Constraints in Term Algebras (Short Survey)

Hubert Comon
CNRS and LRI, Bât. 490
Univ. Paris-Sud
91405 Orsay cedex, France

Unification, which consists in solving equations in the (free) term algebra, is known to be a fundamental operation in many areas of computer science and, in particular, in logic programming. *Disunification*, which consists in solving more complex formulae in the (free) term algebra, also revealed to be a fundamental operation (see [29, 13] for surveys on unification and disunification respectively). Recently, these computations have been seen as *constraint solving* in term algebras and this point of view is actually fruitful.

1 Constraints: a definition

A *constraint system* is defined by a *logical language* $\mathcal{C}$ (which is in practice a fragment of a first-order language), a *structure* M in which the formulae of $\mathcal{C}$ are interpreted and an algorithm which decides, for every $\phi \in \mathcal{C}$, whether ϕ is satisfiable in M or not. There are many examples: $\mathcal{C}$ can be a full first-order language, in which case, the third condition implies the decidability of the (first-order) theory of M. For example, the constraint system could correspond to Presburger arithmetic or the theory of real numbers. It could also be the theory of finite trees, since this theory has been shown decidable [35, 34, 18]. Many other examples will be given later.

Now, constraints can be (and have been) studied for their own mathematical interest. But, they can also be used to *constrain* other formulae. More precisely, given a logical language $\mathcal{L}$, a class of structures $\mathcal{S}$ and a satisfaction relation $\models$ on the one hand, and a constraint system $(\mathcal{C}, M, \Phi)$ on the other hand and given in addition, for each structure S in $\mathcal{S}$, a mapping H_S from the domain D of M into the domain D_S of the structure S, the *constrained logic* consists of

- the language of pairs of formulae (called *constrained formulae*) $\phi | C$ where $\phi \in \mathcal{L}$ and $C \in \mathcal{C}$
- a satisfaction relation defined as follows. Given an assignment σ of the free variables of C into D and an assignment θ of the free variables of ϕ into D_S,

$$\sigma, \theta, S \models \phi | C \quad \text{iff} \quad \begin{cases} M \models C\sigma \\ H_S(\sigma) \circ \theta, S \models \phi \end{cases}$$

where $H_S(\sigma)$ is the assignment which associates each variable x in the domain of σ with $H_S(x\sigma)$.

This definition is somewhat complicated and is not satisfactory in many respects, but everything simplifies when we consider constraints in *term structures* (also called *symbolic constraints*). Indeed, assuming that M is a term structure and that terms of $\mathcal{C}$ are also terms of $\mathcal{L}$, H_S can be (and will be) chosen as the interpretation defined by S. This means that we do no longer need M to define the meaning of a constrained formula:

$$\phi|C \text{ represents } \{\phi\sigma \mid \sigma, M \models C\}$$

a constrained formula is a shorthand for the (infinite) set of its instances corresponding to assignments of its free variables which satisfy the constraint. For example, $P(x,y)|x \neq y$ could represent the set of all formulae $P(t_1,t_2)$ where t_1 and t_2 are two distinct terms. This justifies the use of the symbol $|$ which can be read "such that": its use here does not differ from its use in set definitions (under the comprehension axiom).

Let us conclude these definitions with two remarks: first, this notion of "constraints" is coherent with what is used in practice in logic programming or artificial intelligence, but differs slightly from the "constraints" as they are used in the algebraic specification community. Let us also emphasize that constraints are different from conditions; consider for example the system

$$\begin{cases} f(x) = a \mid a = b \\ a = b \end{cases}$$

where a, b are two distinct constant symbols. Considering these formulae as constrained ones, in which the equality symbol is freely interpreted, the first constrained formula represents the set $\{f(x)\sigma = a\sigma \mid \sigma \models a = b\}$. But $a = b$ is unsatisfiable in the free algebra since a and b are distinct. Hence, this set is empty and the system collapses to the single equation $a = b$. On the contrary, if $|$ is seen as an implication $\leftarrow$, then it is possible to prove $a = b$ using the second equation, and hence, by modus ponens, we prove that $f(x) = a$.

