Lecture 15-17
Advances in Control and Functional Abstractions

prof. Marco Bellia, Dip. Informatica, Università di Pisa

Apr. 15, 2014

Control and Functional Abstractions: Advanced Features

- Decomposition based Programming and Fully Abstract Abstractions
- Problems in Binding and in Scope of Identifiers: An example
- Deep and Shallow bindings in Functions Passed as Parameters
- Lambda Abstractions
- Shallow binding: FUI
- Inductive Programming and Recursively Defined Abstractions
- Memoization e Tail Recursion: Two Programming Techniques
- Divide and Conquer Programming Methodology
- Functions as First Class Values: Higher Order Programming Methodology

Decomposition based Programming and Fully Abstract Abstractions

- Decomposition based Programming consists in:
  - Breaking a problem in distinct, autonomous, independent, subproblems.
  - The solution of the problem is then, obtained from a suitable composition of the solutions of the sub-problems
  - Each subproblem may in turn, to be solved by using the same methodology.
- Hence, in order to support the latter point, a Procedural Language must be equipped with Fully Abstract abstractions.
- Fully Abstract abstraction requires that Procedures/Functions are Programming Units of the language, i.e. they have the same structure of the program:
  - Local entity identifiers;
  - Such entities must Include Nested Abstractions;
  - Hence, Non-local entity identifiers;
  - Control and Data structures as in the main program structure;
  - In addition: Parameter Passing for the caller/callee connection
  - In addition: Return/Exit for giving back the control

Problems in the binding and in the Scope of Identifiers: An Example

- By using Decomposition Based Programming, we can produce the program structure below:

Example

```
{...
  type T =int × int → bool;
  int u...
  function bool F1(int x, int y){...u...};
  procedure P(T g){int u....}
  function T Q(){
    int u ...
    function bool F2(int x, int y){...u...};
    ...
    P(F1); ...P(F2); .... return F2
    }  ...
  F1(...); ... P(Q()); ...}
```

- The function that is bound to identifier $\mathtt{F}_2$ survives to the binding of $\mathtt{F}_2$: What are its nonlocal bindings?
- What is the binding of u? In the invocations, within Q, it could be always, the binding of Q (shallow binding) or otherwise, the binding of the main block, in $\mathtt{P}(\mathtt{F}_1)$, and the one of Q, in $\mathtt{P}(\mathtt{F}_2)$ (deep binding).

Nonlocal Bindings in Functions Passed as Parameters/1

- In principle, four different choices:
  1) Deep Binding in Static Scope: Nonlocal Bindings are those active when Function has been defined
  2) Shallow Binding in Static Scope: Nonlocal Bindings are those active when Function applies
  3) Deep Binding in Dynamic Scope: Nonlocal Bindings are those active when Function has been defined
  4) Shallow Binding in Dynamic Scope: Nonlocal Bindings are those active when Function applies
- Each choice leads to a different way of programming with functions and a different way of using Decomposition Based Programming
- However, solution 2 is unused, in practice, whilst solution 3 is known as *Funarg in Lisp*
- We will consider 1 in detail.

Lambda Abstractions: Nameless Functions

- The choice on how select the nonlocal bindings, is also in parameter passing by procedure
- But first, we consider a new construct for defining functions in a program
- This construct is named Lambda Abstraction: For instance, `fun x→ x = 5` is a lambda abstraction in Ocaml, and similarly, `(x)→ x == 5` will be in Java 8.

Table12bis – Passing Functions as Values
Syntactic Domains
$\mathtt{D} ::= \ldots \mid \mathtt{Function}\ \mathtt{I}(\mathtt{P}_1\ \mathtt{I}_1 \ldots \mathtt{P}_n\ \mathtt{I}_n)\ \mathtt{E} \mid \ldots$
$\mathtt{E} ::= \ldots \mid \mathtt{A} \mid \mathtt{Lambda}(\mathtt{P}_1\ \mathtt{I}_1 \ldots \mathtt{P}_n\ \mathtt{I}_n)\ \mathtt{E} \mid \ldots$

