CSci 2041: Advanced Programming Principles
Functions as First-Class Objects

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Lectures in Spring 2015

The Topic of Study

We want to understand the treatment of functions as first-class objects

The particular questions to be addressed in this context

- What exactly does it mean to be a “first-class object?”
- In what sense does OCaml give functions this kind of treatment?
- What is the use for this kind of treatment in programming?

We will discuss the first two aspects quickly and then look at different programming examples to understand the last

The Notion of a First-Class Object

A good way to understand what this means is to consider all the things we can do with simple data like integers

- We can construct expressions that denote integers

  E.g., we can construct expressions such as `3 + 2`
- We can embed integer expressions in other data

  E.g., we can construct lists and tuples of integers
- Integer expressions can be passed as parameters

  E.g., we can invoke functions like `gcd` and `fact` over such expressions
- We can return integers as the results of function calls
- We can compare two integer expressions

If we can do most of these things with some other type, then we would be giving a first-class treatment to objects of that type

Constructing Expressions of Function Type

We have already seen that this is possible in OCaml

The general form for such an expression

```
fun <name> -> <exp>
```

A concrete example of this kind

```
# fun x -> x + x;;
- : int -> int = <fun>
#
```

We have also seen that we can *name* such expressions

In fact, this is exactly what our function definitions are doing

Embedding Functions in Complex Data

We have not seen concrete examples of kind yet, but it should be clear we can do it

In fact, we already know we can construct types such as

```
(int -> int) list      (int -> int) * int
```

Here are concrete examples of objects of these types

```
# let double x = 2 * x;;
val double : int -> int = <fun>
# let treble x = 3 * x;;
val treble : int -> int = <fun>
# let flist = [double; treble];;
val flist : (int -> int) list = [<fun>; <fun>]
# let fip = (double,5);;
val fip : (int -> int) * int = (<fun>, 5)
#
```

Passing Functions as Parameters

Again, it should be clear we can do this, although we have not seen concrete examples yet

That we can do this, should be clear from the fact that we can construct types such as

```
(int -> int) -> int      (int -> int) * int -> int
```

An actual example to bring this possibility out

```
# let apply (f,i) = f i;;
val apply : ('a -> 'b) * 'a -> 'b = <fun>
# apply (double,5);;
- : int = 10
#
```

Returning Functions as Results

We have seen this possibility already in *curried* functions

```
# let plus x y = x + y;;
val plus : int -> int -> int = <fun>
# let plus5 = plus 5;;
val plus5 : int -> int = <fun>
#
```

In fact, we can *explicitly* return functions as results

```
# let addto x =
    let plusX y = x + y in plusX;;
val addto : int -> int -> int = <fun>
# let plus5 = addto 5;;
val plus5 : int -> int = <fun>
# plus5 7;;
- : int = 12
#
```

Functions as Closures

The last example shows that, to realize static scoping, we should think of functions as a *combination* of two items

- a function valued expression possibly with free identifiers
- an *environment* with bindings for the free identifiers

More specifically, after the following

```
# let addto x =
    let plusX y = x + y in plusX;;
val addto : int -> int -> int = <fun>
# let plus5 = addto 5;;
```

`plus5` should conceptually have a structure like this:
`(fun y -> x + y,[(x,5)])`

This kind of representation of functions is called a *closure*

Functions in OCaml versus Function Pointers in C

Question: Languages like C already allow us to pass pointers to functions in C, so is there anything special about OCaml?

The functions you can treat as objects in C is *statically determined*

In particular, they are limited to the functions that the user has *explicitly defined*

In OCaml, function objects can be *dynamically generated* and there can be an *unlimited* number of them

For example consider the following

```
let addto x = fun y -> x + y
```

Each application of `addto` to a number generates a *new* function

Comparing Function Objects

Question: When should we say two functions are equal?

- We could use equality based on input-output behaviour

  But then we would have to say the following are equal

  ```
  let twice x = 2 * x    let twice' x = x + x
  ```

  However, this kind of equality is often not computable

- We could instead compare using the "form" of the function

  But such comparisons may also require computations

  For example, if `multby` is defined as follows

  ```
  let multby x = let prod y = x + y in prod
  ```

  is `twice` equal to `(multby 2)`?

