1 Functional Programming and ML [part 2]

1.1 Higher order functions (revisited)

Higher order function (also known as **functional** or **functor**) is a function that does at least one of the following:

- takes one or more functions as arguments
- returns a function as result

`map` is a common example of higher order function. It applies a function to every element of a list.

```
f : A -> B
l : List of A
map : (A -> B) -> List of A -> List of B
```

Given the built-in `size` function from `string` to `int` which returns the length of a string, here is an example of application of map:

```
- val names = ["Oslo", "Norway", "uio"];
- size;
val it = fn : string -> int
- map size names;
```

Question. What do you expect as the output from the example above?

Here is an example of higher order (i.e. map) and an anonymous function:

```
- map (fn x => x*x) [1,2,3];
```

Question. What do you expect as the output from the example above?

Another common example of higher order function is `filter`. It applies a predicate to every element of a list.

```
- fun even x = (x div 2) * 2 = x;
- fun filter p nil = nil
    | filter p (x::xs) = if p(x) then x :: (filter p xs)
                         else filter p xs;
- filter even [1, 2, 3, 4, 5];
```

Exercise. Run `filter even` on `[]`, `[1]`, and `[1,2,3]`.

- Which clause of filter will match `[]`?
- Which clause of filter will match `[1]`?
- If an input matches the second clause, what part of the input matches `x` and what matches `xs`?

1.2 Currying and uncurrying (revisited)

The following two functions are isomorphic:

```
- fun fa x y = x + y;      (* curried *)
- fun fb (x,y) = x + y;    (* uncurried *)
```

Currying is the technique of translating a function that takes multiple arguments into a sequence of functions, each with a single argument.

Curried functions allow for **partial application** as shown below:

```
- fun f x y = x + y;
- val add1 = f 1;
- add1 3;
- f 3 1;
```

Informally, here is what happens when the code above runs: Calling `f 1` substitutes `x` by `1` in `f` (this is represented by the first step of execution `~>`). Then, the anonymous function that takes an argument `y` and adds `1` to it is returned (this is represented by the second step of execution `~>`).

```
f 1 ~> f 1 y = 1 + y ~> y => 1 + y
```

We can define a **higher order function** that transforms a curried function into an uncurry'ed function (can also define another higher order function that does the reverse).

```
- fun curry f x y = f (x, y);
val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

Question. Who can explain `curry`'s type?

`curry` is a higher order function. We can interpret the type of `curry` as follows:

- Say `x` has type `'a` and `y` has type `'b`.
- then `(x,y)` has type `('a * 'b)`.
- In the definition of curry, it applies `f` to `(x,y)` and returns something (say of type `'c`), thus `f` must have type `'a * 'b -> 'c`.
- Since `curry` takes `f`, `x`, and `y)` as arguments, then curry has type `( 'a * 'b -> 'c`) `-> 'a * 'b -> 'c`.

Similarly, we can uncurry a function:

```
- fun uncurry f (x,y) = f x y;
val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

1.3 Recursion

(Example from *The little MLer*)

```
- datatype shish_kebab =
  Skewer
  | Onion of shish_kebab
  | Lamb of shish_kebab
  | Tomato of shish_kebab;

- Skewer;
- Onion(Skewer);
- Lamb(Onion(Skewer));
```

```
- fun is_vegetarian(Skewer)    = true
    | is_vegetarian(Onion(x))  = is_vegetarian(x)
    | is_vegetarian(Lamb(x))   = false
    | is_vegetarian(Tomato(x)) = is_vegetarian(x);
```

```
- is_vegetarian( Onion(Tomato(Skewer)) )
- is_vegetarian( Onion(Tomato(Lamb(Skewer))) )
```

1.3.1 Tail recursion

Question. What are the differences between these two implementations of the factorial function `fact` and `fact2`?

```
- open IntInf;
```

```
- fun fact 0 = 1
    | fact n = n * fact (n-1);
```

```
fun factr (0, a) = a
  | factr (n, a) = factr (n-1, n*a);

fun fact2 n = factr (n, 1);
```

As opposed to having `factr` defined at the top-level, note that we could have folded the definition of `factr` inside the definition of `fact2` by using the `let`-construct with `rec`:

```
fun fact2 n =
  let val rec factr = fn (0,a) => a
                       | (n,a) => factr(n-1,n*a);
  in factr (n ,1) end;
```

Tail call: A call to a function `f` is a tail call if it returns the value returned by `f` without performing any further computation.

```
fun g(x) = if x = 0 then f(x) else f(x) * 2
```

In the example above, the call to `f` in the then-branch is a tail call, while the call to `f` on the else-branch is not.

Tail recursive function is a function that is recursive and the recursive calls are tail calls. Tail recursion can be rewritten as iteration.

In theory, tail recursive functions are faster because the same activation record (stack frame) can be reused for each recursive call, thus saving the time to setup/teardown activation records (stack frames) for each call.

In practice, `fact` is slightly faster than `fact2` when running on the SML/NJ compiler.

Surprisingly, if we swap `n*a` by `a*n` in `fact2` it runs **much slower.** Multiplication is not symmetric with respect to time: multiplying a small number by a large one is faster than a large one by a small.

