

Formal verification of a union-find based unification algorithm

Diploma thesis by Guido Vogt

Guidance by Dr. R. Stärk



Institute of Informatics
University of Fribourg, Switzerland

May 1998

Contents

CONTENTS	1
PREFACE	2
1 INTRODUCTION	3
1.1 What is LPTP?.....	3
1.2 Program Verification	3
1.3 Problem description	4
2 UNIFICATION	5
2.1 The notion "Term"	5
2.2 The notion "Substitution"	5
2.3 The notion "Unification"	6
2.4 A simple unification algorithm	6
2.4.1 Example.....	8
2.5 Cycle test	10
3 SPECIFICATION OF THE UNIFICATION ALGORITHM	12
3.1 Specification	12
3.2 Implementation.....	16
4 CORRECTNESS PROOF	20
4.1 An introductory example.....	20
4.2 Some important propositions.....	22
4.2.1 Theorems about "unifiable_terms"	22
4.2.2 Theorems about "unify_terms_part"	22
4.2.3 Theorems about "unify_terms_sub"	23
4.2.4 Definition of "application"	23
4.2.5 Definition of "composition"	23
4.2.6 Termination of the cycle test.....	24
4.2.7 Termination of the union-find algorithm	24
4.2.8 Most general substitution.....	24
APPENDIX A BIBLIOGRAPHY	25
APPENDIX B DISK FILES	26
APPENDIX C FORMAL PROOF	27

Preface

This diploma thesis was written for a diploma in computer science at the Institute of Informatics at the University of Fribourg in Switzerland during the terms of 1997 and 1998.

The main part of the work consists of on-line proving and thus deriving the full correctness proof for the unification algorithm used by LPTP (Logic Program Theorem Prover by R. Stärk). This proof comprises about 13,000 lines.

Thanks go to Dr. R. Stärk for letting me have an inside look at his Logic Program Theorem Prover and for many hours of his support.

Fribourg, May 1998

Guido Vogt

Chapter 1

Introduction

1.1 What is LPTP?

LPTP is an interactive theorem prover for the formal verification of Prolog programs. It is a proof refinement system allowing the user to interactively construct formal proofs. Using the LPTP system, it is possible to generate proofs *deductively* from the assumption to the goal or *goal directed* from the goal backwards to the axioms. LPTP has the ability to automatically search for proofs or parts of proofs. In the simplest case, LPTP just finds the name of a lemma that has been proved already and that is used at a certain point in a proof. In the best case, LPTP finds complete proofs.

LPTP has been designed for correctness proofs of pure Prolog programs. Pure Prolog programs may contain *negation*, *if-then-else* and built-in predicates like *is/2*, *</2* and *call/n + 1*. The programs, however, have to be free of *cut* and database predicates like *assert/1* and *retract/1* which allows program modification during runtime.

The kernel of LPTP is written exactly in the fragment of Prolog that can be treated in LPTP. This means that LPTP uses no single *cut*. Moreover, it is possible to prove properties of LPTP in LPTP itself.

1.2 Program Verification

Program verification is a formal activity aiming for proof of program correctness. The usual way of verifying program correctness is to check that it meets its specifications.

An important notion is *termination*. A Prolog program may or may not *terminate*, which means that it "produces a finite or infinite number of answers" respectively. In the first case, the program may either *succeed* or *fail*, meaning that the given sequence of predicates is met (and thus resulting in the answer "yes") or is not met (resulting in the answer "no"). In the case of non-termination, no statement can be made. It is therefore essential to verify, that each Prolog function (a part of an entire program) implemented, returns only a finite number of responses and thus terminates.

With the three basic notions from above (*termination*, *success* and *failure*), it is possible to verify Prolog programs in LPTP, which meet the restrictions imposed by LPTP described in the previous section.

1.3 Problem description

As seen, LPTP is written in the subset of Prolog that can be processed by LPTP itself, which permits to prove, in principle, the correctness of the system as a whole.

The goal of this diploma thesis is to formally prove, by means of LPTP, the correctness of the union-find based unification algorithm used internally by LPTP and described in chapter 2.

Chapter 2

Unification

2.1 The notion "Term"

In this diploma thesis we will repeatedly refer to the notion of a *term*, so there is a need to clearly define this expression. *Terms* are defined recursively as follows:

- a variable is a term,
- if f is an n -ary function symbol and t_1, \dots, t_n are terms, then $f(t_1, \dots, t_n)$ is a term.

In particular, every constant is a term. A term with no variables is called *ground*. A *subterm* of a term s is a substring of s which in turn is a term. If w is a subterm of s , then w *occurs* in s . In general, there can be several occurrences of a given subterm in a term - for example $f(x)$ and $g(f(x), f(x))$. By definition, every term is a subterm of itself. A subterm s of a term t is called *proper* if $s \neq t$.

It is convenient to view terms as trees. The tree associated with a variable x has just one node, labeled by x itself. The tree associated with $f(t_1, \dots, t_n)$ is obtained by attaching the trees associated with t_1, \dots, t_n , under the root labeled by f . In particular, the tree associated with a constant c has one node, labeled c .

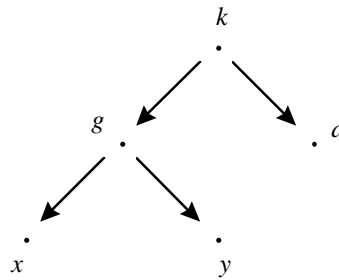


Figure 2.1 The tree associated with the term $k(g(x,y),a)$

2.2 The notion "Substitution"

Substitutions bind variables to terms. A *substitution* is a finite mapping from variables to terms which assigns a term t different from x to each variable x in its domain. This is written as

$$\{x_1/t_1, \dots, x_n/t_n\}$$

where

- x_1, \dots, x_n are different variables,
- t_1, \dots, t_n are terms,
- for $i \in [1, n]$, $x_i \neq t_i$.

Informally, it can be read as: "The variables x_1, \dots, x_n are *bound to* t_1, \dots, t_n , respectively". Example: By applying the substitution $\{y/b\}$ to the term $f(a, y)$, the term $f(a, b)$, with a, b constants and y a variable is obtained.

2.3 The notion "Unification"

Unification is an important notion in Prolog programming. Informally, unification is the process of making terms identical by means of certain substitutions. For example, the terms $f(a, y, z)$ and $f(x, b, z)$, with a, b constants and x, y, z variables, can be made identical by applying to them the substitution $\{x/a, y/b\}$: both sides then become $f(a, b, z)$. However the substitution $\{x/a, y/b, z/a\}$ also makes these two terms identical. Such substitutions are called *unifiers*. The first unifier is to be preferred, because it is "more general", which means that it does not make unnecessary bindings. The second unifier is a "special case" of the first one. More precisely, the first unifier is a *most general unifier* (*mg*u in short) of $f(a, y, z)$ and $f(x, b, z)$ while $\{x/a, y/b, z/a\}$ is not. Therefore, the aim of the unification problem is to answer the question whether for two given input terms there is a substitution, mapping the input terms into the same term. In case of a positive answer, the input terms are said to be *unified* and the appropriate substitution is called a *unifier*.

