DTU Compute
Michael R. Hansen
04-11-2024

Mini-project: Propositional Logic

Imagine you are on an island populated by two kinds of inhabitants: Knights, who always tell the truth, and knaves, who always lie.

You meet three inhabitants: k1, k2 and k3 and ask them about their kinds:

- k1 mumbles and you cannot really understand what he says.
- k2 says: "k1 said he is a knave."
- k3 says: "k2 is lying".

This is an example of a *Knights and Knaves puzzle* originating from the logician R. Smullyan [1], where you, on the basis of utterances from inhabitants should solve the puzzle by deciding what kinds they are.

These puzzles can be formalized, analysed and solved using propositional logic using concepts, techniques and results from the course on discrete mathematics – a prerequisite that is recommended that you take at least simultaneous with this course.

In the previous two mini projects you have seen the role of discrete mathematics and logic as a meta language to make precise the requirements for solutions, express properties of programs and so on, and you may have observed that the mathematical foundation of functional programming languages can be exploited to address properties of programs in a formal manner.

In this exercise we shall study propositional logic as a topic by developing programs for symbolic manipulation of propositions to implement some results you have meet in the course on discrete mathematics. You will see the use of finite trees in connection with symbolic computations, and you may get a closer look at propositional logic, where you, amongst other things, should transform propositions to disjunctive normal form and compute all satisfying assignments for a proposition. Moreover, you should express and solve a couple of Smullyan puzzles.

Propositional Logic

In this assignment you shall consider formulas of propositional logic, also called *propositions*, which are generated from a set of *atoms* $a, b, c, \ldots$ by use of the well-known operators: *negation* $\neg$, *disjunction* $\vee$ and *conjunction* $\wedge$.

Knowing the truth values of atoms, the truth value of propositions can be computed. For example, if $\mathtt{a}$ is true and $\mathtt{b}$ is false then $\mathtt{a} \wedge \neg\mathbf{b}$ is true and $\mathtt{b} \vee \neg\mathbf{a}$ is false, by use of the truth tables for negation, conjunction and disjunction.

An association of truth values to atoms is also called an *assignment*. If $\mathcal{A}$ is the set of all atoms, then an *assignment* is a subset of $\mathcal{A}$, where the atoms contained in an assignment are those being true. For example, the assignment $\{\mathtt{a}\}$ express that $\mathtt{a}$ is true, while false is the truth value of all other atoms (like, for example, $\mathtt{b}$).

The meaning of propositions is defined in terms the *semantic relation*:

$asg \models P$ reads: "proposition P is true for assignment asg

For example:

- $\{\mathtt{a}\} \models \mathtt{a}$ holds,
- $\{\mathtt{a}\} \models \mathtt{b}$ does not hold,
- $\{\mathtt{a}\} \models \mathtt{a} \wedge \neg\mathtt{b}$ holds, and
- $\{\mathtt{a}\} \models (\neg\mathtt{a}) \vee \mathtt{b}$ does not hold.

We write $asg \not\models A$ to denote that $asg \models A$ does not hold.

The semantic relation $asg \models P$ is defined by induction on the structure of propositions as follows[1]:

$asg \models a$	iff	$a \in asg$	atoms
$asg \models \neg P$	iff	$asg \not\models P$	negation
$asg \models P_1 \vee P_2$	iff	$asg \models P_1$ or $asg \models P_2$	disjunction
$asg \models P_1 \wedge P_2$	iff	$asg \models P_1$ and $asg \models P_2$	conjunction

Two propositions P and Q are *equivalent* if they have the same truth value for every assignment, that is $asg \models P$ iff $asg \models Q$, for every assignment asg.

Any other propositional operator can be expressed using $\neg, \vee$, and $\wedge$. (Actually negation together with just disjunction or just conjunction would suffice.) Implication and biimplication (or equivalence), for example, are definable as follows

$P \Rightarrow Q$	is expressed as	$(\neg P) \vee Q$
$P \Leftrightarrow Q$	is expressed as	$(P \Rightarrow Q) \wedge (Q \Rightarrow P)$

[1] 'iff' reads "if and only if".

Part 1: Abstract syntax and semantics

In this part you shall define a type for the abstract syntax of propositions and a function for the semantics of propositions.