- Lambda Abstraction is an expression;
- Lambda Abstraction, when evaluated, results a nameless function
- It is used as a denotable value, in parameter passing (Functional Languages)
- It is used as a storable value in some today languages (JavaScript, C#,...)
- The mechanism could be enriched in order to express recursively defined, nameless functions

Lambda Abstraction: The Computed Function

- The example below, involves three distinct lambda abstractions, all used in parameter passing

Table12bis – Passing Functions as Values
Domini Sintattici
$\mathtt{D} ::= \ldots \mid \mathtt{Function\ I(P_1\ I_1\ \ldots\ P_n\ I_n)\ E} \mid \ldots$
$\mathtt{E} ::= \ldots \mid \mathtt{A} \mid \mathtt{Lambda(P_1\ I_1\ \ldots\ P_n\ I_n)\ E} \mid \ldots$

- When nonlocals occur, the function defined by the lambda abstraction, depends from the binding mechanism, used in the language, for function passing.

Example

$\{\ldots\mathtt{type\ C(t)} = \ldots //\textit{a collection};\ \mathtt{T_f(t) = t} \rightarrow \mathtt{bool};\ \mathtt{T_o(t) = t \times t} \rightarrow \mathtt{t};\ldots$

$\quad\mathtt{function\ C(t)\ Filter(C(t)\ c, T_f(t)\ r)}\{\ldots.\};$

$\quad\ldots\{\ldots\mathtt{C(int)v} = \ldots;$

$\qquad\ldots.\mathtt{Filter}(\mathtt{v}, \mathbf{fun\ x} \rightarrow \mathbf{x > 5})\ldots //\textit{greater than } 5$

$\qquad\ldots.\mathtt{Filter}(\mathtt{v}, \mathbf{fun\ x} \rightarrow \mathbf{x <= 5})\ldots //\textit{lesser or equal to } 5$

$\qquad\ldots\{\ldots\mathtt{type\ T(t) = C(t) \times T_f(t)} \rightarrow \mathtt{C(t)};$

$\qquad\quad\mathtt{function\ C(t)\ QuickSort(C(t)\ c, T(t)\ f, T_o(t)o)}\{\ldots.\};$

$\qquad\quad\ldots.\mathtt{QuickSort}(\mathtt{v}, \mathtt{Filter}, \mathbf{fun(x, y)} \rightarrow \mathbf{(x > y)?y : x})$

Table12.1 – Deep Binding in Static Scope
Semantic Functions
$\mathcal{D}_E[\![\mathtt{D}]\!] : \mathtt{Env} \rightarrow \mathtt{Env}_\perp$
$\mathcal{D}_E[\![\mathtt{Function\ I(P_1 \ldots P_n)E}]\!]_\rho = \mathtt{bind}(\mathtt{I}, \mathtt{F}(\mathtt{g}), \rho)$
$\mathtt{where}\{\mathtt{g} = \lambda(\mathtt{v_1 \ldots v_n}).\lambda \mathtt{s}.(\mathtt{v_r}, \mathtt{s_r})$
$\mathtt{where}\{(\rho_\mathtt{n}, \_, \mathtt{s_n}) = \mathcal{B}[\![(\mathtt{P_1 \ldots P_n})]\!](\rho, (\mathtt{v_1 \ldots v_n}), \mathtt{s})\}$
$\{(\mathtt{v_r}, \mathtt{s_c}) = [\![\mathtt{E}]\!]_{\rho_\mathtt{n}}(\mathtt{s_n})\}$
$\{\mathtt{s_r} = \mathcal{R}[\![(\mathtt{P_1 \ldots P_n})]\!]\rho_\mathtt{n}((\mathtt{v_1 \ldots v_n}), \mathtt{s_c})\}$
Auxiliary Functions
$\mathtt{F} : \mathtt{VFun} \rightarrow \mathtt{Den}$
$\in\mathtt{VFun} : \mathtt{Val} \rightarrow \mathtt{TruthV}$
$\mathtt{VFun} ::= (\mathtt{Val} \cup \mathtt{Den})^\mathtt{n} \rightarrow \mathtt{State} \rightarrow (\mathtt{Val} \times \mathtt{State})_\perp$
$\mathtt{Den} ::= \mathtt{Loc} + \mathtt{ProcFun} + \mathtt{VL} + \mathtt{Code} + \mathtt{VFun}$

