



Tail Recursion

Principles of Functional Programming

Review: Evaluating a Function Application

One simple rule : One evaluates a function application $f(e_1, \dots, e_n)$

- ▶ by evaluating the expressions e_1, \dots, e_n resulting in the values v_1, \dots, v_n , then
- ▶ by replacing the application with the body of the function f , in which
- ▶ the actual parameters v_1, \dots, v_n replace the formal parameters of f .

Application Rewriting Rule

This can be formalized as a *rewriting of the program itself*:

$$\begin{array}{l} \text{def } f(x_1, \dots, x_n) = B; \dots f(v_1, \dots, v_n) \\ \rightarrow \\ \text{def } f(x_1, \dots, x_n) = B; \dots [v_1/x_1, \dots, v_n/x_n] B \end{array}$$

Here, $[v_1/x_1, \dots, v_n/x_n] B$ means:

The expression B in which all occurrences of x_i have been replaced by v_i .

$[v_1/x_1, \dots, v_n/x_n]$ is called a *substitution*.

Rewriting example:

Consider gcd, the function that computes the greatest common divisor of two numbers.

Here's an implementation of gcd using Euclid's algorithm.

```
def gcd(a: Int, b: Int): Int =  
  if b == 0 then a else gcd(b, a % b)
```

Rewriting example:

$\text{gcd}(14, 21)$ is evaluated as follows:

$\text{gcd}(14, 21)$

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

`gcd(14, 21)`

→ `if 21 == 0 then 14 else gcd(21, 14 % 21)`

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

`gcd(14, 21)`

→ `if 21 == 0 then 14 else gcd(21, 14 % 21)`

→ `if false then 14 else gcd(21, 14 % 21)`

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

`gcd(14, 21)`

→ `if 21 == 0 then 14 else gcd(21, 14 % 21)`

→ `if false then 14 else gcd(21, 14 % 21)`

→ `gcd(21, 14 % 21)`

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

`gcd(14, 21)`

→ `if 21 == 0 then 14 else gcd(21, 14 % 21)`

→ `if false then 14 else gcd(21, 14 % 21)`

→ `gcd(21, 14 % 21)`

→ `gcd(21, 14)`

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

`gcd(14, 21)`

→ `if 21 == 0 then 14 else gcd(21, 14 % 21)`

→ `if false then 14 else gcd(21, 14 % 21)`

→ `gcd(21, 14 % 21)`

→ `gcd(21, 14)`

→ `if 14 == 0 then 21 else gcd(14, 21 % 14)`

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

`gcd(14, 21)`

→ `if 21 == 0 then 14 else gcd(21, 14 % 21)`

→ `if false then 14 else gcd(21, 14 % 21)`

→ `gcd(21, 14 % 21)`

→ `gcd(21, 14)`

→ `if 14 == 0 then 21 else gcd(14, 21 % 14)`

→ `gcd(14, 7)`

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

`gcd(14, 21)`

→ `if 21 == 0 then 14 else gcd(21, 14 % 21)`

→ `if false then 14 else gcd(21, 14 % 21)`

→ `gcd(21, 14 % 21)`

→ `gcd(21, 14)`

→ `if 14 == 0 then 21 else gcd(14, 21 % 14)`

→ `gcd(14, 7)`

→ `gcd(7, 0)`

Rewriting example:

gcd(14, 21) is evaluated as follows:

gcd(14, 21)

→ if 21 == 0 then 14 else gcd(21, 14 % 21)

→ if false then 14 else gcd(21, 14 % 21)

→ gcd(21, 14 % 21)

→ gcd(21, 14)

→ if 14 == 0 then 21 else gcd(14, 21 % 14)

→ gcd(14, 7)

→ gcd(7, 0)

→ if 0 == 0 then 7 else gcd(0, 7 % 0)

Rewriting example:

gcd(14, 21) is evaluated as follows:

gcd(14, 21)

