Advanced Programming Assignment 5: QuickCheck Report

AP22 Assignment 5 Group 60
KRW521, CPJ395

October 14, 2022

1 Design and Implementation

1.1 Let It Be

For this part of the assignment, we first should make **Expr** become an instance of the typeclass **Arbitrary** so that the **Expr** can be generated in a property that uses **Expr** while **quickCheck** the property. To avoid generating a huge **Expr** tree, we decide to use $exprN :: Int_{>}GenExpr$ (as mentioned in the lecture as well as the slide) in the generator in our code.
To find the bug in function **simplify**, we write a property in **ExprProperties.hs** to show that the evaluate output of an **Expr** should be equal to the evaluate output of the simplified **Expr**:

```
        prop_eval_simplify :: Expr _> Property
        prop_eval_simplify x =classify (ExprEval.findVarUsed "" x) "have an empty string as identifier" (E.evalTop x === E.evalTop (E.simplify x))

```

and here we use the function **classify** and the helper function **findVarUsed** to find out if there is an empty string as an identifier in the **Expr** generated so that the output of this property run in **test.hs** can be attached the statistics data about the strings that are used as **Ident** in the **Expr** generated. (So we can check the quality of the generator by the statistics data attached in the output of the property run in **test.hs**, and we find out that no more than 5% of the **Expr** generated have an empty string as identifier which is a good quality of the generator)
After **quickCheck** the property, we find out that the **Minus** part of **simplify** has a bug, that is,

```
        Oper Minus (Const c1) (Const c2) _> Const(c1+c2)
```

It uses plus operator in **Minus**, so we fix the bug and correct the code as

```
        Oper Minus (Const c1) (Const c2) _> Const(c1_c2)
```

As for extending the function **simplify**, we add the pattern matching "An **Expr** plus 0 can be simplified to this **Expr**", "An **Expr** minus 0 can be simplified to this **Expr**", "An **Expr** times 0 can be simplified to 0", "An **Expr** times 1 can be simplified to this **Expr**" and "Simplifying the **Let**

name exp bodyExp to **bodyExp** if the bound variable is not used in the **bodyExp**" in the function **simplify**. Because the process of simplify will eliminating some **Expr** and that will cause the evaluate output of an **Expr** do not equal to the evaluate output of the simplified **Expr** in some cases. Concretely, if there is some unbinding error happening in the **Expr** that will be eliminated because of simplifying, the evaluate output of an **Expr** will become an **Left** error while the evaluate output of the simplified **Expr** is another **Left** error or **Right** answer (and that is because function **simplify** eliminates some error **Expr** while simplifying). So we decide to do something in the generator making it cannot generate the **Var** expression in the **Expr** that might be eliminated. So we write another function **exprN'** which is similar to function **exprN**, but it is without the ability to generate **Var** expression, and we call **exprN'** in the generator when it generates the **Expr** that might be eliminated because of simplifying. For example:

```
do
         i <_ fmap ExprAst.Var (arbitrary::Gen String)
         x <_ exprN' (n `div` 2) __stop putting Var in expression x
         y <_ exprN (n `div` 2);
         let id = getIdent i in
            return (ExprAst.Let id x y)
```

For this part of assignment, we only have two helper functions: **getIdent** in **ExprProperties.hs** to get the **Ident** in **Var Ident** and **findVarUsed** in **ExprEval.hs** to find if the input string has been used in a **Var Ident** in an **Expr**.

1.2 A QuickCheck Mystery

Mystery is mainly used to test the binomial search tree (BST) program, which includes four main aspects of testing: validity property, post_condition property, metamorphic property and model based property.

The first test is on the validity property. In this test, we will verify that the binary search tree structure still holds after four operations on **bst**, **insert**, **delete**, **empty** and **union**, which require changing the structure of the tree. The main principle is to check the data structure of the binary search tree after the above operation by calling the **valid** function.

