Homework 2
COMP 302 Programming Languages and Paradigms

Francisco Ferreira and Brigitte Pientka
McGill University: School of Computer Science

Due Date: 6 October 2017

Your homework is due at the beginning of class on Oct 6, 2017. All code files must be submitted electronically using my courses. **Your program must compile.** Solutions should correct (i.e. produce the correct result), be elegant and compact, and take advantage of OCaml's pattern matching. Please consult the style guides posted on the course website to get more information regarding what constitutes good style.

Q1. I can't get no SATisfaction (75 pts)

This question is about reading, understanding and then improving the design of a program. The first part describes a program and its source code, and then asks you to rewrite the program by choosing a different representation for data-types. What you need to write will be short but it requires understanding the program. So read along and pay attention to the role of each function.

Logic is ubiquitous in Computer Science. It is used in many different aspects of the field, reading "On the Unusual Effectiveness of Logic in Computer Science" by Halpern et al. gets your imagination running. In this question we will explore an important problem, whether a formula in Propositional Logic is satisfiable (i.e.: a formula is satisfiable if one can assign values to all its atoms in a way that the formula evaluates to true). This is known as the SAT problem. This problem is an important part of model checking, automated theorem provers and formal methods in general.

Let start defining a small SAT solver[1] with a simple design that we will later improve. First, let's consider propositional logic, these are formulas that link atoms that can be true or false we relate them with conjunctions, disjunctions and negation. The syntax of propositions is thus:

[1]This SAT solver is described in StandardML by [Paulson, 1996]

$$\begin{array}{llcll} \text{Propositions} & P, Q & ::= & A & \text{An atom} \\ & & | & \neg P & \text{A negation (Boolean not)} \\ & & | & P \wedge Q & \text{A conjuction (Boolean and)} \\ & & | & P \vee Q & \text{A disjunction (Boolean or)} \end{array}$$

You can think of atoms as the indivisible propositions, like “Socrates is a human”, in fact we will represent them using strings, the other propositions we will represent with a custom datatype:

```
type prop = Atom of string
          | Not of prop
          | And of prop * prop
          | Or of prop * prop
```

With these datatype in place, we can define implication and bi-implication using the following equivalences:

$$\begin{array}{rcl} P \Rightarrow Q & = & (\neg P) \vee Q \\ P \iff Q & = & (P \Rightarrow Q) \wedge (Q \Rightarrow P) \end{array}$$

We can implement this with two simple functions:

```
let impl (p, q) = Or(Not p, q)
let iff (p, q) = And (impl (p,q), impl (q, p))
```

Given this type, we can represent propositions that arbitrary nest negation, disjunction and conjunction. This makes for an expressive theory but it is difficult to write programs that manipulate such general expressions. To solve this issue one can define some standardized forms, we call these normal forms. In order to do this one needs to transform the expression using rules that do not modify the meaning of the proposition.

A small example: the proposition $\neg(P \wedge \neg P)$ is the proposition that says: it is not the case that `P` and its negation hold at the same time. This is known as the principle of non-contradiction. This propositions is represented as:

```
let nc = Not (And (Atom "P", Not (Atom "P")))
```

Negation Normal Form (NNF)

The first normal form that we want to explore is called *negation normal form* where negation is only applied to atoms. One can reach this form by pushing negations into conjunctions and disjunctions using the following rules, repeatedly replacing:

$$\begin{array}{rcl} \neg\neg P & \text{by} & P \\ \neg(P \wedge Q) & \text{by} & (\neg P) \vee (\neg Q) \\ \neg(P \vee Q) & \text{by} & (\neg P) \wedge (\neg Q) \end{array}$$

This can be achieved with the following function:

```
let rec nnf = function
  | Atom a -> Atom a
  | Not (Atom a) -> Not (Atom a)
  | Not (Not p) -> p
  | Not (And (p, q)) -> nnf (Or (Not p, Not q))
  | Not (Or (p, q)) -> nnf (And (Not p, Not q))
  | And (p, q) -> And (nnf p, nnf q)
  | Or (p, q) -> Or (nnf p, nnf q)
```

This function leaves alone positive (`Atom a`) and negative atoms (`Not (Atom a)`), removes double negations according to the first rule, and it uses the two other rules (called de-Morgan rules in the literature) to push the negations under conjunctions and disjunctions. Finally, the last two lines just recursively call the function under the constructors for conjunction and disjunction.