Deep Binding in Static Scope/2

Table12.1 – Deep Binding in Static Scope
Semantic Functions
$\mathcal{E}[\![\mathtt{E}]\!]_\rho : \mathtt{State} \rightarrow (\mathtt{Val} \times \mathtt{State})_\perp$
Function Invocation
$\mathcal{E}[\![\mathtt{Call\ I}(\mathtt{A}_1 \ldots \mathtt{A}_n)]\!]_\rho(\mathtt{s}) = \mathtt{f}(v_1 \ldots v_n)(\mathtt{s}_n)$
$\mathtt{where}\{((v_1 \ldots v_n), \mathtt{s}_n) = \mathcal{T}[\![(\mathtt{A}_1 \ldots \mathtt{A}_n)]\!]_\rho(\mathtt{s}),\ \mathtt{F}(\mathtt{f}) = \rho(\mathtt{I})\}$
Lambda Abstraction introduction
$\mathcal{E}[\![\mathtt{Lambda}(\mathtt{P}_1 \ldots \mathtt{P}_n)\mathtt{E}]\!]_\rho(\mathtt{s}_d) =$
$\lambda(\mathtt{v}_1 \ldots \mathtt{v}_n).\lambda\mathtt{s}.(\mathtt{v}_r, \mathtt{s}_r)$
$\mathtt{where}\{(\rho_n, \_, \mathtt{s}_n) = \mathcal{B}[\![(\mathtt{P}_1 \ldots \mathtt{P}_n)]\!](\rho, (\mathtt{v}_1 \ldots \mathtt{v}_n), \mathtt{s})\}$
$\{(\mathtt{v}_r, \mathtt{s}_c) = [\![\mathtt{E}]\!]_{\rho_n}(\mathtt{s}_n)\}$
$\{\mathtt{s}_r = \mathcal{R}[\![(\mathtt{P}_1 \ldots \mathtt{P}_n)]\!]\rho_n((\mathtt{v}_1 \ldots \mathtt{v}_n), \mathtt{s}_c)\}$
Parameter Passing : By Function
$\mathcal{T}_1[\![(\mathtt{Fun}(\mathtt{E}))]\!]_\rho((v_1 \ldots v_m), \mathtt{s}_m) =$
$\mathtt{Let}\{(v_{m+1}, \mathtt{s}_{m+1}) =_{\perp_S} \mathcal{E}[\![\mathtt{E}]\!]_\rho(\mathtt{s}_m)\}((v_1 \ldots v_m v_{m+1}), \mathtt{s}_{m+1})$
$\mathcal{B}_1[\![\mathtt{Fun\ I}_k]\!](\rho_{k-1}, (v_k \ldots v_n), \mathtt{s}_{k-1}) =$
$\mathtt{Let}\{\rho_k = \mathtt{bind}(\mathtt{I}_k, \mathtt{v}_k, \rho_{k-1})\}\ (\rho_k, (v_{k+1} \ldots v_n), \mathtt{s}_k)$
$\mathcal{R}_1[\![\mathtt{Fun\ I}_k]\!]_\rho((v_k \ldots v_n), \mathtt{s}_{k-1}) =$
$\mathtt{Let}\{\mathtt{s}_k = \mathtt{s}_{k-1}\}((v_{k+1} \ldots v_n), \mathtt{s}_k)$

Other Uses of Function/Procedure in Parameter Passing

Example

How the definitions in the two previous slides, have to be modified in order to deal with:

- Shallow Binding in Static Scope?
- Deep Binding in Dynamic Scope?

