Tail Recursion

Review: Evaluating a Function Application

One simple rule : One evaluates a function application $\mathtt{f}(e_1, ..., e_n)$

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

Application Rewriting Rule

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

$$\begin{aligned} &\mathtt{def}\ \mathtt{f}(\mathtt{x}_1, ..., \mathtt{x}_n) = \mathtt{B};\ \ ...\ \mathtt{f}(\mathtt{v}_1, ..., \mathtt{v}_n) \\ \rightarrow \quad & \\ &\mathtt{def}\ \mathtt{f}(\mathtt{x}_1, ..., \mathtt{x}_n) = \mathtt{B};\ \ ...\ [\mathtt{v}_1/\mathtt{x}_1, ..., \mathtt{v}_n/\mathtt{x}_n]\,\mathtt{B} \end{aligned}$$

Here, $[\mathtt{v}_1/\mathtt{x}_1, ..., \mathtt{v}_n/\mathtt{x}_n]\,\mathtt{B}$ means:

The expression $\mathtt{B}$ in which all occurrences of $\mathtt{x}_\mathtt{i}$ have been replaced by $\mathtt{v}_\mathtt{i}$.

$[\mathtt{v}_1/\mathtt{x}_1, ..., \mathtt{v}_n/\mathtt{x}_n]$ 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) a else gcd(b, a % b)
```

Rewriting example:

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

```
gcd(14, 21)
```

Rewriting example:

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

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

Rewriting example:

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

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

Rewriting example:

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

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

Rewriting example:

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

```
gcd(14, 21)
→ if (21 == 0) 14 else gcd(21, 14 % 21)
→ if (false) 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if (14 == 0) 21 else gcd(14, 21 % 14)
→→ gcd(14, 7)
```

Rewriting example:

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

```
gcd(14, 21)
→ if (21 == 0) 14 else gcd(21, 14 % 21)
→ if (false) 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if (14 == 0) 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) 14 else gcd(21, 14 % 21)
→ if (false) 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if (14 == 0) 21 else gcd(14, 21 % 14)
↠ gcd(14, 7)
↠ gcd(7, 0)
→ if (0 == 0) 7 else gcd(0, 7 % 0)
```

Rewriting example:

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

```
gcd(14, 21)
→ if (21 == 0) 14 else gcd(21, 14 % 21)
→ if (false) 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if (14 == 0) 21 else gcd(14, 21 % 14)
↠ gcd(14, 7)
↠ gcd(7, 0)
→ if (0 == 0) 7 else gcd(0, 7 % 0)
→ 7
```

Another rewriting example:

Consider `factorial`:

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

```
factorial(4)
```

Another rewriting example:

Consider `factorial`:

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

```
factorial(4)
```

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

Another rewriting example:

Consider `factorial`:

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

`factorial(4)`

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

Another rewriting example:

Consider `factorial`:

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

`factorial(4)`

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

Another rewriting example:

Consider `factorial`:

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

`factorial(4)`

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

Another rewriting example:

Consider `factorial`:

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

`factorial(4)`

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * factorial(0)))`

Another rewriting example:

Consider `factorial`:

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

```
factorial(4)
```

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * factorial(0))))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * 1)))`

Another rewriting example:

Consider `factorial`:

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

`factorial(4)`

$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * factorial(0)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * 1)))`

$\twoheadrightarrow$ `120`

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*.

$\Rightarrow$ 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:

```scala
@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`.