Software Testing – Week 2

John Tyree* Alex Theiakos†

University of Amsterdam (UvA)
Computational Science

1 Jill vs. Joe – JoeJill.hs

Time: 20 minutes to prove on paper. 2 hours to devise appropriate tests.

In order to test the "Jill and Joe" for optimal strategies two different test functions were implemented. For simplicity and without loss of generality, we assume that Joe's cuts are all in the range $[0.5, 1]$. For the Jill case the test is simple since Jill is the one who makes the initial choice and thus determines Joe's strategy. Jill knows that if she declines the initial cut, Joe is going to cut the second cake into two equal pieces. This means that the optimal strategy for Jill, is to accept the piece that Joe offers from the first cake if it is larger than the other piece plus half of the second cake. In mathematical terms:

$$x \geq (1 - x) + \frac{1}{2}$$
$$x \geq 0.75$$

where x is the amount of the first cake that Jill receives. Clearly, Jill should exercise her right to choose only if the initial cut from Joe is more than 0.75.

The Jill optimality test takes one argument, the piece size cut by Joe. A map operation with a list of possible cuts is performed with the above equation. If the return list consists of *True* expressions the strategy is optimal. To perform fewer operations the list of possible cuts is $x \in (0.5, 0.501...1)$. The argument which represents Joe's cut is placed explicitly in the list.

Joe's decision is influenced by Jill's choice. If Jill doesn't accept the cut, he has to cut the second cake in half in order to minimize the loss. If Jill accepts he gets the entire second cake minus a negligibly small piece. To test Joe's strategy, a "Jill" program has to run inside Joe's program, in order to emulate Jill's decision. The test takes as arguments the Jill test and the initial cutoff.

First a map operations in a list of possible cutoffs is performed in order to determine the maximum cake that Joe can have. The result depends on the choice of the Jill program.

$$\text{Joe's cake} = \begin{cases} 1 - x + 1 & \text{if Jill accepts} \\ x + 1/2 & \text{otherwise} \end{cases}$$

*Student no.: 6423035 | E-mail: tyree@science.uva.nl
†Student no.: 6386628 | E-mail: Alexios.Theiakos@student.uva.nl

Then the result of Joes cake given the cutoff passed as an argument is compared with the maximum element of the list produced by the map operation. If it's the same the program returns *True*, i.e. the strategy is optimal.

2 Properties of Propositions – Form2Bool.hs

Time: 1 hour to decide on the meaning of entailment. 5 minutes to implement.

- `contradiction :: Form -> Bool`
  For this test, we simply applied the definition of contradiction as given directly in the book. A contradiction is false for all valuations, it is therefore unsatisfiable. We tested by feeding it known contradictions in one or more variables, such as $p \wedge \neg p$.

- `tautology :: Form -> Bool`
  Again from the definition in the book. $\neg\top \equiv \bot$. We again tested by giving it known tautologies such as $p \vee \neg p$.

- `entails :: Form -> Bool`
  An entailment something which can only be true if the thing it entails is true. Thus if $\Phi \models \Psi$, then $\Phi \rightarrow \Psi$ is a tautology. We tested this giving our function known tautologies involving implication, such as $p \rightarrow p \vee q$ and $p \wedge q \rightarrow q \vee r$.

- `entails :: Form -> Bool`
  Equivalence of two formulas, Φ and Ψ, means that $\Phi \models \Psi$ and $\Psi \models \Phi$. We define it as such directly. We test it by giving it known equivalent formulas like p and p, $p \vee \neg p$ and $q \wedge (q \vee \neg q)$.

3 Transformation to Conjunctive Normal Form – CNF.hs

Time: 5 hours + 2 more to figure out QuickCheck.

For the conversion task we created the following:

- `cnf :: Form -> Form` – This function has the precondition that the formula is both arrowfree and in nnf. It maps to an equivalent function which is in conjunctive normal form.

- `dist :: Form -> Form` – This is a helper function to push disjunctions inside of conjunctions.

To test this we wrote functions to check if a formula is in valid CNF and is equivalent to the original formula.

- `isCnf :: Form -> Bool` – True iff the formula is in conjunctive normal form.

- `containsCnj :: Form -> Bool` – True iff the formula contains a conjunction.

- `containsDsj :: Form -> Bool` – True iff the formula contains a disjunction.

- `prop_equiv :: Form -> Form -> Bool` – True iff the two formulas are equivalent.
- `prop_valid :: Form -> Bool` – True iff the converted formula is in valid CNF.

We now had the tools to check if an arbitrary formula had been properly converted to CNF. We constructed some simple formulas by hand to test these functions. For thorough testing (and personal growth), we created an instance of Arbitrary Form to use with QuickCheck.

To preventing QuickCheck from creating enormous `Form`s due to recursion, we had to introduce a distribution over the values. Additionally we made functions

- `complexity :: Form -> Int` – The number of subformulas.
- `depthBetween :: Int -> Int -> Form -> Bool` – True iff the leaves of the expression tree are between the first two arguments in depth.

to control the size of QuickCheck's arbitrary values.

After handling problems of explosive recursion, we had great success. The following checks n randomly generated formulas of medium size: `runCnfTests n`.