Recursively Defined Abstractions: FUI

- Control and Functional Abstractions are needed for Inductive Programming Methodology (IPM)
- IPM requires Recursive Functions and Procedures
- Recursive Functions and Procedures are usually introduced by combining: Naming (of the function) + Scope (including the function definition)

Table12.ter – Recursive Functions
Semantic Functions
$\mathcal{D}_E[\![\mathtt{D}]\!] : \mathtt{Env} \to \mathtt{Env}_\perp$
$\mathcal{D}_E[\![\mathtt{Function\ I(P_1 \ldots P_n)E}]\!]_\rho = \mathtt{Y}\delta.\mathtt{bind}(\mathtt{I}, \mathtt{F}(\mathtt{g}), \rho)$
$\mathtt{where}\{\mathtt{g} = \lambda(\mathtt{v_1 \ldots v_n}).\lambda\mathtt{s}.(\mathtt{v_r}, \mathtt{s_r})$
$\mathtt{where}\{(\rho_n, \_, \mathtt{s_n}) = \mathcal{B}[\![(\mathtt{P_1 \ldots P_n})]\!](\delta, (\mathtt{v_1 \ldots v_n}), \mathtt{s})\}$
$\{(\mathtt{v_r}, \mathtt{s_c}) = [\![\mathtt{E}]\!]_{\rho_n}(\mathtt{s_n})\}$
$\{\mathtt{s_r} = \mathcal{R}[\![(\mathtt{P_1 \ldots P_n})]\!]\rho_n((\mathtt{v_1 \ldots v_n}), \mathtt{s_c})\}$

- It can be easily reformulated for functions with Shallow Binding.

Inductive Programming Methodology: Use and Implementation

- Use: Inductive Programming Methodology, IPM
- IPM: The problem solution uses an inductive algorithm
  - It requires a well founded ordering, "<", on the data of the definition domain: Each value must have, only finite, descending, chains of predecessors w.r.t "<"
  - The calculated value at each point of the definition domain, depends on the calculated values on the (finitely many) predecessors of the point.
  - Recursively Defined Abstractions behave perfectly, in rephrasing this kind of programming
- Implementation of Recursively Defined Abstractions
  - Stack of AR's: The active AR's are as many as the predecessors on which the computation runs
  - *Memoization* reduces the number of the required AR's
  - *Tail Recursion* removes the need for all them but one

Memoization

- **Memoization** reduces the number of the required AR's in recursive functions
- A memoized function remembers all the values that it has computed in the previous invocations: Then it does not re-computes them.
  - Efficiency
  - Complexity: Memoized factorial may run with constant complexity whilst memoized fibonacci with linear complexity.
- Haskell has a mechanism for declaring a function to be built memoized
- All such mechanisms are based on a (local to function or global) hash table
- The table stores all the computed invocations, from the function, in a way similar to the example, below

Example

```
let rec fact = fun n → match (hash fact n) with
    (true, u) → u
    | otherwise → match (hash fact (n − 1)) with
            (true, u) → let ret = n * u in ((set fact n ret); ret)
            | otherwise → let ret = n * (fact(n − 1))
                                in ((set fact n ret); ret)
```

Tail Recursion

- **Tail Recursion** removes the need for the use of a chain of AR's of size equals to the number of predecessors on which function has to be invoked
- A function g is said to be tail recursive **iff** the value that it computes, at each invocation of g, in the function body, is the value that the function returns;
- A very few of the inductive algorithms are phrased using tail recursive function definitions

Example

```
function int fact(int n){
        (n = 0) ? 1 : n * fact(n - 1); }
```

- But many inductive algorithms are trivially rephrased in that way

Tail Recursion: Rephrasing of Inductive Definitions and Implementation

- A very few of the inductive algorithms are phrased using tail recursive function definitions

Example

```
function int fact(int n){
        (n = 0) ? 1 : n * fact(n − 1); }
```

- But many inductive algorithms are trivially rephrased in that way

Example

```
function int factT(int n, int r){
        (n = 0) ? r : factT(n − 1, n * r); }
```

- Implementation:
  - Each inner invocation, of a tail recursive function, uses the same AR of the first invocation;
  - By copying in it, the new values of the parameters
  - Any other component of AR stays unchanged (cd, cs, ret, val) or is reset (pc, ri)

Divide and Conquer Programming Methodology/1

- It extends Inductive Programming and Decomposition based Programming
  1. The problem is broken into sub-problems, but, of correlated kind
     - correlation is due to the use, in the sub-problems, of the same functionalities that we are inductively defining and using in giving the solution of the initial problem
  2. In considering, each sub-problem, we distinguish two cases:
     - 2.a The problem has immediate solution: Then problem stops by giving the solution
     - 2.b Otherwise: Otherwise: step (1) is iterated on the sub-problem

