Functional Programming 5

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

May 22, 2022

Outline of Talks

1. Basics of functional programming
   - Recursion, dividing problems into sub-cases, immutable variables
2. Mapping
3. Lists and conses
   - Basic structure: helper functions, naming
4. Lambdas, higher order functions
   - list comprehension
   - symbols

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 (Lisp)
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

TRO 1 (Tail Recursion Optimisation)

```
(disassemble 'fact)
0 dup
1 constant  0
2 eqlsign
3 goto-if-nil 1
6 constant  1
7 return
8:1 dup
9 constant  fact
10 stack-ref 2
11 sub1
12 call   1
13 mult
14 return
```

TRO 2

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

uses tail recursion but is not optimisable; this one is:

```
(defun fact (k result)
  (if (zerop k) result
    (fact (1- k) (* k result))))
```

By the time the function calls itself, the first version has to remember `(* k ...)`; the second version doesn't need to remember anything. Compilers can optimise the recursion call away and make it a loop (though it becomes harder to debug as the extra stack frame is missing).

EL does not do *tail recursion optimisation* (TRO) but it is an important technique in functional programming (and some day EL may do it too)

TRO 3

`fact` has an extra parameter, so we need to call `(fact 12 1)`.
Here is `iota` from before in the same style (note, no `nreverse` – meditate on why it is not needed):

```
(defun iota (k result)
  (if (zerop k) result (iota (1- k) (cons k result))))
```

Notice it generates a list of $1, \ldots, k$; sometimes `(iota k)` is expected to generate $0, \ldots, k-1$
One way to fix the extra parameter is to make `result` optional.
Unfortunately, ELisp does not have default values for optional parameters.

```
(defun fact (k &optional result)
  (if (zerop k) (or result 1)
    (fact (1- k) (* k (or result 1)))))
fact
(fact 12)
479001600
```

TRO 4

This would also do the trick, using a local recursive function defined with `labels` to do the actual work:

```
(defun fact (k)
  (labels ((fact1 (k result)
             (if (zerop k) 1 (fact1 (1- k) (* k result)))))
           (fact1 k 1)))
```

Creating a helper function (scoped to the inside of `fact`) which can call itself recursively (hence using `labels` instead of `flet`). Now `fact1` is TRO-able and the recursion will be optimised as a loop.

TRO 5

Using a wrapper function could also help with type checks:

```
(defun fact (k)
  (unless (and (integerp k) (>= k 0))
    (error "Improper argument to fact: %s" k))
  (labels ((fact1 (k result)
             (if (zerop k) 1 (fact1 (1- k) (* k result)))))
          (fact1 k 1)))
```

As we have seen before, this takes the check out of the loop. Previously we used a standalone helper function for `fact`, but in principle a section of code could have local optimisation parameters (at least in CL), allowing for extra optimisation of the helper function as before.

The Same Length Question

From Talk 1, a check that all sequences have the same length:

```
(defun check-same-length (lists)
  "For a list of sequences, check that they are all the same
  (if (endp (cdr lists)) ; zero or one elts
      t
    (check-same-length-1 (length (first lists)) (rest lists
(defun check-same-length-1 (len lists)
  "Helper function for check-same-length: check that all se
  (cond
   ((endp lists) t)
   ((/= (length (first lists)) len) nil)
   (t (check-same-length-1 len (rest lists)))))
(check-same-length '((1 2) [2 3] "AB"))
t
(check-same-length '((1 2) (2) (3 4)))
nil
```

Types

Meditation: what is the difference between

```
(defun process-list (k)
  (if (null k) t
    (reduce #'bar (map 'list #'foo k))))
```

and

```
(defun process-list (k)
  (if (endp k) t
    (reduce #'bar (map 'list #'foo k))))
```

Hint: think of `(process-list [1 2])`

Types

What about this version?

```
(defun process-list (k)
  (declare (type list k))
  (if (null k) nil
    (reduce #'bar (map 'list #'foo k))))
```

Meditation: What does `declare` do? Hint: consider the difference between:

- ELisp checks that the argument is a list when it's passed in
- The programmer *promises* that `k` is a list, and consequences are undefined if it's not
- It's a hint to the compiler that what comes in is likely to be a list, and the compiler is free to ignore it
- It's like a comment

Answer: next slide

Types

Types are important for two reasons:

- Correctness – tracking types (at read/compile time) helps ensure that the program is correct
- Performance and optimisation – the compiler can make optimisations if it knows the specific type (eg. of a sequence or sequence element type)

Types can be specified in two ways:

- By the programmer. It is an error if a programmer-specified type does not match.
  - In CL, the consequences of a type error depend on the circumstances and what the standard/implementation define
- It can be inferred.
  - For example, in the sexp `(+ (floor a 6) (mod b 12))`,
    - `a` must be an integer, `(floor a 6)` is an integer
    - `b` must be an integer; `(mod b 12)` is of type `(and (integer 0 12) fixnum)` (or similar)
    - `+` can specialise to an integer-only (and only two parameters)

Types

Functional languages support type inference. For example, this will not work in ML-based languages:

```
let a = [2; (3,1); "ada"]
in ...
```

as the language will try to infer a list-of-something but it sees a number, a pair and a string.
In contrast, Lisp will cheerfully accept

```
(let ((a '(2 (3 . 1) "ada")))
  ...)
```

Although Emacs's type system is simpler it is worth learning (some of) CL's, partly for understanding functional programming as a paradigm, partly to uderstand Emacs' better.

Types and Inference

In a functional language like F#, types are inferred (in Linux, run `fsharpi` to get the interpreter):

```
> [2;3;4] ;;
val it : int list = [2; 3; 4]
```

Unlike arrays, in Lisp (E or C), we cannot directly specify the elements of a list:

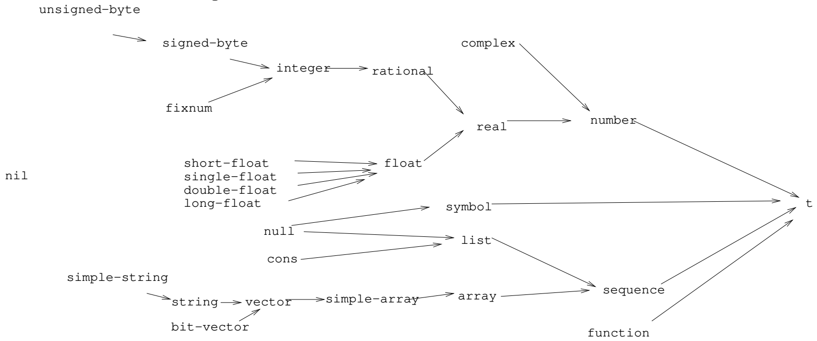
```
(typep '(2 3 4) 'list)
t
```

but in CL we can do

```
(defun intseqp (w) (every #'integerp w))
INTSEQP
(typep '(2 3 4) '(and list (satisfies intseqp)))
T
```

Types

In Lisp, everything is part of a type hierarchy:



This picture illustrates only the more important types. Note four specialised float types: SBCL provides only two, with `long` and `double` being conflated, and `short` and `single` conflated. In EL:

```
pi
3.141592653589793
(type-of pi)
float
```

Types and Inference

In CL, types like `array` and `complex` (but not `list`) are compound types and can optionally specify their element types:

```
(type-of #(1 2 3))
(SIMPLE-VECTOR 3)
(typep #(1 2 3) '(array integer *))
T
```

The `array` specifier is specialised with the type of element in the array and the dimensions. The symbol `*` may be used as a short-hand for "anything." Similarly, a rank-2 array could be specified with dimensions `(* *)`:

```
(typep #2A((1 2) (3 4)) '(array * (* *)))
T
```

Meditation: Note that `(array t)` is a proper subtype of `(array *)` – why?

Types – Generic Programming

Compare this with generic programming where the type is not known (until later):

```
let list_length x =
    let rec list_length1 x l =
        match x with
            | [] -> l
            | _ :: x1 -> list_length1 x1 (1+l)
    in list_length1 x 0
printfn "%d" (list_length [1;2;3])
printfn "%d" (list_length ["abra";"ca";"dabra"])
```

The type of `list_length` will be `'a list -> int`, meaning a list of a type `'a` mapping to an `int`.

map (in F#) would have type

```
('a -> 'b) -> 'a list -> 'b list
```

Types – Generic Programming

Suppose we wanted a list of mixed items: but F# will refuse the following

```
[3;"abc"; 1.2]
```

as the elements are three different types. Instead we use a “discriminated union”:

```
type Mixed =
    | String of string
    | Int of int
    | Float of float
printfn "%d" (list_length [Int 3; String "abc"; Float 1.2])
```

