An Implementation of Unify

Roy M. Turner
Department of Computer Science
University of Maine

Fall, 2010

Contents

1 Introduction

This is a version of the unify algorithm written for use by students in UMaine's COS 140 (Introduction to Artificial Intelligence) class. It is a general-purpose unifier, but it is especially targeted for unification of predicate calculus expressions. It is based on the unify functions in Winston and Horn's early AI book.

The primary entry point to these functions is `unify`. `unify` has two required arguments—expressions—-and one optional argument–a unifier (or *binding list*). It returns two values (using `values`). The first is either T or nil, depending on whether or not the unification was successful. The second is a unifier (or updated unifier, if the optional argument was given) resulting from the unification (if the unification was successful; else the value of the optional argument is returned).

A binding list (unifier) has the form of an association list:

```
((var val)
 (var val)
 ...)
```

where *val* can also be a variable. In the case of a variable bound to another, the order they occur within a binding makes no difference.

Variables in expressions are represented as atoms whose symbol name begins with "?". For example, `?x` is a variable. Several functions are provided for dealing with variables, for example:

- `(variable? thing)` – returns t if *thing* is a variable
- `(varname var)` – strips the leading "?" from *var*
- `(make-var basename)` – creates a new variable based on *basename* by prepending a "?"
- `(find-binding var bindings)` – returns the binding of the variable in the binding list, plus another value (via Lisp's `values` mechanism) to indicate whether the variable was really bound or not
- `(add-binding var val bindings)` – add a new binding to `bindings`
- `(instantiate thing bindings)` – this will replace all variables in `thing` with their values from bindings; this takes an additional keyword argument; see the documentation for this function (below) for information about how to use it to handle unbound variables.

2 Representing FOPC

You are free to represent predicate calculus expressions any way you like, of course. However, a standard way is to represent them in Lisp is something like the following:

FOPC	Lisp representation
$Human(Socrates)$	`(human socrates)`
$Human(x)$	`(human ?x)`
$Human(x) \wedge Roman(x)$	`(and (human ?x) (roman ?x))`
$Pompeian(x) \vee Roman(x)$	`(or (pompeian ?x) (roman ?x))`
$\neg Pompeian(x)$	`(not (pompeian ?x))`
$Bird(x) \rightarrow Flies(x)$	`(implies (bird ?x) (flies ?x))`
$\exists\ x\ Dog(x) \rightarrow Barks(x)$	`(exists (?x) (implies (dog ?x) barks ?x))`
$\forall\ x\ Human(x) \rightarrow Moral(x)$	`(forall (?x) (implies (human ?x) (mortal ?x)))`

3 Symbol Generation

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:

```
> (gensym)
#:G913
> (gensym "FOO")
```

```
#:FOO914
```

when you might rather have `FOO1` as your first "FOO"-based variable.

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.

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 `*intern-package*`).

4 Code

Some package-related bookkeeping. First tell Lisp that we're in the "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. By default, it's the "User" package.

```
[00001] (defvar *intern-package* 'user
[00002]   "Package in which to intern symbols created by unify, etc., functions.")
[00003]
```

First, set up "?" to be a macro character to allow ?X, etc., to be treated like a variable. (Don't worry about the *var* stuff for now.)

```
[00004] (set-macro-character #\?
[00005]    #'(lambda (stream char)
[00006]         (let ((next-char (peek-char nil stream))
[00007]               next foo)
[00008]           (cond
[00009]            ((equal next-char #\))
[00010]             ;;it's a paren, so it's invalid as a variable...just
[00011]             ;; return symbol ?
[00012]             (setq foo (intern "?" *intern-package*))
[00013]             foo)
[00014]            ((equal next-char #\space)
[00015]             (setq foo (intern "?" *intern-package*))
[00016]             foo)
[00017]            (t
[00018]             (setq next (read stream t nil t))
[00019]             (cond
[00020]              ((atom next)
[00021]               ;;return ?atom
[00022]               (multiple-value-bind (thing dummy)
[00023]                                    (intern  (string-append (string #\?)
[00024]                                                            (symbol-name next)))
[00025]                                    thing))
[00026]              (t
[00027]               `(*var* ,next)))))))
[00028]    t)
[00029]
[00030] (unless (fboundp 'string-append)
[00031]   (defmacro string-append (&rest strings)
[00032]     `(concatenate 'string ,@strings)))
[00033]
[00034]
```

Function (unify p1 p2 {bindings})

This is the primary unify function. `bindings` is optional.

```
[00035] (defun unify (p1 p2 &optional bindings)
[00036]   (cond
[00037]    ((variable? p1)
[00038]     (unify-variable p1 p2 bindings))
[00039]    ((variable? p2)
[00040]     (unify-variable p2 p1 bindings))
[00041]    ((and (atom p1) (atom p2))
[00042]     (unify-atoms p1 p2 bindings))
[00043]    ((and (listp p1) (listp p2))
[00044]     (unify-elements p1 p2 bindings))
[00045]    (t (values nil bindings))))
[00046]
```

