CSci 2041: Advanced Programming Principles
Ordering the Evaluation of Expressions

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Lectures in Spring 2015

The Issue Concerning Evaluation Order

- Without side-effects, expression evaluation is the process of rewriting equals by equals to produce a “normal form”

  E.g., consider the expression *(17 * 2) + (4 / 2)*

- Usually, the order in which we do the rewriting is forced

  E.g., arithmetic expressions require an “inside-out” order

- However, sometimes there is a choice and then the order of rewriting *can be* important

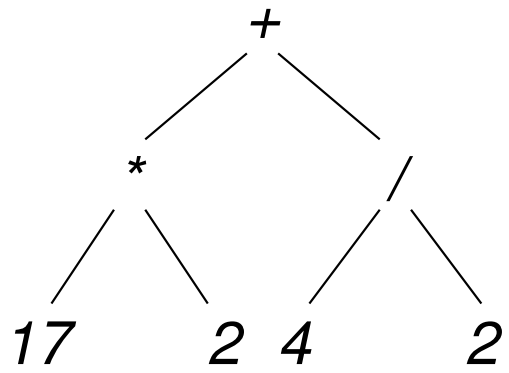
$$\begin{aligned} \mathit{if}(\mathit{true}, E_1, E_2) &\rightarrow E_1 \\ \mathit{if}(\mathit{false}, E_1, E_2) &\rightarrow E_2 \end{aligned}$$

  An outside-in rewriting is required here for recursion

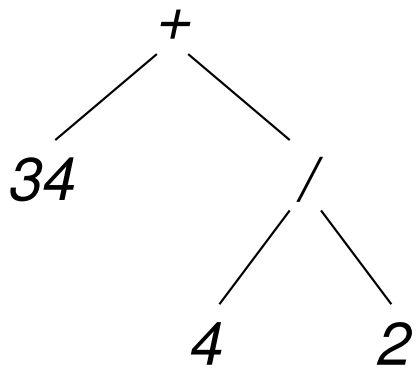
- The issue now: how is computation affected by evaluation and what benefits can be derived by controlling it

Expression Evaluation as Tree Rewriting

We can visualize expressions abstractly as trees



Expression evaluation is then the process of replacing (sub)trees by equivalent ones



The process is repeated till no more replacement is possible, i.e. till a *normal form* or *value* is obtained

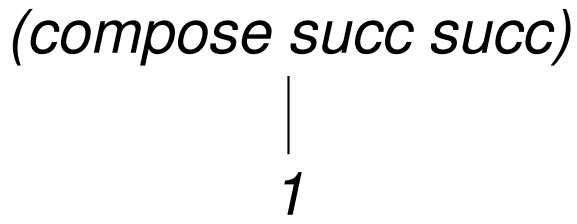
Tree Structures for OCaml Expressions

There is an issue regarding the structure to use when the "operator" can be computed

```
let succ x = x + 1
let compose f g = (fun x -> (f (g x)))
```

What structure to use for *((compose succ succ) 1)*?

The simplest possibility seems to be



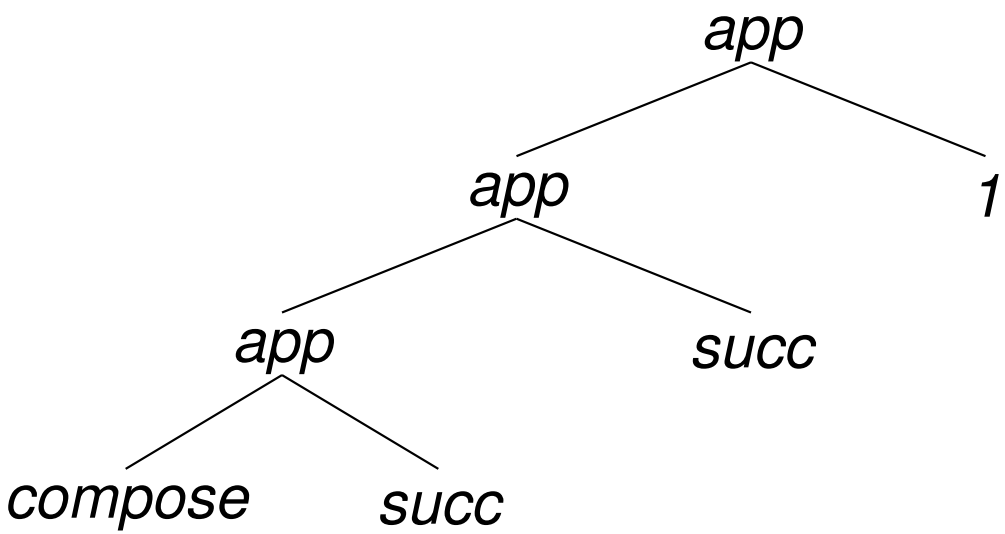
However, this structure is awkward:

