Deutsches
Forschungszentrum
fur Kunstliche
Intelligenz GmbH

Research
Report
TM-94-04

PAntUDE — An Anti-Unification Algorithm for Expressing Refined Generalizations

Cornelia Fischer

May 1994

Deutsches Forschungszentrum fur Kunstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrucken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341

PAntUDE — An Anti-Unification Algorithm for Expressing Refined Generalizations

Cornelia Fischer

DFKI-TM-94-04

This work has been supported by a grant from The Federal Ministry for Research and Technology (FKZ ITWM-FKZ ITW-8902 C4).

PAntUDE — An Anti-Unification Algorithm for Expressing Refined Generalizations

Cornelia Fischer

DFKI GmbH
Erwin-Schrödinger-Straße 57
Postfach 20 80
D-67608 Kaiserslautern

May 12, 1994

Contents

Abstract

In this paper some improvements for the basic algorithm for anti-unification are presented.
The standard (basic) algorithm for anti-unification still may give too general answers with respect to the intended use of the result. 'Too general' means obtaining an unwanted answer when instantiating some variables of the anti-unification output term. To avoid this, a term sometimes should be generalized by regarding certain (semantic) restrictions. In PAntUDE (partial anti-unification with domains and exclusions) two principal improvements of the basic algorithm were implemented and tested: it can use masks for preventing anti-unification of certain arguments (partial anti-unification); it can also use finite domains for enumerating input-term constants instead of introducing a new (universally quantified) variable, and finite exclusions for specifying forbidden constants (anti-unification with domains and exclusions).

1 Introduction

The concept of anti-unification was introduced by Plotkin [Plotkin, 1971] and explained by M.M. Richter [Richter, 1989]. The name '*anti*-unification' indicates its duality to the standard unification algorithm: given two terms, anti-unification will find the lgg (least general generalization) while unification will find the mgu (most general unifier). The dual operation of generalization and specialization can be seen in figure 1. To have the possibility of regaining the original terms one binding for each original term is created during anti-unification. First of all, the term 'least general generalization' must be defined.

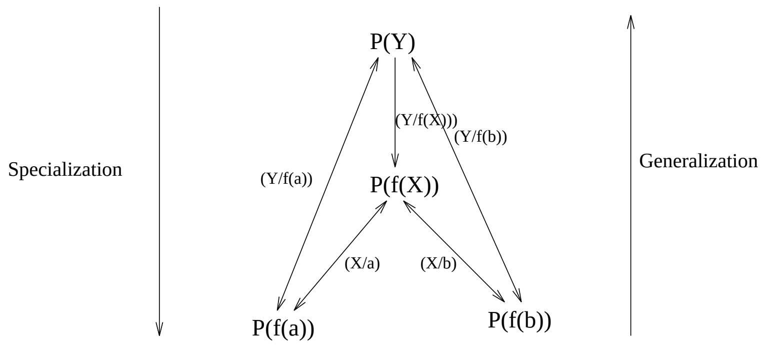

Figure 1: a simple example of specializating/generalizating two terms

It is obvious that a term can be generalized in different ways. The aim now is to find the 'least general generalization' (lgg) of two given terms. The lgg can be seen as the generalization that keeps the anti-unified term t as special as possible so that every other generalization would increase the number of possible instances of t in comparision to the possible instances of the lgg. It is important to find the most specific one, since every instance could be a wrong one in the considered domain. After having realized the importance of the lgg for anti-unification its formal definition will be given:

For two terms ϕ and ψ we say $\phi \succeq \psi$ (spoken ϕ is more general than ψ) $\Longleftrightarrow \exists\sigma$: $\sigma(\phi) \rightarrow \psi$.
Let A_1 ... A_N be some words (literals or terms). The lgg of these terms is the term ψ with $\psi \succeq A_i(1 \succeq i \succeq N)$ and for every other term ϕ with $\phi \succeq A_i$ it is also true that $\psi \succeq \phi$. ψ is now called the *lgg* for A_1 ... A_N. For each A_i there exists a substitution σ_i to transform the lgg back into A_i, i.e. $\phi_i\sigma_i = A_i$

The following example will illustrate the way anti-unification works:

- Input-terms:
  $f(a, g(b, h(X)), c)$
  $f(d, g(j(X), a), c)$

- First-step of anti-unification: take first subterm out off the input-terms, anti-unify them and add the adequate bindings to the lists of bindings:

  $\sigma_1 = ((a/X_1))$
  $\sigma_2 = ((a/X_1)$

- Repeat this basic step of anti-unification until every subterm in the input-terms has been adapted. If necessary use the basic step of anti-unification recursively on subterms.