OCaml and other functional languages disallow comparisons of functions for these reasons

Comparing Function Objects

Concretely, here is what happens when we try to compare function objects in OCaml

```
# let twice x = 2 * x ;;
val twice : int -> int = <fun>
# let twice' x = x + x;;
val twice' : int -> int = <fun>
# let multby x =
  let prod y = x + y in prod;;
val multby : int -> int -> int = <fun>
# twice = twice';;
Exception: Invalid_argument
             "equal: functional value".
# twice = (multby 2);;
Exception: Invalid_argument
             "equal: functional value".
#
```

Usefulness of Functions as First-Class Objects

We will now consider how we can benefit from such treatments of functions in programming

Some of the applications we will consider are the following

- Building “higher-order” methods, i.e. methods for combining other methods
- Realizing modularity and truly parametric polymorphism over data structures
- Realizing control constructs in languages like C
- Making control in functional programming languages explicit

Composing Two Functions

There are many situations in which you may want to apply two computations one after another

E.g, you may want to compile a program and then link the result

We can define a *compose* function as follows

```
# let compose f g = fun x -> f (g x);;
val compose :
  ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
#
```

Note that `compose` allows us to

- combine any two arbitrary functions
- but only if their types are compatible

This is like piping the output of one process into another in Unix, except for the type checking

Lifting Functions over Data Structures

The General Idea

We want to modify every item in a data structure using a function we have at hand

For example, given a set A we may want to describe another set as follows

$$B = \{ f(a) \mid a \in A \}$$

This is referred to as *set comprehension*

When we treat sets as lists, this gives rise to the idea of *mapping* a function over a list

In particular, we want to generate a new list by applying a function to *every* element in a given list

Mapping a Function over a List

We can capture the general idea in a function called `map`

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

Note that this is a function that can map *any* function over a list

This capability is obtained by taking a function as argument, something that makes `map` a *higher-order function*

As an example application, we can define `transpose` very succinctly using `map`

Note also that `map` is available through a `List` library in OCaml and can be used as `List.map`

Evaluation and Implicit Parallelism

In the purest form of functional programming

- we can choose where in an expression to evaluate next
- we can even evaluate in many places simultaneously

This principle gets compromised when we begin relying on a strategy, but it still works in many cases

This is especially true for mapping a function `f` over a finite list `[e1; ...; en]`

The computation reduces to calculating `(f e1)`, ..., `(f en)` and each can be done *in parallel*

In exploiting this principle in practice, a major issue becoming how to *schedule* the parallelism

Aggregating Over a Data Structure

Sometimes we want to calculate a cumulative property over *all* items in a data structure

For example, we may want to do the following kinds of things over a list of numbers

- the sum of all the numbers in the list
- the largest number in the list
- the length of the list
- find the proportion of even numbers versus odd numbers in the list

These kinds of operations involve a traversal (or recursion) over a data structure and the application of a function

Once again, we can parameterize the process by the function to be applied

Aggregation over a List

There are two ways in which we can aggregate over a list

- *Accumulate* a value as we work our way down the list

```
let rec accumulate f lst u =
    match lst with
    | [] -> u
    | (h::t) -> accumulate f t (f h u)
```

- Go down the list and then work our way back up *reducing* the list

```
let rec reduce f lst u =
   match lst with
   | [] -> u
   | (h::t) -> f h (reduce f t u)
```

These functions are also called `foldleft` and `foldright`

Question: What are the types of these functions?

Defining List Functions as Aggregation

Many list functions can be defined as aggregation

- Summing up a list

```
let sumList r = reduce (fun x y -> x + y) r 0
```

  Note that the reduce function can also simply be `(+)`

- Finding the length of a list

```
let length r = reduce (fun x y -> y + 1) r 0
```

- Unzipping a list of pairs

```
let unzip r =
    reduce (fun (x1,x2) (y1,y2) ->
                (x1::y1, x2::y2)) r ([],[])
```

- Mapping over a list

```
let map f l =
    reduce (fun x y -> (f x) :: y) l []
```

Combining Mapping and Aggregation

Many data related computations can be viewed as mapping followed by aggregation over lists

E.g. consider categorizing words by their lengths in a corpus

Specifically, given a text represented by the list

```
["the"; "quick"; "brown"; "fox"; "jumps";
        "over"; "the"; "lazy"; "dog"]
```

we want to reduce it into the following list

```
[ (3, ["dog" ; "fox" ; "the" ]);
  (4, ["lazy" ; "over"]) ;
  (5, ["brown" ; "jumps" ; "quick" ])
]
```

Another similar task: counting the number of occurrences of particular words in text

Categorizing Words in Text by Length (Example)

We can view the computation as a two step process:

- Associating a length with each word in the text

  E.g. we transform the given list into

```
[(3, "the"); (5, "quick"); (5, "brown");
 (3, "fox"); (5, "jumps"); (4, "over");
 (3, "the"); (4, "lazy"); (3,"dog")]
```

