Functional Programming in Perl 6

The Perl Conference in Amsterdam, 9-11 August 2017

Laurent Rosenfeld

Let’s Stop Writing C in Perl

An old saying is that Fortran programmers can write Fortran programs in any language

Let’s Stop Writing C in Perl (2)

- In his excellent book, *Higher Order Perl* (*HOP*), Mark Jason Dominus writes that Perl programmers have been writing C programs in Perl
  - This a a pity, because Perl is far more expressive than C and we can do things that a C programmer would not even dream about
  - Perl uses a lot of concepts borrowed from functional programming (esp. Lisp) making it possible to write shorter and more expressive programs
  - In fact, Perl is semantically closer to Lisp than to C

Applying HOP to Perl 6

- Mark Jason Dominus's *HOP* is about Perl 5
- It has truly inspired me and has changed my way of programming (not only in Perl)

Applying HOP to Perl 6 (2)

- Today, I want to do something similar to *HOP* with Perl 6
  - In 45 minutes, this will have to be much lighter
  - Part of this talk also applies to P5 (although with a slightly different syntax)
  - But P6 has many more functional programming features than P5

What is Functional Programming?

- Functional programming is about:
  - A programming paradigm that treats computation as the evaluation of mathematical functions

What is Functional Programming?

- Functional programming is about:
  - A declarative paradigm done with declarations and expressions rather than statements or instructions: *composing answers*, rather than listing statements

What is Functional Programming?

- Functional programming is about:
  - The output value of a subroutine depends only on the input arguments passed to it (“pure functions”)

What is Functional Programming?

- Functional programming is about:
  - No side-effect,
  - Stateless programs, immutable values, symbolic computation

What is Functional Programming?

- Functional programming is about:
  - Higher order functions, list processing, iterators, pipeline programming, anonymous functions, lambdas, closures, function factories, dispatch tables, lazy lists, currying, etc.

List Operators

- Many Perl operators work on or produce lists:
  - join, split, comb
  - print, say
  - sort
  - for
  - map, grep, reduce, gather … take
  - Meta-operators, hyper-operators, etc.

List Operators (2)

- List operators can be chained to create a data pipeline
- For example, find even numbers smaller than 15:

```
say join '-', map {$_ * 2}, 1..7;    # 2-4-6-8-…-14

# or:
say join '-', grep {$_ %% 2}, 1..15; # 2-4-6-8-…-14

# or:
1..15 ==> grep { $_ %% 2} ==> join "-" ==> say $_;


# or:
(1..7).map({$_ * 2}).say; # Method invocation chain
```

List Operators (3)

- Functions like map or grep are interesting:
  - Because they use a code block (or a subroutine) to define what they do to the data; they're abstract generic functions
  - When a code block or a subroutine can be the parameter (or return value) of another subroutine or function, we speak about *higher order* functions (or *first class* functions)

Why Higher Order Functions?

- Let’s write a subroutine to browse a directory tree and print the names of the files in the tree

```
sub traverse-dir ($path) {
    my @dir-entries = dir $path;
    for @dir-entries -> $entry {
        say $entry if ~$entry.f;
        traverse-dir $entry if $entry.d;
    }
}
traverse-dir '/home/Laurent';
```

- If we need to find the total size, we have to rewrite the subroutine. Same thing if we want to delete .tmp files or archive older files
  - That’s bad. We want to reuse code!

Why Higher Order Functions? (2)

-
- We can pass a subroutine to `traverse-dir2`:

```
sub traverse-dir2(&code-ref, $path) {
    my @dir-entries = dir $path;
    for @dir-entries -> $entry {
        &code-ref($entry) if $entry.f;
        traverse-dir2(&code-ref, $entry) if
$entry.d;
    }
}
sub print-file (IO::Path $file) {  say ~$file;  }
traverse-dir2 &print-file, '/home/Laurent';
```

Why Higher Order Functions? (2)

- `print-file` is a call-back. Now, we can use `traverse-dir2` for computing other things:

```
my $size = 0;
traverse-dir2 { $size += *.s }, '/home/Laurent';
say "Total size is $size";   # -> Total size is 4578552195
```