The next test is for the post_condition property. In this test, we verify that the structure of the binary tree is not changed after the **find**, **insert**, **delete** and **union** operations. That is, we can still find the value we need by using the binary tree search method **find**.

Then comes the test for metamorphic property. This test is divided into two parts: on the one hand, it tests the change in the length of the binary tree after the completion of **insert**, **delete** and **union** operations, and on the other hand, it tests the effect of the **insert** function on the order of the **insert**, **delete** and **union** functions. We can use the **bst:size()** function to get the length of the binary tree. So the idea for the length test is that **insert** will increase the length of the tree, **delete** will decrease the length of the tree, and the length of the new binary tree generated by **union** is equal to the sum of the lengths of the two old binary trees. For the second part of the test, the program uses the auxiliary function **obs_equals(T1, T2)**. This function uses the **to_sorted_list()** function to convert T1 and T2 into an ordered list to compare whether T1 and T2 are structurally identical. From this we can obtain the following three properties:

- when the key is different, the order of insert does not affect the result, and when two inserts use the same key, the second will overwrite the first.
- when the key is different, the order of delete and insert does not affect the result
- the order of union and insert does not affect the result

Finally, there is a model based test. The main idea of this test is to first perform insert, find, empty, delete and union operations on the binary tree, and then convert the binary tree into a list, which is equivalent to first converting the binary tree into a list and then performing the above operations on the list. Therefore, we need to construct suitable auxiliary functions to operate on the list accordingly. Fortunately, we can guarantee that the list can be generated from a binary tree as long as the sequence of keys in the list remains unchanged. In other words, operations on lists are error-free as long as they are based on the sequence of keys. For example, in the **sorted_insert()** function, we just need to insert the **K, V** we want to insert into the place where the following conditions are met.

$$[..., \{Key1, Value1\}, \{K, V\}, \{Key2, Value2\}, ...], Key1 < K < Key2$$

In addition, it is worth mentioning that for the union test we use the auxiliary function **union_model** to unionize two lists, which is based on the principle of inserting each element of the former list into the latter list using the **sorted_insert** function. It is worth noting that the original **sorted_insert** does not take into account the case where the inserted value already exists (**K =:= Key**), which can lead to elements with the same key in the list when the union operation is performed. Therefore, we add the pattern of overwriting the original attribute when the key is the same to the original function.

1.2.1 Going symbolic

To show the failure test cases intuitively, we redo the function **bst** by using symbolic calls (properties are also fixed):

```
bst(Key, Value) ->
    ?LET(KVS, eqc_gen:list({Key, Value}),
        {call, lists, foldl, [fun({K,V}, T) -> insert(K, V, T) end,
↪   empty(), KVS]}).
```

1.2.2 How to find errors

We run our testing properties on different versions of programs and here we plot the table of the bugs of different versions:

Properties\Version	noether	poitras	rhodes	wilson	perlman	rhodes	snyder	wing
arbitrary_valid				X				
insert_valid				X				
empty_valid								
delete_valid				X				
union_valid		X		X		X		
insert_post				X				X
find_post_present	X			X				
find_post_absent			X					
union_post	X	X		X	X	X		
size_insert								
size_delete								
size_union								
insert_insert	X			X				X
insert_delete			X	X			X	X
insert_union	X	X		X	X	X		X
insert_model	X			X				X
find_model								
empty_model								
delete_model			X				X	
union_model		X		X	X	X		

