Sets and Constraint Logic Programming

Agostino Dovier
Università di Verona
Dip. Scientifico-Tecnologico
Strada Le Grazie
37134 Verona (Italy)
dovier@sci.univr.it

Carla Piazza
Università di Udine
Dip. di Matematica e Informatica
Via Le Scienze 206
33100 Udine (Italy)
piazza@dimi.uniud.it

Enrico Pontelli
New Mexico State University
Dept. Computer Science
Box 30001, Dept. CS
Las Cruces, NM 88003 (USA)
epontell@cs.nmsu.edu

Gianfranco Rossi
Università di Parma
Dip. di Matematica
Via M. D'Azeglio, 85/A
43100 Parma (Italy)
gianfr@prmat.math.unipr.it

August 30, 2000

Abstract

In this paper we present a Constraint Logic Programming (CLP) language which provides finite sets—along with basic set-theoretic operations—as first-class objects of the language. The language—called CLP($\mathcal{SET}$)—is an instance of the general CLP framework, and as such it inherits all the general features and theoretical results of this scheme.

The operational semantics of the language relies on the ability to verify satisfiability of any conjunction of positive and negative literals based on the predicate symbols $=, \in, \cup$, and $||$ (i.e., disjointness of two sets) in a (hybrid) universe of *finite sets*. We also review and compare the main techniques considered to represent finite sets in logic languages, and we give, trough programming examples, the taste of the expressive power offered by programming in CLP($\mathcal{SET}$).

Keywords: Declarative Programming, Constraints, Computable Set Theory.

1 Introduction

The notion of *set* is a common component in the design and development of computer programs. Nevertheless, conventional programming languages (e.g., Pascal) usually provide no, or very limited, support for this powerful data abstraction.[1] In recent years, however, many proposals in the field of *declarative programming languages* devoted more attention to the representation and management of sets.

This is the case of various *specification languages*, such as Z [60] and B [2]. In this context, sets have a primary role in providing the suitable high-level data abstractions required to make the language a vehicle for rapid experimentation with algorithms and program design.

Attention to sets has also emerged in the area of *database languages*, and more specifically in the context of *deductive databases* (e.g., LDL [9], COL [1], RelationLog [43]), where sets have been advocated as the most appropriate abstraction to deal with complex and incomplete objects.

More recently, various general-purpose functional and logic programming languages have introduced support for different flavors of sets. In particular, SEL [39] and its successor SuRE [38], CLPS [42],

[1] One notable exception is the procedural programming language SETL [56] which, on the contrary, adopts sets as its primary data structure.

Conjunto [31], and the recently proposed language CLAIRE [14], all provide sets as first-class entities, embedding them in either a functional-logic or constraint logic or object-oriented programming language. Moreover, other logic-based programming languages, such as Gödel [33], its successor Escher [45], and the concurrent language Oz/Mozart [59, 55], support sets through standard libraries of the language. Similarly, the ECL^{i}PS^e system [34] now provides Conjunto as one of its standard constraint libraries.

The experiences reported in all these proposals indicate the adequacy of declarative languages as hosts for set-theoretical constructions. As a matter of fact, declarativeness well combines with the high level of abstraction guaranteed by set constructs. Moreover, the non-determinism implicit in the operational semantics of logic programming languages is a fundamental feature for supporting the execution of many set-related operations.

All the languages mentioned above, however, impose restrictions either on the class of admissible set expressions, or on the computability properties of the expressions themselves. Very often sets are required to be *completely specified* in advance, i.e., no variable elements are allowed in the sets, and the sets themselves must have a predefined size. Alternatively, in many of these proposals, operations can be (safely) applied only to completely specified sets. Set elements are frequently limited to the atomic objects—i.e., nested sets are not allowed—and they can come only from a predefined finite domain of values—e.g., intervals of integers. Languages targeted to specific application domains, such as deductive database languages, are mostly concerned with aggregate operations (e.g., collecting elements with a given property and verifying membership), and they place limited attention on other basic set-theoretical facilities. On the other hand, those languages which provide very flexible and general set manipulation facilities—such as Z and B—usually do not consider computability properties as a primary requirement, being intentionally designed only as formal specification languages.

In this paper we present a logic-based language—called CLP($\mathcal{SET}$)—which combines a very flexible and general use of sets with a relatively efficient and safe execution support for all the set manipulation facilities provided.

To achieve these goals we select the *Constraint Logic Programming* (CLP) scheme as the general computational framework. In this context, sets are seen as primitive objects of the language and basic operations on sets are dealt with as *constraints*.

The class of admissible sets in CLP($\mathcal{SET}$) is considerably more general than those of the other proposals. Sets are allowed to be *nested* and *partially specified*. Sets are allowed to contain variables and other non-ground elements; furthermore, only some of the components of each sets are required to be explicitly enumerated, while the remaining elements can be left unspecified. The ability to treat set operations as constraints allows partially specified sets to be managed in a very flexible way, while preserving the clean declarative structure of (pure) logic and constraint programming languages.

The presence of sets as first-class citizens of the language—rather than providing them as an extension or an addition to the language but not part of the language itself (e.g., in the form of a library [31, 59])—allows us to endow sets with a precise formal semantics, properly integrated within the overall declarative semantics of the host language. In addition, being CLP($\mathcal{SET}$) an instance of the general constraint logic programming framework [35], the semantics of our language can be directly derived from that of this general scheme. Precisely, the semantics of CLP($\mathcal{SET}$)—both the operational, the logical, and the algebraic ones—are those of the general CLP scheme [35], suitable instantiated on the specific structure of interest, called $\mathcal{SET}$. This structure is designed to model *hereditarily finite sets*. All the predefined predicates dealing with sets are viewed as *primitive constraints* of the language. The *constraint solver* is developed accordingly, to allow the execution mechanisms to test constraint satisfiability in the $\mathcal{SET}$ structure.

Similarly to most related proposals, computational efficiency is not a primary requirement of our work. Our concern is mostly devoted to the possibility of describing as many as possible (complex) problems in terms of *executable* set data abstractions, and we would like to do this in the most intuitive and declarative way. On the other hand, and differently from the case of formal specification languages, the effectiveness and computability of the methods proposed is of primary importance. In CLP($\mathcal{SET}$) we strive to guarantee the ability of effectively (though not always efficiently) proving satisfiability of the problem at hand, as well as explicitly computing the set of all possible solutions. The focus of our work,

therefore, is on the *expressive power* of the language, combined with a 'clean' formal definition of it and effective methods for computing solutions.

It is important to observe that the inherent computational complexity of some of the problems at hand can be considerably high. For example, by allowing the programmer to use partially specified sets in his/her programs we introduce the potential of high complexity—e.g., the set unification problem between partially specified sets is known to be NP-complete [40, 22]. Nevertheless, at the implementation level it is possible to accommodate for the different cases, distinguishing between partially and completely specified sets and allowing operations on the latter to be executed in the most efficient way.

Regarding the use of our language to study the existence of computable solutions to set formulae, our work is closely related to the work on *Computable Set Theory (CST)* [12, 20]. CST was mainly developed to answer the need to enhance the inferential engines of theorem provers and for the implementation of the imperative language SETL [56]. The general problem was that of identifying computable classes of formulae of suitable sub-theories of the general Zermelo-Fraenkel set-theory.

A preliminary version of CLP($\mathcal{SET}$) was presented in [29]. The language presented in this paper represents the CLP counterpart of the extended logic programming language $\{log\}$ first presented in [21] and then more extensively described in [22]. An enhancement of the constraint handling capabilities of the original version of CLP($\mathcal{SET}$) was subsequently described in [24]. However, no complete description of the full version of the language has been provided so far. A CLP($\mathcal{SET}$) interpreter—implemented in SICStus Prolog—is available at http://www.math.unipr.it/~gianfr/setlog/interpreter.

The paper is organized as follows. In Section 2 we discuss the *pros* and *cons* of building set abstractions on top of an existing language, such as PROLOG, instead of adding them as first-class citizens of the language. In Section 3 we provide an overview of the different flavors of set abstractions that have been considered in the literature. Section 4 provides a brief overview of Constraint Logic Programming—in the context of a multi-sorted language—and introduces the fundamental notions and notations used throughout this paper. In Section 5 we introduce the representation of finite sets adopted in our language, which is based on the use of the binary element insertion symbol $\{\cdot \mid \cdot\}$ as the set constructor. Section 6 tackles the issue of selecting a suitable collection of set operations (e.g., $=$, $\in$, $\subseteq$, $\cup$) to be provided as primitive constraints in the language—instead of being explicitly *programmed* using the language itself. These choices may deeply affect the expressive power of the language. In Section 7 we compare our representation of sets with another approach widely used in the literature, that relies on the use of the binary union symbol $\cup$ as the set constructor. A precise characterization of the CLP($\mathcal{SET}$) language—namely, its signature, the structure used to assign a meaning to its symbols, the constraint solver procedure $SAT_{\mathcal{SET}}$ and the solved form of a constraint—is given in Section 8. The next section, Section 9, describes in detail the various rewriting procedures used by the $SAT_{\mathcal{SET}}$ constraint solver to rewrite a CLP($\mathcal{SET}$)-constraint to its equivalent solved form. The solved form is designed to guarantee a trivial satisfiability test. Section 10 provides a collection of sample CLP($\mathcal{SET}$) programs, along with some remarks about the notion of programming with sets in general. Section 11 provides the various formal results concerning the CLP($\mathcal{SET}$) language, including an axiomatic characterization of the set theory which captures the semantics of the constraints of CLP($\mathcal{SET}$). In this section we prove the correspondence of this theory with the underlying set structure $\mathcal{SET}$. These results allow us to prove that the constraint satisfiability procedure is correct and complete with respect to the given set theory. Complete proofs of all the results presented in this section are given in the Appendix. Finally, Section 12 compares our proposal with similar proposals in the field of Constraint Logic Programming languages with sets, while Section 13 presents concluding remarks and directions for future research.

Throughout the paper we assume the reader to be familiar with the general principles and notation of logic programming languages.

2 Implementing Sets in PROLOG

The goal of this work is to provide set abstractions as first-class citizens in the context of a (constraint) logic programming language. In order to justify this line of work, it is important to analyze the advantages

of this approach when compared to the simpler scheme which constructs set abstractions on the top of an existing language. Indeed, it is well-known that sets can be "easily" implemented in PROLOG [49].

The traditional approach for dealing with sets in PROLOG relies on the representation of sets as *lists*. Finite sets of terms are easily represented by enumerating all their constituting elements in a list. Since any term can be an element of a list, *nested sets* (i.e., sets containing other sets) are easily accommodated for in this representation. For example, the set $\{a, f(a), \{b\}\}$ can be represented as the list $[a, f(a), [b]]$. Set operations can be implemented by user-defined predicates in such a way to enforce the characteristic properties of sets—e.g., the fact that the order of elements in a set is immaterial—over the corresponding list representations. For example, the following clauses represent a very simple PROLOG implementation of the membership, subset and equality operations for non-nested sets, respectively [49, 62]:

```
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).
subset([], _).
subset([X | T], B) :- member(X, B), subset(T, B).
eqset(A, B) :- subset(A, B), subset(B, A).
```

When sets are formed only by ground elements this approach is satisfactory, at least from an "operational" point of view: the predicates provide the correct answers. On the other hand, when some elements of a set, or part of a set itself, are left unspecified—i.e., they are represented by variables—then this list-based approach presents major flaws. For example, the goal

$$:- \texttt{eqset}([\texttt{a} \mid \texttt{X}], [\texttt{b} \mid \texttt{Y}]) \tag{1}$$

which is intended to verify whether $\{a\} \cup X = \{b\} \cup Y$, will generate the infinite collection of answers

$$\begin{array}{l} \texttt{X} = [\texttt{b}], \texttt{Y} = [\texttt{a}]; \\ \texttt{X} = [\texttt{b}], \texttt{Y} = [\texttt{a}, \texttt{a}]; \\ \texttt{X} = [\texttt{b}], \texttt{Y} = [\texttt{a}, \texttt{a}, \texttt{a}]; \\ \ldots \end{array}$$

instead of the single more general solution which binds `X` to the set $\{b\} \cup S$ ($X = [b \mid S]$) and `Y` to the set $\{a\} \cup S$ ($Y = [a \mid S]$), where S is a new variable. Completeness is also lost in this approach. For example, with the usual PROLOG left-right approach, the computation of the goal $:- \texttt{eqset}([\texttt{a} \mid \texttt{X}], [\texttt{b} \mid \texttt{Y}]), \texttt{X} = [\texttt{b}, \texttt{c}]$ will never produce the correct solution $\texttt{X} = [\texttt{b}, \texttt{c}], \texttt{Y} = [\texttt{a}, \texttt{c}]$.

