Foundation of Functional Languages: Higher Order Functional Programming

- Functional Languages: The main Features
- Syntax Essentials
- Programming Methodologies in Functional Languages
- Higher Order Programming, Iterative and Combinatory Programming
- Foundations: Term Reduction, Reduction Strategies, Combinators and Graph Reduction
Functional Languages: The Main Features

- Referential Transparency in Pure Functional (see Lecture 9-10)
- List Types and Operators (see Lecture 11)
- Structured Values are Fully Expressible Values (see Lecture 11)
- Garbage Collection for Heap re-allocation (see Lecture 18-19)
- First-Class Function Values and Higher Order Functions
- Extensive Polymorphism (see Lecture 11 and Lecture 18-19)
- Functions may return Structured Values
It is impressive how compact is the syntax of functional language (only Functions and Types definitions and Expressions), and How few are the mechanisms that are needed in functional programming, and The amount of different programming methodologies that are well supported by functional programming

All these facts are a trivial consequence of only one fact:

Functions are First-Class Values
Programming Methodologies in Functional Programming

- **Decomposition Based Programming** (see Lecture 15-17)

  **Example**
  
  Decomposition Programming is used in getting the program of the memoized factorial below. The solution consists of 4 components: A shared memory (hashTab), 2 distinct, autonomous, independent, functions (hash and set), and a code that combines all them and realizes the memoized factorial.

- **Inductive Programming** (see Lecture 15-17)
- **Tail Recursion Programming**
- **Memoization Based Programming**

  **Example**
  
  ```
  let rec fact = fun n → if (n=0) then 1 else n*(fact (n-1))
  -- Recursive definition of an inductive algorithm for fact
  
  let fact = fun n → let rec factT = fun n r → if (n=0) then r else (fact (n-1) (n*r)) in
  factT n 1
  -- Tail Recursive definition of inductive factorial
  
  let fact = let hashTab = ref [] in
  let hash = fun n → if (mem_assoc n (!hashTab)) then (true,(assoc n(!hashTab)))
  else (false,raise IntUndef) in
  let set = fun n v → hashTab:=(n,v)::(!hashTab) in
  -- Memoized Part, to be completed with the code of lecture 15-17, of inductive factorial
  
  Divide and Conquer (see Lecture 15-17)

  **Example**
  
  We apply the methodology to define QuickS on lists of a generic type.
Divide and Conquer (see Lecture15-17)

Example

We apply the methodology to define QuickS on lists of a generic type

let rec quickS =
    fun c o →
    if (length c) < 2 then c
    else let sel = hd in
    let u = (sel c) in
    let gt = filter (fun x → (o u x)) c in
    end use of Higher Order
    let lt = filter (fun x → (o x u)) c in
    end use of H.O.: Lambda Abstraction
    (quickS lt o)@((u::(quickS gt o));

– It uses the module "List.ml" but it is not enough to guarantee the full generalization of the algorithm. The module has only "hd" that behaves as a selection function
– Apply to the computation of: quickS [3;5;1;5;0;8;9] (>)

Polymorphism. quickS has Ocaml type:
'alpha list -> ('alpha -> 'alpha -> bool) -> 'alpha list

Example

– Apply quickS to sorting ["aba",78],"a",13),"ab",0)] according to two different pair orderings at your choice.

Higher Order Programming

Iterative and Combinators based Programming
Divide and Conquer (see Lecture15-17)

Example

We apply the methodology to define QuickS on lists of a generic type

let rec quickS =
  fun c o →
  if (length c) < 2 then c
  else let sel = hd in
    let u = (sel c) in
    let gt = filter (fun x → (o u x)) c in  -- use of Higher Order
    let lt = filter (fun x → (o x u)) c in  -- use of H.O.: Lambda Abstraction
    (quickS lt o)@(u::(quickS gt o));

– It uses the module "List.ml" but it is not enough to guarantee the full generalization of the algorithm. The module has only "hd" that behaves as a selection function
– Apply to the computation of: quickS [3;5;1;5;0;8;9] (>)

Higher Order Programming, HOP

– It is pervasive of Programming in Functional Languages
– Hence, it appear also, in combination with all other programming methodologies used in functional Programming
– For instance, in quickS, HOP furnishes the values to make quickS parametric w.r.t. the ordering relation.

Iterative and Combinators Programming
Methodologies: Higher Order Programming - Values

- Higher Order Programming, HOP
  - It is pervasive of Programming in Functional Languages
  - Hence, it appear also, in combination with all other programming methodologies used in functional Programming

- But what are the ingredients of the methodology HOP
  - **Data Extensions through Functional Abstractions**
    - New Domains of values are introduced through New Sets of functions
    - The new functions behave according to the way in which the new values have to be used in the program to be developed
    - Implementation details of the new values have not to be provided from programmer
    - This is much more than of abstract data types of programming languages, since ADT require the definition of an implementation module in order to be used in computation

**Example**

We apply HOP to the definition of ENV in Lecture 11