There are two reasons why two terms may not be unifiable. The first one is illustrated by the following example: $f(g(x, a), z)$ and $f(g(x, b), a)$ are not unifiable, since two constants cannot unify. The second one can be shown when considering the terms $g(x, a)$ and $g(f(x), a)$. No term x can be unified with another term containing x . This is known as the *occur-check* failure.

The *unification problem* is solved by providing an algorithm that

- a) terminates with failure if the terms are not unifiable,
- b) produces a most general unifier if the terms are unifiable.

2.4 A simple unification algorithm

Prolog, as well as many automated theorem provers, uses Robinson's unification algorithm shown in [6]. Robinson's algorithm is easy to implement and efficient enough for terms of small sizes. In the worst case, however, Robinson's unification algorithm is exponential, even if terms are represented by directed acyclic graphs (*dags* in short).

Example: To unify the following equations, most Prolog systems need exponential time.

$$\begin{aligned} ?- \quad & X_0 = f(X_1, X_1), \quad X_1 = f(X_2, X_2), \quad \dots, \quad X_{\{n-1\}} = f(X_n, X_n), \\ & Y_0 = f(Y_1, Y_1), \quad Y_1 = f(Y_2, Y_2), \quad \dots, \quad Y_{\{n-1\}} = f(Y_n, Y_n), \\ & X_0 = Y_0. \end{aligned}$$

It is well known, that the unification problem is polynomial. There exist several fast unification algorithms which are quasi-linear in the worst case (e.g. [3], [4], [7]). LPTP implements one of the fast algorithms. LPTP uses Purdom's simple unification algorithm [5]. As many fast unification algorithms, Purdom's algorithm is based on R.E. Tarjan's results for disjoint-set data structures (union-find).

Definition: A unification graph is a structure $G = (N, F, L, S, P, \Lambda)$ such that

$N \neq \emptyset$,

[N is the set of nodes.]

$\Lambda \notin N$, [Λ is the element "nil".]
 $F \subseteq N$, [F is the set of functional nodes.]
 $L: F \rightarrow \text{Symbols}$, [L is the labeling function. It assigns function symbols to functional nodes.]
 $S: F \rightarrow N^*$, [S assigns a list of argument nodes to each functional node.]
 $P: N \rightarrow N \cup \{\Lambda\}$, [P is the parent function. Equivalence classes are trees.]
 $\forall x \in F (P(x) \neq \Lambda \Rightarrow P(x) \in F)$, [Non-leaf nodes are functional.]
 $\forall x \in N \exists n \in \mathbb{N} P^n(x) = \Lambda$.

Elements of N are called nodes. Elements of F are called function nodes. Elements of $N \setminus F$ are called variables.

L is a labeling function. It assigns to every function node a symbol, namely the function symbol of the node. For each function node $x \in F$ the list $S(x)$ consists of the successor nodes of x also called arguments of x . If $x \in F$ and $S(x) = [y_0, \dots, y_{n-1}]$ then:

- (1) $\text{arity}(x) := n$,
- (2) $\text{arg}(x, i) := y_i$ for $i < n$.

The function P induces a tree structure on G . We write Px instead of $P(x)$. The trees induced by P are called equivalence classes. If $Px = \Lambda$ then x is called a root. If $Px = y$ then y is called the parent node of x . For each node $x \in N$ we define

$$P^*x := y \Leftrightarrow \exists n \in \mathbb{N} (P^n x = y \ \& \ P y = \Lambda),$$

i.e. there exists a non-cyclic path of length n in G from the node x to the root y .

An equivalence relation \sim is defined on the set N by $x \sim y \Leftrightarrow P^*x = P^*y$, i.e. if the two nodes x and y have the same root, P^*x can be understood as the representative of the equivalence class of x .

Definition: A function T is called a solution of a unification graph G , if it assigns to each node $x \in N$ a term $T(x)$ such that

- (1) $T(x) \equiv T(y)$ for all nodes $x, y \in N$ such that $Px = y$ and
- (2) $T(x) \equiv f(T(y_0), \dots, T(y_{n-1}))$ for all $x \in F$ such that $L(x) = f$ and $S(x) = [y_0, \dots, y_{n-1}]$.

A solution T_1 is called more general than T_2 if there exists a substitution \mathbf{s} such that $T_1(x)\mathbf{s} \equiv T_2(x)$ for all nodes $x \in N$.

Definition: A unification graph G is called acyclic if the relation $<$ defined by

$$y < x \Leftrightarrow P^*x \in F \ \& \ \exists i < \text{arity}(P^*x) y = \text{arg}(P^*x, i)$$

is acyclic, i.e. if there exists no sequence $z_0 < z_1 < \dots < z_n$ such that $0 < n$ and $z_0 = z_n$.

Lemma: If G has a solution then G is acyclic.

Definition: A unification graph G is called solved, if G is acyclic and for all nodes $x, y \in F$ such that $Px = y$

- (1) $L(x) = L(y)$,
- (2) $\text{arity}(x) = \text{arity}(y)$,
- (3) $\text{arg}(x, i) \sim \text{arg}(y, i)$ for all $i < \text{arity}(x)$.

Lemma: If G is solved, then it has a most general solution.

The aim of the unification algorithm is to transform a given graph G into a solved graph G' by refining the parent function P . In the correctness proof an imaginary set $O \subseteq N$ of open nodes is required. Initially, $O := \emptyset$. A sketch of the unification algorithm is given below:

```

function unify( $x, y$ )
begin
     $x := P^*(x);$                                 /*  $x := find(x)$  */ (1)
     $y := P^*(y);$                                 /*  $y := find(y)$  */ (2)
    if  $x = y$  then                               (3)
        return(true)                             (4)
    end;                                           (5)
    if  $x \in N \setminus F$  then                   /*  $x$  is a non-functional node */ (6)
         $P(x) := y;$                                /*  $union(x, y)$  */ (7)
        return(true)                             (8)
    elseif  $y \in N \setminus F$  then               /*  $y$  is a non-functional node */ (9)
         $P(y) := x;$                                /*  $union(y, x)$  */ (10)
        return(true)                             (11)
    elseif  $L(x) = L(y)$  and  $arity(x) = arity(y)$  then /*  $x, y$  in the same set */ (12)
         $P(x) := y;$                                /*  $union(x, y)$  */ (13)
        /*  $O := O \cup \{x\}$  */ (14)
        for  $i = 0, \dots, arity(x) - 1$  do (15)
            if  $\neg unify(arg(x, i), arg(y, i))$  then (16)
                return(false) (17)
            end (18)
        end (19)
        /*  $O := O \setminus \{x\}$  */ (20)
    else (21)
        return(false) (22)
    end (23)
end unify.
    
```

Figure 2.2 Unification algorithm

The function *unify* terminates, since the number of roots decreases in each recursive call.

