Higher-Order Functions

Sven-Olof Nyström

Functional Programming 1

Original slides by Tjark Weber based on notes by Pierre Flener, Jean-Noël Monette, Sven-Olof Nyström

Today

1. Definition, Introductory Examples
2. Higher-Order Functions on Lists
3. Application: Polymorphic Ordered Binary Tree
4. Higher-Order Functions on Trees

Table of Contents

Higher-Order Functions

A higher-order function is a function that

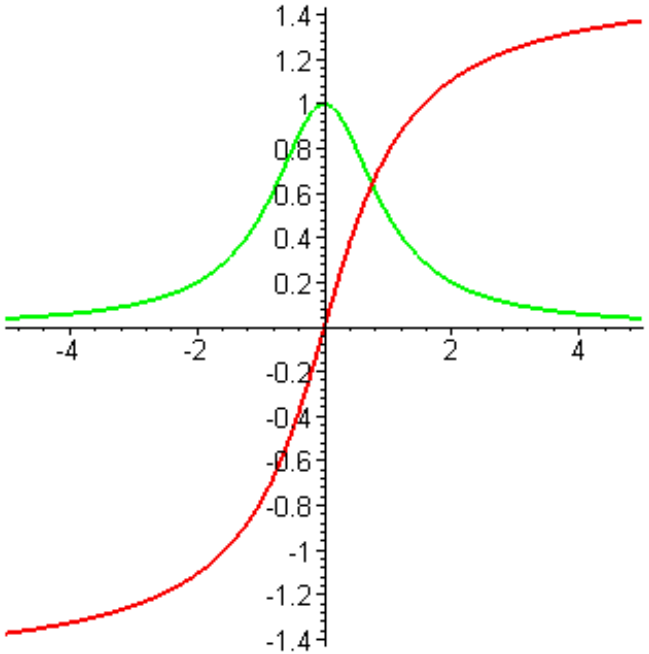
- takes functions as arguments or
- returns a function.

Remarks:

- This is hard to do in most imperative programming languages.
- This is a very powerful mechanism.
  - Abstraction, code re-use

An Example From Mathematics

The derivative in calculus maps (differentiable) functions to functions.



An Example From Last Lecture

```
fun time f =
  let
    val timer = Timer.startCPUTimer ()
    val _ = f () (* do the actual work *)
  in
    Timer.checkCPUTimes timer
  end
```

Here, f is a function (of type unit -> 'a). Thus, time is a higher-order function.

Another Example: Function Composition

Function composition is an operator (i.e., a function written in infix notation) that takes two functions and returns a function!

```
fun o (f,g) x = f (g x);
infix o;
```

What is the type of "o"?

Another Example: Function Composition

Function composition is an operator (i.e., a function written in infix notation) that takes two functions and returns a function!

```
fun o (f,g) x = f (g x);
infix o;
```

What is the type of "o"?

"o" is already predefined. You don't need to define it yourself.

```
> op o;
val it = fn: ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

Example:

```
> fun add1 x = x + 1;
> (add1 o add1) 42;
```

Another Example: Function Composition (cont.)

We can define a function that applies some function f twice:

Another Example: Function Composition (cont.)

We can define a function that applies some function f twice:

```
fun twice f = f o f

> twice add1 42;
> twice (twice add1) 42;
```

More generally, we can define a function that applies some function f n times:

Another Example: Function Composition (cont.)

We can define a function that applies some function *f* twice:

```
fun twice f = f o f

> twice add1 42;
> twice (twice add1) 42;
```

More generally, we can define a function that applies some function *f* *n* times:

```
fun ntimes 0 f x = x
  | ntimes n f x = ntimes (n-1) f (f x)

> ntimes 3 add1 42;
> ntimes 3 twice add1 42;
```

Insertion Sort on Integers

```
fun insert x [] = [x]
  | insert x (y :: ys) =
    if x < y then
      x :: y :: ys
    else
      y :: (insert x ys)

fun isort [] = []
  | isort (x :: xs) = insert x (isort xs)
```

Which part of the code is specific to integers?

Insertion Sort on Any Ordered Type

```
fun insert order x [] = [x]
  | insert order x (y::ys) =
    if order (x,y) then
      x :: y :: ys
    else
      y :: (insert order x ys)

fun isort order [] = []
  | isort order (x::xs) = insert order x (isort order xs)
```

What are the types of these functions?

Insertion Sort: Examples

