Functional Programming 7

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

April 30, 2023

Outline of Talks

- Previous talks (talks 1-3):
  - Introibo
  - Pure Functional Programming Principles
  - Mapping
  - Labels and naming
  - Lists
- This talk (Talk 7):
  - Advanced(ish) Topics (continued)
- Impure Functional? Side Effects
- Category Theory
- Categories and Functions
- Categories and Computation

Still written in the author's spare time!

Very much a personal perspective, and not following any particular textbook. Using *meditations* and *exercises* – solutions to all exercises given during the talks.

Common/Advanced(ish) Features of Functional Languages

1. Lambda (anonymous (unnamed) functions) and *currying*
2. List comprehension
3. Functions – mutually recursive, higher order
4. Symbols
5. Tail recursion
6. Scope and extent
7. Types and type inference
8. Branch-on-pattern-matching and guards
9. **Memoisation**
10. **Lazy evaluation types**
11. Pipes (not the lazy kind) style composition
    - $h(g(f(x))) \equiv$ `(h (g (f x)))` $\equiv x|f|g|h$
12. Monads: theoretical framework for types and computation
13. Applied monads: Maybe, Arrays
14. Bonus section for survivors of MonadLand: Lisp Hacking

Today's talk

is about building on the hard work in Talk 6:

- Lexical scope – specifically closures
  - Indefinite extent
- "Dynamic scope"
  - Indefinite scope
  - Dynamic extent

Today we will be relying on closures ("lexical bindings") to maintain state of pure functions – and impure functions.
A future talk (probably 8) will look at maintaining state in "dynamically scoped" variables.

Memoisation

The classic example of recursion is Fibonacci numbers (A000045):

```
(defun fib (k)
  (let ((call-count 0))
    (labels ((fib-1 (k1)
               (incf call-count)
               (if (<= k1 1) 1
                 (+ (fib-1 (1- k1)) (fib-1 (- k1 2))))))
      (list (fib-1 k) call-count))))
(fib 0)
(1 1)
(fib 10)
(89 177)
(fib 19)
(6765 13529)
```

Exercise: write the inner function functionally-ly (without `incf`). What is the cost? If you know CL, how can CL do better?

Memoisation

Let's make a quick timing macro (this is EL, CL already has one):

```
(defmacro time (&rest body)
  "Time the execution of an expression, returning list
   of value and time"
  `(let* ((#1=#:start (current-time))
          (#2=#:result (progn ,@body))
          (#3=#:time (time-subtract (current-time) #1#)))
     (list #2# (format-time-string "%M:%S.%6N" #3#))))
(time (fib 30))
((1346269 2692537) "00:07.348357")
```

- If you have forgotten about *uninterned symbols*, please refer to Talk 4.
- The formatter above is designed for absolute time so won't work for one hour or more

Memoisation

Pure functions (without side effects) return the same value every time they are called with the same arguments.

Memoisation suggests that for functions that are expensive to calculate, we cache the results. Typically caches are built with hash tables or vectors.

```
(let ((cache (make-hash-table :test #'eql)))
  (defun fib (k)
    (let ((call-count 0))
      (labels ((fib-1 (k1)
                 (incf call-count)
                 (if (<= k1 1) 1
                   (or (gethash k1 cache)
                       (let ((val (+ (fib-1 (1- k1))
                                     (fib-1 (- k1 2)))))
                         (setf (gethash k1 cache) val))))))
        (list (fib-1 k) call-count)))))
```

Memoisation

The cache keeps building:

```
(fib 1)
(1 1)
(fib 10)
(89 19)
(fib 20)
(10946 21)
(fib 30)
(1346269 21)
(fib 30)
(1346269 1)
```

Exercise: how would we cache with a vector? A vector would work well with the non-negative integer argument to `fib`, but would need to grow as needed.
(Norvig has a generic memoiser – we shall return to that later)

Fibonacci numbers – a functional digression

As a pure or mathematical function, `fib` – $f : \mathbb{N}_0 \to \mathbb{N}$ – is defined as

$$
\begin{aligned}
f(0) &= 1, \\
f(1) &= 1, \\
f(k) &= f(k-1) + f(k-2), \quad k \geq 2
\end{aligned}
$$

The function is defined recursively, and the cache remembers the values (such as $20 \mapsto 10946$) as they are built by calls to the function. Hence the cache represents `fib` as a partial function (because the cache is necessarily finite).

