Algorithm Analysis

3EA3 Software Specification and Correctness: Final Report

Navleen Singh - 1302228

Musa Al-hassy

Table of Contents

Introduction 3

Power 4

Approach 4

Final Thoughts 5

Tree Traversal 6

Approach 6

Final Thoughts 7

Merge Sort 8

Approach 8

Final Thoughts 10

Conclusion 11

References 12

Introduction

In this report, the concentration will be on 3 different algorithms and their theoretical proofs in Isabelle - Hol. The three theories we will be discussing are Power, Tree Traversal, Merge Sort. We will start with a quick overview with about the theory, and then how we went about proving the algorithm.

The idea behind these proofs Merge Sort and Tree Traversal was from interview preparations, which will show you how and why it is implemented in a certain way. The proof for how powering a natural number with another natural number works, can be used to show how power full Isabelle Hol.

I wanted to show users with different experience levels on what they can do with Isabelle Hol. For the entire project, you can visit the Github repository. The repository contains the source code, and will be updated in the future. If you would like to commit and add more information you may by creating a pull request.

Power

We will work on proving that $x^{m*n} = (x^m)^n$, all the number will be natural. This is a simple proof that can be used to show other properties of mathematics.

```
theorem pow_mult: "pow x (m * n) = pow (pow x m) n"
```

Approach

We know that $x^n = x * x * x * x * ... * x$. Let's start by defining a recursive property which will multiply x, n many times.

```
primrec pow :: "nat => nat => nat"
  where
    "pow x 0       = Suc 0"
  | "pow x (Suc n) = x * pow x n"
```

This will be the property we will use to do perform x^n. Once we start writing our theorem we want to prove, we will notice that we need to first prove a base case of $x^{m+n} = x^m + x^n$.

```
lemma pow_add: "pow x (m + n) = pow x m * pow x n"
```

We can start by applying the built-in induction on our value n, and then use auto to complete the proof.

```
lemma pow_add: "pow x (m + n) = pow x m * pow x n"
  apply (induct n)
  apply auto
done
```

Now if we go back to our proof for pow_mult we can also start by applying the built-in induction on n. Then we will use apply auto and use the pow_add base case we created.

```
theorem pow_mult: "pow x (m * n) = pow (pow x m) n"
  apply (induct n)
  apply (auto simp add: pow_add)
done
```

Final Thoughts

As you can see the proof for a simple property for mathematics can be show very quickly using Isabelle's built-in methods. This helps show why the property of power is the way it is. We can use the same approach to prove factorial, addition, subtraction, etc. properties.

Tree Traversal

We will work on proving 3 different way on printing a tree but we will add a twist. First, we will mirror a tree and show that it is the same as printing a reversed list.

```
theorem  "preOrder (mirror t) = rev (postOrder t)"

theorem "postOrder (mirror t) = rev (preOrder t)"

theorem "inOrder (mirror t) = rev (inOrder t)"
```

Approach

Since we are working with Trees we need to define the tree datatype. This will be used to define how we will print using preOrder, postOrder, inOrder properties.

```
datatype 'a tree =
  Tip "'a"
  | Node "'a" "'a tree" "'a tree"
```

Now that we have the datatype we can now show how to print the values in a tree with the following method (preOrder, postOrder, inOrder). These properties will use recursion on the tree.

```
primrec preOrder :: "'a tree ⇒ 'a list"
  where
    "preOrder (Tip a)      = [a]"
  | "preOrder (Node f x y) = f#((preOrder x)@(preOrder y))"
```

```
primrec postOrder :: "'a tree ⇒ 'a list"
  where
    "postOrder (Tip a)      = [a]"
  | "postOrder (Node f x y) = (postOrder x)@(postOrder y)@[f]"

primrec inOrder :: "'a tree ⇒ 'a list"
  where
    "inOrder (Tip a)      = [a]"
  | "inOrder (Node f x y) = (inOrder x)@[f]@(inOrder y)"
```

The final property that we will define is mirror. This property can be used to help show many other things, such as a revered list is the same as a mirrored tree. The thermos we will show is that printing a mirrored tree is the same as printing a reversed tree list.

```
primrec mirror :: "'a tree ⇒ 'a tree"
  where
    "mirror (Tip a)      = (Tip a)"
  | "mirror (Node f x y) = (Node f (mirror y) (mirror x))"
```

We will use the same technique for all three proofs. This is another built-in technique to Isabelle – Hol which is induction tactic.

```
apply (induct_tac t)
apply auto
```

Final Thoughts

As you can see the proof for a tree can be help full for showing how they function. There are many properties can be confusing for the programmer to visualize, so by showing a proof they are able to easily see how it functions.