```
> isort (op <) [6,3,0,1,7,8,5,9,2,4];
val it = [0,1,2,3,4,5,6,7,8,9]: int list
> isort (op >) [6,3,0,1,7,8,5,9,2,4];
val it = [9,8,7,6,5,4,3,2,1,0]: int list
> isort String.< ["one","two","three","four","five","six"];
val it = ["five","four","one","six","three","two"]: string list
> isort (op <);
val it = fn: int list -> int list
> isort String.< ;
val it = fn: string list -> string list
> isort (fn ((_,s1),(_,s2)) => String.< (s1,s2))
  [(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five"),(6,"six")];
val it = [(5,"five"),(4,"four"),(1,"one"),(6,"six"),(3,"three"),
   (2,"two")]: (int * string) list
```

Another Example: pair

```
fun pair x y f = f x y;

val pair = fn: 'a -> 'b -> ('a -> 'b -> 'c) -> 'c

> pair 45 34;
poly: : warning: The type of (it) contains a free type variable.
   Setting it to a unique monotype.
val it = fn: (int -> int -> _a) -> _a
>

fun frst p = p (fn x => fn y => x);

> frst (pair 45 54);
val it = 45: int
```

Another Example: pair (cont)

To avoid the warning:

```
> val pair' = (fn x => fn y => fn f => f x y)
            : 'a -> 'a -> ('a -> 'a -> 'a) -> 'a;

 val pair' = fn: 'a -> 'a -> ('a -> 'a -> 'a) -> 'a


> snd (pair' "foo" "bar");
val it = "bar": string
```

Table of Contents

Reflection on the Definition of sum

Remember:

```
fun sum [] = 0
  | sum (x::xs) = x + sum xs
```

There are only two expressions in the function definition that are specific to integers:

- "0", the result for the empty list
- "+", combining the current element with the recursive result

A Generalization

Let us define a function foldr that, when applied to **op** + and 0, is equal to sum:

```
fun foldr f b [] = b
  | foldr f b (x::xs) = f (x, foldr f b xs)
```

Note the similarity between foldr and sum.

A Generalization (cont.)

Now, sum can be defined as

```
fun sum xs = foldr (op +) 0 xs
```

or, equivalently, as

```
val sum = foldr (op +) 0
```

Other Uses of foldr

What will the following compute?

```
> foldr (op * ) 1   [1,2,3,4];

> foldr  Int.max 0   [1,2,3,44,5,6];

> foldr (fn (x,ys) => x::ys) []   [1,2,3,4];

> foldr (fn (x,ys) => x::ys) [5,6,7]   [1,2,3,4];

> foldr (fn (x,ys) => x::x::ys) []   [1,2,3,4];

> foldr (op @) []    [[1,2],[34],[5,6,7,89]];
```

Higher-Order Functions on Lists

Some useful higher-order functions on lists.

```
> map;
val it = fn: ('a -> 'b) -> 'a list -> 'b list

> foldl;
val it = fn: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

> List.filter;
val it = fn: ('a -> bool) -> 'a list -> 'a list
```

These functions are predefined in SML, but we will look at the details.

The map Function

Apply the same function to all elements of a list:

```
(* map f xs
   TYPE: ('a -> 'b) -> 'a list -> 'b list
   PRE: true
   POST: [f(a_1), f(a_2 ),..., f(a_n )], if xs = [a_1, a_2 ,..., a_n]
*)
```

The map Function

Apply the same function to all elements of a list:

```
(* map f xs
   TYPE: ('a -> 'b) -> 'a list -> 'b list
   PRE: true
   POST: [f(a_1), f(a_2 ), ..., f(a_n )], if xs = [a_1, a_2 , ..., a_n]
 *)
fun map f [] = []
  | map f (x :: xs) = f x :: map f xs
```

The map Function: Examples

```
> fun square x = x*x;
val square = fn: int -> int

> map square [1,2,3,4];
val it = [1,4,9,16]: int list

> map (fn x => if x < 3 then 0 else x) [1,2,3,4];
val it = [0,0,3,4]: int list
```

The map Function: Examples (cont.)