Example

```
{.... type T_oint = int × int → bool;
...{...C_int v = ...;
    {...
     function C_int QuickS(C_int c, T_oint o){
        if (Size(c)<2) return c;
        {int u = Sel(c); C_int gt = Filter(c, fun x → o(u, x));
         C_int lt = Filter(c, fun x → o(x, u));
         return Append(QuickS(lt, o), AddE(c, QuickS(gt, o)));
         };
     ....QuickS(v, fun(x,y)→(x>y)?y:x)
```

Divide and Conquer Programming Methodology/2

- A Very Powerful Definition of the Quicksort Algorithm that applies to whatever data structure and, to whatever ordering relation for such data. It uses:
  polymorphisms and
  by function passing parameters.

Example

```
{... type C(t) = ...; T_f(t) = t → bool; T_o(t) = t × t → bool; ...
 function int Size(C(t) c){....}; function t Sel(C(t) c){....};
 function C(t) AddE(t u, C(t) c){....};
 function C(t) Append(C(t) c_1, C(t) c_2){....};
 function C(t) Filter(C(t) c, T_f(t) r){....};
 ...
 {...C(int)v = ...; .....
  {...type T_z(t) = C(t) → int; T_a(t) = C(t) × C(t) → C(t);
          T_e(t) = t × C(t) → C(t); T_s(t) = C(t) → t;
   function C(t)QuickS(C(t)c,T_s(t)s,T_z(t)z,T_a(t)a,T_e(t)e,T_f(t)f,T_o(t)o){
     if(z(c)<2) return c;
     {t u = s(c); C(t) gt = f(c, fun(x) → o(u, x));
      C(t) lt = f(c, fun(x) → o(x, u));
      return a(QuickS(lt, s, z, a, e, f, o), e(u, QuickS(gt, s, z, a, e, f, o)));
      };
   ....QuickS(v, Sel, Size, Append, AddE, Filter, fun(x,y)→(x>y)?y:x)
```

- But there is a bit of confusion in the use of the data to be manipulated, and of the operations to be used for that goal: Its Reading is difficult

Divide and Conquer Programming Methodology: Use of Interfaces/3

- But there is a bit of confusion in the use of the data to be manipulated, and of the operations to be used for that goal: Its Reading is difficult
- should always be passed the operations of the data `C(t)`
- not at all, if the language has Abstract Data Type, or otherwise, Objects, Classes containing the local definitions for: Sel, Size, Append, AddE and Filter.
- Even better, (Haskell, Java) Interfaces that constrain data to a set of classes that satisfy some requirements:

```
  Class Ord(t) => C(t) where //Definitions of the Class operations
               Sel...; Size...; Append...; AddE...; Filter....
```

out of the class, operations are referred by prefixing the name with the Class name and a dot (similarly to the field selector of the record)

Example

```
{.... type To(t) = t × t → bool;
...{...C(int)v = ...;
    {...
     function C(t) QuickS(C(t)c, To(t)o){
       if (C(t).Size(c)<2) return c;
       {t u = C(t).Sel(c); C(t) gt = C(t).Filter(c, fun x → o(u, x));
        C(t) lt = C(t).Filter(c, fun x → o(x, u));
        return C(t).Append(QuickS(lt, o), C(t).AddE(c, QuickS(gt, o)));
        };
     ....QuickS(v, fun(x,y)→(x>y)?y:x)
```

Functions as First Class Values: HOP

Full Higher Order Programming Language

- Functions are First Class Values: It means that functions are the basic data of the programming language;
- Functions may be used as arguments in the invocation of functions;
- Functions may be used as the computed (returned) value of function invocations;
- Advantages: We are at the top of the Procedural Expressivity of a Language:
  - The program **computes** (during its execution) the functions with which to continue the computation;
  - Moreover, the behavior of the constructs of a language, may be completely, defined by semantic functions: These functions are computable functions, of course;
  - Hence, Programs may, if needed, introduce some of such functions:
    - as new data types
    - as new kinds of constructs for the control of new forms of function composition.