For example, we can find the NNF of the principle of non-contradiction. The steps are:

1. Starting from $\neg(P \wedge \neg P)$
2. We apply the second rule to get $\neg P \vee \neg\neg P$
3. Then the first rule to eliminate the double negation to get $\neg P \vee P$.

Conjunctive Normal Form (CNF)

The conjunctive normal form is when a proposition has this form:
$P_1 \wedge \cdots \wedge P_m$ and each P_i is a disjunction of atoms and negated atoms. This form will be crucial to implement our SAT solver.

To put a proposition in CNF we need to write these two simple functions:

```
let rec distribute : prop * prop -> prop = function
  | p, And (q, r) -> And(distribute (p, q), distribute (p, r))
  | And(q, r), p -> And(distribute (q, p), distribute (r, p))
  | p, q -> Or (p, q)

let rec nnf_to_cnf: prop -> prop = function
  | Or(p, q) ->
     (* distribute the disjunction *)
     distribute (nnf_to_cnf p, nnf_to_cnf q)

  | And(p, q) -> And (nnf_to_cnf p, nnf_to_cnf q)
  | Atom a -> Atom a
  | Not p -> Not p
```

The idea of function `nnf_to_cnf` is to distribute disjunctions into its sub-propositions using these two replacements:

$$\begin{array}{ccc} P \vee (Q \wedge R) & \text{by} & (P \vee Q) \wedge (P \vee R) \\ (Q \wedge R) \vee P & \text{by} & (Q \vee P) \wedge (R \vee P) \end{array}$$

Function `distribute` repeatedly rewrites the proposition using these two rules to achieve a CNF proposition. Notice, how the right side of the rewriting is somehow *more in conjunctive normal form* than the left side.

Our running example, the principle of non-contradiction that we stated as $\neg(P \wedge \neg P)$ becomes $\neg P \vee P$. In this case no further rewriting is needed as this proposition is simple enough to also be in CNF.

As a final example, the proposition $\neg(P \vee Q) \wedge P$ is:

- $(\neg P \wedge \neg Q) \vee (\neg P)$ in NNF
- $(\neg P \vee \neg P) \wedge (\neg Q \vee \neg P)$ in CNF

when we mechanically apply the rules.

Finding tautologies

The CNF propositions have a very regular structure that makes computing with them very easy. The first function that we want to implement is whether the proposition is a tautology. Where a tautology is a proposition that is valid for all possible values of its atoms. One classic example is modus ponens :

$$(P \wedge (P \Rightarrow Q)) \Rightarrow Q$$

Modus ponens can represented in our system as:

```
let mp = impl (And (Atom "P",
                    impl (Atom "P", Atom "Q"))
               , Atom "Q")

(* And it can be put in CNF form using this expression: *)
let mp_in_cnf = nnf_to_cnf (nnf mp)
```

To check if a proposition in CNF form $P_1 \wedge \cdots \wedge P_m$ is a tautology, we check that *all* P_i *are tautologies*. Because P_i is disjunction of positive and negative atoms, P_i is a tautology iff there exists a positive and negative atom in P_i. For example, if $P_i = a \vee \neg a \vee b$, then P_i is always true, since either a is true or $\neg a$ is true.

To check that a disjunction P_i is a tautology, we first write a function that extracts the positive and the negative atoms from it and then check that the intersection between negative and positive atoms is not empty. This guarantees that there is at least one atom that appears as positive and negative in disjunction and the disjunction P_i cannot be false.

The following code implements this idea:

```
let rec positives = function
  | Atom a -> [a]
  | Not (Atom _) -> []
  | Or (p, q) -> positives p @ positives q
  | _ -> raise (Invalid_argument "Not␣in␣NNF␣form")
```

```
let rec negatives = function
  | Atom _ -> []
  | Not (Atom a) -> [a]
  | Or (p, q) -> negatives p @ negatives q
  | _ -> raise (Invalid_argument "Not␣in␣NNF␣form")

let rec cnf_tautology = function
  | And (p, q) -> cnf_tautology p && cnf_tautology q
  | p -> not ([] = intersection (positives p) (negatives p))
```

Q1.1 **(5 points)** The function `cnf_tautology` depends on a function called `intersection` that has not been defined yet. Define the function with type:

```
intersection : 'a list -> 'a list -> 'a list
```

This function returns all the elements that are present in both lists at the same time.