  This is clearly a *mapping* action

- Collecting all the words of equal length into a list

  This will produce the list we want

```
[ (3, ["dog" ; "fox" ; "the" ]);
  (4, ["lazy" ; "over"]) ;
  (5, ["brown" ; "jumps" ; "quick" ])
]
```

  This is obviously an aggregation or *reduce* action

Putting the Pieces Together (Example)

We mainly need to write the mapping and reducing functions

- The mapping function is easy

```
fun y -> (String.length y, y)
```

- For the reduce function we define

```
addword : int * string ->
            (int * string list) list ->
            (int * string list) list
```

to assimilate a given length, word pair into an existing list

The function to do the full computation is then the following

```
let words_by_length =
  compose (fun l -> reduce addword l [])
          (List.map (fun y -> (String.length y,y)))
```

The Map-Reduce Combination

Many computations over web-based data fit this general paradigm

An example of this kind: computing the common Facebook friends between two friends

In such cases, the mapping operation can be quite complex

Here the implicit parallelism noted in map comes in quite handy: we map in parallel and then aggregate

In fact, general frameworks with “pluggable” map and reduce functions have been developed to exploit this structure

Look on the course website for a paper about the scheduling considerations underlying this kind of framework

Scanning Lists Based on Predicates

Some typical operations on lists of this sort are the following

- Filtering a list using a property

```
let rec filter p l =
     match l with
     | [] -> []
     | (h::t) -> if (p h) then h :: (filter p t)
                 else (filter p t)
```

- Checking if every item in the list satisfies a property

```
let rec forall p l =
   match l with
   | [] -> true
   | (h::t) -> (p h) && (forall p l)
```

- Checking if some item in the list satisfies a property

```
let rec forsome p l =
   match l with
   | [] -> false
   | (h::t) -> (p h) || (forsome p l)
```

Filtering as Map-Reduce

Many of the "list scanning based on predicates" functions can be viewed as instances of the map-reduce paradigm

- Map the predicate over the list to replace each element with a boolean subcomponent
- Aggregate using the boolean subcomponent

For example, we can define `forall` as follows

```
let forall p l =
    reduce (fun x y -> x && y)
           (map p l)
           true
```

If the predicate is costly to compute, this scheme can benefit from frameworks for parallel implementation

Parametric Polymorphism and Functions as Objects

Our treatment of parameterized data structures often assumes the availability of certain operations over the parameter type

For example, binary search trees are determined by

- a data structure for trees over elements of any chosen type
- an equality and ordering relation over the element type
- an invariant concerning the representation that is based on the ordering relation

To correctly define functions over such data, operations over the parameter type must be included in their conceptualization

Any systematic way of doing this must eventually be based on treating the operations as objects

Polymorphism and Type Correctness

A function that has a polymorphic type must work at *any* instance type for true type correctness

However, this means that in the function definition, we must not use operators that work only at certain types

As an example, our current implementation of binary tree functions *does not* have this property and hence is flawed

More specifically, if the binary tree representation is realized by

```
type 'a btree =
  | Empty | Node of 'a * 'a btree * 'a btree
```

then we have the following situation

- we can have binary trees with function data
- trying to "find" a function object in the binary tree will result in a runtime type error (exception)

Type Problems with Binary Tree Representation

In more detail, consider the following interaction

```
# let rec find t i =
  match t with
  | Empty -> false
  | (Node (ind,l,r)) ->
      if (i = ind) then true
      else if (i < ind) then (find l i)
           else (find r i);;
val find : 'a btree -> 'a -> bool = <fun>
# let funbtree = Node ((fun x -> x), Empty, Empty)
val funbtree : ('a -> 'a) btree =
                    Node (<fun>, Empty, Empty)
# find funbtree (fun x -> x);;
Exception: Invalid_argument "equal: functional value".
#
```

In other words, `find` is determined to be polymorphic and it takes `funbtree` as argument only to yield a runtime type error

Using Function Arguments to Solve the Problem

One way to overcome the problem is to include the needed operations as parameters

- the function will have a type that includes the needed operators
- an application of the function will be type correct only if we supply operators that work at the required types

For example, if we define `find` to have the type

```
('a btree) -> 'a ->
    ('a -> 'a -> bool) -> ('a -> 'a -> bool)
            -> bool
```

then, to use it, we will actually have to supply it with equality and comparison operators that work at the item type

Parameterization with Functions (Contd)

In more detail, we would define `find` as follows using this idea

```
let rec find' t i eq lss =
  match t with
  | Empty -> false
  | (Node (ind,l,r)) ->
      if (eq i ind) then true
      else if (lss i ind) then find' l i eq lss
           else find' r i eq lss
```