2 On the use of constraints

It should be clear from the definition that (symbolic) constraints can enhance the expressiveness of a logical language, since they allow for a schematization of possibly infinite sets of formulae. This ability has been used in many situations:

- in constraint logic programming (e.g. [28])
- to construct (counter)models [7]
- to forbid particular instances [30]
- to express control strategies at the formulae level [40]
- to avoid the combinatorial explosion of semantic unification (e.g. [22])

Even when the set which is schematized by a constraint is not infinite, it can be very large and might be represented in a more compact way. For example, constraints may express structure sharing (see e.g. [32]): $f(x,x,x,x) \mid x = t$ where t is a big term is a compact representation of $f(t,t,t,t)$ avoiding duplications.

Constraints are also used to store irrelevant (from the computational point of view) parts of a formula. This is the case for equational constraints and the so-called *basic strategy* (see [2, 39] for recent developments).

Finally, constraints are interpreted in a fixed domain for which dedicated efficient solving techniques can be used. Hence, it is worth characterizing some decidable fragment in a logic programming language and leave it as constraint, thus taking advantage of the knowledge on the domain (see [23] for an example of automatic detection of such fragments). This was originally one of the main motivations for adding constraints into logic programming languages: for example linear equations over the integers are not solved using the logic programming langage (which is in principle possible), but rather using another syntax and another deduction mechanism. And, as shown in [28], constraint systems can be nicely combined with any logic programming language (without equality).

3 Examples of symbolic constraints

3.1 Equations

The most well-known example of symbolic constraints is given by unification problems. In such a constraint system, the logical language consists of (disjunction of existentially quantified) conjunctions of equations between terms. The equations are interpreted in the free term algebra $T(F)$ (this is the classical interpretation) or in some quotient $T(F)/_{=_E}$ by a finitely generated congruence $=_E$. Using these constraints in logic programs or automated deductions prevents applying substitutions, which may be an expensive operation in case of duplications. That is why they are used since the very beginning in logic programming. In case of interpretations in quotient algebras, equations are also more relevant than unifiers since there might be a very large minimal complete set of unifiers (doubly exponential w.r.t. the size of the equations, in the case of associative-commutative (AC) function symbols), whereas the satisfiability of an equation system is much simpler (NP-complete in the case of AC symbols) [32]. We will not survey the equational theories for which unification is decidable, but refer to [29] instead.

Several extensions of the syntax, for example mixing integer expressions and terms, allow one to schematize infinite sets of terms (as constraints do). Solving equations over this extended syntax has been investigated in several recent papers [9, 15, 43] with applications in logic programming.

3.2 Equational formulae

More generally, *equational formulae* are arbitrary first-order formulae over an alphabet F of function symbols and the equality predicate symbol. Assuming that they are interpreted in the free term algebra, there are several decision techniques which lead to complete axiomatizations of the algebra of finite trees (see [35, 33, 34, 18, 36] and others). These axiomatizations differ, depending on the finiteness of F: when F is finite, the complete axiomatization consists of what is known as “Clark's axioms of equality” plus the *domain closure axiom*

$$\forall x, \bigvee_{f \in F} \exists \vec{x_f}. x = f(\vec{x_f}).$$

Such formulae have been used for solving various problems in rewriting theory (see [13]) and as a constraint system in automated theorem proving, searching simultaneously for a proof and for a counter-model [7].

Equational formulae can be generalized in various directions. One of them consists in adding *sort constraints*, which is studied in the next section. Another generalization consists in considering non-free interpretations: we assume a finite set of equational axioms E, generating a congruence $=_E$ on the term algebra. For example, some symbols can be assumed commutative, or associative and commutative. The first-order theory of a quotient algebra $T(F)/_{=_E}$ quickly becomes undecidable: a single associative symbol suffices [41], or an associative-commutative symbol [44]. Decidability results include the case where E is a set of flat permutative axioms [35], ground axioms [12] and E is a set of *shallow* equations, a class which encompasses the two previous ones [17].

