Quick and Dirty RTP

Roy M. Turner

Spring, 2021

Description

This file defines a very rough version of a resolution theorem prover as an example of how one can be written. It doesn't handle equality statements (e.g., `(= now 2021)`), but it *does* handle computable predicates.

Header, variables

Make use of unify[1] and the messages utilities I gave you.

[1] This exposed a bug—after all these years!

```
(in-package cl-user)

(load "unify")
(load "messages")
(shadowing-import 'msg:msg)
(use-package 'msg)
```

Computable predicates are defined here, in the form `(predicate . function)`:

```
(defvar *computable-predicates*
    '((gt . >)))
```

Here are our old friends, the Marcus axioms:

```
(defvar *axioms*
    '(;; human(Marcus)
      ((human Marcus))
      ;; Pompeian(Marcus)
      ((Pompeian Marcus))
      ;; born(Marcus,40)
      ((born Marcus 40))
      ;; forall x human(x) => mortal(x)
      ((not (human ?x1)) (mortal ?x1))
      ;; forall x Pompeian(x) => died(x,79)
      ((not (Pompeian ?x2)) (died ?x2 79))
      ;; erupted(volcano,79)
      ((erupted volcano 79))
      ;; forall x, t1, t2 mortal(x) & born(x,t1) & gt(t2-t1,150) => dead(x,t2)
      ((not (mortal ?x3)) (not (born ?x3 ?t1)) (not (gt (- ?t2 ?t1) 150)) (dead ?x3 ?t2))
```

```
       ;; now = 2021
       ((= now 2021))
       ;; forall x, t [alive(x,t) => ~dead(x,t)] & [~dead(x,t) => alive(x,t)]
       ((not (alive ?x4 ?t3)) (dead ?x4 ?t3))
       ((dead ?x5 ?t4) (alive ?x5 ?t4))
       ;; forall x,t1,t2 died(x,t1) & gt(t2,t1) => dead(x,t2)
       ((not (died ?x6 ?t5)) (not (gt ?t6 ?t5)) (dead ?x6 ?t6))
       ))
```

Make SBCL thing shut up about functions being called before they're used, so it behaves like a sane Lisp:

```
#+:sbcl (declaim (sb-ext:muffle-conditions style-warning))
```

RTP and helper

I take the approach here of passing the *actual* theorem to rtp, which then negates it and passes it to a "helper function", `rtp-recur`, that does the actual work. This allows me to use recursion without having to worry about "Have I negated the axiom yet?" If I were really fancy, I'd use `flet` or some such so as not to clutter up the namespace; but this is a quick version, remember?

I set a futility cut-off at 30 inferences.

```
(defun rtp (theorem axioms &key bindings (limit 30))
```

I'm using set-of-support, as I advised you folks to do, by having `rtp-recur` always try to refute the previous resolvent. To use the unit preference strategy, I just sort the axioms in order of increasing number of literals. Note that this doesn't do much for Marcus & co.

```

  (setq axioms (sort (copy-list axioms) #'(lambda (a b)
                                              (< (length a) (length b)))))
```

Negate the theorem and call `rtp-recur`, return appropriate thing:

```
  (multiple-value-bind  (contradiction blist)
      (rtp-recur (negate theorem) axioms :bindings bindings :limit limit)
    (if contradiction
      (fmsg "Theorem can be proven.  Bindings = ~s." blist)
      (fmsg "Cannot prove the theorem."))
    (values (not contradiction) blist)))
```

Here's `rtp-recur`. I use `with-indentation` to give a notion of the recursion. First, I check for having been passed `nil` for the theorem—this is the base case for terminating recursion. I next check for hitting the futility cut-off, and return `:fail` if I've hit it.

```
(defun rtp-recur (clause axioms &key bindings (limit 30))
  (with-indentation
      (vfmsg "Attempting to find contradiction with clause ~s." clause)
    (dfmsg "[current limit=~s]" limit)
    (vdfmsg "[bindings=~s]" bindings)

    (cond
     ((null clause)
      (values  nil bindings))
     ((= limit 0)
      (fmsg "Reached futility cut-off; assuming theorem cannot be proved.")
      (values :fail bindings))
     (t
```

This is the guts of the function. I apologize for the messiness; if I were *really* doing this, I would split this into more comprehensible pieces—and I'd likely do it all recursively and get rid of the extremely ugly `return-from` in the process.

Basically, I loop through the theorem's (contained in `clause`) literals, looking at each to see if it is a computable predicate or if it resolves with any literal of any axiom (i.e., via the other two loops).

