CSCE 625-600 Program #2
Resolution Theorem Prover

See eCampus for due dates, etc.

February 27, 2020

1 Introduction

In this programming assignment, you will write a resolution theorem prover for first-order logic in LISP. You may use a different language but example code will be provided in LISP only.

2 Representation of clauses

Use a list of clauses, where each clause is in the following form:

```
"(" <clause-number> <positive-literal-list> <negative-literal-list> ")"
```

where `<clause-number>` is an integer, `<positive-literal-list>` is a list of positive literals and `<negative-literal-list>` is a list of negative literals. Variables are represented as single symbols (atomic terms) such as X, Y, etc., and constants are represented as (A), (B), i.e., a single atom list. Predicates and functions are represented in the usual prefix notation: P(X) is (P X), and F(X) is (F X).

For example, $P(x, y) \vee \neg Q(x, a) \vee R(f(x))$ where x is a variable a is a constant, and $f(\cdot)$ is a function, can be expressed as below.

```
Positive literals: (P X Y) (R (F X))
Negative literal:  (Q (A))
Resulting clause:  (1   ( (P X Y) (R (F X)) )   ( (Q X (A)) )   )
```

A whole theorem will contain a list of such clauses. For example,

```
(
   (1 ( (P X Y) (R (F X)) )    ( (Q X (A)) )    )
   (2 ( (R (F Z)) )            ( (P Z (A)) )    )
   (3 ( (Q W V) )              ( )              )
)
```

Note that if the given clause only contains positive or negative literals but **not both**, the empty one should be represented as `( )`, such as in clause 3 above (which would be simply $Q(w, v)$).

Initially, the list will contain the premise clauses, followed by clauses derived from the negated conclusion. As resolvents are generated from these clauses, they are added to the end of the list. For simplicity, you may assume that the sets of symbols used as predicate names, function names, and variable names are unique in any given problem (as in the example above), i.e. you can do all the necessary pre-processing by hand before feeding in the problem to the prover. For example, clauses 1, 2, and 3 above do not have any shared variable.

However, as you go along with resolution, you may have to check if the two clauses drawn from the set of clauses have overlapping variables. To check this, you may have to extract all the arguments of the literals. For this, use `mapcan`, which applies a function on each element of a given list and concatenates the result. There are other `map..` functions that may be of interest to you as well, such as `mapcar`.

```
* (mapcan #'cdr '((p x y) (r (f x))))
(X Y (F X))

* (mapcar #'cdr '((p x y) (r (f x))))
((X Y) ((F X)))
```

Basically what you are doing above is `(cdr '(p x y))` which gives you `(x y)` and `(cdr '(r (f x)))` which gives you `((f x))`, which is then appended (for `mapcan`). See the next section for some more details on renaming variables to avoid confusion.

3 Coding the Resolution Algorithm

General instructions

Here are some general instructions you should follow:

1. Print out each input clause and each clause that is produced by resolution; number the printed clauses and show the numbers of the clauses from which they are derived.
2. Remember that more than one resolution of a given pair of clauses may be possible. We will not worry about resolving on factors of clauses. For example, resolving the two clauses below

$$C_1 : P(x) \vee P(John) \vee Q(x)$$

$$C_2 : \neg P(y)$$

will result in

$$C_3 : P(John) \vee Q(x)$$

$$C_4 : P(x) \vee Q(x).$$

Note that you can then resolve C_2 and C_3 to get $Q(John)$ as well, which is what you'd get when you resolve the factor of C_1 and C_2 in a single step. This is a rather contrived example, but you will encounter cases where the intermediate clauses such as C_4 are needed. See below for more on this.

3. A unification algorithm is provided in the file `sunify.lsp`. The clauses must first be rewritten so that they have no variables in common in case there are repeated variables across two clauses. To avoid any confusion, simply replace every variable in all clause with a new symbol (do it consistently so that the same variable is replaced with the same new symbol) before you start.

