List Comprehensions in Java

Franco Arolfo
(Dated: January 28, 2016)

In many programming languages such as Haskell or python is popular to write list comprehensions, as have in algebra $\{x \mid x \in \mathbb{R} \wedge x \equiv 0 \mod 2\}$ as the list of all even numbers in $\mathbb{R}$. This article proposes an implementation of list comprehensions in Java, and then provides the implementation of the functions *map* and *filter* using the proposed code. Also, it supports binding more than one variable to the list's definition, so we will be able to define lists of cartesian products and to add a relation that must hold between the two variables, like $\{(x, y) \mid x \in \{1, 2, 3\} \wedge y \in \{4, 5, 6\} \wedge 2 * x = y\}$

I. INTRODUCTION

We may remember from algebra that we can define lists as *lists comprehensions*, which use a special notation called set-builder notation. For example,

$$\{(x, y) \mid x \in \mathbb{R} \wedge y \in \mathbb{R} \wedge x * y = x + y\}$$

denotes the set of all pairs (x, y) such that x and y are real numbers and the product of both numbers equals the sum.

Today, this syntactic construct is integrated in many programming languages, such as Haskell

```
[x * 2 | x <- [0 .. 99], x * x > 3]
```

python

```
S = [2*x for x in range(100) if x**2 > 3]
```

C#

```
var ns = Enumerable.Range(0, 100)
         .Where(x => x*x > 3)
         .Select(x => x*2);
```

Scala

```
val s = for (x <- Stream.from(0);
         if x*x > 3) yield 2*x
```

Ruby

```
(0..100).select { |x| x**2 > 3 }
         .map { |x| 2*x }
```

and more. But the Java language does not provide a syntactic construct for this concept.

The usage of list comprehensions in Haskell motivated this article. Bringing the set-builder notation to Java, this *different way of thinking problems*. Instead of using *for* or *while* statements in your programs, you may work with *map* and *filter* functions, but take into account that thinking problems as list comprehensions is also another good way to practice programming.

The structure of this article follows:
1. Set-builder notation and list comprehensions definitions as we saw in algebra.
2. Some words about the implementation of Haskell's list comprehensions. The implementation of *map* and *filter* with list comprehensions in Haskell.
3. The proposed implementation of list comprehensions in Java. The implementation of *map*, *filter* and the cartesian product with our new list constructor in Java.
4. Discussion. Is this useful at industrial level? Possible enhancements and future work.

II. SET BUILDER NOTATION

In set theory there is a popular way of defining sets: *set-builder notation*. Any list defined in this way is called a list comprehension. For example, let's define the set of all integers that are even and are bigger or equals than 5

$$\{x \mid x \in \mathbb{Z} \wedge x \text{ is even} \wedge x \geq 5\}$$

Which is read as *give me the set of all x such that x belongs to the set of the integer numbers, x is even and x is greater than 5.*

Going formal, a set-builder notation expression is composed by three sections: a variable, a colon or vertical bar separator, and a logical predicate, which are contained in curly brackets.[1].

We may also quantify variables, either by using the existential quantifier, like this definition of the set of all natural even numbers

$$\{k \mid k \in \mathbb{N} \wedge \exists n \in \mathbb{N}, k = 2n\}$$

or the universal quantifier, negations, etc.

Sometimes, we have to decide if we want our set-builder notation to be able to quantify not only under elements that belong to a set, but also under sets. If we do this, we may encounter some situations like the one described as the Russell's Paradox

$$\{S \mid S \notin S\}$$

which is read as *give me the set of all sets S such that S does not contain themselves.*

Let's write the functions *map* and *filter* with the set-builder notation we learned. For the first one, we want the set of all the elements that are in a set S but with some transformation applied, let's say the function $f : S \longrightarrow U$. We could write our version of *map* like this(with some syntax sugar on the variable section)

$$\{fx \mid x \in S\}$$

And now we can do the same for the *filter* function

$$\{x \mid x \in S \wedge px\}$$

where x must belong to the set S and hold the predicate $p : S \longrightarrow bool$.