Similar problems arise also with the following unification problems:

$$\begin{array}{l} :- \texttt{eqset}(\texttt{X}, [\texttt{a} \mid \texttt{X}]) \\ :- \texttt{eqset}([\texttt{A}, \texttt{b} \mid \texttt{X}], [\texttt{c} \mid \texttt{X}]) \end{array}$$

as well as with other set operations implemented using lists.

Making the implementation of set predicates more sophisticated may help in solving correctly a larger number of cases. For example, one can easily modify the PROLOG implementation described above to explicitly identify the cases in which arguments are variables, and dealing with them as special cases. In similar ways one can try to account for nested sets, by using `eqset` instead of standard unification in the definition of the set operations. Nevertheless, there are cases in which there is no simple finite equational representation of the (possibly infinite) solutions of a goal involving set-theoretic operations. Consider the goal

$$:- \texttt{subset}([\texttt{a} \mid \texttt{X}], [\texttt{a} \mid \texttt{Y}]) \tag{2}$$

used to represent the query $\{a\} \cup X \subseteq \{a\} \cup Y$. It is easy to observe that each X which is a subset of Y will represent a solution to the above goal. However, this simple fact is not expressible at all by adopting a direct PROLOG implementation.

These problems can be solved by moving from conventional PROLOG to the more general context of Constraint Logic Programming. In this context, set operations are viewed as *constraints* over a suitable set-theoretic domain, and computed answers are expressed in terms of irreducible constraints. For example, an atom such as `subset(X,Y)` which, intuitively, indicates that X must be a subset of Y, with X, Y variables, can be conveniently considered as an irreducible constraint, and kept unchanged as part of the computed answer. Thus, one possible answer for goal (2) could be the constraint `subset(X,Y)`.

Similar considerations hold also when the negative counterparts of the basic set-theoretic operations, such as *"not equal"* and *"not member"*, are taken into account. The use of the list-based implementation of sets, in conjunction with the *Negation as Failure* rule for negation (provided by most implementations of PROLOG), leads to a similar poor behavior as in the previously discussed cases. Also in this context, viewing these set operations as constraints provides a more convenient solution. For example, the answer to the goal :- {`a`} `neq` {`X`}, where `neq` is interpreted as the inequality operation, would be the (irreducible) constraint `X neq a`. The same goal solved in the PROLOG representation of sets incorrectly leads instead to a failure.

These observations lead to the following conclusions. First of all, if one has to deal only with ground sets, then it is likely he/she has no need for anything more sophisticated than the usual PROLOG implementation of sets. This is no longer true if one has to deal with partially specified sets.

The ability to deal with partially specified sets strongly enhances the expressive power of the language. As a matter of fact, there are many problems—especially combinatorial problems and problems in the NP class—which can be coded as set-based first-order logic formulae. Furthermore, the ability to compute with partially specified sets allows one to enhance conciseness and declarativeness of the resulting programs.

Most PROLOG implementations provide built-in features, such as the setof predicate, for building a set intensionally rather than extensionally; this means that a set is defined as the collection of all elements satisfying a giving a property, instead of explicitly enumerating all elements belonging to the set. This is a very common way of defining a set in the practice of mathematics, and the availability of such a feature considerably enhances the expressive power of language. Unfortunately, the PROLOG solution suffers from a number of hindrances, as summarized for instance in [50]: elements are collected in a list (not a set), there are problems with variables in lists of solutions and problems with global vs. local variables. The usual view of setof is that of an added higher-order feature which is hardly accommodated for in the formal semantic structure of the host language.

Most of these problems can be overcome using CLP($\mathcal{SET}$) as host language, since its set manipulation facilities allow one to define the set aggregation mechanisms in the language itself, without sacrificing the desired logical meaning of the setof predicate. The only required addition is support for rules containing negative literals in their body [10]. This issue and possible solutions have been discussed in detail elsewhere [10, 27].

3 Which kind of sets?

The first step in the definition of a language over sets is the precise characterization of the flavors of *set* supported by the language.

Formal set theory traditionally focuses on sets as the only entities in the domain of discourse. In our context we extend this view by allowing arbitrary *atomic*—i.e., non-set—entities as first-class citizens of the language. Atoms will be allowed to appear as members of sets but no element will be allowed to belong to an atom. Thus, CLP($\mathcal{SET}$) allows the representation of *hybrid* sets (as opposed to *pure* sets).

The second criteria used to characterize the class of admissible sets focuses on the cardinality of the sets. In the context of this work we will restrict our attention only to *finite sets*. Sets can contain as elements either atoms—*flat sets*—or other sets—*nested sets*. Many practical applications have demonstrated the need for nested sets. Thus, in our framework we intend to allow sets containing a finite number of elements, each being either an atom or another finite set. This class of sets is commonly indicated as *hereditarily finite hybrid sets*.

Example 3.1

a	*is an atom*
$\{a, b, c\}$	*is a flat set with three atomic elements*
$\{\emptyset, a, \{b, c\}\}$	*is a set with three elements:*
	the empty set, an atom, and a (nested) set with two elements

Remark 3.2 *As far as pure sets are concerned, results coming from Computable Set Theory [12] ensure that a constraint built with the signature of the language we are presenting is satisfiable if and only if it is satisfiable over the universe of hereditarily finite (and well-founded) sets. Thus, in this context, working on finite sets is not a restriction.*

An orthogonal criteria used to characterize the class of admissible sets derives from the kind of notation used to describe sets. It is common to distinguish between sets designated via explicit enumeration (*extensional sets*)—e.g., $\{a, b, c\}$—and sets described through the use of properties and/or characteristic functions (*intensional sets*)—e.g., $\{X : \varphi[X]\}$. In this paper we restrict our attention to *extensional* sets. The problems implied by the introduction of *intensional* sets, have been addressed in other related works [9, 10]. It is well accepted that this problem is strongly connected with that of introducing *negation* in a logic programming language [5]. A proposal for using constructive negation to embed intensional sets in a preliminary version of the CLP language described in this paper is presented in [27].

The relaxation of the non-cyclicity of membership leads to the notion of *hypersets*. Hypersets can be described as rooted labeled graphs and concretely rendered as systems of equations in canonical form [3]. Dealing with hypersets requires replacing the notion of equality between ground terms with the notion of ground graphs having the same *canonical representative*. The axioms of set theory also need to be modified to include a form of the anti-foundation axiom, typical of hyperset theory. A formal characterization of hypersets and the definition of a suitable unification algorithm dealing with them have been given in [4]. However, a precise embedding of hypersets in the language presented in this paper, and more convincing motivations for such an extension, still need further investigation, and are outside the scope of this work.

Other classes of aggregates have also been considered in the literature. In particular, various frameworks have introduced the use of *bags* or *multisets* (e.g., [18]) where repeated elements are allowed to appear in the collection. Work on introducing multisets in the context of CLP is still in progress at present. Nevertheless, the representation techniques and the set theory described in the next sections are adequate to accommodate for multisets through minor modifications. An analysis of the problems concerned with the introduction of multisets—as well as sets and compact-lists—is reported in [25].

4 A brief review of Constraint Logic Programming

In this section we provide a brief overview of *Constraint Logic Programming (CLP)* (as presented for instance in [35]) to introduce the fundamental notions and notations used throughout this paper. We consider here the general case of a multi-sorted first-order language $\mathcal{L}$. $\mathcal{L}$ is defined by

- a finite set $\mathsf{Sort} = \{\mathsf{Sort}_1, \ldots, \mathsf{Sort}_\ell\}$ of *sorts*
- a *signature* Σ composed by a set $\mathcal{F}$ of constant and function symbols, and a set Π of predicate symbols
- a denumerable set $\mathcal{V}$ of logical variables.

In the rest of this paper we will adopt the following convention: capital letters X, Y, Z, etc. will be used to represent variables, f, g, etc. to represent function symbols, and p, q, etc. to represent predicate symbols. The symbols τ_i will be used to denote a generic subset of Sort. We will also use the symbol $\equiv$ to denote *syntactic equality* between terms.

The arity function $ar : \Sigma \longrightarrow \mathbb{N}$ associates an arity with each symbol in Σ. Moreover, each element f in $\mathcal{F}$ is associated with a tuple $\langle \tau_1, \ldots, \tau_{ar(f)}, \tau_{ar(f)+1} \rangle$ (the sort of f) which describes the sort of the

arguments and of the result of f; each element p of Π is associated with a tuple $\langle \tau_1, \ldots, \tau_{ar(p)} \rangle$; finally, each variable V is associated with a subset τ_i of Sort.

$T(\mathcal{F}, \mathcal{V})$ ($T(\mathcal{F})$) denotes the set of first-order terms (resp., ground terms) built from $\mathcal{F}$ and $\mathcal{V}$ (resp., $\mathcal{F}$) which respect the sorts of the symbols. Given a term $f(t_1, \ldots, t_n)$ in $T(\mathcal{F}, \mathcal{V})$, if f has sort $\langle \tau_1, \ldots, \tau_n, \tau_{n+1} \rangle$ and t_i has sort τ_i, then we will say that the term is of sort τ_{n+1}. Given a sequence of terms $t_1, \ldots, t_n$, $vars(t_1, \ldots, t_n)$ is used to denote the set of all variables which occur in at least one of the terms t_i.

An *atomic formula* (or, simply, an *atom*) is an object of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol in Π with arity n, and the t_i's are terms in $T(\mathcal{F}, \mathcal{V})$ which respect the sort associated to p.

The set of predicate symbols Π is assumed to be composed of two disjoint sets, Π_c and Π_u: Π_c is the set of *constraint (predicate) symbols*, while Π_u is the set of user-defined predicate symbols. Each atomic formula $p(t_1, \ldots, t_n)$ where p is a constraint symbol (i.e., a symbol from Π_c) is called a *primitive constraint*.

A *constraint* is a first-order formula which belongs to a subset $\mathcal{C}$ of all the first-order formulae that can be built using the primitive constraints. The subset $\mathcal{C}$ is chosen according to some—usually syntactical—criteria and it is typically assumed to be closed under conjunction. Moreover, it is often assumed that the equality symbol '$=$' belongs to Π_c and that $\mathcal{C}$ contains all the primitive equality constraints $s = t$ with s, t terms from $T(\mathcal{F}, \mathcal{V})$.

A *CLP program* P is a finite set of rules of the form

$$A :- c, B.$$

where A is a $\langle \Pi_u, \mathcal{F}, \mathcal{V} \rangle$-atom, c is a constraint and B is a (possibly empty) conjunction of $\langle \Pi_u, \mathcal{F}, \mathcal{V} \rangle$-atoms. A is called the *head* of the rule and c, B is called the *body*. A *goal* is a rule with empty head. Note that only well-formed—i.e., respecting the assigned sorts—terms and atoms are allowed to occur in programs and goals. As part of the tradition in logic programming, the comma (',') will be used instead of $\wedge$ to denote conjunction in concrete programs. Similarly, we assume that all (free) variables in a clause are universally quantified in front of the clause itself. As a consequence, variables occurring only in the body of a clause can be seen as existentially quantified.

Example 4.1 *Let Σ contain the constant symbols a, b, the function symbol f, with $ar(f) = 1$, the user-defined predicate symbol p, with $ar(p) = 1$, and the binary constraint predicate symbol $=$. Let us also assume that all symbols have the same sort. A sample constraint is:*

$$\forall \mathtt{Z}(\mathtt{X} \neq \mathtt{f}(\mathtt{Z})), \mathtt{Y} = \mathtt{a}, \mathtt{Z} \neq \mathtt{b}.$$

A sample clause and goal are:

$$\mathtt{p}(\mathtt{X}) :- \mathtt{X} = \mathtt{f}(\mathtt{Y}), \mathtt{p}(\mathtt{Y}).$$
$$:- \mathtt{Z} \neq \mathtt{b}, \mathtt{p}(\mathtt{Z}).$$

Note that the primitive constraints in the body of a rule can also occur negated. For the sake of readability we will use $\not\pi(t_1, \ldots, t_n)$ to denote $\neg\pi(t_1, \ldots, t_n)$, for any constraint predicate symbol π. Thus, for instance, $s \neq t$ will be used to represent $\neg(s = t)$.

