A Unification Algorithm for Infinite Trees

Kuniaki Mukai

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

Abstract.

A simple unification algorithm for infinite trees has been developed. The algorithm is designed to work efficiently under structure sharing implementations of logic programming languages, e.g., Prolog. The data structure used are pairs of equation lists and sets of multi-equations. It represents the configurations of unification processes. A relation, called "is covered by", between two terms is introduced to terminate the algorithm. The fundamental operations are to compute the frontier set of two given terms and to test the relation between them. One of the main features of the algorithm is that the answer to the test is gained as a by-product of the frontier computation. Since there are some subtle points as to whether the algorithm will terminate, a termination proof is shown.

1. Introduction

The objective of this paper is to explain a simple and efficient unification algorithm for infinite trees. In Colmerauer[2], the idea of infinite trees into Prolog, in order to eliminate the need of occur check from unification processes, was introduced. He gave a general algorithm and a proof of its correctness and termination in his paper[2]. For any given two terms, the general algorithm has to select the smaller one in view of length. Of course it is not efficient to measure the length at the unification process. We found a relation which is called "is covered by", explained later in this paper. The relation is able to play the same role "smaller" relation above for the termination.

To describe our unification model, we will use a set of multi-equations to represent variable binding information. A configuration of the unification process is represented by an ordered pair of a list of equations and a set of multi-equations. We view unification processes as transformations between two configurations.

We want to build a complete and efficient unifier into Prolog with structure sharing implementation, for example Dec-10 Prolog. Our method does not need to copy any term, because terms that appear in the unification process are subterms of some term occurring in the input configuration.

Now, we will briefly explain a key point of the algorithm. Let's imagine the following situation: a current value of a variable v is a term b, and v=t is the current equation, where t is a non variable term. With this situation, we first test whether b "is covered by" t.

If this is not the case, we change it so that the new value of v will be t. During the test, we get the list of new equations of terms to be unified. This means that we do not any particular steps to test the relation "is covered by". All other points of the algorithm are usual.

Our algorithm is simple and it is easy to show its correctness using induction with regard to the basic transformation steps of configurations. It is not obvious, however, whether the algorithm terminates, and at this point of time, we have no short and clear explanation for termination without a formal proof. So, we will explain a detailed proof of termination.

We can reduce the termination problem by the fact that it is impossible for any infinite sequence, say (Ti;i>=1), to satisfy the condition: for each i >=1, 1) Ti is a finite set of (finite) terms, 2) for each term x in T(i+1), there exists a term y in Ti such that x is a proper subterm of y. If it existed, we would be able to make a downsequence of positive integers from the sequence (Ti;i>=1) by getting the maximal size Ni of all terms in Ti for each i>=1. But the integer sequence (Ni;i>=1) is obviously impossible.

2. Basic Definitions

We write a system of equations as a list of equations. For example, ⟨x=1,u=f(x),y=x⟩ is a system of equations. The length of the system is 3, the top of the system is the equation x=1. The form V=t is called a multi-equation, where V is a finite set of variables and t is a term. We use sets of multi-equations for variable-value bindings. For example, the result of the unification

x=y, z=2, u=v, y=1

is written as the set of multi-equations

{{x,y}=1,{z}=2,{u,v}="undef"}.

We use the special term "undef" for the undefined value.

Definition. A configuration of unification is an ordered pair of the form (R,M), where R is a list of equations of the form term=term, and M is a set of multi-equations.

Example. The configuration

(⟨x=y,y=z⟩,{{x}=f(x),{y}=f(y),{z}=f(z)})

represents the situation:
1) the current values of variables x, y and z are f(x), f(y) and f(z),and
2) remaining pairs of terms to be unified are {x,y} and {y,z}.

Definifition. SUBTERM(t) is the set of all subterms of t, and SUBTERM'(t) is the set of all proper subterms of t. For the special term "undef", we use the following definitions:

SUBTERM("undef")={"undef"},
SUBTERM'("undef")={} (empty set).

Example.
SUBTERM(f(g(x),1))={f(g(x),1),g(x),x,1}

SUBTERM'(f(g(x),1))={g(x),x,1}.

For convenience, we will extend the definitions SUBTERM and SUBTERMS' for lists of equations, lists of multi-equations, and configurations. Let R and M be a list of equations and a set of multi-equations.

