ST: Lab Assignment 3

Elias El Khaldi Ahanach, Dylan Bartels, Wojciech Czabanski, Quinten Heijn

September 24, 2017

Assignment 1

Time required: 55 minutes.

In the assignment there are 4 functions defined to test whether a form is a contradiction, tautology, whether 2 forms are logically entailed or logically equivalent.

The functions are as follows:

1. **Contradiction** - Tested by checking that a form is not *satisfiable*.
2. **Tautology** - Tested by checking that a form is satisfied for every valuation.
3. **Logical entailment** - Tested by using the following equivalence: $(\text{p} \implies \text{q}) \iff (\neg\ \text{p} \lor \text{q})$
4. **Logical equivalent** - Tested by checking that the evaluated values of two forms are equal for every valuation.

The functions have been tested manually for both *True* and *False* cases.

Assignment 2

Time required: 45 minutes.

For testing the parser we made a selection of interesting test cases of well-formed and ill-formed formulas:

- Well-formed
  1. Empty operator: "+()"
     $\hookrightarrow$ evaluates correctly to a conjuncture operator applied on nothing.
  2. Filled operator: "+(1 2 3 4)"
     $\hookrightarrow$ evaluates correctly to a conjuncture operator applied on the list of atoms.
  3. Single atom: "1"
     $\hookrightarrow$ evaluates correctly to a single atom.
  4. Basic formula using all operators: "+(−0 ∗ (0)(0 ==> 0)(0 <=> 0))"
     $\hookrightarrow$ evaluates all the operators correctly.
  5. Large integer atoms: "100000000000000000000"
     $\hookrightarrow$ evaluates incorrectly to the single atom '7766279631452241920', which hints that there is overflow happening.
- Ill-formed
  1. Miss used implication and equivalence:
     (a) Forgotten atom: "(0 <=>)"
     (b) Added atom: "(0 <=> 00)"
     (c) Forgotten atom: "(==> 0)"
     (d) Added atom: "(00 ==> 0)"

$\hookrightarrow$ all evaluated correctly to an empty list.

2. Missing bracket: "$*(1 + (2 - 3)$"
   $\hookrightarrow$ evaluates correctly to an empty list.
3. Added brackets: "(3)"
   $\hookrightarrow$ evaluates to an empty list. This is correct behavior, since brackets are only used to bind atom to operators.
4. Trailing characters: "$* + (2 - 3)randomcharacters$"
   $\hookrightarrow$ Is parsed correctly, the trailing characters are ignored. It's up for discussion if this is actually appropriate behavior, since the formula was ill-formed. For example: when we parse the string: "$+ (2 - 3)3)$", the parser returns $[+(2 - 3)]$. When we actually might have mend to parse "$* (+(2 - 3)3)$". Now we don't know anything went wrong and it would have been saver if the parser just returned an empty list.

Assignment 3

Time required: 180-240 minutes.

To transform a propositional formula to CNF the following algorithm can be used (taken from "Logic in Computer Science", M. Huth). The postconditions

```
function CNF (f):
/* precondition: f implication free and in NNF */
/* postcondition: CNF (f) computes an equivalent CNF for f*/
begin function
        case
        f is a literal : return f
        f is f1 AND f2 : return CNF (f1) AND CNF (f2)
        f is f1 OR f2 : return DISTR (CNF (f1), CNF (f2))
        end case
end function
```

In this pseudocode the function takes input that is already implication free and in NNF (postcondition). In our implementation we included this as one of the steps so our cnf function can take any correct propositional formula as input.
DISTR is a subfunction that helps get rid of the disjunctions. It's implemented using the following algorithm:

```
function DISTR (f1, f2):
/* precondition: f1 and f2 are in CNF */
/* postcondition: DISTR (f1, f2) computes a CNF for f1 OR f2 */
begin function
        case
        f1 is f11 AND f12 : return DISTR (f11, f2) AND DISTR (f12, f2)
        f2 is f21 AND f22 : return DISTR (f1, f21) AND DISTR (f1, f22)
        otherwise (= no conjunctions) : return f1 OR f2
        end case
end function
```

We implemented the CNF function in Haskell using the algorithms above. To make the input NNF and implication-free we used the functions that were given in Lecture3.
For convenience we added a format function to make the output of the CNF function more readable in particular cases (see code).

Assignment 4

Time required: 300 minutes.

Bonus Assignment 5

Time required: 240 minutes.

The solution consists of two sections: the clause converter *cnf2cls* and the test infrastructure.

Converter The converter simply maps disjunctions to a list of atoms identifiers and conjunctions into lists of disjunctions.

Automated test The automated test is done by using the property *prop_ isEquiv* which, in converts the input form to CNF, then to clause form and eventually compares the valuations of both input and converted forms.

1. The input can be generated by the formula generator from Assignment 4
2. The output of the Clause converter is defined as follows: cnf2cls.cnf
3. The proposed test property is the logical equivalence of the input formula and the output clause
4. Test execution:

```
prop_isEquiv :: Form -> Bool
prop_isEquiv f = checkEquivalence f (cnf2cls (cnf f))
quickCheck (forAll (arbitrary :: Gen Form) prop_isEquiv)
```

Checking for equivalence The main point of interest in the test infrastructure is the *checkEquivalence* function which compares a form and a Clauses object for logical equivalence. In order to achieve that, it:

1. Obtains the number of distinct atoms in a clause.
2. Generates all the valuations for the clause atoms.
3. Compares the evaluations for the input form and the clause form for each valuation combination.