# Tree Constraints

Rayna Dimitrova

Universität des Saarlandes, Saarbrücken 66123, Germany
rayna@mpi-sb.mpg.de

**Abstract.** This paper deals with constraint systems with tree domains. The two domains that we primarily consider are finite constructor trees and rational trees. We also give the general idea of feature trees - a generalization of the ordinary trees, corresponding to first order terms. We describe the types of constraints that are used in these three domains. The paper gives an overview of the algorithms for deciding solvability of systems of equations between terms and computing a finite representation of the set of all solutions of a solvable system. It also presents an algorithm for deciding solvability of systems containing both equations and disequations.

## 1 Introduction

Unification tries to identify two symbolic expressions by replacing certain subexpressions (variables) by other expressions. One considers terms that are built from function symbols and variable symbols. The whole process can be seen as solving systems of equations between terms. This is in fact a constraint satisfaction problem, where the variables are the variables occurring in the terms and the domains of all variables are the same - the set of all finite trees or the set of all infinite trees. The constraints in this case are described as equations between terms.

Unification was first introduced by J.A. Robinson in 1965 as a basic operation of his resolution principle, used in automated theorem provers. The development of Prolog and other logic programming languages has led to new research in this field. In Prolog II Colmerauer has extended the domain to infinite trees and has included disequations.

First, we provide some preliminary definitions. Let V and Σ be disjoint sets of *variables* and *function symbols* respectively, which do not contain the symbol "=" and let the set V be infinite. Each function symbol has an associated *arity*. Function symbols with arity 0 are called *constant symbols*.

### 1.1 Finite Constructor Trees

Consider the set of all finite trees with nodes labeled with function symbols. They are called *finite constructor trees*. The formal definition is the following:

- A single node labeled with a constant symbol is a finite constructor tree.

- A node labeled with a function symbol with arity n with a list of n finite constructor trees is a finite constructor tree.

In Figure 1 there is an example of a finite constructor tree. f, g and h are function symbol with arity 3, 2 and 1 respectively. a and b are constant symbols.

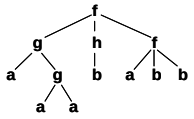


**Fig. 1. Example of finite constructor tree.**

### 1.2 Terms

If we allow leaf nodes of finite trees to be labeled with variables, we obtain *terms*. The variables replace subtrees, i.e. they play the role of placeholders. The result of replacing all variables in a term by finite trees is a finite tree and if we allow replacement by infinite trees then the result is an infinite tree. This is the formal definition:

- Every variable is a term.
- Every constant symbol is a term.
- If $t_1,\ldots,t_n$ are terms and f is a function symbol of arity n then $f(t_1,\ldots,t_n)$ is a term.

An example term is f(g(a,x),h(y),f(a,z,b)), where the variables are x, y and z.

The *size* of the term is defined to be the number of occurrences of variables and function symbols in it. The size of the above term is 10.

Terms that do not contain any variables are called *ground terms*. They are actually finite trees. The term f(g(a,g(a,a)),h(b),f(a,b,b)) is ground.

### 1.3 Term Equations

The atomic constraints are *equations between terms* i.e. ordered pairs (s,t) where s and t are terms, written in the form s = t. The goal is to make the two terms syntactically identical by assigning trees to the variables occurring in the terms s and t.

By *systems of equations* we mean finite sets of equations.

A *tree-assignment* σ is a finite set of ordered pairs of the form:

$$\sigma = \{x_1:=r_1, x_2:=r_2,\ldots, x_n:=r_n\}$$

where $r_i$'s are (possibly infinite) trees and $x_i$'s are distinct variables.

If a given term t contains no other variables than the $x_i$'s in σ then tσ denotes the tree defined by:

- if $t = x_i$ then $t\sigma = r_i$
- if $t = f(t_1,\ldots,t_m)$ then tσ is the tree with root labeled with the function symbol f and subtrees $t_1\sigma,\ldots,t_m\sigma$.