Through testing, we found that the **noether and wing** version have problems in the detection of operations related to insert. After analyzing the problem, we found that the **noether** version of insert does not overwrite when it encounters the insertion of an existing key, resulting in problems. The **poitras** version has a problem in the operation about union. After analysis, we found that the **poitras, perlman and rhodes** version of the union function does not use the value of the first tree to overwrite when two binary trees have elements with the same key but let the new generated tree have two elements with the same key. The problem with **robinson and snyder**'s version are in the delete function, its delete function doesn't seem to work well. The **wilson** version has a very basic problem, his binary tree structure does not seem to meet the requirements, which causes problems with all three methods except empty, find and delete, which do not insert values into the binary tree. We find that **Property** union_post, insert_union and union_model give the most helpful output, and we do think that symbolic call helps us a lot find the bugs and analyse them. In the example shown below, we first insert { 0, -2 } into a tree with one node {2, -1} and union the inserted tree with the tree with one node {0, 1}. But after inserting {0, 2} the previous tree has the same key value as the next tree, "0". The correct way to handle this would be to use the value of the first tree to overwrite the second tree, but **poitras**' program outputs a binary tree with two identical key values, which is clearly not the case. We can thus check that there may be a problem with the union of the poitras version of the program.

```
prop_insert_union: .......Failed! After 7 tests.
{0,-2,
 {call,lists,foldl,[#Fun<test_bst.19.66689528>,leaf,[{2,-1}]]},
 {call,lists,foldl,[#Fun<test_bst.19.66689528>,leaf,[{0,1}]]}}
   [{0, 1}, {0, -2}, {2, -1}] /= [{0, -2}, {2, -1}]
```

2 Assessment of The Code

2.1 Completeness

All functions are completed, and the completion of all functions are as follows:

Class of Function	Function Name	Completion
Basic functionality in Expression Evaluation	evalTop	Completed
Basic functionality in Expression Evaluation	simplify	Completed
Haskell Testing Property	prop_eval_simplify	Completed
test_bst	prop _arbitrary_valid	Completed
test_bst	prop_insert_valid	Completed
test_bst	prop_empty_valid	Completed
test_bst	prop_delete_valid	Completed
test_bst	prop_union_valid	Completed
test_bst	prop_insert_post	Completed
test_bst	prop_find_post_present	Completed
test_bst	prop_find_post_absent	Completed
test_bst	prop_union_post	Completed
test_bst	prop_size_insert	Completed
test_bst	prop_size_delete	Completed
test_bst	prop_size_union	Completed
test_bst	prop_insert_insert	Completed
test_bst	prop_insert_delete	Completed
test_bst	prop_insert_union	Completed
test_bst	prop_insert_model	Completed
test_bst	prop_find_model	Completed
test_bst	prop_empty_model	Completed
test_bst	prop_delete_model	Completed
test_bst	prop_union_model	Completed

2.2 Correctness

After running Haskell part on the online TA and Erlang part's properties on the **bst.erl** , all test cases were ok. And the condition of correctness is as follows:

Class of Function	Function Name	Test Result
Basic functionality in Expression Evaluation	evalTop	OK
Basic functionality in Expression Evaluation	simplify	OK
Haskell Testing Property	prop_eval_simplify	OK
test_bst	prop_arbitrary_valid	OK
test_bst	prop_insert_valid	OK
test_bst	prop_empty_valid	OK
test_bst	prop_delete_valid	OK
test_bst	prop_union_valid	OK
test_bst	prop_insert_post	OK
test_bst	prop_find_post_present	OK
test_bst	prop_find_post_absent	OK
test_bst	prop_union_post	OK
test_bst	prop_size_insert	OK
test_bst	prop_size_delete	OK
test_bst	prop_size_union	OK
test_bst	prop_insert_insert	OK
test_bst	prop_insert_delete	OK
test_bst	prop_insert_union	OK
test_bst	prop_insert_model	OK
test_bst	prop_find_model	OK
test_bst	prop_empty_model	OK
test_bst	prop_delete_model	OK
test_bst	prop_union_model	OK

```
Erlang/OTP 25 [erts-13.1] [source] [64-bit] [smp:12:12] [ds:12:12:10] [async-threads:1] [jit:ns]

Eshell V13.1  (abort with ^G)
1> c(test_bst).
{ok,test_bst}
2> code:load_abs("bst").
{module,bst}
3> eqc:module(test_bst).
Starting Quviq QuickCheck Mini version 2.02.0
   (compiled for R25 at {{2022,9,29},{14,39,40}})
prop_arbitrary_valid: ....................................................................................................
OK, passed 100 tests
prop_insert_valid: .....................................................................................................
OK, passed 101 tests
prop_empty_valid: .....................................................................................................
OK, passed 101 tests
prop_delete_valid: ....................................................................................................
OK, passed 100 tests
prop_union_valid: .....................................................................................................
OK, passed 101 tests
prop_insert_post: .....................................................................................................
OK, passed 101 tests
prop_find_post_present: .....................................................................................................
OK, passed 101 tests
prop_find_post_absent: ....................................................................................................
OK, passed 100 tests
prop_union_post: .....................................................................................................
OK, passed 101 tests
prop_size_insert: ....................................................................................................
OK, passed 100 tests
prop_size_delete: .....................................................................................................
OK, passed 101 tests
prop_size_union: .....................................................................................................
OK, passed 101 tests
prop_insert_insert: ....................................................................................................
OK, passed 100 tests
prop_insert_delete: ....................................................................................................
OK, passed 100 tests
prop_insert_union: ....................................................................................................
OK, passed 100 tests
prop_insert_model: ....................................................................................................
OK, passed 100 tests
prop_find_model: ....................................................................................................
OK, passed 100 tests
prop_empty_model: ....................................................................................................
OK, passed 100 tests
prop_delete_model: ....................................................................................................
OK, passed 100 tests
prop_union_model: .....................................................................................................
OK, passed 101 tests
[]
4>
```

