CSci 2041: Advanced Programming Principles

Spring 2018, Van Wyk

Expression Evaluation

This document describes and illustrates how we can evaluate expressions by hand by rewriting subexpressions to value or other expressions.

This is done for three evaluation strategies

- call by value semantics, also called eager evaluation
- call by name semantics
- lazy evaluation

We treat clauses in function definitions as rewrite rules. The left hand sides are the patterns that are searched for and the right hand sides replace the matched pattern, instantiating function arguments (the pattern variables) in the process.

We can see call by value semantics as rewriting the expression from the "bottom up" or "inside out." Roughly speaking, inner expressions (sub expressions lower in the tree) are evaluated before outer expressions (those higher up in the tree).

But call by name and lazy evaluation can be seen as rewriting the expression from the "top down" or "outside-in." We will also see how the use of "where" clauses allow us simulate lazy evaluation using terms instead of drawing out directed acyclic graphs.

In both cases we typically go left-to-right for sub expressions at the same level.

A simple example

Consider the following functions.

```
g x = 5
h x y = h y x
f x = g (h x x)
```

In call by value semantics (also known as eager evaluation in functional languages) we must evaluate function arguments down to values before they are used by the function.

Function application is evaluated by rewriting the function call (the function and its arguments) with the function body, replacing the argument variables with their values in the process.

In this example, the rewriting process goes as follows:

$\ \underline{\texttt{f 3}}$
$= \texttt{g (}\underline{\texttt{h 3 3)}}$
$= \texttt{g (}\underline{\texttt{h 3 3)}}$
$= \texttt{g (}\underline{\texttt{h 3 3)}}$
...

The underlined portion is the part that was rewritten. Note that this process does not terminate for this example.

For call by name and lazy evaluation, the rewriting process is the same. It is:

$\ \underline{\texttt{f 3}}$
$= \underline{\texttt{g (h 3 3)}}$
$= \texttt{5}$

A more interesting example

Consider the following definitions for functions `take` and `makefrom`. These are clearly not written in OCaml syntax. Instead there are separate definitions for different values of the input. In OCaml, we would have written `match` expression inside the single function definition instead. The meaning is the same, but this style makes evaluation by rewriting easier to do by hand.

```
take n [] = []
take 0 (x::xs) = []
take n (x::xs) = x::take (n-1) xs

makefrom 0 v = []
makefrom n v = v :: makefrom (n-1) (v+1)
```

Let us now evaluate the expression `take 2 (makefrom 4 5)` under the different evaluate strategies.

Call by value In call by value semantics we must pass fully evaluated values to expressions. Thus we evaluate from the inside-out — looking for innermost expressions to evaluate before enclosing expressions are evaluated.

```
  take 2 (makefrom 4 5) — the initial expression
= take 2 (5 :: makefrom (4-1) (5+1))
= take 2 (5 :: makefrom 3 (5+1))
= take 2 (5 :: makefrom 3 6)
= take 2 (5 :: 6 :: makefrom (3-1) (6+1))
= take 2 (5 :: 6 :: makefrom 2 (6+1))
= take 2 (5 :: 6 :: makefrom 2 7)
= take 2 (5 :: 6 :: 7 :: makefrom (2-1) (7+1))
= take 2 (5 :: 6 :: 7 :: makefrom 1 (7+1))
= take 2 (5 :: 6 :: 7 :: makefrom 1 8)
= take 2 (5 :: 6 :: 7 :: 8 :: makefrom (1-1) (8+1))
= take 2 (5 :: 6 :: 7 :: 8 :: makefrom 0 (8+1))
= take 2 (5 :: 6 :: 7 :: 8 :: makefrom 0 9)
= take 2 (5 :: 6 :: 7 :: 8 :: [])
= 5 :: take (2-1) (6 :: 7 :: 8 :: [])
= 5 :: take 1 (6 :: 7 :: 8 :: [])
= 5 :: 6 :: take (1-1) (7 :: 8 :: [])
= 5 :: 6 :: take 0 (7 :: 8 :: [])
= 5 :: 6 :: []
```

The reduction steps continue until the expressions has been transformed into a *value*.