Definition. SUBTERM(R) is the set of all subterms of the nonvariable left or right hand side of some equation in R. SUBTERM'(R) is the set of all proper subterms of the nonvariable left or right hand side of some eqation in R. SUBTERM(M) is the set of all subterms of the right hand side of some multi-equation in M. SUBTERM'(M) is the set of all the proper subterms of the right hand side of some multi-equation in M. SUBTERM((R,M)) is the union of SUBTERM(R) and SUBTERM(M). SUBTERM'(M) is the union of SUBTERM'(R) and SUBTERM'(M).

Example.

SUBTERM(<x=1,y=f(z)>) = {1,f(z),z}.
SUBTERM'(<x=1,y=f(z)>) = {z}.
SUBTERM({{x,y}=1,{z}=f(z),{u}="undef"}) = {1,f(z),z,"undef"}.
SUBTERM'({{x,y}=1,{z}=f(z),{u}="undef"}) = {z}.
SUBTERM((<x=f(y)>,{{x,y}=f(y)})) = {f(y),y}.
SUBTERM'((<x=f(y)>,{{x,y}=f(y)})) = {y}.

Definition. TERM(R) is the set of all nonvariable left or right hand side of some equation in R. TERM(M) is the set of all the right hand side of some multi-equation in M. TERM((R,M)) is the union of TERM(R) and TERM(M).

Example.

TERM(<x=f(x),y=z>) = {f(x)}.
TERM({{x}=1,{y,z}=2}) = {1,2}.
TERM((<x=f(x),y=z>,{{x}=1,{y,z}=2})) = {f(x),1,2}.

Let v, C, and M be a variable, a variable class, and a set of multi-equations.
Definitin. If there is a unique multi-equaton in M of the form C=b for some b, we write BIND(C,M) for b. If there is a unique multi-equation, say C'=b', in M such that v is in C', we write CLASS(v,M) and VALUE(v,M) for C' and b'.

Example.

BIND({z,u},{{x,y}=1,{z,u}=2,{v,w}=3}) = 2.
CLASS(z,{{x,y}=1,{z,u}=2,{v,w}=3}) = {z,u}.
VALUE(z,{{x,y}=1,{z,u}=2,{v,w}=3}) = 2.

3. Unification Algorithm

An initial configuration of our algorithm is the form (R,M), where R is a system of input equations. Without loss of generality, we can suppose that 1) either right or left hand side of each equation in R is a variable, 2) for each variable v occurring in R or M, there is a unique variable class C occurring in M such that v is in C, and 3) the special term "undef" does not occur in R. Since our basic transformations defined below conserve the properties, we supposed that these three conditions always hold for any configuration.

Our unification process terminates if and only if the current R is empty, or FRONTIER operation defined below returns "clash". "clash" means the failure of the unification.

The primitive operations on trees(terms) in the unification process are to compute the "frontier" of two given terms and to test whether the one given term "is covered by" the other one.

Definition. Let t and u be terms. FRONTIER is the function which satisfies the following conditions.
1) FRONTIER(t,u) = <t=u> if t or u is a variable.
2) FRONTIER(f(t1,t2,...,tr),f(u1,u2,...,ur))=F1+F2+...+Fr, where r>=0, f is a functor, for each i (1=<i=<r), Fi=FRONTIER(ti,ui) and Fi is not "clash", and "+" is the concatenation operator for lists.
3) FRONTIER(t,u) = <>, i.e., empty list if t or u is "undef".
4) FRONTIER(t,u) = "clash" otherwise.

Example.

FRONTIER(f(1,x),f(y,2)) = <1=y,x=2>.
FRONTIER(g(1),g(2)) = "clash".

Definition. For two given terms, t and u, we say t covers u if and only if:
1) t is "undef" or
2) u is not "undef" and FRONTIER(t,u) = <t1=v1,t2=v2,...,tr=vr> for some r>=0, where for each i (1=<i=<r) vi is a variable or atomic term.

Example.

f(g(1),2) covers f(x,y).
f(x,y) is covered by f(g(1),2).
f(x,g(y)) does not cover f(g(x),y).

Remark. If a term t1 is an instance of another term t2, then t1 covers t2. Therefore this relation is a generalization of the instance relation. The covering test and the frontier computation above can be made into a single procedure. More precisely, provided FRONTIER(t1,t2), the time complexty to test the covering relation between t1 and t2 is only proportional to the length of the frontier.

Next, we will define two basic transformations. Suppose the configuration (R,M) is given. The resulting configuration (R',M') is defined as follows.

