Functional Programming

jens.jensen@stfc.ac.uk
0000-0003-4714-184X
CC-BY 4.0

January 30, 2022

Outline

- This talk
  - Introibo
  - Pure Functional Programming Principles
- Future talks
  - Advanced(ish) Functional Programming
  - Impure Functional? Side Effects
  - Category Theory
  - Categories and Functions
  - Categories and Computation

Let us get together and study. As the Poet once said,

> *The full-spread power of Emacs is calming and excellent to the soul*

The same is true for category theory, and Lisp. And mostly true for functional programming.

This talk

This talk is the first of $n > 2$ on functional programming with (E)Lisp; it was given to RSEmacs 01 Feb 2022.
`https://m-x-research.github.io/`

Rules and Terminology

- Interrupt to ask questions – we will be covering a lot of different ground. Best to ask before we are on a different topic. The first part of this talk is meant to be more interactive, the second ("monadland") is less so.
- Some terminology:
  - "s.t." == "such that"
  - iff == "if and only if"
  - I might use the word "map" for $\mapsto$, as in `sqrt` maps 9 to 3
  - EL == Emacs Lisp; CL = Common Lisp (`M-x slime`)
  - Inferior languages = languages which are not Lisp
- MonadLand Arrows (to be explained later):
  - $\to$ for arrow, function, functor
  - $\overset{\bullet}{\to}$ for natural transformation
  - $\rightharpoonup$ for adjunction

Functional programming

Introduction to Functional Programming (Theory and Practice) with (Emacs) Lisp (one tends to use parentheses a lot (like this one) when writing Lisp talks)

The author will attempt a synthesis of four different views:

- CS (IANACS): types, lambda calculus
- Maths: category theory
- "Programming theory": pure and impure functional
- "Programming practice": Emacs Lisp (instead of, say, Haskell)

Written in the author's Copious Spare Time™, any opinions (and mistakes) are his own (unless stated otherwise). Lisp examples are ELisp unless indicated otherwise. It assumes familiarity with ELisp.

Principles of Functional Programming

Pure functional programs are "defined" by these basic features:

- Variables (`let`, `let*` etc) are immutable
  - No `setq`/`setf` or modifier `...f` (eg `incf`)
  - Use *recursion* instead of `loop` or `do`, `do*`, etc
- Functions are called *without side effects*
  - No `n...` functions (`nreverse` etc) ...
  - ... except when the side effect has no effect (see later)
- Can pass functions as argument
- Divide and conquer strategy for solving problems

Later we shall zoom in on advanced features of functional programming in general and in Lisp(s) in particular.

Why Functional?

- Elegance (much of the time)
- Correctness
  - Programmer-readable correctness (loop invariants become parameter constraints)
  - Compiler-inferred correctness (stronger type checks)
- Cleaner code (examples later)
- Like learning Lisp, it makes you a Better Programmer©

Why not functional?

We shall see examples of these during the talk (and how to mitigate (some of) them):

- Inefficiencies:
  - copy lists/arrays to update them
  - temporary lists being created
  - more function calls? (stack frames etc)
    - `setq` is usually cheaper than a function call
- Some patterns are difficult to program functionally
  - Arrays are a good example, particularly multidimensional
    - Particularly arbitrary-dimensional (generic) arrays which most languages other than CL struggle with
  - In fact, arrays have theoretical, practical and lispological "issues"

Functions as arguments

```
((lambda (x y) (+ x y)) 2 3)
5
(funcall (lambda (x y) (+ x y)) 2 3)
5
(let ((plus #'+)) (funcall plus 2 3))
5
(defun fred (x)
  (funcall (if (evenp x)
               #'sqrt
             (lambda (y) (- y)))
           x))
fred
(mapcar #'fred '(1 2 3 4 5 6))
(-1 1.4142135623730951 -3 2.0 -5 2.449489742783178)
```

Example

```
(defun fact (k)
  (if (zerop k) 1 (* k (fact (1- k)))))
fact
(fact 12)
479001600
```

- Uses $n! = n(n-1)!$ for $n > 0$, and $0! = 1$
- Should check for $n < 0$ or n not a number
  - In "true" functional languages like Haskell and F#, the type is *inferred* (from $*$ and -)
  - (Some) Lisps do something similar but usually less strict

Example - A Functional Koan

This is CL from the author's Advent of Code(AoC) 2021, 01 Dec.

```
(defun solve1 (data)
  "Count number of numbers that increase in a list"
  (count-if #'plusp (mapcar #'- (cdr data) data)))
```

- `mapcar` maps as long as the shortest of its input lists
- In efficiency terms, an intermediate list of length $n-1$ (for input of length n) is created
  - This list will need to be *gc*ed, eventually
  - As a (functional) Lisp programmer, you have to consider efficiency - both memory and time
- EL's `mapcar` only accepts unary functions (and a single list)

Example - A Less Elegant Koan

Another short example from the same code base:

```
(defun sum-3-window (list)
  "For a list of inputs, return the sum of
   sliding length 3 windows"
  (mapcar (lambda (a b c) (+ a b c))
          list (cdr list) (cddr list)))
```

- No unnecessary lists created
- Exercise: what if the length were passed in?
- Again won't work with EL's `mapcar`

Example – A Solver Wrapper

Also from the author's AoC21, allowing different inputs for testing:

```
(defun solve (data solve-func)
  (cond
    ((streamp data) (funcall solve-func data))
    ((pathnamep data)
      (with-open-file (foo data
                           :direction :input :if-does-not-exist :error)
        (solve foo solve-func)))
    ((stringp data)
      (solve (make-string-input-stream data) solve-func))
    (t (error "Cannot process ~A" (type-of data)))))
```

- This is CL rather than EL but you can see how it works
- In CL, pathnames are distinct from strings (though the latter can implicitly convert to the former)
- etypecase would have been a better choice for this particular problem

Example: an *n*-ary plus

Let's assume we have a 2-ary plus (#'+) and we want to build an *n*-ary plus.