- 1) Assume that T is a solution of G : If $x, y \in N$ and $T(x) = T(y)$ then *unify*(x, y) returns **true** and transforms G into a graph G' such that T is still a solution of G' .
- 2) If *unify*(x, y) returns **true**, (thus transforming G into G') and if $u \sim v$ then $u \sim' v$.
- 3) If *unify*(x, y) returns **true**, (thus transforming G into G') then $x \sim' y$.
- 4) If *unify*(x, y) returns **true**, (thus transforming G into G') then $O = O'$.
- 5) Assume that *unify*(x_0, y_0) returns **true**, (thus transforming G into G') and further assumed that for all $x \in F \setminus O$ and $y \in N$ such that $P x = y$ we have
 - a) $L(x) = L(y)$,
 - b) $arity(x) = arity(y)$,
 - c) $arg(x, i) \sim arg(y, i)$ for all $i < arity(x)$,
 then we have for the graph G' the same, i.e. for all $x \in F \setminus O$ and $y \in N$ such that $P' x = y$ we have
 - a) $L(x) = L(y)$,
 - b) $arity(x) = arity(y)$,
 - c) $arg(x, i) \sim' arg(y, i)$ for all $i < arity(x)$.

For a proof of the above properties, see Appendix A.

2.4.1 Example

Given are two terms $x := f(g(s, t), h(t))$ and $y := f(u, v)$ which are to be unified. Figure 2.3 shows the tree structures. The numbers next to the nodes show a consecutive numbering used throughout the example.

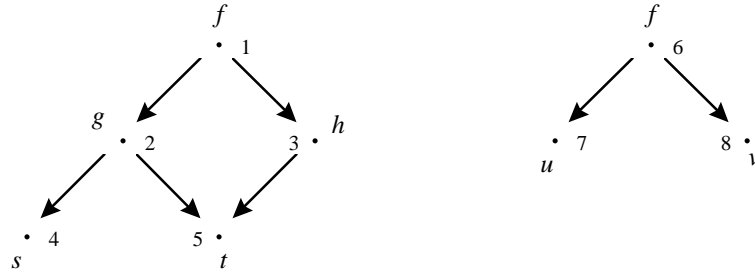


Figure 2.3 The tree structure of the terms $f(g(s,t),h(t))$ and $f(u,v)$

As a first step, the sets N, F as well as the functions L and S are determined:

$$N = \{1, 2, 3, 4, 5, 6, 7, 8\},$$

$$F = \{1, 2, 3, 6\},$$

$$L(1) = f, L(2) = g, L(3) = h,$$

$$L(6) = f,$$

$$S(1) = [2, 3], S(2) = [4, 5], S(3) = [5],$$

$$S(6) = [7, 8].$$

When calling *unify* with arguments x and y defined above, the two terms are re-defined to point to nodes number one and six, by lines (1) and (2) of the algorithm, respectively. (This is the well known *find* operation, which returns an identifier specifying the set to which an element belongs to). Since $L(1) = L(6)$ and $\text{arity}(1) = \text{arity}(6)$, line (12) applies, resulting in a new tree structure shown in figure 2.4, (this is the well known *union* operation, which combines two given sets) and following two recursive calls of the function:

- *unify*(2,7) evaluating to *unify*($g(s,t),u$) and
- *unify*(3,8) evaluating to *unify*($h(t),v$).

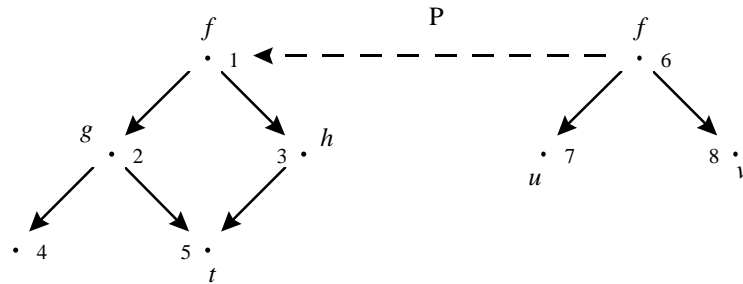


Figure 2.4 The tree structure after the first *union* operation

The two recursive calls each add another vertex to the tree, connecting nodes 2 and 7 as well as nodes 3 and 8 as shown in figure 2.5. The algorithm then terminates, because the criteria in line (3) is always met from then on.

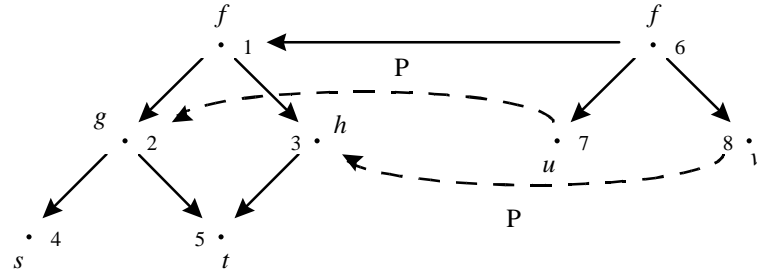


Figure 2.5 The tree structure after termination of the algorithm

Finally, the function T can be determined and the unification can be resolved:

$$T(1) = f(g(s,t),h(t),$$

$$T(2) = g(s,t),$$

$$T(3) = h(t),$$

$$T(4) = s,$$

$$T(5) = t,$$

$$T(6) = T(1),$$

$$T(7) = g(T(4),T(5)) = g(s,t),$$

$$T(8) = h(T(5)) = h(t).$$

As a final result, the substitution unifying the two terms is: $\{u/g(s,t),v/h(t)\}$.

2.5 Cycle test

Finally, it must be verified, that the unification graph obtained by the algorithm in figure 2.2, is cycle-free. This is done by the functions *cycle* and *test*, illustrated in figures 2.6 and 2.7.

Let (G, \rightarrow) be a directed graph.

```

function cycle( $G$ )
begin
   $W := \emptyset;$  (1)
  foreach  $x \in G$  do (2)
    if  $\neg \text{test}(x, \emptyset)$  then (3)
      return(false) (4)
    end; (5)
  end; (6)
  return(true) (7)
end cycle.