- Replace every subterm in the first term by its adequate binding. During this step take care that every used substitution is compatible with the substitutions in the second term.

- Result after substitution:
  $lgg = f(X_1, g(X_2, X_3), c)$
  $\sigma_1 = ((a/X_1)(b/X_2)(h(X)/X_3))$
  $\sigma_2 = ((d/X_1)(j(X)/X_2)(a/X_3))$

Anti-unification is an elementary step for quite a lot of learning algorithms. The goal of anti-unification is to detect a least general generalization (lgg) of some given terms or facts. Most of the time, the lgg detected by anti-unification still is too general. E.g. given the two facts `likes(john,mary)` and `likes(peter,jill)` as an input, the lgg of these two facts would be `likes(X,Y)`. It is obvious that this is much too general, but with the standard approach of anti-unification you would not get a more specialized solution.
Another problem is that sometimes your intention is to find out whether a subcase (specified through special values of one or more subterms) of some given terms have something in common. In this case, not all of the given terms should be generalized but only those with a certain value in this very subterm.

Example 1.1 Let the knowledge base contain facts representing information about the atom structure of three elements, carbon, silicon and hydrogen. The first argument of atom is the index number, the second one is the name, the third one contains some properties of this element (metall (m), semimetall (sm), semiconductor (sc), gaseous (g); electron configuration of the last three shells; weight and a list of possible oxidations).

```
   atom(6, name(C, Carbon), prop(sc, eco(2, 4, 0), 12.011, oxi(+2,
+4, -4)))
   atom(14, name(Si, Silicon), prop(sc, eco(2, 8, 4), 28.0855,
oxi(+2, +4, -4)))
   atom(1, name(H, Hydrogen), prop(g, eco(1, 0, 0), 1.00, oxi(+1, -1,
0)))
```

with the knowledge that you want to learn something about elements with the subterm

```
 oxi (+2, +4, -4).
```

These facts would be generalized to
`atom (`X_1`, name (`X_2`, `X_3`), prop (sc, eco (`X_4`, `X_5`, `X_6`), `X_7`, oxi (`X_9, X_{10}, X_{11}`)))`
in spite of loosing most of the interesting facts you were looking for. In PAntUDE we were trying to solve some of these problems. First the algorithm on which the improvements took place will be presented.

2 The Basic Algorithm

The basic algorithm for anti-unification works in quite a simple way: The input are two terms and the algorithm returns a list containing the lgg of the two input-terms and the bindings to transform each input-term into the obtained lgg. In addition to the input-terms there may also be given a list of initial bindings for one or both of the input terms. This is realized with two key parameters (bind1 for the first term and bind2 for the second one). The algorithm can fail only in one case: the top-symbols or the length of the two terms are different. This makes sense since without this restriction you would get just a new variable when anti-unifying the facts of a whole knowledge base and this would be wrong since top-symbols must not be variables but only predicates are allowed. Below two examples (one with key-parameter and one without it) are given to show the way anti-unification works:

Example 2.1 Taking two of the facts from the first example, anti-unification can start:

term1 = `atom(6, name(C,Carbon), prop(sc,eco (2,4,0),12.011,oxi (+2,+4,-4)))`
term2 = `atom(14, name(Si,Silicon), prop(sc,eco(2,8,4),28.0855,oxi(+2,+4,-4)))`
will return the anti-unified term:
`atom(X1, name(X2,X3), prop(sc,eco(2,X4,X5),X6,oxi(2,4,-4)))`

with the substitution (the algorithm represents substitutions as lists of pairs)

bind1: `((12.011 X6) (0 X5) (4 X4) (CARBON X3) (C X2) (6 X1))` for term 1 and
bind2: `((28.0855 X6)) (4 X5) (8 X4) (SILICON X3) (SI X2) (14 X1))` for term 2

Example 2.2 This example will show how given substitutions take place in an anti-unification step:

term1 =`atom(6, name(C,Carbon), prop(sc, eco(2,4,0), 12.011, oxi (+2,+4,-4)))`
term2 = `atom(14, name(Si,Silicon), prop(sc, eco(2,8,4), 28.0855, oxi(+2,+4,-4)))`
with the given initial bindings

bind1=`((C SHORT) (12.011 WEIGHT))`
bind2=`((Silicon CHEMICAL-NAME))`

will return the anti-unified term:

```
atom (X1,name(SHORT,CHEMICAL-NAME), prop(sc,eco(2,X2,X3),WEIGHT,oxi(2,4,-4)))
```

In our implementation of anti-unification you need not always exactly two terms to be generalized but as many terms as you like may be given to the system to be anti-unified. The algorithm will anti-unify all terms matching in the top-symbol and return a list with the obtained lggs. So anti-unification makes obvious, which characteristic items two or more terms have in common. In the example given above the lgg showed us of course the formal structure but also that Carbon and Silicon have the same oxidation-numbers and an equal number of electrons on the first orbit. The bindings which may be added to the input can help to ease the understanding for the reader.

