Unify

UMaine COS 470/570

Spring 2019

File info/modification history

```
;;; Author: Roy Turner <rturner@maine.edu>, UMaine
;;; Date: Created: 1/25/90
;;; Modifications:
;;;   o 3/14/90: prepared for class
;;;   o 13:04 - Thu Nov 5, 1998 -rmt- Modifications added for COS 470, F98:
;;;      o newSymbol symbol generator added.  Call is:
;;;             (newsymbol <foo>)
;;;        where <foo> is a symbol or a string.  A new, unique symbol based
;;;        on that is returned.  Any trailing numerals are stripped from the
;;;        symbol, and then a new number is appended to make a unique name.
```

Introduction

This file contains functions to perform unification on predicate calculus expressions. They are modifications of the unify functions in Winston and Horn.

Unify has two required arguments, the things to be matched; and one optional argument, a *unifier* (or *binding list*). It returns two values (using Lisp's `values` function). The first is either `T` or `nil`, depending on whether or not the unification was successful. The second is a unifier—this is either a new binding list or the one that was passed in, modified with new bindings.

A binding list has the following form:

$$((var_1 val_1)(var_2 val_2)...(var_n val_n))$$

where var_i is a FOPC (first-order predicate calculus) variable: a symbol whose name begins with `?`, e.g., `~?x`.

Note that a FOPC variable is *not* a Lisp variable! Lisp will be unaware of any value that it has; its value is solely one with respect to a binding list. So if there are two binding lists:

```
A: ((?X PLUTO) (?Y MICKEY))
```

and

```
B: ((?X MICKEY) (?Y GOOFY))
```

then `?X`'s value with respect to A is `PLUTO` and its value with respect to B is `MICKEY`.

In addition to `unify`, this file also contains functions for dealing with FOPC variables and for *instantiating* a Lisp atom or list using a binding list: that is, replacing all of the variables in the atom/list with their corresponding values.

The functions in this file work on two versions of Common Lisp that I have access to, Allegro Common Lisp (ACL) and Steel Bank Common Lisp (SBCL). If you use a different Lisp, you may have to make changes!

Predicate calculus in Lisp

Here are some suggestions about how to represent predicate calculus expressions in Lisp.

FOPC syntax	Lisp version
Human(Socrates)	(Human Socrates)
Human(x)	(Human ?x)
Human(x) ∧ Roman(x)	(and (Human ?x) (Roman ?x))
Pompeian(x) ∨ Roman(x)	(or (Pompeian ?x) (Roman ?x))
¬Pompeian(x)	(not (Pompeian ?x))
Bird(x) ⇒ Flies(x)	(implies (Bird ?x) (Flies ?x))
Human(Mother-of(?))	(Human (Mother-of ?x))
∀x∃y Person(x) ∧ Person(y) ⇒ Loves(x,y)	(forall (?x) (exists (?y) (implies (and (Person ?x) (Person ?y)) (Loves ?x ?y))))

Note that you won't need all this for your resolution theorem prover! Since all your knowledge will be in conjunctive normal form (CNF), there will be no qualifiers (i.e., ∀, ∃) and the only connective used will be `OR`. With regards to only using `OR`, this means you don't have to specify the `OR` at all unless you just want to:

```
(OR (HUMAN MARCUS) (CAT MARCUS))
```

could just be represented as

```
((HUMAN MARCUS) (CAT MARCUS))
```

Variables for FOPC

Note that there are some functions in this file to help you deal with FOPC variables, i.e., those that have a question mark before their name. Here are ones that will be useful:

- {(variable? *thing*)} – returns true if *thing* is a variable:

  ```
  (variable '?x) ==> t
  (variable 'x) ==> nil
  ```

- {(varname *var*)} – returns the name of *var* if it is a variable:

  ```
  (varname '?x) ==> X
  ```

- {(make-var *name*)} – returns a variable whose name is *name*:

  ```
  (make-var 'x) ==> ?X
  ```

  You can use the `newSymbol` function (see below) to create a unique variable name:

```
(make-var (newSymbol 'x)) ==> ?X1
(make-var (newSymbol 'x)) ==> ?X2
```

- {(find-binding *var*)} – gives you back the variables value with respect to some binding list“

  ```
  (find-binding '?x '((?y 3) (?x 4))) ==> (?X 4)
  ```

- {(add-binding `var val bindings`)} – adds a binding for a variable to a binding list