```ocaml
# let bindP = fun i d e -> fun j -> if (j=i) then d else e j;;
val bindP : 'a -> 'b -> ('a -> 'b) -> 'a -> 'b = <fun>
# let findP = fun i e -> e i and emptyP = fun i -> i;;
# let anEnv = bindP 3 5 emptyP;;
val anEnv : int -> int = <fun>
```

- **Control Extensions through Functional Abstractions**
But what are the ingredients of the methodology HOP

- **Data Extensions through Functional Abstractions**
  
  This is much more than abstract data types of programming languages, since ADT require the definition of an implementation module in order to be used in computation.

**Example**

We apply HOP to the definition of ENV in Lecture 11

```ocaml
# let bindP = fun i d e -> fun j -> if (j=i) then d else e j;;
val bindP : 'a -> 'b -> ('a -> 'b) -> 'a -> 'b = <fun>
# let findP = fun i e -> e i and emptyP = fun i -> i;;
# let anEnv = bindP 3 5 emptyP;;
val anEnv : int -> int = <fun>
# findP 3 anEnv;;
- : int = 5
```

Ocaml can define it better, by using types that highlight the different set of values and forbid illegal uses of the values.

**Example**

We apply HOP to the definition of ENV in Lecture 11

```ocaml
# type ide = I of string;;
# type den = L of int | C of int;;
# type env = ide -> den;;
# let bind = fun (i:ide) (d:den) (e:env) -> fun (j:ide) -> if (j=i) then d else e j;;
val bind : ide -> den -> env -> ide -> den = <fun>
... -- complete and run
```

- **Control Extensions through Functional Abstractions**
But what are the ingredients of the methodology HOP

- Data Extensions through Functional Abstractions
- **Control Extensions through Functional Abstractions**

- New Control Structures are introduced through New Sets of functions
- The new functions combines in new ways the data and the functions to be used in the program to be developed
- Implementation details of the new control structures (namely, the structure of the Activation Records, etc) have not to be provided from programmer
- This is the why, in giving denotational semantics, we can use a functional language as the defining metalanguage: All the control mechanisms of all languages are easily formalized without adding useless, implementation details

---

**Example**

We apply HOP to a combinatory, recursive, definition of factorial

```
let cITE = fun p f g -> fun n -> if (p n) then (f n) else (g n);;
let cmp = fun f g n -> g(f n) and cC = fun f g n -> g n (f n) and cK = fun f g -> f ;;
let im = (+)(-1) and ip = fun n m -> n*m;;
let rec fact = fun n -> (cITE ((=)0) (cK 1) (cC (cmp im fact) ip) n);;
```

Comment the use of the argument n, (apply denotational semantic) and run: `fact 3`

---

\[^a\] by \(\eta\)-conversion it is equivalent to combinatory `cITE ((=)0) (cK 1) (cC (cmp im fact) ip)`
But what are the ingredients of the methodology HOP

- Data Extensions through Functional Abstractions
- **Control Extensions through Functional Abstractions**
  - We revisit step by step, the program development

Example

1. We introduce "cITE" that applies to 3 functions and a value "n" and returns the application of "if-then-else" to the 3 expressions resulting from distributing "n" to each function:
   
   ```ocaml
   let cITE = fun p f g n -> if (p n) then (f n) else (g n);
   ```

Example

2. We introduce "cmp" that computes ordinary function composition:
3. cC that compute a different way of doing function composition
4. cK that ignores second argument and returns the first one

   ```ocaml
   let cmp = fun f g n -> g(f n) and cC = fun f g n -> g n (f n) and cK = fun f g -> f ;;
   ```

Example

5. We introduce im and ip to use subtraction and multiplication as values in Ocaml (that has various syntactic idiosyncracies)

   ```ocaml
   let im = (+)(-1) and ip = fun n m -> n*m;;
   ```

Example