3 Partial Anti-Unification

In the following I will present a method improving the standard algorithm of section 2. It was developed in the KES (Knowledge Evolution System) group of the VEGA (Validation and Exploration with Global Analys) project at the DFKI Kaiserslautern based on ideas by Knut Hinkelmann. To obtain new interesting facts out of the knowledge base, it is more important to see which parts of the two terms did not change. For example - if we want to learn something about chemical elements - we could ask what all elements with the oxidation number 2 4 -4 have in common. For this purpose we can not use a standard anti-unification because this would build the lgg of all given elements and so give unwanted (too general) answers. So we need some tools to inhibit the anti-unification with elements that don't have the desired values. Thus the algorithm has an extra key-parameter: 'keep'. In this parameter, every position in the input terms you do not want to be changed are marked by an special sign. An anti-unification can only take place, if the two terms are unifiable in this very (marked) positions.

Example 3.1 This example will show the usefulness of partial anti-unification for finding interessting knowledge

- Input:
  ```
  atom(26, name(Fe, Ferrum), prop(m, eco(8, 14, 2), 55.847, oxi(+2,
  +3)))
  atom(62, name(Sm, Samarium), prop(m, eco(24, 8, 2), 150.36,
  oxi(+2, +3)))
  atom(77, name(Ir, Iridium), prop(m, eco(32, 15, 2), 192.22,
  oxi(+3, +4)))
  atom(58, name(Ce, Cerium), prop(m, eco(20, 8, 2), 140.115,
  oxi(+3, +4)))
  atom(76, name(Os, Osmium), prop(m, eco(32, 14, 2), 190.2, oxi(+3,
  +4)))
  atom(1, name(H, Hydrogen), prop(g, eco(1, 0, 0), 1.00, oxi(+1,
  -1)))
  ```

- Partial anti-unification of these facts with the keep-parameter

  ```
   keep (- - - - $)
  ```

  will result in the following facts:

  $atom(X_1, name(X_2, X_3), prop(m, eco(X_4, X_5, 2), X_6, oxi(+2, +3)))$

  $atom(Y_1, name(Y_2, Y_3), prop(m, eco(Y_4, Y_5, 2), Y_6, oxi(+3, +4)))$

  $atom(1, name(H, Hydrogen), prop(g, eco(1, 0, 0), 1.00, oxi(+1, -1)))$

- The resulting information can be interpreted in the following way: Elements with the same oxidation behaviour seem to have to same outer appeareance (metal). In addition the number of electrons on the last shell is equal. If instead of parial anti-unification the basic anti-unification algorithm would have been used on these facts, the result would be $atom(X_1, name(X_2, X_3), prop(X_4, eco(X_5, X_6, X_7), X_8, oxi(X_9, X_{10})))$.
  This would not have been useful to gain any new knowledge.

4 Anti-Unification with Finite Domains and Finite Exclusions

In this version of anti-unification - introduced without patialness in [Boley, 1994] - domains and exclusions are used instead of variables. A domain is a term listing all possible values a correspoding free variable should take, and an exclusion (the 'negation' of a domain) lists all values such a variable is not allowed to take. In the representation language RELFUN finite domains and exclusions are treated as first-class citizens. Most of the following examples are taken out of the aboved-mentioned paper.
When anti-unifying two terms with different constants in corresponding positions, PAntUDE yields a `dom` term containing these constants, not a (sometimes too general) new variable. For a constant and a structure it has to yield a new variable since current `dom` terms cannot contain structures. Generally (constants can be treated as singleton domains), *domain anti-unification* of two `dom` terms yields their union (unification: intersection). Identical `dom` terms can directly yield one copy unchanged, short-cutting spurious unions.

The complementary *exclusion anti-unification* for a variable and an exclusion yields a variable in the manner classic anti-unification handles variable/constant pairings. It yields the intersection (unification: union) of two `exc` terms. For an exclusion and a constant (singleton domain) it yields the `exc` term minus the constant.

4.1 Anti-unification with finite domains and finite exclusions

Generally, the *domain-exclusion anti-unification* of a `dom` and an `exc` term, in any order, yields the `exc` term with the elements of the `dom` term set-theoretically subtracted (unification: domain with exclusion subtracted). An empty-exclusion outcome, as usual, represents an always successful new variable. Altogether, the domain/exclusion complementarity commutes nicely with the unification/anti-unification duality.