3.3 Feature constraints

Feature trees have been introduced in order to represent linguistic information. These are rational trees whose nodes are labeled with sorts (out of an infinite set of symbols) and whose edges are labeled with *features* out of another infinite set of symbols. Contrary to constructor trees, each node may have an arbitrary number of unordered sons. The logic of feature trees (as in [5], but there are also other feature logics with different atomic formulae) involve unary predicates Ax (saying that the tree has sort A at its root), binary predicates xfy (y is a direct subtree of x, the edge relating x and y being labeled with f), unary predicates xf (x has feature f) and equality. There is however no function symbol in the logic. There are many results about feature constraints. We refer to [5] for a recent result: a complete axiomatization of the theory of feature trees.

3.4 Membership constraints

(Symbolic) constraints actually define sets of assignments which satisfy the constraints. Instead of using formulae, these sets can be defined by means of automata or grammars. For example, order-sorted signatures [25] (which are nothing but finite bottom-up tree automata) are used to constrain variables to belong to recognizable sets of trees. However, this is often hidden in the order-sorted algebra framework. The point of view “sorts as membership constraints” allows for a number of improvements. For example, there is no need of “regularity” (often required in the order-sorted framework) because unifiers are replaced with solved forms which may involve membership constraints. Another example is the design of critical pair lemmas for order-sorted rewriting [14].

Membership constraints consist of an (infinite) family of membership predicate symbols $\in \zeta$ which are interpreted as recognizable subsets of the term algebra $T(F)$. The satisfiability of equational formulae remains decidable with this additional construction [10]. These formulae have been used for solving problems in rewriting theory (e.g. "sufficient completeness" and "inductive reducibility" [13, 10]).

3.5 Ordering constraints

Ordering constraints are used as a means for expressing ordered strategies. Actually, using ordering constraints solves the old problem of non-orientable equations in rewriting theory by considering $s = t$ as two constrained rewrite rules $s \to t \mid s > t$ and $t \to s \mid t > s$. In other words, the equation can be considered as a rule from left to right or as a rule from right to left, depending on the instance which is considered. Moreover, the constraint keeps track of which instances remain to be considered in further deductions (see [40] for a completeness result in a more general setting).

Here, the logical language consists of purely existential formulae, using a set of function symbols F and the two predicate symbols $=$ and $\geq$. Several interpretations of the ordering have been considered:

- Venkataraman in [45] interprets $\geq$ as a subterm ordering, showing the decidability of the system (and undecidability of the first-order theory). However, such an ordering is useless for applications in rewriting theory, since it is not compatible with the term structure.

- The adequate orderings for the applications in automated deduction are the *reduction orderings* (see [20]) which are total on ground terms. A typical example of such an ordering is the *lexicographic path ordering* extending a total precedence, whose existential fragment has been shown decidable [11] (it is actually NP-complete [38]). This result has been extended to other total recursive path (quasi-)orderings [31]. The decidability of the full first-order theory of these orderings is an open question (problem 24 in [21]).

- The theory of partial recursive path orderings appears to be even more difficult; the Σ_4 fragment has been shown undecidable [44]. The decidability of the existential fragment of any such ordering is open. The only hint for this problem is the recent result of [6]: the *positive* existential fragment of the theory of tree embedding is decidable. (Tree embedding is the most simple recursive path ordering: it is the intersection of all simplification orderings).

- Interpreting $\geq$ as *encompassment* (a term t encompasses u if there is an instance of u which is a subterm of t), it is possible to express some properties such as *inductive reducibility* or (sometimes) *sufficient completeness* using first order formulae (see [8]). The first-order theory of a finite number of unary predicate symbols of the form $\geq t_i$ has been shown decidable in [8].