- Callback functions can be named:

```
sub do-twice(&code) { code() for 1..2;}
sub greet {say "Hello world"}
do-twice &greet;                      # prints "Hello world" twice
```

- Or they can be anonymous subroutines:

```
my $greet2 = sub {say "Hello world"};
do-twice $greet2;                    # prints "Hello world" twice
```

Anonymous Subroutines

- We can even pass the subroutine directly:

```
do-twice sub { say "Hello world"}; # prints "Hello world" twice
```

Anonymous Subroutines

- Here, since we have no arguments, it can even be a bare code block

```
do-twice { say "Hello world"};     # prints "Hello world" twice
```

Anonymous Subroutines

- And even if we have arguments, it can still work, as in this example with the `reduce` built-in taking to (placeholder) arguments:

```
my $sum = reduce { $^a + $^b }, 1..20; # -> 210
```

Lambdas

- The `reduce` example is actually a lambda
  - In maths and CS, a lambda is a nameless function
- This code to capitalize cities is also a lambda:

```
.say for map {.tc}, <amsterdam london paris rome madrid>;
```

Lambdas

- This code to find even numbers is also a lambda:

```
my @evens = grep { $_ %% 2 }, 1..17;
                     # -> [2 4 6 8 10 12 14 16]
```

- And so is the “pointy block” syntax used twice here for displaying the multiplication tables

```
for 1..9 -> $mult {
    say "Multiplication table for $mult";
    for 1..9 -> $val {
        say "$mult * $val = { $mult * $val }";
    }
}
```

Closures

- A closure is a function that can access to the lexical variables available when it was defined:

```
sub create-counter(Int $count) {
    my $counter = $count;
    sub increment-count {
        return $counter++
    }
    return &increment-count;
}
my &count-from-five = create-counter(5);
say &count-from-five() for 1..6;
                     # prints numbers 5 to 10
```

Closures (2)

- The closure example is somewhat contrived, because I wanted to show a *named* closure
- Most closures are *anonymous*, but they don't need to be, as the previous example shows

Closures as Function Factories

- Closures make it possible to dynamically create functions at run time

```
sub create-counter(Int $count) {
    my $counter = $count;
    return sub { $counter++ }
}
my &count-from-five = create-counter(5);
my $count-from-ten = create-counter(10);
say &count-from-five() for 1..6;
                        # prints numbers 5 to 10
say $count-from-ten() for 1..3;
                        # prints numbers 10, 11, 12
```

Closures as Function Factories

- We've created a function factory:
  - We can create as many counter functions as we need by just calling the create-counter subroutine
  - We can also store our functions in a dispatch table

A Function Factory for the Alphabet

- We want to split a file containing a list of words into 26 files, one for each letter of the alphabet

```
sub create-sub (Str $letter) {
    my $fh = open :w, "Letter_$letter.txt";
    return sub (Str $line) { $fh.say($line) }
}
my %dispatch;
for 'a'..'z' -> $letter {
    %dispatch{$letter} = create-sub($letter)
}
```

Function Factory for the Alphabet (2)

- We've dynamically created a dispatch table with 26 anonymous functions.
- Using it is very easy:

```
for "words.txt".IO.lines -> $word {
    %dispatch{$0}($word) if $word ~~ / ^ (<[a..z]>) /;
}
```

Function factory for the Alphabet (3)

- We can also dynamically create the dispatch table entries only if and when needed:

```
sub create-sub (Str $letter) {
    my $fh = open :w, "Letter_$letter.txt";
    return sub (Str $line) { $fh.say($line) }
}
my %dispatch;
for map {.lc}, < alpha bravo charlie Zulu > -> $word { \
    my $letter = ~$0 if $word ~~ / ^ (<[a..z]>) /;
    %dispatch{$letter} = create-sub $letter
        unless %dispatch{$letter}:exists;
    %dispatch{$letter}($word);
}
```

Iterators

- Suppose we want a function like `map` that process one item at a time, on demand from a consumer process
  - We can start by building an iterator with a closure

```
sub create-iter(@array) {
    my $index = 0;
    return sub { @array[$index++];}
}
```