The example below will help to understand the way anti-unification with domains works: Given the 24 relational facts for the predicate `separates`:

```
separates(pacific,canada,japan).
separates(pacific,mexico,japan).
separates(pacific,usa,japan).
separates(atlantic,canada,denmark).
separates(atlantic,canada,france).
separates(atlantic,canada,germany).
separates(atlantic,canada,italy).
separates(atlantic,canada,spain).
separates(atlantic,canada,sweden).
separates(atlantic,canada,uk).
separates(atlantic,mexico,denmark).
separates(atlantic,mexico,france).
separates(atlantic,mexico,germany).
separates(atlantic,mexico,italy).
separates(atlantic,mexico,spain).
separates(atlantic,mexico,sweden).
separates(atlantic,mexico,uk).
separates(atlantic,usa,denmark).
separates(atlantic,usa,france).
separates(atlantic,usa,germany).
separates(atlantic,usa,italy).
separates(atlantic,usa,spain).
separates(atlantic,usa,sweden).
separates(atlantic,usa,uk).
```

A simple method for (least general) generalization of these facts is pairwise *domain anti-unification* of the input facts. Thus we get the result

```
separates(dom[pacific,atlantic],dom[canada,mexico,usa],
 dom[denmark,france,germany,italy,spain,sweden,uk,japan]).
```

This fact is more specific than the term *seperates (X,Y,Z)* resulting from the basic anti-unification algorithm. However this can still be too general. Using our knowledge from geography, we can see that it represents a number of wrong facts. For example, the pacific doesn't intersect Canada from Denmark. These wrong conclusions can be extracted out of the anti-unified term by combining domain-exclusion anti-unification with partial antiunification.

An example of exclusion anti-unification can take two versions of a fact as input:

```
likes(X,exc[mary,claire,linda]).  % Everybody likes all except MCL
likes(john,exc[mary,tina]).    % John likes all except Mary & Tina
```

Anti-unification generalizes them via an intersection of the exclusions in the second argument:

```
likes(X,exc[mary]).              % Everybody likes all except Mary
```

This is the least general generalization of the input facts since exactly the subexclusion common to both facts is kept. In cases where we have a closed universe, say $\{ann, claire, john, linda, mary, peggy, susan, tina\}$ the inputs can be rewritten as complementary domain facts:

```
likes(X,dom[ann,john,peggy,susan,tina]).   %  (*)
likes(john,dom[ann,claire,john,linda,peggy,susan]).
```

Domain anti-unification via union generalizes them to

```
likes(X,dom[ann,claire,john,linda,peggy,susan,tina]).
```

which is the complement of the exclusion-anti-unification result above. Finally, domain-exclusion anti-unification of the input facts

```
likes(X,exc[mary,claire,linda]).
likes(john,dom[mary,tina]).   %  (**)
```

via subtraction generalizes them to

```
likes(X,exc[claire,linda]).
```

Here, the exclusion is minimally weakened (its extension being minimally enlarged) to accomodate what is specified by the domain. This can again be illustrated for the case of a closed universe: anti-unify `(*)` with `(**)` and re-complement the result. Such least general generalizations by domain-exclusion anti-unification thus remove `dom-exc` contradictions in a set of clauses, e.g. about John's liking of Mary in the above input facts; similarly, exclusion anti-unification removes the less obvious `exc-exc` contradictions concerning constants that occur in only one of the exclusions, e.g. about John's liking of, say Claire, in the previous input facts. This may be exploited for 'theory revision' of knowledge bases containing exclusion terms.

4.2 Partial anti-unification with finite domains and finite exclusions

Using the keep-operator described in Section 3 on the first term of the above example of input terms, these facts can be generalized to

```
separates(pacific,dom[canada,mexico,usa],japan).
separates(atlantic,dom[canada,mexico,usa],
 dom[denmark,france,germany,italy,spain,sweden,uk]).
```

It is easy to see that this isn't a generalization but only a compression of the facts, since there can't be extracted any new information out of the new facts. This occurs because all countries separated from another country by the same ocean are always identical. So let us add just one new fact to our little knowledge base:

```
separates(atlantic,panama,denmark).
```

Using PAntUDE on the facts above together with this new fact we will get a real generalization:

```
separates(pacific,dom[canada,mexico,usa],japan).
separates(atlantic,dom[canada,mexico,usa,panama],
 dom[denmark,france,germany,italy,spain,sweden,uk]).
```

This generalized `atlantic` fact expresses more information than the input facts, namely an induction from Denmark to the other European countries (which happens to be empirically true); again multiplying out the result makes these induced facts explicit:

```
separates(atlantic,panama,france).
. . .
separates(atlantic,panama,uk).
```

It will need quite a lot of domain knowledge to decide which parameters to keep and which are allowed to be generalized.