Figure 1: Output of Erlang part's properties run on the **bst.erl**

2.3 Efficiency

The efficiency of our program is also at a high level.

2.4 Robustness

Class of Function	Function Name	Robustness
Basic functionality in Expression Evaluation	evalTop	Strong
Basic functionality in Expression Evaluation	simplify	Strong
Haskell Testing Property	prop_eval_simplify	Strong
test_bst	prop_arbitrary_valid	Strong
test_bst	prop_insert_valid	Strong
test_bst	prop_empty_valid	Strong
test_bst	prop_delete_valid	Strong
test_bst	prop_union_valid	Strong
test_bst	prop_insert_post	Strong
test_bst	prop_find_post_present	Strong
test_bst	prop_find_post_absent	Strong
test_bst	prop_union_post	Strong
test_bst	prop_size_insert	Strong
test_bst	prop_size_delete	Strong
test_bst	prop_size_union	Strong
test_bst	prop_insert_insert	Strong
test_bst	prop_insert_delete	Strong
test_bst	prop_insert_union	Strong
test_bst	prop_insert_model	Strong
test_bst	prop_find_model	Strong
test_bst	prop_empty_model	Strong
test_bst	prop_delete_model	Strong
test_bst	prop_union_model	Strong

2.5 Maintainability

Class of Function	Function Name	Maintainability
Basic functionality in Expression Evaluation	evalTop	Good
Basic functionality in Expression Evaluation	simplify	Good
Haskell Testing Property	prop_val_simplify	Good
test_bst	prop_arbitrary_valid	Good
test_bst	prop_insert_valid	Good
test_bst	prop_empty_valid	Good
test_bst	prop_delete_valid	Good
test_bst	prop_union_valid	Good
test_bst	prop_insert_post	Good
test_bst	prop_find_post_present	Good
test_bst	prop_find_post_absent	Good
test_bst	prop_union_post	Good
test_bst	prop_size_insert	Good
test_bst	prop_size_delete	Good
test_bst	prop_size_union	Good
test_bst	prop_insert_insert	Good
test_bst	prop_insert_delete	Good
test_bst	prop_insert_union	Good
test_bst	prop_insert_model	Good
test_bst	prop_find_model	Good
test_bst	prop_empty_model	Good
test_bst	prop_delete_model	Good
test_bst	prop_union_model	Good