While `fib` remains a pure function, *calling it changes the function*:

- In theoretical terms, the partial function is possibly expanded to be defined on more of the domain;
- In practical terms, the function becomes faster, gradually

Hence evaluation has an effect of mapping (the value of) `fib` to another value: $\texttt{fib}_h \mapsto \texttt{fib}_{h+1}$ (where h is the number of times `fib` has been called). For every value k for which $\texttt{fib}_h$ is defined (has cache), $\texttt{fib}_h(k) = \texttt{fib}_{h+1}(k)$.

Memoisation – a mathematical digression

Digression. $a_{i+1} = a_i + a_{i-1}$ can be expressed

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a_i \\ a_{i-1} \end{pmatrix} = \begin{pmatrix} a_{i+1} \\ a_i \end{pmatrix}$$

and in the next section (lazy evaluation) we shall use that. Thus with $a_0 = a_1 = 1$,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} a_{n+1} \\ a_n \end{pmatrix}$$

Taking $\alpha = \frac{1+\sqrt{5}}{2}$ and $\beta = \frac{1-\sqrt{5}}{2}$, we get

$$\begin{pmatrix} \frac{\alpha}{\sqrt{2+\alpha}} \\ \frac{1}{\sqrt{2+\alpha}} \end{pmatrix} \text{ and } \begin{pmatrix} \frac{\beta}{\sqrt{2+\beta}} \\ \frac{1}{\sqrt{2+\beta}} \end{pmatrix}$$

are orthonormal eigenvectors of the matrix corresponding to eigenvalues α and β, respectively,

Fibonacci numbers – a mathematical digression

... whence for all n,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} \frac{\alpha}{\sqrt{2+\alpha}} & \frac{\beta}{\sqrt{2+\beta}} \\ \frac{1}{\sqrt{2+\alpha}} & \frac{1}{\sqrt{2+\beta}} \end{pmatrix} \begin{pmatrix} \alpha^n & 0 \\ 0 & \beta^n \end{pmatrix} \begin{pmatrix} \frac{\alpha}{\sqrt{2+\alpha}} & \frac{1}{\sqrt{2+\alpha}} \\ \frac{\beta}{\sqrt{2+\beta}} & \frac{1}{\sqrt{2+\beta}} \end{pmatrix}$$

Since we are interested only in one of the values, we get

$$\begin{pmatrix} * \\ a_n \end{pmatrix} = \begin{pmatrix} * & * \\ \frac{1}{\sqrt{2+\alpha}} & \frac{1}{\sqrt{2+\beta}} \end{pmatrix} \begin{pmatrix} \alpha^n & 0 \\ 0 & \beta^n \end{pmatrix} \begin{pmatrix} \frac{\alpha}{\sqrt{2+\alpha}} & \frac{1}{\sqrt{2+\alpha}} \\ \frac{\beta}{\sqrt{2+\beta}} & \frac{1}{\sqrt{2+\beta}} \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

(where $*$ means "don't care") which reduces to

$$a_n = \alpha^n \frac{1+\alpha}{2+\alpha} + \beta^n \frac{1+\beta}{2+\beta}$$

This is a *closed form* for the Fibonacci numbers.

Fibonacci numbers – a mathematical digression

```
(defconst alpha (/ (+ 1 (sqrt 5)) 2))
alpha
(defconst beta (/ (- 1 (sqrt 5)) 2))
beta
(defconst alpha-frac (/ (+ alpha 1) (+ alpha 2)))
alpha-frac
(defconst beta-frac (/ (+ beta 1) (+ beta 2)))
beta-frac
(defun fib (n)
  (+ (* alpha-frac (expt alpha n))
     (* beta-frac (expt beta n))))
fib
(fib 10)
89.00000000000003
(fib 100)
5.73147844013819e+20
(fib 0)
1.0
```

Fibonacci numbers – a mathematical digression

We can now compare this implementation to the previous:

```
(fib 1)
0.9999999999999999
(fib 19)
6765.000000000004
```

We can make two additional improvements to the code:

- Since a rounding error has been introduced, it would make sense to use `round` on the result
- Noticing that $|\beta| < 1$, the contribution of the `beta` part becomes negligible:

```
(defun fib (n)
  (round (* alpha-frac (expt alpha n))))
(mapcar #'fib '(0 1 2 3 4 5 6 7 8 9 10 11))
(1 1 2 3 5 8 13 21 34 55 89 144)
```

Fibonacci numbers – a functional digression

Whether implemented as

```
(defun fib (k)
  (round (* alpha-frac (expt alpha k))))
```

(with the constants defined as before) or

```
(defun fib (k)
  (labels ((fib-1 (k1)
      (if (<= k1 1) 1
         (+ (fib-1 (1- k1)) (fib-1 (- k1 2))))))
    (fib-1 (1+ k))))
```

these functions return the same value for every *valid* value of `k` (`k`$\in \{0, 1, ...\}$), so, being pure, they are *the same function* (exercise: the statement is only "mostly true" – why?) Though in practice they have hugely different runtimes – does theory care?

Lazy evaluation

Simple generators: generating an infinite sequence

```
(defun simple-generator (init step)
  (let ((w init))
    (lambda () (prog1 w (setq w (funcall step w))))))
simple-generator
(setq m (simple-generator 1 (lambda (k) (ash k 1))))
(closure ...)
(funcall m)
1
(funcall m)
2
(funcall m)
4
(funcall m)
8
```

Lazy evaluation

Simple generators: generating an infinite sequence

```
(defun list-generator (lst)
  (let ((y (copy-seq lst)))
    (rplacd (last y) y)
    (lambda () (prog1 (car y) (setq y (cdr y))))))
list-generator
(setq n (list-generator '(1 2 3)))
(closure ...)
(funcall n)
1
(funcall n)
2
(funcall n)
3
(funcall n)
1
(funcall n)
2
```

Lazy evaluation

Back to FizzBuzz:

```
(defun make-fizzbuzz ()
  (let ((fizz (list-generator '(nil nil fizz)))
        (buzz (list-generator '(nil nil nil nil buzz)))
        (num (simple-generator 1 #'1+)))
    (lambda ()
      (tidy-up (funcall fizz) (funcall buzz) (funcall num)))))

(defun tidy-up (f b n)
  (if (and f b) 'fizzbuzz
    (or f b n)))

(let ((u (make-fizzbuzz))
      (v (make-vector 30 0)))
  (map 'vector (lambda (x) (funcall u)) v))
[1 2 fizz 4 buzz fizz 7 8 fizz buzz 11 fizz ...]
```

Pipes and Series – Pipes

For a more sophisticated view of lazy evaluation, we turn to Pipes next. This is necessarily a very simple invitation to the topic; for more details see [Norvig].

The basic idea behind pipes is a pair of a value and a next step which is not evaluated unless (and until) it is needed. Suppose we want to represent the non-negative integers:

```
(cons 0 #'1+)
```

Things to note:

- Norvig's implementations are more sophisticated than the one presented here
- Norvig's code was written for CL but (given that it works approximately like the code presented here) should work for EL with a few mods

Pipes and Series – Pipes

The pipe would be expanded as needed, one step at a time:

```
(defun pipe-step (p)
  (let ((pp (last p)))
    (rplacd pp (cons (funcall (cdr pp) (car pp)) (cdr pp))))
  p)
```

and we can call it

```
(pipe-step (pipe-step (pipe-step (cons 0 #'1+))))
(0 1 2 3 . 1+)
```

Exercise: write our beloved factorial function using reduce on a pipe (as described here)

Exercise: would it be better to have the step in front? What would the factorial look like in this case? Alternatively, use the list-as-FIFO hack to avoid the $O(n)$ complexity of `last`.

Pipes and Series – yield

As a short digression, let's look at a *generator* using *yield*. We will temporarily switch to another possibly recognisable scripting language.