   For example,

```
* (load "sunify.lsp")

; Loading #p"/user/choe/sunify.lsp".
T
* (unify '(p x) '(p (a)))

((X A))
* (unify '(p x (f x)) '(p (g x) (f (a))))

NIL
* (unify '(p x (f x)) '(p (g y) (f (g (a)))))

((Y A) (X G (A)))
* (unify '(p x) '(p y))

((X . Y))
```

   The result is `NIL` if not unifiable or a list of subsitutions represented as A-lists. So, `((X A))` contains a single subsitution `(X A)` which is actually `(X . (A))`. See the last example where this is more explicit.

   If you need to generate new symbols, use `(intern (symbol-name (gensym)))`. Try running this several times and see what kind of symbol you get. For example, if you have `(1 ((P X)) ((Q X (A))) )` and you want to replace x with a unique new variable, you can do something as follows, using the the `(subst <replace-pattern> <pattern> <expression>)` function. Note that `G1303` is the newly created symbol.

```
* (subst (intern (symbol-name (gensym))) 'x '(1 ((P X)) ((Q X (A)))))

(1 ((P G1303)) ((Q G1303 (A))))
*
```

3.1 Resolution algorithm

Use the **two-pointer method** to select pairs of clauses for resolution; initialize the pointers so that you will be using the **set-of-support strategy**. This can be done in the following way:

1. Initialize: Set the “inner loop” pointer to the front of the list of clauses. Set the “outer loop” pointer to the first clause resulting from the negated conclusion.

2. Resolve: If the clauses denoted by the two pointers can be resolved, produce the resolvent (you may have to go through the positive and negative literal lists in the two clauses looking for a single complementary literal). Add the resolvent to the end of the list of clauses and print it out. (Note: There may be more than one possible resolvent from a pair of clauses. In that case, generate multiple resolvants based on all possible resolvable pairs. See below. Also, if you generated a clause that is already in the list of clauses, do not add it.) If the resolvent is empty (**False**), stop; the theorem is proved.

3. Step: Move the "inner loop" pointer forward one clause. If the "inner loop" pointer has not reached the "outer loop" pointer, go to the Resolve step. Otherwise, reset the "inner loop" pointer to the front of the list of clauses and move the "outer loop" pointer forward one clause. If the "outer loop" pointer goes beyond the last clause in the list, stop; the theorem cannot be proved (no more clauses are left that are resolvable). The "two-pointer method" is a breadth-first method that will generate many duplicate clauses.

When resolving two clauses, for simplicity, we will not find the factor. Instead, for all possible resolvents, generate a list of resulting clauses. For example, suppose you have to resolve the two clauses below.

```
( 1 ( (P X) (P (A)) )     ( (Q X) ) ) ; clause 1
( 2 ( )                   ( (P (A)) ) ) ; clause 2
```

In this case, in the first clause, there are two (P ·)s, so, what you do is produce two clauses, by first unifying positive (P X) and negative (P (A)) to get

```
( 3 ( (P (A) )            ( (Q (A)) ) ) ; clause 3
```

and next, unifying positive (P (A)) and negative (P (A)) to get the following.

```
( 4 ( (P X) )           ( (Q X) ) ) ; clause 4
```

So, you will get two clauses (clauses 3 and 5) by resolving the clauses clause 1 and 2.

For duplicate checks, simply use the `(dupe ...)` function in `dupeclause.lsp` in the src/ directory. This is a really simplistic, literal duplicate checker, so it will say certain clauses are not duplicates when they really are (logically), but that's fine, we just want to do a quick elimination of the obvious cases. For example, try this:

```
* (dupe '(1 ((q x) (p y)) ()) '(2 ((p y) (q x)) ()))

T

* (dupe '(1 ((q x) (p y)) ()) '(2 ((p y) (q (a))) ()))

NIL

* (dupe '(1 ((q x) (p y)) ()) '(2 ((p y) (q z)) ()))

NIL
```

3.2 Unit preference

Implement unit preference, and compare the performance in:

- the number of resolution steps taken to derive **False**, and/or
- if it ever reaches the answer or not.