```

Figure 2.6 The function *cycle*

```

function test( $x, P$ )
begin
  if  $x \in W$  then                                     (1)
    return(true)                                       (2)
  elseif  $x \in P$  then                                 (3)
    return(false)                                     (4)
  else                                                 (5)
    foreach  $y \in \{y \in G : x \rightarrow y\}$  do      (6)
      if  $\neg \text{test}(y, P \cup \{x\})$  then              (7)
        return(false)                                 (8)
      end;                                           (9)
    end;                                           (10)
     $W := W \cup \{x\};$                                (11)
    return(true)                                     (12)
  end;                                           (13)
end test.

```

Figure 2.7 The function *test*

The successors of a node x are only computed provided $x \notin W$ and $x \notin P$ (lines (1) and (3) in figure 2.7). Thus, for each node x the function *test* is called at most twice with first argument x .

1) Assuming:

- $P = \{x_1, \dots, x_n\}$,
- $x_1 \rightarrow \dots \rightarrow x_n$,
- $x_n \rightarrow y$,
- *test*(y, P) returns **false**.

Then a cycle must exist in the graph.

2) If $x \in W$ then there exists no path $y_0 \rightarrow \dots \rightarrow y_n$ such that

- $x = y_0$,
- there exists a $i < n$ such that $y_i = y_n$.

3) If *test*(x, P) returns **true** then $x \in W$.

Chapter 3

Specification of the unification algorithm

This chapter describes the specifications as well as the implementation of the unification algorithm used by LPTP. The code can also be found in the Prolog source-file named "mgu.pl" on the accompanying disk. Remark: " $\neg G$ " means "not G ", and " $(G_0 \rightarrow G_1 ; G_2)$ " means "if G_0 then G_1 else G_2 ".

3.1 Specification

A term is a variable or a function symbol with terms as arguments as seen above. Example: The variable X is encoded as $\$(x)$, the term $f(X,Y,c)$ is encoded as $[f, \$(x), \$(y), [c]]$. Note that the symbol "\$" is a unary constructor in Prolog. Instead of $\$(x)$ one could write $\text{var}(x)$ as well.

In this case, terms are coded in Prolog as follows:

term(T) means: T is a term.

```
term( $(X) ) :-
    atomic(X).

term( [X|TL] ) :-
    atomic(X),
    termL(TL).
```

Similarly, a term-list is specified:

termL(TL) means: TL is a list of terms.

```
termL( [] ).

termL( [T|TL] ) :-
    term(T),
    termL(TL).
```

By definition, a term is a subterm of itself. If a term $T1$ occurs in a second term $T2$, then $T1$ is a subterm of $T2$.

subterm(T1,T2) means: $T1$ is a subterm of $T2$.

```
subterm(T, T).

subterm(T, [_|TL] ) :-
    subtermL(T, TL).
```

subtermL(T1,TL) means: *T1* is a subterm of an element of a list of subterms *TL*.

```
subtermL(T1,TL) :-
  member(T2,TL),
  subterm(T1,T2).
```

var_form(T) means: *T* has the encoded form of a variable.

```
var_form($(_)).
```

size(T,N) means: The size of the term *T* equals *N*.

N is a successor number, e.g. 0, *s*(0), *s*(*s*(0)), *s*(*s*(*s*(0))), The size of a variable-form equals *s*(0) = 1.

```
size($(_),s(0)).

size([_|TL],s(N)) :-
  sizeL(TL,N).
```

sizeL(TL,N) means: The size of the list of terms *TL* equals *N*.

Remark: *plus/3* is defined in *ltp/lib/nat/nat.pl*.

```
sizeL([],0).

sizeL([T|TL],N3) :-
  size(T,N1),
  sizeL(TL,N2),
  plus(N1,N2,N3).    % N3 = N1 + N2
```

substitution(S) means: *S* is a substitution. A substitution *S* is a list of bindings of the form *bind(X,T)*, where *X* is the name of a variable and *T* is a term. There is at most one binding of the form *bind(X,T)* for *X* in *S*.

```
substitution([]).

substitution([bind(X,T)|S]) :-
  atomic(X),
  term(T),
  substitution(S),
  \+ domain(X,S).
```

domain(X,S) means: The variable *X* is in the domain of the substitution *S*. There exists a binding *bind(X,T)* in *S*.

```
domain(X,S) :-
  member(bind(X,_),S).
```

apply(T1,S,T2) means: *T2* is the result of applying the substitution *S* to the term *T1*.

```
apply($ (X), S, T) :- assoc(X, S, T).
```

```
apply([X|T1L], S, [X|T2L]) :-  
  applyL(T1L, S, T2L).
```

applyL(T1L, S, T2L) means: *T2L* is the result of applying the substitution *S* to each term in the list of terms *T1L*.

```
applyL([], _, []).
```

```
applyL([T1|T1L], S, [T2|T2L]) :-  
  apply(T1, S, T2),  
  applyL(T1L, S, T2L).
```

assoc(X, S, T) means: *T* is the result of applying the substitution *S* to the variable *X*.

```
assoc(X, [], $ (X)).
```

```
assoc(X, [bind(X, T) | _], T).
```

```
assoc(X, [bind(Y, _) | S], T) :-  
  \+ X = Y,  
  assoc(X, S, T).
```

class(C) means: *C* is a class (a tree of terms). A class has the form *cl(T, P)*. *T* is the term representing the root of the class *C*. *P* is a list of the children of the term *T*. *P* is called a partition. It is a list of classes. If the root of a class is a variable, then all its members must be variables too.

```
class(cl(T, P)) :-  
  term(T),  
  partition(P),  
  \+ partition_member(T, P),  
  \+ not_var_class(T, P).
```

not_var_class(T, P) means: *P* is a partition containing no variables.

```
not_var_class($ (_, P)) :-  
  partition_member(T, P),  
  \+ var_form(T).
```

partition(P) means: *P* is a partition (a list of disjoint classes).

```
partition([]).
```

```
partition([C|P]) :-  
  class(C),  
  partition(P),  
  disjoint(C, P).
```

disjoint(C, P) means: The class *C* and the partition *P* are disjoint. This has to be done in two steps because of the lack of a "not *a* and not *b*"-construct in Prolog.

```

disjoint(C,P) :-
  \+ not_disjoint(C,P).

not_disjoint(C,P) :-
  class_member(T,C),
  partition_member(T,P).

```

class_solution(C,S) means: The substitution S is a solution of the class C , meaning that the substitution S unifies all the terms of the class C .

```

class_solution(C,S) :-
  \+ not_class_solution(C,S).

not_class_solution(C,S) :-
  class_member(T1,C),
  class_member(T2,C),
  apply(T1,S,T3),
  apply(T2,S,T4),
  \+ T3 = T4.

