Operator Calculus

Functional programming and mathematical programming are similar in that both support a formal semantics. Here are some examples of lambda calculus abstraction and evaluation.

```
F1[x] = +[x,1]                      x+1

F2[x,y] = *[x,-[2,y]]               x(2-y)
```

Abstract F1:

```
F1[x] = +[x,1]                x-->#

F1 = [#,+1 #]
```

Apply F1:

```
F1[3] = [#,+1 #] 3 = (subst 3 # (+1 #)) = +1 3 = 4
```

Abstract F2:

```
F2[x,y] = *[x,-[2,y]]         x-->%  y-->@

F2 = [%, *% (-2 y)] = [@, [%, *% (-2 @)]]
```

Apply F2:

```
F2[3,8] = [@, [%, *% (-2 @)]] 3 8 = [@, [%, *% (-2 @)] 3] 8
```

Note: The binding `x=3` must descend inward so that the abstraction `x=%` is paired correctly with the binding `3`. The direct descent used above is *non-standard* (I have changed the notation slightly so that the rule is easier to follow).

There are two possible evaluation orders, *normal* and *applicative*. These align with the mathematical and accumulative forms of recursion.

Normal evaluation expands functions within functions, without accumulating intermediate results. When all functions are evaluated, the entire expression is then simplified. Should some bindings not be available, normal evaluation returns the remaining function. In normal order, a function that is not fully defined could be passed for expansion (lenient semantics).

Applicative evaluation is standard in programming languages. Innermost functions are evaluated, reduced to ground, and then that reduced result is handed to the next outer function. In applicative order, all variables to be evaluated are guaranteed to be bound (strict semantics).

Evaluation strategies can be mixed, any step can be of either type. As with all algebraic languages, the order of application of substitutions does not matter.

Applicative Order, innermost leftmost first (data-driven, eager, call-by-value)

```
F2[3,8] = [@, [%, *% (-2 @)] 3] 8

        = [@, (subst 3 % (*% (-2 @))] 8

        = [@, (*3 (-2 @))] 8

        = (subst 8 @ (*3 (-2 @))

        = (*3 (-2 8)) = *3 -6 = -18
```

Normal Order, outermost leftmost first (demand-driven, lazy, call-by-need)

```
F2[3,8] = [@, [%, *% (-2 @)] 3] 8

        = (subst 8 @ [%, *% (-2 @)]) 3

        = [%, *% (-2 8)] 3

        = (subst 3 % (*% (-2 8)))

        = (*3 (-2 8)) = -18
```

Composing `F1` and `F2`:

```
F1[F2[x,y]]

F1 = [#,+1 #]
F2 = [@, [%, *% (-2 @)]]

F3 = F1[F2] = [#,+1 #] [@, [%, *% (-2 @)]]
```

Evaluating `F3`:

```
F3[3,8] = [#,+1 #] [@, [%, *% (-2 @)]] 3 8

        = [#,+1 #] [@, [%, *% (-2 @)] 3] 8
```

Note that there are three forms in sequence: `F1 F2 8.` We could apply `F1` to `F2` first (normal), or we could apply `F2` to `8` (mixed), or we could apply the inner abstraction in `F2` to `3` (applicative).

Applicative Order:

```
F3[3,8] = [#,+1 #] [@, [%, *% (-2 @)] 3] 8

        = [#,+1 #] [@, (subst 3 % (*% (-2 @)))] 8

        = [#,+1 #] [@, (*3 (-2 @))] 8

        = (subst [@, (*3 (-2 @))] # (+1 #)) 8

        = +1 [@, (*3 (-2 @))] 8

        = +1 (subst 8 @ (*3 (-2 @))

        = +1 (*3 (-2 8)) = -17
```

Normal Order:

```
F3[3,8] = [#,+1 #] [@, [%, *% (-2 @)] 3] 8

        = [#,+1 #] (subst 8 @ [%, *% (-2 @)]) 3

        = [#,+1 #] [%, *% (-2 8)] 3

        = [#,+1 #] (subst 3 % (*% -6))

        = [#,+1 #] (*3 -6)

        = [#,+1 #] -18

        = (subst -18 # (+1 #))

        = +1 -18 = -17
```

In the following example of the recursive factorial function, the `Y` combinator in lambda calculus specifies the recursive use of a function. Its behavior is to substitute the function definition whenever an operator hole is filled with the name of the recursive function.

```
FACTORIAL:     (define FAC (n) (if n=0 then 1 else (* n (FAC (-1 n)))))

Abstract  n --> %   FAC --> Y #

FAC = Y [#, [%, (COND (=0 %) 1 (* % (# (-1 %))))]]

FAC 2 = Y [#, [%, (COND (=0 %) 1 (* % (# (-1 %))))]] 2

      = Y [#, [%, (COND (=0 %) 1 (* % (# (-1 %))))] 2]

      = Y [#, (subst 2 % (COND (=0 %) 1 (* % (# (-1 %)))))]

      = Y [#, (COND (=0 2) 1 (* 2 (# (-1 2))))]

      = Y [#, (COND False 1 (*2 (# 1)))]

      = Y [#, (*2 (# 1))]

      = (subst FAC # (*2 (# 1)))

      = *2 (FAC 1)

      = *2 (Y [#, [%, (COND (=0 %) 1 (* % (# (-1 %))))]] 1)

      = *2 Y [#, [%, (COND (=0 %) 1 (* % (# (-1 %))))] 1]

      = *2 Y [#, (subst 1 % (COND (=0 %) 1 (* % (# (-1 %)))))]

      = *2 Y [#, (COND (=0 1) 1 (* 1 (# (-1 1))))]

      = *2 Y [#, (COND False 1 (* 1 (# 0)))]

      = *2 Y [#, (*1 (# 0))]

      = *2 (subst FAC # (*1 (# 0)))

      = *2 (*1 (FAC 0))

      = *2 (*1 (Y [#, [%, (COND (=0 %) 1 (* % (# (-1 %))))]] 0))

      = *2 *1 Y [#, [%, (COND (=0 %) 1 (* % (# (-1 %))))] 0]

      = *2 *1 Y [#, (subst 0 % (COND (=0 %) 1 (* % (# (-1 %)))))]

      = *2 *1 Y [#, (COND (=0 0) 1 (* 0 (# (-1 0))))]

      = *2 *1 Y [#, (COND True 1 (*0 (# -1)))]

      = *2 *1 Y [#, 1]

      = *2 *1 (subst FAC # 1)

      = *2 *1 1 = 2
```