```ocaml
let rec fact = fun n -> cITE ((=)0) (cK 1) (cC (cmp im fact) ip) n;;
```
But which are the ingredients of the methodology HOP

- Data Extensions through Functional Abstractions

- **Control Extensions through Functional Abstractions**
  - The previous slide showed the way to use HOP in Control Extensions
  - Programmer defines the HOP control functions of which the program needs:
    - In order to implement a specific algorithm or way of computing,
    - Or to satisfy any other requirement of the program development or of its use
  - However, 40 years of Functional Programming produced a great quantity of HOP control functions
  - Functional Languages are equipped with several Libraries of **Functionals** with this purpose
  - These Libraries of Functionals differ one another for the kind of applications in which such Functionals are of general use
But which are the ingredients of the methodology HOP

- Data Extensions through Functional Abstractions

- Control Extensions through Functional Abstractions
  - The previous slide showed the way to use HOP in Control Extensions
  - Programmer defines the HOP control functions ...
  - However, 40 years of FP produced a great quantity of HOP control ...
  - Functional map is one of them: It applies one function to each element of one list

Example

map behaves in this way: \( \text{map } g [e_1; \ldots; e_n] \) returns \( [(g \ e_1); \ldots; (g \ e_n)] \)

Some applications:
- \( \text{map } (\text{fun } x \rightarrow x+1); \) computes...
- \( \text{map } \text{fact}; \) computes...
- \( \text{map } (\text{fun } x \rightarrow x > 10); \) computes...

An Ocaml, sequential definition of Map follows (but in parallel computing it has an obvious, different definition):