- It does not show *(compose succ succ)* as a tree structure that can also be rewritten
- Rewriting seems now to need to change intermediate parts of a tree

Tree Structures for OCaml Expressions (Contd)

The generally accepted way is to make application explicit using a special operator called *app*

Making application explicit in this way yields the following tree structure for *((compose succ succ) 1)*:



When visualized in this way, we have to articulate how to rewrite at an *app* node

This gives rise to different evaluation schemes for expressions

Call-by-Value and Call-by-Name Evaluation

Two common approaches to evaluating expressions like $(e_1\ e_2)$

Call by Value	*Call by Name*
Evaluate e_1 to expose function f (functions constitute values)	Evaluate e_1 to expose function f
Evaluate e_2 to produce value v	Evaluate e_2 only to the extent needed for function application
Apply f to v	Apply f to the result

E.g. consider evaluating `(take 2 (downfrom 3))` given

```
let rec downfrom n =
    match n with
    | 0 -> [] | n -> n :: downfrom (n-1)

let rec take n l =
    match (l,n) with
    | ((_,0) | ([],_)) -> []
    | ((h::t), n) -> h :: (take (n-1) t)
```

Call-by-Value vs Call-by-Name Evaluation (Example)

The sequence of rewritings for call-by-value

```
(take 2 (downfrom 3))  -->
  (take 2 (3 :: (downfrom 2))) -->
  (take 2 (3 :: 2 :: (downfrom 1))) -->
  (take 2 (3 :: 2 :: (downfrom 0))) -->
  (take 2 (3 :: 2 :: 1 :: [])) -->
  (3 :: (take 1 (2 :: 1 :: []))) -->
  (3 :: 2 :: (take 0 (1 :: []))) -->
  (3 :: 2 :: [])
```

The sequence of rewritings for call-by-name

```
(take 2 (downfrom 3)) -->
  (take 2 (3 :: (downfrom 2))) -->
  (3 :: (take 1 (downfrom 2))) -->
  (3 :: (take 1 (2 :: (downfrom 1)))) -->
  (3 :: 2 :: (take 0 ((downfrom 1)))) -->
  (3 :: 2 :: [])
```

Note: We show only the function calls in such sequences, using the *match* and *if-then-else* expressions only implicitly

The Structure of Values

Whether we use call-by-value or call-by-name, expressions must be completely simplified before presentation

The structure of such expressions that are called *values*

- A builtin or user defined constructor, applied again to a requisite number (possibly none) of values, e.g.
  - Builtin constants like `1`, `'a'`, and `"string"`
  - User defined constants like `Empty`, `Mon`, `Red`
  - Expressions with builtin constructors like `1 :: 2 :: nil`
  - Expressions with user-defined constructors like `(Node (1,Empty,Empty))`
- Objects with a function structure at the top-level; the structure of such objects is not explicitly shown

```
# (fun x -> x);;
- : 'a -> 'a = <fun>
#
```

Contrasting Call-by-Value and Call-by-Name

- Call-by-name can avoid computations in arguments if their values are not required as we saw in the last example
- However, call-by-name can repeat computations in arguments if their values are needed several times

  E.g., consider computing `(square (fact 5))` given

```
let square x = x * x

let rec fact n =
  match n with
  | 0 -> 1
  | n -> n * fact (n - 1)
```

  In the naive approach, we will end up computing `(fact 5)` twice

- To avoid repeated computations, we can use graph representations of expressions to realize *sharing*

Terminology and Nuances Concerning Evaluation

- If we reorder computation of the function and argument parts, it is still call-by-value; OCaml does this
- Call-by-value evaluation is also called *applicative order* and *eager evaluation*

  The most common form of evaluation, underlying also languages like C and Java
- Call-by-name is also called *normal-order evaluation*
- Call-by-name evaluation *with sharing* is also called *call-by-need* and *lazy evaluation*