Functions as First Class Values: HOP/2

Including functions in the domain of the computable values of the language, is not difficult at any level (semantic, methodological)

- We had already extended the domain of the the computable values by including functions:
  - when functions are used as arguments of function (or procedure) invocations;
  - when functions are introduced by Lambda Abstractions
- Now we extend the domain of storable values to include functions that can be assigned to variables, once returned from an invocation (as in the example in the next slide)

Table12.quater – Functions as Storable Values

Semantic Functions

$$
\begin{aligned}
&\mathcal{E}[\![\mathtt{Call\ I}(\mathtt{A}_1\ldots\mathtt{A}_n)]\!]_\rho(\mathtt{s}) = \mathtt{g}(v_1\ldots v_n)(\mathtt{s}_n)\\
&\qquad \mathtt{where}\{((v_1\ldots v_n), \mathtt{s}_n) = \mathcal{T}[\![(\mathtt{A}_1\ldots\mathtt{A}_n)]\!]_\rho(\mathtt{s})\\
&\qquad\qquad \mathtt{g} = \mathcal{Q}(\mathtt{I})(\rho)(\mathtt{s}),\\
&\qquad\qquad\quad \mathtt{where}\{\mathcal{Q}(\mathtt{I})(\rho)(\mathtt{s}) =\\
&\qquad\qquad\qquad \mathtt{match}\ \rho(\mathtt{i})\ \mathtt{with}\ (\mathtt{F}(\mathtt{h}) \rightarrow \mathtt{h})\\
&\qquad\qquad\qquad\qquad (\mathtt{F}_\mathtt{M}(\mathtt{l}) \rightarrow \mathtt{look}(\mathtt{l}, \mathtt{s}))\}\}
\end{aligned}
$$

Auxiliary Functions

$\mathtt{F}_\mathtt{M} : \mathtt{VFun} \longrightarrow \mathtt{Mem}$

Auxiliary Domains

$\mathtt{Mem} ::= \mathtt{VL} + \mathtt{VFun}$

Higher Order Programming: Implementation

Including functions in the domain of the computable values of the language, is not difficult at any level (semantic, methodological)

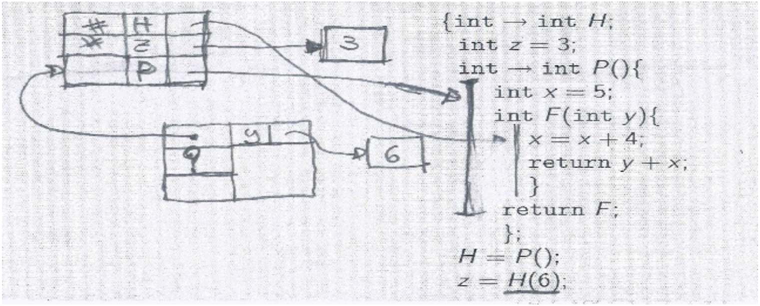
- At a first look, nothing to add to the "AR's" based implementation
- But when invocation H(6) runs, in the last line:

Example

```
{ int → int H;
  int z = 3;
  int → int P(){
     int x = 5;
     int F(int y){
        x = x + 4;
        return y + x;
        }
     return F;
     };
  H = P();
  z = H(6);
```

Higher Order Programming: Implementation/2

Where can H, i.e. F, find its nonlocal binding when invocation `H(6)` runs?:



Two solutions:

- Currying + Lambda Lifting (for static scope)
- Stack with AR Retention

Excercises

Exercise1.

Complete in Ocaml, the definition of the memoized factorial, discussed in the slides on the memoization. The definition must use a local hash table. The hash table can be reduced to a simple list of pairs or to a suitable function. Then

- a Discuss the structure of the defined function, in particular the language constructs that have been used in the implementation of the memoization part;
- b Apply it to the computation of `5!` and comment the resulting performance compared with that of the non-memoized version;

Exercise2.

Give, in Ocaml, a tail recursive definition of a function that computes the n-th of the Fibonacci series

Exercise3.

Write, in Ocaml, the QuickSort definition, given in the previous slides. Then:

- a List the functions involved in the definition and say what are operations of `C(t)` and what are of `t`. Then, say the number of the different data types that are involved in QuickSort;
- b Apply the definition to the case of a list of strings that must be ordered by size and must remove strings that do not contain the character "a";