Types – Generic Programming

In C++, the types are managed in exactly the same way:

```
template<typename X>
std::size_t
list_length1(typename std::list<X>::const_iterator list,
             typename std::list<X>::const_iterator nil,
             std::size_t count)
{
    if(list == nil)
        return count;
    return list_length1<X>(++list, nil, 1+count);
}

template<typename X>
std::size_t
list_length(std::list<X> const &list)
{
    return list_length1<X>(list.cbegin(), list.cend(), 0);
}
```

Types – Generic Programming

In contrast, Lisp will (unless told otherwise) assume the most generic type:

- Lists are always assumed to be lists of `t`
- Arrays are assumed to be arrays of `t` (CL: unless spec'd)

```
(defun list-length (l)
  (labels ((list-length1 (l c)
             (if (endp l) c
              (list-length1 (cdr l) (1+ c)))))
    (list-length1 l 0)))
list-length
(list-length '(3 "abc" 1.2))
3
```

This will happily work with any type (including user-defined types):

- The Lisp code is not duplicated for each type, like the C++ code is
- However, (as in C++), specialisations are possible in CLOS (CL only)

Pattern Matching and Guards

A common example in (other) functional language is a sort of generalisation of cond (if such a thing can be imagined) where each case is a *pattern* with an optional *guard* – this is a contrived example in F#:

```
let rec func data sum =
    match data with
        | [] -> sum
        | (a,b) :: rest when a>b -> func rest (sum+a)
        | (_,b) :: rest -> func rest (sum+b)
        | _ -> failwith "could not parse data"
let test = [(3, 1) ; (4,1); (2,3) ; (-1, 2)]
in printfn "%d" (func test 0)
```

When run, it prints 12. Meditation: which simple refactoring would improve this function? (answer in a few slides)

Pattern Matching and Guards

The Mixed type from before can be deconstructed similarly:

```
type Mixed =
    | Float of float
    | String of string
    | Int of int

let explain (elt : Mixed) =
    match elt with
        | Float g -> printfn "Float %f" g
        | String s -> printfn "String %s" s
        | Int i -> printfn "Int %d" i

List.iter explain [Int 2; Float 2.71828; String "ada"]
```

In Lisp the same effect can be achieved with typecase (which works like case but matches on the type)

Pattern Matching and Guards

Lisp has the pattern matching function from macros' lambda lists as a standalone generalisation of `let`:

```
(destructuring-bind (a (b . c)) '(2 ((f g . y)))
  (list a b c))
(2 (f g . y) nil)
```

This will raise an error:

```
(destructuring-bind (a (b c)) '(1 2) (list a b c))
```

This means we can use it to bind variables in an expression, but not easily in a negative match.

Pattern Matching and Guards

Obviously we could write the function the conventional way (still without the helpful refactoring):

```
(defun func (data sum)
  (cond
   ((endp data) sum)
   ((> (caar data) (cdar data))
     (func (rest data) (+ sum (caar data))))
   ((and (consp (car data)) (atom (cdar data)))
     (func (rest data) (+ sum (cdar data))))
   (t (error "Failed to process ~S" data))))
```

That third test is a bit hand-made (it tests for a cons cell). Things to note:

- Unlike F#, it doesn't make any bindings in the clauses
- It doesn't pattern match in the sense that `destructuring-bind` does

Pattern Matching and Guards

Of course we could still write the pattern explicitly:

```
(cond
  ((endp data) sum)
  ((ignore-errors
      (destructuring-bind (a . b) (first data)) (> a b))
    (func (rest data) (+ sum a)))
  ...
```

Here, `ignore-errors` will return `nil` if `destructuring-bind` fails to match; if it *does* match, the value of the test `(> a b)` is returned. As before, if the clause is true, the function is called recursively.

In CL's Alexandria library, there is a `destructuring-case`. In EL, dash has `-when-let` (as a clause) and `-let` can do destructuring.

Pattern Matching and Guards

Here is a cleaner way of handling the problem:

```
(defun func (sum val)
  (destructuring-bind (a . b) val
    (+ sum (if (> a b) a b))))
func
(reduce #'func
  '((3 . 1) (4 . 1) (2 . 3) (-1 . 2))
  :initial-value 0)
12
```

Incidentially, `loop` can also destructure:

```
(loop for (a . b) in '((3 . 1) (4 . 1) (2 . 3) (-1 . 2))
      summing (max a b))
12
```