1.4 State in ML: cells

In functional programming, there is an effort to separate "side-effectful" from pure expressions. In ML:

- assignment are restricted to reference cells
- cells have a different type, separation of mutable/immutable enforced by type checker

L-value: The location of a variable (aka cell) is called its L-value. L for *left.*

R-value: The value stored in a variable (cell) is called its R-value. R for *right.*

```
- x := 10;
```

Here we create a cell with initial value of 0, note that the type of `x` is not `int` but `int ref`:

```
- val x = ref 0;
val x = ref 0 : int ref
```

Question. What happens when we run the code below?

```
- x + 1;
```

The code above does not type check: we cannot add an `int` to an `int ref`.

Question. What does the code below do?

```
- !x + 1;
```

It "dereferences" x and adds 1 to the value of x.

Assignment: We can assign to a reference cell with `:=`. Assignment "doesn't return anything" other than `unit`. `unit` is both the type of expressions that cause side effects as well as the value returned when those expressions run.

```
- x := 10;
val it = () : unit
- x := !x + 1;
val it = () : unit
```

Programs have no access to a reference cell's address.

```
- val y = ref "Apple";
val y = ref "Apple" : string ref
- y := "Fried green tomatoes";
val it = () : unit
```

The storage requirement for the string`Apple` is less than for the string `Fried green tomatoes`. When assigning `Fried green tomatoes` to `y`, the compiler can simply make `y` point to the memory location of the longer string.

1.5 Abstraction

Question. What is abstraction? Why is it useful?

1. Hide details (encapsulation)
2. Tame complexity
3. Allow for code reuse
4. Communicate the relative importance of things

   etc

Abstraction can be accomplished without language support; for example, by how we think of a problem (or algorithm) at the level of diagrams or of pseudo code.

In this section we talk about **language abstraction,** meaning, abstractions supported at the programming language level.

Language support for abstraction can be broken down into categories:

- Control flow abstraction
  - `if-then-else` as opposed to `goto`
  - functions
  - continuations

- Data abstraction
  - data types
  - for example, number in binary (two's complement, binary-coded decimal, etc) versus integers
- Syntactic abstraction
  - macro systems and meta-programming features.
  - makes it easier to grow a language or to implement domain specific languages (DSLs)

Refinement: the opposite of abstraction is refinement.

- By "opposite" we do not mean that because abstraction is "good," refinement must be "bad."
- Refinement starts at the abstract and works towards the concrete.
- It is very useful, for example, when generating programs from specification.

Next we will look at data abstraction.

1.5.1 Data abstraction

Decouple the "usage of" from the "representation of" a data structure.

- Usage: interface
- Representation: implementation

Two main forms of data abstraction:

- Abstract data types
- Modules

Other lectures also talked about object orientation (not covered here).

1.5.1.1 Abstract Data Types (ADTs)

(Abstract Data Type or ADT, not to be confused with Algebraic Data Type which is also ADT).

ADTs were first proposed by Barbara Liskov and Stephen N. Zilles in 1974, as part of the development of the CLU language.

An ADT is a type plus a set of operations on its values.

- The underlying data structure(s) that support(s) the ADT are not directly accessible.
- The ADT is manipulated with operations (ie. an interface).

Representation-independence

In ML, one early way of defining an abstract data type was with the `abstype` construct.

The presence of `abstype` in ML, however, is "historic." Its use has been supplanted by *signatures* and *structures,* which are part of the ML module system.

Today, ADTs are often implemented as modules.

1.5.1.2 Modules

Can use ADTs to define for example stacks, queues, trees, maps, lists, etc. Can use a module to bundle related data types together.

```
stacks, queues, trees, maps, lists -> collection
```

Structures

- An ML structure is a module, which is a collection of type, value, and structure declarations.

Signatures

Signatures are module interfaces.

- A module may have more than one signature and a signature may have more than one associated module.
- If a structure satisfies the description given by a signature, the structure "matches" the signature.

Functors

- Functors are functions from structures to structures.

Example: Signature definition (interface).

```
signature POINT =
 sig
   type point
   val mk_point : real * real -> point  (*constructor*)
   val x_coord : point -> real             (*selector*)
   val y_coord : point -> real             (*selector*)
   val move_p : point * real * real -> point
 end;
```

Structure definition (Implementation)

```
structure pt : POINT =
 struct
   type point = real * real
   fun mk_point(x,y) = (x,y)
   fun x_coord(x,y) = x
   fun y_coord(x,y) = y
```

```
   fun move_p((x,y):point,dx,dy) = (x+dx, y+dy)
 end;
```

To be able to use the implementation:

```
- open pt;
```

```
- val pt = mk_point(10.0,10.0);
- y_coord(pt);
- move_p (pt, ~1.0, 2.0);
- val pt2 = mk_point(10.0,10.0);
- pt1 + pt2;    (*addition not defined on points*)
```

See http://www.smlnj.org/doc/basis/pages/sml-std-basis.html for an overview of the structures and signatures in The Standard ML Basis Library. Follow the link: Top-level Environment to see which functions are available in the top level environment, i.e. which you can use without prefixes.