Modul- Fortgeschrittene Programmierkonzepte

Bachelor Informatik

13- Streams

Prof. Dr. Marcel Tilly

Fakultät für Informatik, Cloud Computing

Vers. 23.1.2021

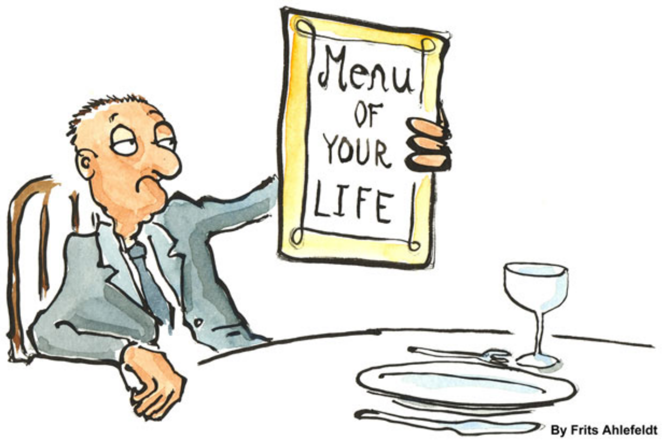
Agenda for today!

Technische Hochschule Rosenheim

On the menue for today:

- FP WHY?
- FP Recap
- Streams

Enjoy!



Why is FP useful?

Technische Hochschule Rosenheim

There are a couple of *pros* why FP makes sense:

- Reduces code redundancy
- Improves modularity
- Helps us to solve complex problems
- Increases maintainability
- No-Side effects -> Parallel execution of code

BUT...

There are some thougths against FP:

- Writing pure functions is easy, but combining them into a complete application is where things get hard
- The advanced math terminology (monad, monoid, functor, etc.) makes FP intimidating
- For many people, recursion doesn't feel natural
- Pure functions and I/O don't really mix
- Using only immutable values and recursion can potentially lead to performance problems, including RAM use and speed
- Because you cannot mutate data, you copy data all the time

FP Recap

Reduction

Working on lists (and later streams), we defined three main methods:

- `filter` utilizing a `Predicate<T>` to retain only certain elements,
- `map` utilizing a `Function<T, R>` to transform a list of elements of type `T` to a list of type `R`
- `forEach` utilizing a `Consumer<T>` that accepts (in order of the list).

Working on lists, we defined those *recursively*:

```
static <T> List<T> filter(List<T> xs, Predicate<T> p) {
    if (xs.isEmpty()) return xs;
    else if (p.test(xs.head)) return list(xs.head, filter(xs.tail, p));
    else return filter(xs.tail, p);
}
```

FP Recap

Can we remember how to formuate `map` and `forEach`:

FP Recap

Technische Hochschule Rosenheim

Can we remember how to formuate `map` and `forEach`:

```
static <A, B> List<B> map(List<A> xs, Function<A, B> f) {
    if (xs.isEmpty()) return empty();
    else return list(f.apply(xs.head), map(xs.tail, f));
}
```

FP Recap

Can we remember how to formuate `map` and `forEach`:

```
static <A, B> List<B> map(List<A> xs, Function<A, B> f) {
    if (xs.isEmpty()) return empty();
    else return list(f.apply(xs.head), map(xs.tail, f));
}
```

```
static <A> void forEach(List<A> xs, Consumer<A> c) {
    if (xs.isEmpty()) return;
    else {
        c.accept(xs.head);
        forEach(xs.tail, c);
        // return (added for clarity)
    }
}
```

Reduction

Here are three key observations:

1. All three methods "iterate" over the list, i.e. all elements are visited.
2. The `forEach` method is *tail recursive*, as in the recursive call is the very last one prior to `return`.
3. The `filter` and `map` methods return another list, while `forEach` returns nothing (`void`).

In this (final) chapter, we'll talk about list (or stream) *reduction*, that is reducing a sequence of values to a single value.

Reduction

Let's start with a simple example: **sum all numbers of a list**

```
static int sum(List<Integer> xs) {
    if (xs.isEmpty()) return 0;  // sum of an empty list is zero
    else return xs.head + sum(xs.tail);
}
```

For `list(1, 3, 3, 7)`, this function evaluates (unfolds) to

Reduction

Let's start with a simple example: **sum all numbers of a list**

```
static int sum(List<Integer> xs) {
    if (xs.isEmpty()) return 0;  // sum of an empty list is zero
    else return xs.head + sum(xs.tail);
}
```

For `list(1, 3, 3, 7)`, this function evaluates (unfolds) to

```
sum(list(1, 3, 3, 7))
-> 1 + sum(list(3, 3, 7))
-> 1 + (3 + sum(list(3, 7)))
-> 1 + (3 + (3 + sum(list(7))))
-> 1 + (3 + (3 + (7 + sum(empty()))))
-> 1 + (3 + (3 + (7 + 0)))
-> 1 + (3 + (3 + 7))
-> 1 + (3 + 10)
-> 1 + 13
-> 14
```

Reduction

As you can see, the recursion expands until the terminal case is reached, and the first `return` happens. Then the addition is done all the way back up the call stack.

