Functional Programming 795 Assignment 1

Simeon Boshoff, 22546510, 22546510@sun.ac.za

September 9, 2022

1 Porting the Haskell implementation of the Numbers Game to Java

At first, I will say that the problem of the Numbers Game is much more difficult to solve with a program than it might look on the surface. For the test example, given in class, which was the number set [1,3,7,10,25,50] with a target of 765, resulting in 780 possible solutions - by hand one might be able to calculate two or three, thus programming a computer to generate all 780 proved to be rather difficult. I will divide this section into subsections, explaining every part of the Java implementation which I found of importance, as well as comparing it to the Haskell version.

1.1 Generating choices

The first step of solving the Numbers Game is to create a set of all different subsets and combinations of those subsets, as well as generating those subsets with a subset of the original set. Sounds like a mouthful? Definitely. The Haskell implementation makes use of the "Combinatorial functions" to achieve this, and can be denoted by the following steps:

1. Create subsets of the original set.
2. Create all permutations of the original set, by recursively mapping the elements of the interleave function to the list that the permutation function returns.
3. Create the list of all possible choices by mapping the list of all permutations, to the list of all subsets.

As an example:

1. Original set = [1,2,3]. Subsets = [[],[1],[2],[3],[1,2],[1,3],[1,2,3]]. As we can see, all of these subsets can be reorganized to obtain every possible combination of sets of numbers.
2. Permutations = [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]].
3. As we can see, permutations give all the different combinations of the full original set. Thus, if we create subsets with every one of those combinations, we obtain every possible combination to group these numbers. Choices = [[],[3],[2],[2,3],[3,2],[1],[1,3],[3,1],[1,2],[2,1],[1,2,3],[2,1,3], [2,3,1],[1,3,2],[3,1,2],[3,2,1]].

So, for the Java implementation, I made use of a single recursive function, instead of the 5 functions used in the Haskell implementation.

The first important part to figure out is which loops to use, to obtain the whole set of subsets of choices. The "Choices" function, (A), calls the "ChoicesRecurse" function, (B), for every possible depth from 1 to the length of the list n's, and then for every different index as a starting number of the set.

(B) then makes subsets recursively, based on the depth of the specific loop, "rope".

Already it is clear to see that the Haskell implementation is far cleaner and easier to read from a user perspective. Haskell also requires a lot less code.

```java
1 usage  Mr S Boshoff
public static void Choices(List<Integer> ns) {
    int len = ns.size();
    Choices_List = new ArrayList<>();
    Choices_List.add(new ArrayList<>());

    for (int x = 0; x < len; x++) {
        for (int y = 0; y < len; y++) {
            List<Integer> list_x = new ArrayList<>();
            ChoicesRecurse(list_x,  rope: x+1, y);
        }
    }
}
```

(A) Choices base function

```java
2 usages  Mr S Boshoff *
public static void ChoicesRecurse(List<Integer> results, int rope, int source) {

    results.add(nlist.get(source));

    if (rope == 1) {
        Choices_List.add(results);
        return;
    }
    if (nlist.isEmpty()) {
        return;
    }

    for (int x = 0; x < nlist.size(); x++) {
        if (x != source && !results.contains(nlist.get(x))) {
            List<Integer> newList = new ArrayList<>(results);
            ChoicesRecurse(newList,  rope: rope - 1, x);
        }
    }
}
```