A general function to replace variables in lists is also provided, `instantiate`:

```
CL-USER> (instantiate '?x '((?z Marcus) (?x Caesar)))
CAESAR
CL-USER> (instantiate '(not (Human ?z)) '((?z Marcus) (?x Caesar)))
(NOT (HUMAN MARCUS))
CL-USER>
```

Symbol generator

Lisp has a basic facility to generate new symbols, `gensym`. However, although you can specify a different basename (so all your new symbols don't come out as `GEN101`, `GEN102`, etc.), there are two problems. First, all symbols draw from the same numeric sequence. So:

```
CL-USER> (gensym)
#:G913
CL-USER> (gensym "FOO")
#:FOO914
```

when you might rather have `FOO1` as your first generated variable, and maybe `X1` as another, etc.

The second problem is more serious: the symbols returned exist, but are not *interned* in any symbol table. This means you can use them, but you can't refer to them by name anywhere until you intern them yourself, which is annoying.

To avoid this, I include here some code cribbed from our MaineSAIL utilities. This implements a `SymbolGenerator` class to keep track of prefixes for new symbols and keep separate counts for them, as well as interning them (in the package specified by `*intern-package*`).

Symbol generator code

This class holds the state of symbol generation. This was copied from Utilities, where a class was needed; if doing this from scratch for this course, I'd have used just a hash table.

```
(defclass SymbolGenerator ()
  ((counterTable :initform (make-hash-table :test #'equal))))
```

This method returns a unique symbol based on `prefix`, which can be a string or a symbol. The parameter `package` controls where the thing is interned, and `intern` determines if it is interned at all (sometimes, it's not needed).

```
(defun newSymbol (&optional prefix &key (package) (intern t))
  (with-slots (counterTable) *symbolGenerator*
   (let (num sym)
     (cond
       ((symbolp prefix)
 ;; convert to string, call again:
 (newSymbol (symbol-name prefix) :package package :intern intern))
       ((stringp prefix)
 ;; get rid of trailing numerals:
 (setq prefix (string-right-trim "0123456789" prefix))
 ;; try new symbol names until we find one that is not in use:
 (loop do
       (cond
        ((setq num (gethash prefix counterTable))
 ;; number exists for this prefix -- new number is just incremented
 ;; one:
 (setq num (1+ num))
 (setf (gethash prefix counterTable) num))
        (t
 ;; no number yet:
 (setf (gethash prefix counterTable) 1)
 (setq num 1)))
     until (not (find-symbol
 (setq sym (string-append prefix
  (princ-to-string num))))))
 ;; found one, create the symbol...
 (setq sym (make-symbol sym))
 (when intern      ;then intern the symbol:
   (setq sym (if package
       (intern (format nil "~a~s" prefix num) package)
       (intern (format nil "~a~s" prefix num)))))
 sym)
       (t
 ;; then can't do any better than regular old gensym:
 (gensym))))))
```

This is a bit of a kludge, creating a symbol generator here, rather than having you do it in your program. Of course, this really should all be a separate file, say `symbol-generator.lisp`, with `unify.lisp` creating an instance of `symbolGenerator`.

```
(defvar  *SymbolGenerator* (make-instance 'SymbolGenerator))
```

Unify code

Package-related bookkeeping

First tell Lisp that we're in the `CL-USER` package, then define `*intern-package*`, which is used by `newSymbol` to determine where to put newly-created symbols that you have asked it to create.

If your Lisp happens to call the user package something else (ACL used to call it `USER`, for example), then this falls back to whatever the current package is.

```
(defvar *intern-package* (or (find-package "CL-USER") *package*)
  "Package in which to intern symbols created by unify, etc., functions.")
```

Macros

First, define `string-append`, which I use instead of the less-readable `concatenate` function.

```
(unless (fboundp 'string-append)
  (defmacro string-append (&rest strings)
    `(concatenate 'string ,@strings)))
```

Now, set up `?` to be a *macro character* to allow `?X`, etc., to be treated like a variable. (Don't worry about the `*var*` stuff; that's a remnant from another project.)

```
(set-macro-character #\?
   #'(lambda (stream char)
 (let ((next-char (peek-char nil stream))
       next foo)
   (cond
    ((equal next-char #\))
     ;;it's a paren, so it's invalid as a variable...just
     ;; return symbol ?
     (setq foo (intern "?" *intern-package*))
     foo)
    ((equal next-char #\space)
     (setq foo (intern "?" *intern-package*))
     foo)
    (t
     (setq next (read stream t nil t))
     (cond
      ((atom next)
       ;;return ?atom
       (multiple-value-bind (thing dummy)
    (intern  (string-append (string #\?)
    (symbol-name next)))
    thing))
      (t
       `(*var* ,next)))))))
   t)
```

unify

Here is the `unify` function. It takes two things (atoms or lists) as input, along with an optional binding list. It returns two values (using `values`): the first is true if the two things matched, and the second is a binding list. If there was no match, the binding list returned is the same as the one passed in, which if there *was* a match, then it is the initial binding list updated with any new variable matches.

