Functional Programming – Part 1: General Concepts

SYNTAX OF FUNCTIONAL PROGRAMS

The notation is inspired by the mathematical definition of functions, using equations.

EXAMPLE – We can compute the square of a number using a function **square** defined by the equation **square x = x * x**

EXECUTION OF PROGRAMS

The role of the computer is to evaluate and display the results of the expressions that the programmer writes, using the available functions (like a sophisticated calculator)

EXAMPLE – When we type an expression, such as **square 6** the computer will display its result 36

As in mathematics, expressions may contain *numbers, variables* or *names of functions. For instance, **36, square 6 and x * x** are expressions*

EVALUATION

*To evaluate **square 6*** to the computer will use the definition of **square** and replace this expression by **6 * 6**. Then using the predefined ***** operation, it will find the result 36.

THE PROCESS OF EVALUATING AN EXPRESSION IS A **SIMPLIFICATION PROCESS,** also called **REDUCTION***, or* **EVALUATION PROCESS.**

The goal is to obtain the *value* or *normal form* associated to the expression, by a series of reduction steps.

For every evaluation will always give the same answer therefore is pure language

The meaning of an expression is its value

EXAMPLES OF EVALUATION

Example:

- Square 6 -> 6 * 6 -> 36

Example:

- ((3 + 1) + (2 + 1)) -> ((3 + 1) + (3)) -> (4 + 3) -> 7 – 7 is the value denoted by the expression ((3+1) + (2 + 1))
  There may be serval reduction sequences for an expression, for example this expression has another evaluation/reduction which is also correct
- ((3+1) + (2 + 1)) -> (4 + (2 + 1) -> (4 + 3) -> 7

In both cases the value is the same.

PROPERTIES OF EVALUATION

1. **UNICITY OF NORMAL FORMS:** *pure functional languages* (like Haskell) the value of an expression is uniquely determined by its components and is independent of the order of reduction.
2. **NON-TERMINATION**: Not all reduction sequences lead to a value some reduction sequences do not terminate.

MORE EXAMPLES OF EVALUATION

Example:

Let us define a constant function: **fortytwo x = 42** and another function: **infinity = infinity + 1**

- The evaluation of **infinity** never reaches a normal form, it will not terminate, it is defined in terms of itself.
- For the expression **fortytwo infinity** some reduction sequences do not terminate, but those which terminate give the value 42 (unicity of normal forms) – which is dependent on the strategy. And this is if we were to use call-by-value strategy we would get stuck in the infinite loop, however if we were to use call by name, fortytwo function would straight away print out it normal form.

STRATEGIES

Although the normal form is unique, the order of reduction is important

The Strategy of evaluation defines the reduction sequences that the language implements. Most popular strategies:

- *Call-by-name **(normal order):*** reduce first the application using the definition of the function and then the argument,
- *Call-by-name **straight away takes values like in the example 4 before evaluating the parameters, which in some cases my save trouble from non-terminating programs***

When the function is as so:
def f(x:Int, y:Int, z:Int) = x+x
When: x=(4), y=(7-4), z=(2+3)

- Call-by-Value:
  f(4,3,5) -> 4+4 -> 8 [3 steps]
- Call-by-Name:
  (4+4) -> 8 [2 steps]

- *Call*-by-value **(*applicative order)**:* evaluate first the argument and then the application using the definition of the function.
- Call-by-value **evaluates the parameters first, and then performs computation.**

When the function is as so:
def random(i : Int) = (i, i, i)
When: i = util.random.nextInt(10) /*random number between 0 and 9*/

- Call-by-Value:
  (3, 3, 3) /*same random value for each value of i*/
- Call-by-Name:
  (7, 1, 4) /*new random number is generated for each value of i*/

- Different strategies require different number of reduction steps (efficiency)
- Call-by-name **ALWAYS FINDS THE VALUE IF THERE IS ONE.**
  SOMETIMES MAY be wasteful SQUARE (3 + 3) could evaluate (3+3) twice.
- Call-by-value is **IN GENERAL MORE EFFICIENT, BUT MAY FAIL TO FIND A VALUE**
- Haskell uses a strategy called *lazy evaluation* which guarantees that if an expression has a normal form evaluator will find it.
- **LAZY EVALUATION = CALL-BY-NAME + SHARING**
- **Sharing** means that temporary data is physically stored if it is used multiple times. And does not need to be evaluated again.

FUNCTIONS VALUES

Technically functions are also values, even tough we cannot display them or print them.

As in mathematics a function is a mapping that associates to each element of a given type ***A***, called the ***domain*** of the function, and element of type ***B, codomain.***

f : *A -> B*

If a function *f* of type *A -> B* is applied to an argument x of type *A*, it gives a result (f x) of type *B.*

- *Here the syntax to associate a type to a function is:*
  - *square :: Int -> Int*
  - *fortytwo :: Int -> Int*

Functions are defined in terms of equations.

EXAMPLE:

- square x = x * x
- min x y = if x < y then x else y

GUARDED EQUATION EXAMPLE:

```
sign x
    | x < 0 = -1
    | x == 0 = 0
    | x > 0 = 1
```

- This is equivalent to but clearer than**: if x < - then -1 else if x == 0 then 0 else 1.**

FUNCTION APPLICATION

Application is denoted by juxtaposition(zestawienie): (f x)

EXAMPLE:

- (square 3)
- (sign -5)
- (min 3 -5)

To avoid writing too many brackets there are some conventions:

- Applications has precedence over other operations " **square 3 + 1** " means **(square 3) + 1.** Therefore, the precedence in this one will return 10 where if we were to add brackets around **(3 + 1)** and not square we would get 16.
- Application associates to the left: **square square 3** means **(square 3) + 1**
- We will not write the outermost brackets: **square 3** instead of **(square 3)**

RECURSIVE DEFINITIONS

In the definition of a function " f " we can use the function f:

- fact :: Int -> Int
- fact n = if n == 0 then 1 else n * (fact (n – 1))

Recursive definitions are evaluated by simplification, as any other expression.