Constraints and programs in the CLP language based on Σ are interpreted with respect to a selected Σ-structure. A Σ-*structure* (or, simply, a *structure*) $\mathcal{A}$ is composed by a tuple $\mathcal{D} = \langle \mathcal{D}_1, \ldots, \mathcal{D}_\ell \rangle$ of non-empty sets $\mathcal{D}_i$—the *domain* of the sort Sort_i—and by an interpretation function $(\cdot)^{\mathcal{A}}$. The function $(\cdot)^{\mathcal{A}}$ assigns functions and relations on $\mathcal{D}$ to the symbols of Σ, respecting the arities and sorts of the symbols. A *valuation* σ of a formula φ is an assignment of values from $\mathcal{D}$ to the free variables of φ, respecting the sorts of the variables. σ can be extended to terms in a straightforward manner. In the case of formulae, as for instance in [54, 16], we write $\varphi[\sigma]$, instead of $\sigma(\varphi)$, to denote the application of a valuation to a formula. σ is a *successful valuation* if $\varphi[\sigma]$ is true in $\mathcal{D}$.

$CLP(\mathcal{A})$ denotes a particular instance of a CLP language based on the structure $\mathcal{A}$. This instance is further characterized by the considered signature Σ, the class of constraints $\mathcal{C}$, a constraint theory $\mathcal{T}$ which describes the logical semantics of the constraints in $\mathcal{C}$, and a *constraint solver*, that is a procedure which is used to check the satisfiability in $\mathcal{A}$ of constraints from $\mathcal{C}$. Strictly speaking, the class of CLP languages is

parametric with respect to the collection of all these components [36]. For simplicity, however, we prefer to use the notation in which an instance of the general CLP scheme is simply identified by its underlying structure $\mathcal{A}$.

Given a structure $\mathcal{A}$, it is also common to identify a class *Adm* (a possibly strict subset of $\mathcal{C}$) of constraints that can be used in a CLP program, called the class of *admissible constraints*. Roughly speaking, *Adm* is the class of constraints for which the constraint solver is effectively capable of deciding satisfiability in $\mathcal{A}$ [64]. In other words, the constraint solver is guaranteed to be *complete* with respect to the formulae *Adm* [36].

Example 4.2 *Let Σ contain a collection $\mathcal{F}$ of constant and function symbols and the binary constraint predicate symbol $=$. As in Example* 4.1*, assume that all symbols have the same sort which is interpreted to $T(\mathcal{F})$ (i.e., the Herbrand Universe). Let the interpretation domain A be the set of the finite trees built in the usual way from symbols in $\mathcal{F}$. The interpretation function allows Σ-terms to be mapped to trees in A in the usual standard way. The interpretation of $=$ is the identity relation over A. $\langle A, (\cdot)^{\mathcal{H}} \rangle$ is the Herbrand structure $\mathcal{H}$ over Σ, as used in* PROLOG*. Clark's Equality Theory is a complete axiomatization for this structure [46].*

The admissible constraints can be simply all the conjunctions of equations of the form $t_1 = t_2$, where t_1 and t_2 are Σ-terms. This is exactly the case of PROLOG*. The standard unification algorithm is a constraint solver for this domain. Alternatively, the admissible constraints can include also (universally quantified) disequations. According, for instance, to [15] one can choose the admissible constraints to be $s = t$ and $\forall \bar{Z}(s \neq t)$, where s and t are terms and $\bar{Z}$ a (possibly empty) set of variables (subset of $vars(s,t)$). This enlargement of the set of admissible constraints requires more sophisticated constraint solvers, such as those presented in [15, 64].*

The operational semantics of CLP is typically given in terms of derivations from goals. We give here a simplified version of the detailed definition [35]. *Derivations* are sequences of state transformations. Each state is a pair $\langle c \,|\, G \rangle$ where c is an admissible constraint and G is a conjunction of Π_u-atoms. A transformation is obtained by selecting an atom $p(t_1, \ldots, t_n)$ from G and a clause (renamed with new variables) $p(s_1, \ldots, s_n) :- c', B$ from the program. If

$$c'' \equiv (c \wedge c' \wedge s_1 = t_1 \wedge \cdots \wedge s_n = t_n)$$

is *consistent*—i.e., if it is satisfiable in the underlying structure $\mathcal{A}$—and d is a constraint such that $\mathcal{A} \models \vec{\forall}(d \rightarrow c'')$ then a new state, represented by $\langle d \,|\, G', B \rangle$, is obtained, where G' is G without $p(t_1, \ldots, t_n)$. A simple derivation can be realized by simply taking $d \equiv c''$. However, when c'' contains a disjunction of constraints, d can be non-deterministically chosen as any element of the corresponding disjunctive normal form of c''.

A derivation from the state $\langle c \,|\, G \rangle$ terminates either when all possible derivation steps produces an inconsistent constraint c''(failure), or when G is empty (success). In this second case, the constraint part c represents the computed answer. As explained in [35], c may contain redundant information, and a more precise solution can be obtained by an ad-hoc procedure *infer*, developed to simplify the output (e.g., by computing the projection of c on the variables of interest).

Example 4.3 *Consider the language of Example* 4.2 *with the admissible constraints fixed to be conjunctions of equations and disequations. The following is a (well-formed) program in this language:*

$$\begin{array}{ll} P: & \mathtt{p(X,Y)} :- \mathtt{X} = \mathtt{a}, \mathtt{Y} = \mathtt{a}. \\ & \mathtt{p(X,Y)} :- \mathtt{X} \neq \mathtt{a}, \mathtt{q(Y)}. \\ & \mathtt{q(X)} :- \mathtt{X} = \mathtt{f(Y)}. \end{array}$$

One possible derivation for the goal $:- p(X_1, X_2)$ from P is

$$\begin{array}{lll} \langle true \;\; | \, p(X_1, X_2) \rangle & \mathtt{p(X,Y)} :- \mathtt{X} \neq \mathtt{a}, \mathtt{q(Y)} & \\ \searrow & \downarrow & \\ & \langle X_1 = X, X_2 = Y, X \neq a \,|\, q(Y) \rangle & \mathtt{q(X')} :- \mathtt{X'} = \mathtt{f(Y')} \\ & \searrow & \downarrow \\ & \langle X_1 = X, X_2 = Y, X \neq a, Y = X', X' = f(Y') \,| \quad \rangle & \end{array}$$

The computed answer is the constraint in the last line; however, it can be simplified to the formula $X_1 \neq a, X_2 = f(Y')$.

5 Set Representation

The representation of finite sets adopted in the majority of the proposals dealing with sets in logic-based languages [9, 22, 33, 39, 63, 42] is based on the use of a binary function symbol, e.g, `scons`, as set constructor, interpreted as the *element insertion* operator. Roughly speaking, $\texttt{scons}(x, y)$ denotes the set obtained by adding x as an element to the set y, i.e., $\{x\} \cup y$. This is analogous to the representation usually adopted for lists in PROLOG, and like lists it is well suited for recursive programming. Alternative representations will be discussed and compared in Section 7.

In this paper we adopt this list-like solution, using the binary function symbol $\{\cdot \mid \cdot\}$ as the set constructor. Therefore, $\{x \mid y\}$ will represent the set $\{x\} \cup y$.

Moreover, we introduce two distinct sorts: Set and Ker. Intuitively, Set is the sort of all the terms which denote sets and Ker is the sort of all other terms. Therefore, the sort of the function symbol $\{\cdot \mid \cdot\}$ is

$$\langle \{\mathsf{Set}, \mathsf{Ker}\}, \{\mathsf{Set}\}, \{\mathsf{Set}\} \rangle$$

In addition, a constant symbol from $\mathcal{F}$, say $\emptyset$, is selected and used to denote the *empty set*. Its sort is $\langle \{\mathsf{Set}\} \rangle$. Thus, a term t is of sort Set if and only if

- $t \equiv X$ and X is of sort Set;
- $t \equiv \emptyset$;
- $t \equiv \{t_1 \mid t_2\}$ and t_2 is of sort Set.

Any term of sort Set is called a *set term*.

The other function and constant symbols of arity $n \geq 0$ have the following sort:

$$\langle \underbrace{\{\mathsf{Set}, \mathsf{Ker}\}, \ldots, \{\mathsf{Set}, \mathsf{Ker}\}}_{n}, \{\mathsf{Ker}\} \rangle$$

Terms whose main function symbol is of sort Ker, as well as variables of sort Ker, are called *non-set terms* (or, *kernels*).

In the previous works on CLP($\mathcal{SET}$) [29, 20, 24] we did not distinguish between different sorts. As a consequence, the term y in $\{x \mid y\}$ can be a term denoting a set as well as a term denoting a non-set entity—e.g., $\{a \mid a\}$. This leads to the enlargement of the domain of discourse to include the so-called *colored sets*, i.e., sets which are built by adding elements to a non-set object. Sets built starting from a non-set object k are called colored sets, and k is the *color*, or kernel, of the set (e.g., $\{a \mid a\}$ is a colored set based on the color a). In spite of their theoretical interest, colored sets seem to have little practical utility when used in the context of a logic language with (hybrid) sets. In addition, handling colors in the constraint management procedures turns out to be cumbersome [29]. In contrast, the choice of using a multi-sorted language considered in this paper allows a more intuitive presentation of sets and the design of more compact constraint solving algorithms. The use of sorts implies, in particular, that terms such as $\{a \mid a\}$ are ill-formed and are not accepted in CLP($\mathcal{SET}$).

The function symbol $\{\cdot \mid \cdot\}$ is an interpreted symbol, used to construct sets. Precisely, $\{\cdot \mid \cdot\}$ fulfills the following equational axioms [20, 22]:

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

Axiom (Ab) states that duplicates in a set do not matter (*Absorption property*). Axiom $(C\ell)$ states that the order of elements in a set is irrelevant (*Permutativity property*). These two properties capture the intuitive idea that, for instance, the set terms $\{a \mid \{b \mid \emptyset\}\}$, $\{b \mid \{a \mid \emptyset\}\}$, and $\{a \mid \{b \mid \{a \mid \emptyset\}\}\}$ all denote

the same set $\{a, b\}$. Observe that duplicates do not occur in a set, but they may occur in the set term that denotes it. This corresponds also to the observation that the set term $\{x \mid y\}$, which denotes the set $\{x\} \cup y$, does not necessarily require $x \notin y$ to hold. As we will see in the CLP($\mathcal{SET}$) programming examples (Section 10), restrictions on y, if needed, have to be explicitly stated using non-membership constraints. This approach is different from others in the literature—e.g., [37] introduces a set constructor, called `dscons`, which implicitly requires $x \notin y$ to hold.

For the sake of simplicity, hereafter we will use the more natural notation $\{t_1, \ldots, t_n \mid t\}$ as a syntactic sugar to denote the term $\{t_1 \mid \cdots \{t_n \mid t\} \cdots\}$. Moreover, the notation $\{t_1, \ldots, t_n\}$ will be used in the particular case where $t = \emptyset$. Finally, note that when $n = 0$, the term $\{t_1, \ldots, t_n \mid t\}$ actually refers to the term t.

Example 5.1 *(Set terms) Let Σ contain the symbols $\emptyset$, $\{\cdot \mid \cdot\}$, a, b, c, and $f(\cdot)$ (i.e., the symbol f has arity 1) and let X be a variable of sort* Set.