A tree-assignment σ is a *tree-solution* of an equation s = t iff $s\sigma \equiv t\sigma$. The only solution of the equation

$$f(g(a,x),h(y),f(a,z,b)) = f(g(a,g(a,a)),h(b),f(a,b,b))$$

is the tree-assignment in Figure 2.

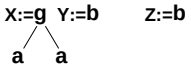


**Fig. 2. Tree-assignment.**

A tree-assignment is a tree-solution of a system of equations E if it is a tree-solution of every equation in E. Two systems of equations are called *equivalent* if they have the same set of tree-solutions.

First, we consider the problem of deciding solvability of a system of equations over the set of *finite constructor trees*. For a system that is solvable in finite trees, i.e. one that has a tree-solution of finite trees, we want to find a finite representation of the set of solutions. The equation x = f(x) has no tree-solution in finite trees. If we assume that there is such a tree-solution σ and the height of xσ is n then the height of f(x)σ should be n+1 and thus the two trees cannot be equal.

### 1.4 Rational Trees

An infinite tree is a tree with infinite set of nodes. It suffices to consider a special class of infinite trees called *rational trees*. A rational tree is a possibly infinite tree that has finitely many distinct subtrees and all nodes are finitely branching.


**Fig. 3.Solution of x = f(x) .**

A rational tree may be transformed into a finite possibly cyclic directed graph by merging all the nodes from which the same subtrees start. The nodes in this graph can represent different nodes of the tree, as long as the subtrees starting at these nodes are the same. This is called sharing of subtrees. The best we can do is to have one node for each distinct subtree - this is the minimal representation of the tree. Even the minimal representation of a finite tree may contain fewer nodes than the tree itself. A tree solution of the equation x = f(x) in rational trees and its finite representation are given in Figure 3.

### 1.5 Feature Trees

Records are an important data structure in programming languages and are necessary for the object oriented approach. They are present in all imperative languages as well as in modern functional languages. They can be modeled in logic programming with the help of trees, called feature trees. These are trees whose nodes are labeled with symbols called sots and whose edges are labeled with symbols called features. The features labeling the edges correspond to the field names of records. The edges departing from a particular node must be labeled with distinct features. Feature trees can be finite or infinite. With the help of infinite feature trees, cyclic data structures are represented in a convenient way. Finite constructor trees and rational trees can be seen as feature trees where the features labeling the edges departing from a node are consecutive positive integers. In Figure 4 is given a feature tree with sorts *circle*, *point*, *2*, *15*, *19* and features *radius*, *center*, *xval*, *yval*.

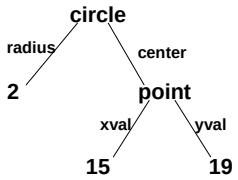


**Fig. 4. Feature tree.**

In the case of finite and rational trees, the atomic constraints we consider are equations between terms. For feature trees we need also other atomic constraints in order to be able to express their properties:

- sort constraints x : circle
- feature constraints x [ radius ] y
- arity constraints x { radius , center }
- equations between variables x = y

The constraints are described as first order formulas obtained from the primitive forms specified above. In comparison to standard tree constraint system, this provides more flexibility, since it is possible to express a constraint about one feature without saying anything about the existence of other features.

## 2 Solving Systems of Equations

### 2.1 Solving Systems of Equations with Domain the Set of Finite Trees

In Section 2.1 by tree-solution we mean a tree solution of finite trees. There is one special kind of equation systems, namely the systems in solved form.

**2.1.1 Solved Form** We say that the equation system $E = \{x_1 = t_1, \ldots, x_n = t_n\}$ is in solved form iff $x_1,\ldots,x_n$ are distinct variables that do not occur in the right hand side of any equation. The variables $x_1,\ldots,x_n$ are called *eliminable*. It is obvious that such a system is solvable. Any tree-assignment of finite trees to the variables occurring in $t_1,\ldots,t_n$ determines a unique tree-solution of the system in solved form, and conversely any tree-solution in finite trees of the system can be obtained in that way.