5 Conclusion

Several tests with PAntUDE were made on realistic sample knowledge bases, including fact bases on chemical properties of certain materials. The entire implementation is realized in Common Lisp (lucid-4-1) on sun-workstations. The complete system of PAntuDE is listed in the appendix and also available on ftp.

A Listing of PAntUDE

```
;;; This is an complete version containing all the implemented improvements
;;; of the basic anti-unification algorithm for
;;; generalizing two terms. The form of variables is the one used in
;;; the Colab-system. February 1994.

;;;; Copyright (c) 1994 by Cornelia Fischer. This program may be freely
;;;; copied, used, or modified provided that this copyright notice is included
;;;; in each copy of this code and parts thereof.

;;;; This version of anti-unification takes a complete KB of facts as input
;;;; and returns
;;;; a list containing the anti-unification found for the given terms.
;;;; In addition to the basic algorithm you may add a keep-operator to the
;;;; arguments. this operator makes sure, that the two terms have to match in
;;;; at this very argument position. For example the two terms f(a b) f(c d)
;;;; used with the keep-operator f($ -) wouldn't be anti-unified, because a
;;;; and b don't match. In addition the algorithm don't care, if the length
;;;; of the two terms is equal: the shorter term will be enlarged with the
;;;; missing elements from the other term before anti-unification takes place.

(SETQ *PRINT-PRETTY* T)


(defun Pantude (WB_list &optional (keep nil) &key (toprint nil))
  (remove-duplicates WB_list :test #'equal)
  (setq erg nil)
       (do* (
             (lga_erg (car WB_list) (car not_antiunified))
             (not_antiunified (cdr WB_list) (cdr not_antiunified)))
           ((null not_antiunified))
           (let ((bind1 nil)
                 (bind2 nil)
                 (liste nil))
                (do ((nau not_antiunified (cdr nau)))
                    ((null nau))
                    (let ((erg (lga lga_erg (car nau) keep bind1 bind2 toprint)))
                         (if (car erg) ;antiunification sucessfull
                             (setq lga_erg (car erg) bind1 (cadr erg)
                                   bind2 (caddr erg))
                             (if liste
                                 (setq liste (append liste (list(car nau))))
                                 (setq liste (list (car nau)))
                             )
                         )
```

```
                      )
                   )
                   (setq not_antiunified liste)
            )
            (if erg
                (setq erg (append erg (list lga_erg)))
                (setq erg (list lga_erg))
            )
      )
  (format t "~%~%
   ***************************************************************************~%
   ***************************************************************************~%
             Ergebnis: ~s~%
   ***************************************************************************~%
   ***************************************************************************~%" erg)
erg
)


(defun dom-t (x) (and (consp x) (eq 'dom (car x))))

(defun exc-t (x) (and (consp x) (eq 'exc (car x))))

(defun exc-intersection (x y)
  (mk-exc (intersection (cdr x) (cdr y) :test #'equal)))

(defun dom-union (x y)
  (mk-dom (union (cdr x) (cdr y) :test #'equal)))

(defun exc-dom (x y)
  (mk-exc (set-difference (cdr x) (cdr y) :test #'equal)))

(defun mk-dom (elist)
  (cond ((null elist) nil)
        ((null (cdr elist)) (car elist))
        (t (cons 'dom elist))))

(defun mk-exc
       (elist)
  (cond ((null elist) 'id)
        (t (cons 'exc elist))))


(defun vari-t (x)  ; test if x is a variable
   (AND (consp x) (equal (car x) 'vari)))

(defun dom-anti-unify1 (a b new_term)
   ;;; Returns a new_term which anti-unifies a & b
   (cond ((OR (eql a b)
               (vari-t a)
               (vari-t b))
           (if (vari-t a)
               (list new_term a)
               (list new_term b)
           )
          )
          ((AND (atom a)(atom b))  ; two different constants appear. Create a new domain
               (list new_term (mk-dom (list a b)))
          )
          ((dom-t a)
           (cond ((dom-t b) (list new_term (dom-union a b)))
                 ((exc-t b) (list new_term (exc-dom b a)))
                 (t (list new_term (dom-union a (list 'dom b)))) ; add b to domain
           )
          )
          ((exc-t a)
           (cond ((dom-t b) (list new_term (exc-dom a b)))
                 ((exc-t b) (list new_term(exc-intersection b a)))
                 (t (list new_term(exc-dom a (list 'dom b)))) ; restrict b from exception
           )
          )