(B) Choices recursive function

1.2 The expression class

Because Java allows for the use of complex objects, my Java implementation of the Expression can do a bit more than the Haskell version. The most important aspect of the Expr class (C), is that there are two built-in functions inherent to the class, namely "calculate_val", which can recursively calculate the integer value of the expression, and "makeString", which creates the string representation of the expression. To create an "Expr", one must either input a single integer value, or an operator (which is defined by an enum) with two other expressions. Another useful feature of this class is that every time a new expression is created, the value can be calculated and its validity checked. In section 2 I will discuss the performance benefits of Java, and I expect that this evaluation plays a big role.

```java
public static class Expr {
    // FOR A SINGLE VALUE
    int val;
    // FOR AN EXPRESSION
    Expr xpr1;
    Expr xpr2;
    Operator op;
    // FOR BOTH
    Boolean valid = true;

    public Expr(int value) { this.val = value; }

    public Expr(Expr xpr1, Operator operator, Expr xpr2) {
        this.xpr1 = xpr1;
        this.xpr2 = xpr2;
        this.op = operator;
        this.val = calculate_val(this.xpr1, this.op, this.xpr2);

        if (this.val < 0 || !xpr1.valid || !xpr2.valid) {
            valid = false;
        }
    }
    public static int calculate_val(Expr e1, Operator op, Expr e2) {...}

    public String makeString() {...}
}
```

(C) Expr class

Unfortunately, this model does consume more memory, as all this functionality is embedded in each expression. However, I do not expect this to be a problem.

1.3 Coding the Java versions of solutions' and solutions"

Solutions' and solutions" are methods in the Haskell code that both utilize some form of pruning when calculating possible answers. Solutions' has the addition of checking that the two expressions are valid when combining them. This allows for the evaluation of expressions that are not just on the

surface level and ensures that the moment at which an expression is deemed invalid, it will no longer be expanded upon.

For the Java implementation, I have incorporated the same "Valid" and "Combine" functions as in the Haskell.

```java
public static List<Expr> Combine1(Expr e1, Expr e2) {

    List<Expr> result = new ArrayList<>();

    result.add(new Expr(e1, Operator.ADD, e2));
    result.add(new Expr(e1, Operator.SUB, e2));
    result.add(new Expr(e1, Operator.MUL, e2));
    result.add(new Expr(e1, Operator.DIV, e2));

    return result;
}
```

(D) Combine

```java
public static List<Expr> Combine2(Expr e1, Expr e2) {
    List<Expr> result = new ArrayList<>();

    if (!e1.valid || !e2.valid) {
        return result;
    }

    if (Valid1(Operator.ADD, e1.val, e2.val)) {
        result.add(new Expr(e1, Operator.ADD, e2));
    }
    if (Valid1(Operator.SUB, e1.val, e2.val)) {
        result.add(new Expr(e1, Operator.SUB, e2));
    }
    if (Valid1(Operator.MUL, e1.val, e2.val)) {
        result.add(new Expr(e1, Operator.MUL, e2));
    }
    if (Valid1(Operator.DIV, e1.val, e2.val)) {
        result.add(new Expr(e1, Operator.DIV, e2));
    }

    return result;
}
```

(E) Combine'

```java
public static List<Expr> Combine3(Expr e1, Expr e2) {
    List<Expr> result = new ArrayList<>();

    if (!e1.valid || !e2.valid) {
        return result;
    }

    if (Valid2(Operator.ADD, e1.val, e2.val)) {
        result.add(new Expr(e1, Operator.ADD, e2));
    }
    if (Valid2(Operator.SUB, e1.val, e2.val)) {
        result.add(new Expr(e1, Operator.SUB, e2));
    }
    if (Valid2(Operator.MUL, e1.val, e2.val)) {
        result.add(new Expr(e1, Operator.MUL, e2));
    }
    if (Valid2(Operator.DIV, e1.val, e2.val)) {
        result.add(new Expr(e1, Operator.DIV, e2));
    }

    return result;
}
```

(F) Combine"

The main differences, which we can see in (D), (E), and (F) are stipulated as follows:

- **Combine :** When combining expressions with an operator, no evaluation is done, and every possible expression combination is returned. The expressions are then checked if valid when they have been generated fully. The class automatically marks their validity upon creation, so no other function is needed.
- **Combine' :** The biggest addition to Combine' is that expressions are checked for validity when they are constructed. An expression will only be added to the list, if and only if both expressions are valid. Valid', (G), is used to determine if it is valid.
- **Combine" :** This function is similar to Combine', as it also checks validity during the creation of expressions. However, Combine" makes use of aggressive pruning of any expressions which have commutative properties. Thus, when adding, only expressions where the first has a smaller or equal value than the other are considered. Multiplication when one of the expression's values is 1, and division where the denominator is 1, are also not considered. This is done by the Valid" function (H). Important to note, that when using this pruning, there are fewer solutions in the final solution set.

```java
public static boolean Valid1(Operator op, int x, int y) {
    switch(op) {
        case ADD:
        case MUL:
            return true;
        case SUB:
            return x > y;
        case DIV:
            return x % y == 0;
        default:
            return false;
    }
}
```

(G) Valid'

```java
public static boolean Valid2(Operator op, int x, int y) {
    switch(op) {
        case ADD:
            return x <= y;
        case SUB:
            return x > y;
        case MUL:
            return x != 1 && y != 1 && x <= y;
        case DIV:
            return y != 1 && x % y == 0;
        default:
            return false;
    }
}
```

(H) Valid"

1.4 Reflecting on Java vs Haskell code

The rest of the Java program is virtually identical to the Haskell code in its operation. Overall, I would say that the biggest difference to me would be how Haskell iterates through lists. In the Java implementation, there are a total of 11 "for"-loops, whilst in Haskell, all list iteration is done inherently. Graham Hutton explains many times that Haskell code should be clear, concise and correct. Through working with the Haskell implementation of the Numbers Game, it is easy to understand when the

program is doing, once you understand Haskell syntax. The way in which everything is implemented is clear, and the whole program was written in less than half the number of lines used in my Java implementation. As an example, coding the Split function, which takes in a list and creates different pairs of elements within that list, in Java, proved to be challenging due to the large number of loops required.

```java
public static List<List<List<Integer>>> Split(List<Integer> ns) {
    List<List<List<Integer>>> A = new ArrayList<>();

    for (int x = 0; x < ns.size() - 1; x++) {
        List<List<Integer>> B = new ArrayList<>();

        List<Integer> C1 = new ArrayList<>();
        List<Integer> C2 = new ArrayList<>();

        int k = x + 1;

        for (int z = 0; z < k; z++) {
            C1.add(ns.get(z));
        }
        for (int z = k; z < ns.size(); z++) {
            C2.add(ns.get(z));
        }
        B.add(C1);
        B.add(C2);
        A.add(B);
    }
    return A;
}
```

(I) Java "Split"

```haskell
split :: [a] -> [([a],[a])]
split []     = []
split [_]    = []
split (x:xs) = ([x],xs) : [(x:ls,rs) | (ls,rs) <- split xs]
```

(J) Haskell "Split

When comparing (I) and (J), we can see that both return a list of lists of lists of integers. However, Haskell can easily denote this by using only a few brackets and parentheses. 22 lines required versus 4 - and the Haskell is much easier to understand. This was shown as a clear example to illustrate why a functional language like Haskell is much more efficient to write a program such as Numbers Game in, since it is so much easier to write the syntax of such a complex function. The Haskell allows for more intuitive code, as the Java needs to be written in the exact way that Java requires so that the computer "understands" what you are trying to do. I also believe that one would be at a smaller risk for making dumb programming errors such as not iterating through the whole list or something like that, as Haskell code "is what it is". Another thing that one could note would be that for the Haskell split function, different cases for input had to be explicitly coded, however, in the Java, certain operations just do not occur if the conditions are not met.

To conclude, in my humble opinion, Haskell is a much better programming language to code the Numbers Problem in, given that the programmer understands Haskell syntax well. I would say that Java is a lot more beginner friendly since a lot of the Haskell syntax isn't easily understood. However, once you know what you are doing, you can write powerful programs in a small number of lines.

2 Benchmarking Java vs Haskell implementations

Benchmarking was done by manually running the Haskell and Java applications with different input. Wall clock time was used, and the appropriate modifications to the Haskell code have been made. Important note, before results are discussed, is that IO introduces a significant amount of processing time. Both programs have been coded to output a solution the moment it is generated, to ensure that the results are fair.