Now, when we use `find` to look for an item in a tree

- we will have to provide an equality and less-than function
- these functions would themselves have to be well-defined and hence guaranteed to work at the element type

Packaging Data Objects with Function Components

The earlier solution is still unsatisfactory for the following reason

> *Binary search trees themselves need to be parameterized by the ordering relation to be coherent*

In particular, separating the ordering relation from the representation makes the "invariant" statement incoherent

A solution to this problem: combine the ordering relations with the tree representation in the type definition

E.g., we can define a type for binary search trees as follows:

```
type 'a bstree =
  {data : 'a btree;
   eq : 'a -> 'a -> bool; lss : 'a -> 'a -> bool;}
```

Note that the type now carries enough information to state the data invariant completely

Data Objects with Function Components (Contd)

With this kind of packaging, we can correctly realize parametric polymorphism based only on the data object

For example, consider the following interaction

```
# let find {data = t; lss = lss;} i =
    let rec find' t =
      match t with
      | Empty -> false
      | Node (ind,l,r) ->
         if (eq i ind) then true
         else if (lss i ind) then find' l
              else find' r
    in find' t;;
val find : 'a bstree -> 'a -> bool = <fun>
#
```

Note that `find` takes only a tree and an element as argument but type checking now guarantees the absence of type errors

A (Brief) Comparison with Class Definitions

The binary search tree type is like an interface definition in Java

```
type 'a bstree =
  {data : 'a btree;
   eq : 'a -> 'a -> bool; lss : 'a -> 'a -> bool;}
```

An "object" of this type must bring along implementations of the operations described in the type

One slightly unsatisfactory aspect is that we must "build" the object each time by packaging the operations with it

We will see the idea of modules later as a way for realizing a similar functionality without the need for such explicit packaging

Neither of these approaches assumes an inheritance structure, which could actually be a Good Thing

Controlling Evaluation via Higher-Order Functions

There is a default order for evaluating `(e1 + e2)` in OCaml

> *First evaluate* `e1` *then evaluate* `e2` *then add the two values*

Suppose we want to change this so that `e2` is evaluated first and only then is `e1` evaluated

We can realize this by rewriting the expression as follows:

- Transform `e2` into the expression `(fun f -> e2')` where `e2'` calculates `e2` and then "feeds" it to `f`
- Apply this transformed expression to `fun x -> e1 + x`

The function expression used in the second step is indicating how to *continue* evaluation after calculating `e2`

Higher-order functions that have this kind of behaviour are often called *continuations*

Using Continuations (An Example)

Suppose we have the following function definitions

```
let square x = x * x        let twice x = 2 * x
```

Now, in the expression `((twice 3) + (square 5))`, `(twice 3)` will be evaluated first then `(square 5)`

To change this, we can restructure the expression as follows

```
(fun f -> f (square 5)) (fun x -> (twice 3) + x)
```

The same effect is also realized by defining `square'` as

```
let square' x c = c (x * x)
```

and evaluating

```
(square' 5 (fun x -> (twice 3) + x))
```

The `square'` function now takes a continuation as argument

Transforming General Recursion into Tail Recursion

In general recursion, we have to return to the calling location to complete the computation

With functions as objects, we can instead think of passing the rest of the computation to the called function as a continuation

For example, consider the `sumlist` function we saw earlier

```
let rec sumlist lst =
  match lst with
  | [] -> 0
  | (h::t) -> h + (sumlist t)
```

What we will think of doing instead is defining a function

```
cont_sumlist: int list -> (int -> 'a) -> 'a
```

This function will calculate the sum of the items in its list argument and feed the result to its continuation argument

General Recursion to Tail Recursion (Contd)

More specifically, given the definition

```
let rec sumlist lst =
  match lst with
  | [] -> 0
  | (h::t) -> h + (sumlist t)
```

we transform it into the following

```
let rec cont_sumlist lst c =
  match lst with
  | [] -> (c 0)
  | (h::t) -> cont_sumlist t (fun x -> c (h + x))
```

To sum up a list, we would call this function with the identity function as the starting continuation

```
# cont_sumlist [1;2;3;4;5] (fun x -> x);;
- : int = 15
#
```

General Recursion to Tail Recursion (Contd)

Note that the "continuation passing style" does not actually change the structure of the computation

In particular, both of the function calls

```
sumlist [1;2;3]
cont_sumlist [1;2,3] (fun x -> x)
```

will eventually evaluate the expression `1 + 2 + 3 + 0`

This is different from the accumulator based version

```
let rec acc_sumlist lst acc =
  match lst with
  | [] -> acc
  | (h::t) -> acc_sumlist t (h + acc)
```