          ((dom-t b)
           (list new_term(dom-union b (list 'dom a)))  ; add a to domain
          )
          ((exc-t b)
           (list new_term(exc-dom a (list 'dom b)))    ; restrict a from exceptions
          )
          ((not (equal (first a) (first b)))  ; two different function-symbols appear. Cr
          (list new_term (mk-dom (list a b)))
          )
          (t ;increase depth of anti-unification
             (do ((i 1 (+ i 1)))
                 ((> i (length (rest a))))
                 (setq new (first a))
                  (if (> i 1)
                      (setq new (anti-unify2 (nth i a) (nth i b) new))
                      (setq new (anti-unify1 (nth i a) (nth i b) new)))
             )
             (list new_term new)
          )
   )
 )
```

```
(defun dom-anti-unify2 (a b new_term)
  ;;; Returns a new term which anti-unifies a & b
  (cond ((OR (eql a b)
             (vari-t a)
             (vari-t b))
         (if (vari-t a)
             (append new_term (list a))
             (append new_term (list b))
         )
        )
        ((AND (atom a)(atom b))  ; two different constants appear. Create a new domain
             (append new_term (list (mk-dom (list a b))))
        )
        ((dom-t a)
         (cond ((dom-t b) (append new_term (list (dom-union a b))))
               ((exc-t b) (append new_term (list (exc-dom b a))))
               (t (append new_term (list (dom-union a (list 'dom b))))) ; add b to doma
         )
        )
        ((exc-t a)
         (cond ((dom-t b) (append new_term (list (exc-dom a b))))
               ((exc-t b) (append new_term (list (exc-intersection b a))))
               (t (append new_term(exc-dom a (list 'dom b)))) ; restrict b from excepti
         )
        )
        ((dom-t b)
         (append new_term(dom-union b (list 'dom a)))  ; add a to domain
        )
        ((exc-t b)
         (append new_term(exc-dom a (list 'dom b)))    ; restrict a from exceptions
        )
        ((not (equal (first a) (first b)))  ; two different function-symbols appear. Cr
        (append new_term (list (mk-dom (list a b))))
        )
        (t ;increase depth of anti-unification
           (do ((i 1 (+ i 1)))
               ((> i (length (rest a))))
               (setq new (first a))
                (if (> i 1)
                    (setq new (anti-unify2 (nth i a) (nth i b) new))
                    (setq new (anti-unify1 (nth i a) (nth i b) new)))
           )
        (append new_term new)
   ))
```

```
)



(defvar *use-gensyms* t)  ; Uses gensym to create unique variables if T
                          ; otherwise uses copy-symbol

(defun ajust_term (big small &aux term)
 (setq term big)
 (do ((i 0 (+ i 1)))
     ((>= i (length small)))
     (setq term (substitute (nth i small) (nth i term) term :start i))

 )
term
)

(defun lga (term1 term2 keep bind1 bind2 toprint)
  (if (not (equal (car bind1) 't)) ;add t at the beginning of bind1
      (setq bind1 (append '(t) bind1))
  )
  (if (not (equal (car bind2) 't)) ;add t at the beginning of bind2
      (setq bind2 (append '(t) bind2))
  )

  (if (not (= (length term1)(length term2)))
       ;; adjust length of terms
      (if (> (length term1) (length term2))
          (setq term2 (ajust_term term1 term2))
          (setq term1 (ajust_term term2 term1))
      )
  )
  (cond ((equal term1 term2)
         (if toprint
             (format t "~%anti-unified term: ~s ~%~%" term1)
         )
        )
        (keep
           (keep-lga term1 term2 (search-matching-keep term1 keep) bind2 bind1 toprint)
        )
        (t (keep-lga term1 term2 nil bind1 bind2 toprint))
  )
)


(defun keep-lga (term1 term2 keep bind1 bind2 toprint &aux new_term (error nil))
(if toprint
  (format t "~%~% *********************************************~%term1: ~s~%term2: ~s
)
  (if (equal (first term1)(first term2))
    (let* ((new_term (first term1)))
      (do* ((l 1 (+ l 1))
            (keep? keep (cdr keep?))
            (act (first keep?)(first keep?))
            (liste1 (cdr term1)(cdr liste1))
            (liste2 (cdr term2)(cdr liste2)))
           ((OR (>= l (length term1)) error))
           (cond ((equal act '$)
                  (if (not (equal (car liste1) (car liste2)))
                      (setq error t)
                      (if (= l 1)
                          (setq new_term (list new_term (car liste1)))
                          (setq new_term (append new_term (list (car liste1))))
                      )
                  )
                 )
                  ((OR (equal act 'V)(equal act 'v))
                     (if (= l 1)
                          (setq new_term (list new_term (car liste1)))
                          (setq new_term (append new_term (list (car liste1))))
