DALL'UNIFICAZIONE INSIEMISTICA AI VINCOLI SU INSIEMI
FROM SET UNIFICATION TO SET CONSTRAINTS

Gianfranco Rossi

SOMMARIO/*ABSTRACT*

In questo articolo riassumiamo brevemente alcuni dei problemi più interessanti che nascono quando si permette di trattare gli insiemi come oggetti di "prima classe" in un linguaggio logico, dall'unificazione di insiemi ben-fondati e non alla soluzioni di vincoli su insiemi.

In this paper, we briefly summarize some of the most challenging issues that arise when allowing sets to be dealt with as first-class objects in a logic language, ranging from set unification of well-founded and non-well-founded sets to set constraint solving.

Keywords: Set unification, hypersets, set constraints.

1 Introduction

Sets are familiar mathematical objects, and they are often used as a high-level abstraction to represent complex data structures, whenever the order and repetitions of elements are immaterial.

In the last two decades, a number of proposals have emerged where sets are dealt with as primitive objects of a (first-order) logic language. In this context, sets are often represented as first-order terms, called *set terms*, built from symbols of a suitable alphabet, using selected function symbols as set constructors. Furthermore, the language usually provides the typical set-theoretic operations to manipulate set objects.

These short notes summarize some of the most challenging issues arising when manipulating finite sets in a logic language. In Section 2 we briefly review *set unification*, i.e., the key problem of solving equations between set terms. In Section 3 the unification problem is extended to the case of non-well-founded sets. In Section 4 we introduce set constraints as a way to allow set-theoretic operations other than set equality to be taken into account. Finally, in Section 5 we briefly review proposals aiming at making set constraint solving more effective.

2 Set Unification

Intuitively, the set unification problem is the problem of computing (or simply testing the existence of) an assignment of values to the variables occurring in two set terms which makes them denote the same set.

Various forms of set unification have been used in various application areas, such as (see [13]): deductive databases, AI and its various sub-fields (e.g., Automated Deduction and Natural Language Processing), program analysis and security, declarative programming languages with sets.

Set unification can be thought of as an instance of E-unification, i.e., unification modulo an equational theory E, where the identities in E capture the properties of set terms—i.e., the fact that the ordering and repetitions of elements in a set are immaterial.

The equational theory E is strongly related to the representation adopted for set terms. Two main approaches have been presented in the literature: the *union-based representation*, and the *list-like representation*. The union-based representation makes use of the union operator ($\cup$) to construct sets. This representation has been often used when dealing with the problem of set unification on its own, where set unification is dealt with as an ACI unification problem—i.e., unification in presence of operators satisfying the *Associativity*, *Commutativity*, and *Idempotence* properties (e.g., [6]).

The list-like representation builds sets using an *element insertion* constructor (typically denoted by $\{\cdot \mid \cdot\}$). With this approach, the finite set $\{t_0, \ldots, t_n\}$ is represented by a sequence of element insertions

$$\{t_0 \mid \{\cdots \{t_n \mid \emptyset\} \cdots\}\},$$

where $t_0, \ldots, t_n$ are either individuals or sets. While this representation restricts the number of set variables which can occur in each set term to one, on the other hand it allows sets to be viewed and manipulated in a fashion sim-

ilar to *lists*. As a matter of fact, this representation has been adopted in a number of logic and functional-logic programming languages with sets (e.g., $CLP(\mathcal{SET})$ [10]).