Importantly, the time that it takes to generate the first solution has been negated, due to the speed at which Haskell can find the first solution (in the order of microseconds), whilst Java averages out to about 25 milliseconds before the first solution is printed.

The input used to test both programs is as follows:

Difficulty	SET	TARGET
1	[1,2,3]	5
2	[1,2,3,4]	5
3	[1,2,3,6,10]	5
4	[1,3,7,10,25,50]	765
5	[1,2,3,4,30,42]	362
6	[1,3,5,7,10,25,50]	732
7	[1,2,3,4,5,6,7]	22

Figure 1. Input

2.1 Haskell benchmark

The results obtained by benchmarking the Haskell implementation are shown in Figure 2. As can be observed, there are no values at difficulty levels 5 and 6 for the normal Haskell implementation, as the program did not find all solutions in a reasonable time (10 minutes+). It is clear from the figure that there is a significant speedup for every added method of pruning, with solutions" being the most optimal. The y-scale is logarithmic, so a small difference in the graph means a significant difference in reality.

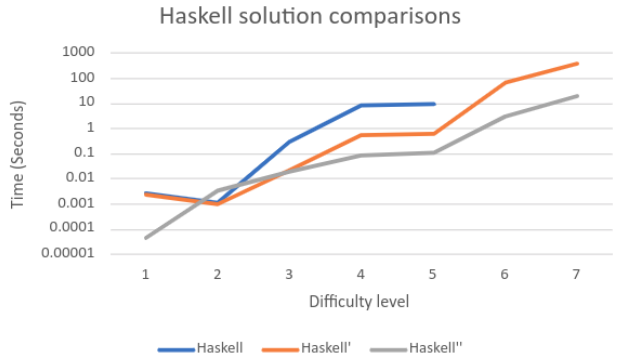

Figure 2. Haskell benchmark

2.2 Java benchmark

The Java benchmark shows the same results as the Haskell implementation, however, there is a smaller discrepancy between the normal and the prime methods. The Java PrimePrime implementation beats all others.

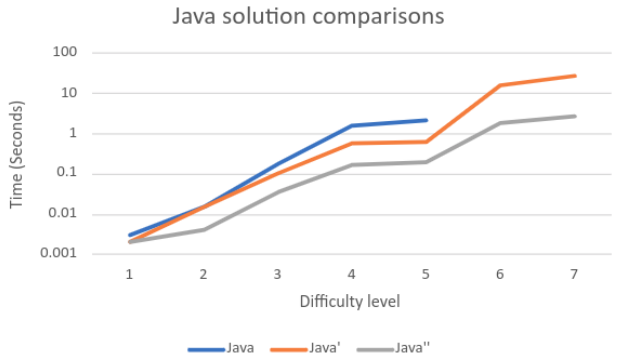

Figure 3. Java benchmark

2.3 Java vs Haskell benchmark

To compare the benchmarking results of both the Haskell and the Java, the individual results have been plotted for each method of "solutions". An interesting pattern emerges, which can be observed in Figure 4 and Figure 5, as we can see that for the lower difficulty levels the Haskell is usually the fastest. This can be attributed to the fact that data structures and other variables are initialized in the Java, which causes slow-downs. For higher difficulty problems, the Java performs faster, by some margin. There was an instance where the Java program found all solutions in 16 seconds, and the Haskell just kept running for minutes on end.