3.6 Set constraints

Many other symbolic constraints have been studied. But it is a too long work to list all of them. Let us conclude with *set constraints* for which many recent beautiful results have been obtained. (And there is still some work to do!). A *set expression* is built from a finite alphabet of function symbols, set variables and the intersection, union an complement symbols. Then, set constraints are finite conjunctions of formulae $e \subseteq e'$ where e, e' are set expressions. These formulae are interpreted assigning set variables to subsets of the term algebra $T(F)$.

Such constraints have been used for the analysis of logic and functional programs (see [26, 1, 23]). The case of *definite* constraints has been solved in [26] and the general case has been further studied by quite different means in [1, 3, 24]. There are two extensions which are still under investigation: adding negative constraints of the form $e \not\subseteq e'$ and adding the *projection* construction, which consists roughly of the inverse of applying a function symbol (see [26]). These extensions have been conjectured decidable.

4 Constraint solving

To *solve* a constraint not only means to decide its satisfiability. More precisely, a constraint solving algorithm is specified by:

- the constraint system $(\mathcal{C}, M, \Phi)$
- a subset $\mathcal{S}$ of $\mathcal{C}$ called the set of *solved forms*

$\mathcal{S}$ has to fulfill some requirements (see [13]), in particular, every formula of $\mathcal{C}$ should be equivalent (in M) to a solved form, and every solved form should be trivially satisfiable or trivially unsatisfiable. However, there is still some room for choosing the solved forms. For example, in the case of unification (in free algebras), one can choose *tree solved forms* or *DAG solved forms* as explained in [29].

Once solved forms have been chosen, constraint solving algorithms can be designed using rewriting techniques. A set of rewrite rules on formulae has to be given, their correctness proved (every formula in $\mathcal{C}$ is rewritten to an equivalent formula w.r.t. M), as well as termination (*any* rewriting sequence is finite) and completeness (every irreducible formula is a solved form). There are several advantages for this method:

- The rules can be redundant (and this is actually a desirable property). Then the termination proof may be complex, but it yields termination proof for all algorithms obtained by making the control deterministic. For example (as we will see below) tree solved forms (Robinson's unification algorithm [42, 27]) and DAG solved forms (corresponding to Martelli and Montanari's unification algorithm [37]) are obtained by strengthening the control on the same set of rules.
- There is a feed-back on the theory, since the rewrite rules are actually an axiomatization of M (see [13]).
- The constraint solving algorithms are automatically *incremental* in the following sense: in order to solve $\phi \wedge \psi$, it is possible to use the result of solving ψ.
- We expect to use rewriting tools for proving termination of the constraint solving rules, as we try to show in the following example.

A toy example

We consider the classical unification problems: formulae are conjunctions of equations between terms; they are interpreted in the free term algebra $T(F, X)$. The equality symbol is considered as symmetric (i.e. there is no difference between $s = t$ and $t = s$).

Given a conjunction of equations ϕ, the *occur-check relation* $\geq_\phi$ is the relation on the free variables of ϕ defined as the smallest reflexive-transitive relation which contains $x \geq_\phi y$ as soon as there is an equation $x = t[y]$ in ϕ. (The notation $t[y]$ means that y occurs in t). A variable is *solved* in ϕ if it has only one occurrence, as a member of an equation of ϕ. Let $U(\phi)$ be the set of unsolved variables of ϕ.

