AP - Assignment 5

Nicolas Gram Dyhrman — lzg210
Mikkel Bilgrav Madsen — tdg783

October 14, 2022

Design and implementation choices

Let it be

Our generator is written s.t. it divides the test cases into 4 different expression types, e.g. oneOf with cases "Oper plus", "Oper Minus", "Oper Times" , "Let" and "Var". This is done s.t. each case of the expression type will be covered by approximately 20% of a given test run.
We have made two properties for the QuickCheck test suite, prop_eval_simplify, which checks that the evaluation of the simplified expr and the unsimplified expr is the same. (e.g. legal simplification), and a prop_simplify_length, which checks that a simplified expr A is always shorter than or equal to a non simplified expression A.
We did not run stastistics on our Haskell QuickCheck Generator due to time constraints. In combination with this, we made some extensions to the simplify function to optimize it even further. Most of these optimizations to the simplifications have to do with subtracting or adding zero, simply returning the other expression, multiplying by 1 returns the expressions 1 is multiplied with, and multiplying with zero always returns 0. In other words from basic mathematics, we know that

$$a * 1 = a, \quad a * 0 = 0, \quad a + 0 = a, \quad a - 0 = a, \quad a * (-1) = (-a)$$

Now some of these hold for all expressions, while some for constants, and we implemented this through pattern matching on the "$Oper\ \_\ \_\ \_$" structure. Another part of the optimization is let bindings, where if a variable being bound is not used further in the expression, we simply evaluate without the binding. the searchForVarInTree function is defined in the file and searches the expression tree for that particular var, and if it finds it, continues, and if it does not, executes the body without the binding (as it will not change the result).

Mystery

The mystery part of the assignment revolves around building a QuickCheck testing suite, and running it against eight implementations in a blackbox manner, to locate and diagnose bugs in said implementations. We have chosen to use eqc:symbolic when creating.
To check the quality of our bst generator (e.g bst), we checked it with the prop_measure function, which uses the collection function to gather all the lengths of the different trees generated in a test run, and displays their probability distribution. We have chosen to add a few more atomic units to the atom_key() generator, and would one want more diverse testing, one could use larger and larger sets of atom_keys, and max length of bst trees (as defined in the bst generator) and hence get larger and more diverse trees for testing.
We believe our distribution to be sufficient for the task at hand, and the results can be replicated and are seen when running the tests (at the start of the testing suite).
We also chose not to implement the shrinking, and have not tried to do so, as this is mostly used to create better-shrinking algorithms for the counterexamples. But since it was able to shrink our test to a level where they were easy to understand we did not find the need to do so.

Assessment

Completeness

Let it be

We have met every requirement given in the assignment text, and added cases for simplify which simplifies values 0 and 1 for certain mathematical operations and to simplify let expressions.

Mystery

In terms of the requirements given in the assignment text, we have implemented everything. It is to some degree impossible to say that we have completely exhausted possible properties since that is impossible. Still, we did find bugs in every implementation we ran our QuickCheck on, with a wide array of properties.

Correctness

Let it be

For our implementation, we could have been using some more properties, to check that our implementation is correct, also our helper function could have used more unit tests to ensure that it works.

Mystery

We have found at least one bug present in each of the implementations, and have with a high degree of accuracy (we believe) narrowed the bugs down to specific functions. We implemented every requirement stated in the assignment text and then handed out a working example of bst does not fail any test, and neither does onlineTA. The symbolic calls used could be better since we are still using the same method, for our anonymous function as just an argument.
We also found that the model property tests revealed the most errors of the different types, and were often also good at explaining what went wrong in themselves. Although after implementing the symbolic calls the metamorphic calls that suddenly persisted and could tell us a lot about the trees became very helpful too. But we found that the connection between the counter-examples was the best for diagnosing a bug.

Efficiency

Let it be

Our efficiency of the function simplify is quite fine as we don't seem to do many extra tasks to complete this. The only thing we do that would slow down the function would be the *searchForVarInTree* where we have to look through big parts of the tree to see if a variable is present.

Mystery

Efficiency does not (to us) seem to be something we should focus on in this part of the assignment, but we have no reason to believe that our test should be ineffective unless we should have used ?ONCEONLY.

Robustness

Let it be

Since the function we are testing is solely the simplify function, and error-handling on this specific implementation seems out of the scope of the assignment, we do not have any checks in order to catch exceptions. Our generator generates specific terms, which we know to be of a specific type, hence error handling seems redundant.

Mystery

Since our task is creating a test suite, the actual error handling of the implementations we are testing against is not done by us. We do however implement a ?WHENFAIL clause in many of our properties, s.t. we can print specific information about the failing test case to the console, s.t. we can get a better overview of what exactly may be the cause of the bug.
A stub from a test is show below of how the ?WHENFAIL has been used:

```
?FORALL({K1, K2, V1, V2, T},
        {atom_key(), atom_key(), int_value(), int_value(),
         bst(atom_key(), int_value())},
  ?WHENFAIL(
    io:fwrite('atom1: ~w\nvalue1: ~w\natom2: ~w\nvalue2: ~w\nbst: ~w\n',
                     [ K1, V1, K2, V2, getInfoTreeStruckture(T)]),
```

where the getInfoTreeStruckture function is an auxiliary function to gain some of the inserts from the symbolic call.
Now what happens when a test fails, is that we print information we find necessary (K1,V1,K2,V2 and T in this case), s.t. when we are running the tests, we are able to get a clear view of what exactly went wrong with the implementation.