Function (unify-atoms p1 p2 bindings)

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.

```
[00047] (defun unify-atoms (p1 p2 bindings)
[00048]   (values (eql p1 p2) bindings))
[00049]
```

Function (unify-elements p1 p2 bindings)

This looks through the elements of two lists, making sure that corresponding elements unify and maintaining appropriate bindings.

```
[00050] (defun unify-elements (p1 p2 bindings)
[00051]   (let (blist matched?)
[00052]     (multiple-value-setq (matched? blist)
[00053]       (unify (first p1) (first p2) bindings))
[00054]     (cond
[00055]      ((null matched?)
[00056]       (values nil bindings))
[00057]      ((multiple-value-setq (matched? blist)
[00058]         (unify (rest p1) (rest p2) blist))
[00059]       (values matched? blist))
[00060]      (t
[00061]       (values nil bindings)))))
[00062]
```

Function (unify-variable p1 p2 bindings)

This unifies a variable (P1) with an arbitrary expression (P2), updating bindings as necessary

```
[00063] (defun unify-variable (p1 p2 bindings)
[00064]   (cond
[00065]    ((eql p1 p2)
[00066]     (values t bindings))
[00067]    (t
[00068]     (let ((binding (find-binding p1 bindings)))
[00069]       (if binding
[00070]           (unify (extract-value binding) p2 bindings)
[00071]         (if (inside? p1 p2 bindings)
[00072]             (values nil bindings)
[00073]             (values t (add-binding p1 p2 bindings))))))))
[00074]
```

Function (find-binding var bindings &optional not-one-of)

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.

```
[00075] (defun find-binding (var bindings &optional not-one-of)
[00076]   (let ((binding
[00077]          (car (member var bindings :test #'(lambda (a b)
[00078]                                              (let ((poss (cond
[00079]                                                           ((eql a (car b))
[00080]                                                            (cadr b))
[00081]                                                           ((eql a (cadr b))
[00082]                                                            (car b)))))
[00083]                                                (when (and poss
[00084]                                                           (not (member poss not-one-of)))
[00085]                                                  t)))))))
[00086]     (cond
[00087]      ((null binding) (values nil nil))
[00088]      ((eql var (car binding))
[00089]       (values binding t))
[00090]      (t (list var
[00091]               (values (car binding) t))))))
[00092]
[00093]
[00094]
[00095]
[00096]
```

Function (extract-value binding)

This just returns the value portion of a binding.

```
[00097] (defun extract-value (binding)
[00098]   (cadr binding))
[00099]
[00100]
```

Function (inside? var expr bindings)

Function (inside-or-equal? var expr bindings)

These together return T if `var` occurs in expression `expr`. Probably `inside-or-equal?` should be eliminated and its code incoprorated into `inside?` via an flet.

```
[00101] (defun inside? (var expr bindings)
[00102]   (if (equal var expr)
[00103]       nil
[00104]       (inside-or-equal? var expr bindings)))
[00105]
[00106] (defun inside-or-equal? (var expr bindings)
[00107]   (cond
[00108]    ((equal var expr) t)
[00109]    ((and (not (variable? expr)) (atom expr)) nil)
```

```
[00110]    ((variable? expr)
[00111]     (let ((binding (find-binding expr bindings)))
[00112]       (when binding
[00113]         (inside-or-equal? var (extract-value binding) bindings))))
[00114]    (t (or (inside-or-equal? var (first expr) bindings)
[00115]           (inside-or-equal? var (rest expr) bindings)))))
[00116]
```

Function (add-binding var val bindings)

This adds a new binding of the form `(var val)` to `bindings`, or creates a new binding list if `bindings` is nil.

It returns a binding list, and is not destructive.

```
[00117] (defun add-binding (var val bindings)
[00118]   (if (eq '_ var)
[00119]       bindings
[00120]       (cons (list var val) bindings)))
[00121]
```

Function (variable? thing)

Returns t if `thing` is a variable, else returns nil.

```
[00122] (defun variable? (thing)
[00123]   (or (and (listp thing)
[00124]            (equal (car thing) '*var*))
[00125]       (and (symbolp thing)
[00126]            (equal (char (symbol-name thing) 0)
[00127]                   #\?))))
[00128]
```

Function (varname var)

Returns the name of the variable `var`.

```
[00129] (defun varname (var)
[00130]   (cond
[00131]     ((and (consp var)
[00132]           (consp (cdr var)))
[00133]      (cadr var))
[00134]     ((equal (char (string var) 0) #\?)
[00135]      (intern  (string-left-trim '(#\?) (string var))
[00136]               (find-package *intern-package*)))))
[00137]
[00138]
[00139]
[00140]
```

Function (make-var var)

Create a new variable whose name is `var`. If `var` is `x1`, for example, the new one will be `?X1`.

```
[00141] (defun make-var (var)
[00142]   (intern (concatenate 'string "?"
[00143]                        (cond
[00144]                         ((stringp var) var)
[00145]                         (t (symbol-name var))))))
[00146]
[00147]
[00148]
[00149]
[00150]
```