Now consider the scheme of rules for unification given below:

Decompose	$f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$	$\to$	$s_1 = t_1 \wedge \ldots \wedge s_n = t_n$
Coalesce	$x = y \wedge P$	$\to$	$x = y \wedge P\{x \mapsto y\}$
			If $x \neq y$ and $x, y \in U(P \wedge x = y)$
Clash	$f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$	$\to$	$\perp$
			If $f \neq g$
Eliminate	$x = s \wedge P$	$\to$	$x = s \wedge P\{x \mapsto s\}$
			If $x \in Var(P)$, $x \notin Var(s)$, $s \notin X$
Check*	$x_1 = t_1[x_2]_{p_1} \wedge \ldots \wedge x_n = t_n[x_1]_{p_n}$	$\to$	$\perp$
			If there is an i such that $p_i \neq \Lambda$
Trivial	$s = s$	$\to$	$\top$
Merge	$x = s \wedge x = t$	$\to$	$x = s \wedge s = t$

If **Decompose**, **Coalesce**, **Check*** do not apply, $s, t \notin X$ and x is maximal w.r.t. $\geq_\phi$ among the variables occuring at least twice as a member of an equation

In these rules we use some classical notations of [20], some of which are recalled below. *Positions* are strings of natural numbers and terms can be viewed as mappings from positions to symbols of F. The empty string is denoted Λ (the symbol at position Λ in a term t is the *root symbol* of t). $t|_p$ is the subterm of t at position p and the notation $t[x]_p$ is used either to indicate that $t|_p = x$ or for the term obtained by replacing $t|_p$ with x. $\{x_1 \mapsto t_1; \ldots; x_n \mapsto t_n\}$ is the substitution replacing the distinct variables $x_1, \ldots, x_n$ with $t_1, \ldots, t_n$ respectively. $Var(e)$ is the set of variable symbols occurring in e (whatever e can be).

Note that in these rules, we relax the classical condition on the sizes: $|s| \leq |t|$ in the merge rule (see [29]) and put instead a condition of maximality on x and assume the system decomposed. Whether these conditions can be relaxed without losing termination was stated as open problem 39 in [21]. We also assume here that there are structural rules for $\wedge$: $\perp \wedge P \to \perp$, $\top \wedge P \to P$ and $P \wedge P \to P$. Moreover $\wedge$ is assumed to be associative and commutative.

The rule system is terminating (modulo the associativity-commutativity of $\wedge$ and the commutativity of $=$). For, we use the *associative path ordering* [4] on formulae[1], extending the precedence on $F \cup X$ defined by:

[1] We don't recall here the definition of associative path ordering. In this particular case, since we only have one AC symbol ($\wedge$) it coincides with the *recursive path ordering* [19] where all function symbols have a multiset status.

- every variable[2] is larger than any function symbol
- every function symbol is larger than $=$ which is in turn larger than $\wedge$
- variables are compared according to the occur check relation

This last statement has to be made precise since the occur-check relation actually depends on the formula which is considered. In fact, we consider the (maybe infinite) union of all occur-check relations at any step in the computation. This definition depends on the transformation, but it does not depend on a particular formula ϕ, and we need not compute effectively this relation. It may happen that variables are *equivalent* w.r.t. this relation, in which case, they are considered as identical from the associative path ordering point of view.

Note that the associative path ordering has the subterm property and it is monotonic (see [4]). Hence, for proving the termination, we only have to prove that every left hand side of a rule is (strictly) larger than the corresponding right-hand-side:

- For the structural rules and for **Trivial**, **Check*** and **Clash** the decreasingness is obvious.
- **Decompose** is strictly decreasing because $=> \wedge$ in the precedence and

  $$f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) >_{apo} s_i = t_i$$

  by monotonicity and the subterm property.
- **Eliminate** is strictly decreasing because x is strictly larger than the variables of s (it is larger by definition, and it cannot be equivalent to any variable of s since x becomes solved after applying the rule, hence no rule can produce an equation with x in its right hand side). Moreover, variables are larger than function symbols in the precedence, which shows that $x >_{apo} s$.
- **Merge** is strictly decreasing, for the same reason as above: since x is assumed to be maximal in the decomposed system, it cannot be smaller than any variable of s or t, even after further transformations.
- **Coalesce** keeps the problem equivalent w.r.t. $\geq_{apo}$ since x and y are equivalent in the precedence. However, it can only be applied a finite number of times. Hence we can reason modulo this rule, i.e. modulo the strict equivalence on variables.