Now we present an algorithm that transforms any solvable equation system into an equivalent solved form equation system. For any unsolvable equation system the algorithm halts with failure. The algorithm is based on the following set of simplification rules:

### 2.1.2 Simplification Rules

**Decomposition:**

$$f(t_1,\ldots,t_n) = f(s_1,\ldots,s_n)$$

is replaced by the equations

$$t_1 = s_1, \ldots, t_n = s_n .$$

**Difference failure:** If the system contains an equation of the form

$$f(t_1,\ldots,t_n) = g(s_1,\ldots,g_m) ,$$

where f and g are different function symbols, then the algorithm halts with failure.

**Deletion:** If the system contains an equation

$$x = x ,$$

where x is a variable, delete this equation.

**Anteposition:** An equation

$$t = x ,$$

where x is a variable and t is not a variable is replaced by the equation

$$x = t .$$

**Variable Elimination:** If the system contains an equation

$$x = t ,$$

where x is a variable, t is a term different from x and x has another occurrence in the system then if x appears in t the algorithm halts with failure, otherwise x is replaced by t in every other equation.

It can easily be shown that these rules preserve the set of solutions of the system, i.e. if the equation system E′ is obtained by applying one of the above rules to an equation system E, then the two systems are equivalent.

**2.1.3 Solved Form Algorithm** Start with the initial system of equations. Nondeterministically choose one of the rules that can be applied to the current system and apply this rule. The algorithm terminates when no rule can be applied or when failure has been returned.

**Theorem 1** The solved form algorithm applied to a system of equations E will return an equivalent system of equations E′ in solved form iff E is solvable. It will return failure otherwise.

**Proof**:

**Termination**
For a system of equations we consider the ordered pair of natural numbers, consisting of:

- the number of variables occurring in the equations that are not eliminated (a variable x is *eliminated* in a system of equations if the system contains an equation x = t, x does not occur in t and x does not occur in any other equation of the system)
- the sum of the sizes of the terms on the left hand side of the equations.

Let us assume that there is an infinite sequence of systems

$$E_0, E_1, \ldots, E_n, \ldots,$$

where $E_0$ is the initial system and each $E_{i+1}$ is obtained from $E_i$ by an application of one of the simplification rules. Consider the corresponding sequence of ordered pairs. First, note that none of the rules introduces new variables into the system and once a variable is eliminated, it continues to be an eliminated variable in all successive systems of the sequence.

The application of rule Decomposition or Deletion strictly decreases the second component of the pair. The application of Anteposition does not increase the second component. On the other hand, the rule Anteposition can be applied only finitely many times without applying any of the other rules. None of these three rules increases the first component of the ordered pair, since they do not introduce new variables in the system. The application of rule Variable Elimination strictly decreases the first component of the ordered pair. According to the above observations, there is a strictly decreasing infinite subsequence of the sequence of ordered pairs with respect to the lexicographic order of NxN. This contradicts to the well-foundedness of this order.

**Correctness**

Assume that the algorithm has not returned failure and let E′ be the resulting system. Since each of the rules transforms a system of equations into an equivalent one, E and E′ are equivalent. All left hand sides of equations in E′ are variables, since otherwise at least one of the rules Decomposition, Difference Failure or Anteposition

could be further applied. All these variables are different and do not appear anywhere else in the system, otherwise the Variable Elimination rule or the Deletion rule could be further applied. Hence, E′ is in solved form and since it is equivalent to E, E is solvable.

Assume that the algorithm terminates with failure. Since each rule transforms a system of equations into an equivalent one, the last system E′ is equivalent to the initial system E. If the failure occurred by an application of rule Difference Failure then E′ is not solvable, since no matter what tree-assignment is applied to the two terms, the two trees obtained will have different labels for their roots and thus cannot be identical. If the failure occurred by an application of Variable Elimination, then E′ is not solvable since the equation x = t to which the rule is applied is not solvable. Thus, in this case E is not solvable. The proof is completed.

The worst case running time of this algorithm is exponential in the number of symbols in the initial system, because during the process we may end up with terms whose size is exponential in the number of symbols in the initial system.

There are more efficient unification algorithms, which employ a representation of terms as directed acyclic graphs, and recast the problem of unification as the construction of a certain kind of equivalence relation on terms.

