Lab session Software Testing, week 2

Fabiën Tesselaar, Lulu Zhang, Tina Curlinovska, Tim Gosen

September 2013

Contents

1 Triangles

1.1 Pre-condition

a, b and c are the desired lengths of the sides of a triangle in integers.

1.2 Post-condition

In this test report we will give short discussion on the results that we expected and the actual results, as well as how we tested the corectness of our program. The program consists of checking the shape of every triangle by implementing defined mathematical rules for each type of triangle.

- "NoTriangle" if the combination of the 3 entered numbers for triangle's sides is not possible; For instance, with negative side lengths
- "Equilateral" if the entered numbers are sides of equilateral triangle, i.e. they are equal;
- "Isosceles" if any two lengths of the sides are equal;

- "Rectangular" if the 3 numbers are lengths of triangle's sides that is rectangular, meaning that it fits the Pythagorean theorem;
- "Other" if the entered numbers are the lengths of triangle's sides that is not rectangular, isoclines or equilateral;

1.3 Test results

As there is an infinite set of numbers (Integer), we cannot prove that our program works completly for all given numbers a, b, c. However, we tested it on a range of known triangles and provided some edge cases:

1. Edges 1 2 5 should be a NoTriangle;
2. Edges 5 5 5 should be Equilateral;
3. Edges 1 2 2 should be Isosceles;
4. Edges 3 4 5 should be Rectangular;
5. Edges 6 4 5 should be Other;
6. The order of the edges given to the function should not matter. We impemented a function that proves that the permutation of the 3 edge numbers and check if the result contains only shapes of the same kind;
7. The program should not return incorrect triangle shapes. We implemented a function that goes through every triangle where all edges rang from 0 to 5 (i.e 0 0 0 and 5 5 5 and everything in between). There should only be one combination that returns "Rectangular" shape (3, 4, 5 as previously stated);

Result of running "testRectangularTriangle" to find all Rectangular combinations in the range of 0 to 5 edged triangles: [(Rectangular,[3,4,5]), ..., (Rectangular,[5,4,3])]

2 Definitions

Contradiction A formula is a contradiction when there is no solution (all possible variations of input result in false). The pre-condition is that it must be a formula. The post-condition is a boolean, indicating if the given formula is in fact a contradiction. In our code we check if any variation of input is True, and negate that outcome. So when there is no solution, the function returns True, False otherwise.

We test the function with a known contradiction $P \wedge \neg P$, $\neg(P \rightarrow Q) \wedge \neg(Q \rightarrow P)$ and known formulas that should not be a contradiction (for example $P \vee \neg P$).

Tautology A formula is tautology only when no matter true or false the variants are, the formula will be true. The precondition of our method is that the input should be a formula. The post condition is a boolean. If it is "True", the formula is a tautology; if not, it means the formular is not a tautology.

We use 4 formulas to test our methods: $P \vee \neg P$ which should always be true, $P \wedge \neg P$ which should always be false, $P \vee Q$ and $P \wedge Q$ which should be satisfactory. After comparing the result values from our Haskell program with the results they should have, our definition methods are proved to be right.

Logical entailment Logical entailment means the conclusion always follow from its premises. Two formulas are logical entailment, only when the implication from antecedent to consequent is tautology. The preconditions are two formulas and the post condition is a boolean, which means whether these two formulas are logical entailment or not.

We test our definition function with 3 sets of formulas: $(P \rightarrow Q) \rightarrow P$ and P, $\neg P$ and P, $P \Leftrightarrow Q$ and $(P \wedge Q) \vee (\neg P \wedge \neg Q)$. The second one should be false and the other two should be true, which cater to the results from our program.

Logical equivalence Two statements are logically equivalent only when they have the same truth value in every model. The preconditions of our checking method are two formulas, and the post condition is a boolean which means whether they are logically equivalent or not.

Two sets of formulas are used to test our definition program: $(P \vee Q) \wedge (P \vee R)$ and $(P \vee Q) \wedge (P \vee R)$, $(P \vee Q) \wedge (P \vee R)$ and $(P \vee R) \wedge (P \vee Q)$. The first set of formula is identical and the second one is just rearranged. Both of them should be logically equivalent, which have the same results from our definition methods.

3 Transforming formulas to CNF

3.1 Pre-condition

The input is a formula conform the implementation (type Form) which is arrowfree (no implications or equivalence) and nnf (no double negations).

3.2 Post-condition

The output formula is conform the implementation (type Form) and in NNF.

3.3 Test results

We need to prove two things with testing:

- The output is in CNF;
- The output is logically equivalent to the input;

We wrote a test function that implements the rules for cnf and check that the input is arrowfree and double negation free before checking for a correct cnf (this is done to improve the lazyness, there is no point in checking for a correct CNF if the formula is not arrowfree). This is to prove the first point. In the previous exercise we wrote a function that checks the logical equivalence of two formulas, so we could use that one to prove our second point. Hence there is one final test function

```
isCorrectCnf :: Form -> Form -> Bool
isCorrectCnf f cnfF = (isCnf cnfF) && (equiv f cnfF)
```

that returns a Bool. True if $cnfF$ is a correct Cnf and logically equivalent to F.

The isCnf function checks that the formula is a conjunction and that all members of the conjunction are in fact in cnf. But as soon as a disjunction is found inside that conjunction, that disjunction can no longer contain other conjunctions. All other patterns are cnf (given that the formula is arrow free and double negation free, but that is guaranteed by checking it upfront in the isCnf function).

Concluding, we have 4 test functions.

- isArrowFree, that checks if a formula is arrowfree
- isNegNegFree, that checks if a formula is double negation free
- isCnf, that checks if a formula is in CNF after isArrowFree and isNegNegFree
- isCorrectCnf, that takes a formula and its CNF and checks if it is in CNF and if it is locgically equivalent.

We use the following examples to prove that this works:

```
*SolWeek2> formTest3
+(*(-2 1) *(2 -1))
*SolWeek2> isCnf formTest3
False
*SolWeek2> cnf (nnf (arrowfree formTest3))
*(*(+(-2 2) +(-2 -1)) *(+(1 2) +(1 -1)))
*SolWeek2> isCnf (cnf (nnf (arrowfree formTest3)))
True
*SolWeek2> isCorrectCnf formTest3 (cnf (nnf (arrowfree
formTest3)))
True
```

This is the most readable test example. Its easy to see the test-formula is not in CNF, but it is after

```
cnf (nnf (arrowfree x)))
```

Then both test return true since it is in cnf and it is the the correct CNF for that function. For more test examples we would like you to refer to the source code.