Mathematically(ish), we define:

$$\begin{aligned} \mathrm{plus}() &= 0 && \text{(empty list);} \\ \mathrm{plus}(a) &= a && \text{(one element);} \\ \mathrm{plus}(a, b \cdots) &= a + \mathrm{plus}(b \cdots) && \text{(recursively)} \end{aligned}$$

The effect is that of

$$\mathrm{plus}(a, b, c, d) = a + (b + (c + d))$$

n-ary +

Turning this into code, we get

```
(defun my+ (my-list)
  "Sum an arbitrary number of elements"
  (cond
    ((= (length my-list) 0) 0)
    ((= (length my-list) 1) (first my-list))
    (t (+ (first my-list) (my+ (rest my-list))))))
```

This works and is functional, but it is not a good implementation for several reasons. Your meditation exercise (before turning to the next slide) is to ponder why.

n-ary +

- (Usability) First of all, we would like to call `(my+ 1 2 3 4)` instead of `(my+ '(1 2 3 4))`
- (Efficiency) Second, the `length` function is called which will calculate the length every time; we just need to know if the list is empty (or has a single element)
- (Correctness) Calling the function on a long list will exhaust the stack (which could lead to heisenbugs as we don't know how much space is left on the stack)
- (Correctness) floatologically speaking, adding numbers naïvely can lead to precision loss etc. (we shall ignore this in this talk)

n-ary +

```
(defun my+ (&rest my-list)
  "Sum an arbitrary number of elements"
  (cond
    ((endp my-list) 0)
    ((endp (cdr my-list)) (first my-list))
    (t (+ (first my-list) (apply #'my+ (rest my-list))))))
```

Notice the *four* differences with the earlier version. (`car` is synonymous with `first` and `cdr` with `rest`; the choice is mostly stylistic. The author prefers `first`/`rest` for functional divide-and-conquer and `c*r` for everything else (see also Naming, later))

Meditation: there is a more subtle performance issue (hint: how many times is `length`, resp. `endp` called for a list of length $n \geq 2$?) And we haven't dealt (yet) with the *correctness* issues.

n-ary +

Of course endp is much, much cheaper than `length` in general

```
(defun my+ (&rest my-list)
  "Sum an arbitrary number of elements"
  (if (endp my-list) 0 (my+1 my-list)))

(defun my+1 (my-list)
  "Helper function for my+: sum elements of a non-empty list
  (if (endp (cdr my-list)) (first my-list)
    (+ (first my-list) (my+1 (rest my-list)))))
```

Performance testing would be needed to decide whether it is worth it

Meditation exercise: Could we have passed in the function as well (à la `mapcar`)? Also, we still haven't dealt with the *correctness* issues.

n-ary +

Of course, Emacs's + is already *n*-ary. Using built-ins:

```
(+ 1 2 3 4 5)
15
(apply #'+ '(1 2 3 4 5))
15
(funcall #'+ 1 2 3 4 5)
15
(reduce #'+ '(1 2 3 4 5))
15
```

or with the function bound in a variable:

```
(let ((plus #'+)) (funcall plus 3 4))
7
(let ((plus #'+)) (apply plus '(3 4)))
7
```

Meditation: why do `reduce` and `apply` seem to do exactly the same thing?

Elegance vs Efficiency

In a list `seqs` of sequences (again from AoC), all have the same length:

```
(apply #'= (mapcar #'length seqs))
```

Meditation: this is short, functional and elegant.

Usually, "short" and "functional" imply "elegant" (and *inefficient*...?)

It is indeed not the most *efficient* implementation in general. Why not? Does it matter? If it matters, how can it fixed (functionally)?

Example – legitimate use of side effects

Sometimes, using functions with side effects (i.e. they alter their argument) is safe, correct and functional:

```
(defun integer-digits (n radix)
  "Split a non-negative integer into its digits"
  (labels ((i-dig (n)
             (if (< n radix) (list n)
               (let ((q (floor n radix))
                     (r (mod n radix)))
                 (cons r (i-dig q))))))
    (nreverse (i-dig n))))
(integer-digits 2345 10)
(2 3 4 5)
```

(comments on next slide)

- `labels` define local functions that can call themselves (or each other)
- The temporary list is an r-value (in C++-speak), so using `nreverse` on it avoids creating more cons cells
  - Indeed, it is illegal (or at least a very bad idea) to call `nreverse` *et al* on l-values
  - Not all `n` functions are consistently named: `sort` and `stable-sort` also destroy their arguments
- In CL, `floor` returns both the quotient and the remainder, so is more efficient

Example – Reduce Koan

The inverse of `integer-digits`:

```
(defun digits-to-integer (digits radix)
  "Turn list of digits into integer with given radix"
  (reduce (lambda (a b) (+ (* a radix) b)) digits))
(digits-to-integer '(2 3 4 5) 10)
2345
```

Some items for when you meditate on this code:

- Can we express the *type constraint* that `digits` is a list of integers k s.t. $0 \leq k <$ `radix`?
  - Actually it just has to be a *sequence*
  - `(digits-to-integer "ABCD" 100)` = `65666768`
- What happens if `digits` is of length 1? Of length 0?