Let v=t or t=v be the top of R, where v is a variable and t is a term. Although the R-component of a configuration is used as either a stack or a queue in the algorithm, it is treated as a set in the following definition for brevity.

Definition. Let (R,M) and (R',M') be two configurations. We write (R,M)->(R',M') if and only if one of the following conditions holds.

RULE1: t is a variable, CLASS(v,M)=CLASS(t,M), M'= M,and R'=R-{v=t}, where "-" is the difference operator for sets.
RULE2: t is a variable, CLASS(v,M) is not CALSS(t,M), M'=(M-{CLASS(v,M)=VALUE(v,M),CLASS(t,M)=VALUE(t,M)}) U {C=z}, and R'=R-{v=t} U FRONTIER(VALUE(v,M),VALUE(t,M)), C=CLASS(v,M) U CLASS(t,M), and z is "undef" if both VALUE(v,M) and VALUE(t,M) are "undef", otherwise any of them which is not "undef".
RULE3: t is not a variable, VALUE(v,M) is not covered by t, M'=(M-{CLASS(v,M)=VALUE(v,M)}) U {CLASS(v,M)=t}, and R'=(R-{v=t}) U FRONTIER(t,VALUE(v,M)).
RULE4: t is not a variable, VALUE(v,M) is covered by t, M'=M and R'=(R-{v=t}) U FRONTIER(t,VALUE(v,M)).

Example.
RULE1: (<x=y>,{{x,y}=1}) -> (<>,{{x,y}=1})

```
RULE2:  (<x=y>,{{x}="undef",{y}=1}) -> (<>,{{x,y}=1})
RULE3:  (<x=f(y)>,{{x,y}=f(f(x))}) -> (<y=f(x)>,{{x,y}=f(y)})
RULE4:  (<x=f(f(x))>,{{x,y}=f(y)}) -> (<y=f(x),{{x,y}=f(y)})
```

Algorithm.

Input data: a configuration, say (R0,M0).
Output data: "clash" or a set of multi-equations.

Method: 0) R=:R0 and M=:M0.
1) if R is empty then return M.
2) if (R,M)->(R',M') for some R' and M' then R=:R' and M=:M', otherwise return "clash".
3) go to 1). []

Example. Let's solve the equations <x=f(x),y=f(f(y)),y=x>.

```
          (<x=f(x),y=f(f(y)),y=x>,
           {{x}="undef",{y}="undef"})

        ->(<y=f(f(y)),y=x>,
           {{x}=f(x),{y}="undef"})              (RULE4)

        ->(<y=x>,
           {{x}=f(x),{y}=f(f(y))})              (RULE4)

        ->(<x=f(y)>,
           {{x,y}=f(x)})                        (RULE2)

        ->(<y=x>,
           {{x,y}=f(x)})                        (RULE4)

        ->(<>,
           {{x,y}=f(x)})                        (RULE1)
```

The output of this computation is {{x,y}=f(x)}, which means that the value of and y is the infinite tree(term) f(f(f(.... In the example, there is no application of RULE3.

Example. This example shows that the relation "is covered by" is essential for the termination of unification processes.

```
          (<x=f(y,f(g(y),x)),x=f(g(y),x)>,
           {{x}="undef",{y}="undef"})

        ->(<x=f(g(y),x)>,
           {{x}=f(y,f(g(y),x)),{y}="undef"})    (RULE4)

        ->(<g(y)=y,x=f(g(y),x)>,
           {{x}=f(g(y),x),{y}="undef"})         (RULE3)

        ->(<x=f(g(y),x)>,
           {{x}=f(g(y),x),{y}=g(y)})            (RULE4)

        ->(<y=y,x=x>,
           {{x}=f(g(y),x),{y}=g(y)})            (RULE4)

        ->(<>,
           {{x}=f(g(y),x),{y}=g(y)})            (RULE1,RULE1)
```

Remark. It is easy to check that if we do not replace the value in

RULE3 above, the unification process does not terminate.

4. Proof of Termination

4.1 Proof for Queue Version

In this subsection, we treat a system of equations as a queue in view of the basic transformations described above.

Definition. We write (R1,M1)=>(R2,M2) if and only if the following conditions hold: (R2,M2) is obtained from (R1,M1) by successive applications of basic transformations n (>0) times, where n is the length of R1. ("=>" is used only in the proof for the queue version.)

Example.