First another helper function:

```
def isprime(l, k):
    """ check whether no number in l divides k """
    for p in l:
        if k % p == 0:
            return False
    return True
```

(This is trial division, a very inefficient implementation but we get to efficiency later)

Pipes and Series – yield

The algorithm – (Incremental) Trial Division:

```
def prime():
    yield 2
    yield 3
    primes = []
    p = 5
    while True:
        if isprime(primes, p):
            yield p
            primes.insert(0, p)
        p += 2
        if isprime(primes, p):
            yield p
            primes.insert(0, p)
        p += 4
```

Pipes and Series – yield

The point is the code is generating primes and remembering them; the list of primes is used to test further candidates.

Generate the first 30,000 primes (and timing it):

```
gen = prime()
start = dt.datetime.now()
for _ in range(30000):
    e = next(gen)
delta = dt.datetime.now() - start
print("{}.{}\n".format(delta.seconds, delta.microseconds))
```

Pipes and Series – yield

Let's do it in CL before we return to EL. CL doesn't have *yield* because it doesn't really need one. Instead, we use a *condition* to *signal* the next value:

```
(define-condition yield ()
  ((value :reader yield-value :initform nil :type t))
  (:documentation "condition yielding a value from a generat
```

Conditions are very powerful and, like in python, can be extended to pass information back to the generator, handle multiple simultaneous generators, etc. (Errors and warnings are subclasses of conditions)

Pipes and Series – yield

```
(defun make-prime-generator ()
  (let ((primes nil)
        (y (make-condition 'yield)))
    (flet ((isprime (k)
             (notany (lambda (divisor)
                       (zerop (mod k divisor))) primes))
           (yield (k)
             (setf (slot-value y 'value) k)
             (signal y)))
      (lambda ()
        (yield 2)
        (yield 3)
        (let ((p 5))
          (loop
            (when (isprime p) (yield p))
            (incf p 2)
            (when (isprime p) (yield p))
            (incf p 4)))))))
```

Pipes and Series – yield

Generating 30,000 primes:

```
(defun timing (&optional (count 30000))
  (time
   (let ((gen (make-prime-generator)))
     (handler-bind
         ((yield #'(lambda (c)
                     (when (zerop (decf count))
                       (return-from timing
                         (yield-value c))))))
       (funcall gen)))))
```

Note the subtly different control flow: there is no *next* in this case; the generator keeps running till the gatherer stops it, as opposed to the gatherer asking directly for the next value.

Pipes and Series – yield

Emacs, too, has a yield:

```
(require 'generator)
(iter-defun gen-primes ()
  (iter-yield 2)
  (iter-yield 3)
  (let ((primes nil)
        (p 5))
    (flet ((primep (k)
                   (notany (lambda (x) (zerop (mod k x))) pr
      (loop
       (when (primep p) (iter-yield p)
             (push p primes))
       (setq p (+ p 2))
       (when (primep p) (iter-yield p)
             (push p primes))
       (setq p (+ p 4))))))
gen-primes
```

Pipes and Series – yield

```
(let ((y (gen-primes)))
  (loop repeat 20 collect (iter-next y)))
(2 3 5 7 11 13 17 19 23 29 31 37 ...)
```

Using the timing macro from earlier and the following wrapper

```
(defun mkprimes (k)
  (let ((y (gen-primes)))
    (decf k)
    (loop repeat k do (iter-next y))
    (iter-next y)))
```

which returns the last (the kth) prime, we can compare the performance.

Pipes and Series – yield

Results of the timings on the author's famously slow laptop:

iterations	Highest prime	python yield	CL yield	EL yield
10,000	104,729	8.6s	4.2s	132s
30,000	350,377	86.3s	42.8s	1356s
50,000	611,953	251.5s	117.3s	-
100,000	1,299,709	1075s	490.7s	-

CL (SBCL) is faster than python which is a bit apples and oranges: python is a scripting language; CL is not. The EL code is not compiled; the EL compiler doesn't like `gen-primes`

Remember our algorithm can be improved a lot:

- Only test division up to $\sqrt{p}$
- We push larger primes to the front but smaller primes more commonly divide candidates
- If we know the upper bound (which we don't, in general), a vector of bools (Sieve of Eratosthenes) may be the quicker option

Exercise: make it (either version) go faster

Pipes and Series – yield

Digressing again, this code is represented as a Haskell version of Trial Division [O'Neill]:

```
primes = sieve [2..]
sieve (p : xs) = p : sieve [x | x <- xs, x 'mod' p > 0]
```

Oh no! It is much shorter than ours! A literal translation into Lisp (using `loop` as list comprehension, see Talk 4) becomes

```
(defun sieve (pxs)
   (cons (car pxs)
         (sieve (loop for x in (cdr pxs)
                      when (> (mod x (car pxs)) 0)
                        collect x))))
```

except it has an infinite loop and will run out of stack (it's not TRO), and it needs an infinite list of integers (2..) as input.