III. LIST COMPREHENSIONS IN HASKELL

Haskell is one of the programming languages that supports writing list comprehensions. Let's consider the example

```
[toUpper c | c <- s]
```

where s :: String is a string such as "Hello". Strings in Haskell are lists of characters; the generator $c <- s$ feeds each character of s in turn to the left-hand expression toUpper c, building a new list. The result of this list comprehension is "HELLO".[2]

And with multiple generators, we have

```
[(i,j) | i <- [1,2], j <- [1..4]]
```

yielding the result

```
[(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),
(2,3),(2,4)]
```

In Haskell, list comprehensions are translated into equivalent definitions in terms of *map*, and *concat*[3]. The translation rules are

```
[e |True] = [e]
[e | q] = [e | q, True]
[e | b, Q] = if b then [e | Q] else []
[e | p <- xs, Q] = let ok p = [e | Q]
                       ok _ = []
                   in concat (map ok xs)
```

Now, we can write the functions *map* and *filter* as a list comprehensions as we did on the previous section

```
map f xs = [f x | x <- xs]
filter p xs = [x | x <- xs, p x]
```

IV. LIST COMPREHENSIONS IN JAVA

Here is the proposed solution for list comprehensions in Java, where a new class *ListComprehension* has been added.

$$\{x \mid x \in \{1,2,3,4\} \wedge x \text{ is even}\}$$

would be now expressed in Java as

```
Predicate<Integer> even = x -> x % 2 == 0;

List<Integer> evens =
          new ListComprehension<Integer>()
  .suchThat(x -> {
     x.belongsTo(Arrays.asList(1,2,3,4));
     x.is(even);
});
```

Note that we are using lambdas here, so Java 8 is required.

One of the main goals when defining this Java API is that we want the user to read this code and read the same concepts as in the algebraic set-builder notation: *give me the set (or list comprehension) of all x such that x belongs to the list of [1,2,3,4] and x is even.*

And if you want to modify the output of the expression with some function, for example

$$\{x * 2 \mid x \in \{1,2,3,4\} \wedge x \text{ is even}\}$$

we can write it in Java as

```
List<Integer> duplicated =
          new ListComprehension<Integer>()
  .giveMeAll((Integer x) -> x * 2)
  .suchThat(x -> {
     x.belongsTo(Arrays.asList(1,2,3,4));
     x.is(even);
});
```

Now we can introduce our implementation of the functions *map* and *filter* with our new library

```
public List<T> map(List<T> list,
                   Function<T,T> f) {
        return new ListComprehension<T>()
          .giveMeAll((T t) -> f.apply(t))
          .suchThat(s -> {
             s.belongsTo(list);
          });
}

public List<T> filter(List<T> list,
                      Predicate<T> p) {
        return new ListComprehension<T>()
          .suchThat(s -> {
             s.belongsTo(list);
             s.holds(p);
          });
}
```

We have also the method *is* as an alias of *holds*.

Let's consider now the case of binding two variables, we when we do the cartesian product. In a mathematical-like syntax this would look like

$$\{(x, y) \mid x \in \{1, 2\} \land y \in \{3, 4\}\}$$

and in Java we would now write

```
List<Pair<Integer,Integer>> cartesianProd
    = new ListComprehension<Integer>()
  .suchThat((x,y) -> {
    x.belongsTo(Arrays.asList(1,2));
    y.belongsTo(Arrays.asList(3,4));
  });
```

Let's go now with something more ambitious: adding a predicate that must hold between those two variables. Like

$$\{(x, y) \mid x \in \{1, 2, 3\} \land y \in \{4, 5, 6\} \land 2 * x = y\}$$

This would be achieved in Java by writing

```
BiPredicate<Integer,Integer> condition =
  (x, y) -> x * 2 == y;

List<Pair<Integer,Integer>> doublePairs
    = new ListComprehension<Integer>()
  .suchThat((x,y) -> {
    x.belongsTo(Arrays.asList(1,2,3));
    y.belongsTo(Arrays.asList(4,5,6));
    x.holds(condition);
  });
```

Full code is available in the Apendix A and hosted at https://github.com/farolfo/list-comprehensions-in-java. Note how the code does not contain any *for*, *while* or *if*, only *maps* and *filters*.

V. DISCUSSION

We got able to implement the functions *map* and *filter* using no more than list comprehensions written in set-builder notation, Haskell and Java code, the latest one with an implementation of this article. We were also able to bind more than one variable, and define the cartesian product of two sets of integers with a relation between the variables.

Our new implementation lacks of some functionality though, such as quantifiers *exists* and *for-all* under elements, quantifiers under sets (which we have in set-builder notation), among others. Could it be possible to express quantification under sets in the Java implementation? How the Russells's Paradox case would behave?

Another thing to take into consideration is the usage of list comprehensions in production code. We have seen how Java adopted some functional programming fundamentals in its newer releases (Java 8), such as lambdas, streams that let us use *map*(called transform) and *filter*, etc; may be is not too crazy to think that Java will adopt list comprehensions at some point, as Scala, Clojure and other programming languages have done already.