  - And then use it as follows :

```
sub iter-map (&code-ref, $iter) {
    return &code-ref($iter);
}
my $iterator = create-iter(1..200);
say iter-map { $_ * 2 }, $iterator() for 1..4; # -> 2, 4, 6, 8
```

Iterators and Infinite Lists

- We can use our iterator on infinite lists

```
my $iterator = create-iter(1..*);
say iter-map { $_ * 2 }, $iterator() for 1..4;
                              # -> 2, 4, 6, 8
say iter-map { $_ * 2 }, $iterator() for 1..4;
                              # -> 10, 12, 14, 16
```

Iterators and Infinite Lists (2)

- The sequence operator can produce more diversified lists (finite or infinite)

```
my $evens = (0, 2 ... Inf);     # (...)
say $evens[18..21];             # -> (36 38 40 42)
my $geometric-progression = 1, 2, 4 ... 32;
                                # (1 2 4 8 16 32)
# Using a generator
say (1, { $_ + 2 } ... 11);     # -> (1 3 5 7 9 11)

# Lazy infinite list of Fibonacci numbers
my @fibo = 0, 1, -> $a, $b { $a + $b } ... *;
say @fibo[0..10];         # -> (0 1 1 2 3 5 8 13 21 34 55)
# Or:
my @fibo = 0, 1, * + * ... *;
say @fibo[^10];           # -> (0 1 1 2 3 5 8 13 21 34)
```

Currying

- Currying is a basic technique of FP languages
  - This takes a function with two pamameters and returns a function taking only one parameter

```
sub make-add (Numeric $added-val) {
    return sub ($param) {$param + $added-val;}
}
my &add_2 = make-add 2;
say add_2(3); # -> 5
```

Currying (2)

- Currying an existing subroutine with the `assuming` method

```
sub add (Numeric $x, Numeric $y) {return $x + $y}
my &add_2 = &add.assuming(2);
add_2(5); # -> 7
my &add_4 = &add.assuming(4);
Add_4(5); # -> 9
```

Currying and the Whatever Operator

- A flexible way to curry a sub or expression:

```
my &third = * / 3;
say third(126); # -> 42
```

- The whatever star (*) is a placeholder for an argument, the expression returns a closure.
- Somewhat similar to $_, but it does not need to exist when the declaration is made

```
say map 'foo' x * , (1, 3, 2);  # (foo foofoofoo foofoo)
```

- You can use multiple whatever terms in the same expression

```
my &add = * + *;
say add(4, 5); # -> 9
```

Going Further in FP

- So far, we have seen how to use techniques coming from functional programming
- These are very useful and can help you make your code shorter and more expressive
- However, I would recommend you to try to use a real functional programming style
- It may not be adapted for every problem, but some will really benefit

The Merge Sort Algorithm (1)

- Non functional (procedural) version:

```
sub merge-sort (@out, @in, $start = 0, $end = @in.elems) {
    return if $end - $start < 2;
    my $middle = ($end + $start) div 2;
    merge-sort(@in, @out, $start, $middle);
    merge-sort(@in, @out, $middle, $end);
    merge-lists(@out, @in, $start, $middle, $end);
}
sub merge-lists (@in, @out, $start, $middle, $end) {
    my $i = $start;
    my $j = $middle;
    for $start..$end - 1 -> $k {
        if $i < $middle and ($j >= $end or @in[$i] <= @in[$j]) {
            @out[$k] = @in[$i];
            $i++;
        } else {
            @out[$k] = @in[$j];
            $j++;
        }
    }
}
```

The Merge Sort Algorithm (FP)

- Functional implementation of merge sort:

```
sub merge-sort (@in) {
    return @in if @in < 2;
    my $middle = @in.elems div 2;
    my @first  = merge-sort(@in[0 .. $middle - 1]);
    my @second = merge-sort(@in[$middle .. @in.end]);
    return merge-lists(@first, @second);
}
sub merge-lists (@one, @two) {
    my @result;
    loop {
        return @result.append(@two) unless @one;
        return @result.append(@one) unless @two;
        push @result, @one[0] < @two[0] ?? shift @one !! shift @two;
    }
}
```

