Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 6

Effective ML

Topics covered:

- Using standard combinators
- Abstract types
- Tail recursion and accumulators
- Forcing evaluation
- Minimizing consing

Using standard combinators

It often turns out that a few *combinators* are very useful: we can implement practically anything by plugging them together, especially given higher order functions.

For example, the `itlist` function:

$$\begin{aligned}&\text{itlist } f\ [x_1,\ x_2,\ \ldots\ ,x_n]\ b\\&= f\ x_1\ (f\ x_2\ (f\ x_3\ (\cdots(f\ x_n\ b))))\end{aligned}$$

can often be used to avoid explicit recursive definitions over lists. We define it as:

```
- fun itlist f [] b = b
    | itlist f (h::t) b =
        f h (itlist f t b);
> val itlist = fn : ('a -> ('b -> 'b)) ->
                      'a list -> 'b -> 'b
```

Functions of this sort are often built in to functional langauges, and sometimes called something like `fold`.

Itlist examples (1)

For example, here is a function to add up all the elements of a list:

```
- fun sum l =
        itlist (fn x => fn sum => x + sum)
                   l 0;
> val sum = fn : int list -> int
- sum [1,2,3,4,5];
> val it = 15 : int
- sum [];
> val it = 0 : int
```

If we want to multiply the elements instead, we change it to:

```
- fun prod l =
  itlist (fn x => fn prod => x * prod)
             l 1;
> val prod = fn : int list -> int
- prod [1,2,3,4,5];
> val it = 120 : int
```

Itlist examples (2)

Here is a filtering function:

```
- fun filter p l = itlist
    (fn x => fn s =>
     if p x then x::s else s) l [];
```

and here are some logical operations over lists:

```
- fun forall p l = itlist
     (fn h => fn a => p(h) andalso a)
     l true;
- fun exists p l = itlist
     (fn h => fn a => p(h) orelse a)
     l false;
```

and some old favourites:

```
- fun length l =
     itlist (fn x => fn s => s + 1) l 1;
- fun append l m =
     itlist (fn h => fn t => h::t) l m;
- fun map f l = itlist
     (fn x => fn s => (f x)::s) l [];
```

Itlist examples (3)

We can implement set operations quite directly using these combinators as building blocks.

```
- fun mem x l =
      exists (fn y => y = x) l;
- fun insert x l =
     if mem x l then l else x::l;
- fun union l1 l2 =
      itlist insert l1 l2;
- fun setify l =
      union l [];
- fun Union l =
      itlist union l [];
- fun intersect l1 l2 =
      filter (fn x => mem x l2) l1;
- fun subtract l1 l2 =
      filter (fn x => not (mem x l2)) l1;
- fun subset l1 l2 =
      forall (fn t => mem t l2) l1;
```

Abstract types

An abstract type starts with an ordinary recursive type, and then imposes restrictions on how the objects of the type can be manipulated.

Essentially, users can only interact by a particular 'interface' of functions and values.

The advantage is that users *cannot* rely on any other internal details of the type.

This improves modularity, e.g. it is easy to replace the implementation of the abstract type with a new 'better' (smaller, faster, not patented ...) one, without requiring any changes to user code.

Example: sets (1)

Here is an abstract type for integer sets.

```
- abstype set = Set of int list with
    val empty = Set []
    fun set_insert a (Set s) =
      Set (insert a s)
    fun set_union (Set s) (Set t) =
      Set (union s t)
    fun set_intersect (Set s) (Set t) =
      Set (intersect s t)
    fun set_subset (Set s) (Set t) =
      subset s t
    end;
```

Users can't access the internal representation (lists), which is only done via the interface functions.

Example: sets (2)

Many of the operations are much more efficient if we keep the lists sorted into numerical order. For example:

```
- fun sunion [] [] = []
    | sunion [] l = l
    | sunion l [] = l
    | sunion (h1::t1) (h2::t2) =
        if h1 < h2 then
          h1::(sunion t1 (h2::t2))
        else if h1 > h2 then
          h2::(sunion (h1::t1) t2)
        else h1::(sunion t1 t2);
> val sunion = fn : int list ->
                    int list -> int list
```

We can change the internal representation to use sorted lists (or balanced trees, or arrays, ...) and no changes are needed to code using the abstract type.

Storing local variables

Recall our definition of the factorial:

```
#fun fact n = if n = 0 then 1
              else n * fact(n - 1);
```

A call to `fact 6` causes another call to `fact 5` (and beyond), but the computer needs to save the old value `6` in order to do the final multiplication.