A Appendix: ExprEval.hs

```haskell
module ExprEval where

import ExprAst
import qualified Data.Map.Strict as M
import Data.Map(Map)

type Env = Map String Int

oper :: Op -> (Int -> Int -> Int)
oper Plus = (+)
oper Minus = (-)
oper Times = (*)

eval :: Expr -> Env -> Either String Int
eval (Const n) _ = return n
eval (Oper op x y) env = (oper op) <$> eval x env <*> eval y env
eval (Var v) env = case M.lookup v env of
                     Nothing -> Left ("Unknown identifier: "++v)
                     Just val -> return val
eval (Let v e body) env = do
  val <- eval e env
  eval body $ M.insert v val env

evalTop e = eval e M.empty

simplify e =
  case e of
    Oper Plus (Const 0) e2 -> e2
    Oper Plus e1 (Const 0) -> e1
    Oper Plus (Const c1) (Const c2) -> Const(c1+c2)
    Oper Minus e1 (Const 0) -> e1
    Oper Minus (Const c1) (Const c2) -> Const(c1-c2)
    Oper Times (Const 1) e2 -> e2
    Oper Times (Const 0) _ -> Const(0)
    Oper Times e1 (Const 1) -> e1
    Oper Times _ (Const 0) -> Const(0)
    Oper Times (Const c1) (Const c2) -> Const(c1*c2)
    Oper op e1 e2 -> Oper op (simplify e1) (simplify e2)
```

```
    Let v e body _> case findVarUsed v body of
      True _> Let v (simplify e) (simplify body)
      False _>  simplify body
    _ _> e


findVarUsed:: String_>Expr_>Bool
findVarUsed str e = case e of
  Const _ _> False
  Var name_> if name == str
    then
      True
    else
      False
  Oper _ e1 e2 _> (findVarUsed str e1)||(findVarUsed str e2)
  Let _ e1 e2 _>(findVarUsed str e1)||(findVarUsed str e2)


```

B Appendix: ExprProperties.hs

```haskell
module ExprProperties where

import Test.QuickCheck

import ExprAst
import qualified ExprEval as E
import qualified ExprEval



exprN :: Int _> Gen Expr
exprN 0 = oneof [fmap ExprAst.Const (arbitrary::Gen Int), fmap ExprAst.Var (arbitrary::Gen String)]
exprN n = oneof
    [fmap ExprAst.Const (arbitrary::Gen Int)
    ,fmap ExprAst.Var (arbitrary::Gen String)
    , do
         x <_ exprN' (n `div` 2)
         y <_ exprN' (n `div` 2)
         return (ExprAst.Oper Plus x y)
   , do
         x <_ exprN' (n `div` 2)
         y <_ exprN' (n `div` 2)
         return (ExprAst.Oper Minus x y)
   , do
         x <_ exprN' (n `div` 2)
         y <_ exprN' (n `div` 2)
         return (ExprAst.Oper Times x y)
   , do
         i <_ fmap ExprAst.Var (arbitrary::Gen String)
         x <_ exprN' (n `div` 2) __stop putting Var in expression x
         y <_ exprN (n `div` 2);
         let id = getIdent i in
            return (ExprAst.Let id x y)]


exprN' :: Int _> Gen Expr
exprN' 0 = fmap ExprAst.Const (arbitrary::Gen Int)
```

```haskell
exprN' n = oneof
    [fmap ExprAst.Const (arbitrary::Gen Int)
    , do
         x <_ exprN' (n `div` 2)
         y <_ exprN' (n `div` 2)
         return (ExprAst.Oper Plus x y)
   , do
         x <_ exprN' (n `div` 2)
         y <_ exprN' (n `div` 2)
         return (ExprAst.Oper Minus x y)
   , do
         x <_ exprN' (n `div` 2)
         y <_ exprN' (n `div` 2)
         return (ExprAst.Oper Times x y)
   , do
         i <_ fmap ExprAst.Var (arbitrary::Gen String)
         x <_ exprN' (n `div` 2)
         y <_ exprN' (n `div` 2);
         let id = getIdent i in
            return (ExprAst.Let id x y)]

instance Arbitrary Expr where
   arbitrary = sized exprN

prop_eval_simplify :: Expr _> Property
prop_eval_simplify x =classify (ExprEval.findVarUsed "" x) "have an empty string as identifier" (E.evalTop x === E.evalTop (E.simplify x))
__prop_eval_simplify x = E.evalTop x === E.evalTop (E.simplify x)

getIdent :: Expr _> String
getIdent (ExprAst.Var id) = id
getIdent _ = ""


