CS:4420 Artificial Intelligence
Spring 2019

Homework 3

Part B

Due: Friday, Mar 15 by 11:59pm

This assignment has two parts, A and B, both to be done *individually.* This document describes Part B which consist of both programming and written problems.

Download the accompanying OCaml source file `hw3B.ml` and enter your solutions in it where indicated. When you are done, submit it on ICON with the same name. Make sure you *write your name in that file* where indicated.

Each of your answers *must be free of static errors.* You may receive no credit for problems whose code contains syntax or type errors.

Pay close attention to the specification of each problem and the restrictions imposed on its solution. Solutions ignoring the restrictions may receive only partial credit or no credit at all.

File `hw3B.ml` contains a number of auxiliary functions for your convenience. You are encouraged them to use them as needed to implement the functions above or to run your tests. Some test examples are provided as well but you should not take them to be exhaustive. You are allowed to use your own helper functions as needed as well as any OCaml library fucntions. Those for lists may be most helpful.[1] Use recursion, pattern matching and combinators to write clean code.

Solutions that maximize the use of combinators will receive extra credit

In this assignment you will complete the OCaml implementation of a simple build but relatively sophisticated resolution procedure. Since resolution procedures operate on CNF formulas, which are essentially sets of clauses, you will also complete the implementation of a procedure that converts propositional formulas to sets of clauses.

Note The function specifications below may include some requirements on the input values (e.g., they are positive, sorted, and so on). Implement the functions by simply assuming that the requirements are satisfied. Do not worry about what a function does when those requirements are violated.

[1] `https://msdn.microsoft.com/library/a2264ba3-2d45-40dd-9040-4f7aa2ad9788`

1 Clauses in OCaml

In file `hw3B.ml` propositions are implemented as terms of the algebraic datatype `prop` as in Part A. Clauses are implemented as lists of literals, element of the `lit` type which is a synonum of `int`. Positive integers represent positive literals, negative integers represent negative literals, and 0 represents the `True` literal. The empty clause `[]` stands for `False` (the empty disjunction). The literals in a clause are sorted in strictly increasing order for greater efficiency of clause manipulating functions.

```
type lit = int
type clause = lit list
type cnf = clause list
```

In the conversion from propositions to clauses, a positive literal like `3`, say, corresponds to the propositional variable `(P 3)` while `-3` corresponds to `(Not (P 3))`. So a clause like `[-3; 1; 2]` represents for the proposition `(Or (Or (Not 3, 1), 2))` or, equivalently, `(Or (Not 3, Or (1, 2)))`. Sets of clauses are implemented as clause lists such as `[[-3; 1; 2]; [-1; 2; 4][-2; 4]]`.

Using all the types above, implement the following functions.

1. A clause c_1 *subsumes* a clause c_2 if every literal of c_1 is also a literal of c_2. Subsumed clauses can be eliminated during resolution because they are provably redundant. Implement function `subsumes: clause -> clause -> bool` which returns `true` if its first input subsumes the second, and returns `false` otherwise.

2. A clause c_1 *properly subsumes* a clause c_2 if it subsumes c_2 but it is not subsumed by it. Implement function `psubsumes: clause -> clause -> bool` which returns `true` if its first input properly subsumes the second, and returns `false` otherwise.

3. A clause set *cs subsumes* a clause *c* if there is a clause in *cs* that subsumes *c*. Implement function `setSubsumes: cnf -> clause -> bool` which returns `true` if its first input subsumes the second, and returns `false` otherwise.

4. A clause is a *tautology* if it valid. This is the case exactly when the clause contains the *true* literal (0) or a literal and its complement. In resolution, it is useful to recognize and discard tautologies because they too are redundant. Implement function `isaTautology: clause -> bool` which returns `true` if its input is a tautology, and returns `false` otherwise.

2 CNF Conversion in OCaml

In this problem you will develop some of the main pieces of a procedure that converts a proposition, expressed as a term of type `prop`, to its conjunctive normal form. The conversion procedure essentially follows the steps seen in class and described in the textbook and the class notes. The main difference is that it eventually produces a set of clauses expressed as a term of type `cnf`.

1. Implement function `elimIff: prop -> prop` which takes a proposition and returns an equivalent one containing no applications of `Iff`.

2. Implement function `elimImpl: prop -> prop` which takes a proposition with no applications of `Iff` and returns an equivalent one with no applications of `Impl`.

3. Implement function `pushNot: prop -> prop` which takes a proposition with no applications of `Iff` and `Impl`, and returns an equivalent one in *negation normal form (NNF)* where all applications of `Not`, if any, apply to a propositional variable. In the NNF reduction, convert `(Not False)` to `True` and `(Not True)` to `False`.