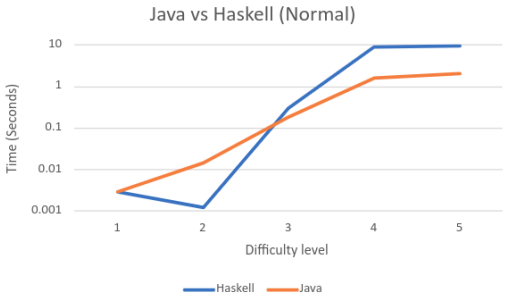

Figure 4. Normal

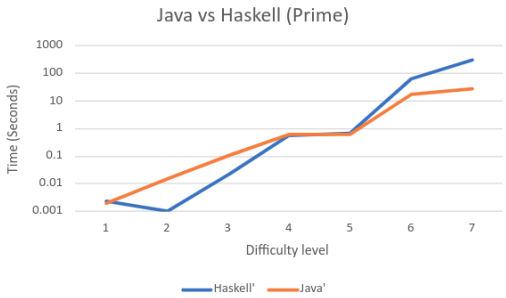

Figure 5. Prime

Figure 6 shows the two implementations more evenly matched, as some computational roadblocks are solved by not attempting to print as many solutions. The Haskell implementation, however, takes significantly longer when calculating the most difficult problem.

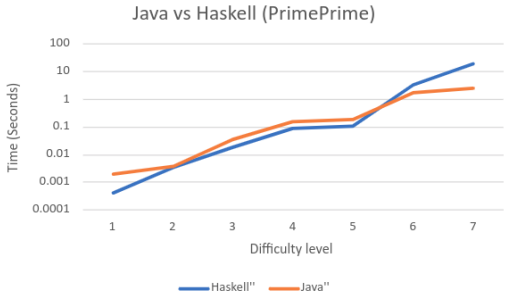

Figure 6. PrimePrime

To conclude the benchmarking, after results have been compiled, it is clear to see that Haskell is faster when calculating smaller problems, but slower for larger problems. I attribute this to the fact that the Haskell implementation does not store the value of the expressions. As a result, the value of expressions is re-calculated a significant number of times. Figure 7 shows the difference in the amount of RAM used when both programs are executing at the same time. The Java implementation thus performs more efficiently when the problems are large.

Process Name	User	% CPU	ID	Memory
java	22546510@su	38	5315	3.5 GiB
numbers-game-exe	22546510@su	32	4976	287.7 MiB

Figure 7. System resources comparison

3 Adding functionality

3.1 Returning the number of solutions

To return the number of solutions found by the program, the code in (K) was written. I do not consider it the best way to find the number of solutions, however, it works. I used it in unit testing to ensure

that the correct number of solutions have been found for given problems.

```haskell
n_solutions :: [Int] -> Int -> Int
n_solutions ns t = length (solutions ns t)

n_solutions' :: [Int] -> Int -> Int
n_solutions' ns t = length (solutions' ns t)

n_solutions'' :: [Int] -> Int -> Int
n_solutions'' ns t = length (solutions' ns t)
```

(K) Functions calculating number of solutions

3.2 Adding Numbers Game type

A Numbers Game is defined as a method in which a set of numbers, referred to as a "block", and a target value is used to generate combinations of equations that all evaluate to the target value.

```haskell
-- Type to describe instance
data Game = Game {
    target::Int,
    blocks::[Int]
} deriving Show
```

(L) Game type

3.3 Implementing the Arbitrary type class

For random input to be generated to test the program, the nature of that input needs to be defined and rules need to be set up to ensure that the testing input follows the procedures of the game. In (K), I have imported Arbitrary from QuickCheck, and the function "choose", which I assume randomly selects a number from a given range.

```haskell
import Test.QuickCheck.Arbitrary
import Test.QuickCheck (choose)
```

(K) Importing Arbitrary

To create an Arbitrary instance, we have to define the above-mentioned rules for which values can be generated. As we have defined our game type, we require:

- **Target :** integer value between 1 and 999.
- **Low number set :** a list containing between 1 and 4 values between 1 and 10.
- **High number set :** a list containing between 1 and 2 values between 11 and 99.

