FUNCTIONAL AND CONCURENT PROGRAMMING

SI4

Pascal URSO

MORE ABOUT FUNCTIONS

Closure, Currying, Tail Reccursion

CLOSURE

CLOSURE

- **Functions have a context**
- Example
  - ```
    Lst<Integer> greaterThan(Lst<Integer> l, int v) {
        return filter(x -> x > v, l);
    }
    ```
  - `v` is defined in the context of the lambda `x -> x > v`
- **Closure = function + context**
- **What we use to call "function" is a closure!**
  - Function may use or not this context
    ```

CLOSURE AND METHOD REFERENCES

- **Every functional interface has a context**
- Example
  - ```
    class Counter {
        int v = 0;
        public int next() {
            return v++;
        }
    }
    ```
  - counterInstance::next has v in its context
  - Counter::next does not
    ```

CLOSURE AND VARIABLES

- Functions cannot modify local variables
  - ```
    static void print(Lst<T> l) {
        int i = 0;
        iter(x -> { System.out.println((i++) + ": " + x); }, l);
    }
    ```
- Accessed variables must be (implicitly) final
  - ```
    static void print(Lst<T> l) {
        int i = 0; i += 1;
        iter(x -> { System.out.println(i + ": " + x); }, l);
    }
    ```

CLOSURE AND SIDE-EFFECT

- But closures can access mutable objects
  - ```
    static void print(Lst<T> l) {
        final AtomicInteger i = new AtomicInteger(0);
        iter(x -> { System.out.println(i.getAndIncrement() + ": " + x); }, l);
    }
    ```
- Or even modify (instance or class) attributes
  - ```
    static int i;
    static void print(Lst<T> l) {
        i = 0;
        iter(x -> { System.out.println((i++) + ": " + x); }, l);
    }
    ```

NON-SAFE CLOSURE

- A closure that uses a mutable context is "NON-SAFE" (aka non-functional)
- No longer a pure function (something else may modify the context)
  - Non repetable
  - Difficult to test
  - Difficult to parallelise
  - …

SAFETY AND PARALLELISM

- A non-safe closure is still **parallelisable** if the context is thread-safe
  - AtomicInterger, AtomicBoolean, AtomicLong, AtomicLongArray, …
  - **synchronized** instructions
  - Synchronized data structures: Collections.synchronizedList(), …
- But easily **not distributable**
  - Ref shared nothing infrastructures

NON-FUNCTIONALITY

- This print is not functional
  - ```
    static int i;
    static void print(Lst<T> l) {
        i = 0;
        iter(x -> { System.out.println((i++) + ": " + x); }, l);
    }
    ```
- This one is not (but still thread-safe)
  - ```
    static void print(Lst<T> l) {
        final AtomicInteger i = new AtomicInteger(0);
        iter(x -> { System.out.println(i.getAndIncrement() + ": " + x); }, l);
    }
    ```

CURRYING

CURRYING

- Translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument
- Example
  - f.apply(a, b) => fc.apply(a).apply(b)
  - `T x R → U` => `T → (R → U)`

  - `int add(int x, int y) { return x + y; }`
  - `Function<Integer,Integer> addc(int x) { return y -> x + y; }`

CURRYING ADVANTAGE

- Make a closure functional
-
```
class Foo {
    Integer b;
    Lst<Integer> calc(Lst<Integer> l) {
        return map(x -> x * b, l);
    }

    Lst<Integer> calcf(Lst<Integer> l) {
        Function<Integer, Function<Integer, Integer>> cf =
                y -> x -> x * y;
        return map(cf.apply(b), l);
    }
}
```
- The map in calcf is thread-safe while not in calc

CURRYING USAGE EXAMPLE

- Have more than 2 parameters (beside BiFunction)
  - `T1 x T2 x … x TN → R => T1 → T2 → … → TN → R`
- Builder Pattern
  - ```
    class Letter {
        static AddReturnAddress builder(){
            return returnAddress
                    -> closing
                    -> dateOfLetter
                    -> insideAddress
                    -> salutation
                    -> body
                    -> new Letter(returnAddress, insideAddress, dateOfLetter,
    salutation, body, closing);
        }

        interface AddReturnAddress { Letter.AddClosing withReturnAddress(String
    returnAddress); }
    ```
    ```

TAIL RECURSION

TAIL RECURSION

- ***A function is tail recursive*** *if the recursive call is the* ***last statement*** *that is executed by the function*
  - ```java
    static void print(int n) {
         if (n < 0) return;
         System.out.print(" " + n);
         print(n - 1);
    }
    ```
- But not
  - ```java
    static int fact(int n) {
         if (n == 0) return 1;
         return n * fact(n - 1);
    }
    ```

WHY TAIL RECURSION?

- A tail recursion fonction can be executed as a loop
  - So without call stack comsuption
  - Many languages compile/interpret tail recursion as loop (Tail Call Optimization)
  - But not (yet) Java ☹
    - Only with specific libraries (e.g. https://github.com/Judekeyser/tail_rec)
- Better control on time complexity
  - Double recursion : $O(2^n)$ versus tail recursion : $O\left(n.c_{body}(n)\right)$ - Usually
  - But still programming easiness of recursion

TAIL ACCUMULATOR

- Tail recursive fonction often use helper functions with supplementary parameters for
  - Loop variables
  - Or acummulator (the result being calculated)
- Example
  - ```
    static long power(int base, int exponent) {
        return power(base, exponent, 1);
    }

    static long power(int base, int exponent, int res) {
        if (exponent == 0) return res;
        return power(base, exponent - 1, res * base);
    }
    ```
- Helps memoization (dynamic programming approach)
    ```

COMPLEXITY COMPARED

Function	Non-tail rec	Tail rec
reverse list	$O(n^2)$	$O(n)$
fibonnaci	$O(2^n)$	$O(n)$
palindrome	$O(n^2)$	$O(n)$
edit distance	$O(2^n)$	$O(n^2)$
…		

- Not always easy (e.g. tree traversals) or possible without stack simulation (e.g. quicksort)