A *term directed acyclic graph* (*term dag*) is a directed, acyclic graph whose nodes are labeled with function symbols or variables. Its outgoing edges from any node are ordered, and the outdegree of any node labeled with a function symbol f is equal to the arity of f. The outdegree of a node labeled with a variable is 0.In such a graph; each node has a natural interpretation as a term.

Terms are represented as term dags with *unique shared occurrence of variables*. For a given term, such a graph can be built in *O(n)* time, where n is the size of the term. The equations are imposed one after another as each of them is transformed into a term dag, representing the two terms that have to be unified. Each variable in the system is represented by a unique node in the graph corresponding to the system.

The algorithm constructs an equivalence relation between terms (nodes in the graph), where the equivalence classes consist of terms that are unified. Equivalence classes are represented as trees with nodes – the nodes of the term dag and edges – *forward links*, with a class representative at the root. To determine whether two terms are equivalent, it is only necessary to find the roots of the trees and check for identity. To join two equivalence classes, one class is made a subtree of the other's root using a forward link. To reduce the height of the trees, when following the forward links to the root to determine equivalence, the paths can be compressed by pointing all nodes encountered directly at the root.

We maintain a stack of pairs of terms to be unified.

If we have to unify the two terms s and t, we first follow the forward links to get the representatives of the respective equivalence classes. Several cases arise:

1. If the representative is the same, then the two terms are already unified and we are done. This is a generalization of the deletion rule. In this way, we avoid doing work that is already done.
2. If at least one of the representatives is a variable, then we merely add a forward link from a variable to the other representative. Thus, we unite the two equivalence classes, and connect the variable with the term that is substituted for it.

3. If both representatives are not variables and the labels of the nodes are different function symbols, then the equation s = t has no solution - there is a clash. The algorithm terminates with failure. This corresponds to the Difference Failure rule.
4. If both representatives are not variables and the labels of the nodes are the same function symbol, then we add a forward link from one of the terms to the other and in this way unite the two classes. Then we put on the stack the pairs of the respective children of the two nodes in the term dag. This is in fact the decomposition rule.

If all equations are imposed without a clash, we check the graph, obtained by adding the forward links to the term dag, for cycles. This pass through the graph to check for acyclicity replaces the repeated calls of the occurs check. If this graph contains a cycle, then the system is not solvable in finite trees, otherwise it is solvable, and we can extract the solved form following the forward links for the variables.
If the representation of the equivalence relation between the terms is implemented with an efficient union-find method, then the complexity of the algorithm is quasi-linear.

**2.1.4 Substitutions** *Substitutions* are functions from the set of variables into the set of terms over the alphabet ΣUV. Let σ be a substitution.

The *domain* of σ is the set of variables

$$\mathrm{dom}(\sigma) = \{x \mid \sigma(x) \neq x\} .$$

If dom(σ) is the finite set $\{x_1,\ldots,x_n\}$ then we can represent it explicitly as the set of bindings of the variables in its domain, e.g.,

$$\sigma = \{ x_1 := t_1 , \ldots , x_n := t_n \} .$$

We will consider only substitutions with finite domains.

The application of a substitution to a term is defined with induction on the structure of terms, analogously to the application of a tree-assignment to a term, but the result of the application is a term.

A substitution

$$\theta = \{ x_1 := t_1 , \ldots , x_n := t_n \}$$

is called *grounding* if $t_1,\ldots,t_n$ are ground terms. Grounding substitutions correspond to tree-assignments with all trees finite, since ground terms can be seen as finite trees.

Now we define the notion of solution of an equation system in terms of substitutions.

A *solution* θ of the equation set

$$E = \{ s_1 = t_1 , \ldots , s_n = t_n \}$$

is a grounding substitution such that

$$s_1 \theta \equiv t_1 \theta , \ldots , s_n \theta \equiv t_n \theta .$$

A more general notion is the notion of *unifier* of a system of equations. It is again a substitution such that

$$s_1 \theta \equiv t_1 \theta , \ldots , s_n \theta \equiv t_n \theta ,$$

but the terms that we substitute are allowed to contain variables.

