Higher Order Perl

Finlay Thompson

14 March 2006

Talk Outline

- What is this all about ?
- Some FP techniques and ideas
- Loops, recursion, iterators, chasing tails, loops
- Currying and anonymous subroutines
- Conclusions

What is this all about?

- Functional programming techniques
- Mark J Dominus' book "Higher-Order Perl" describes using these techniques in perl
- Perl and FP are an awkward match, not always the most natural
- Run directly into the Perl Best Practices advice, "Keep it simple"

Functional Programming at Uni

- Pure functions and side effects
- Loops and recursion
- First class functions and currying
- Contrived examples (Fibonacci numbers ?!)

Pure Functions and Side Effects

Pure functions always return same result for same input.

For example putting some input into a standard form is usually a pure function:

```
sub normalise_name {
    my $name = shift;
    my $output = '';
    foreach my $word (split(/\s+/, $name)) {
        $word = lc($word);
        $word = ucfirst($word);
        $output .= $word;
    ]
    return $name;
}
```

Pure Functions and Side Effects

Functions with "side effects" are not pure. Side effects involves changing the state of the program or world outside of the funtion. For example all IO operations are not pure functions.

Another example involves state transformations:

```
my $names = {};
sub normalise_and_count_name {
    my $name = shift;
    $name = join(" ", map {ucfirst($_)} split(/\s+/, lc($name)));
    my $num = $names->{$name}++;
    return ($name, $num);
}
```

Pure Functions and Side Effects

Why do we care ?

- pure functions can be safely "memoized". That is the results can be cached in memory to avoid needing to recalculate the function.

- pure functions are easier to debug and optimise.

However a lot of code cannot be converted into pure functions.

The IO Monad in haskell is all about seperating pure and impure functions.

Loops and Recursion

Simple experiment to compare a loop and its corresponding recursive implementation. Here is a simple loop:

```
sub sum_numbers {
    my $numbers = shift;
    my $count = 0;
    while (my $number = pop @$numbers) {
        $count += $number;
    }
    return $count;
}
```

Loops and Recursion

Now we do it with a recursive function:

```
sub sum_numbers {
    my $numbers = shift;
    return 0 unless @$numbers;
    my $number = pop @$numbers;

    return $number + sum_numbers($numbers);
}
```

Loops and Recursion

What about in haskell ?

```
sum_numbers :: [Int] -> Int
sum_numbers [] = 0
sum_numbers (n:ns) = n + sum_numbers ns
```

Loops and Recursion

How do they compare ?

```
      Loop:  6 wallclock secs
        ( 5.85 usr +  0.00 sys =  5.85 CPU) @ 1709.40/s (n=10000)
 Recursive: 14 wallclock secs
        (14.78 usr +  0.01 sys = 14.79 CPU) @ 676.13/s (n=10000)
```

Loops and Recursion

Compared with the haskell version:

```
tangaroa:$ time cat random_numbers.txt | perl count.pl
Count = 499245

real    0m0.019s user    0m0.020s sys     0m0.000s
tangaroa:$ time cat random_numbers.txt | perl rec_count.pl
Count = 499245

real    0m0.042s user    0m0.034s sys     0m0.010s
tangaroa:$ time cat random_numbers.txt | ./count
Count = 499245

real    0m0.258s user    0m0.253s sys     0m0.006s
```

Iterators

- Like filehandles
- Lazy evaluation improves performance
- Convert recursion to use an iterator by managing state

Iterators

File handles are familiar iterators

```
open(FILEHANDLE, 'filename');
while (<FILEHANDLE>) {
    # do something
}
```

Iterators

Using a closure and a code ref to make an iterator

```
sub range {
    my ($m, $n) = @_;
    return sub {
        $m <= $n ? $m++ : undef;
    };
}
my $it = range(3, 5);
my $nextval = $it->();
```

Iterators

- Recursion involves pushing and poping state values (inefficient)
- Iterators maintain can maintain there own state (more efficient)
- Only return results when needed (flow control and lazy)

Chasing the tail

What if instead of poping the call stack we just passed control to the new function call and left the stack alone ?

This involves converting recursive calls into iterator loops, and we are back to loops.

First Class Functions

Most functional languages treat functions the same as data, they are 'First class citizens'.

Perl does not. However, we can get a code ref, which allows us to pass functions as arguments.

This is very useful a way of managing code reuse.

The function 'map' does this:

```
my @bigger_numbers = map {$_+10} @number;
```

The map function accepts a code ref as an argument.

First Class Functions

Dispatching functions makes for easy to maintain code:

```
my $dispatch = {
    home      => \&display_home,
    list      => \@display_list,
    whatever => \&display_whatever,
};
return $dispatch->{$code}->(@args);
```

Can easily add new functions into list.

First Class Functions

Currying is a nickname for the idea of partially evaluating functions. Some more Haskell:

```
log_lines :: String -> String -> IO ()
log_lines "Prefix" :: String -> IO ()
```

This is awkward, but possible, to emulate in perl.

First Class Functions

```
sub curry {
    my $f = shift;
    return sub {
        my $first_arg = shift;
        my $r = sub { $f->($first_arg, @_) };
        return @_ ? $r->(@_) : $r;
    };
};
BEGIN { *c_log = curry(sub {
    my ($p, $m) = @_;
    print LOG $p . $m
}; }
*p_log = c_log { 'Prefix' };
```

Conclusions

- Functional programming rocks
- Perl is not a functional programming language
- Perl is still cool