[1] Wikipedia, *Set-builder notation* `https://en.wikipedia.org/wiki/Set-builder_notation`

[2] Haskell Wiki, *List Comprehension* `https://wiki.haskell.org/List_comprehension`

[3] Philip Wadler, *Comprehending Monads* University of Glasgow

Appendix A: Java implementation of List Comprehensions

```
import org.apache.commons.lang3.tuple.Pair;

import java.util.*;
import java.util.function.*;
import java.util.stream.Collectors;

@SuppressWarnings("unchecked")
public class ListComprehension<T> {

    private Function<T, T> outputExpression = x -> x;
    private BiFunction<T, T,?> pairOutputExpression = (x,y) -> Pair.of(x,y);

    ListComprehension() {
    }

    public void setOutputExpression(Function<T, T> outputExpression) {
        this.outputExpression = outputExpression;
    }

    public void setPairOutputExpression(BiFunction<T, T,?> pairOutputExpression) {
        this.pairOutputExpression = pairOutputExpression;
    }

    public List<T> suchThat(Consumer<Var> predicates){
        Var x = new Var<T>();
        predicates.accept(x);
        return (List<T>) x.value().stream().map(outputExpression).collect(
                Collectors.toList());
    }

    public List<Pair<T, T>> suchThat(BiConsumer<Var, Var> predicates){
        Var x = new Var<T>();
        Var y = new Var<T>();
        predicates.accept(x,y);
        return (List<Pair<T, T>>) x.value(y).stream()
                .map(pair -> pairOutputExpression.apply((T) ((Pair) pair)
                .getLeft(), (T) ((Pair) pair).getRight()))
                .collect(Collectors.toList());
    }

    public ListComprehension<T> giveMeAll(Function<T, T> resultTransformer) {
        this.setOutputExpression(resultTransformer);
        return this;
    }

    public ListComprehension<T> giveMeAll(BiFunction<T, T, ?> resultTransformer) {
        this.setPairOutputExpression(resultTransformer);
        return this;
    }

    public class Var<T> {
        private Set<T> belongsTo = new HashSet<>();
        private Set<Predicate<T>> predicates = new HashSet<>();
        private Set<BiPredicate<T, T>> biPredicates = new HashSet<>();

        Var() {
            this.predicates.add(x -> true);
```

```
    this.biPredicates.add((x, y) -> true);
}

public Var belongsTo(List<T> c) {
    this.belongsTo.addAll(c);
    return this;
}

public Var is(Predicate<T> p) {
    this.predicates.add(p);
    return this;
}

public Var holds(Predicate<T> p) {
    return is(p);
}

public Var is(BiPredicate<T, T> p) {
    this.biPredicates.add(p);
    return this;
}

public Var holds(BiPredicate<T, T> p) {
    return is(p);
}

public List<T> value() {
    return intersect(predicates.stream()
            .map(condition -> belongsTo.stream()
                    .filter(condition)
                    .collect(Collectors.toList()))
            .collect(Collectors.toList()));
}

private List<T> intersect(List<List<T>> lists) {
    return belongsTo.stream()
            .filter(x -> lists.stream()
                    .filter(list -> list.contains(x)).count() == lists.size())
            .collect(Collectors.toList());
}

public List<Pair<T, T>> value(Var yVar) {
    List<BiPredicate<T, T>> allBiPredicates = new LinkedList<>();
    allBiPredicates.addAll(this.biPredicates);
    allBiPredicates.addAll((Collection<? extends BiPredicate<T, T>>)
                            yVar.biPredicates.stream()
            .map(new Function<BiPredicate<T, T>,BiPredicate<T, T>>() {
                @Override
                public BiPredicate apply(BiPredicate p) {
                    return new BiPredicate<T, T>() {
                        @Override
                        public boolean test(T x, T y) {
                            return p.test(y,x);
                        }
                    };
                }
            }).collect(Collectors.toList()));
```

```
            List<Pair<T, T>> pairs = new LinkedList<>();

            this.value().stream().map(new Function<T, Boolean>() {
                @Override
                public Boolean apply(T x) {
                    yVar.value().stream().map(new Function<T, Boolean>() {
                        @Override
                        public Boolean apply(T y) {
                            return pairs.add(Pair.of(x, y));
                        }
                    }).collect(Collectors.toList());
                    return null;
                }
            }).collect(Collectors.toList());

            return pairs.stream().filter(pair -> holdsAll(allBiPredicates, pair))
                    .collect(Collectors.toList());
        }

        public boolean holdsAll(List<BiPredicate<T,T>> predicates, Pair<T,T> pair) {
            return predicates.stream().filter(p -> p.test(pair
                            .getLeft(),pair.getRight())).count() == predicates.size();
        }

        public <G> List<G> concat(List<List<G>> lists) {
            List<G> list = new LinkedList<>();
            lists.stream().map(l -> {
                list.addAll(l);
                return l;
            }).collect(Collectors.toList());
            return list;
        }
    }
}
```