                     )
                    (setq bindings
                            (anti-unify1 term1 term2 term1 term2 bind1 bind2)
                           bind1 (cdar bindings) bind2 (cdadr bindings)
                    )
                  )
                  (t
                  (if (> l 1)
                      (setq new_term
                        (dom-anti-unify2 (nth l term1) (nth l term2) new_term))
                      (setq new_term
                        (dom-anti-unify1 (nth l term1) (nth l term2) new_term))
                  )
                  )
           )
      )
      (if error
          (progn
            (if toprint
               (format t "keine Anti-Unifitation moeglich, da nicht mit keep-Forderung
            )
            (list nil term1 term2)
          )
          (progn
                (let ((bindings bind2))
                ;;; replace every expression in term1 by its more generell expression c
                     (do* ((expr-list (car bindings) (cadr bind-2))
                           (bind-2 bindings (cdr bind-2)))
                          ((null bind-2))
                          (if (listp expr-list)
                              (let ((new-expr (cadr expr-list))
                                    (old-expr (car expr-list))
                                   )
                             ; search in term1 for expression old-expr
                             ; and replace it by new-expr
                            (setq new_term
                                      (my_substitute new-expr old-expr new_term term2 b
                                 )
                              )
                          )
                     )
               )
               (if toprint
                  (format t "~%anti-unified term: ~s ~%~%" new_term)
               )
               (list new_term bind2 bind1)
          )
      )
     )
     (progn
       (if toprint
         (format t "keine Anti-Unifitation moeglich !!!!")
       )
       (list nil term1 term2)
     )
    )
)


(defun anti-unify1 (a b term1 term2 bind1 bind2)
  ;;; Returns a most general binding list which anti-unifies a & b
  (cond ((eql a b) (list bind1 bind2))
        ((vari-t a) (make-new a b bind1 bind2 0 term1 term2)
                    (list bind1 bind2))
        ((vari-t b) (make-new b a bind1 bind2 1 term1 term2)
                    (list bind1 bind2))
        ((or (atom a)(atom b))   ;create new variable and add to bindings
         (let* ((old (find-binding a (rest bind2)))
                (old2 (find-binding b (rest bind1)))
                (new (if old old
                             (if old2 old2
                                (list 'vari (gentemp))
                              )
                      )
                 )
                )
               (add-binding new a b bind2 bind1)
               (add-binding new b a bind1 bind2)
          )
          (list bind1 bind2)
         )
      ;; create new variable and add it to bindings
      ;; if two different function-symbols appear
         ((not (equal (first a) (first b)))
          (let* ((old (find-binding a (rest bind2)))
                 (old2 (find-binding b (rest bind1)))
                 (new (if old old
                              (if old2 old2
                                       (list 'vari (gentemp))
                              )
                      )
                 )
                )
               (add-binding new a b bind2 bind1)
               (add-binding new b a bind1 bind2)
          )
          (list bind1 bind2)
         )
         (t ;increase depth of anti-unification
            (do ((i 1 (+ i 1)))
                ((> i (length (rest a))))
                 (let ((bindings
                       (anti-unify1 (nth i a) (nth i b) term1 term2 bind1 bind2)))
                      (setq bind1 (car bindings))
                      (setq bind2 (cadr bindings))
                 )
            )
            (list bind1 bind2)
   ))
)

(defun make-new (var b bind1 bind2 order term1 term2)
  ;;; Unify a variable with an other term
  (if (and (vari-t b) (var-eq var b)) ;both variables are equal,
                     ;  this doesn't change anything in the bindings
      (list bind1 bind2)
      (if (OR (vari-t b)
               ; to make sure that there is no conflict of variables
```

```
                  ; with same name and different bindings
                  (AND (>(my_count var term1) 1) (= order 0))
                  (AND (>(my_count var term2) 1) (= order 1))
             )
             (let (( new (list 'vari (gentemp))))
                 (if (= order 1)
                     (progn
                           (add-binding new var b bind1 bind2)