```

C Appendix: Test.hs

```haskell

import Test.Tasty
import Test.Tasty.HUnit
import Test.Tasty.QuickCheck

import ExprAst
import qualified ExprEval as E
import qualified ExprProperties as EP
import ExprAst (Op(Minus))

main :: IO ()
main = defaultMain testsuite

testsuite =
  testGroup "Testing expression evaluation and simplification"
  [ testGroup "A few unit_tests"
    [ testCase "Eval: 2 + 2"
      (Right 4 @=? E.evalTop (Oper Plus (Const 2) (Const 2)))
    ,  testCase "Eval: 4 _ 3"
      (Right 1 @=? E.evalTop (Oper Minus (Const 4) (Const 3)))
    ,  testCase "Eval: 3 * 4"
      (Right 12 @=? E.evalTop (Oper Times (Const 3) (Const 4)))
    , testCase "Eval: let x = 3 in x * x"
      (Right 9 @=? E.evalTop (Let "x" (Const 3)
                              (Oper Times (Var "x") (Var "x"))))
    , testCase "Simplify: x + (2 + 2)"
      (Oper Plus (Var "x") (Const 4) @=?
          E.simplify (Oper Plus (Var "x") (Oper Plus (Const 2) (Const
↪   2))))
    , testCase "Simplify: 0 + 2"
      ((Const 2) @=?
          E.simplify  (Oper Plus (Const 0) (Const 2)))
    , testCase "Simplify: 3 _ 0"
      ((Const 3) @=?
          E.simplify  (Oper Minus (Const 3) (Const 0)))
    , testCase "Simplify: x * (3 * 2)"
      (Oper Times (Var "x") (Const 6) @=?
          E.simplify (Oper Times (Var "x") (Oper Times (Const 3) (Const
↪   2))))
```

```
    , testCase "Simplify: 0 * (0 + 2)"
      ((Const 0) @=?
          E.simplify  (Oper Times (Const 0) (Oper Plus (Const 0) (Const
↪  2))))
    , testCase "Simplify: 1 * 6"
      ((Const 6) @=?
          E.simplify  (Oper Times (Const 1) (Const 6)))
    , testCase "Simplify: let"
      ((Const 6) @=?
          E.simplify  (Let "x" (Const 1) (Oper Times (Const 1) (Const
↪  6))))
    ]
  , quickChecks
  ]

quickChecks =
  testGroup "QuickCheck tests"
  [ testProperty "Evaluating a simplified expression does not change its
↪  meaning"
    EP.prop_eval_simplify
  ]
```