```

partition_solution(P,S) means: The substitution S is a solution of the partition P .

```

partition_solution([],_).

partition_solution([C|P],S) :-
  class_solution(C,S),
  partition_solution(P,S).

```

unifier(T1,T2,S) means: The substitution S is a unifier of $T1$ and $T2$.

```

unifier(T1,T2,S) :-
  apply(T1,S,T3),
  apply(T2,S,T3).

```

unifierL(TL1,TL2,S) means: The substitution S is a unifier of the term-lists $TL1$ and $TL2$.

```

unifierL([],[],_).

unifierL([T1|TL1],[T2|TL2],S) :-
  unifier(T1,T2,S),
  unifierL(TL1,TL2,S).

```

solved(P) means: The partition P is in solved form. If C is a class of the partition P with root $[F|S_1, \dots, S_m]$ and $[G|T_1, \dots, T_n]$ is an element of C , then F is equal to G , m is equal to n , and the term S_i is equivalent to T_i with respect to P .


```

solved(P) :-
  \+ not_solved(P).

not_solved(P) :-
  member(cl([X1|T1L],P1),P),
  partition_member([X2|T2L],P1),
  (
    \+ X1 = X2
  ;
    \+ equivalentL(T1L,T2L,P)
  ).

```

equivalent(T1,T2,P) means: *T1* and *T2* are equivalent with respect to the partition (equivalence relation) *P*. *T1* and *T2* belong to the same class of *P*.

```

equivalent(T1,T2,P) :-
  find(P,T1,T),
  find(P,T2,T).

```

equivalentL(T1L,T2L,P) means: The elements of two term-lists *T1L* and *T2L* are equivalent in twos.

```

equivalentL([],[],_).

equivalentL([T1|T1L],[T2|T2L],P) :-
  equivalent(T1,T2,P),
  equivalentL(T1L,T2L,P).

```

3.2 Implementation

unifiable_terms(T1,T2) means: *T1* and *T2* can be unified using the union-find algorithm.

```

unifiable_terms(T1,T2) :-
  unify_terms_part(T1,T2,_).

```

unify_terms_part(T1,T2,P) means: *T1* and *T2* can be unified using the union-find algorithm. The result is the solved, cycle-free partition *P*.

```

unify_terms_part(T1,T2,P) :-
  union_find(T1,T2,[],P),
  cycle_free(P).

```

unify_terms_sub(T1,T2,S) means: Try to unify the two terms *T1* and *T2* using the union-find algorithm. If they are unifiable, convert the solved, cycle-free partition *P* into a unifying substitution.

```

unify_terms_sub(T1,T2,S) :-
  unify_terms_part(T1,T2,P),
  partition_sub(P,P,[],S).

```

union_find(T1,T2,P1,P4) means:

```

union_find(T1,T2,P1,P4) :-
  find_delete(P1,T1,P2,C1),
  (
    class_member(T2,C1) ->
      P4 = P1
  ;
    find_delete(P2,T2,P3,C2),
    C1 = cl(T3,Q1),
    C2 = cl(T4,Q2),
    (
      var_form(T3) ->
        P4 = [cl(T4,[C1|Q2])|P3]
    ;
      var_form(T4) ->
        P4 = [cl(T3,[C2|Q1])|P3]
    ;
      T3 = [Tag|T1L],
      T4 = [Tag|T2L],
      union_findL(T1L,T2L,
        [cl(T4,[C1|Q2])|P3],P4) % P(x) := y
    )
  ).

```

union_findL(T1L,T2L,P1,P3) means:

```

union_findL([],[],P,P).

union_findL([T1|T1L],[T2|T2L],P1,P3) :-
  union_find(T1,T2,P1,P2),
  union_findL(T1L,T2L,P2,P3).

```

class_member(T,C) means: The term *T* belongs to the class *C*.

```

class_member(T,cl(T,_)).

class_member(T,cl(_,P)) :-
  partition_member(T,P).

```

partition_member(T,P) means: The term *T* belongs to one of the classes of *P*.

```

partition_member(T,[C|_]) :-
  class_member(T,C).

partition_member(T,[_|P]) :-
  partition_member(T,P).

```

find(P,T1,T2) means: The term *T1* belongs to the class with root *T2* in partition *P*.

```

find([],T,T).

find([C|P],T1,T2) :-
  (
    class_member(T1,C) ->
      C = cl(T2,_)
  ;
    find(P,T1,T2)
  ).

```

find_delete(P1,T,P2,C) means: Find the class *C* of partition *P1* to which the term *T* belongs to. Delete *C* in *P1* to obtain partition *P2*.

```

find_delete([],T,[],cl(T,[])).

find_delete([C1|P1],T,P3,C2) :-
  (
    class_member(T,C1) ->
      C2 = C1,
      P3 = P1
  ;
    find_delete(P1,T,P2,C2),
    P3 = [C1|P2]
  ).

```

cycle_free(P) means: The partition *P* is cycle free.

```

cycle_free(P) :-
  roots(P,TL),
  cycle_freeL(TL,P,[],[],_).

```

roots(P,TL) means: The term-list *TL* is the list of the roots of the classes of the partition *P*.

```

roots([],[]).

roots([cl(T,_)|P],[T|TL]) :-
  roots(P,TL).

```

cycle_freeL(TL,P,C,WF1,WF2) means: Check whether the terms in the term-list *TL* are in the cycle-free portion of partition *P*. *C* is the path to the terms in *TL*. *WF1* is a list of nodes which are already in the cycle-free part of *P*. *WF2* is the output list of nodes which are in the cycle-free part of *P*. *WF2* extends *WF1*. *WF2* is a so-called topological ordering of *P*.

```

cycle_freeL([],_,_,WF,WF).

cycle_freeL([T1|T1L],P,C,WF1,WF3) :-
  find(P,T1,T2),
  (
    member_check(T2,C) ->
      fail
  ;
    member_check(T2,WF1) ->
      cycle_freeL(T1L,P,C,WF1,WF3)
  ;
    var_form(T2) ->
      cycle_freeL(T1L,P,C,[T2|WF1],WF3)
  ;
    T2 = [_|T2L],
    cycle_freeL(T2L,P,[T2|C],WF1,WF2),
    cycle_freeL(T1L,P,C,[T2|WF2],WF3)
  ).

```

member_check(X,L) means: Check whether an element *X* belongs to a list *L*.

```

member_check(X,[Y|L]) :-
  (
    X = Y ->
      true
  ;
    member_check(X,L)
  ).