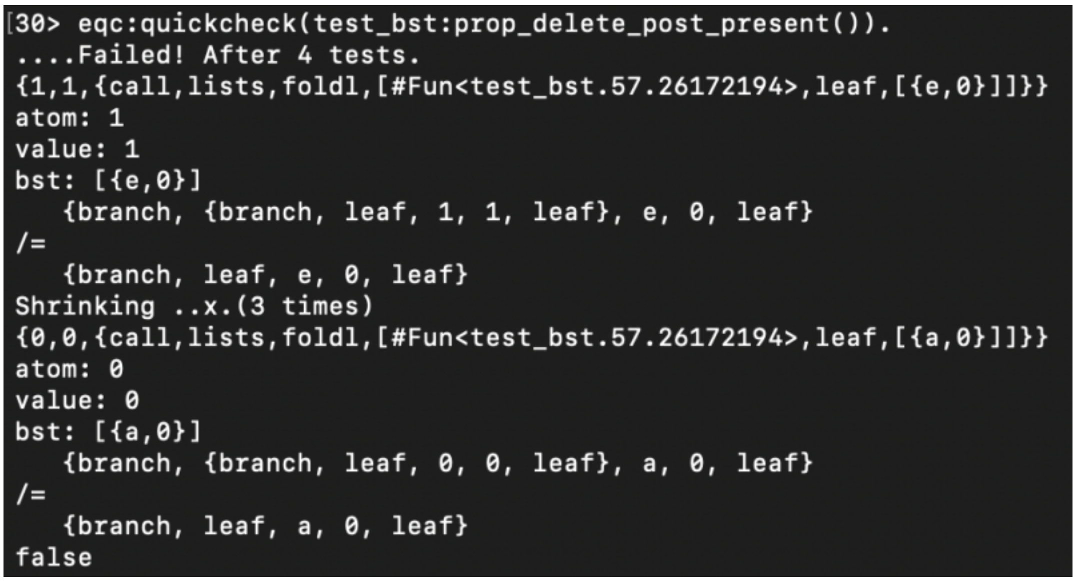
Maintainability

Our test suites in both *Let it be* and *Mystery* are very modular in nature as they are built upon the QuickCheck modules. One can simply add a property in the respective modules

1 Found Bugs in Mystery

Below is a list of the bugs we have located, divided by name of the implementation, with our best assessment of what most likely caused the bug. This is not to say that we know for a fact what is causing the bug, as we only have evidence of what *may* be the cause.

We do however specify in which functions the bug is present. First, we will show how to find a bug using the counterexample from the console:



```
[30> eqc:quickcheck(test_bst:prop_delete_post_present()).
....Failed! After 4 tests.
{1,1,{call,lists,foldl,[#Fun<test_bst.57.26172194>,leaf,[{e,0}]]}}
atom: 1
value: 1
bst: [{e,0}]
   {branch, {branch, leaf, 1, 1, leaf}, e, 0, leaf}
/=
   {branch, leaf, e, 0, leaf}
Shrinking ..x.(3 times)
{0,0,{call,lists,foldl,[#Fun<test_bst.57.26172194>,leaf,[{a,0}]]}}
atom: 0
value: 0
bst: [{a,0}]
   {branch, {branch, leaf, 0, 0, leaf}, a, 0, leaf}
/=
   {branch, leaf, a, 0, leaf}
false
```

Figure 1: Robinson failing on prop_delete_post_present()

Now in figure one we have Robinson failing a test on the prop_delete_post_present() property. What happens in the property is that we insert an element in a tree, deletes that element and then checks that the tree is the same as the original. The RHS of the property is the original tree returned, and the LHS is the insert/delete called on the orignal tree. What we can see is that Robinson does seem to insert the correct element, but that he fails to delete it again. This combined with other counter examples of delete, led us to conclucde that Robinson's delete function does in fact not delete anything, and hence is malfunctioning.
We also found that the symbolic calls really do help in creating the output so we can easily get the input function to create the tree, and easily debug the test when it fails. Below is a table of the bugs found in each implementation.

1. **noether**: Noether has an issue when calling insert, in that if the key he is trying to update already exists, he does not update the value of the given key, but instead keeps it's old value.
2. **Perlman**: When calling Union, Perlman is overwriting the value of duplicate keys if duplicate keys are found, where Perlman should keep the originale value instead.
3. **Poitras**: Has a bug with union, as when he unionizes two trees, they do not adhere to the standard of binary search trees, as he puts a lower key to the right where it should be to the left. secondly when unionizing he does not check if two trees has the same keys, thereby creating invalid binary search trees.
4. **Rhodes**: (almost same bug as Poitras) Has a bug with union, as when he unionizes two trees, they do not adhere to the standard of binary search trees, as he puts a lower key to right where it should be to the right. It seems that Rhodes is just filling in the trees in a single direction as if it was a list. Secondly when unionizing he does not check if two trees has the same keys, thereby creating invalid binary search trees.
5. **Robinson**: Has issues with his delete function. It does not seem to delete any elements at all.
6. **Snyder**: Snyder has issues regarding delete/2, as Snyder will delete the entire tree, and not just the single key that should be deleted by delete.
7. **Wilson**: has an issue regarding not checking for duplicate-keys when inserting into a bst, thereby making multiple nodes possess the same key.
8. **Wing**: has problems with insert, it seems that the implementation overwrites previous inserted data, instead of actually inserting into the tree.

Figure 2: A table of the bugs found with counterexamples in QuickCheck