EXAMPLE:

fact 0 -> if 0 == 0 = then 1 else 0 * (fact (n – 1))

-> if True then 1 else 0 * (fact (n – 1))

-> 1

Note the operational semantics of the conditional:

1. First evaluate the Boolean condition
2. If the result is true, then evaluate only the expression in the left branch (then)
3. If the result is false, then evaluate only the expression in the right branch (else)

Therefore, the only reduction sequence for the expression **fact 0** is the one shown above, and this program is terminating. However, if we start **fact -1,** the reduction sequence does not terminate to avoid this problem we should write:

```
fact ::  Int -> Int
fact n
    | n > 0 = n * (fact (n - 1))
    | n == 0 = 1
    | n < 0 = error "negative argument"
```

Error is a predefined function that takes a string as argument. When evaluated it causes immediate termination of the evaluator and displays the string.

FUNCTION DEFINITIONS BY **PATTERN MATCHING**

*Upon evaluation the argument of a function is compared against those of the possible definitions, **in order they are provided**, until either a match is found, or the patterns are **exhausted**(we matched all definitions with no success)*

For example, the previous definition of factorial could instead be written like this

Example

```
fact ::  Int -> Int
fact 0 = 1
fact n = n * (fact (n - 1))
```

EXERCISE

> Write three definitions, one using a conditional, one using guarded equations, and one using pattern matching, for a function `fib n` that gives the *n*th value of the Fibonnaci sequence, in which each value is the sum of the two previous values, that is:
>
> $$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

Conditional

```
fibb x = if x == 1 then 1 else if x == 2 then 1 else fib(x - 1) + fib(x - 2)
```

Guarded

```
fibb x
     | x == 1 = 1
     | x == 2 = 1
     | fib(x - 1) + fib(x - 2)
```

Recursive

```
fibb 1 = 1
fibb 2 = 1
fibb x = fib(x - 1) + fib(x - 2)
```

LOCAL DEFINITIONS

As in mathematics, we can write local variables with use of where or let/in:

F X = A + 1 WHERE A = X / 2 OR EQUIVALENTLY F X = LET A = X / 2 IN A + 1

The syntax ***let …. in*** and ***where*** are used to introduce local definitions, available just on the right-hand side of the equation that we are writing.

We can write serval local definitions:

f x = SQUARE (successor x) where SQUARE z = z * z; successor x = x + 1

EVALUATION OF LOCAL DEFINITION

e[a] where **a** = **e'**

1. Evaluate **e'**, obtaining a result **r**
2. Replace **a** by **r** in **e**
3. Evaluate **e**{a -> r}

Example: magnitude a b = sqrt (asq + bsq) where asq = a * a; bsq = b * b

```
magnitude 3 4 -> sqrt (asq + bsq) where asq = a * a; bsq = b * b
              -> sqrt (asq + bsq) where asq = 3 * 3; bsq = 4 * 4
              -> sqrt (asq + bsq) where asq = 9; bsq = 4 * 4
              -> sqrt (9 + bsq) where bsq = 4 * 4
              -> sqrt (9 + bsq) where bsq = 16
              -> sqrt (9 + 16)
              -> sqrt (25)
              -> 5
```

ARITHMETIC

- **Arithmetic operators are primitive functions usually written using infix notation, 3 + 4**
- We can hover use them in prefix notation if we enclose them in brackets for example "(+) 4 5 "
- Standard functions may be made infix by quoting with backticks for example 10 'div' 2
- **+, -, *, /** are left associative: for example**, 3 – 1 – 2** means **(3 – 1) – 2**
- Function application takes precedence: square **1 + 4 * 2** means **(square 1) + (4 * 2)**

FIRST CLASS FUNCTIONS

Just like in objective languages the objects are main constructs in functional languages we use functions as our main constructs.

Functions may take functions as arguments/parameters or return functions as values/output

For example, functional composition is itself a predefined function written infix as ***" f . g "***

This means that we can use function within function if it has the same type. Like in a quad example.

- DOT "." Syntax used to describe composition of functions.

```
(.)  ::  (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

We can only compose functions whose types match.

Example

```
square ::  Int -> Int
square x = x * x

quad ::  Int -> Int
quad = square .  square
```

Curried functions and partial application

All functions in Haskell take only one argument. Applying a function of ***n > 1*** arguments yield another function with ***n – 1*** arguments. This allows one to create new functions by partial application.

- All arithmetic operators in Haskell are also curried functions

The alternative is to pass multiple arguments as one in a tuple but this does not allow partial application.

Example

```
plusTuple (Int, Int) -> Int
plusTuple (x, y) = x + y
```

We can convert between curried and uncurried functions using the primitive functions curry and uncurry.

```
curry ::  ((a, b) -> c) -> (a -> b -> c)
uncurry ::  (a -> b -> c) -> ((a, b) -> c)
```

Example

```
plus ::  Int -> Int -> Int
plus = curry plusTuple
```

Example

```
plus ::  Int -> Int -> Int
plus x y = x + y
plus 3 ::  Int -> Int

plusThree ::  Int -> Int
plusThree y = 3 + y
plusThree 2 = 5

(plus 3) 2 ::  Int
(plus 3) 2 = 5
```

Example

```
(*) ::  Int -> Int -> Int
(*) 6 ::  Int -> Int
((*) 6) 4 = 24

(+) ::  Int -> Int -> Int
(+) 1 ::  Int -> Int
((+) 1) 4 = 5
```