The low number and high number sets are then combined to obtain a set which is used as input for the solutions function. (M) shows the codes of this operation.

```haskell
instance Arbitrary Game where arbitrary = genGame

genGame = do {target <- choose(1,999);
    ll <- sequence [choose(1,10) | _ <- [1..4]];
    rl <- sequence [choose(11,99) | _ <- [1..2]];
    return (Game target(ll++rl)) }
```

(M) Arbitrary instance

3.4 The comparison of expressions

As we know, the expression type in Haskell consists of either an integer value or two other expressions with an operator. The following code, shown in (N), illustrates how the Ord class was implemented to allow the program to order expressions according to length, and maximum values inside the expression.

- **Length of expressions :** To calculate the length of an expression, the left and right parts of such an expression are recursively evaluated to find the number of expressions. When the expression does not contain any other expressions, simply a 1 is returned.
- **Maximum of expressions :** To calculate the maximum element of an expression, the expression is once again split into two parts, which are searched recursively. This returns the largest value found in an expression.
- **Comparing expressions :** To compare two expressions, first the lengths of the expressions are obtained and then compared to each other, after which the appropriate ordering is returned. If the lengths are the same, then the maximum values inside the expressions are calculated, and the same procedure is followed as above. If the maximum values are also equal, then the "EQ" operator is returned.

When determining the order of expressions, the "compareExpr" function is called directly, and the output operators are used.

```haskell
--1. Length of Expr

lenExpr :: Expr -> Int
lenExpr (Val n) = 1
lenExpr (App _ l r) = lenExpr l + lenExpr r

--2. Max of Expr

maxExpr :: Expr -> Int
maxExpr (Val n) = n
maxExpr (App _ l r) | (maxExpr l > maxExpr r) = maxExpr l
                    | (maxExpr l <= maxExpr r) = maxExpr r


--3. Compare Exprs. Return True if e1 <= e2, else return False

compareExpr :: Expr -> Expr -> Ordering
compareExpr e1 e2 | lenExpr(e1) < lenExpr(e2) = LT
                  | lenExpr(e1) > lenExpr(e2) = GT
                  | maxExpr(e1) < maxExpr(e2) = LT
                  | maxExpr(e1) > maxExpr(e2) = GT
                  | otherwise = EQ

instance Ord Expr where
   compare e1 e2 = (compareExpr e1 e2)
```

(N) Ord class

4 Testing

Question 4 of the assignment has unfortunately not been completed. However, (O) shows the implementation of testing each solution method in the Haskell program.

```haskell
main :: IO ()
main = do   putStrLn ("Testing solutions [1,3,7,10,25,50] 765 == 780")
            putStrLn (show ((n_solutions [1,3,7,10,25,50] 765) == 780))
            putStrLn ("Testing solutions' [1,3,7,10,25,50] 765 == 780")
            putStrLn (show ((n_solutions' [1,3,7,10,25,50] 765) == 780))
            putStrLn ("Testing solutions'' [1,3,7,10,25,50] 765 == 780")
            putStrLn (show ((n_solutions'' [1,3,7,10,25,50] 765) == 49))

            putStrLn ("Testing solutions [1,2,3,4] 5 == 226")
            putStrLn (show ((n_solutions [1,2,3,4] 5) == 226))
            putStrLn ("Testing solutions' [1,2,3,4] 5 == 226")
            putStrLn (show ((n_solutions' [1,2,3,4] 5) == 226))
            putStrLn ("Testing solutions'' [1,2,3,4] 5 == 226")
            putStrLn (show ((n_solutions'' [1,2,3,4] 5) == 20))



n_solutions :: [Int] -> Int -> Int
n_solutions ns t = length (solutions ns t)

n_solutions' :: [Int] -> Int -> Int
n_solutions' ns t = length (solutions' ns t)

n_solutions'' :: [Int] -> Int -> Int
n_solutions'' ns t = length (solutions'' ns t)
```

(O) Spec.hs implementation

References

[1] Brink van der Merwe's classes and course slides. Stellenbosch University 2022.

[2] Graham Hutton's Youtube Haskell course. Available [https://www.youtube.com/playlist?list=PLF1Z-APd9zK7usPMx3LGMZEHrECUGodd3]

[3] Various classmates who have helped to explain certain parts of the assignment.