Programming Languages & Paradigms
PROP HT 2011

Lecture 14

Functional Programming III – Functional Programming in Clojure

Beatrice Åkerblom
beatrice@dsv.su.se

Limits for Recursion

- There is a limit on the number of possible nested functions that is machine-specific
  - Imperative languages solves this through not using recursion for everything
  - Functional languages solves this through tail-call-optimisation
  - Recursion from a tail position is in many ways like a structured goto, and has more in common with an imperative loop than it does with other kinds of recursion
  - Clojure does not automatically use tail-call-optimisation, but we can explicitly ask for it using `recur`


```
user=> (defn add-up
         "adds all the numbers below a given limit"
         ([limit] (add-up limit 0 0))
         ([limit current sum]
           (if (< limit current)
              sum
              (add-up limit (+ 1 current) (+ current sum)))))
#'user/add-up

user=> (add-up 3)
6

user=> (add-up 10000)
StackOverflowError   java.lang.Number.<init> (Number.java:32)
```



```
user=> (defn add-up
         "adds all the numbers below a given limit"
         ([limit] (add-up limit 0 0))
         ([limit current sum]
            (if (< limit current)
                sum
                (recur limit (+ 1 current) (+ current sum)))))
#'user/add-up

user=> (add-up 3)
6

user=> (add-up 10000)
50005000

(fn [x] (recur x) (println x))
; java.lang.UnsupportedOperationException:
;    Can only recur from tail position
```

Side Effects

- Clojure avoids side effects, but some tasks are, by nature, side effects, e.g.:
  - IO
  - Explicit state management
  - Java interaction
- Using `do`, all the expressions will be evaluated, but only the last one will be returned:

```
user=> (do
          (println "hello")
          (println "from")
          (println "side effects")
          (+ 5 5))
hello
from
side effects
10

user=> (do (let [in (read-line)]
         (println in)))
hej
hej
nil
```


Side Effects, cont'd

- Functions can also be written to contain side effects, by providing multiple expressions instead of just one as the body of the function

```
user=> (defn square
   "Squares a number, with side effects"
   [x]
   (println "Squaring" x)
   (println "The return value will be" (* x x))
   (* x x))
#'user/square

user=> (square 8)
Squaring 8
The return value will be 64
64
```

- This can also be done for loops

Functional Programming Techniques

"One can write FORTRAN in any language"

Functional Programming

- First-class functions & immutable data

Earlier conclusion: We “can” write our OO programs in C

We “can” write our functional programs in C

Functions are First Class Objects

- Functions are first-class objects that can be:
  - dynamically created at any time during runtime
  - used in the same way as any value
  - stored in Vars, held in lists and other collection types
  - passed as arguments to and returned as the result of other functions

```
user=> (def my-funcs [make-a-set make-a-set-2 print-down-from])
#'user/my-funcs

user=>  (nth my-funcs 0)
#<user$make_a_set user$make_a_set@44755866>

user=> ((nth my-funcs 0) 2 3 4 5)
#{2 3 4 5}
```


Pure functions

- Pure functions are functions without side effects
  - do not depend on anything but arguments
  - only influence on the outside world is through return value
  - use immutable data
- Though Clojure is designed to minimize and isolate side-effects, it’s by no means a purely functional language
- Pure functions are easy to develop, test, and understand, and you should prefer them for many tasks

Higher-order Functions

- A higher-order function, or functional form, is one that either takes functions as parameters or yields a function as its result, or both

  Form: `h ≡ f ° g`

  which means `h(x) ≡ f(g(x))`

- Example:

  `f(x) ≡ x + 2`

  `g (x) ≡ 3 * x`

  then `h ≡ f ° g` yields `(3 * x)+ 2`

Functions as Arguments

- Functions that take other functions as arguments are very useful and very common

```
user=> (map pos? '(1 -2 3 -4 5))
(true false true false true)

user=> (filter pos? '(1 -2 3 -4 5))
(1 3 5)

user=> (reduce + '(1 -2 3 -4 5))
3

...
```


Functions as Arguments, cont'd

```
user=> (defn arg-switch
         [f arg1 arg2]
         (list (f arg1 arg2) (f arg2 arg1)))
#'user/arg-switch

user=> (arg-switch str "clo" "jure")
("clojure" "jureclo")

user=> (arg-switch - 2 5)
(-3 3)

user=> (arg-switch (fn [a b] (/ a (* b b))) 2 3)
(2/9 3/4)
```


Functions as Return Values

```
user=> (defn rangechecker
         [min max]
         (fn [num]
             (and (<= num max) (<= min num))))
#'user/rangechecker

user=> (def myrange (rangechecker 5 10))
#'user/myrange

user=> (myrange 7)
true

user=> (myrange 22)
false
```

Closures

- A closure is a first class function that contains values as well as code, the locals from the context in which it was defined

```
user=> (def times-two
          (let [x 2]
            (fn [y] (* y x))))
#'user/times-two

user=> (times-two 12)
24

user=> (defn times-n [n]
  (let [x n]
    (fn [y] (* y x))))
#'user/times-n

user=> (def times-four (times-n 4))
#'user/times-four

user=> (times-four 10)
40
```