You can catch the two variables using the Lisp "special forms" `multiple-value-setq` and `multiple-variable-bind`. The first works like `setq`, but for multiple things:

```
(multiple-value-setq (a b) (values 1 2))
```

will set `a` to 1 and `b` to 2, e.g. The other form works like a `let`:

```
(setq a 3
      b 4)
(multiple-value-bind (a b)
    (values 1 2)
  (print a)
  (print b))
(print a)
(print b)
```

would print 1, 2, 3, 4.

```
(defun unify (p1 p2 &optional bindings)
  (cond
   ((variable? p1)
    (unify-variable p1 p2 bindings))
   ((variable? p2)
    (unify-variable p2 p1 bindings))
   ((and (atom p1) (atom p2))
    (unify-atoms p1 p2 bindings))
   ((and (listp p1) (listp p2))
    (unify-elements p1 p2 bindings))
   (t (values nil bindings))))
```

unify-atoms

Two non-variable atoms unify iff they are equal. This is basically a function included for clarity of the main function; it should probably be re-defined sometime as a macro. It takes two things to compare and a binding list, and it returns the same thing as `unify`.

```
(defun unify-atoms (p1 p2 bindings)
  (values (eql p1 p2) bindings))
```

unify-elements

This looks through the elements of two lists, making sure that corresponding elements unify and maintaining appropriate bindings. It returns the same things as `unify`.

```
(defun unify-elements (p1 p2 bindings)
  (let (blist matched?)
    (multiple-value-setq (matched? blist)
```

```
     (unify (first p1) (first p2) bindings))
   (cond
    ((null matched?)
     (values nil bindings))
    ((multiple-value-setq (matched? blist)
(unify (rest p1) (rest p2) blist))
     (values matched? blist))
    (t
     (values nil bindings)))))
```

unify-variable

This unifies a variable (the first argument) with an arbitrary expression (the second one), updating bindings as necessary. It returns the same things as `unify`.

```
(defun unify-variable (p1 p2 bindings)
  (cond
   ((eql p1 p2)
    (values t bindings))
   (t
    (let ((binding (find-binding p1 bindings)))
      (if binding
   (unify (extract-value binding) p2 bindings)
 (if (inside? p1 p2 bindings)
     (values nil bindings)
     (values t (add-binding p1 p2 bindings)))))))))
```

find-binding

This looks up a variable's binding in `bindings`. The binding can have the variable in the `car` or the `cadr`. However, since this is used elsewhere (i.e., `instantiate-variable`), we have to handle the case where we found the variable as the binding of another variable – in this case, we don't want to just return the first variable! So you can specify a list of variables in `not-one-of` that `var` won't be allowed to bind to.

A second value is returned (via `values`) that indicates whether or not a binding was found. This allows you to distinguish this from the case in which the variable was bound, but to `nil`.

```
(defun find-binding (var bindings &optional not-one-of)
  (let ((binding
  (car (member var bindings
       :test #'(lambda (a b)
 (let ((poss (cond
      ((eql a (car b))
       (cadr b))
      ((eql a (cadr b))
       (car b)))))
   (when (and poss
      (not (member poss not-one-of)))
     t)))))))
```

```
    (cond
     ((null binding) (values nil nil))
     ((eql var (car binding))
      (values binding t))
     (t (list var
       (values (car binding) t))))))
```

extract-value

This just extracts the value portion of a binding. It should really be a macro, since it's so simple.

```
(defun extract-value (binding)
  (cadr binding))
```

inside? and inside-or-equal?

The functions `inside?` and `inside-or-equal?` each return true if `var` occurs in `expr`.