- Not only shorter code, it captures the essence of the merge sort algorithm and is much simpler to understand and implement

The Quick Sort Algorithm

- This is a non-FP implementation of QS :

```
sub quicksort(@input) {
    sub swap ($x, $y) {
        (@input[$x], @input[$y]) = @input[$y], @input[$x];
    }
    sub qsort ($left, $right) {
        my $pivot = @input[($left + $right) div 2];
        my $i = $left;
        my $j = $right;
        while $i < $j {
            $i++ while @input[$i] < $pivot;
            $j-- while @input[$j] > $pivot;
            if $i <= $j {
                swap $i, $j;
                $i++;
                $j--;
            }
        }
        qsort($left, $j) if $left < $j;
        qsort($i, $right) if $j < $right;
    }
    qsort(0, @input.end)
}
my @array = pick 20, 1..100;
quicksort @array;
say @array;  # -> [4 7 10 20 25 27 28 30 41 47 51 57 60 64 68 76 90 93 94 99]
```

Quick Sort Algorithm (Scala-like)

- Functional implementation of Quick Sort

```
sub quicksort(@input) {
    return @input if @input.elems <= 1;
    my $pivot = @input[@input.elems div 2];
    return flat quicksort(grep {$_ < $pivot}, @input),
        (grep {$_ == $pivot}, @input),
        quicksort(grep {$_ > $pivot}, @input);
}
```

Quick Sort Algorithm (Scala-like)

- The code is three times shorter
- Just as for merge sort, the FP code captures very neatly the essence of the algorithm
- It copies data instead of in-place sorting, so it is less efficient for some cases (but it is probably easier to run parallel processes)

Quick Sort Algorithm (Haskell-like)

- An example of quick sort in Haskell

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smaller = quicksort [a | a <- xs, a <= x]
        larger  = quicksort [a | a <- xs, a > x]
    in  smaller ++ [x] ++ larger
```

- Quick sort in Haskell-like Perl 6

```
multi quicksort ( [] ) { () }
multi quicksort ( [$x, *@xs] ) {
    flat quicksort(grep * <=  $x, @xs),
         $x,
         quicksort(grep * > $x, @xs)
}
```

Conclusion

- Any questions?
- Additional readings:
  - *Higher Order Perl*: http://hop.perl.plover.com/book/
  - *Think Perl 6* (chapter on functional programming): http://greenteapress.com/wp/think-perl-6/
  - These slides: https://github.com/LaurentRosenfeld/Perl-6-Miscellaneous/Talks/English
    - They can be used under the terms of the Creative Commons Attribution ShareAlike License (CC-BY-SA)

- Thank you for listening.

Bonus Slides

Small Benchmark of Quick Sort

- Comparing non-FP, FP and parallel FP

```
sub qs-FP-para ($level is copy, @input) {
   return @input if @input.elems <= 1;
   my $pivot = @input[@input.elems div 2];
   if (++$level < 4) {
      my $p1 = start { qs-FP-para $level, grep {$_ < $pivot}, @input};
      my $p2 = start { qs-FP-para $level, grep {$_ > $pivot}, @input};
      my (@one, @two) = await $p1, $p2;
      return flat @one, (grep {$_ == $pivot}, @input), @two;
   } else {
      return flat qs-FP-para($level, grep {$_ < $pivot}, @input),
          (grep {$_ == $pivot}, @input),
          qs-FP-para($level, grep {$_ > $pivot}, @input);
   }
}
```

- Results:

```
Non functional: 0.1322154
Functional: 0.3297219
Parallel FP: 0.16110757
```

Functional Quick Sort in Scala

- The same FP QS algorithm in Scala (example by Martin Odersky, EPFL, the creator of Scala, in the *Scala by Example* book)

```
def sort(xs: Array[Int]): Array[Int] = {
  if (xs.length <= 1) xs
  else {
    val pivot = xs(xs.length / 2)
    Array.concat(
      sort(xs filter (pivot >)),
           xs filter (pivot ==),
      sort(xs filter (pivot <)))
  }
}
```