Various authors have investigated the problem of set unification using the list-like representation [3, 12, 23, 8]. In particular, the algorithm presented in [9] considers an equational theory E containing the two identities (Ab) and $(C\ell)$ stating the fundamental properties of the set constructor $\{\cdot \mid \cdot\}$:

$$\begin{array}{lrcl} (Ab) & \{X \mid \{X \mid Z\}\} & \approx & \{X \mid Z\} \\ (C\ell) & \{X \mid \{Y \mid Z\}\} & \approx & \{Y \mid \{X \mid Z\}\}. \end{array}$$

The core of the unification algorithm is very similar in structure to the traditional unification algorithms for standard Herbrand terms (e.g., [20]). The main difference is represented by the reduction of equations between set terms, $\{Y_1 \mid V_1\} = \{Y_2 \mid V_2\}$. The algorithm allows also to account for equations of the form $X = \{t_0, \ldots, t_n \mid X\}$, with $X \notin vars(t_0, \ldots, t_n)$, which turns out to be satisfiable for any X containing $t_0, \ldots, t_n$ thanks to (Ab) and $(C\ell)$). As an example, given the set unification problem

$$\{X \mid S\} = \{1, 2\}$$

(where $\{1, 2\}$ is a syntactic shorthand for $\{1 \mid \{2 \mid \emptyset\}\}$) the algorithm non-deterministically computes the following (complete) set of solutions: $X = 1 \wedge S = \{2\}$, $X = 1 \wedge S = \{1, 2\}$, $X = 2 \wedge S = \{1\}$, $X = 2 \wedge S = \{1, 2\}$.

A general survey of the problem of unification in presence of sets, across different set representations and different admissible classes of set terms, can be found in [13].

The computational complexity properties of the set unification have been investigated by Kapur and Narendran [18], who established that these decision problems are NP-complete. Complexity of the set unification operation, however, depends on which forms of set terms (e.g., flat or nested sets, with zero, one, or more set variables) are allowed. The form of set terms in turn is influenced by the set constructors used to build them. Thus, different complexity results can be obtained for different classes of set terms. For instance, while the set equivalence test of ground set terms denoting flat sets, such as $\{a, b, c\}$ and $\{b, c, a\}$, is rather easy, when the decision problem deals with nested set terms involving variables it becomes NP-complete.

Various authors have considered simplified versions of the $(Ab)(C\ell)$ problem obtained by imposing restrictions on the form of the set terms. In particular, various works have been proposed to study the simpler cases of matching[1] and unification of *Bound Simple* set terms, i.e., bound set terms of the form $\{s_1, \ldots, s_n\}$, where each s_i is either a constant or a variable [4, 16].

[1] *Set matching* can be seen as a special case of set unification, where variables are allowed to occur in only one of the two set terms which are compared.

3 Hypersets

Sets considered so far are the so called *hereditarily finite* sets, i.e. sets with a finite number of elements, all of which are themselves hereditarily finite. This definition leaves still a further possibility for infinity. Let us consider the sets x and y that satisfy the equations $x = \{\emptyset, y\}, y = \{x\}$. They are hereditarily finite, but they hide an infinite descending chain $x \ni y \ni x \ni y \ni \cdots$. These sets in which, roughly speaking, membership can form cycles are called *non-well-founded sets* (or *hypersets*). Hypersets are very important in some areas, such as concurrency theory, but hyperset theory has been applied in a number of areas of logic, linguistics, and computer science, as well.

Introducing hypersets as a data structure in a logic programming language requires a unification algorithm that is able to deal with objects denoting hypersets. All set unification algorithms cited in the previous section, however, consider well-founded sets only.

An hyperset unification algorithm is shown in [1]. The key idea underlying this algorithm is that of enlarging the domain of discourse from terms (i.e., finite trees) over the signature Σ to *directed labeled graphs* over Σ, possibly with cycles. This data structure, when involving the interpreted function symbol $\{\cdot \mid \cdot\}$ used as the set constructor, can be regarded as a convenient way to denote hypersets. For instance, a solution to the equation $X = \{X\}$ is a cyclic graph which can be interpreted as an hyperset containing itself as its only element. In addition, a notion of *bisimulation* which applies to this kind of graphs is defined and the interpretation domain is taken as the set of directed labeled graphs over Σ modulo the equivalence relation induced by bisimulation.