Pipes and Series – yield

Notice the distinction between yielding and normal closures as the alternative:

```
(iter-defun gen-primes ()
  ...
  (let ((primes nil))
    (iter-yield something...)))
```

vs the closure (this is pseudocode)

```
(defun gen-primes ()
  (let ((primes nil))
    (lambda () (do-something-with primes)
               (updatef primes))))
```

In a sense yield feels more "functional" as it does not have to update its state explicitly. Though we shall cover functions with state (impure functions?) in the next talk.

Pipes and Series – Series

Series are different: they do not have memory of the past that pipes have. They also won't work in EL without a lot of modification, so we cover them only briefly

```
(defun fact (k)
  (declare (type unsigned-byte k))
  (series:collect-product
    (series:scan-range :from 1 :upto k)))
```

`collect-product` is short-hand for reduce using `*`; it even works for no elements ($0! = 1$) since 1 is the identity element for `*`

In general one needs to be *very* careful when handling (infinite) series: they are lazy, but *any* attempt to print them evaluates them fully. This includes the REPL and the debugger!

Pipes and Series – Series

A more sophisticated series generates Fibonacci numbers:

```
(values
  (series:scan-fn '(values integer integer) ; type
    (lambda () (values 1 0))                ; init
    (lambda (x y) (values (+ x y) x))))     ; step
```

This works exactly like the $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ we saw earlier, stepping two sequences (corresponding to a_{n+1} and a_n) in parallel. Note that unlike pipes and (our example) generators, series have no (long term) memory. Point-to-ponder: how could a series generate primes?

Pipes and Series – Series

Infinite series must be handled like dynamite:

```
(series:cotruncate
  (series:scan-fn '(values integer integer)
    (lambda () (values 1 0))
    (lambda (x y) (values (+ x y) x)))
  (series:scan (make-array 20)))
#Z(1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
#Z(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
```

or they will explode... here, the second sequence is used to "take 20" and is ignored when the value is used.

Pipes and Series – Series

And finally, FizzBuzz as a Series:

```
(defun make-fizzbuzz ()
  "Return fizzbuzz as a series"
  (let ((fizz (series:series nil nil 'fizz))
        (buzz (series:series nil nil nil nil 'buzz))
        (num (series:scan-range :from 1 :type 'integer)))
    (series:mapping ((f fizz) (b buzz) (n num))
                    (if (and f b) 'fizzbuzz (or f b n)))))
CL-USER> (series:cotruncate (make-fizzbuzz)
            (series:scan-range :upto 30))
#Z(1 2 FIZZ 4 BUZZ FIZZ 7 8 FIZZ BUZZ 11 FIZZ 13 14
FIZZBUZZ 16 17 FIZZ 19 BUZZ FIZZ 22 23 FIZZ BUZZ 26
FIZZ 28 29 FIZZBUZZ 31)
#Z(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29 30)
```

Summary

- Pure functions always return the same value given the same argument
  - Implementations can benefit from caching, particularly if defined recursively
  - Closures useful to implement the cache
  - The cache *becomes* the function (in some sense)
- Generators generate the next step in a sequence every time they are called
  - Lazy – values generated as needed even in infinite series
  - Coroutines (yield) are alternatives to holding state in closures
  - Compare calling function ('next') directly with in-Lisp evaluation
  - Functionally, they become a sequence $x_0, f(x_0), f(f(x_0)), \ldots$ or $x_i := f(x_{i-1}),\ i > 0$

References

GLS Steele, Guy L: *Common Lisp, the Language* (2nd Ed)

Norvig , P: *Paradigms of Artificial Intelligence Programming*

O'Neill , M E: *The Genuine Sieve of Eratosthenes*, DOI:10.1017/S0956796808007004

A000045 https://oeis.org/A000045