Class SymbolGenerator

Holds the state of symbol generation. [Copied from Utilities, where a class was needed; if doing this from scratch for this class, I'd have used just a hash table.]

```
[00151] (defclass SymbolGenerator ()
[00152]   ((counterTable :initform (make-hash-table :test #'equal))))
[00153]
```

Function (newSymbol &optional prefix &key package (intern t))

This returns a unique symbol based on `prefix`, which can be a string or a symbol. `package` controls where the thing is interned, and `intern` determines if it is at all.

```
[00154] (defun newSymbol (&optional prefix &key (package) (intern t))
[00155]   (with-slots (counterTable) *symbolGenerator*
[00156]    (let (num sym)
[00157]      (cond
[00158]        ((symbolp prefix)
[00159]         ;; convert to string, call again:
[00160]         (newSymbol (symbol-name prefix) :package package :intern intern))
[00161]        ((stringp prefix)
[00162]         ;; get rid of trailing numerals:
[00163]         (setq prefix (string-right-trim "0123456789" prefix))
[00164]         ;; try new symbol names until we find one that is not in use:
[00165]         (loop do
[00166]               (cond
[00167]                ((setq num (gethash prefix counterTable))
[00168]                 ;; number exists for this prefix -- new number is just incremented
[00169]                 ;; one:
[00170]                 (setq num (1+ num))
[00171]                 (setf (gethash prefix counterTable) num))
[00172]                (t
[00173]                 ;; no number yet:
[00174]                 (setf (gethash prefix counterTable) 1)
[00175]                 (setq num 1)))
[00176]             until (not (find-symbol
[00177]                         (setq sym (string-append prefix
[00178]                                                  (princ-to-string num))))))
[00179]         ;; found one, create the symbol...
[00180]         (setq sym (make-symbol sym))
[00181]         (when intern                          ;then intern the symbol:
[00182]           (setq sym (if package
[00183]                         (intern (format nil "~a~s" prefix num) package)
[00184]                         (intern (format nil "~a~s" prefix num)))))
[00185]         sym)
```

```
[00186]        (t
[00187]         ;; then can't do any better than regular old gensym:
[00188]         (gensym))))))
[00189]
[00190]
[00191]
```

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`.

```
[00192] (defvar  *SymbolGenerator* (make-instance 'SymbolGenerator))
[00193]
```

Function (instantiate thing bindings &key (if-unbound :first))

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 documentation for `instantiate-variable`.

```
[00194] (defun instantiate (thing bindings &key (if-unbound :first))
[00195]   (cond
[00196]    ((variable? thing)
[00197]     (instantiate-variable thing bindings :if-unbound if-unbound))
[00198]    ((atom thing)
[00199]     thing)
[00200]    (t
[00201]     (cons (instantiate (car thing) bindings :if-unbound if-unbound)
[00202]           (instantiate (cdr thing) bindings :if-unbound if-unbound)))))
[00203]
```

Function (instantiate-variable var bindings &key (if-unbound :first))

This will instantiate a variable against 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, 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 we have these bindings:

```
b = ((?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
```

```
[00204] (defun instantiate-variable (var bindings &key (if-unbound :first))
[00205]   (multiple-value-bind (found val)
[00206]       (inst-var var bindings)
[00207]     (cond
[00208]      (found val)
[00209]      ((eql if-unbound :first)
```

```
[00210]       var)
[00211]      ((eql if-unbound :last)
[00212]       (cadr val))
[00213]      (t
[00214]       if-unbound))))
[00215]
```

`inst-var` is just a helper-function for `instantiate-variable`.

```
[00216] (defun inst-var (var bindings &optional (depth 0))
[00217]   (loop with deeper-var = nil
[00218]       for binding in bindings
[00219]       do
[00220]         (when (member var binding)
[00221]           (let (found
[00222]                 (val (if (eql var (car binding) )
[00223]                          (cadr binding)
[00224]                        (car binding))))
[00225]             (cond
[00226]              ((not (variable? val))
[00227]               (return (values t val)))
[00228]              ((multiple-value-setq (found val)
[00229]                 (inst-var val
[00230]                           (remove binding bindings :test #'equal)
[00231]                           (1+ depth)))
[00232]               (return (values t val)))
[00233]              ((variable? (cadr val))
[00234]               (when (or (null deeper-var)
[00235]                         (> (car val) (car deeper-var)))
[00236]                 (setq deeper-var val))))))
[00237]       finally
[00238]         ;; if we get here, we haven't returned from the things above --
[00239]         ;; meaning we haven't found var in bindings at all!  In this case, we
[00240]         ;; need to return the variable itself as the value, though noting that
[00241]         ;; we haven't found a real binding.
[00242]         (return (values nil
[00243]                           (if (or (null deeper-var)
[00244]                                   (<= (car deeper-var) depth))
[00245]                             (list depth var)
[00246]                           deeper-var)))))
[00247]
[00248]
[00249]
[00250]
[00251]
[00252]
[00253]
[00254]
[00255]
[00256]
```

Index