A solved form equation system, equivalent to a given solvable equation system represents the set of all tree-solutions of the system. In a similar way, a unifier of a special kind of such a system represents the set of its solutions in the domain of finite trees.

The composition of two substitutions

$$\theta_1 = \{ x_1 := t_1 , \ldots , x_n := t_n \}$$

and

$$\theta_2 = \{ y_1 := s_1 , \ldots , y_m := s_m \}$$

is denoted $\theta_1 \circ \theta_2$. It is obtained from the set of bindings

$$\{x_1 := t_1 \theta_2 , \ldots , x_n := t_n \theta_2 , y_1 := s_1 , \ldots , y_m := s_m \}$$

by removing all elements of the form $y_k := s_k$, where $y_k \equiv x_j$ for some j, as well as all elements of the form $x_i := x_i$ .

A unifier $\mu$ is a *most general unifier* (or *mgu* for short) of the equation set E if $\varphi$ is a unifier for E iff there exists $\alpha$ such that,

$$\varphi = \mu \circ \alpha ,$$

i.e. every unifier of the system can be obtained by a most general unifier by a composition with another substitution.

The notion of most general unifier of a system of equations corresponds to the notion of solved form. From a solved form equation system equivalent to a given equation system we obtain in a natural way a most general unifier of the given system. From

$$E = \{ x_1 = t_1 , \ldots , x_n = t_n \}$$

we get

$$\theta = \{ x_1 := t_1 , \ldots , x_n := t_n \} .$$

### 2.2 Solving Systems of Equations with Domain the Set of Rational Trees

Now we are interested in solvability of an equation system over the set of infinite trees. It suffices to consider only the set of rational trees. In Section 1.3 we showed that there is an equation that has a tree-solution in the domain of rational trees but does not have a tree-solution of finite trees.

If the unification algorithm does not perform the “occur in” check, i.e. the unification of a variable with a term already containing this variable is allowed, the solution might be in infinite trees.

Again, there is one special type of equation systems, namely the *solved form*, but it differs from the solved form in the case of finite trees. This difference results from the

fact that an equation between a variable and a term containing this variable is solvable in rational trees.

**2.2.1 Solved Form** We say that the system of equations $\{x_1 = t_1 , \dots , x_n = t_n\}$ is in solved form if $x_1, \dots ,x_n$ are distinct variables and $t_1,\dots, t_n$ are arbitrary terms.

Note that the variables $x_1,\dots,x_n$ may appear in the right hand side of the equations.

**Theorem 2** Each system in solvable form has at least one tree-solution.

The following result is obvious:

A system of equations containing an equation of the form

$$f(s_1, \dots , s_n) = g(t_1, \dots ,t_m) ,$$

where f and g are different function symbols has no tree-solution. We call it *unsolvable form*.

**2.2.2 Simplification Rules** Now we introduce the five rules for transforming systems of equations, on which the algorithm for deciding solvability over the set of rational trees is based.

**Decomposition:**

$$f(t_1,\dots,t_n) = f(s_1,\dots,s_n)$$

is replaced by the equations

$$t_1 = s_1, \dots, t_n = s_n .$$

**Deletion:** If the system contains an equation

$$x = x ,$$

where x is a variable, delete this equation.

**Anteposition:** An equation

$$t = x ,$$

where x is a variable and t is not a variable is replaced by the equation

$$x = t .$$

**Variable Elimination:** If the system contains an equation

$$x = y ,$$

where x and y are distinct variables and x has another occurrence in the system then replace x by y in every other equation.

**Merge:** If x is a variable, $t_1$ and $t_2$ are not variables and

$$|t_1| \leq |t_2|$$

then replace

$$x = t_1, x = t_2$$

by

$$x = t_1, t_1 = t_2 .$$

By |t| we denote the size of the term t. The comparison of the sizes of the terms in this rule is necessary to ensure termination. However, this test can be avoided by introducing intermediate variables, so that every term contains no more than one functional symbol.