The algorithm in [1] can be seen as an adaptation of the set unification algorithm of [9]. Many of the changes required to move from set to hyperset unification are the same needed when moving from standard unification to unification over (uninterpreted) *rational trees*, for which a number of efficient algorithms have been proposed in the literature (e.g., [21]). In particular the hyperset unification algorithm in [1] works on Herbrand systems of equations, avoiding full variable substitution and adding simple non-membership constraints to avoid the possibly endless repeated insertions of the same elements into hypersets.

4 Set Constraints

The algorithms cited above focus only on *equality* between set terms. Besides equality, however, other basic set operations, such as membership, inclusion, union, etc., are usually required for dealing with sets in a more general way.

A number of proposals have been put forward in the last fifteen years in which general set-based formulae are considered and procedures to check their consistency are developed. Most of these proposals have emerged in the context of Constraint (Logic) Programming (see, e.g., [15]).

In this context, set-theoretical operations are conveniently dealt with as *(set) constraints*, that is arbitrary conjunctions of positive and negative atomic predicates built using a fixed finite set of predicate symbols denoting set-theoretical operations, whose variables can range over the domain of sets. Systems of (set) constraints are solved as a whole by suitable *(set) constraint solvers*, which are able to reduce the given constraints either to false or to a simplified form from which it is easier to obtain a solution (i.e., a substitution for the variables in the given constraints that make them satisfiable in the selected interpretation). For example,

$$X \in S \wedge T = S \cup R \wedge X \notin T$$

is a set constraint, where R, S, and T are set variables, that the set constraint solver can reduce to false.

Set based formalisms allow a natural formulation of a number of problems, in quite different areas: combinatorial search problems, warehouse location problems, diagnostic related problems (e.g., VLSI circuit verification), program analysis, network design problems (e.g., weight setting). Dealing with such formulations as constraints allow, on the one hand, to solve these problems even if not all sets are (fully) known a priori, and, on the other hand, to compute solutions efficiently, provided constraint reasoning enables the solver to prune the search space.

As an example, the following is a very compact formulation as a set constraint of the well-known map coloring problem for a map of three regions, $R1$, $R2$, $R3$ (where $R1$ borders $R2$ and $R2$ borders $R3$), using two colors, $c1$ and $c2$:

$$\{R1, R2, R3\} = \{c1, c2\} \wedge M = \{\{R1, R2\}, \{R2, R3\}\} \wedge \{c1\} \notin M \wedge \{c2\} \notin M.$$

A complete set constraint solver, i.e., one which is always able to decide if a given constraint is satisfiable or not is presented in [10]. The constraint language is based on constructed sets using the list-like representation and it provides the usual set-theoretic operations as primitive constraints. Sets are allowed to be nested and to contain unknown elements (i.e., uninstantiated logical variables). The constraint solver rewrites any given constraint C into an equi-satisfiable disjunction of constraints in *solved form*—proved to be correct and terminating. In particular the solver uses the set unification algorithm developed in [9] to deal with (set) equalities. A constraint in solved form is guaranteed to be satisfiable in the corresponding structure. Therefore the ability to obtain a solved form guarantees that the original constraint is satisfiable.

This constraint structure has been exploited to obtain a specific instance of the general Constraint Logic Programming scheme, called CLP($\mathcal{SET}$) [10]. A Java implementation of (most of) CLP($\mathcal{SET}$) facilities for set management has been recently developed and made available as part of a Java library, called JSetL [22], intended to support declarative programming in an object-oriented language.

The study of set constraints is strongly related to work in the so-called *Computable Set Theory* area (C.S.T.), a fruitful research stream born in the 1970's at the New York University thanks to the initial ideas and subsequent stimulus of J. T. Schwartz (see [7] for a general survey). Work in C.S.T. has identified increasingly larger classes of computable formulae of suitable sub-theories of the general Zermelo-Fraenkel set-theory for which satisfiability is decidable. Further extensions of these classes are still under investigation at present. Recent related work is described in [19]. However, efforts in this area are mainly concerned with decidability results, rather than computing solutions like it is usually required in constraint programming.