The recursion depth is as many as there are list elements.

Can we formulate it differently?

Tail recursion

Alternative:

```
static int sum(List<Integer> xs, int z) {
    if (xs.isEmpty()) return z;
    else return sum(xs.tail, z + xs.head);
}
```

which evaluates to

```
sum(list(1, 3, 3, 7), 0)
-> sum(list(3, 3, 7), 0 + 1)
-> sum(list(3, 7), 1 + 3)
-> sum(list(7), 4 + 3)
-> sum(empty(), 7 + 7)
-> 14
```

Depending on the language, they can be realized as a for-loop reusing the stack variables.

Tail recursion

Technische Hochschule Rosenheim

Definition

A recursive function is **tail recursive** if the final result of the recursive call is the final result of the function itself.

Let's consider another example: **joining Strings together by concatenating them**

```
static String join(List<String> xs, String z) {
    if (xs.isEmpty()) return z;
    else return join(xs.tail, z + xs.head));
}
```

taken from https://wiki.haskell.org/Tail_recursion

Towards `Reduce`

The `sum` and `join` functions look almost identical -- the only difference being the `Integer` and `String` types.

Why not generalize?

Towards `Reduce`

The `sum` and `join` functions look almost identical -- the only difference being the `Integer` and `String` types.

Why not generalize?

```
static <T> T reduce(List<T> xs, T z) {
    if (xs.isEmpty()) return z;
    else return reduce(xs.tail, z + xs.head);
}
```

Towards `Reduce`

The `sum` and `join` functions look almost identical -- the only difference being the `Integer` and `String` types.

Why not generalize?

```
static <T> T reduce(List<T> xs, T z) {
    if (xs.isEmpty()) return z;
    else return reduce(xs.tail, z + xs.head);
}
```

> **OOPs!** Unfortunately, the `+` operator is only defined for basic types (including `java.lang.String`), and Java does not support operator overloading.

Towards `Reduce`

But look closer what the `+` actually is: it is a binary operation to combine two values to a single value of the same type. Both `String` and `Integer` actually offer such methods:

```
static int reduce(List<Integer> xs, int z) {
    if (xs.isEmpty()) return z;
    else return sum(xs.tail, Integer.sum(z, xs.head));
}

static String reduce(List<String> xs, String z) {
    if (xs.isEmpty()) return z;
    else return join(xs.tail, z.concat(xs.head));
}
```

Introduce `apply`

Let's isolate the operation, using the interface `java.util.function.BinaryOperator<T>`:

```
interface BinaryOperator<T> {
    T apply(T t1, T t2);
}
```

```
static <T> T reduce(List<T>, T z, BinaryOperator<T> op) {
    if (xs.isEmpty()) return z;
    else return reduce(xs.tail, op.apply(z, xs.head), op);
}
```

Introduce `apply`

```
reduce(list(1, 3, 3, 7), 0, (i, j) -> Integer.sum(i, j));  // 14
reduce(list(1, 3, 3, 7), 0, Integer::sum);
```

```
reduce(list("a", "b", "c", "d"), "", (a, b) -> a.concat(b));  // abcd
reduce(list("a", "b", "c", "d"), "", String::concat);
```

It may sound odd, but `forEach` is actually a special case of `reduce`:

```
reduce(list(1, 3, 3, 7), 0, (i, j) -> {
        System.out.println(j);
        return j;
    });
```

Left Fold

The `reduce` function is a bit restricted:

> it only works to reduce elements of type `T` to another `T`.

This might be a problem: consider the case where you sum up a very long list of potentially large `Integer`s -- you may run into an overflow.

The solution to this would be to add the `Integer`s from the list to a [BigInteger]() which is of arbitrary precision.

In terms of a `for`-loop, this would be

```
BigInteger sum = BigInteger.ZERO;
for (Integer i : xs) {
    sum = sum.add(BigInteger.valueOf(i));
}
```

Left Fold

So our hypothetical `reduce` function for this would be

```
static BigInteger reduce(List<Integer> xs, BigInteger z) {
    if (xs.isEmpty()) return z;
    else return reduce(xs.tail, z.add(BigInteger.valueOf(xs.head)));
}
```

```
reduce(list(1, 3, 3, 7), BigInteger.ZERO);
```

By now, you probably already guessed it: we'll isolate the actual operation!

Left Fold

We need a function that takes a `BigInteger` (the accumulator), adds an `Integer`, and returns a `BigInteger`.