$\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ *(i.e.,* $\{\mathtt{a} \mid \{\mathtt{b} \mid \{\mathtt{c} \mid \emptyset\}\}\}$ *)*	*is a set term*
$\{\mathtt{a} \mid \mathtt{X}\}$	*is a set term (a partially specified set)*
$\mathtt{f}(\{\mathtt{a}, \{\mathtt{a}, \mathtt{b}\}\})$	*is a non-set term.*
$\mathtt{f}(\{\mathtt{a} \mid \mathtt{b}\}\})$	*is an ill-formed term.*

6 Primitive Operations on Sets

Once a representation for sets has been selected, the next question in the design of a language with sets is which of the basic set operations (e.g., $=$, $\in$, $\subseteq$, $\cup$) should be *built-in* in the language—i.e., part of Π_c—and which, on the contrary, should be *programmed* using the language itself—i.e., part of Π_u. The choice of built-in operations should be performed according to various criteria, such as expressive power, completeness, effectiveness, and efficiency.

Let us start by assuming that $=$, $\in$, together with their negative counterparts, are the only primitive operations of the language. Therefore, we assume that the signature Σ of the language contains the binary constraint predicate symbols $=$ and $\in$, and that the admissible constraints are conjunctions of literals of the form $t = s$, $t \neq s$, $t \in s$, $t \notin s$, where s and t are terms. Moreover, we assume these symbols have the following sorts: $\langle\{\mathsf{Set}, \mathsf{Ker}\}, \{\mathsf{Set}, \mathsf{Ker}\}\rangle$ for $=$, and $\langle\{\mathsf{Set}, \mathsf{Ker}\}, \{\mathsf{Set}\}\rangle$ for $\in$. We will refer to this language as the *base language*. This is actually the same language presented in [29, 22], save for sorts.

The intuitive meaning of $t = s$ is the equality between s and t modulo the equational theory T, where T contains the two axioms (Ab) and $(C\ell)$ described in the previous section. Given an equational theory E and a conjunction of equations $C \equiv (s_1 = t_1 \wedge \cdots \wedge s_n = t_n)$ the *(decision) E-unification problem* is the problem of deciding whether $E \models \vec{\exists} C$. A substitution σ is an *E-unifier* of two terms s, t if $s^\sigma =_E t^\sigma$—i.e., s^σ and t^σ belong to the same E-congruence class. $\bigcup \Sigma_E(s, t)$ denotes the *set of all E-unifiers of s and t* [58]. Thus, a solver for $=$ can be implemented using a general *$(Ab)(C\ell)$-unification algorithm* which is able to compute a complete set of T-unifiers for any two terms s and t. In particular, the algorithm is capable of dealing with (possibly nested) set unification problems of the form:

$$\{s_1, \ldots, s_m \mid u\} = \{t_1, \ldots, t_n \mid v\}$$

where $m, n \geq 0$, s_i and t_j are generic (well-formed) terms, and u, v can be either variables or $\emptyset$. One such unification algorithm will be described in detail in the next section as part of the constraint solver procedure of the CLP($\mathcal{SET}$) language.

The atomic formula $t \in s$ captures the traditional *membership* relation. $t \in s$ is satisfiable if and only if s is a set term and t occurs as an element in s.

$t \neq s$ and $t \notin s$ represent the negative counterparts of the equality ($=$) and membership ($\in$) predicates.

Other basic operations on sets, such as union, subset, and intersection, can be easily defined in this base language, as shown in [22]. For example, the $\subseteq$ operation of sort $\langle\{\mathsf{Set}\}, \{\mathsf{Set}\}\rangle$, representing the relation

$$s \subseteq t \leftrightarrow \forall Z (Z \in s \rightarrow Z \in t),$$

can be implemented in the base language as:

$$\emptyset \subseteq \mathtt{S1}.$$
$$\{\mathtt{A} \mid \mathtt{S1}\} \subseteq \mathtt{S2} :\!- \mathtt{A} \in \mathtt{S2}, \mathtt{S1} \subseteq \mathtt{S2}.$$

where A, S1, S2, are variables. Note that the universal quantification used in the logical definition is rendered in the implementation by recursive rules, which allow us to express iteration over all the elements of a set.

Strictly speaking, also $=$ and $\in$ (and their negative counterparts) can be defined one in terms of the other:

$$\mathtt{s} = \mathtt{t} :\!- \mathtt{s} \in \{\mathtt{t} \mid \emptyset\}.$$

and

$$\mathtt{s} \in \mathtt{t} :\!- \mathtt{t} = \{\mathtt{s} \mid \mathtt{t}\}.$$

Therefore, only one of them is strictly required.

Minimizing the number of predicate symbols in Π_c has the advantage of reducing the number of different kinds of constraints to be dealt with and, hopefully, simplifying the language and its implementation. On the other hand, this choice may lead to problems in efficiency and effectiveness, similar to those encountered with the implementation of sets using PROLOG's lists discussed in Section 2. For example, using the above definition of $\subseteq$, the computed solutions for the goal

$$:\!- \mathtt{S1} \subseteq \mathtt{S2}$$

(S_1 and S_2 variables) are:

$$[\mathtt{S1}/\emptyset] \quad [\mathtt{S1}/\{\mathtt{E1}\}, \mathtt{S2}/\{\mathtt{E1} \mid \mathtt{Z}\}] \quad [\mathtt{S1}/\{\mathtt{E1}, \mathtt{E2}\}, \mathtt{S2}/\{\mathtt{E1}, \mathtt{E2} \mid \mathtt{Z}\}] \quad \cdots$$

that is, the computation blindly proceeds into the extensional generation of all solutions, opening an infinite number of choices.

The problems with the implementation of subset—and of other similar operations, such as union and intersection—using the base language originate from the recursive nature of the implementation itself, which in turn is a consequence of the need to use universal quantifications in its logical definition. As a matter of fact, from [23]:

> *Let $\mathcal{L}$ be the language $\{\emptyset, \{\cdot \mid \cdot\}, =, \in\}$ and let $\mathcal{T}$ be some reasonable theory of sets for this language (like the one we assume underlies CLP($\mathcal{SET}$) and that will be presented in detail in Section 11.1). For any model M of $\mathcal{T}$ there is no quantifier-free formula φ in $\mathcal{L}$, $vars(\varphi) = \{X, Y, Z_1, \ldots, Z_n\}$, such that $M \models \forall XY(X \subseteq Y \leftrightarrow \exists Z_1 \cdots Z_n \, \varphi)$.*

In other words, it is not possible to express $\subseteq$ in the language $\{\emptyset, \{\cdot \mid \cdot\}, =, \in\}$ without using universal quantification. This implies also that constraints based on $\subseteq$ are more expressive than those of the base language. Similar results can be given for union and intersection, since $X \subseteq Y$ is equivalent to both $X \cup Y = Y$ and $X \cap Y = X$. Also a symmetrical result holds, namely, it is not possible to express directly $\{\cdot \mid \cdot\}$ using $\cup$ (unless an additional constructor—e.g., the singleton set—is introduced in Σ).

Actually, a restricted form of universal quantifiers, called Restricted (or Bounded) Universal Quantifiers (RUQ), has been shown to be sufficiently expressive to define the most commonly used set operations [22, 12]. RUQs are formulae of the form $\forall X(X \in S \rightarrow \varphi[X])$, where φ is any first-order formula containing X. As shown in [22], this restricted form of universal quantification can be easily implemented in the base language with sets considered so far, using recursive rules. However, also in this case we may encounter the same problems mentioned above, namely the possibility of generating infinite sets of answers.

To avoid these problems, one can enrich the base language with additional primitive constraints, that implement new operations for set manipulation. Our choice is to add to the base language two new constraint predicate symbols, $\cup_3$ and $||$. Hence, the admissible constraints now are the conjunctions of (positive, negative) literals based on the symbols $=$, $\in$, $\cup_3$ and $||$. The sorts of the new symbols are:

$\langle\{\mathsf{Set}\},\{\mathsf{Set}\},\{\mathsf{Set}\}\rangle$ for $\cup_3$, and $\langle\{\mathsf{Set}\},\{\mathsf{Set}\}\rangle$ for $||$. The predicate $\cup_3$ captures the notion of *union of sets*: $\cup_3(r,s,t)$ is satisfiable if t is the set resulting from the union of the sets r and s—i.e., $t = r \cup s$. The predicate $||$ is used to verify disjointness of two sets: $s||t$ is satisfiable if s and t are sets and they have no elements in common. In the context of Computable Set Theory $\not\exists$ is denoted by $\ni\in$. The idea behind this notation is the following: given two relations R and S, $xRSy$ holds if and only if there is z such that xRz and zSy. Choosing R as $\ni$ and S as $\in$, this is exactly the definition of $\not\exists$.

The choice of these additional primitive constraints is motivated by the observation that, if properly managed, they allow us to express most of the other usual set operations as simple open formulae without having to resort to any universal quantification. In particular, consider the predicate $\subseteq (r,s)$, along with the other two predicates $\cap_3(r,s,t)$ and $\setminus_3(r,s,t)$, with the following intuitive meaning: $\cap_3(r,s,t)$ is satisfiable if t is the set resulting from the intersection of the sets r and s—i.e., $t = r \cap s$—while $\setminus_3(r,s,t)$ is satisfiable if t is the set obtained as the result of the difference between the sets r and s—i.e., $t = r \setminus s$. Both symbols $\cap_3$ and $\setminus_3$ are assumed to have sort $\langle\{\mathsf{Set}\},\{\mathsf{Set}\},\{\mathsf{Set}\}\rangle$,

Proposition 6.1 *Literals based on predicate symbols: $\subseteq$, $\cap_3$, and $\setminus_3$ can be replaced by equivalent* conjunctions of literals *based on $\cup_3$ and $||$.*

Proof. (sketch) The following equivalences hold:

$$\begin{array}{rll}
s \subseteq t & \text{if and only if} & \cup_3(s,t,t) \\
s \not\subseteq t & \text{if and only if} & \not\cup_3(s,t,t) \\
\cap_3(r,s,t) & \text{if and only if} & \exists R,S(\cup_3(R,t,r) \wedge \cup_3(S,t,s) \wedge R||S) \\
\not\cap_3(r,s,t) & \text{if and only if} & \exists T(\cap_3(r,s,T) \wedge T \neq t) \\
\setminus_3(r,s,t) & \text{if and only if} & \exists W\ (\cup_3(t,r,r) \wedge \cup_3(s,t,W) \wedge \cup_3(r,W,W) \wedge s||t) \\
\not\setminus_3(r,s,t) & \text{if and only if} & \exists T(\setminus_3(r,s,T) \wedge T \neq t)
\end{array}$$

□

Similar result holds also for $\triangle$ ($s \triangle t = s \setminus t \cup t \setminus s$).

Remark 6.2 *Negative $\cap_3$ and $\setminus_3$ literals could be replaced in several other ways. For instance, $\not\cap_3(r,s,t)$ is equivalent to:*

$$t \not\subseteq r \vee t \not\subseteq s \vee (t \subseteq r \wedge t \subseteq s \wedge \cup_3(t,r,R) \wedge \cup_3(t,s,S) \wedge (r \neq R \vee s \neq S))$$

We do not enter here in such a discussion; at the implementation level one can make the desired choices.

Remark 6.3 *An advantage of colored sets with respect to conventional sets is that having $\cup_3$ as a primitive constraint, and assuming that colored sets are properly accounted for—that is, the union of two sets is allowed only if they are based on the same color—then it is possible to replace all literals based on the predicate symbols $\in$ and $=$ with literals based only on $\cup_3$—$s \in t$ if and only if $\cup_3(t,t,\{s \mid t\})$ and $s = t$ if and only if $\cup_3(s,s,t)$. However, while this could be of theoretical interest, efficiency and simplicity consideration led us to consider the whole collection of primitive constraints—$=$, $\in$, $\cup_3$, $||$, and their negative counterparts—when developing the constraint solving algorithms [24].*

7 Alternative Representations of Sets

Alternative methods for representing sets have been considered in the literature. In particular, one of the most popular approaches relies on the use of the binary union symbol $\cup$ as the set constructor [7, 11, 44]. In this case, $\mathcal{F}$ is required to contain (at least) the binary function symbol $\cup$ and the constant symbol $\emptyset$. $\cup$ fulfills the equational axioms:

$$\begin{array}{lrcl}
(A) & (X \cup Y) \cup Z & = & X \cup (Y \cup Z) \\
(C) & X \cup Y & = & Y \cup X \\
(I) & X \cup X & = & X
\end{array}$$

while $\emptyset$ is interpreted as the *identity* of the operation $\cup$:

$$(1) \quad X \cup \emptyset \quad = \quad X\,.$$

Intuitively, $\cup$ and $\emptyset$ have the meaning of the set union operator and empty set, respectively.