```

partition_sub(P1,P2,S1,S2) means: Go through the classes of partition *P1* and look for variables. Take variables. Expand them to terms according to the partition *P2*. Add the bindings to *S1*. The result is *S2*.

```
partition_sub([],_,S,S).

partition_sub([C|P1],P2,S1,S3) :-
    class_sub(C,P2,S1,S2),
    partition_sub(P1,P2,S2,S3).

class_sub(cl($(X),P1),P2,S1,S2) :-
    partition_term($(X),P2,T),
    partition_sub(P1,P2,[bind(X,T)|S1],S2).

class_sub(cl([_|_],P1),P2,S1,S2) :-
    partition_sub(P1,P2,S1,S2).
```

partition_term(T1,P,T2) means: Expand the term *T1* into term *T2* according to partition *P*.

```
partition_term(T1,P,T3) :-
    find(P,T1,T2),
    (
        var_form(T2) ->
            T3 = T2
    ;
        T2 = [Tag|T1L],
        partition_termL(T1L,P,T2L),
        T3 = [Tag|T2L]
    ).
```

partition_termL(T1L,P,T2L) means: Expand every term in the term-list *T1L* into the term-list *T2L* according to partition *P*.

```
partition_termL([],_,[]).

partition_termL([T1|T1L],P,[T2|T2L]) :-
    partition_term(T1,P,T2),
    partition_termL(T1L,P,T2L).
```

Chapter 4

Correctness Proof

This chapter illustrates some of the most important theorems, lemmas and corollaries used in the correctness proof. The entire correctness proof can be found in the files provided on the accompanying disk. The proof is 13,463 lines long. A listing and the contents of the disk files is given in appendix B and C of this diploma thesis.

For each theorem, lemma, corollary or definition, the names of the file holding the mentioned proof is given in square brackets [].

4.1 An introductory example

As an introductory example the general requirements for proving a given program are shown by means of the program *assoc*. *Assoc(X,S,T)* means, *T* is the result of applying the substitution *S* to the variable *X*. The corresponding Prolog code is:

```
assoc(X, [], $(X) ) .
assoc(X, [bind(X,T) | _], T) .
assoc(X, [bind(Y,_) | S], T) :-
    \+ X = Y,
    assoc(X, S, T) .
```

Figure 4.1 Program *assoc*.

The first step is to define the program output. In the given case, *T* must be in term-form. This is ensured by the lemma *assoc:types* shown in figure 4.2 and found in file [substitution.pr].

Lemma 1 [*assoc:types*] $\forall x,s,t (\mathbf{S} \text{ assoc}(x,s,t) \wedge \mathbf{S} \text{ atomic}(x) \wedge \mathbf{S} \text{ substitution}(s) \rightarrow \mathbf{S} \text{ term}(t)).$

Proof.

Induction₀: $\forall x,s,t (\mathbf{S} \text{ assoc}(x,s,t) \rightarrow \mathbf{S} \text{ atomic}(x) \wedge \mathbf{S} \text{ substitution}(s) \rightarrow \mathbf{S} \text{ term}(t)).$

Hypothesis₀: none.

Assumption₂: $\mathbf{S} \text{ atomic}(x) \wedge \mathbf{S} \text{ substitution}([]).$

Thus₂: $\mathbf{S} \text{ atomic}(x) \wedge \mathbf{S} \text{ substitution}([]) \rightarrow \mathbf{S} \text{ term}(\$x).$

Conclusion₁: $\mathbf{S} \text{ atomic}(x) \wedge \mathbf{S} \text{ substitution}([]) \rightarrow \mathbf{S} \text{ term}(\$x).$

Hypothesis₁: none.

Assumption₂: $\mathbf{S} \text{ atomic}(x) \wedge \mathbf{S} \text{ substitution}([\text{bind}(x,t)|v_0]).$ **D** $\mathbf{S} \text{ substitution}([\text{bind}(x,t)|v_0])$
by completion. $\mathbf{S} \text{ term}(t).$

Thus₂: $\mathbf{S} \text{ atomic}(x) \wedge \mathbf{S} \text{ substitution}([\text{bind}(x,t)|v_0]) \rightarrow \mathbf{S} \text{ term}(t).$

Conclusion₁: $\mathbf{S} \text{ atomic}(x) \wedge \mathbf{S} \text{ substitution}([\text{bind}(x,t)|v_0]) \rightarrow \mathbf{S} \text{ term}(t).$

Hypothesis₁: $\mathbf{S} \text{ atomic}(x) \wedge \mathbf{S} \text{ substitution}(s) \rightarrow \mathbf{S} \text{ term}(t)$ and $x \neq y$ and $\mathbf{S} \text{ assoc}(x,s,t).$

Assumption₂: $\mathbf{S} \text{ atomic}(x) \wedge \mathbf{S} \text{ substitution}([\text{bind}(y,v_0)|s]).$ **D** $\mathbf{S} \text{ substitution}([\text{bind}(y,v_0)|s])$
by completion.

Thus₂: $\mathbf{S} \text{ atomic}(x) \wedge \mathbf{S} \text{ substitution}([\text{bind}(y,v_0)|s]) \rightarrow \mathbf{S} \text{ term}(t).$

Conclusion₁: $\mathbf{S} \text{ atomic}(x) \wedge \mathbf{S} \text{ substitution}([\text{bind}(y,v_0)|s]) \rightarrow \mathbf{S} \text{ term}(t).$

Assumption₀: $\mathbf{S} \text{ assoc}(x,s,t) \wedge \mathbf{S} \text{ atomic}(x) \wedge \mathbf{S} \text{ substitution}(s).$

$\mathbf{S} \text{ atomic}(x) \wedge \mathbf{S} \text{ substitution}(s) \rightarrow \mathbf{S} \text{ term}(t).$

$\mathbf{S} \text{ atomic}(x) \wedge \mathbf{S} \text{ substitution}(s) \rightarrow \mathbf{S} \text{ term}(t).$ $\mathbf{S} \text{ term}(t).$

Thus₀: $\mathbf{S} \text{ assoc}(x,s,t) \wedge \mathbf{S} \text{ atomic}(x) \wedge \mathbf{S} \text{ substitution}(s) \rightarrow \mathbf{S} \text{ term}(t).$

Figure 4.2 LPTP-induction-proof of the lemma *assoc:types*.

The upper-case, bold-faced "D" in the proof symbolizes a definition. The upper-case, bold-faced "S" symbolizes *succeeds*, indicating that the subsequent predicate returns the answer "yes". This means the answer produced by the *assoc* program is indeed in term-form.

As a next step, it must be shown that the *assoc*-function actually exists. This is proven by means of lemma *assoc:existence* shown in figure 4.3. For the sake of readability, we now refrain from displaying the proof, thus only showing the lemma itself. The complete proof however, can be found in [substitution.pr].