Closures, cont'd

```
user=> (defn divisible [denom]
  (fn [num]
    (zero? (rem num denom))))
#'user/divisible

user=> ((divisible 3) 6)
true

user=> ((divisible 3) 7)
false

user=> (filter (divisible 4) (range 10))
(0 4 8)
```


Currying vs Partial Application

- In e.g. ML and Haskell, all functions are really unary functions (i.e. they accept a single argument) and functions of n arguments are actually unary functions returning closures in n "layers"
  - The function f (x, y, z) -> N will in fact be built up by several functions (x -> (y -> (z-> N)))
- Partial application returns a function which takes fewer arguments, the others having been bound. Applying one value to a function taking three arguments will give us a new function taking two arguments

```
user=> (def add-four (partial + 4))
#'user/add-four
user=> (add-four 4)
8
```


Referential Transparency

- If a function of some arguments always results in the same value and changes no other values within the greater system, then it's essentially a constant, or referentially transparent (the reference to the function is transparent to time)
  - Function call may be replaced by its value without changing the program's behaviour

```
user=> (defn square
   ([num] (* num num)))
#'user/square
user=> (square 4)
16

user=> (print (.getTime (now)))
1323776904436nil
```

- Pure functions are referentially transparent by definition. Most other functions are not referentially transparent, and those that are must be proven safe by code review.

Immutability

- Immutable data is critical to Clojure's approach to both FP and concurrency.
- When all data is immutable, "update" translates into "create a copy of the original data, plus my changes."
  - This will use up memory quickly!

Persistent Data Structures

- Persistent data structures always preserves the previous version of itself when it is modified
- This means that they are effectively immutable
  - their operations do not make (visibly) updates in-place
  - operations always yield a new updated structure
- All data structures in Clojure are persistent

Persistent Data Structures

- Persistent data structures always preserves the previous version of itself when it is modified
- This means that they are effectively immutable
  - their operations do not make (visibly) updates in-place
  - operations always yield a new updated structure
- All data structures in Clojure are persistent

Note that the word persistent here has nothing to do with disk storage or data bases

Persistent Data Structures in Clojure

- All versions will have the same update and lookup complexity guarantees
- The specific guarantees depend on the collection type

- Java array (non-persistent):

```
user=> (def ds (into-array [:willie :barnabas :adam]))
(:willie :barnabas :adam)
user=> user=> #'user/ds

user=> (seq ds)    ;repl doesn't know how to print Java objects
(:willie :barnabas :adam)
```

- Change to the Java array `ds` happens in- place, obliterating any historical version:

```
user=> (aset ds 1 :quentin) ;set pos 1 to :quentin
:quentin

user=> (seq ds)
(:willie :quentin :adam)
```

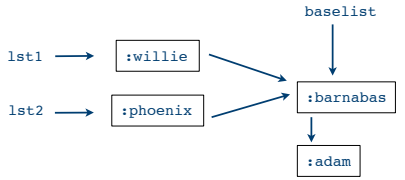
Persistent Data Structures in Clojure, cont'd

- Clojure's persistent data structures give a different result:

```
user=> (def ds [:willie :barnabas :adam])
#'user/ds

user=> ds
[:willie :barnabas :adam]

user=> (def ds1 (replace {:barnabas :quentin} ds))
#'user/ds1

user=> ds
[:willie :barnabas :adam]

user=> ds1
[:willie :quentin :adam]
```

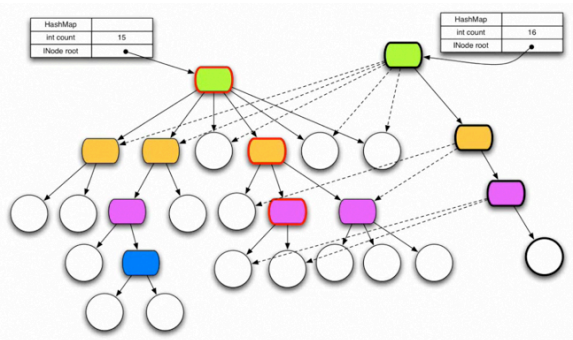

Persistent Data Structures in Clojure, cont'd

```
user=>  (def baselist (list :barnabas :adam))
#'user/baselist
user=> (def lst1 (cons :willie baselist))
#'user/lst1
user=> (def lst2 (cons :phoenix baselist))
#'user/lst2

user=> baselist
(:barnabas :adam)
user=> lst1
(:willie :barnabas :adam)
user=> lst2
(:phoenix :barnabas :adam)

user=> (= (next lst1) (next lst2))
true
user=> (identical? (next lst1) (next lst2))
true
```



Persistent Data Structures in Clojure, cont'd

- Data structures in Clojure are implemented as trees
- Untouched parts are reused in "copy"
- Structures that can't be reached will be garbage collected



The End

References

- Sebesta, R., “Concepts of Programming Languages”
- Biancuzzi, F. and Warden, S.,“Masterminds of Programming – Conversations with the Creators of Major Programming Languages”, 2009
- Fogus, M. and Houser, C., “The Joy of Clojure”, 2011
- Halloway, S., “Programming Clojure”, 2009
- VanderHart, L. and Sierra, S., “Practical Clojure”, 2010
- clojure.org
- http://blog.higher-order.net/2009/02/01/understanding-clojures-persistentvector-implementation/