The $\cup$-based representation leads to more complex constraint solving procedures than those needed for the base CLP language. Consider, for example, the problem of handling $=$ constraints in the two different representation schemes: given any two $\mathcal{F} \cup \mathcal{V}$ terms ℓ and r, we need to determine a complete set of T-unifiers of $\ell = r$, where T is the underlying equational theory which describes the relevant properties of the set constructor symbols (i.e., either $\{\cdot \mid \cdot\}$ or $\cup$).

When $\mathcal{F}$ is composed by $\cup$, $\emptyset$, and by an arbitrary number of constant symbols, the unification problem belongs to the class of *$ACI1$-unification problems with constants*. Various solutions to this problem have been studied in the literature [7, 11, 44]. $ACI1$-unification with constants does not distinguish explicitly between sets and elements of sets. This makes it difficult to handle set unification when sets are defined by enumerating their elements, especially when elements are allowed to be variables. For example, the problem

$$\{X_1, X_2, X_3\} = \{a, b\} \tag{3}$$

(which admits 6 distinct solutions) is difficult to handle using $ACI1$-unification. One could map this to the $ACI1$-unification problem

$$X_1 \cup X_2 \cup X_3 = a \cup b \tag{4}$$

by interpreting the constants a and b as the singleton sets $\{a\}$ and $\{b\}$, and then "filtering" the 49 distinct $ACI1$-unifiers. This process involves discarding the solutions in which (at least) one of the X_i's is mapped to $\emptyset$ or to $a \cup b$. This is an impractical way of solving this problem in the general case. E.g., the problem $X_1 \cup \cdots \cup X_7 = a \cup b$ admits 16129 unifiers instead of the 126 of $\{X_1, \ldots, X_7\} = \{a, b\}$ [6].

Furthermore, this technique does not allow nested sets to be taken into account at all. Conversely, the $\{\cdot \mid \cdot\}$-based representation naturally accommodates for *nested* sets. Thus, for instance, problem (1) can be rendered directly as $\{X_1, X_2, X_3\} = \{a, b\}$, i.e., $\{X_1 \mid \{X_2 \mid \{X_3 \mid \emptyset\}\}\} = \{a \mid \{b \mid \emptyset\}\}$, and set unification algorithms working with the $\{\cdot \mid \cdot\}$-based representation of sets [39, 22, 63, 6] return exactly the 6 most general unifiers without the need of any filtering of solutions.

Therefore, the $\{\cdot \mid \cdot\}$-based representation allows us to solve set unification problems which cannot be expressed using $ACI1$-unification with constants. On the other hand, the $\cup$-based representation allows to write set terms that cannot be directly expressed using a $\{\cdot \mid \cdot\}$-representation, as for instance the term $X \cup a \cup Y$. The $\{\cdot \mid \cdot\}$-based representation, in fact, can only represent the union of a sequence of singletons with, eventually, a single variable.

A viable approach to tackle the problems described above when using the $\cup$-based representation is to introduce a unary free functor $\{\cdot\}$ in Σ. Under this assumption, the set $\{s_1, \ldots, s_m\}$ can be described as $\{s_1\} \cup \cdots \cup \{s_m\}$. A proposal in this direction is [8] which shows how to obtain a *general* $ACI1$ unification algorithm by combining $ACI1$-unification for $\emptyset$, $\cup$ and constants, with unification in the free theory for all other symbols. The generality of the combination procedure of [8], however, leads to the generation of a large number of non-deterministic choices which makes the approach hardly applicable in practice. A more practical specialized algorithm for general $ACI1$ unification has been recently proposed [28]. The solution described in this paper, in contrast, assumes that the $\{\cdot \mid \cdot\}$-based representation of sets is used—thus allowing to preserve its advantages—but it introduces in addition a union operator as a primitive constraint of the language.

A detailed and more complete analysis of the various approaches to set unification can be found in [28].

Other proposals for representation of sets in a (constraint) logic programming context have appeared in the literature:

- [41, 57] make use of an infinite collection of function symbols of different arity; the set $\{a_1, \ldots, a_n\}$ therefore is encoded as the term $\{\}_n(a_1, \ldots, a_n)$, using the n-ary functor $\{\}_n$.

  This approach allows only to express set terms with a known upper-bound to their cardinality. No partially specified sets can be described in this language. Moreover, stating equality in axiomatic form requires a non-trivial axiom scheme, such as: *for each pair of natural numbers m and n,*

$$\{\}_m(X_1,\ldots,X_m) = \{\}_n(Y_1,\ldots,Y_n) \quad \leftrightarrow \quad \bigwedge_{i=1}^{m}\bigvee_{j=1}^{n} X_i = Y_j \wedge \bigwedge_{j=1}^{n}\bigvee_{i=1}^{m} X_i = Y_j.$$

- In [31] sets are intended as subsets of a finite domain D of objects. At the language level, each ground set is represented as an individual constant, where all constants are partially ordered to reflect the $\subseteq$ lattice.

To summarize, the choice of using $\{\cdot \mid \cdot\}$ as the set constructor symbol is justified by:

- the desire to reduce as much as possible the non-determinism generated by unification,
- $\{\cdot \mid \cdot\}$ naturally supports iteration over the elements of a set through recursion, in a list-like fashion,
- it is easy to adapt the constraint solver to other data structures akin to sets, such as multisets and compact lists (as shown in [25]) .

The power of the union operator can be recovered by introducing the predicate symbol $\cup_3$ in the language. Moreover, since we deal with union as a constraint—rather than as an interpreted function symbol—we are not forced to apply variable substitutions which involve union operators (e.g., $X = Y \cup Z$), but we can store them as primitive constraints (e.g., $\cup_3(Y, Z, X)$), and still guarantee the global satisfiability of the constraint.

8 The Language

CLP($\mathcal{SET}$) is an instance of the general CLP scheme. As such it inherits from this scheme its general features: the (syntactic) form of a program, the notion of constraint, and the operational and logical semantics [35].

To characterize the CLP($\mathcal{SET}$) language, however, we must further define the signature upon which it is built, the structure used to assign a meaning to its symbols, and the input and output for the constraint solver. All these components will be described in this section. Some of them will be further elaborated in the successive sections, in order to provide more technical and precise details.

8.1 Syntax

The signature Σ upon which the language CLP($\mathcal{SET}$) is based is composed by the set $\mathcal{F}$ of function symbols that includes $\emptyset$ and $\{\cdot \mid \cdot\}$, by the set $\Pi_c = \{=, \in, \cup_3, ||, \mathsf{set}\}$ of constraint predicate symbols, and by a denumerable set $\mathcal{V}$ of variables. set is a unary predicate symbol used for "sort checking". Its meaning and use will be explained in the sequel.

The sorts of the function symbols in $\mathcal{F}$ and of the predicate symbols in Π_c are those introduced in Sections 5 and 6. Moreover, each user defined predicate symbol p in Π_u has the sort:

$$\langle \underbrace{\{\mathsf{Set}, \mathsf{Ker}\}, \ldots, \{\mathsf{Set}, \mathsf{Ker}\}}_{ar(p)} \rangle$$

Definition 8.1 *A term (atom, constraint, clause, goal, program) is* well-formed *if it is built from symbols in $\Sigma \cup \mathcal{V}$ respecting the sorts of the symbols it involves.*

When not specified otherwise, we will implicitly assume that all entities we deal with are well-formed. Detection and management of ill-formed entities will be discussed later on in this section.

Definition 8.2 *The* primitive constraints *in CLP($\mathcal{SET}$) are all the positive literals built from symbols in $\Pi_c \cup \mathcal{F} \cup \mathcal{V}$. A* constraint *is a conjunction of primitive constraints and negations of primitive constraints, with the exception of literals of the form $\neg\mathsf{set}(\cdot)$.*

The CLP($\mathcal{SET}$) syntax relies on the usual syntactic conventions of CLP and PROLOG.

Example 8.3 *The following is a (well-formed) CLP($\mathcal{SET}$) program*

```
collect_p( ∅, Y ).
collect_p( { X | R }, Y ) :-
    X ∉ R,
    p(X,Y),
    collect_p( R,Y ).
```

8.2 Interpretation

We define now the structure $\mathcal{SET} = \langle \mathcal{S}, (\cdot)^{\mathcal{S}} \rangle$ which allows us to assign a precise meaning to the syntactic entities we have introduced so far.

The interpretation domain $\mathcal{S}$ is a subset of the set of ground terms built from symbols in $\mathcal{F}$, i.e., $T(\mathcal{F})$, respecting the sorts. Terms are partitioned into equivalence classes according to the set-theoretical properties expressed by the two axioms (Ab) and $(C\ell)$. Thus, for example, $\{b, a\}$, $\{a, a, b\}$, $\{a, b, b\}$, are all placed in the same equivalence class. One of the objects in an equivalence class is selected—according to a suitable criteria—as the *representative* of the equivalence class. The interpretation function is designed to map each syntactic entity t not to the t itself (as in the standard Herbrand interpretation used in pure logic programming) but to the representative of the equivalence class t belongs to. This guarantees that all terms in the same class have the same "meaning"; in particular, if they are set terms, they denote the same set.

More formally, let us consider the least congruence relation $\cong$ over $T(\mathcal{F})$ which contains the equational axioms (Ab) and $(C\ell)$. This relation induces a partition of $T(\mathcal{F})$ into equivalence classes. The set of these classes will be denoted by $T(\mathcal{F})/\cong$. A total ordering $\prec$ on $T(\mathcal{F})$ can be used to identify a *representative* term from each congruence class in $T(\mathcal{F})/\cong$ [22]. For instance, assuming $a \prec b$, $\{a, b\}$ is the representative term of the class containing $\{b, a\}$, $\{a, a, b\}$, $\{a, b, b\}$, and so on. We define a function τ that maps each ground term t to its representative:

- $\tau(f(t_1, \ldots, t_n)) = f(\tau(t_1), \ldots, \tau(t_n))$, if $f \in \Sigma$, $f \not\equiv \{\cdot \mid \cdot\}$ and $ar(f) = n \geq 0$;
- $$\tau(\{t_1 \mid t_2\}) = \begin{cases} \tau(t_2) & \text{if } \tau(t_2) \equiv \{s_1, \ldots, \tau(t_1), \ldots, s_n \mid \emptyset\} \\ \{s_1, \ldots, s_i, \tau(t_1), s_{i+1}, \ldots, s_n \mid \emptyset\}) & \text{if } \tau(t_2) \equiv \{s_1, \ldots, s_n \mid \emptyset\}, s_i \prec \tau(t_1) \prec s_{i+1} \end{cases}$$

We assume that the test of the second condition implies that $\tau(\{t_1 \mid t_2\}) = \{\tau(t_1), s_1, \ldots, s_n \mid \emptyset\}$ when $\tau(t_1) \prec s_1$ and $\tau(\{t_1 \mid t_2\}) = \{s_1, \ldots, s_n, \tau(t_1) \mid \emptyset\}$ when $s_n \prec \tau(t_1)$.

Hence, each set term is mapped to a set term in a normalized form, where duplicates have been removed and elements are listed in a predefined order. On the other hand, a non-set term which does not containing any set term, e.g., $f(a)$, is mapped to the term itself. The *domain* $\mathcal{S}$ is therefore defined as:

$$\mathcal{S} = \{\tau(t) \ : \ t \in T(\mathcal{F})\}$$

$\mathcal{S}$ can be split in two domains according to the sort of its elements:

$$\begin{aligned} \mathcal{S}_1 &= \{s \in \mathcal{S} \ : \ s \text{ is of sort } \mathsf{Set}\} \\ \mathcal{S}_2 &= \{s \in \mathcal{S} \ : \ s \text{ is of sort } \mathsf{Ker}\} \end{aligned}$$

$\mathcal{S}_1$ and $\mathcal{S}_2$ are disjoint sets, and they represent the domains of the sorts Set and Ker.