(<x=y,y=z>,{{x}=f(x),{y}=f(y),{z}=f(z)}) =>
(<x=y,x=z>,{{x,y,z}=f(x)}).

Lemma 1. If (R1,M)=>(R2,M) then TERM(R2) is a subset of SUBTERM'(R1)

Proof. From the definition of "=>", there exists a series of configurations ((Si,Ni);0=<i=<n) such that (S0,N0)->(S1,N1)->...->(Sn,Nn), where S0=R1, N0=M, Nn=M, and n is the length of R1.

Suppose there exists a term d in TERM(R1) but not in TERM(R2). Then, from the definition of "=>", we can select an integer j, a variable v, a term t, a variable class C, and a term b, satisfying all of the following conditions:
1) 1=<j=<n-1, the top of Sj is either t=v or v=t,
2) v is in C, BIND(C,M)=b, BIND(C,Nj)=b,
3) b is not in TERM(R1), b is not covered by t,
4) d is in TERM(FRONTIER(t,b)).

From 2), and since b is not covered by t, BIND(C,N(j+1)) must be t. Since b is not in TERM(R1), BIND(C,Ni) is not b for each i (j<i=<n). Since Nn=M, these imply that BIND(C,M) is not b. This is a contradiction to 2). Therefore, TERM(R2) is a subset SUBTERM'(R1). []

Corollary 2. There does not exist an infinite sequences of configurations ((Ri,M);i>=1) such that (R1,M) => (R2,M) => ... .
Proof. If the sequence exists, for any integer i>=1, TERM(R(i+1)) is a subset of SUBTERM'(Ri). As we have noted in the introduction, it is impossible. []

Lemma 3. There does not exist an infinite sequences of configurations ((Ri,Mi);i>=1) such that all of the following conditions hold:
1) (R1,M1) => (R2,M2) => ... => (Rn,Mn) => ...,
2) for each k>=1 and variable class C occurring in Mk, there exists such j (j>k) that BIND(C,Mj) is not BIND(C,Mk).
3) Mi's numbers of elements are equal to each other, i.e. no application of rule RULE2 appear in the sequence (i>=1).

Proof. From the infinite sequence above, we derive a contradiction. From 2), for any k there exists such j (k<j) that for any variable class C, the cardinality of the set
{i;k<i=<j, BIND(C,Mi) is not BIND(C,M(i-1))} is at least 2.

For each variable class C occurring in the sequence, let i(C) be the maximal integer i (i=<j) such that BIND(C,Mi) is not BIND(C,Mj). From the condition for j, for each C, i(C) must be greater than k, and BIND(C,Mj) is in TERM(Ri(C)). Since for any i>k TERM(Ri) is a subset of SUBTERM'((Rk,Mk)),BIND(C,Mj) is in SUBTERM'((Rk,Mk)).

By successive applications of this process, we can build the sequence k1<k2<... such that TERM((Rk(i+1),Mk(i+1))) is a subset of SUBTERM'((Rk(i),Mk(i))) (i>=1). This is, as said before, impossible. []

Theorem 4.4. There does not exist an infinite sequence such that

(R1,M1) => (R2,M2) => ... => (Rn,Mn) => ...

Proof. We can prove this by induction with regard to the number of elements of M1.

1) Suppose that the number of elements of M1 is 1. Because of the corollary 1, there exist no integer k>=1 such that Mk=M(k+1)=M(k+2)=.... On the other hand, the lemma 2 says that it is impossible for Mi to change infinitely many times. So, the foundation is proved.

2) Suppose that the number of elements of M1 is m+1, and that the theorem holds for the sequence such that the number of the variable classes of the sequence is at most m.

If for some k>=1, the number of the elements of Mk is less than that of M(k-1), then from the induction hypothesis, the sequence (Rk,Mk)=>(R(k+1),M(k+1)) => ... is finite. Then, the theorem holds in this case. Therefore, we suppose that the set of all variable classes occuring in Mi is independent of i (i>=1).

We derive a contradiction from the existence of the infinite sequence. For each i>=1, let Li be the set of all the common multi-equations in Mj (j>=i). And let L be the union of all Li (i>=1). L is not empty because of lemma2. Fix integer k>=1 so that for all j>=k L is a subset of Mj.