Now, the system is terminating. If we remove the **Merge** rule, the system is complete w.r.t. tree solved forms and we can find Robinson's unification algorithm as an instance. If we remove the **Eliminate** rule, the system is complete w.r.t. DAG solved forms and we can find an instance of Martelli and Montanari's unification algorithm.

Similar techniques have been applied for the termination proofs of more powerful constraint systems [16, 14].

[2] Be careful that variables of the unification problem are seen as constants in the rewriting process! Only the *logical* variables can be instantiated.

Acknowledgments

Thanks to A. Boudet and J.-P. Jouannaud for reading a former version of this paper.

References

[1] A. Aiken and E. Wimmers. Solving systems of set constraints. In *Proc. 7th IEEE Symp. on Logic in Computer Science*, Santa Cruz, CA, 1992.

[2] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation and superposition. In D. Kapur, editor, *Proc. 11th Int. Conf. on Automated Deduction, Saratoga Springs, NY, LNCS 607*. Springer-Verlag, June 1992.

[3] L. Bachmair, H. Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Proc. 8th IEEE Symp. Logic in Computer Science, Montréal*, 1993.

[4] L. Bachmair and D. A. Plaisted. Termination orderings for associative-commutative rewriting systems. *Journal of Symbolic Computation*, 1(4):329–349, Dec. 1985.

[5] R. Backofen and G. Smolka. A complete and recursive feature theory. Research Report RR–92–30, DFKI, Saarbrücken, Sept. 1992.

[6] A. Boudet and H. Comon. About the theory of tree embedding. In *Proc. CAAP 93*, 1993. LNCS 668.

[7] R. Caferra and N. Zabel. A method for simultaneous search for refutations and models by equational constraint solving. *Journal of Symbolic Computation*, 13(6):613–642, June 1992.

[8] A.-C. Caron, J.-L. Coquidé, and M. Dauchet. Encompassment properties and automata with constraints. In *Proc. RTA 93*, 1993.

[9] H. Chen and J. Hsiang. Logic programming with recurrence domains. In *Proc. 18th Int. Coll. on Automata, Languages and Programming, Madrid, LNCS 510*, 1991.

[10] H. Comon. Equational formulas in order-sorted algebras. In *Proc. 17th Int. Coll. on Automata, Languages and Programming, Warwick, LNCS 443*, Warwick, July 1990. Springer-Verlag.

[11] H. Comon. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Science*, 1(4):387–411, 1990.

[12] H. Comon. Complete axiomatizations of some quotient term algebras. In *Proc. 18th Int. Coll. on Automata, Languages and Programming, Madrid, LNCS 510*, July 1991.

[13] H. Comon. Disunification: a survey. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, 1991.

[14] H. Comon. Completion of rewrite systems with membership constraints. In W. Kuich, editor, *Proc. 19th Int. Coll. on Automata, Languages and Programming, LNCS 623*, Vienna, 1992. Springer-Verlag. An extended version is available as LRI Research Report number 699, Sept. 1991.

[15] H. Comon. On unification of terms with integer exponents. Research Report 770, L.R.I, Univ. Paris-Sud, Orsay, Aug. 1992. To appear in *Mathematical Systems Theory.*

[16] H. Comon and C. Delor. Equational formulas with membership constraints. Technical report, Laboratoire de Recherche en informatique, Mar. 1991. To appear in Information and Computation.

[17] H. Comon, M. Haberstrau, and J.-P. Jouannaud. Decidable properties of shallow equational theories. In *Proc. 7th IEEE Symp. Logic in Computer Science*, Santa Cruz, 1992. Also Research Report 718, Dec. 1991, Laboratoire de Recherche en Informatique, Orsay, France.

[18] H. Comon and P. Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7:371–425, 1989.

[19] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1):69–115, Feb. 1987.