For this, you will have to sort the clauses in terms of the number of clauses in them.

3.3 Question answering

For both provers above, include the question answering feature. Introduce a new predicate $Answer(x)$ so that you can retrieve the answer. Note that $Answer(x)$ should be treated as having a value of **False** so when you generate that predicate by itself, you've reached the end.

4 Submission Instruction

4.1 Testing your prover

Run your provers on these three problems and summarize the results (see the file `theorems.lsp`):

1. Howling hound,
2. Drug dealer and customs official, and
3. Coyote and roadrunner.

Do all of these reduce to **False**?

In addition to the above:

4. Convert the Harmonia example in `slide04.pdf`, page 133, into a resolution problem format in Lisp, and run the prover. Once you have derived **False**, change the theorem into a question answering form, and run your prover once again the retrieve the answer: who is Harmonia's grandparent.
5. Write an example theorm on your own that has at least 5 clauses (including the clauses from the negated conclusion) that is true, and test it. You can start very simple, with just two clauses, then you can increase the number.

Name and run your prover like this:

- `(two-pointer *problem* 6)`

- `(unit-preference *problem*)`

where `*problem*` is a global variable pointing to the list of clauses (e.g. `*howl-hound*`), and 6 is the first clause of the negated conclusion (assume that the clause number begins with 1).

4.2 Submitting

Follow the instructions **exactly** to avoid any penalty. Any deviation from these instructions will result in a 10-point (out of 100) penalty.

Submit a single zip file containing the following:

- Zip file name should be `<your-UIN>-prog2.zip`. For example, `123004567-prog2.zip`.
- In the zip file, there should be a single folder named `<your-UIN>-prog2`. For example, `123004567-prog2`.
- In the `<your-UIN>` folder (e.g., `123004567-prog2`), include the following:
  - `prover.lsp` (your main file). If using a different language, include your source code here. The main file should be prover.xxxx.
  - `sunify.lsp` and other files needed to run the prover.
  - `report.pdf`
  - `README.txt`

The `README.txt` is a plain text file that contain compliation instructions and example commands of how to run your program. Include this for all languages, including LISP.

The `report.pdf` file should contain the following:

1. The Harmonia theorem represented in Lisp.
2. Test results of `two-pointer` and `unit-preference` on the five theorems (howling hound, customs official, roadrunner, Harmonia, and your own theorem).
   - Draw a table where the rows are the theorems and the columns are the two theorem proving methods. Thus, you'll get 10 results in all.
   - In each cell, enter the number of resolution steps taken to derive **False**.
   - If it takes too long, stop it after say 10 minutes and indicate that in the table, and report why you think it failed. Make sure that your prover works for trivial cases (theorems consisting of 2 to 3 clauses).
3. Explain what you think are the differences in the resulting resolution steps between the two different theorem proving methods.
4. Print out the full resolution steps in each case.

The grading criteria are as follows:

1. `README`: 5% (0% if the grader is unable to compile and run based on the provided instructions)
2. `report.pdf`: 15
3. Program: 80% (`two-pointer`: 60%; `unit-preference`: 20%)

Appendix

The coyote and roadrunner theorem is as follows:

1. Every coyote chases some roadrunner.
2. Every roadrunner who says “beep-beep” is smart.
3. No coyote catches any smart roadrunner.
4. Any coyote who chases some roadrunner but does not catch it is frustrated.
5. (Conclusion) If all roadrunners say “beep-beep”, then all coyotes are frustrated.

More examples at http://www.cs.utexas.edu/users/novak/resosol.html. All theorems in this assignment are originally by Gordon Novak (see the above url).

All program files and theorem files are located in http://courses.cs.tamu.edu/choe/20spring/src/.