D Appendix: test_bst.erl

```erlang

-module(test_bst).

-import(bst, [empty/0, insert/3, delete/2, find/2, union/2]).
-import(bst, [valid/1, to_sorted_list/1, keys/1]).

-include_lib("eqc/include/eqc.hrl").

%% The following two lines are super bad style, except during development
-compile(nowarn_export_all).
-compile(export_all).


% %%% A non-symbolic generator for bst, parameterised by key and value generators
% bst(Key, Value) ->
%         ?LET(KVS, eqc_gen:list({Key, Value}),
%             lists:foldl(fun({K,V}, T) -> {call, bst, insert, [K, V, T]} end,
%                         {call, bst, empty, []},
%                         KVS)).

% bst(int_key(), int_value()) ->
%     ?LAZY(
%         eqc_gen:frequency ([{1, {call, bst, empty, []}},
%                             {4, ?LETSHRINK([T], [bst(int_key(), int_value())], {call, bst, insert, [int_key(), int_value(), T]})}])
%         ).

bst(Key, Value) ->
    ?LET(KVS, eqc_gen:list({Key, Value}),
        {call, lists, foldl, [fun({K,V}, T) -> insert(K, V, T) end, empty(), KVS]}).

% example key and value generators
int_key() -> eqc_gen:int().
atom_key() -> eqc_gen:elements([a,b,c,d,e,f,g,h]).
```

```
int_value() -> eqc_gen:int().


%%% invariant properties

% all generated bst are valid
prop_arbitrary_valid() ->
    ?FORALL(T, bst(int_key(), int_value()),
            valid(eqc_symbolic:eval(T))).

% if we insert into a valid tree it stays valid
prop_insert_valid() ->
    ?FORALL({K, V, T}, {int_key(), int_value(), bst(int_key(),
↪   int_value())},
            valid (insert(K, V, eqc_symbolic:eval(T)))).

prop_empty_valid() ->
            valid (empty()).

prop_delete_valid() ->
    ?FORALL({K, T}, {int_key(), bst(int_key(), int_value())},
            valid (delete(K, eqc_symbolic:eval(T)))).

prop_union_valid() ->
    ?FORALL({T1, T2}, {bst(int_key(), int_value()), bst(int_key(),
↪   int_value())},
            valid (union(eqc_symbolic:eval(T1), eqc_symbolic:eval(T2)))).




%%% -- postcondition properties

prop_insert_post() ->
    ?FORALL({K1, K2, V, T},
            {int_key(), int_key(), int_value(), bst(int_key(),
↪   int_value())},
            eqc:equals(find(K2, insert(K1, V, eqc_symbolic:eval(T))),
                       case K1 =:= K2 of
```

```
                           true ->  {found, V};
                           false -> find(K2, eqc_symbolic:eval(T))
                       end)).

prop_find_post_present() ->
  %  k v t. find k (insert k v t) === {found, v}
    ?FORALL({K, V, T}, {int_key(), int_value(), bst(int_key(), int_value())},
            eqc:equals(find(K, insert(K, V, eqc_symbolic:eval(T))),
                       {found, V})).


prop_find_post_absent() ->
     %  k t. find k (delete k t) === nothing
    ?FORALL({K, T}, {int_key(), bst(int_key(), int_value())},
            eqc:equals(find(K, delete(K, eqc_symbolic:eval(T))),
                        nothing)).

prop_union_post() ->
    %  k v t1 t2. find k (union t1 (insert k v t2)) === {found, v}
    ?FORALL({T1, T2, K, V},
            {bst(int_key(), int_value()), bst(int_key(), int_value()), int_key(), int_value()},
            eqc:equals(find(K, union(insert(K, V, eqc_symbolic:eval(T1)), eqc_symbolic:eval(T2))),
                        {found, V})).



%%% -- metamorphic properties

%% the size is larger after an insert
prop_size_insert() ->
    %  k v t. size (insert k v t) >= size t
    ?FORALL({K, V, T}, {int_key(), int_value(), bst(int_key(), int_value())},
            bst:size(insert(K, V, eqc_symbolic:eval(T))) >= bst:size(eqc_symbolic:eval(T))).

prop_size_delete() ->
```

```
    %  k t. size (delet k t) <= size t
    ?FORALL({K, T}, {int_key(), bst(int_key(), int_value())},
            bst:size(delete(K, eqc_symbolic:eval(T))) =< bst:size(eqc_symbolic:eval(T))).