Lemma 2 [*assoc:existence*] $\forall x,s (\mathbf{S} \text{ substitution}(s) \wedge \mathbf{S} \text{ atomic}(x) \rightarrow \exists t \mathbf{S} \text{ assoc}(x,s,t)).$

Figure 4.3 Lemma *assoc:existence*.

Furthermore, it must be shown, that the term computed by *assoc* is unique. For this purpose another lemma called *assoc:uniqueness* is set up. It is shown in figure 4.4 [substitution.pr].

Lemma 3 [*assoc:uniqueness*] $\forall x,s,t_1,t_2 (\mathbf{S} \text{ assoc}(x,s,t_1) \wedge \mathbf{S} \text{ atomic}(x) \wedge \mathbf{S} \text{ substitution}(s) \wedge \mathbf{S} \text{ assoc}(x,s,t_2) \rightarrow t_1 = t_2).$

Figure 4.4 Lemma *assoc:uniqueness*.

So far it has been demonstrated that:

- a function defined by *assoc* exists,
- the calculated result is unique,
- the calculated result is a term.

The next step is to demonstrate, whether or not the implementation of the function ever comes to a result at run-time, i.e. whether or not the program terminates. Therefore, a lemma called *assoc:termination* is defined. It is shown in figure 4.5 with the appropriate proof [part1.pr].

Lemma 4 [*assoc:termination*] $\forall x,s,t(\mathbf{S} \text{ substitution}(s) \wedge \mathbf{S} \text{ atomic}(x) \rightarrow \mathbf{T} \text{ assoc}(x,s,t))$.
Proof.
 Induction₀: $\forall s(\mathbf{S} \text{ substitution}(s) \rightarrow \forall x,t(\mathbf{S} \text{ atomic}(x) \rightarrow \mathbf{T} \text{ assoc}(x,s,t)))$.
 Hypothesis₁: none. $\mathbf{S} \text{ atomic}(x) \rightarrow \mathbf{T} \text{ assoc}(x,[],t)$.
 Conclusion₁: $\forall x,t(\mathbf{S} \text{ atomic}(x) \rightarrow \mathbf{T} \text{ assoc}(x,[],t))$.
 Hypothesis₁: $\forall x,t(\mathbf{S} \text{ atomic}(x) \rightarrow \mathbf{T} \text{ assoc}(x,s,t))$ and $\mathbf{S} \text{ atomic}(x)$ and $\mathbf{S} \text{ term}(t)$ and $\mathbf{S} \text{ substitution}(s)$ and $\mathbf{F} \text{ domain}(x,s)$.
 Assumption₂: $\mathbf{S} \text{ atomic}(v_0)$. $\exists t \mathbf{S} \text{ assoc}(v_0,s,t)$ by Lemma [*assoc:existence*]. $\text{gr}(x)$ by Axiom[*atomic:gr*].
 $\mathbf{T} \text{ assoc}(v_0, [\text{bind}(x,t)|s], v_1)$ by completion.
 Thus₂: $\mathbf{S} \text{ atomic}(v_0) \rightarrow \mathbf{T} \text{ assoc}(v_0, [\text{bind}(x,t)|s], v_1)$.
 Conclusion₁: $\forall v_0, v_1(\mathbf{S} \text{ atomic}(v_0) \rightarrow \mathbf{T} \text{ assoc}(v_0, [\text{bind}(x,t)|s], v_1))$.
 Assumption₀: $\mathbf{S} \text{ substitution}(s) \wedge \mathbf{S} \text{ atomic}(x)$. $\forall x,t(\mathbf{S} \text{ atomic}(x) \rightarrow \mathbf{T} \text{ assoc}(x,s,t))$.
 $\mathbf{S} \text{ atomic}(x) \rightarrow \mathbf{T} \text{ assoc}(x,s,t)$. $\mathbf{T} \text{ assoc}(x,s,t)$.
 Thus₀: $\mathbf{S} \text{ substitution}(s) \wedge \mathbf{S} \text{ atomic}(x) \rightarrow \mathbf{T} \text{ assoc}(x,s,t)$.

Figure 4.5 Lemma *assoc:termination*.

This lemma is also proved by induction.

4.2 Some important propositions

In this section, the most important theorems, lemmas and corollaries required for the correctness proof are investigated in detail.

For the sake of clarity, only the most important steps leading to the desired proof are shown. Lesser important steps are omitted.

4.2.1 Theorems about "unifiable_terms"

The two following theorem state that if $?s$ is a substitution unifying the terms $?t1$ and $?t2$, then *unifiable_terms* terminates and succeeds.

[part11.pr]

Theorem 1 [*unifiable_terms:characterization*] $\forall t_1, t_2(\mathbf{S} \text{ term}(t_1) \wedge \mathbf{S} \text{ term}(t_2) \rightarrow (\mathbf{S} \text{ unifiable_terms}(t_1, t_2) \leftrightarrow \exists s \mathbf{S} \text{ substitution}(s) \wedge \mathbf{S} \text{ unifier}(t_1, t_2, s)))$.

[part6.pr]

Theorem 2 [*unifiable_terms:termination*] $\forall t_1, t_2(\mathbf{S} \text{ term}(t_1) \wedge \mathbf{S} \text{ term}(t_2) \rightarrow \mathbf{T} \text{ unifiable_terms}(t_1, t_2))$.

4.2.2 Theorems about "unify_terms_part"

[part2.pr]

Corollary 3 [*unify_terms_part:types*] $\forall t_1, t_2, p(\mathbf{S} \text{ term}(t_1) \wedge \mathbf{S} \text{ term}(t_2) \wedge \mathbf{S} \text{ unify_terms_part}(t_1, t_2, p) \rightarrow \mathbf{S} \text{ partition}(p))$.

[part6.pr]

Theorem 4 [*unify_terms_part:termination*] $\forall t_1, t_2, p(\mathbf{S} \text{ term}(t_1) \wedge \mathbf{S} \text{ term}(t_2) \rightarrow \mathbf{T} \text{ unify_terms_part}(t_1, t_2, p))$.