```ocaml
let rec map = fun f l -> match l with
    | [] -> []
    | x::lR -> (f x)::(map f lR)
```

Methodologies: HOP - Iterative Programming

But which are the ingredients of the methodology HOP

- Data Extensions through Functional Abstractions
- **Control Extensions through Functional Abstractions**
  
  - However, 40 years of FP produced a great quantity of HOP control ...
  - Iteration in FP: Functionals may be used to iterate functions on index ranges that are collected into lists
  - Iteration in FP: Fold’s are collectively called the functionals having this use

**Example**

fold_left behaves in this way:  
fold_left g a [e₁;...;en] returns g(...(g (g a e₁) e₂)...enen

fold_right behaves in this way:  
fold_right g [e₁;...;en] b returns g e₁ (g e₂ (...(g en b)...))

Some applications:

- fold_left (−) 100;; computes...
- fold_right (+);; computes...

and again, by introducing intervals:

- let rec nTom = fun n m -> if n>m then [] else (if n=m then [m] else n::(nTom (n+1) m));;

We give an iterative, combinatory, factorial:

- let fact = cmp (ntom 1) (fold_left (ip) 1);;

where cmp and ip are the function composition and the integer product of the previous slides

Comment and run fact for computing the 3!

**Example**

Use iterative HOP in getting a program for defining the size of lists
Table 20 – Functional Languages: Syntax

<table>
<thead>
<tr>
<th>Program Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>D</strong> ::= <strong>F</strong></td>
</tr>
<tr>
<td><strong>F</strong> ::= <strong>I</strong>= <strong>A</strong></td>
</tr>
<tr>
<td><strong>A</strong> ::= <strong>fun I</strong> → <strong>E</strong></td>
</tr>
<tr>
<td><strong>E</strong> ::= <strong>A</strong></td>
</tr>
<tr>
<td><strong>T</strong> ::= ... (scalar, tuple, <strong>map</strong>, ...)</td>
</tr>
<tr>
<td>... (list : fund. meth.)</td>
</tr>
<tr>
<td>... (Concrete : fund. meth.)</td>
</tr>
<tr>
<td>... (Abstract : fund. meth.)</td>
</tr>
</tbody>
</table>

- **α - reduction**
  \[ \text{fun } I_1 \rightarrow E \rightarrow \rightarrow \text{fun } I_2 \rightarrow [I_2/I_1]E \quad (\text{when } I_2 \notin \text{Free}(E)) \]

- **β - reduction**
  \[ (\text{fun } I \rightarrow E)E_0 \rightarrow [E_0/I]E \quad (\text{when } \text{Free}(E_0) \cap \text{Bound}(E) = \{\}) \]

- The above rules are for function application.

- Additional reduction rules are needed for the constructs introducing special class of data and associated operations, and concrete and abstract data.
α - reduction

\[ \text{fun } I_1 \rightarrow E \implies \text{fun } I_2 \rightarrow [I_2/I_1]E \quad (\text{when } I_2 \not\in \text{Free}(E)) \]

β - reduction

\[ (\text{fun } I \rightarrow E)E_0 \implies [E_0/I]E \quad (\text{when } \text{Free}(E_0) \cap \text{Bound}(E) = \{\}) \]

Example

let f = fun n m -> if n=0 then 1 else m and g = fun n -> g(n+1);

\[ f \ 0 \ (g \ 3) \implies f \ 0 \ (g \ 4) \implies ... \]

\[ f \ 0 \ (g \ 3) \implies \text{if } [0/n, (g \ 3)/m]n=0 \text{ then } [0/n, (g \ 3)/m]1 \text{ else } [0/n, (g \ 3)/m]m \implies 1 \]

Different Reduction Strategies are used in Functional Languages, in determining the sub-term to be reduced:

- External or Normal evaluation results in defining Most Defined functions (Haskell, Miranda)
- Internal or Eager Evaluation results in defining Less Defined functions (ML, Ocaml)

Example

let add = fun x y -> x+y;;

\[ \text{add } y \implies [z/y](\text{fun } x \rightarrow x+y)y \implies ... \]

it requires that add be α-reduced before ...

Combinatory programs do not need α-reduction and limit β-reduction only to the replacement of function names with their definition

Example

let add = (+);; -- this is the combinatory definition of add

\[ \text{add } y \implies (+)y \]

\[ \text{add } y \ 3 \implies y+3 \]
The use of combinators avoid the use of bindings for parameters
Thus ruling out the need of $\alpha$-reduction and limiting the use of $\beta$-reduction
Thus simplifying the reduction semantics and allowing graph reduction instead of term reduction
We apply it to the iterative, combinatory, factorial:

\[
\text{fact} = \text{cmp} (\text{nTom} \ 1) (\text{fold} (\text{prod}) \ 1) \tag{1}
\]

in the computation of:

\[
\text{fact} \ 2
\]

The images in next slide show the reduction graph produced by the reduction of:

\[
\text{fold} (\text{prod}) \ 1 (\text{nTom} \ 1 \ 2)
\]

\[\text{fold stands for fold\_left and prod for integer product}\]
Reduction of the expression $\text{fold}(\text{prod})1(n\text{Tom} \ 1 \ 2)$ that results from the invocation $(\text{fact} \ 2)^2$

$^2$Symbol @ stands for the application symbol, i.e. $(E_1 \ E_2)$ is $(E_1 @ E_2)$ and is drawn as a tree rooted at @ and having $E_1$ and $E_2$ as left and right sub-graphs.
Exercise1.

a. Apply HOP methodology to the definition, in Ocaml, of the Homogeneous Heap in Lecture 3-4-5-6: In particular, define operation for allocation, deallocation, and for computing the number of blocks that are free. You cannot use the imperative features of Ocaml;

b. Show the behaviour of the new data by running the given definitions in the case of an heap of 4 blocks, all initially free. Then, run for the allocation of 4 blocks, followed from the deallocation of the first and then, of the third of them. Finally, run for asking the number of free blocks.

c. Comment the use of types to get the definitions of point (a). Rewrite the definitions of (a) by using types, if such definitions do not use types in a way to forbid illegal uses of data and of operations.

Exercise2.

a. Give, in Ocaml, a memoized definition of the binomial coefficient \( C_n^k \);

b. Discuss adequately, the choice of the hashTab;

c. Run it repeatedly for the computation of the coefficients of \((1 + x)^3\) and of \((1 + x)^2\) and discuss the execution statistics.

Exercise3.

a. Give a combinatory, iterative, definition of map in Ocaml;

b. Discuss adequately, the choice of the combinators that you have used in the development;

c. Run map \((\text{prod } 5) \ [2;3]\) in an interactive session of Ocaml and check execution for the expected answer;

d. Show the computation of map \((\text{prod } 5) \ [2;3]\) when map is the combinatory definition, given in (a);

Exercise4.

Give points (a,b,c,d) of exercise3, in the case of a combinatory, but inductive, definition of fold_left: Use fold_left \((\text{prod}) \ 1 \ [2;3]\) for the expression to be run in (c) and (d).

---

3. It is the coefficient of the term \(x^k\) in the polynomial development of \((1 + x)^n\). Hence, it is such that: \( C_0^n = 1 \) (for \(n \geq 0\)), \( C_k^0 = 0 \) (for \(k > 0\)), \( C_k^n = C_k^{n-1} + C_{k-1}^{n-1} \) (for \(n,k > 0\))