Probably sometime I should eliminate `inside-or-equal?` should be eliminated and its code incorporated into `inside?` via an `flet`.

```
(defun inside? (var expr bindings)
  (if (equal var expr)
      nil
      (inside-or-equal? var expr bindings)))

(defun inside-or-equal? (var expr bindings)
  (cond
   ((equal var expr) t)
   ((and (not (variable? expr)) (atom expr)) nil)
   ((variable? expr)
    (let ((binding (find-binding expr bindings)))
      (when binding
 (inside-or-equal? var (extract-value binding) bindings))))
   (t (or (inside-or-equal? var (first expr) bindings)
   (inside-or-equal? var (rest expr) bindings)))))
```

add-binding

This function adds a new binding of `var` to `val` to the `bindings`. It returns the updated binding list; it is non-destructive, so the original binding list is unchanged.

```
(defun add-binding (var val bindings)
  (if (eq '_ var)
      bindings
      (cons (list var val) bindings)))
```

variable?

This returns true if `thing` is a variable. Again, don't worry about the `*var*` stuff.

```
(defun variable? (thing)
  (or (and (listp thing)
           (equal (car thing) '*var*))
      (and (symbolp thing)
           (equal (char (symbol-name thing) 0)
                  #\?))))
```

varname

This returns the name of `var`, i.e., it strips off the leading `?`.

```
(defun varname (var)
  (cond
    ((and (consp var)
          (consp (cdr var)))
     (cadr var))
    ((equal (char (string var) 0) #\?)
     (intern  (string-left-trim '(#\?) (string var))
       (find-package *intern-package*)))))
```

make-var

This creates a new variable whose name is `var` by putting a `?` before its name.

```
(defun make-var (var)
  (intern (concatenate 'string "?"
        (cond
 ((stringp var) var)
 (t (symbol-name var))))))
```

instantiate

This “instantiates” a thing, for example, a clause or literal, by replacing all variables with their bindings. To see how unbound variables are handled, see the below.

```
(defun instantiate (thing bindings &key (if-unbound :first))
  (cond
   ((variable? thing)
    (instantiate-variable thing bindings :if-unbound if-unbound))
   ((atom thing)
    thing)
   (t
    (cons (instantiate (car thing) bindings :if-unbound if-unbound)
   (instantiate (cdr thing) bindings :if-unbound if-unbound)))))
```

instantiate-variable

This will instantiate a variable using a set of bindings. This means that if the variable is bound to another variable, that variable's binding will be chased down, etc., until a value is found.

The keyword parameter `if-unbound` determines what happens if the variable is unbound, or if it is bound to a variable that ultimately is unbound. If the value is `:first`, then the first variable is left in the expression; if `:last`, then the last variable found is left. If nil, then `nil` is returned. (Actually, that's not quite true: whatever it is other than `:first` or `:last` is returned – so you can have it return, e.g., `:unbound`, if you like).

For example, suppose the variable `b` contains these bindings:

```
((?x 2) (?y ?x) (?z ?a)).
```

The behavior is as follows, where `=>` means returns:

```
(instantiate-variable '?x b) => 2
(instantiate-variable '?y b) => 2
(instantiate-variable '?z b) => ?Z
(instantiate-variable '?z b :if-unbound nil) => nil
(instantiate-variable '?z b :if-unbound :last) => ?A
```

```
(defun instantiate-variable (var bindings &key (if-unbound :first))
  (multiple-value-bind (found val)
      (inst-var var bindings)
    (cond
     (found val)
     ((eql if-unbound :first)
      var)
     ((eql if-unbound :last)
      (cadr val))
     (t
      if-unbound))))
```

This is just a helper function for `instantiate-variable`.

```
(defun inst-var (var bindings &optional (depth 0))
  (loop with deeper-var = nil
      for binding in bindings
      do
 (when (member var binding)
   (let (found
 (val (if (eql var (car binding) )
  (cadr binding)
        (car binding))))
     (cond
      ((not (variable? val))
       (return (values t val)))
      ((multiple-value-setq (found val)
 (inst-var val
   (remove binding bindings :test #'equal)
   (1+ depth)))
       (return (values t val)))
      ((variable? (cadr val))
```

```
      (when (or (null deeper-var)
      (> (car val) (car deeper-var)))
(setq deeper-var val))))))
     finally
;; if we get here, we haven't returned from the things above --
;; meaning we haven't found var in bindings at all!  In this case, we
;; need to return the variable itself as the value, though noting that
;; we haven't found a real binding.
(return (values nil
  (if (or (null deeper-var)
(<= (car deeper-var) depth))
    (list depth var)
  deeper-var)))))
```