1. Declare a type `Prop<'a>` for propositions so that
   - `A "a" : Prop<string>` represents the atom a,
   - `A("a",3) : Prop<string*int>` represents the atom $(\mathrm{a}, 3)$,
   - `Dis(A "a", A "b") : Prop<string>` represents the proposition $a \vee b$,
   - `Con(A "a", A "b") : Prop<string>` represents the proposition $a \wedge b$, and
   - `Neg(A "a"): Prop<string>` represents the proposition $\neg a$.
2. Declare a function `sem: Prop<'a> -> Set<'a> -> bool` so that

   sem p asg = true iff $asg \models p$ holds

Part 2: Negation Normal Form

A proposition is in *negation normal form* if the negation operator just appears as applied directly to atoms.

3. Declare function `toNnf` p transforming a proposition p into an equivalent proposition in negation normal form, using the de Morgan laws:

   $\neg(P \wedge Q)$ is equivalent to $(\neg P) \vee (\neg Q)$
   $\neg(P \vee Q)$ is equivalent to $(\neg P) \wedge (\neg Q)$

   and the law: $\neg(\neg P)$ is equivalent to P.

The correctness of `toNnf` is addressed in two steps.

4. First, declare a function `onNnf: Prop<'a> -> bool` that can decide whether a proposition is on negation normal form, and use property-based testing to validate that `toNnf` p is on negation normal form for propositions p of type `Prop<string>`.

The second step should validate that p is equivalent to `toNnf` p, that is, `sem` p asg = `sem` (`toNnf` p) asg, for every proposition p and assignments asg.

Notice that if `FsCheck.Check.Quick` is used to validate this property for propositions of type `Prop<string>`, the generated strings will to a large extend be different for very many of the generated pairs of propositions and assignments. Thus, there will be a tendency to assign false to most atoms in the generated propositions. To address this inadequacy, a type with a small number of values could be used as a type for atoms.

5. Declare a type `Finite` having n values. Choose n small; but be aware that it should be meaningful to let `FsCheck.Check.Quick` generate 100 random assignments.

6. Use property-based testing to validate that p is equivalent to `toNnf` p, where p : `Prop<Finite>`.

Part 3: Disjunctive Normal Form

A *literal* is an atom or the negation of an atom and a *elementary conjunction* is a conjunction of literals:

$$lit_1 \wedge lit_2 \wedge \cdots \wedge lit_m$$

where $m > 0$ and $lit_j, 1 \leq j \leq m$, is a literal.

A proposition is in *disjunctive normal form* if it is a disjunction of elementary conjunctions, that is, it has the form:

$$EC_1 \vee EC_2 \vee \cdots \vee EC_n$$

where $n > 0$ and $EC_i, 1 \leq i \leq n$, is an elementary conjunction.

7. Declare a function `dnf` that transforms a proposition in negation normal form into an equivalent proposition in disjunctive normal form using the distributive laws:

$$\begin{array}{lll} a \wedge (b \vee c) & \text{is equivalent to} & (a \wedge b) \vee (a \wedge c) \\ (a \vee b) \wedge c & \text{is equivalent to} & (a \wedge c) \vee (b \wedge c) \end{array}$$

8. Declare a function `toDnf` that transforms a proposition to disjunctive normal form.

9. Declare a function `onDnf` that decides whether a proposition is on disjunctive normal form.

10. Use property-based testing to validate that for every proposition p

    - `toDnf` p is on disjunctive normal form and
    - `toDnf` p is equivalent to p.

Part 4: Satisfying assignments

The disjunctive normal form can be used to extract the assignments satisfying a proposition. Consider, for example, the following elementary conjunction:

$$a \wedge \neg b \wedge c$$

having three atom a, b and c. It is directly observed that an assignment where a and c are true and b is false satisfies the proposition.

On the other hand, no assignment can satisfy the elementary conjunction:

$$\neg a \wedge b \wedge a$$

as atom a occurs both in a not negated and a negated form. Such elementary conjunctions are called *inconsistent*. In general, an elementary conjunction (or a list of literals) is called *consistent* if for no atom a the elementary conjunction (or list) contains both a and $\neg a$.