From the definitions of L and the basic transformations, TERM(Rj) is a subset of SUBTERM'((Rk,Mk-L)) for any j>k. By a similar method used in lemma2, we can construct the infinite sequence of integers k=<l1<l2<l3<... such that TERM((Rl(i+1),Ml(i+1)-L)) is a subset of SUBTERM'((Rl(i),Ml(i)-L)) (i>=1). But this is impossible, so the theorem is proved. []

4.2 Proof for Stack Version

Now, we will briefly give a termination proof for the stack version. Only points that are different from those in the queue version are included. The target is to derive a contradiction from the infinite sequence 1):

1) (R1,M1)->(R2,M2)->...->(Rn,Mn)->...

From the sequence, we can construct j1<j2<..., satisfying the following conditions 2) and 3):

2) For each k>=1, only one side of the equation, say Ek, of the top of Rj(k) is a variable, say v(k). We write t(k) for the other side of Ek.
3) Each Ek is an element of FRONTIER(VALUE(v(k-1),Mj(k-1)),t(k-1)),

(k>1).
From the infinite sequence j1<j2<..., we can get the subsequence i1<i2<... such that the following conditions 4) and 5) hold.

4) For each k (k>=1), L is a subset of Mi(k).

5) For each k (k>=1), TERM((<E(k+1)>,Mi(k+1)-L)) is a subset of SUBTERM'((<Ek>,Mi(k)-L).

It is easy to show that 5) is impossible. We have the contradiction.

The rest of the proof is to show 2) and 3).
Let X be the set {Wi;i>=1}. We assume that if i is not j then Wi is not Wj (i,j>=1). We say Wi is greater than Wj if and only if the following conditions 6) and 7) hold:

6) i<j,

7) for each k (i=<k=<j), the length of Wk does not exceed that of Wi.

From the definition of "->", X becomes a partially ordered set with respect to the relation. It is trivial to show that each connected component of X is a tree, and that the number of the components is equal to the length of R1. So, some component, say T, of X must be infinite (Konig's lemma). Since the number of branches at each node in X is at most finite, there is at least one infinite path through T. If T is written to be the set {Wj(k);k>=1}, the corresponding subsquence of 1) is wrtten as 8).

8) ((Rj(k),Mj(k));k>=1)

In the sequence 1), only finite applications of the RULE2 appear. For, the application decrease the number the variable classes. If we neglect the initial segment of 6) which is finite and sufficiently large, we get the sequence which satisfies 2) and 3).

5. Conclusion

We have proposed a simple and efficient unification algorithm for infinite trees, and have explained the algorithm as it applies to simple data structures consisting of equations and multi-equations. The unifier runs up and down along trees, locally, in a depth-first way, but globally breadth-first. Basic operations are to compute the frontier of two given terms and to test the relation between them that the one term is covered by the other. We can perform the computation and the test at once within a time proportional to at most the smaller size of the terms.

Each term which appears in the unification process is always a subterm of some term occurring in the input configuration. The algotithm does not need any special inner representation of terms. So we think it is easy for the algorithm to be built into the ordinary Prolog implementations with structure sharing, for example, DEC-10 Prolog[6,7]. We hope it will work in the average meaning as fast as the ordinary and conventional unifiers without "occur check".

The correctness of the algorithm is clear but termination not obvious. Therefore so we have described a detailed proof for the termination.

ACKNOWLEDGMENT

Kazuhiro Fuchi, Director of ICOT Research Center, and Tishio Yokoi, Chief of 3rd Labolatory, inspired me in this work. I also want to thank my colleague, especially Dr.Takashi Chikayama, 2nd Laboratory

for their valuable comments.

REFERENCES

1. Martelli, A., and Montanari, U. An Efficient Unification Algoritm. ACM Trans. on Programming Lang. and Syst., Vol.4, No.2, April 1982, pages 258-282.
2. Colmerauer, A. Prolog and Infinite Trees. Logic Programming, Academic Press, 1982.
3. Chang, C.L., Lee, C.R. Symbolic Logic and Mechanical Theorem Proving. Academic Press, New York, 1973.
4. Paterson, M.S., and Wegmann, M. Linear Unification. J.Comput.Syst.Sci. 16, 2(April 1978), 348-375.
5. Courcelle, B. Foundation of Infinite Trees. Theoretical Foundation of Programming Methodology, D.Reidel, 1982, 417-471.
6. Bowen, D.L. : DECsystem-10 PROLOG USER'S MANUAL, Dept. of AI, University of Edingburgh,1981.
7. Warren, D.H.D : Implementing Prolog - Compiling Predicate Logic Programs, Dept. of AI, University of Edingburgh Research Reprt 39&40, 1977.