Other classes of aggregates (akin to sets) have also been considered in the literature. In particular, various frameworks have introduced the use of *multisets* where repeated elements are allowed to appear in the collection. An analysis of the problems concerned with the introduction of multisets—as well as sets and lists—is reported in [11, 12].

5 Efficient Set Constraint Solving

The proposals for (general) set constraints cited above do not take into account efficiency adequately to allow them to be effectively applied in many concrete applications. For example the CLP($\mathcal{SET}$) solvers often use a generate & test approach, that non-deterministically assigns values to variables as soon as those values are available. For instance, given the constraint $X \in \{1, 2, 3, 4, 5\} \wedge X \neq 10$, the CLP($\mathcal{SET}$) solver enumerates all possible values of X before asserting that the constraint holds.

A number of proposals have been developed in the last fifteen years that consider more restricted forms of set constraints but equipped with constraint solving techniques that allow them to be processed in a quite more effective way. Works along these lines include [5, 14, 17].

In these proposals constraint variables have a *finite domain* attached to them. In the case of set constraints, the domain is a collection of sets, usually specified as a *set interval* $[l, u]$, where l and u are known sets (typically, of integers). $[l, u]$ represents a lattice of sets induced by the subset partial ordering relation $\subseteq$ having l and u as the greatest lower bound and the least upper bound, respectively. The constraint solver exploits the information that the domain of variables provides to efficiently compute simplified forms of the original constraint or to detect failures. In its simplest form, the solver uses a local propagation algorithm that attempts to enforce consistency on the values in the variable domains by removing values that cannot form part of a solution to the system of constraints. For example, given the set constraint

$$S \in \{1\}..\{1, 2, 3, 4\} \wedge X \subseteq S \wedge Y \subseteq S \wedge \#X = 2 \wedge Z = Y \backslash X$$

where S, X, Y, and Z are set variables and $\#X$ denotes the cardinality of the set X, the constraint solver in [5] is able to infer that the constraint is satisfiable provided $\#Z \leq 2$ holds.

Most of these consistency algorithms are incomplete, so they have to be combined with a backtracking search procedure to produce a complete constraint solver. For example, in the example above, such a procedure allows to enumerate all possible solutions for Z: $Z = \{1\}$, $Z = \{1, 2\}$, $Z = \{1, 3\}$, $Z = \{1, 4\}$.

While these constraint languages turn out to allow more efficient handling of set constraints with respect to the proposals cited in the previous section (e.g., CLP($\mathcal{SET}$)), the latter allows more general form of sets to be dealt with: elements can be of any type, possibly other sets, and possibly unknown (e.g., $\{X, \{a, 1\}\}$). For example the set-based formulation of the map coloring problem shown above can be written—and solved—using CLP($\mathcal{SET}$) but not using the constraint language in [14] and [5].

A current line of research (see [2]) is trying to combine the general set representation and management of proposals like CLP($\mathcal{SET}$), with the efficient constraint solving of "Finite Domain" solvers, in order to have the expressive power of the former while retaining the execution efficiency of the latter.

REFERENCES

[1] D. Aliffi, A. Dovier, and G. Rossi. From Set to Hyperset Unification. *J. of Functional and Logic Programming*, 1999(10):1–48, 1999.

[2] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. Integrating Finite Domain Constraints and CLP with Sets. In D. Miller, ed., *Proc. of 5th ACM-SIGPLAN Int'l Conf. on Principles and Practice of Declarative Programming*, 219–229, ACM Press, 2003.

[3] P. Arenas-Sánchez and A. Dovier. A Minimality Study for Set Unification. *J. of Functional and Logic Programming*, 1997(7):1–49, 1997.

[4] N. Arni, S. Greco, and D. Saccà. Matching of Bounded Set Terms in the Logic Language LDL++. *J. of Logic Programming*, 27(1):73–87, 1996.