  The most prominent exemplar of this style: Haskell
- Call-by-name/lazy evaluation brings some exciting ideas to programming as we shall presently understand

Strict Versus Non-Strict View of Functions

The two evaluation approaches can produce different results

For example, suppose we have the definitions

```
- fun g x = 5
- fun h x y = h y x
```

Then consider what happens when we evaluate `(g (h 3 3))`

The different possibilities can be justified through recourse to two different views of functions

Strict View Function is defined only for defined arguments

Non-Strict View Results of functions can be known even for arguments that are not defined

The strict view underlies eager evaluation

The non-strict view underlies lazy evaluation

Actually, in a computational setting, there is an argument to be made for taking the non-strict view

Lazy Evaluation and Infinite Objects

Delaying evaluation till needed gives us an effective way to represent and use *infinite objects*

For example consider the following definition

```
let rec nat_nums n = (n :: (nat_nums (n + 1)))
```

Then the expression `(nat_nums 0)` is not meaningful with eager evaluation but *is* fine with lazy evaluation

Furthermore it can even be quite useful, giving us a finite representation of the natural numbers

Thus, suppose we have the function

```
let rec take n l =
  match (l,n) with
  | ((_,0) | ([],_)) -> []
  | ((h::t), n) -> h :: (take (n-1) t)
```

Then consider the expression `(take 5 (nat_nums 0))`

Delaying Evaluation in an Eager Language

The key observation: In eager (as well as lazy) evaluation, we do not look inside function bodies till the function is applied

Thus, we can delay evaluation by "hiding" the expression under a function structure

Since the only purpose for the argument is to indicate when to evaluate the expression, it can be of `unit` type

For example, contrast the following code

```
let x = 3 + 4
in  ... x ...
```

with an alternative version using a "dummy" function

```
let x () = 3 + 4
in  ... x () ...
```

In the second case, the evaluation of `3 + 4` will be delayed till the application

An Abstraction for Delaying in OCaml

We can encapsulate the idea just discussed in the following declarations

```
type 'a delay = Delay of (unit -> 'a)

let mkLazy e = Delay e
let force (Delay e) = e ()
```

Using this abstraction, the earlier example becomes

```
let x = Delay (fun () -> 3 + 4)
in  ... (force x) ...
```

OCaml provides an abstraction of this kind in the *Lazy* library module

Improving the Delaying Abstraction

A problem with the present abstraction is that we evaluate the expression afresh each time we do a lookup

We can improve this by using a reference and updating

```
type 'a delay = 'a delay_aux ref
type 'a delay_aux =
 | Val of 'a
 | Delayed of (unit -> 'a)

let mkLazy e = ref (Delayed e)
let force e =
  match !e with
   | Val v -> v
   | Delayed dv ->
       let v = dv() in (e := Val v; v)
```

Now we evaluate the expression only on demand and we evaluate it only once

Realizing Infinite Objects in Eager Languages

We will realize such objects as *streams*

- A stream represents a *promise* to provide successive components of the data when asked
- When actually asked for a component, it provides it and also a promise for providing the remaining parts

This is a generalization of the *producer-consumer* paradigm realized through *co-routines* in other languages

- a “producer” procedure periodically provides data for a “consumer” procedure to use
- when the consumer procedure needs data, it issues a demand and suspends till the producer provides it

We will only consider *infinite* streams; finite streams can be realized by tweaking our definitions a little

Realizing Streams in OCaml

The following declarations provide an encapsulation of streams

```
type 'a stream =
   Stream of (unit -> 'a * 'a stream)

let mkStream f = Stream f
let nextStream (Stream f) = f ()
```

To "make" a stream, we provide a function that generates the items, one at a time

The interface for using the stream is provided by `nextStream`

For example, we can construct a stream representing the infinite sequence of natural numbers as follows

```
let rec fromN n =
  mkStream (fun () -> (n, from (n+1)))

let natStream = (fromN 1)
```

Using Streams (Example)

We can define the function for "taking" some number of items from a stream as follows:

```
let rec take n s =
   match n with
   |  0 -> []
   |  n ->
        let (x,rst) = nextStream s in
          (x :: take (n-1) rst)
```

Observe the way the consumer function (`take`) periodically issues a request for another item in the stream

An example of the use of the function

```
# take 5 natStream;;
- : int list = [1; 2; 3; 4; 5]
#
```