[20] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–309. North-Holland, 1990.

[21] N. Dershowitz, J.-P. Jouannaud, and J. W. Klop. Open problems in rewriting. Technical report, CWI, Amsterdam, Feb. 1991.

[22] E. Domenjoud. AC unification through order-sorted AC1 unification. *Journal of Symbolic Computation*, 14(6):537–556, Dec. 1992.

[23] T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. 6th IEEE Symp. Logic in Computer Science, Amsterdam*, pages 300–309, 1991.

[24] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints using tree automata. In *Proc. 10th Symposium on Theoretical Aspects of Computer Science, Würzburg, LNCS*, 1993.

[25] J. Goguen and J. Meseguer. Order-sorted algebra I: Partial and overloaded operators, errors and inheritance. Draft, Computer Science Lab., SRI International, 1987.

[26] N. Heintze and J. Jaffar. A decision procedure for a class of set constraints. In *Proc. 5th IEEE Symp. Logic in Computer Science, Philadelphia*, June 1990.

[27] J. Herbrand. Recherches sur la théorie de la démonstration. Thèse d'Etat, Univ. Paris, 1930. Also in: Ecrits logiques de Jacques Herbrand, PUF, Paris, 1968.

[28] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. 14th ACM Symp. Principles of Programming Languages, Munich*, 1987.

[29] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*. MIT-Press, 1991.

[30] J.-P. Jouannaud and C. Marché. Termination and completion modulo associativity, commutativity and identity. *Theoretical Comput. Sci.*, 104:29–51, 1992.

[31] J.-P. Jouannaud and M. Okada. Satisfiability of systems of ordinal notations with the subterm property is decidable. In *Proc. 18th Int. Coll. on Automata, Languages and Programming, Madrid, LNCS 510*, 1991.

[32] C. Kirchner, H. Kirchner, and M. Rusinowitch. Deduction with symbolic constraints. *Revue Française d'Intelligence Artificielle*, 4(3):9–52, 1990. Special issue on automatic deduction.

[33] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.

[34] M. J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proc. 3rd IEEE Symp. Logic in Computer Science, Edinburgh*, pages 348–357, July 1988.

[35] A. I. Mal'cev. Axiomatizable classes of locally free algebras of various types. In *The Metamathematics of Algebraic Systems. Collected Papers. 1936-1967*, pages 262–289. North-Holland, 1971.

[36] G. Marongiu and S. Tulipani. Decidability results for term algebras. Preprint 9, AILA, 1991.

[37] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, Apr. 1982.

[38] R. Nieuwenhuis. Simple LPO-constraint solving methods. Research Report LSI–93–9–R, Departament de Llenguatges i sistemes informàtics, Universitat Politècnica de Catalunya, 1993. To appear in Information Processing Letters.

[39] R. Nieuwenhuis and A. Rubio. Basic superposition is complete. In B. Krieg-Bruckner, editor, *Proc. European Symp. on Programming, LNCS 582*, pages 371–389, Rennes, 1992. Springer-Verlag.

[40] R. Nieuwenhuis and A. Rubio. Theorem proving with ordering constrained clauses. In D. Kapur, editor, *Proc. 11th Int. Conf. on Automated Deduction, Saratoga Springs, NY, LNCS 607*. Springer-Verlag, June 1992.

[41] W. V. Quine. Concatenation as a basis for arithmetic. *Journal of Symbolic Logic*, 11(4), 1946.

[42] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[43] G. Salzer. On unification of infinite sets of terms and its applications. In Voronkov, editor, *Proc. LPAR 92, LNCS 624*, pages 409–421, 1992.

[44] R. Treinen. A new method for undecidability proofs of first order theories. *Journal of Symbolic Computation*, 14(5):437–458, Nov. 1992.

[45] K. N. Venkataraman. Decidability of the purely existential fragment of the theory of term algebras. *J. ACM*, 34(2):492–510, 1987.