We'll do so with the interface `java.util.function.BiFunction<T, U, R>` (but tying `T` and `R`), and naming it `foldl` (read: *fold left*).

```
static <T, R> R foldl(List<T> xs, R z, BiFunction<R, T, R> op) {
    if (xs.isEmpty()) return z;
    else return foldl(xs.tail, op.apply(z, xs.head), op);
}
```

```
foldl(xs, BigInteger.ZERO, (b, i) -> b.add(BigInteger.valueOf(i)));
```

Left Fold

The function is called *left fold*, since the list is folded *to the left*, if you were to look at the evaluation:

```
foldl(list(1, 3, 3, 7), 0)
-> foldl(list(3, 3, 7), 0+1)
-> foldl(list(3, 7), 1+3)
-> foldl(list(7), 4+3)
-> foldl(empty(), 7+7)
-> 14
```

and visualized that a list, the operations are performed in this order:

```
// start at the bottom left!
        op
       / \
      op  7
     / \
    op  3
   / \
  op  3
 / \
z   1
```

Left Fold

Look at that list again, doesn't it look oddly familiar?

If we define `z` as the empty list, and `op` is the list constructor, you end up with the *reverse* of the original list:

```
foldl(list(1, 3, 3, 7), List.<Integer>empty(),
    (xs, x) -> list(x, xs));  // 7, 3, 3, 1
```

Right Fold

Let's go back to the original, *non-tail-recursive* definition of `sum`:

```
static int sum(List<Integer> xs) {
    if (xs.isEmpty()) return 0;  // sum of an empty list is zero
    else return xs.head + sum(xs.tail);
}
```

```
static BigInteger sum(List<Integer> xs, BigInteger z) {
    if (xs.isEmpty()) return z;
    else return BigInteger.valueOf(xs.head).add(sum(xs.tail, z));
}
```

Right Fold

If you isolate the operation (`+` or `.add()`, respectively), you end up with a *right fold*:

```
static <T, R> R foldr(List<T> xs, R z, BiFunction<T, R, R> op) {
    if (xs.isEmpty()) return z;
    else return op.apply(xs.head, foldr(xs.tail, z, op));
}
```

```
foldr(list(1, 3, 3, 7), BigInteger.ZERO,
    (i, b) -> BigInteger.valueOf(i).add(b));  // 14
```

Right Fold

Again, look at the order of operations:

```
 op
 / \
1   op
    / \
   3   op
       / \
      3   op
          / \
         7   z
```

To complete the top most operation, you need decend all the way down the fold.

Right Fold

Technische Hochschule Rosenheim

Again, does that look familar? If we define `z` as a list and `op` as the list construction, we end up with `append`:

```
foldr(xs, List.<Integer>list(49), (z, zs) -> list(z, zs));
// 1, 3, 3, 7, 49
```

If we add in some logic, we get `map`:

```
foldr(xs, List.<Integer>empty(), (z, zs) -> list(z*z, zs));
// squares: 1, 9, 9, 49
```

And even `filter`:

```
// retain all values less than 5
foldr(xs, List.<Integer>empty(), (z, zs) -> {
    if (z < 5) return zs;
    else return list(z, zs);
});
// 7
```

Tail Recursive Map

Unfortunately, right fold is *not* tail-recursive, making it an undesirable operation.

The trick is to apply a left fold twice:

- in a first step, we'll use `foldl` to *reverse* the list,
- then we'll use it *again* to reverse it to its original order and applying the mapping function:

```
static <T, R> List<R> maptr(List<T> xs, Function<T, R> op) {

    List<T> reverse = foldl(xs, empty(), (ys, y) -> list(y, ys));

    List<R> mapped = foldl(reverse, empty(),
        (ys, y) -> list(op.apply(y), ys));

    return mapped;
}
```

Streams in Java

Generation

- `Stream.of(...)` with array or varargs
- `Collection.stream()`, if supported
- `Stream.generate(...)` using a generator
- Popular APIs, e.g. `Pattern.compile("\\W").splitAsStream("hello world");`

Streams in Java

Technische Hochschule **Rosenheim**

Intermediate Operations

You already know most of the intermediate operations:

- `filter(Predicate<T> p)` removes/skips unwanted elements in the stream
- `map(Function<T, R> f)` transforms a `Stream<T>` into a `Stream<R>` using the provided `Function`
- `sorted(Comparator<T> comp)` returns a sorted stream
- `concat(Stream<T> s)` appends another stream
- `distinct()` removes duplicates
- `skip(int n)` and `limit(int n)` skip elements and truncate the stream

Technische Hochschule Rosenheim

flatMap

Another notable intermediate operation is `flatMap` which transforms a stream of sequences (lists, streams, etc.) into a single *flat* sequence.

```
// list-of-lists
Stream<List<Integer>> lol = Stream.of(
  Arrays.asList(1, 2),
  Arrays.asList(3, 4),
  Arrays.asList(5)
);

Stream<Integer> integerStream = lol.flatMap(al -> al.stream());
integerStream.forEach(System.out::print);  // 12345
```

Iteration and Reduction

Last week, we already talked about `forEach(Consumer<T> c)` which can be used to iterate over the whole stream, and pass each element to the `Consumer`.