The *interpretation function* $(\cdot)^{\mathcal{S}}$ over Σ is defined as:

- $(t)^{\mathcal{S}} = \tau(t)$ for any term t in $T(\mathcal{F})$.

Moreover, let t, t_i, u_i, v_i, $i \geq 0$, be elements of $\mathcal{S}$ of sort $\mathsf{Set} \cup \mathsf{Ker}$, and let s, s_i be elements of $\mathcal{S}$ of sort Set. The interpretation function for symbols in Π_c is defined as:

- $t_1 =^{\mathcal{S}} t_2$ if and only if $t_1 \equiv t_2$
- $t \in^{\mathcal{S}} s$ with $s \equiv \{u_1, \ldots, u_n\}$, $n \geq 0$, if and only if $\exists i \leq n, t \equiv u_i$

- $\cup_3^{\mathcal{S}}(s_1, s_2, s_3)$ with $s_1 \equiv \{t_1, \ldots, t_n \,|\, \emptyset\}$, $s_2 \equiv \{u_1, \ldots, u_m \,|\, \emptyset\}$, $s_3 \equiv \{v_1, \ldots, v_k \,|\, \emptyset\}$, if and only if $\{t_1, \ldots, t_n, u_1, \ldots, u_m\}^{\mathcal{S}} \equiv \{v_1, \ldots, v_k\}$, with $n, m, k \geq 0$
- $s_1 || ^{\mathcal{S}} s_2$ with $s_1 \equiv \{t_1, \ldots, t_n \,|\, \emptyset\}$, $s_2 \equiv \{u_1, \ldots, u_m \,|\, \emptyset\}$, if and only if $\forall i \leq n, j \leq m$, $t_i \not\equiv u_j$, with $n, m \geq 0$
- $\mathsf{set}^{\mathcal{S}}(t)$ if and only if $t \equiv \{u_1, \ldots, u_n \,|\, \emptyset\}$, $n \geq 0$.

Example 8.4 *Let us consider a valuation $\sigma : \mathcal{V} \longrightarrow \mathcal{S}$ such that*

$$\sigma(X) = a, \sigma(Y) = \{a, b \,|\, \emptyset\}, \sigma(Z) = \{b \,|\, \emptyset\}, \sigma(W) = \{a, b \,|\, \emptyset\}.$$

We have that $\sigma(X) \in \mathcal{S}_1$, while $\sigma(Y), \sigma(Z), \sigma(W) \in \mathcal{S}_2$.

1. *σ is a successful valuation for the constraint $X \in Y \wedge X \in W$ in the structure $\mathcal{SET}$.*
2. *σ is a successful valuation for the constraint $\{X \,|\, Y\} = Y$ on $\mathcal{SET}$ since*

$$\sigma(\{X \,|\, Y\}) = \{\sigma(X) \,|\, \sigma(Y)\}^{\mathcal{S}} = \{a \,|\, \{a, b \,|\, \emptyset\}\} = \{a, b \,|\, \emptyset\}$$

3. *σ is a valuation for the constraint $\cup_3(\emptyset, Z, Y)$, but not a successful one; in fact, $\sigma(\cup_3(\emptyset, Z, Y))$ is not satisfied in $\mathcal{SET}$, since 'a' is neither an element of $\emptyset$ nor an element of Z, while it is an element of Y.*
4. *σ is not a valuation for the constraint $X||Y$ since it assigns the non-set object 'a' to the variable X that must be of sort* Set *in order to fulfill the literal $X||Y$.*

The interpretation of the negative literals built using the symbols in Π_c is obtained by simply considering the negation of the interpretation for the corresponding positive literals, still restricted to well-formed formulae. Thus, for instance, $t \notin X$ is the negation of $t \in X$, provided X is of sort Set. Otherwise, if X is not of sort Set, then both $t \in X$ and $t \notin X$ are unsatisfiable in the underlying set structure $\mathcal{SET}$.

Remember that all (free) variables in a goal are assumed to be existentially quantified in front of the goal itself. Therefore, proving that a constraint is satisfiable in a given structure means proving that there exists at least one valuation of its free variables that makes this goal true in the given structure. Thus, for example, $X = A \wedge A \neq B$ is satisfied by the valuation σ such that $\sigma(X) = \emptyset, \sigma(A) = \emptyset, \sigma(B) = \{\emptyset\}$. Conversely, $A \in X \wedge X = a$ is unsatisfiable.

8.3 Constraint solving

The *admissible constraints* in CLP($\mathcal{SET}$) are all the constraints introduced in Sect. 8.1. The constraint satisfiability test is performed by the procedure $SAT_{\mathcal{SET}}$ (the CLP($\mathcal{SET}$) constraint solver). $SAT_{\mathcal{SET}}$ non-deterministically transforms the given constraint C to either false, error, or to a finite collection of constraints in *solved form*.

Definition 8.5 *Let C be a constraint. A literal c of C is in* solved form *if it is in one of the following forms:*

(i) $X = t$, and X does not occur in t and in the rest of C;

(ii) $X \neq t$, and X does not occur in t;

(iii) $t \notin X$, and X does not occur in t;

(iv) $\cup_3(X_1, X_2, X_3)$, with $X_1 \not\equiv X_2$, and there are no disequations of the form $X_i \neq t$ or $t \neq X_i$ in C for any $i = 1, 2, 3$;

(v) $X_1||X_2$ and $X_1 \not\equiv X_2$;

(vi) $\mathsf{set}(X)$.

A constraint C is in solved form if it is empty or all its components are simultaneously in solved form.

The selected solved forms are defined to allow trivial verification of satisfiability. As we will prove in Theorem 11.4, a constraint in solved form turns out to be always satisfiable in the considered structure $\mathcal{SET}$. The solved form also captures the notion of a constraint which cannot be further simplified, also called an *irreducible constraint*. As such, a constraint in solved form is returned as part of the computed answer whenever the computation terminates with success.

Note that the conditions for the solved form of $\cup_3$ constraints (condition *(iv)*) are aimed at disallowing unsatisfiable constraints such as

$$\cup_3(X, Y, Z) \wedge \cup_3(X, Y, W) \wedge Z \neq W.$$

Similarly, the condition *(v)* for $||$ constraints is aimed at avoiding constraints like

$$X||X \wedge X \neq \emptyset,$$

which are not satisfiable in the context we are dealing with.

Example 8.6 *Consider the constraint (not in solved form)*

$$X \in \{A, B\} \wedge \{X\} \neq \{A, B\}$$

(X, A, B variables). Given this constraint as input to $SAT_{\mathcal{SET}}$, the procedure will non-deterministically produce the two constraints in solved form:

$$X = A \wedge A \neq B$$

and

$$X = B \wedge A \neq B.$$

Both constraints are trivially satisfiable in the underlying set structure $\mathcal{SET}$.

Example 8.7 *The $ACI1$ unification problem mentioned in Section 7, $X_1 \cup X_2 \cup X_3 = \{a, b\}$ can be written as a conjunction of two primitive constraints:*

$$\cup_3(X_1, X_2, X) \wedge \cup_3(X, X_3, \{a, b\}).$$

The execution of $SAT_{\mathcal{SET}}$ on this constraint will return (non-deterministically) the 49 distinct answers. One of them is: $X_1 = \{a\}, X_2 = \{b\}, X = \{a, b\}, X_3 = \{a\}$.

Moreover, we will prove—see Section 11.3—that, given a well-formed constraint C the disjunction of all the constraints in solved form generated by $SAT_{\mathcal{SET}}(C)$ is *equi-satisfiable* to C. Therefore, if $SAT_{\mathcal{SET}}$ is able to transform a given constraint C to (at least) a constraint in solved form C' then it can be concluded that C is satisfiable. Otherwise, if $SAT_{\mathcal{SET}}$ can rewrite C to only error or false (depending on whether $SAT_{\mathcal{SET}}$ detects a sort violation or not) then C is unsatisfiable.

We assume that all checks needed to detect the presence of ill-formed syntactic objects are performed at run-time. Therefore, we admit that constraints passed to the $SAT_{\mathcal{SET}}$ procedure may contain ill-formed terms and literals, and that ill-formed terms and literals may be generated during the constraint simplification process due to the instantiation of variables. The run-time check of sorts is performed by introducing set constraints that are suitably managed within the $SAT_{\mathcal{SET}}$ procedure. This is why $SAT_{\mathcal{SET}}$ can terminate also with error as its result.

Note that both false and error are used to denote the logical value *false*. We prefer to distinguish between these two constants since we consider important from a practical point of view to be able to distinguish type errors from the other cases where a negative answer is return. This is also the choice adopted in most CLP systems.

9 CLP($\mathcal{SET}$) Constraint Simplification Procedures

In this section we describe the rewriting procedures used by $SAT_{\mathcal{SET}}$ to rewrite a CLP($\mathcal{SET}$)-constraint to its equivalent solved forms.

A constraint C can be conveniently seen as the conjunction $\bigwedge_\alpha C_\alpha$ where $\alpha \in \{=, \in, \cup_3, ||, \neq, \notin, \not\downarrow_3, / ||, \mathsf{set}\}$ and each constraint C_α is composed only by the positive literals based on α. For each such α we define a non-deterministic rewriting procedure which is able to rewrite the C_α component of a given constraint C into a disjunction of constraints C' in solved form.

The various rewriting procedures are shown in Figures 1–9. In these figures we will rely on the following notation: s, s_i, t, t_i are generic terms, f, g are distinct function symbols and X, Y, Z are variables. $\{_ \,|\, _\}$ is a generic term which has $\{\cdot \,|\, \cdot\}$ as its outermost function symbol. The symbol $\equiv$ denotes the syntactic equality. If a variable occurs only in the right-hand side of a rewriting rule, then it must be intended as a "newly generated" variable, distinct from all the others. In the algorithms we denote these variables by N, N_1, and N_2.

The constraint rewriting procedures assume that well-formedness of the input constraint has been already checked before entering the procedures themselves. This implies that the initial constraint C may already contain a number of constraints based on set.

Finally, note that all rules in each rewriting procedure are defined in a such a way they can be applied to the input constraint only in a mutually exclusive manner.

9.1 = constraints

= constraints are managed by the set unification algorithm defined in [22] and described in Fig. 1. For any given conjunction of equations C, possibly involving set terms, this algorithm is able to compute through non-determinism each element of a complete set of solutions to C. Solutions are expressed as conjunctions of = constraints in solved form.

The function tail is used to compute the "tail" of a set term, i.e., the innermost non-set term appearing as second argument of $\{\cdot \,|\, \cdot\}$. This procedure can be defined as follows:

$$\begin{cases} \mathsf{tail}(\emptyset) &= \emptyset \\ \mathsf{tail}(X) &= X \qquad \text{if } X \text{ is a variable} \\ \mathsf{tail}(\{s \,|\, t\}) &= \mathsf{tail}(t) \end{cases}$$

The procedure equal is a generalization of the traditional unification between Herbrand terms. The key additional step is represented by steps (9) and (10) which provide rewriting of equalities between set terms, by essentially unfolding them into equalities between the elements of the two sets.

equal is the only rewriting procedure that must take into account the fact that the constraint it is dealing with can be ill-formed. This could happen when, applying variable substitution, a variable which is constrained by a set constraint to be of sort Set is instantiated to a non-set term. For the sake of efficiency, however, we have preferred not checking the set constraints after each substitution. We have instead added a special case, rule (11), and a control in rule (5), to detect the possible creation of ill-formed terms and, in this case, to rewrite the constraint C to error, thus causing equal to terminate immediately.