prop_size_union() ->
    %  t1 t2. size (union t1 t2) == size t1 + size t2
    ?FORALL({T1, T2}, {bst(int_key(), int_value()), bst(int_key(), int_value())},
            bst:size(union(eqc_symbolic:eval(T1), eqc_symbolic:eval(T2))) =< bst:size(eqc_symbolic:eval(T1)) + bst:size(eqc_symbolic:eval(T2))).



obs_equals(T1, T2) ->
     eqc:equals(to_sorted_list(T1), to_sorted_list(T2)).

prop_insert_insert() ->
    ?FORALL({K1, K2, V1, V2, T},
            {int_key(), int_key(), int_value(), int_value(),
             bst(int_key(), int_value())},
            obs_equals(insert(K1, V1, insert(K2, V2, eqc_symbolic:eval(T))),
                       case K1 =:= K2 of
                           true ->  insert(K1, V1, eqc_symbolic:eval(T));
                           false -> insert(K2, V2, insert(K1, V1, eqc_symbolic:eval(T)))
                       end)).

prop_insert_delete() ->
    ?FORALL({K1, K2, V, T},
            {int_key(), int_key(), int_value(), bst(int_key(), int_value())},
            obs_equals(delete(K1, insert(K2, V, eqc_symbolic:eval(T))),
                        case K1 =:= K2 of
                            true -> delete(K1, eqc_symbolic:eval(T));
                            false -> insert(K2, V, delete(K1, eqc_symbolic:eval(T)))
                        end)).
```

```

prop_insert_union() ->
    ?FORALL({K, V, T1, T2},
            {int_key(), int_value(), bst(int_key(), int_value()), bst(int_key(), int_value())},
            obs_equals(union(insert(K, V, eqc_symbolic:eval(T1)), eqc_symbolic:eval(T2)), insert(K, V, union(eqc_symbolic:eval(T1), eqc_symbolic:eval(T2))))).


%%% -- Model based properties
model(T) -> to_sorted_list(T).


prop_insert_model() ->
    ?FORALL({K, V, T}, {int_key(), int_value(), bst(int_key(), int_value())},
            equals(model(insert(K, V, eqc_symbolic:eval(T))),
                   sorted_insert(K, V, delete_key(K, model(eqc_symbolic:eval(T)))))).

prop_find_model() ->
    ?FORALL({K, T}, {int_key(), bst(int_key(), int_value())},
            equals(find(K, eqc_symbolic:eval(T)),
                case find_key(K, model(eqc_symbolic:eval(T))) of
                    nothing -> nothing;
                    _   -> {found, find_key(K, model(eqc_symbolic:eval(T)))}
                end)).

prop_empty_model() ->
    equals(model(empty()), []).

prop_delete_model() ->
    ?FORALL({K, T}, {int_key(), bst(int_key(), int_value())},
            equals(model(delete(K, eqc_symbolic:eval(T))),
                delete_key(K, model(eqc_symbolic:eval(T))))).

prop_union_model() ->
    ?FORALL({T1, T2}, {bst(int_key(), int_value()), bst(int_key(), int_value())},
```

```
            equals(model(union(eqc_symbolic:eval(T1), eqc_symbolic:eval(T2))),
                (union_model(model(eqc_symbolic:eval(T1)), model(eqc_symbolic:eval(T2)))))).


-spec delete_key(Key, [{Key, Value}]) -> [{Key, Value}].
delete_key(Key, KVS) -> [ {K, V} || {K, V} <- KVS, K =/= Key ].

-spec sorted_insert(Key, Value, [{Key, Value}]) -> nonempty_list({Key, Value}).
sorted_insert(Key, Value, [{K, V} | Rest]) when K < Key ->
    [{K, V} | sorted_insert(Key, Value, Rest)];
sorted_insert(Key, Value, [{K, _} | Rest]) when K =:= Key ->
    [{Key, Value} | Rest];
sorted_insert(Key, Value, KVS) -> [{Key, Value} | KVS].

-spec find_key(Key, [{Key, Value}]) -> Value.
find_key(_, []) -> nothing;
find_key(Key, [{K, V}|Rest]) ->
    case Key =:= K of
        true  -> V;
        false -> find_key(Key, Rest)
    end.

-spec union_model([{Key, Value}], [{Key, Value}]) -> [{Key, Value}].
union_model([], KVS2) -> KVS2;
union_model([{K, V}|Rest], KVS2) -> union_model(Rest, sorted_insert(K, V, KVS2)).



%% -- Test all properties in the module: eqc:module(test_bst)


```