

```
- fun outl (inl a) = a;
! Toplevel input:
! fun outl (inl a) = a;
!     ^^^^^^^^^^^^^^^^^
! Warning: pattern matching is not
           exhaustive

> val outl = fn : ('a, 'b) sum -> 'a
```

Lists (1)

An important type is the type of finite lists:

```
- datatype ('a)list =
      Nil
    | Cons of 'a * ('a)list;
> datatype 'a list
  con Nil = Nil : 'a list
  con Cons = fn : 'a * 'a list -> 'a list
```

We imagine `Nil` as the empty list and `Cons` as a function that adds a new element on the front of a list. The lists $[]$, $[1]$, $[1,2]$ and $[1,2,3]$ are written:

```
Nil;
Cons(1,Nil);
Cons(1,Cons(2,Nil));
Cons(1,Cons(2,Cons(3,Nil)));
```

Lists (2)

Actually, this type is already built in. The empty list is written `[]` and the recursive constructor `::`, has infix status. (You can make your own identifier `f` infix by writing `infixr f`.) Thus, the above lists are actually written:

```
- [];
> val it = [] : 'a list
- 1::[];
> val it = [1] : int list
- 1::2::[];
> val it = [1, 2] : int list
- 1::2::3::[];
> val it = [1, 2, 3] : int list
```

The version that is printed can also be used for input:

```
- [1,2,3,4,5] = 1::2::3::4::5::[];
> val it = true : bool
```

Pattern matching over lists

We can now define functions by pattern matching in the usual way. For example, we can define functions to take the head and tail of a list:

```
- fun hd (h::t) = h;
! Toplevel input:
! fun hd (h::t) = h;
!     ^^^^^^^^^^^^^^
! Warning: pattern matching is not
           exhaustive
> val hd = fn : 'a list -> 'a
- fun tl (h::t) = t;
! Toplevel input:
! fun tl (h::t) = t;
!     ^^^^^^^^^^^^^^
! Warning: pattern matching is not
           exhaustive
> val tl = fn : 'a list -> 'a list
```

ML warns us that they will fail when applied to an empty list.

Recursive functions over lists

It is possible to mix pattern matching and recursion. This is natural since the type itself is defined recursively. For example, here is a function to return the length of a list:

```
- fun length [] = 0
    | length (h::t) = 1 + length t;
> val length = fn : 'a list -> int
- length [5,3,1];
> val it = 3 : int
```

Alternatively, this can be written in terms of our earlier 'destructor' functions `hd` and `tl`. This style of function definition is more usual in many languages, notably LISP, but the direct use of pattern matching is often more elegant.

Trees

Lists can be though of as tree structures, but are rather 'one-sided'. Here is a type of binary trees with integers at the branch nodes:

```
- datatype tree =
    Leaf
  | Br of (tree*int*tree);
> datatype tree
  con Leaf = Leaf : tree
  con Br = fn : tree * int * tree -> tree
```

For example, the following recursive function adds up all the integers in a tree:

```
- fun treesum Leaf = 0
    | treesum (Br(t1,n,t2)) =
        treesum t1 + n + treesum t2;
> val treesum = fn : tree -> int
```

Such tree structures are often useful for representing the syntax of formal languages, e.g. arithmetic expressions, C programs.

The subtlety of recursive types

Consider the following:

```
- datatype ('a)embedding =
    K of ('a)embedding->'a;
```

This looks suspicious because it embeds the function space $A \to B$ inside A. In fact it only embeds the *computable* functions. It allows us to define recursive functions without explicit use of recursion:

```
- fun Y h =
    let fun g (K x) z = h (x (K x)) z
     in g (K g)
    end;
- val fact = Y (fn f => fn n =>
      if n = 0 then 1 else n * f(n - 1));
> val fact = fn : int -> int
- fact 6;
> val it = 720 : int
```