Example 9.1 *Given the equation*

$$\{X \,|\, R\} = \{Y \,|\, S\},$$

X, Y, R, and S variables, the set unification algorithm returns the four solutions:

$$X = Y, R = S$$
$$X = Y, S = \{X \,|\, R\}$$
$$X = Y, R = \{Y \,|\, S\}$$
$$R = \{Y \,|\, N\}, S = \{X \,|\, N\}, \mathsf{set}(N).$$

equal(C) : while $C_=$ is not in solved form and $C \neq$ false and $C \neq$ error do apply one of the following rules to C:			
(1)	$X = X \wedge C'$	$\mapsto$	C'
(2)	$t = X \wedge C'$ t is not a variable	$\mapsto$	$X = t \wedge C'$
(3)	$X = f(t_1, \ldots, t_n) \wedge C'$ $f \not\equiv \{\cdot \mid \cdot\}, X \in vars(t_1, \ldots, t_n)$	$\mapsto$	false
(4)	$X = \{t_0, \ldots, t_n \mid t\} \wedge C'$ t is $\emptyset$ or a variable, $X \in vars(t_0, \ldots, t_n)$	$\mapsto$	false
(5)	$X = t \wedge C'$ $X \notin vars(t)$, t is a set term $\mathsf{orset}(X) \notin C'$	$\mapsto$	$X = t \wedge C'[X/t]$
(6)	$X = \{t_0, \ldots, t_n \mid X\} \wedge C'$ $X \notin vars(t_0, \ldots, t_n)$	$\mapsto$	$X = \{t_0, \ldots, t_n \mid N\} \wedge \mathsf{set}(N) \wedge C'$
(7)	$f(s_1, \ldots, s_m) = g(t_1, \ldots, t_n) \wedge C'$ $f \not\equiv g$	$\mapsto$	false
(8)	$f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \wedge C'$ $f \not\equiv \{\cdot \mid \cdot\}$	$\mapsto$	$s_1 = t_1 \wedge \cdots \wedge s_n = t_n \wedge C'$
(9)	$\{t \mid s\} = \{t' \mid s'\} \wedge C'$ tail(s),tail(s') are not the same variable	$\mapsto$	$C' \wedge$ *any of* (i) $t = t' \wedge s = s'$ (ii) $t = t' \wedge \{t \mid s\} = s'$ (iii) $t = t' \wedge s = \{t' \mid s'\}$ (iv) $s = \{t' \mid N\} \wedge \{t \mid N\} = s' \wedge \mathsf{set}(N)$
(10)	$\{t_0, \ldots, t_m \mid X\} = \{t'_0, \ldots, t'_n \mid X\} \wedge C'$	$\mapsto$	$C' \wedge$ *any of* (i) $t_0 = t'_j \wedge \{t_1, \ldots, t_m \mid X\} = \{t'_0, \ldots, t'_{j-1}, t'_{j+1}, \ldots, t'_n \mid X\}$ (ii) $t_0 = t'_j \wedge \{t_0, \ldots, t_m \mid X\} = \{t'_0, \ldots, t'_{j-1}, t'_{j+1}, \ldots, t'_n \mid X\}$ (iii) $t_0 = t'_j \wedge \{t_1, \ldots, t_m \mid X\} = \{t'_0, \ldots, t'_n \mid X\}$ (iv) $X = \{t_0 \mid N\} \wedge \{t_1, \ldots, t_m \mid N\} = \{t'_0, \ldots, t'_n \mid N\} \wedge \mathsf{set}(N)$ for any j in $\{0, \ldots, n\}$.
(11)	$X = f(t_1, \ldots, t_n) \wedge \mathsf{set}(X) \wedge C'$ $f \not\equiv \emptyset, f \not\equiv \{\cdot \mid \cdot\}$	$\mapsto$	error

Figure 1: Equality Rewriting Rules

The algorithm in Figure 1, with only minor differences due to the absence of any notion of sort, has been presented and explained in [22]; it has also been described in a different context in [25]. Thus, we will not give further details here.

General $(Ab)(C\ell)$-unification algorithms—i.e., algorithms capable of handling a signature containing arbitrary free function symbols—have been proposed in [39, 22, 63, 6, 28]. The algorithm in [39] is weaker than the others, since its aim is to solve matching problems instead of unification problems. [63] solves exactly the same problems as ours, using an elegant algorithm based on membership constraints. Moreover, the algorithm reduces the generation of redundant solutions of [22]. Finally, the algorithm in [6], though less elegant than that of [63], further reduces the number of redundancies. In particular, it is proved to be minimal for a number of sample set unification problems that are proposed as 'benchmarks'.

9.2 $\in$ constraints

Membership constraints of the form $s \in t$ can be completely eliminated by replacing them with suitable equality constraints. This is justified by the following equivalence that holds in the underlying set theory (see Section 11.1 for the precise definition of this theory):

$$s \in t \leftrightarrow \exists N\,(t = \{s \mid N\} \wedge \mathsf{set}(N)).$$

Hence, the rewriting procedure for $C_\in$ (Figure 2) simply generates new equality constraints. These new constraints, in general, will be further processed by the solver for the $=$ constraints presented in the previous section. It is important to observe that no $\in$ constraints are left in the final solved form.

The rewriting procedure for $C_\in$, called member, is shown in Figure 2.

Example 9.2 *The constraint*

$$a \in \{X, b, Y \mid Z\}$$

is non-deterministically reduced as follows:

$$\begin{array}{rcll} a = X & \mapsto & X = a & \mathsf{equal}(2) \\ a = b & \mapsto & \mathsf{false} & \mathsf{equal}(7) \\ a = Y & \mapsto & Y = a & \mathsf{equal}(2) \\ a \in Z & \mapsto & Z = \{a \mid N\}, \mathsf{set}(N) & \mathsf{member}(3) \end{array}$$

member(C) : while $C_\in$ is not in solved form and $C \neq$ false and $C \neq$ error do apply one of the following rules to C:			
(1)	$s \in \emptyset \wedge C'$	$\mapsto$	false
(2)	$r \in \{s \mid t\} \wedge C'$	$\mapsto$	$C' \wedge$ *any of* (i) $r = s$ (ii) $r \in t$
(3)	$t \in X \wedge C'$	$\mapsto$	$X = \{t \mid N\} \wedge \mathsf{set}(N) \wedge C'$

Figure 2: Membership rewriting procedure

9.3 $\neq$ constraints

The rewriting procedure for $C_\neq$, called not_equal, is shown in Figure 3.

Most of the rules of not_equal are rather straightforward consequences of the axiomatization of $=$ for Herbrand terms (Clark's Equality Theory—see also Section 11.1). Some remarks are needed regarding rule (8). The constraint $\{s \mid r\} \neq \{u \mid t\}$ needs to be replaced either by the constraint $N \in \{s|r\} \wedge N \notin \{u|t\}$ or by the constraint $N \in \{u|t\} \wedge N \notin \{s|r\}$, where N denotes a new variable. This corresponds to the intuitive idea that two sets are different if one contains an element which does not appear in the other.

not_equal(C) :

while $C_{\neq}$ is not in solved form and $C \neq$ false and $C \neq$ error do

apply one of the following rules to C:

(1)	$f(s_1, \ldots, s_m) \neq g(t_1, \ldots, t_n) \wedge C'$	$\mapsto$	C'
(2)	$f(s_1, \ldots, s_n) \neq f(t_1, \ldots, t_n) \wedge C'$ $f \not\equiv \{\cdot \mid \cdot\}, n > 0$	$\mapsto$	$C' \wedge$ *any of* (i) $s_1 \neq t_1$ $\vdots$ $\vdots$ (n) $s_n \neq t_n$
(3)	$s \neq s \wedge C'$ s is a constant or a variable	$\mapsto$	false
(4)	$t \neq X \wedge C'$ t is not a variable	$\mapsto$	$X \neq t \wedge C'$
(5)	$X \neq f(t_1, \ldots, t_n) \wedge C'$ $f \not\equiv \{\cdot \mid \cdot\}, X \in vars(t_1, \ldots, t_n)$	$\mapsto$	C'
(6)	$X \neq \{t_1, \ldots, t_n \mid t\} \wedge C'$ $X \in vars(t_1, \ldots, t_n)$	$\mapsto$	C'
(7)	$X \neq \{t_1, \ldots, t_n \mid X\} \wedge C'$ $X \notin vars(t_1, \ldots, t_n)$	$\mapsto$	$C' \wedge$ *any of* (i) $t_1 \notin X$ $\vdots$ $\vdots$ (n) $t_n \notin X$
(8)	$\{s \mid r\} \neq \{u \mid t\} \wedge C'$	$\mapsto$	$C' \wedge$ *any of* (i) $N \in \{s \mid r\} \wedge N \notin \{u \mid t\}$ (ii) $N \in \{u \mid t\} \wedge N \notin \{s \mid r\}$

Figure 3: Rewriting procedure for disequations

Observe that the newly generated membership constraints can be immediately removed by applying the member procedure.

Example 9.3 *The constraint*

$$f(a, \{b, c\}) \neq f(X, \{X, Y\})$$

is non-deterministically reduced to either $X \neq a$ *or* $\{b, c\} \neq \{X, Y\}$ *(rule* (2)*). This second constraint leads to the following alternative solutions (after an additional application of the* member *procedure):* $b \notin \{X, Y\}$, $c \notin \{X, Y\}$, $X \notin \{b, c\}$, *and* $Y \notin \{b, c\}$. *The application of the* not_member *procedure described in the next section will simplify these and lead to the final solutions:*

$$\begin{array}{l} X \neq b, Y \neq b \\ X \neq c, Y \neq c \\ X \neq b, X \neq c \\ Y \neq b, Y \neq c. \end{array}$$

9.4 $\notin$ constraints

The rewriting procedure for $C_{\notin}$, called not_member, is shown in Figure 4. It is based on the axiomatic definition of $\{\cdot \mid \cdot\}$ (Section 11.1), on the fact that standard (non-set) Herbrand terms are treated as atomic (non-set) entities, and on the requirement of disallowing membership to form cycles—i.e., the well-founded nature of $\in$. The key step is represented by rule (2), which reduces a constraint of the form $r \notin \{s \mid t\}$ to the equivalent conjunction $r \neq s \wedge s \notin t$.

not_member(C) : while $C_{\notin}$ is not in solved form and $C \neq$ false and $C \neq$ error do apply one of the following rules to C:			
(1)	$s \notin \emptyset \wedge C'$	$\mapsto$	C'
(2)	$r \notin \{s \mid t\} \wedge C'$	$\mapsto$	$r \neq s \wedge r \notin t \wedge C'$
(3)	$t \notin X \wedge C'$ $X \in vars(t)$	$\mapsto$	C'

Figure 4: Rewriting procedure for negated membership

Note that the correctness of rule (3) requires that the variable X is constrained to be of sort Set by a constraint $\mathsf{set}(X)$. However, since we have assumed that correctness of the sorts has been already checked before entering the rewriting procedure, we can assume that the constraint $\mathsf{set}(X)$ is already present in C.

Example 9.4 *The constraint*

$$\{c \mid X\} \neq \{b, c\}$$

is rewritten by rule (8i) *of* not_equal *to* $N \in \{c \mid X\}, N \notin \{b, c\}$. *Then, using* member *and* equal *it is rewritten to:*

(i) $N = c, c \notin \{b, c\}$ *that is further rewritten by* not_member *to* $N = c, c \neq b, c \neq c, c \notin \emptyset$, *and from this to* false *by* not_equal*;*

(ii) $X = \{N \mid N_1\}, \mathsf{set}(N_1), N \notin \{b, c\}$ *that* not_member *rewrites to the solved form constraint:*

$$X = \{N \mid N_1\}, \mathsf{set}(N_1), N \neq b, N \neq c.$$

9.5 $\cup_3$ constraints

The rewriting procedure for $C_{\cup_3}$, called union, is shown in Figure 5. Rule (1) infers the equality between s and t from a constraint stating $t = s \cup s$. Rules (2) and (3) take care of the simple cases in which one of the arguments is the empty set. Rules (4) and (5), instead, manage the cases in which we know that at least one argument of the constraint is a non-empty set. These rules are based on the following observations:

- if $\{t \mid s\} = \xi_1 \cup \xi_2$ then t belongs to either ξ_1 or ξ_2 (or both)
- if $\xi = \{t \mid s\} \cup \xi_2$ then t belongs to ξ (and possibly also to ξ_2).

Rules (6) and (7) deal with variable arguments and they are needed to reach the solved form.