The rules Decomposition, Deletion and Anteposition are the same as in the case of finite trees. The rule Variable Elimination is different, since it allows only replacement of variable by another variable but not with an arbitrary term.

Easily is proven that these rules preserve the set of solutions of the system, i.e. if the equation system E´ is obtained by applying one of the above rules to an equation system E, then the two systems are equivalent.

Every repeated application of these transformations on an initial system, leads after a finite number of steps to a dead end where no transformation can be applied. Such a system we call *simplified form*. In other words again we have termination. The proof is similar to the proof of the termination for the solved form algorithm for finite trees and is given in [2].

The following result follows from the above observations about solved form and unsolvable form.

**Theorem 3** The simplified form has a solution iff each of the equations is of the form x = t where x is a variable.

**2.2.3 Simplification Algorithm** Start with the initial system of equations E. Nondeterministically choose a rule that can be applied to the current system and apply this rule. Since each sequence of such applications terminates finally we obtain a system E´, such that E´ is equivalent to E and E´ is a simplified form.

Let us assume that E´ contains an equation $t_1 = t_2$ where $t_1$ is not a variable. Then $t_2$ is also not a variable, since otherwise the Anteposition rule could be applied to E´. Thus, we have an equation of the form

$$f(\ldots) = g(\ldots).$$

f and g should be different function symbols, because otherwise the Decomposition rule could be applied. Hence, E´ is unsolvable form, and thus E is not solvable.

Consider the other case when all equations in E´ have variables at the left hand side. These variables should be distinct, since otherwise the Merge rule could be applied to E´. Thus, the system E´ is a solvable form and has at least one tree-solution. Hence, E is solvable. In this case E´ has the form:

$$x_1 = y_1$$

$$\dots$$

$$x_n = y_n$$

$$x_{n+1} = f(\dots)$$

$$\dots$$

$$x_m = g(\dots)$$

$x_1,\dots, x_n$ do not appear anywhere else in the system, since otherwise the variable elimination rule could be applied.

If there is no cycle of variables in the last (m – n) equations then there is a solution of E in finite trees, otherwise there is no solution of E in finite trees.

There is also a quasi-linear algorithm for deciding solvability of systems of equations over the set of rational trees. It is essentially the algorithm for solving equations over the set of finite trees using term dags, but without performing the acyclicity check at the end.

## 3. Systems of Equations and Disequations

Now we investigate systems containing both equations and disequations. First, we consider the case of deciding solvability of such systems over the class of finite trees and after that, we deal with the case of rational trees. Let us assume that the domain is infinite. It is obvious, that in the case of a finite domain, algorithms for deciding solvability of a system exist and that a suitable finite representation of the set of solutions is this set itself.

A tree-assignment σ is a *tree-solution* to a system consisting of a set of equations E and disequations

$$s_1 \neq t_1 , \dots ,s_n \neq t_n$$

if it is a tree-solution of E and it does not solve any of the equations

$$s_1 = t_1 , \dots ,s_n = t_n .$$

For each disequation

$$s \neq t$$

we define the corresponding *complementary equation* to be

$$s = t .$$

### 3.1. Solvability and Entailment

The following observation is obvious

**Proposition 1** A system consisting of a set of equations E and a single disequation $s \neq t$

- is not solvable in finite trees iff each tree-solution of finite trees of E is a solution of the complementary equation s = t, i.e. if E entails s = t.
- is not solvable in rational trees iff each tree-solution of E is a solution of the complementary equation s = t, i.e. if E entails s = t.

### 3.2. Independence Theorem

**Theorem 4** A system S consisting of a set of equations E and disequations $s_1 \neq t_1, \ldots, s_n \neq t_n$ is solvable iff each of the systems $E \cup \{s_i \neq t_i\}$ is solvable.

The proof of this theorem can be found in [4].

The above result allows us to consider each of the disequations separately from the other disequations in the system.

### 3.3. Algorithm for Finite Trees

In the case of finite trees, the set of all solutions to an equation system is represented by a solved form equivalent to the given system or by a most general unifier. Thus, it suffices to compute a solved form of the system of equations E, obtain the corresponding most general unifier and check whether it unifies the terms s and t.