Satisfiable propositions

Now that we can check if a proposition is a tautology we can use a bit of classical logic to write a SAT solver. We call a proposition *unsatisfiable* if its negation is a tautology. And finally we call a proposition *satisfiable* if it is not unsatisfiable.

The three functions on propositions are then:

```
let taut p = cnf_tautology (cnf p)
let unsat p = taut (Not p)
let sat p = not (unsat p)
```

Types are our strength

While this implementation works, there are a couple of issues that should bother us. First, if we call `nnf_to_cnf` with a proposition that is not in nnf it may return the wrong answer. The same thing happens if we call `cnf_tautology` with a proposition not in CNF. And second, the functions `positives` and `negatives` raise exceptions! if we did not have the propositions in CNF. While some of these problems might be mitigated by hiding some functions from the user, we can do better by using types.

For example, a data type that can only represent propositions in NNF would be:

```
(* Atoms and their negations *)
type signed_atom
  = PosAtom of string
  | NegAtom of string

(* In NNF negations can only be applied to atoms, this datatype only
   allows for propositions in NNF *)
type nnf
  = AndN of nnf * nnf
  | OrN of nnf * nnf
  | AtomN of signed_atom
```

Now, let's rewrite our SAT solver using this kind of idea. In the file `hw2_q1.ml` fill in the following questions:

Q1.2 **(10 points)** Implement `to_nnf : prop -> nnf` to convert propositions to NNF.

Q1.3 **(20 points)** Design a type called `cnf` that is only able to represent propositions in CNF, follow the ideas from type `nnf` for inspiration.

Q1.4 **(20 points)** Implement the functions:

- `distribute : cnf * cnf -> cnf`
- `nnf_to_cnf : nnf -> cnf`

Be mindful, that depending on your specification of type `cnf` you might need to also define other helper functions.

Q1.5 **(5 points)** Implement the functions that collect positives and negatives from a proposition in conjunctive normal form.

Q1.6 **(10 points)** Finally we are ready to implement the function `cnf_tautology: cnf -> bool` that tells us when a proposition in CNF is a tautology.

Pattern matching in all these functions should be exhaustive, the point of this exercise is to design the types that allow us to implement functions that do not fail at run-time.

This concludes this question, take a moment to reflect about how pattern matching helps with the manipulation of symbolic expressions, and how good definitions help us write functions that are safe at run-time.

If after you finish this assignment you want to learn more (better algorithms and interesting programs with logic) you can read the "Handbook of Practical Logic and Automated Reasoning" by Harrison. It contains pointers to ideas to extend this implementation so it performs well with big formulas and many more algorithms.

References

Joseph Y. Halpern, Robert Harper, Neil Immerman, Phokion G. Kolaitis, Moshe Y. Vardi, and Victor Vianu. On the Unusual Effectiveness of Logic in Computer Science. *Bulletin of Symbolic Logic*, 7(2):213236, 2001. doi: 10.2307/2687775.

John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996.

Q2. If we have a proof it's because its proven (25pts)

Hand in your proofs as a pdf file q2.pdf

When writing a tail recursive function, one may come up with this function:

```
let rec rev_append l1 l2 = match l1 with
  | [] -> l2
  | x::xs -> rev_append xs (x::l2)
```

This function reverses the first list and appends it to the second. A similar function is available int the OCaml standard library. And in fact, the reverse function is implemented using this function. This is a nice implementation because it is tail recursive.

However, if you are asked to implement this function you might come up with:

```
let rec append l1 l2 = match l1 with
  | [] -> l2
  | x::l1' -> x::(append l1' l2)

let rev l = match l with
  | [] -> []
  | x::xs -> xs @[x]

let rev_append' l1 l2 = append (rev l1) l2
```

This is a more obvious implementation of a function that reverses its first parameter and appends the result to the second.

Q2.1 (25 points) Prove that the `rev_append` and `rev_append'` are equivalent.