Merge Sort

We will work on proving the correctness of Merge Sort and if it actually sorts a list of elements. This one the more advanced proofs/implementations in Isabelle – Hol, but this is what most companies will use. This approach can be used to prove the correctness of other algorithms.

```
theorem "sorted(sort xs)"
```

Approach

We can find the pseudocode of merge sort online but we will need to define them as functions in Isabelle. We can just use natural numbers. The first function will take in two lists and return a merged list. Second function will take a list and return the sorted list. If we think about it the second function should call the first function to merge the two sorted lists.

```
fun sort :: "nat list ⇒ nat list"
  where
    "sort [] = []"
  | "sort [x] = [x]"
  | "sort xs = (
                let mid = length xs div 2 in
                merge (sort (take mid xs)) (sort (drop mid xs))
              )"
```

```
fun merge :: "nat list ⇒ nat list ⇒ nat list"
  where
    "merge [] ys = ys"
  | "merge xs [] = xs"
  | "merge (x # xs) (y # ys) = (
                                  if x ≤ y then
                                    x # merge xs (y # ys)
                                  else
                                    y # merge (x # xs) ys
                                )"
```

Since our theorem checks if the list is sorted, we have to define two properties which help use check if a list is sorted. The properties will return True or False.

```
primrec less :: "nat ⇒ nat list ⇒ bool"
  where
    "less value []     = True"
  | "less value (x#xs) = (value <= x & less value xs)"
```

The above property checks a value against all the other elements in the list and if it is less than the others. Now we can use this in a recursive statement which goes through all the elements and checks if it is sorted.

```
primrec sorted :: "nat list ⇒ bool"
  where
    "sorted []     = True"
  | "sorted (x#xs) = (less x xs & sorted xs)"
```

We realize that we need to prove the base cases before we can prove our theorem and show that merge sort is correct. We will insure that our x value is less than or equal to our y value, we will use the induction tactic on our list xs.

```
lemma compare_values [simp]: "x ≤ y ⟹ less y xs ⟶ less x xs"
  apply (induct_tac xs)
  apply auto
done
```

Now we will insure that the x value is less than all the values in the two lists respectively and if they are merged that value is still the least.

```
lemma less_merge [simp]: "less x (merge xs ys) = (less x xs ∧ less x ys)"
  apply (induct xs ys rule: merge.induct)
  apply auto
done
```

Our final case we will make sure that the merged list is sorted and its equal to the two lists sorted separately. We use some of the predefined proofs in the Main file we include at the very top.

```
lemma sorted_merge [simp]: "sorted (merge xs ys) = (sorted xs ∧ sorted ys)"
  apply(induct xs ys rule: merge.induct)
  apply (auto simp add: linorder_not_le order_less_le)
done
```

Now we can tell Isabell to apply the induction tactic on the list xs.

```
theorem "sorted(sort xs)"
  apply (induct_tac xs rule: sort.induct)
  apply auto
done
```

Final Thoughts

As you can see we showed that the functional and logical approach to merge sort does actually sort the list. In the same way, we can prove other algorithms and their correctness. This is the most help full for companies, so that they know the new algorithm that they developed will work for all cases.

Conclusion

Overall, I found that the project helped me show properties of math and functionalities of programs, that I would have never realized before. The different approaches to proving different properties, give Isabelle a strong standing in the proof languages. It is very help and forgiving with the built-in proof helpers.

References

http://spark.woaf.net/isabelle_cheat_sheet.html

http://www.cs.ru.nl/~freek/100/

https://isabelle.in.tum.de/documentation.html

https://en.wikipedia.org/wiki/Merge_sort

https://www.geeksforgeeks.org/merge-sort/

http://isabelle.in.tum.de/coursematerial/PSV2009-1/