→ if 21 == 0 then 14 else gcd(21, 14 % 21)

→ if false then 14 else gcd(21, 14 % 21)

→ gcd(21, 14 % 21)

→ gcd(21, 14)

→ if 14 == 0 then 21 else gcd(14, 21 % 14)

→ gcd(14, 7)

→ gcd(7, 0)

→ if 0 == 0 then 7 else gcd(0, 7 % 0)

→ 7

Another rewriting example:

Consider factorial:

```
def factorial(n: Int): Int =  
  if n == 0 then 1 else n * factorial(n - 1)
```

factorial(4)

Another rewriting example:

Consider factorial:

```
def factorial(n: Int): Int =  
  if n == 0 then 1 else n * factorial(n - 1)
```

factorial(4)

→ if 4 == 0 then 1 else 4 * factorial(4 - 1) \rightarrow 4 * factorial(3)

Another rewriting example:

Consider factorial:

```
def factorial(n: Int): Int =  
  if n == 0 then 1 else n * factorial(n - 1)
```

factorial(4)

→ if 4 == 0 then 1 else 4 * factorial(4 - 1) \rightarrow 4 * factorial(3)

\rightarrow 4 * (3 * factorial(2))

Another rewriting example:

Consider factorial:

```
def factorial(n: Int): Int =  
  if n == 0 then 1 else n * factorial(n - 1)
```

factorial(4)

→ if 4 == 0 then 1 else 4 * factorial(4 - 1) \rightarrow 4 * factorial(3)

\rightarrow 4 * (3 * factorial(2))

\rightarrow 4 * (3 * (2 * factorial(1)))

Another rewriting example:

Consider factorial:

```
def factorial(n: Int): Int =  
  if n == 0 then 1 else n * factorial(n - 1)
```

factorial(4)

→ if 4 == 0 then 1 else 4 * factorial(4 - 1) \rightarrow 4 * factorial(3)

\rightarrow 4 * (3 * factorial(2))

\rightarrow 4 * (3 * (2 * factorial(1)))

\rightarrow 4 * (3 * (2 * (1 * factorial(0))))

Another rewriting example:

Consider factorial:

```
def factorial(n: Int): Int =  
  if n == 0 then 1 else n * factorial(n - 1)
```

factorial(4)

→ if 4 == 0 then 1 else 4 * factorial(4 - 1) β -> → 4 * factorial(3)

→ 4 * (3 * factorial(2))

→ 4 * (3 * (2 * factorial(1)))

→ 4 * (3 * (2 * (1 * factorial(0))))

→ 4 * (3 * (2 * (1 * 1)))

Another rewriting example:

Consider factorial:

```
def factorial(n: Int): Int =  
  if n == 0 then 1 else n * factorial(n - 1)
```

factorial(4)

→ if 4 == 0 then 1 else 4 * factorial(4 - 1) \rightarrow 4 * factorial(3)

\rightarrow 4 * (3 * factorial(2))

\rightarrow 4 * (3 * (2 * factorial(1)))

\rightarrow 4 * (3 * (2 * (1 * factorial(0))))

\rightarrow 4 * (3 * (2 * (1 * 1)))

\rightarrow 24

What are the differences between the two sequences?

Tail Recursion

Implementation Consideration: If a function calls itself as its last action, the function's stack frame can be reused. This is called *tail recursion*.

⇒ Tail recursive functions are iterative processes.

In general, if the last action of a function consists of calling a function (which may be the same), one stack frame would be sufficient for both functions. Such calls are called *tail-calls*.

Tail Recursion in Scala

In Scala, only directly recursive calls to the current function are optimized.

One can require that a function is tail-recursive using a `@tailrec` annotation:

```
@tailrec  
def gcd(a: Int, b: Int): Int = ...
```

If the annotation is given, and the implementation of `gcd` were not tail recursive, an error would be issued.

Exercise: Tail recursion

Design a tail recursive version of factorial.