More Examples of Stream Definitions

- A stream representing the factorial sequence

```
let rec fact n m () =
      (m, Stream (fact (n+1) (n*m)))
let factStream = mkStream (fact 1 1)
```

- A stream representing the fibonnaci sequence

```
let rec fib f s () = (f, Stream (fib s (f+s)))
let fibStream = mkStream (fib 1 1)
```

- A stream that merges two streams of items of the same type

```
let rec merge s1 s2 () =
     let (x,rst) = nextStream s1 in
         (x, Stream (merge s2 rst))
let mergeStreams s1 s2 = Stream (merge s1 s2)
```

A Stream of the Prime Numbers (Example)

Eratosthenes' sieve: Suppose we have an initial sequence of primes in hand

- The first natural number that is not divisible by any of these primes is the next prime
- Once we have the next prime, we add it to the collection of filters for finding yet the next prime

A program that implements this idea must define two things

- A means for filtering a stream of numbers by divisibility by a given number

```
sift : int -> int stream ->
          unit -> int * int stream
```

- A means for picking the next number in a given stream and then filters the rest by divisibility by that number

```
sieve : int stream -> unit -> int * int stream
```

A Stream of the Prime Numbers (Example)

An implementation of the idea

```
let rec sift a s () =
        let (x,rst) = nextStream s in
         if (x mod a = 0) then (sift a rst ())
         else (x, Stream (sift a rst))

let rec siftStream a s = mkStream (sift a s)

let rec sieve s () =
      let (x,rst) = nextStream s in
        (x, Stream (sieve (siftStream x rst)))

let sieveStream s = mkStream (sieve s)

let primesStream = sieveStream (fromNStream 2)
```

Laziness and Efficiency in Computations

The basic idea: by delaying work, you can often avoid it entirely

An Example

Consider trees represented by the following datatype

```
type 'a tree =
    Empty | Leaf of 'a |
    Node of ('a tree) * ('a tree)
```

We call the collection of leaves in left-to-right order in such a tree its *frontier*

Suppose that we want to check if two trees have the same frontier

Here is a way that this might be done

- Collect the data in the leaves into two lists
- Check the two lists for equality

A fine point: might it be better to intersperse the two processes?

Comparing the Frontiers of Trees

A program for comparing trees based on their frontiers

```
let rec frontier t =
   match t with
   | Empty -> []
   | (Leaf x) -> [x]
   | (Node (t1,t2))-> (frontier t1) @ (frontier t2)

let rec equal l1 l2 =
   match (l1,l2) with
   | ([],[]) -> true
   | ((a::x),(b::y)) -> a = b && equal x y
   | (_,_) -> false

let samefrontier x y =
          equal (frontier x) (frontier y)
```

With lazy evaluation, frontier generation and equality checking will in fact take place in lock-step; e.g., consider

```
samefrontier (Node (Leaf 5, Node (Leaf 7, Leaf 3)))
             (Node (Leaf 3, Node (Leaf 7, Leaf 8)))
```

Comparing Trees in an Eager Language

With a little extra effort, the frontier generation and comparison processes can be explicitly interleaved

The key idea: think of comparing *lists* of trees or forests

```
let rec compforests f1 f2 =
   match (f1,f2) with
    | ([],[]) -> true
    | ((Empty::x,y) | (x,Empty::y)) -> compforests x y
    | ((Leaf a)::x),((Leaf b)::y)) ->
          (a=b) && (comforests x y)
    | ((Node (l,r))::x),y) ->
          compforests (l::r::x) y
    | (x,((Node (l,r))::y)) ->
          compforests x (l::r::y)
    |(_,_) -> false

let samefrontier t1 t2 = compforests [t1] [t2]
```

Explicit bookkeeping replaces the implicit one in lazy evaluation

Summarizing Lessons About Expression Evaluation

- There are interesting differences between *cbv/eager* and *cbn/lazy* evaluation, especially with rich expressions
- Each style provides its own take on efficiency
  - Lazy eval automatically avoids unnecessary computations whereas some effort is needed under eager eval
  - Eager eval automatically avoids repeating computations whereas more work is needed for this under lazy eval
- Delayed evaluation is automatic under lazy eval and can be realized by using dummy functions under eager eval
- Delayed evaluation can be used to realize streams that provide an interesting and useful programming capability