```
> map square;
val it = fn: int list -> int list

> map (map square);
val it = fn: int list list -> int list list

> map (map square) [[1,2,34],[5]];
val it = [[1,4,1156],[25]]: int list list

> map String.<;
val it = fn: (string * string) list -> bool list

> map String.< [("a","ab"), ("hello", "bye")];
val it = [true, false]: bool list
```

foldl and foldr

Recurse over a list:

```
fun foldl f b [] = b
  | foldl f b (x::xs) = foldl f (f (x,b)) xs

fun foldr f b [] = b
  | foldr f b (x::xs) = f (x, foldr f b xs)
```

Which one is tail-recursive, foldl or foldr?

foldl and foldr: Examples

Sum the elements of a list:

```
> foldl (op +) 0 [0,2,21,4,6];
```

```
> foldr (op +) 0 [0,2,21,4,6];
```

Are they equivalent?

Which one would you use? Why?

foldl and foldr: Examples (cont.)

Check that all elements of a list are even:

```
> foldl (fn (x,y) => y andalso x mod 2 = 0) true
    [0,2,21,4,6];

> foldr (fn (x,y) => y andalso x mod 2 = 0) true
    [0,2,21,4,6];
```

Are they equivalent?

Which one would you use? Why?

foldl and foldr: Examples (cont.)

Compare

```
> foldl (op ::) [];
```

```
> foldr (op ::) [];
```

Are they equivalent?

Which one would you use? Why?

foldl and foldr: Examples (cont.)

What does this function compute?

```
fun even_or_none (x, NONE) =
      if x mod 2 = 0 then SOME x else NONE
  | even_or_none (_, SOME x) =
    SOME x
```

Compare

```
> foldl even_or_none NONE;
> foldr even_or_none NONE;
```

Are they equivalent? Which one would you use for what?

foldl and foldr: More Examples

Try

```
> foldl (fn (x,n) => n+1) 0;
> foldr (fn (x,n) => n+1) 0;
> foldl (fn (x,n) => x) 0;
> foldr (fn (x,n) => x) 0;
> fun mystery f = foldr (fn (x,xs) => f x :: xs) [];
```

Filter

Obtain the elements in a list that satisfy a given property:

```
(*  filter  p xs
    TYPE: ('a -> bool) -> 'a list -> 'a list
    PRE: true
    POST: the list of elements in xs for which p is true
 *)
```

Filter

Obtain the elements in a list that satisfy a given property:

```
(*  filter  p xs
    TYPE: ('a -> bool) -> 'a list -> 'a list
    PRE: true
    POST: the list of elements in xs for which p is true
 *)
fun filter p [] = []
  | filter p (x::xs) =
      if p x then
          x :: filter p xs
      else
          filter p xs
```

Filter: Examples

```
> filter (fn x => x<6) [6,3,0,1,8,5,9,3];

> filter (fn x => x<6);
```

(The function `List.filter` is predefined in SML.)

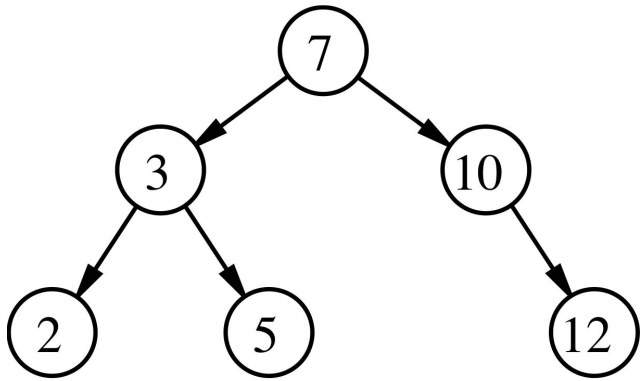
Table of Contents

Ordered Binary Trees

An **ordered binary tree** (or **binary search tree**) is a binary tree which is ordered, i.e.,

- all labels in the left subtree are *smaller* than the root label, and
- all labels in the right subtree are *larger* than the root label, and
- both subtrees are ordered.

Example:



Any ordered type may be used for the labels (e.g., `int`, `real`, `string`, ...).

Ordered Binary Trees in SML

In SML:

Ordered Binary Trees in SML

In SML:

datatype 'a obtree = Void | Node **of** 'a obtree * 'a * 'a obtree

Note:

- The order is not part of the datatype.
- It is (unfortunately) possible to build obtrees that are *not* ordered.

Searching a Label