If a literal is s computable predicate or a negated computable predicate, then I check the truth of that. It may look a little weird, since if the literal evaluates to `nil`, we get rid of it and recur on all the other pieces of the clause. That's because a computable predicate only "resolves" if it would have matched the negated form of itself in the axiom set, if there were one; so that means that since the thing in the axiom set would have been true, this has to be false in order to "resolve".

```
      (loop for literal in clause
          when (and (computable-predicate? literal)
                    (not (true-predicate? literal bindings)))
          do (multiple-value-bind (success blist)
                 (rtp-recur (instantiate (remove literal clause :test #'equal)
                                         bindings)
                            axioms
                            :bindings bindings
                            :limit (1- limit))
               (if (not success)
                 (return-from rtp-recur (values nil blist))))
          else do
               (loop for axiom in axioms do
                     (loop for aliteral in axiom do
                           (multiple-value-bind (success blist)
```

```
                                (resolves? literal aliteral bindings)
                              (dfmsg "[~s and ~s~a resolve]"
                                     literal aliteral (if success "" "do not"))
                              (when success
                                (vfmsg "Resolved: ~s" clause)
                                (vfmsg "    with: ~s" axiom)
                                (multiple-value-setq (success blist)
                                  (rtp-recur
                                   (instantiate (resolvent literal clause aliteral axiom  blist)
                                                blist)
                                   axioms :bindings blist :limit (1- limit)))
                                (when (null success)
                                  (return-from rtp-recur (values nil blist))))))))
          (dfmsg "[finished with loops; failure]")
          (values :fail bindings)))))
```

Just some functions to see if something *is* a computable predicate, and, if so, to evaluate it.

```
(defun computable-predicate? (literal)
  (if (eql 'not (car literal))
    (computable-predicate? (negate-literal literal))
    (get-predicate-function (car literal))))

(defun true-predicate? (literal &optional bindings)
  (let (value)
    (setq literal (instantiate literal bindings))
    (setq value
      (cond
       ((unbound-var-in-literal? literal)
        t)
       ((eql 'not (car literal))
        (not (true-predicate? (cadr literal) bindings)))
       (t
        (eval `(,(get-predicate-function (car literal)) ,@(cdr literal))))))
    (dfmsg "[computable predicate ~s is ~a]"
           literal (if  value "true" "false"))
    value))

(defun get-predicate-function  (pred)
 (cdr (assoc pred *computable-predicates*)))

```

If there is a variable in the computable predicate's literal with no value, then we can't really evaluate the literal. Consequently, we return "t" whether the literal is positive or negative to force `rtp-recur`

Quick and Dirty RTP

to skip over it and continue, leaving it in any resolvent so it can be evaluated later in the process.

```
(defun unbound-var-in-literal? (lit)
  (cond
   ((null lit) nil)
   ((listp lit)
    (or (unbound-var-in-literal? (car lit))
        (unbound-var-in-literal? (cdr lit))))
   ((variable? lit) t)
   (t nil)))
```

This sees is two literals resolve with one another; the next function computes the resolvent of two clauses that have two of their literals match in this way.

```
(defun resolves? (lita litb &optional bindings)
  (unify (negate-literal lita) litb bindings))

(defun resolvent (clause-lit clause axiom-lit axiom &optional bindings)
  (append (remove clause-lit clause :test #'equal)
          (remove axiom-lit axiom  :test #'equal)))
```

A really kludgy negate function that accepts a clause and returns its negation. Well, as long as the clause isn't *too* complicated. If it has the form:

```
((and A B C))
```

where A, B, and C are literals, then it will return

```
((not A) (not B) (not C))
```

using our good friend de Morgan's Law. [2]All my clauses, even the unary ones, have an implied OR, and so are lists of literals.} If the clause just contains a single literal, then the (trivial) negation of that is returned, using `negate-literal`. And if the clause is an ORed list of literals (i.e., a list of more than two literals), then it returns the negation of the first one, with another value containing a list of the remaining *clauses*: after all, something like that would turn into multiple clauses. Note that this is as complicated as I handle – no embedded ORs or ANDs or NOTs them, etc.

[2] {

```
;;
;; Note: this only handles one level of nesting for ANDed or ORed clauses!
;;
```

Quick and Dirty RTP

```
(defun negate (clause)
  (cond
   ((null clause) nil)
   ((> (length clause) 1)
    ;; then it's equiv to (or a b c), so return two values, ((not a)) and a
    ;; list of the negated forms of the others, as per de Morgan's law:
    (values (list (negate-literal (car clause)))
            (mapcar #'(lambda (a) (negate (list a))) (cdr clause))))
   ((eql 'and (caar clause))
    ;; then it's (and a b c), so return (~a ~b ~c) as per de Morgan's:
    (mapcar #'negate-literal (cdr (car clause))))
   (t (list (negate-literal (car clause))))))

(defun negate-literal (lit)
  (if (eql 'not (car lit))
    (cadr lit)
    (list 'not lit)))
```

And, finally, allow SBCL to whine about warnings again:

```
#+:sbcl (declaim (sb-ext:unmuffle-conditions style-warning))
```