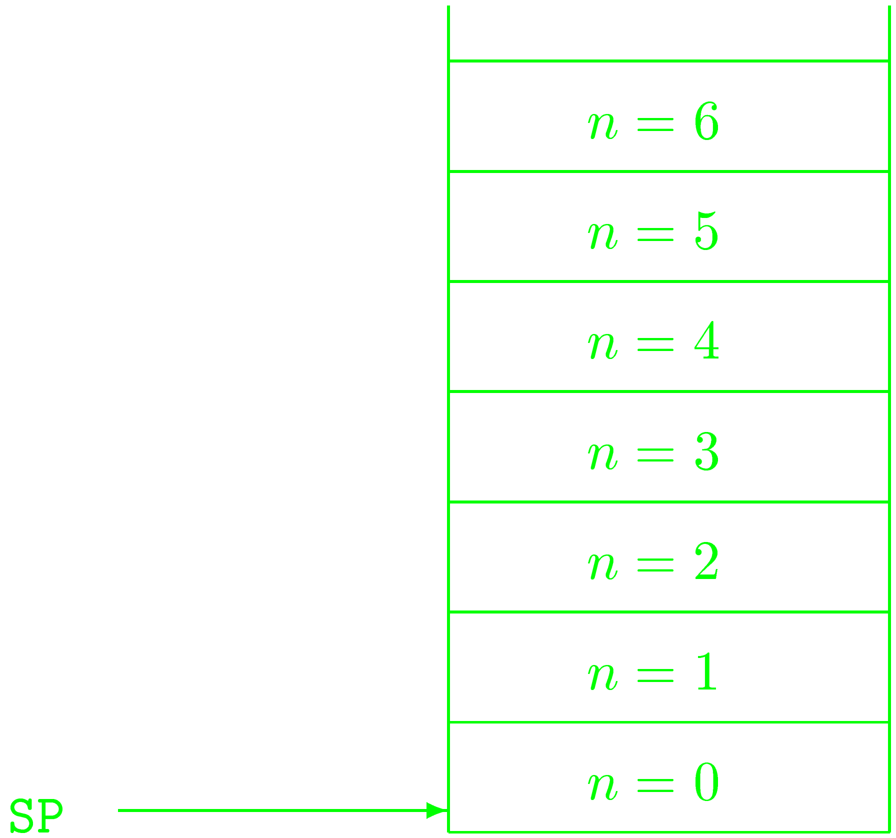
Therefore, the local variables of a function, in this case `n`, cannot be stored in a fixed place, because each instance of the function needs its own copy.

Instead each instance of the function is allocated a frame on a *stack*.

This technique is similar in ML to most other languages that support recursion, including C.

The stack

Here is an imaginary snapshot of the stack during the evaluation of the innermost call of `fact`:



Note that we use about n words of storage when there are n nested recursive calls. In many situations this is very wasteful.

A tail recursive version

Now, by contrast, consider the following implementation of the factorial function:

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

The recursive call is the whole expression; it does not occur as a proper subexpression of some other expression involving values of variables.

Such a call is said to be a *tail call* because it is the very last thing the calling function does.

A function where all recursive calls are tail calls is said to be *tail recursive*.

Why tail recursion?

If a function is tail recursive, the compiler is clever enough to realize that the same area of storage can be used for the local variables of each instance.

We avoid using all that stack.

The additional argument `x` of the `tfact` function is called an *accumulator*, because it accumulates the result as the recursive calls stack up, and is then returned at the end.

Working in this way, rather than modifying the return value on the way back up, is a common way of making functions tail recursive.

In a sense, a tail recursive implementation using accumulators corresponds to an iterative version using assignments and while-loops.

The only difference is that we pass the state around explicitly.

Forcing evaluation (1)

Recall that ML does not evaluate inside lambdas. Therefore it sometimes pays to pull expressions outside a lambda when they do not depend on the value of the bound variable.

For example, we might code the efficient factorial by making `tfact` local:

```
  - fun fact n =
        let fun tfact x n =
              if n = 0 then x
              else tfact (x * n) (n - 1)
         in tfact 1 n
        end;
```

However the binding of the recursive function to `tfact` does not get evaluated until `fact` sees its arguments.

Moreover, it gets reevaluated each time even though it doesn't depend on `n`.

Forcing evaluation (2)

A better coding is as follows:

```
- val fact =
      let fun tfact x n =
            if n = 0 then x
            else tfact (x * n) (n - 1)
       in tfact 1
      end;
```

In cases where the subexpression involves much more evaluation, the difference can be spectacular.

Most compilers do not do such optimizations automatically.

However it falls under the general heading of *partial evaluation*, a big research field.

Forcing evaluation (3)

An alternative coding is to use `local`:

A better coding is as follows:

```
- local fun tfact x n =
              if n = 0 then x
              else tfact (x * n) (n - 1)
      in fun fact n = tfact 1 n
     end;
> val fact = fn : int -> int
```

The local declaration is invisible outside the body of `fact`.

Minimizing consing (1)

The space used by type constructors ('cons cells') is not allocated and deallocated in such as straightforward way as stack space.

In general, it is difficult to work out when a certain cons cell is in use, and when it is available for recycling. For example in:

```
val l = 1::[] in tl l;
```

the cons cell can be reused immediately. However if `l` is passed to other functions, it is impossible to decide at compile-time when the cons cell is no longer needed.

Therefore, space in functional languages has to be reclaimed by analyzing memory periodically and deciding which bits are needed. The remainder is then reclaimed. This process is known as *garbage collection*.

Minimizing consing (2)

We can often make programs more space and time efficient by reducing consing. One simple trick is to reduce the usage of `append`. By looking at the definition:

```
- fun append [] l = l
    | append (h::t) l = h::(append t l);
> val append = fn :
      'a list -> 'a list -> 'a list
```

we can see that it generates n cons cells where n is the length of the first argument. For example, this implementation of reversal:

```
- fun rev [] = []
    | rev (h::t) = append (rev t) [h];
> val rev = fn : 'a list -> 'a list
```

is very inefficient, generating about $n^2/2$ cons cells, where n is the length of the list.

Minimizing consing (3)

A far better version is:

```
- local fun reverse [] acc = acc
          | reverse (h::t) acc =
                reverse t (h::acc)
    in fun rev l = reverse l []
   end;
> val rev = fn : 'a list -> 'a list
```

This only generates n cons cells, and has the additional merit of being tail recursive, so we save stack space.

One can also avoid consing in pattern-matching by using `as`, e.g. instead of rebuilding a cons cell:

```
fn [] => []
  | (h::t) => if h < 0 then t else h::t;
```

using

```
fn [] => []
  | (l as h::t) => if h < 0 then t else l;
```