Here, the call `(acc_sumlist [1;2;3] 0)` will lead to evaluating the expression `(3 + 2 + 1 + 0)`

Modelling Java-Style Programs in OCaml

Three things have to be treated to capture the behaviour of Java programs in OCaml

- We need to represent states as given by the values of variables

  we can represent states using tuples of values

- We need to treat the evaluation of expressions in a given state

  expressions become functions from states (given by tuples) to values

- We need to capture the impact of statements such as assignment or while statements

  statements become functions that take a state and produce a new state, i.e. they are “state transformers”

Modelling State

The idea here is simple: we use tuples of as many components as there are variables in our program

For example, suppose our program uses only the integer variables `X`, `Y` and `Z`

Then we can define the type for states as follows:

```
type state = int * int * int
```

and we designate, for example, `X`, `Y` and `Z` to be the first, second and third components

Thus a state where the values of `X`, `Y` and `Z` are `15`, `6` and `7` would be represented by `(15,6,7)`

To make the correspondence between a variable and tuple item transparent we use `get`functions to lookup variable values, e.g.

```
let getY (x,y,z) = y
```

Encoding Expressions

Expressions denote values but what value they denote depends on the state

Thus, expressions are *functions from states to values*

For example, consider the expression `X + Y * Z`

We can represent this by the function

```
let x_plus_y_times_z =
    fun s -> (getX s) + (getY s) * (getZ s)
```

Examples of evaluating this “expression” in different states

```
# x_plus_y_times_z (5,8,4);;
- : int = 37
# x_plus_y_times_z (14,2,3);;
- : int = 20
#
```

Encoding Assignment Statements

Assignments have two components

- A lefthand side that is a variable
- A righthand side that is an expression

Moreover, their impact is to transform a given state into a new state by changing the value of a variable

One way to model them is to use one function for each variable such as

```
let putX exp =
   fun s -> let (x,y,z) = s in (exp s,y,z)
```

E.g., the assignment `X = X + Y * Z` is represented by

```
putX x_plus_y_times_z
```

The type of this expression is `state -> state` as expected

Encoding Other Statement Forms

All other statements in a Java-like language are obtained by using *statement constructors*, e.g.

- the *sequencing* mechanism for combining two statements into one
- the *if-then-else* constructor for combining two statements and a boolean expression
- the *while-do* constructor for combining a a boolean expression and a statement

We can encode these by using *higher-order functions*

For example, for *sequencing* we would define

```
seq : (state -> state)
        -> (state -> state) -> state -> state
```

Realizing Sequencing and If-Then-Else

- To execute

  ```
  S1; S2;
  ```

  we modify the state using `S1` first and then using `S2`

  But this is simply the *composition* of two state transformers

  ```
  let seq stat1 stat2 =
          fun s -> (stat2 (stat1 s))
  ```

- To execute

  ```
  if exp then stat1 else stat2
  ```

  we evaluate `exp` and use it to decide which of `stat1` and `stat2` to execute

  This translates into the following code

  ```
  let ifstat exp stat1 stat2 =
    fun s -> if (exp s) then (stat1 s)
                 else (stat2 s)
  ```

Encoding the While Statement Constructor

For this, we can use the following “unravelling the loop” idea

```
while exp do stat =
   if exp then
     { stat ; while exp do stat}
   else ()
```

Encoding this in OCaml is now easy

```
let rec whilestat exp stat =
   fun s ->
      ifstat exp
             (seq stat (whilestat exp stat))
             (fun x -> x) s
```

One thing to note: the explicit function construct is needed otherwise the definition will result in an infinite loop

Encoding An Actual Program

To check our understanding, lets try to encode the following program

```
z = 1;
x = 1;
while (x <= y) {
 z = z * x;
 x = x + 1;
}
```

This program calculates the factorial of whatever value is initially stored in `y`

Encoding the Factorial Program

Here is what we get as the encoding of the factorial code

```
let one = fun s -> 1
let z_times_x = (fun s -> (getZ s) * (getX s))
let x_plus_one = (fun s -> (getX s) + 1)
let x_lesseq_y = (fun s -> (getX s) <= (getY s))

let factexp =
    seq (putZ one)
        (seq (putX one)
             (whilestat x_lesseq_y
                        (seq (putZ z_times_x)
                             (putX x_plus_one))))
```

We can also run it as follows:

```
# factexp (0,5,0);;
- : int * int * int = (6, 5, 120)
#
```

Some Additional Exercises

Here are some further things to try:

- Encode other statement constructors like `do-while` and `for`
- Encode other interesting programs in a similar way and try running them

You can find the code for the constructs considered here in the file `code/impprog.ml`