For a proposition in disjunctive normal form

$$EC_1 \vee EC_2 \vee \cdots \vee EC_n$$

a list of consistent literal lists can be obtained from the consistent elementary conjunctions. For example the list of consistent literal lists for

$$(a \wedge \neg b) \vee (\neg a \wedge b \wedge a) \vee (\neg a \wedge b \wedge c)$$

is $[[a; \neg b]; [\neg a; b; c]]$ as the middle elementary conjunction is inconsistent. In such lists we do not care about the sequence of elements and duplicated elements[2].

11. Declare a function `ecToList` to extract the list of literals occurring in an elementary conjunction.
12. Declare a function to test consistency of a list of literals.
13. Declare a function `toECLists` p that gives a list of lists of literals, where the literal lists come from the consistent elementary conjunctions of p's disjunctive normal form.
14. Introduce an abbreviation for implication $\Rightarrow$ and construct F# values for the two propositions:

$$(\neg P \Rightarrow \neg Q) \Rightarrow (P \Rightarrow Q)$$
$$(\neg Q \Rightarrow \neg P) \Rightarrow (P \Rightarrow Q)$$

    - Find by yourself the satisfying assignments for these two propositions.
    - Apply `toEClists` and reflect over whether the results match your expectations.

[2]One could also declare a function to extract a set of satisfying assignments. The list of list representation may be easier to interpret and is therefore chosen here.

Part 5: Solving puzzles

The general setting for Knights and Knaves puzzles was given in the introduction. Consider now the following puzzle that also appears in [1].

Three inhabitants: k1, k2 and k3, are talking about themselves:

- k1 says: "All of us are knaves."
- k2 says: "Exactly one of us is a knight."

To solve the puzzle you should determine what kinds of citizens k1, k2 and k3 are.

15. Declare a type `Inhabitant` having three values `K1`, `K2` and `K3`.
16. Model the two utterances of the puzzle as a proposition `puzzle1:Prop<Inhabitants>`. You may introduce an abbreviation for biimplication $\Leftrightarrow$.
    Hint: Use atoms k_i, $i = 1, 2, 3$, for the three inhabitants and construct your proposition so that k_i being true means that `K`i is a knight, and k_i being false means that `K`i is a knave.
17. Solve first the puzzle by yourself. Then use `toEClists` to analyse `puzle1`. Reflect over whether the result matches your expectation.
18. Solve the puzzle from the introduction by yourself. Then model and solve the puzzle using `toEClists`. Reflect over whether the result matches your expectation.

Part 6: Exponential blow-up of disjunctive normal forms

The *Boolean satisfiability problem* **SAT** is the problem of deciding whether a given proposition has a satisfying assignment. This decision problem was the first problem shown to be an NP-complete problem, where NP stands for nondeterministic polynomial and complete means that it is at least as hard to decide as any other problem in NP.

Without going into formal details, NP is the set of decision problems that can be verified in polynomial time and SAT is in NP because for a give proposition p and assignment (witness) asg, the `sem` function can verify efficiently (in polynomial time) the truth of p for assignment asg. But notice that there are 2^n possible assignments for a proposition having n atoms.

Many famous problems are NP-complete and only solutions having a worst-case exponential running time have been found for these problems.

The satisfiability of a formula in disjunctive normal form is easily checked (in polynomial time) as we have seen above.

We shall, therefore, illustrate that the transformation to disjunctive normal form may give an exponential blow-up of the size of propositions. In particular, we shall consider propositions of the following form:

$$(p_1 \vee q_1) \wedge (p_2 \vee q_2) \wedge \cdots \wedge (p_n \vee q_n) \tag{1}$$

and illustrate that the disjunctive normal forms of such propositions will have 2^n elementary conjunctions.

19. Declare an F# function `badProp` n representing proposition (1). You may use atoms like `A("P",`i`)` and `A("Q",`i`)` in the declaration of `badProp`.

20. Compute `toEClists(badProp` n`)` for a small number of cases and check that the resulting lists indeed have 2^n elements.

There are many excellent SAT-solving tools (implemented using other techniques than those used in this exercise). They all have an exponential worst-case complexity, but have, despite of that, been used to solve huge problem instances in many areas of computer science.

References

[1] Raymond M. Smullyan, *What is the name of this book? The Riddle of Dracula and Other Logical Puzzles*, Dover Publications, 2011 (Originally published by Prentice-Hall, 1978)