[part12.pr]

Theorem 5 [*unify_terms_part:uniqueness*] $\forall t_1, t_2, p_1, p_2 (\mathbf{S} \text{ unify_terms_part}(t_1, t_2, p_1) \wedge \mathbf{S} \text{ unify_terms_part}(t_1, t_2, p_2) \rightarrow p_1 = p_2).$

[part7.pr]

Theorem 6 [*unify_terms_part:success*] $\forall t_1, t_2, s (\mathbf{S} \text{ term}(t_1) \wedge \mathbf{S} \text{ term}(t_2) \wedge \mathbf{S} \text{ substitution}(s) \wedge \mathbf{S} \text{ unifier}(t_1, t_2, s) \rightarrow \exists p (\mathbf{S} \text{ unify_terms_part}(t_1, t_2, p) \wedge \mathbf{S} \text{ partition_solution}(p, s))).$

[part11.pr]

Theorem 7 [*unify_term_part:solved*] $\forall t_1, t_2, s (\mathbf{S} \text{ term}(t_1) \wedge \mathbf{S} \text{ term}(t_2) \wedge \mathbf{S} \text{ unify_terms_part}(t_1, t_2, p) \rightarrow \mathbf{S} \text{ solved}(p)).$

4.2.3 Theorems about "unify_terms_sub"

[part8.pr]

Theorem 8 [*unify_terms_sub:types*] $\forall t_1, t_2, s (\mathbf{S} \text{ term}(t_1) \wedge \mathbf{S} \text{ term}(t_2) \wedge \mathbf{S} \text{ unify_terms_sub}(t_1, t_2, s) \rightarrow \mathbf{S} \text{ substitution}(s)).$

[part9.pr]

Theorem 9 [*unify_terms_sub:termination*] $\forall t_1, t_2, s (\mathbf{S} \text{ term}(t_1) \wedge \mathbf{S} \text{ term}(t_2) \rightarrow \mathbf{T} \text{ unify_terms_sub}(t_1, t_2, s)).$

[part12.pr]

Theorem 10 [*unify_terms_sub:uniqueness*] $\forall t_1, t_2, s_1, s_2 (\mathbf{S} \text{ unify_terms_sub}(t_1, t_2, s_1) \wedge \mathbf{S} \text{ unify_terms_sub}(t_1, t_2, s_2) \rightarrow s_1 = s_2).$

[part8.pr]

Theorem 11 [*unify_terms_sub:existence*] $\forall t_1, t_2, s (\mathbf{S} \text{ term}(t_1) \wedge \mathbf{S} \text{ term}(t_2) \wedge \mathbf{S} \text{ substitution}(s) \wedge \mathbf{S} \text{ unifier}(t_1, t_2, s) \rightarrow \exists s_1 (\mathbf{S} \text{ unify_terms_sub}(t_1, t_2, s_1) \wedge \text{composition}(s_1, s, s))).$

[part11.pr]

Theorem 12 [*unify_terms_sub:unifier*] $\forall t_1, t_2, s (\mathbf{S} \text{ term}(t_1) \wedge \mathbf{S} \text{ term}(t_2) \wedge \mathbf{S} \text{ unify_terms_sub}(t_1, t_2, s) \rightarrow \mathbf{S} \text{ unifier}(t_1, t_2, s)).$

4.2.4 Definition of "application"

By the next definition it is shown that an application is in fact a function. The application of a substitution $?s$ to the term $?t$ is written as $(?t // ?s)$.

[substitution.pr]

Definition 13 [//2] $\forall t_1, s, t_2 (\mathbf{S} \text{ term}(t_1) \wedge \mathbf{S} \text{ substitution}(s) \rightarrow (t_1 // s = t_2 \leftrightarrow \mathbf{S} \text{ apply}(t_1, s, t_2))).$

4.2.5 Definition of "composition"

A predicate *composition* is defined to express that $?s3$ is the substitution, which arises by applying to a term T the substitutions $?s1$ and $?s2$ in turns.

[substitution.pr]

Definition 14 [composition/3] $\forall s_1, s_2, s_3 (\text{composition}(s_1, s_2, s_3) \leftrightarrow \forall x (\mathbf{S} \text{ atomic}(x) \rightarrow \$ (x) // s_1 // s_2 = \$ (x) // s_3))$.

4.2.6 Termination of the cycle test

This theorem states: If P is a partition, then the cycle-test for P terminates. (A partition is a list of disjoint classes, which again is a tree of terms).

[part6.pr]

Theorem 15 [cycle_free:termination] $\forall p (\mathbf{S} \text{ partition}(p) \rightarrow \mathbf{T} \text{ cycle_free}(p))$.

4.2.7 Termination of the union-find algorithm

By this next theorem it is proven that if $T1$ and $T2$ are both terms and $P1$ is a partition, then the union-find algorithm terminates. It is expressed in the following way:

[part4.pr]

Theorem 16 [union_find:termination] $\forall t_1, t_2, p_1, p_2 (\mathbf{S} \text{ term}(t_1) \wedge \mathbf{S} \text{ term}(t_2) \wedge \mathbf{S} \text{ partition}(p_2) \rightarrow \mathbf{T} \text{ union_find}(t_1, t_2, p_1, p_2))$.

4.2.8 Most general substitution

Finally, it is proven that if two terms are unifiable, the union-find algorithm computes a most general substitution.

[part12.pr]

Theorem 17 [unify_terms_sub:most:general] $\forall t_1, t_2, s_1, s_2 (\mathbf{S} \text{ term}(t_1) \wedge \mathbf{S} \text{ term}(t_2) \wedge \mathbf{S} \text{ partition}(s_2) \wedge \mathbf{S} \text{ unifier}(t_1, t_2, s_2) \wedge \mathbf{S} \text{ unify_terms_sub}(t_1, t_2, s_1) \rightarrow \text{composition}(s_1, s_2, s_2))$.

Appendix A

Bibliography

- [1] Krzysztof R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.
- [2] Thomas H. Corman. *Introduction to Algorithms*. 485-487
- [3] A. Martelli and U. Montanari. *An efficient Unification Algorithm*. ACM Transactions on Programming Languages and Systems, 4(2): 258-282. 1982.
- [4] M. S. Paterson and M. N. Wegman. *Linear Unification*. J. of Computer and System Sciences, 16: 158-167, 1978.
- [5] Paul W. Purdom. *A Practical Unification Algorithm*. Information Sciences, 55:123-127, 1991.
- [6] J. A. Robinson. *A machine-Oriented Logic Based on the Resolution Principle*. J. of the Association for Computing Machinery, 12(1): 23-41, 1965.
- [7] Peter Ruzicka and Igor Privara. *An Almost Linear Robinson Unification Algorithm*. Acta Informatica 27:61-71, 1989.
- [8] Robert F. Stärk. *LPTP: A Logic Program Theorem Prover*. June 1996.
- [9] Robert F. Stärk. *A Simple Unification Algorithm*. February 1996.

Appendix B

Disk Files

1. mgu.pl
2. axioms.pr
3. substitution.pr
4. subterms.pr
5. part1.pr
6. part2.pr
7. part3.pr
8. part4.pr
9. part5.pr
10. part6.pr
11. part7.pr
12. part8.pr
13. part9.pr
14. part10.pr
15. part11.pr
16. part12.pr

Appendix C

Formal proof