```
(* search compare x t
   TYPE: ('a * 'a -> order) -> 'a -> 'a obtree -> bool
   PRE: t is ordered (wrt. compare)
   POST: true iff t contains x
*)
```

Searching a Label

```
(* search compare x t
   TYPE: ('a * 'a -> order) -> 'a -> 'a obtree -> bool
   PRE: t is ordered (wrt. compare)
   POST: true iff t contains x
 *)
fun search compare _ Void =
        false
  | search compare x (Node (l,v,r)) =
      case compare (x,v) of
        EQUAL => true
      | LESS => search compare x l
      | GREATER => search compare x r
```

Datatypes With Higher-Order Constructors

Consider again the previous definition of 'a obtree and its shortcomings:

- The functional argument compare must be passed to each function that operates on the tree (search, insertion, deletion, etc.).
- The compare order is not explicitly associated with the tree. Care must be taken to use the same order for each operation on a tree.

We introduce a new datatype for ordered binary trees that contains the order explicitly:

```
datatype 'a obtree_with_order =
    TreeWithOrder of ('a * 'a -> order) * 'a obtree
```

Searching a Label (Again)

```
(* search x t
   TYPE: 'a -> 'a obtree_with_order -> bool
   PRE: true
   POST: true iff t contains x
 *)
```

Searching a Label (Again)

```
(* search x t
   TYPE: 'a -> 'a obtree_with_order -> bool
   PRE: true
   POST: true iff t contains x
 *)
fun search x (TreeWithOrder (compare, obtree)) =
  let
    fun search' Void = false
      | search' (Node (l,v,r)) =
          case compare (x,v) of
            EQUAL => true
          | LESS => search' l
          | GREATER => search' r
  in
    search' obtree
  end
```

Table of Contents

Higher-Order Functions on Trees

Like for lists, it is possible to define generic (i.e., higher-order) functions on trees. For instance:

- We can define an analog of the `map` function, to apply some given function to all labels in a tree.
- Folding (i.e., generic recursion) over trees can be defined, similar to `foldr` for lists.

We use binary trees as an example, but the same might be done for other types of trees.

```
datatype 'a btree = Void | Node of 'a * 'a btree * 'a btree
```

Mapping Over a Binary Tree

A map function for binary trees:

```
fun map_btree f Void =
      Void
  | map_btree f (Node (x,l,r)) =
      Node (f x, map_btree f l, map_btree f r)
```

Examples:

```
> map_btree;
> map_btree Int.toString;
> map_btree Int.toString (Node (1, Void, Node (2, Void, Void)));
```

Folding Over a Binary Tree

A fold function can be defined for any inductive datatype. In general, this function takes one argument for each datatype constructor.

Thus, for binary trees,

```
fun fold_btree n v Void =
      v
  | fold_btree n v (Node (x, l, r)) =
      n (x, fold_btree n v l, fold_btree n v r)
```

What is the type of fold_btree ?

Folding Over a Binary Tree: Examples

```
> fold_btree;
val it = fn: ('a * 'b * 'b -> 'b) -> 'b -> 'a btree -> 'b
```

Tree height:

Folding Over a Binary Tree: Examples

```
> fold_btree;
val it = fn: ('a * 'b * 'b -> 'b) -> 'b -> 'a btree -> 'b
```

Tree height:

```
fold_btree (fn (x,l,r) => 1 + Int.max (l,r)) 0
```

Mirror image:

Folding Over a Binary Tree: Examples

```
> fold_btree;
val it = fn: ('a * 'b * 'b -> 'b) -> 'b -> 'a btree -> 'b
```

Tree height:

```
fold_btree (fn (x,l,r) => 1 + Int.max (l,r)) 0
```

Mirror image:

```
fold_btree (fn (x,l,r) => Node (x,r,l)) Void
```

In-order list of all labels:

Folding Over a Binary Tree: Examples

```
> fold_btree;
val it = fn: ('a * 'b * 'b -> 'b) -> 'b -> 'a btree -> 'b
```

Tree height:

```
fold_btree (fn (x,l,r) => 1 + Int.max (l,r)) 0
```

Mirror image:

```
fold_btree (fn (x,l,r) => Node (x,r,l)) Void
```

In-order list of all labels:

```
fold_btree (fn (x,l,r) => l @ [x] @ r) []
```