4. Complete the given implementation of function `distributeOr: prop -> prop` which takes a proposition in NNF with no applications of `Iff` and `Impl`, and returns one where no applications of `And` occur within an application of `Or`.

3 Resolution in OCaml

The given source code provides an implementation of a resolution-based prover similar in spirit to the one described in Figure 7.12 of the textbook but with a number of enhancements to make it more scalable. It also defines the following types:

```
type queryAnswer = Yes | No | Unknown
type resolAnswer = Sat |  Unsat | Stop
```

The first models possible answers to a query to a knowledge base. The second models possible answers given by a resolution procedure.

The main procedure, implemented in the `resolution` function, takes a clause set *cs* as input and tries to derive the empty clause from it applying the resolution rule systematically. If it succeeds then it has proven *cs* unsatisfiable and so it returns `Unsat`. It can fail to derive the empty clause in one of two cases:

a. it *saturates*, that is, it generates all possible non-redundant consequences of the input set with none of them being the empty clause;

b. it runs out of computational resources expressed by a positive integer bound, provided as input, on the number of derivation rounds.

In the first case, the procedure has proven that *cs* is satisfiable and so it returns `Sat`. In the second case, the process is inconclusive and the procedure returns `Stop`.[2]

At each round, the procedure maintains three sets of clauses: *new* clauses, *processed* clauses, and *old* clauses. The input clauses and any newly derived clauses go in *new*. As long as *new* is non-empty, a clause *c* is chosen and removed from it at each round. If the *c* is the empty clause, the procedure stops with `Unsat`. If the clause is non-empty but it is a tautology or is subsumed by the clauses in *processed* or *old*, it is tossed away for being redundant. Otherwise, *c* is moved to *processed*. However, it is first used to remove from *processed* and *old* any clauses that are properly subsumed by *c* because such clauses are now redundant.

When *new* becomes empty, the procedure chooses a clause *c* from *processed* and computes all resolvents between *c* and the clauses in *old*. Those resolvents are added to *new* and *c* is moved to *old*. When both *new* and *processed* are empty, the procedure has saturated and stops with `Unsat`.

[2] The resource bound has been added for your convenience, in case you want to experiment with large or otherwise hard input problems.

At each round the initial input bound n is decremented by 1. When the bound reaches 0 the procedure stops with `Stop`.

The resolution function is fully implemented in the given code. You are to implement the following auxiliary functions as well as the main procedure for our resolution-based prover.

1. Implement function `resolve: lit -> clause -> clause -> clause` which takes a non-zero literal n, a clause c_1 containing n, and a clause c_2 containing $-n$ and returns the resolvent of the two clauses with respect to n.

   Example: `(resolve (-2) [-3; -2; 4] [-3; 1; 2])` should be `[-3; -1; 4]`.

2. Implement function `prove: prop list -> prop -> int -> queryAnwer` which takes a list kb of propositions, a proposition a, and a positive integer n. It returns `Yes` if it can prove by n rounds of resolution that kb entails a ($kb \models a$); it returns `No` if it can prove by n rounds or resolution that kb does not entail a; it returns `Unknown` otherwise.

4 Using the Prover

Use the prover implemented by `prove` to answer the following questions:

1. $\mathbf{False} \overset{?}{\models} \mathbf{True}$
2. $\{a, \neg a\} \overset{?}{\models} \mathbf{False}$
3. $a \wedge b \overset{?}{\models} a \Leftrightarrow b$
4. Is the set $\{a \vee b, a \Rightarrow d, c \Rightarrow d, \neg d, b \Leftrightarrow c\}$ satisfiable?
5. Is the formula $a \Rightarrow (b \Rightarrow a)$ valid?

Provide the answers in an OCaml comment, explaining how you used `prove` to get those answers.

5 Proving Queries

In this problem you are to encode certain facts as values of type `prop` and use function `prove` from the previous problem to prove or disprove entailments between them. Consider the following description of unicorns and their properties.

> If the unicorn is mythical then it is immortal. However, if it is not mythical then it is a mortal mammal. If the unicorn is either immortal or a mammal (or both) then it is horned. The unicorn is magical if it is horned.

1. Express the facts above as propositions of type `prop`. Assign each proposition to an OCaml variable as indicated in the source file.

2. Use `prop` to determine which of the following facts are entailed by the set of facts in the previous question.

(a) The unicorn is mythical.
(b) The unicorn is magical.
(c) The unicorn is horned.

Provide your answers in an OCaml comment, explaining how you used `prove` to get those answers.