This week, we learned about the `reduce` functions, which are implemented in Java as `reduce(T identity, BinaryOperator<T> op)` and the more more generic `reduce(U identity, BiFunction<U, ? super T, U> op, BinaryCombiner<U> com)`.

```
Stream.of(1, 3, 3, 7).reduce(0, Integer::sum));
// 14

Stream.of(1, 3, 3, 7).reduce(BigInteger.ZERO,
    (bi, i) -> bi.add(BigInteger.valueOf(i)),
    (bi1, bi2) -> bi1.add(bi2)));  // combine identity with first result
// 14
```

> NOTE: The second operation can often be defined simpler as a `map` followed by a `reduce`

Terminal Operations

Technische Hochschule Rosenheim

- Use `.forEach(Consumer<T> c)` to pass each element to the `Consumer`
- Use `reduce` to combine (and optionally map) elements of a stream.

```
Stream.of(1, 3, 3, 7).reduce(0, Integer::sum));
// 14

Stream.of(1, 3, 3, 7).reduce(BigInteger.ZERO,
    (bi, i) -> bi.add(BigInteger.valueOf(i)),
    (bi1, bi2) -> bi1.add(bi2)));  // combine identity with first result
// 14
```

- Use `collect` to collect/distribute elements to other structures.

```
List<Integer> list1 = new LinkedList<>();
Stream.of(1, 3, 3, 7).forEach(i -> list.add(i));

// or shorter, using collect
List<Integer> list2 = Stream.of(1, 3, 3, 7).collect(Collectors.toList()));
```

Collectors 1/2

Another powerful tool provided by the Java Streams API is `collect` which is a special form of stream reduction.

The idea is to iterate over the stream and pass each element to a *combiner* that builds up a data structure. A classic example is to turn a `Stream` into a `List`:

```
List<Integer> list1 = new LinkedList<>();
Stream.of(1, 3, 3, 7).forEach(i -> list.add(i));

// or shorter, using collect
List<Integer> list2 = Stream.of(1, 3, 3, 7).collect(Collectors.toList()));
```

Collectors 2/2

Java provides a [lengthy list of collectors] for your convenience. Here are a few examples from the docs, most notably `groupingBy` und `partitioningBy`.

```
// Accumulate names into a TreeSet
Set<String> set = people.stream()
    .map(Person::getName)
    .collect(Collectors.toCollection(TreeSet::new));

// Convert elements to strings and concatenate them, separated by commas
String joined = things.stream()
    .map(Object::toString)
    .collect(Collectors.joining(", "));

// Compute sum of salaries of employee
int total = employees.stream()
    .collect(Collectors.summingInt(Employee::getSalary));

// Group employees by department
Map<Department, List<Employee>> byDept = employees.stream()
    .collect(Collectors.groupingBy(Employee::getDepartment));

// Compute sum of salaries by department
Map<Department, Integer> totalByDept = employees.stream()
    .collect(Collectors.groupingBy(Employee::getDepartment,
        Collectors.summingInt(Employee::getSalary)));

// Partition students into passing and failing
Map<Boolean, List<Student>> passingFailing = students.stream()
    .collect(Collectors.partitioningBy(s -> s.getGrade() <= 400));
```

Finding Values in a Stream

Technische Hochschule Rosenheim

Use `findFirst()`, `min()` or `max()` to find values, returns `Optional<T>`!

- Often you need to find certain values in a stream, such as `findFirst()`, `min()` or `max()`.
- Since these are methods that are often used on streams that are potentially empty, they return an [Optional](#).
- Optionals are similar to futures, as in you can `get()` the content if it `isPresent()`.
- They can also be mapped to another `Optional`, or used as a `.stream()`.

Verifying Values in a Stream

Another frequent use case: Verify if *all*, *any* or *none* of the elements in a stream match a certain criteria.

Use the `allMatch`, `anyMatch` and `noneMatch` functions, which take a `Predicate<T>` as argument.

Parallel Processing

There is a separate document on [parallel streams](), but in short, just use `parallelStream()` to enable parallel processing.

For example, to group People by their gender, you can use

```
Map<Person.Gender, List<Person>> byGender = allPeople
    .stream()
    .collect(Collectors.groupingBy(Person::getGender));

// or parallel
ConcurrentMap<Person.Gender, List<Person>> byGender = allPeople
    .parallelStream()
    .collect(Collectors.groupingByConcurrent(Person::getGender));
```

Final Thought!

•