Algorithm Analysis

3EA3 Software Specification and Correctness: Final Report

Navleen Singh - 1302228

Musa Al-hassy

Table of Contents

Introduction

This report serves the purpose of analyzing 3 different algorithms and their theoretical proofs in Isabelle - Hol. The three theories being discussed are Power, Tree Traversal, and Merge Sort. We begin with a quick overview about the theory that serves as basis for these algorithms, and then investigate how one goes about proving each algorithm.

The idea behind the proofs for Merge Sort and Tree Traversal came from interview preparations, which will make clear how and why it is implemented in a particular way. The proof for how powering a natural number with another natural number works can be used to show how powerful Isabelle – Hol is.

Users of any skill level can accomplish many tasks using Isabelle - Hol. The entire project is available in a 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 do so by creating a pull request.

Power

We begin by proving that $x^{m*n} = (x^m)^n$, with all natural numbers. 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 is the property we will use to perform x^n. Once we start writing the 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 functionality on our value n, and then use auto (another feature of Isabelle - Hol) to complete the proof.

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

Going back to our proof for pow_mult, we can similarly 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

It is evident that, using Isabelle – Hol, the proof for a simple, mathematical property can be constructed 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.

Tree Traversal

We move on to proving 3 different ways of printing a tree, but with an added 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 to print using the 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 using 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 define is mirror. This is a useful property which can be helpful in proving many other properties, such as the issue at hand: A reversed list is equivalent to a mirrored tree. The theorem is attempting to prove 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 of Isabelle – Hol, named induction tactic.

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

Final Thoughts

It is evident that the proof for a tree can be helpful for understanding the recursive definition and their overall functionality. There are many properties that can be confusing for the programmer to visualize, but in understanding the proof they are able to easily understand how it functions.

Merge Sort

We will work on proving the correctness of Merge Sort and proving whether or not it actually sorts a list of elements. This is one of the more advanced proofs/implementations in Isabelle – Hol, but is also more valuable in industry because of the widespread use of the algorithm itself. The following 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 will need to define functions in Isabelle. We can use natural numbers to simplify the proof. The first function will take in two lists and return a merged list. The second function will take a list and return the sorted list. It follows that 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 must verify whether or not the list is sorted, we have to define two properties to accomplish this. The properties must return a Boolean, as true and false are the only valid values.

```
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 determines 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)"
```

One can now see that we need to prove the base cases before we can prove our theorem and verify that merge sort is correct. We will ensure that our x value is less than or equal to our y value, and, to do this, 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 ensure that the x value is less than all the values in the two lists, one after the other. 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 involves making sure that the merged list is sorted and is 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 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

It is now evident that the functional and logical approach to merge sort does in fact sort the list. In the same way, we can prove other algorithms and their correctness. This skill holds an extremely high value in industry, because it allows developers to verify that the new algorithm that they created will work as expected for all cases.

Conclusion

Overall, this project has helped me learn how to show properties of math and prove expected functionalities of programs hand in hand. The different approaches to proving a multitude of properties gives Isabelle a strong standing in the set of proof languages. It is very helpful and forgiving, providing users with a myriad of built-in proof helpers.

References

(n.d.). Retrieved from http://www.cs.ru.nl/~freek/100/

(n.d.). Retrieved from http://isabelle.in.tum.de/coursematerial/PSV2009-1/

Cheat Sheet for the Isabelle/HOL Theorem Prover in ProofGeneral. (n.d.). Retrieved from http://spark.woaf.net/isabelle_cheat_sheet.html

Merge sort. (2018, April 15). Retrieved from https://en.wikipedia.org/wiki/Merge_sort

Merge Sort. (2018, March 08). Retrieved from https://www.geeksforgeeks.org/merge-sort/

Nipkow, T., Paulson, L., Wenzel, M., Klein, G., Haftmann, F., Weber, T., & Hölzl, J. (n.d.). Retrieved from https://isabelle.in.tum.de/documentation.html