Evaluation in Functional Languages

Evaluation of an expression in a functional language can be described as a rewriting of the expression into a canonical form with the function definitions as the rewrite rules.

Rewrite a function application by replacing the function application by the body of the function, substituting the actual arguments for formals, and renaming variables if needed.

Example:

Given

```
double x = x + x
```

Evaluate

```
double 4
```

```
double 4 ==>
4+4 ==>
8
```


Evaluation Order

Two possible orders of rewriting:

- Normal order - rewrite the leftmost occurence of a function application.
- Applicative order - rewrite the innermost occurrence of a function application first. (which one first and why?)
- Lazy order evaluation - don't evaluate the parameters of a function until you need them.

Normal order always gives same results as lazy evaluation, but may end up evaluating an expression more times.

Examples: Consider

```
double x = x + x
average x y = (x + y) / 2
```

To avoid confusion about infix notation, let's re-express this as:

```
double x = plus x x
average x y = divide (plus x y) 2
```

Evaluate:

```
double (average 2 4)
```


Example of Normal vs. Applicative Order

Using normal order evaluation:

```
double (average 2 4) =>
plus (average 2 4) (average 2 4) =>
plus (divide (plus 2 4) 2) (average 2 4) =>
plus (divide 6 2) (average 2 4) =>
plus 3 (average 2 4) =>
plus 3 (divide (plus 2 4) 2) =>
plus 3 (divide 6 2) =>
plus 3 3 =>
6
```

Notice that `(average 2 4)` was evaluated twice ... lazy evaluation would cache the results of the first evaluation.

Using applicative order evaluation:

```
double (average 2 4) =>
double (divide (plus 2 4) 2) =>
double (divide 6 2) =>
double 3 =>
plus 3 3 =>
6
```


Different Semantics of Normal vs. Applicative

Now consider:

```
my_if True x y = x
my_if False x y = y
```

Evaluate:

```
my_if (less 3 4) (plus 5 5) (divide 1 0)
```

Normal order evaluation:

```
my_if (less 3 4) (plus 5 5) (divide 1 0) =>
my_if True (plus 5 5) (divide 1 0) =>
(plus 5 5) =>
10
```

Applicative order evaluation:

```
my_if (less 3 4) (plus 5 5) (divide 1 0) =>
my_if True (plus 5 5) (divide 1 0) =>
my_if True 10 (divide 1 0) =>
DIVIDE BY ZERO ERROR
```

Evaluation Order Cont'd

Two Important Properties:

- If there is any evaluation order that will terminate and that will not generate an error, normal order evaluation will terminate and will not generate an error.
- ANY evaluation order that terminates without error will give the same result as any other evaluation order that terminates without error.

A more complicated example

Define:

```
twice f x = f (f x)
s n = plus n 1
```

Produces:

```
twice twice s 0 =>
twice (twice s) 0 =>
(twice s) ((twice s) 0)      ... or dropping the extra parens:
twice s (twice s 0) =>
s (s (twice s 0)) =>
s (s (s (s 0))) =>
s (s (s 1)) =>
s (s 2) =>
s (3) =>
4
```


Twice Example Cont'd

Actually, the previous evalutuation is cheating a bit -- I happen to know that both twice and s are **strict** in both of their arguments, so I switched to applicative order evaluation. The correct order of evaluation is really:

```
twice twice s 0 =>
twice (twice s) 0 =>
(twice s) ((twice s) 0)      ... or dropping the extra parens:
twice s (twice s 0) =>
s (s (twice s 0)) =>
plus (s (twice s 0)) 1 =>
plus (plus (twice s 0) 1) 1 =>
plus (plus (twice s 0) 1) 1 =>
plus (plus (s (s 0)) 1) 1 =>
plus (plus (plus (s 0) 1) 1) 1 =>
plus (plus (plus (plus 0 1) 1) 1) 1 =>
plus (plus (plus 1 1) 1) 1 =>
plus (plus 2 1) 1 =>
plus 3 1 =>
4
```

Definition: a function f is **strict** in an argument if that argument is always evaluated whenever an application of f is evaluated.

If a function is strict in an argument, we can safely evaluate the argument first if we need the value of applying the function.

Lazy Evalution and Stricness Analysis

We can use lazy evaluation on an ad hoc basis (e.g. for `if`), or for all arguments. If for all arguments, we can improve efficiency using strictness analysis.

`plus a b` is strict in both arguments

`if x y z` is strict in x, but not in y and z

We can do some analysis and sometimes decide if a user-defined function is strict in some of its arguments:

Examples:

`double x`
is strict in x

`squid n x = if n=0 then x+1 else x-n`
is strict in n and x

`crab n x = if n=0 then x+1 else n`
is strict in n but not x

If a function is strict in an argument x, it is correct to pass x by value (even with normal order evaluation semantics).