[5] F. Azevedo. Cardinal: A Finite Sets Constraint Solver. *Constraints*, 12(37):93–129, 2007.

[6] W. Büttner. Unification in the Data Structure Sets. In J. K. Siekmann, ed., *Proc. of the 8th Int'l Conf. on Automated Deduction*, v. 230, 470–488, Springer-Verlag, 1986.

[7] D. Cantone, E. G. Omodeo, and A. Policriti. *Set Theory for Computing. From Decision Procedures to Declarative Programming with Sets*. Monographs in Computer Science, Springer-Verlag, 2001.

[8] E. Dantsin and A. Voronkov. A Nondeterministic Polynomial-Time Unification Algorithm for Bags, Sets, and Trees. In W. Thomas, ed., *FoSSaCS'99*, *LNCS* 1578, 180–196, Springer-Verlag, 1999.

[9] A. Dovier, E. G. Omodeo, E. Pontelli, and G. Rossi. {log}: A Language for Programming in Logic with Finite Sets. *J. of Logic Programming*, 28(1):1–44, 1996.

[10] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and Constraint Logic Programming. *ACM TOPLAS*, 22(5):861–931, 2000.

[11] A. Dovier, C. Piazza, and G. Rossi. A uniform approach to constraint-solving for lists, multisets, compact lists, and sets. *ACM Transactions on Computational Logic*, 9(3), 2008.

[12] A. Dovier, A. Policriti, and G. Rossi. A uniform axiomatic view of lists, multisets, and sets, and the relevant unification algorithms. *Fundamenta Informaticae*, 36(2/3):201–234, 1998.

[13] A. Dovier, E. Pontelli, and G. Rossi. Set unification. *Theory and Practice of Logic Programming*, 6:645–701, 2006.

[14] C. Gervet. Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints*, 1(3):191–244, 1997.

[15] C. Gervet. Constraints over Structured Domains. In F.Rossi, P. van Beek, and T. Walsh, ed's, *Handbook of Constraint Programming*. Elsevier, 2006.

[16] S. Greco. Optimal Unification of Bound Simple Set Terms. In *Proc. of Conf. on Information and Knowledge Management*, 326–336, ACM Press, 1996.

[17] P. Hawkins, V. Lagoon, and P. J. Stuckey. Solving Set Constraint Satisfaction Problems using ROBDDs. *J. of AI Research*, 24: 109–156, 2005.

[18] D. Kapur and P. Narendran. Complexity of Unification Problems with Associative-Commutative Operators. *J. of Automated Reasoning*, 9:261–288, 1992.

[19] V. Kuncak. Polynomial Constraints for Sets with Cardinality Bounds. *FoSSaCS'99*, *LNCS* 4423, Springer-Verlag, 2007.

[20] A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM TOPLAS*, 4:258–282, 1982.

[21] A. Martelli and G. Rossi. Efficient Unification with Infinite Terms in Logic Programming. In *Proc. of FGCS'84: Int'l Conf. on Fifth Generation Computer Systems*, 1984.

[22] G. Rossi, E. Panegai, and E. Poleo. JSetL: A Java Library for Supporting Declarative Programming in Java. *Software-Practice & Experience*, 37:115–149, 2007.

[23] F. Stolzenburg. An Algorithm for General Set Unification and Its Complexity. *J. of Automated Reasoning*, 22(1):45–63, 1999.

6 Contacts

Gianfranco Rossi
Dipartimento di Matematica
Università di Parma
Viale G. P. Usberti 53/A
43100 Parma (Italy)
Phone: +39 (0521) 90.6909
E-mail: gianfranco.rossi@unipr.it

7 Biography

Gianfranco Rossi received a degree in Computer Science from the University of Pisa in 1979. Since November 2001 he is a Full Professor of Computer Science at the University of Parma. His research activity has been mainly devoted to Programming Languages, with special attention to Logic Programming languages. Since December 2006 he is President of the Association of Logic Programming (GULP).