Call by name In call by name semantics, we work from the outside towards the inside. This means we look for expressions to reduce beginning at the outside or outer-most expression.
First, recall out functions, copied from the previous page:

```
take n [] = []
take 0 (x::xs) = []
take n (x::xs) = x::take (n-1) xs

makefrom 0 v = []
makefrom n v = v :: makefrom (n-1) (v+1)
```

This is the evaluation sequence for call by name semantics:

```
  take 2 (makefrom 4 5)
```
— we must expand `makefrom` to determine if it matches the first pattern of `take` or not, yielding:
```
= take 2 (5 :: makefrom (4-1) (5+1))
```
— a rule for `take` now matches, yielding:
```
= 5 :: take (2-1) (makefrom (4-1) (5+1))
```
— again, `makeform` must be expanded to see if the first `take` clause matches, so evaluate its first argument
```
= 5 :: take (2-1) (makefrom 3 (5+1))
= 5 :: take (2-1) ((5+1) :: makefrom (3-1) ((5+1)+1))
```
— now we need to check the 2nd `take` pattern and thus need to evaluate the first argument
```
= 5 :: take 1 ((5+1) :: makefrom (3-1) ((5+1)+1))
= 5 :: (5+1) :: take (1-1) (makefrom (3-1) ((5+1)+1))
= 5 :: 6 :: take (1-1) (makefrom (3-1) ((5+1)+1))
= 5 :: 6 :: take (1-1) (makefrom 2 ((5+1)+1))
= 5 :: 6 :: take (1-1) ( ((5+1)+1) ::  makefrom (2+1) (((5+1)+1)+1) )
= 5 :: 6 :: take 0 ( ((5+1)+1) ::  makefrom (2+1) (((5+1)+1)+1) )
= 5 :: 6 :: []
```

Lazy Evaluation Lazy evaluation is an optimization of call by name that ensures that expressions, even if they are used more than once inside a function body, are only evaluated once.
First, recall out functions, copied from the previous page:

```
take n [] = []
take 0 (x::xs) = []
take n (x::xs) = x::take (n-1) xs

makefrom 0 v = []
makefrom n v = v :: makefrom (n-1) (v+1)
```

We will create `where` clauses in our text-based evaluations to track expressions passed into functions for a formal argument that occurs more than once in the body of the function.
Thus for evaluations of `take` we will not need these clauses since each formal argument only appears once or zero times. The same is true for the first clause of `makefrom`. But the second clause uses `v` in two places and thus we will create `where` clauses for that one.
For uses of this second clause of `makefrom` we will only create a `where` clause when the actual argument is *not* a value.
This is the evaluation sequence for call by name semantics:

```
  take 2 (makefrom 4 5)
= take 2 (5 :: makefrom (4-1) (5+1))
  — since the argument for v is a value (namely, 5), no where clause is used
= 5 :: take (2-1) (makefrom (4-1) (5+1))
= 5 :: take (2-1) (makefrom 3 (5+1))
= 5 :: take (2-1) (v :: makefrom (3-1) (v+1))
  where v = 5+1
— since the argument for v was an expression and not a value we created a where clause
= 5 :: take 1 (v :: makefrom (3-1) (v+1))
  where v = 5+1
= 5 :: v :: take (1-1) (makefrom (3-1) (v+1))
  where v = 5+1
= 5 :: 6 :: take (1-1) (makefrom (3-1) (6+1))
— since v is now a value we removed the where clause and inlined the value
= 5 :: 6 :: take (1-1) (makefrom 2 (6+1))
= 5 :: 6 :: take (1-1) (v ::  makefrom (2-1) (v+1))
  where v = 6+1
= 5 :: 6 :: take 0 (v ::  makefrom (2-1) (v+1))
  where v = 6+1
= 5 :: 6 :: []
  where v = 6+1
= 5 :: 6 :: []
— we now drop the where clause since we are done, the value of 6+1 was not needed
```

While the previous example showed the creation of `where` clauses it was rather simple since we never needed more than one variable.
Consider the following definitions of `even` and `foo`. Note that both have parameters named `x`.

```
even x = x mod 2 = 0
foo x = if even (x+1) then x+3 else x+4
```

The evaluation proceeds as before for lazy evaluation, but we must be careful about names in this case.

```
  foo (1+2)
= if even (x+1) then x+3 else x+4
  where x = 1+2
= if x' mod 2 = 0 then x+3 else x+4
  where x = 1+2
        x' = x+1
  — we needed to choose a new name for the x in even since it was already used.
= if x' mod 2 = 0 then 3+3 else 3+4
  where x' = 3+1
= if x' mod 2 = 0 then 3+3 else 3+4
  where x' = 4
= if 4 mod 2 = 0 then 3+3 else 3+4
= if 0 = 0 then 3+3 else 3+4
= if true then 3+3 else 3+4
= 3+3
= 6
```

This same need to rename the names used as function arguments also sometimes occurs in recursive functions. Since the “unwinding” of the recursion may result in multiple copies of sub-expressions from the function body being present at once, the formal argument names must be changed.

An easy way to avoid any renaming problems is to always pick a “fresh” (unused) name to use for each function argument when a function is applied.