If the solved form algorithm terminates with failure then the system E is unsolvable and thus the whole system is unsolvable.

Let θ be the most general unifier obtained from the solved form. If θ unifies s and t then each solution of E is a solution of the equation

$$s = t$$

and thus, the whole system does not have a solution.

If θ does not unify s and t then, since the domain is infinite, there is a solution of E that does not solve the complementary equation and thus the system is solvable.

Unlike the case of systems consisting only of equations between terms we have no finite representation of the set of the solutions of a system containing both equations and disequations if there are infinitely many function symbols in Σ.

**Theorem 5** Let there be an infinite number of function symbols in Σ. Apart from the trivial case when the system is unsolvable or its collection of disequations is redundant, no finite set of most general unifiers can provide an explicit representation for its solutions.

### 3.4 Algorithm for Rational Trees

In the case of infinite trees, we do not have the notion of most general unifier and hence we cannot use the algorithm for rational trees. However, there is an efficient algorithm for deciding solvability of such systems, which employs the unification algorithm using term dags. It is the following:
First, we impose all the equations. If there is a clash then the set of equations is not solvable and thus the whole system is not solvable.

If we impose all equations successfully, then we impose the complementary equation corresponding to the disequation. There are three possible cases:

1. If there is a clash while the complementary equation is being imposed, then no tree-solution of the set of equations is a tree-solution of the complementary equation, i.e. the complementary equation is disentailed by the set of equations, and thus the whole system is solvable.
2. If the complementary equation is successfully imposed and a forward link for some of the variables that appear in the equations is being introduced, then since the domain is infinite, there is a tree-solution of the set of equations that is not a tree-solution of the complementary equation, and thus the whole system is solvable.
3. If the complementary equation is successfully imposed without introducing a forward link for any of the variables that appear in the equations, then every tree-solution of the set of equations is a tree-solution of the complementary equation, i.e. the complementary equation is entailed by the set of equations, and thus the whole system is unsolvable.

Note, that the algorithm discovers clash as early as possible. Its time complexity is quasi-linear if an efficient union-find algorithm is used. It can also be modified to work in the case when equations and disequations are imposed in an arbitrary order. Moreover, the modified algorithm has the same worst case running time and is incremental, i.e. it avoids redoing work when further constraints arrive.

## 4. Conclusions

This paper is a brief overview of the algorithms for deciding solvability of systems of equations and systems, containing both equations and disequations between terms. We considered two possible domains for these constraint problems, namely the set of finite constructor trees and the set of rational tees. For system of equations we provided first the naïve approach that in both cases has an exponential worst case running time. We presented also a quasi-linear algorithm, similar to the one proposed by Huet in [3], based on the representation of terms as dags. Linear unification algorithms for the case of finite trees can be found in [5] and [6].There is also another type of trees-feature trees, that turns out to be very useful in logic programming. An incremental, quasi-linear algorithm for deciding solvability in this domain is given in [7].

## References


1. F. Baader and W. Snyder. Unification Theory, In *Handbook of Automated Reasoning*, Elsevier Science Publishers, pages 439-526, 2001.
2. A Colmerauer. Prolog and infinite trees. In K. Clark and S. –A. Tärnlund, editors, *Logic Programming*, pages 153-172, Academic Press, 1982.
3. G. Huet. Resolution d'equations dans des langages d'ordre 1, 2,…, ω. These de Doctorat d'Etat, l'Universite Paris VII, Sept. 1976.
4. J.-L. Lassez, M. J. Maher, and K. G. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 15, pages 587-625, Morgan Kaufman, 1988.
5. A. Martelli and U. Montanari. An efficient unification algorithm. *In ACM Transactions on Programming Languages and Systems* 4(2), pages 258-282, 1982.
6. M. S. Paterson and M. N. Wegman. Linear unification. In *Journal of Computer and System Sciences* 16(2), pages 158-167, 1978.
7. G. Smolka and R. Treinen. Records for Logic Programming. In *Journal of Logic Programming,* 18(3), 1994.