                           (add-binding new b var bind2 bind1)
                     )
                     (progn
                           (add-binding new var b bind2 bind1)
                           (add-binding new b var bind1 bind2)
                     )
                 )
             )
             (if (= order 1)
                 (progn
                       (add-binding var b var bind2 bind1)
                       (add-binding var var var bind1 bind2)
                 )
                 (progn
                       (add-binding var b var bind1 bind2)
                       (add-binding var var var bind2 bind1)
                 )
             )
          )
     )
)


(defun var-eq (var1 var2)
  ;;; Return T if the two variables are equal
   (eql var1 var2))

(defun get-binding (var bindings)
  ;;; Get the variable binding for var
   (setq gefunden nil)
   (setq erg nil)
   (do* ((liste (rest bindings) (cdr liste))
         (aktuell (car liste) (car liste)))
        ((OR (not liste) gefunden))
        (if (var-eq var (cadr aktuell))
            (progn (setq gefunden t)
                   (setq erg  aktuell)
            )
        )
   )
   erg
)

(defun add-binding (var val val3 bindings bind2)
  ;;; Add the binding of var to val to the existing set of bindings
  (setq test (find-binding val (rest bindings)))
  (if (not (AND test
                (find-binding2 test val3 (cdr bind2))
           )
      )
      (setf (rest bindings) (cons (list val var) (rest bindings)))
  )
  bindings)

(defun find-binding (val bindings &optional (erg nil) (gefunden nil))
  (do* ((liste bindings (cdr liste))
        (aktuell (car liste) (car liste)))
       ((OR gefunden (not liste)))
       (if (OR (AND (vari-t val)
                    (vari-t (car aktuell))
                    (var-eq val (car aktuell)))
               (equal val (car aktuell))
           )
           (progn (setq gefunden t)
                  (setq erg (cadr aktuell))
           )
       )
  )
  erg
)


(defun find-binding2 (val val_old bindings &optional (gefunden nil))
  (do* ((liste bindings (cdr liste))
        (aktuell (car liste) (car liste)))
       ((OR gefunden (not liste)))
       (if (OR (AND (vari-t val)
                    (vari-t (cadr aktuell))
                    (equal val_old (car aktuell))
                    (var-eq val (cadr aktuell)))
               (AND (equal val (cadr aktuell))
                    (equal val_old (car aktuell))
               )
           )
           (setq gefunden t)
       )
  )
  gefunden
)




(defun my_substitute (new old term term2 bind &aux new_term)
  ;; substitute every appereance of old at actual top-level by new

(do ((i 0 (+ i 1)))
    ((>= i (length term)))
    ;; make sure that actual position may be replaced by new
    (if (AND (listp term) (listp term2)
             (NOT (equal (nth i term) (nth i term2)))
                ; the actual terms are not equal, so they have to be replaced
             (equal (cadr (assoc (nth i term2) bind :test 'equal)) new)
                ; found corresponding binding in bind
        )
     (progn
        (if (equal (nth i term) old)
            ; if actual element is equal the element to be replaced, replace it in term
                  (nsubstitute new old term :test 'equal :count 1 :start i)
        )
     )
    (if (listp (nth i term))
        (nsubstitute (my_substitute new old (nth i term)(nth i term2) bind)(nth i term)
    )
  )
)
term
)

(defun my_count (item term &aux (counter 0))
;;; counts how often item appears in term (with all subterms)

(if (AND (listp term)(NOT (equal (car term) 'vari)))
  (do ((i 0 (+ i 1)))
      ((>= i (length term)))
      (if (equal item (nth i term))
          (setq counter (+ 1 counter))
          (setq counter (+ counter (my_count item (nth i term))))
      )
  )
)
counter
)
```

```
(defun search-matching-keep (term keep)
(setq res nil)
  (do* ((l keep (cdr l))
         (keep? (car l) (car l)))
        ((null l))
        (if (equal (car keep?)(car term))
            (setq res (cdr keep?))
        )
  )
res
)
```

References

[Boley, 1994] Harold Boley. Finite domains and exclusions as first-class citizens. In Roy Dyckhoff, editor, *Fourth International Workshop on Extensions of Logic Programming, St.Andrews, Scotland, March 1993, Preprints and Proceedings.* Springer, 1994. Also available as Research Report RR-94-07, Feb. 1994, DFKI, P.$\mathcal{O}$. Box, D-67608 Kaiserslautern.

[Plotkin, 1971] G. D. Plotkin. A further note on inductive generalization. In D.Michie B. Meltzer, editor, *Machine Intelligence 6*, pages 101 – 124. Elsevier North-Holland, New York, 1971.

[Richter, 1989] Michael M. Richter. *Prinzipien der Künstlichen Intelligenz.* Teubner Verlag, 1989.