Example 9.5 *Consider the constraint*

$$\cup_3(\{X \mid \emptyset\}, \{Y \mid Z\}, V).$$

This constraint satisfies the conditions of rule (5) *of* union. *Let us follow only one of the possible non-deterministic computations this rule opens. Assume the first case (case* (*i*)*) is selected, thus rule* (5) *rewrites the given constraint to* $\{X \mid \emptyset\} = \{X \mid N_1\}, X \notin N_1, V = \{X \mid N\}, X \notin N, \cup_3(N_1, \{Y \mid Z\}, N)$.

equal *generates the solution* $N_1 = \emptyset$ *for the first constraint, while the last constraint can be again reduced using rule* (5) *of* union. *Selecting again case* (*i*)*, rule* (5) *rewrites this constraint to:* $\{Y \mid Z\} = \{Y \mid N_1'\}, Y \notin N_1', N = \{Y \mid N'\}, Y \notin N', \cup_3(N_1', N_1, N')$. *From this we obtain, among others, the equation* $N_1' = Z$ *(from the first set-set equality constraint). By applying the substitutions obtained from the equations* $N_1 = \emptyset$, $N_1' = Z$, *and* $N = \{Y \mid N'\}$ *we get:*

$$X \notin \emptyset, V = \{X, Y \mid N'\}, X \notin \{Y \mid N'\}, Y \notin Z, Y \notin N', \cup_3(Z, \emptyset, N')$$

Note that the newly generated variables N, N_1, *and* N_1' *can be completely removed after the application of the relevant substitutions.*

Now, the first constraint is simply eliminated from the constraint; union *(rule* (3)*) applied to the last constraint returns* $N' = Z$*; after the application of this substitution and other simple rules, we get the final solution (in solved form):*

$$X \neq Y, V = \{X, Y \mid Z\}, X \notin Z, Y \notin Z.$$

9.6 $||$ constraints

The rewriting procedure for $C_{||}$, called disj, is shown in Figure 6. It is able to rewrite constraints stating disjointness of two terms until the solved form is reached. Rule (1) guarantees that $\emptyset$ is disjoint from any other set. Rule (2) expresses the fact that two identical sets are disjoint only if they are empty. Rules (3) and (4) implement the intuitive notion of disjointness, by ensuring the existence of elements in one set do not belong to the other set.

Example 9.6 *The constraint*

$$\{X, Y\} || \{a \mid Z\}$$

is rewritten by disj *to* $X \neq a, X \notin Z, a \notin \{Y\}, \{Y\} || Z$. disj *rewrites the primitive constraint* $\{Y\} || Z$ *to* $Y \notin Z, \emptyset || Z$, *and then it completely eliminates the primitive constraint* $\emptyset || Z$. *Finally, after the application of the other procedures, we get:*

$$X \neq a, Y \neq a, X \notin Z, Y \notin Z.$$

union(C) :

while $C_{\cup_3}$ is not in solved form and $C \neq$ false and $C \neq$ error do

apply one of the following rules to C:

(1)	$\cup_3(s,s,t) \wedge C'$	$\mapsto$	$s = t \wedge C'$
(2)	$\cup_3(s,t,\emptyset) \wedge C'$ $s \not\equiv t$	$\mapsto$	$s = \emptyset \wedge t = \emptyset \wedge C'$
(3)	$\cup_3(\emptyset,t,X) \wedge C'$ or $\cup_3(t,\emptyset,X) \wedge C'$ $t \not\equiv \emptyset$	$\mapsto$	$X = t \wedge C'$
(4)	$\cup_3(s_1,s_2,\{t_1 \mid t_2\}) \wedge C'$ $s_1 \not\equiv s_2$	$\mapsto$	$\{t_1 \mid t_2\} = \{t_1 \mid N\} \wedge t_1 \notin N \wedge C' \wedge$ *any of* (i) $s_1 = \{t_1 \mid N_1\} \wedge t_1 \notin N_1 \wedge \cup_3(N_1,s_2,N)$ (ii) $s_2 = \{t_1 \mid N_1\} \wedge t_1 \notin N_1 \wedge \cup_3(s_1,N_1,N)$ (iii) $s_1 = \{t_1 \mid N_1\} \wedge t_1 \notin N_1 \wedge s_2 = \{t_1 \mid N_2\} \wedge t_1 \notin N_2 \wedge \cup_3(N_1,N_2,N)$
(5)	$\cup_3(\{t_1 \mid t_2\},t,X) \wedge C'$ or $\cup_3(t,\{t_1 \mid t_2\},X) \wedge C'$ $t \not\equiv \{t_1 \mid t_2\}, t \not\equiv \emptyset$	$\mapsto$	$\{t_1 \mid t_2\} = \{t_1 \mid N_1\} \wedge t_1 \notin N_1 \wedge X = \{t_1 \mid N\} \wedge t_1 \notin N \wedge C' \wedge$ *any of* (i) $t_1 \notin t \wedge \cup_3(N_1,t,N)$ (ii) $t = \{t_1 \mid N_2\} \wedge t_1 \notin N_2 \wedge \cup_3(N_1,N_2,N)$
(6)	$\cup_3(X,Y,Z) \wedge Z \neq t \wedge C'$ $X \not\equiv Y$	$\mapsto$	$\cup_3(X,Y,Z) \wedge C' \wedge$ *any of* (i) $N \in Z \wedge N \notin t$ (ii) $N \in t \wedge N \notin Z$ (iii) $Z = \emptyset \wedge t \neq \emptyset$
(7)	$\cup_3(X,Y,Z) \wedge X \neq t \wedge C'$ or $\cup_3(Y,X,Z) \wedge X \neq t \wedge C'$ $X \not\equiv Y$	$\mapsto$	$\cup_3(X,Y,Z) \wedge C' \wedge$ *any of* (i) $N \in X \wedge N \notin t$ (ii) $N \in t \wedge N \notin X$ (iii) $X = \emptyset \wedge t \neq \emptyset$

Figure 5: Rewriting Procedure for $\cup_3$

disj(C) : while $C_{\|\|}$ is not in solved form and $C \neq$ false and $C \neq$ error do apply one of the following rules to C:			
(1)	$\emptyset \,\|\|\, t \wedge C'$ or $t \,\|\|\, \emptyset \wedge C'$	$\mapsto$	C'
(2)	$X \,\|\|\, X \wedge C'$	$\mapsto$	$X = \emptyset \wedge C'$
(3)	$\{t_1 \mid t_2\} \,\|\|\, X \wedge C'$ or $X \,\|\|\, \{t_1 \mid t_2\} \wedge C'$	$\mapsto$	$t_1 \notin X \wedge X \,\|\|\, t_2 \wedge C'$
(4)	$\{t_1 \mid s_1\} \,\|\|\, \{t_2 \mid s_2\} \wedge C'$	$\mapsto$	$t_1 \neq t_2 \wedge t_1 \notin s_2 \wedge t_2 \notin s_1 \wedge s_1 \|\| s_2 \wedge C'$

Figure 6: Rewriting Procedure for $||$

9.7 $\not\cup_3$ constraints

The primitive constraints based on $\not\cup_3$ can be completely eliminated; hence, $C_{\not\cup_3}$ is empty at the end of the simplification process. This elimination process is performed by the rewriting procedure not_union, shown in Figure 7. The procedure contains a single non-deterministic rule, which relies on the traditional extensionality principle for equality between sets (see Sect. 11.1).

This procedure is considerably simpler than its positive counterpart union. This fact has a logical justification. In our context, truth of $Z = X \cup Y$ is equivalent to the truth of the formula:

$$\forall N\,(N \in Z \leftrightarrow (N \in X \vee N \in Y)) \tag{5}$$

On the other hand, verifying $Z \neq X \cup Y$ leads to the logical formula (obtained complementing formula (5)):

$$\exists N\,((N \in Z \wedge V \notin X \wedge N \notin Y) \vee (N \notin Z \wedge N \in X) \vee (N \notin Z \wedge N \in Y)) \tag{6}$$

Thus, occurrences of $\not\cup_3$ lead to existentially quantified formulae, that are easier to handle than a universally quantified one in our context.

not_union(C) : while $C_{\not\cup_3}$ is not in solved form and $C \neq$ false and $C \neq$ error do apply the following rule to C:			
(1)	$\not\cup_3(s_1, s_2, s_3) \wedge C'$	$\mapsto$	$C' \wedge$ *any of*
			(i) $N \in s_3 \wedge N \notin s_1 \wedge N \notin s_2$
			(ii) $N \in s_1 \wedge N \notin s_3$
			(iii) $N \in s_2 \wedge N \notin s_3$

Figure 7: Rewriting Procedure for $\not\cup_3$

Example 9.7 *Consider the constraint*

$$\not\cup_3(X, Y, \{a, b\}).$$

Let us follow one of the possible branches of its non-deterministic rewriting. By rule (1)*, case* (i)*, the constraint is reduced to* $N \in \{a, b\}, N \notin X, N \notin Y$*. The first conjunct leads to two solutions,* $N = a$ *and* $N = b$*. If we consider the first one, we obtain*

$$a \notin X, a \notin Y$$

while the second leads to

$$b \notin X, b \notin Y.$$

Both these constraints are in solved form.

9.8 $\not|$ constraints

The primitive constraints based on $\not|$ can be completely eliminated. Hence, $C_{\not|}$ is empty at the end of the simplification process. The rewriting procedure for $\not|$, called not_disj, is shown in Figure 8. The procedure contains a single rule which is used to ensure the existence of an element which lies in the intersection of the two sets.

Also in this case a logical consideration can be performed. $X||Y$ is equivalent to $\forall N\,(N \in X \rightarrow N \notin Y)$ while its negation is equivalent to the simpler existential formula $\exists N\,(N \in X \wedge N \in Y)$.

not_disj(C) : while $C_{\not|}$ is not in solved form and $C \neq$ false and $C \neq$ error do apply the following rule to C:			
(1)	$s \not| t \wedge C'$	$\mapsto$	$N \in s \wedge N \in t \wedge C'$

Figure 8: Rewriting Procedure for $\not|$

Example 9.8 *Let us consider the constraint*

$$\{a\} \not| \{X, b\}.$$

The not_disj *procedure reduces this constraint to* $N \in \{a\}, N \in \{X, b\}$. *The application of the* member *procedure to the first conjunct leads to* $a \in \{X, b\}$. *Once again the* member *procedure can be used to produce non-deterministically the two reductions* $X = a$ *and* $a = b$. *Only the first one will lead to a solved form.*

9.9 set constraints

set constraints are used to state which terms are constrained to belong to the sort Set. They are generated either by the $SAT_{\mathcal{SET}}$ rewriting procedures, to constrain newly generated variables occurring as tail variables of set terms, or by the set_infer procedure called inside $SAT_{\mathcal{SET}}$ (see Sect. 9.10). set constraints can be also added to a program by the user. The rewriting procedure for C_{set}, called set_check, is shown in Figure 9.

set_check(C) : while C_{set} is not in solved form and $C \neq$ false and $C \neq$ error do apply one of the following rules to C:			
(1)	$\mathsf{set}(\emptyset) \wedge C'$	$\mapsto$	C'
(2)	$\mathsf{set}(\{t \mid s\}) \wedge C'$	$\mapsto$	$\mathsf{set}(s) \wedge C'$
(3)	$\mathsf{set}(f(t_1, \ldots, t_m)) \wedge C'$, $f \not\equiv \{\cdot \mid \cdot\}, f \not\equiv \emptyset$	$\mapsto$	error

Figure 9: Rewriting Procedure for set

It is possible to prove that, with the exception of the newly generated tail variables, the individual rewriting procedures presented are capable of generating only well-formed constraints, as long as the input constraint is itself well-formed (Lemma 13.7).

9.10 The (Set) Constraint Solver

$SAT_{\mathcal{SET}}$ combines the constraint rewriting procedures for the different types of primitive constraints described in the previous sections. Given a constraint C, $SAT_{\mathcal{SET}}$ calls the rewriting procedures on the input constraint C in a predetermined order, as defined in the procedure STEP in Figure 10.

```
STEP(C) :   not_union(C);
            not_disj(C);
            member(C);
            union(C);
            disj(C);
            not_member(C);
            not_equal(C);
            equal(C).
```

Figure 10: The procedure STEP

The first three calls in STEP remove all the occurrences of $\not\in_3$, $\not|$, and $\in$ constraints from the constraint C. Observe also that the remaining rewriting procedures do not generate any constraints of type $\not|$ or $\not\in_3$—thus, in the successive executions of STEP the first two actions will be no-ops. The subsequent execution of the union and disj procedures allows the removal of all the occurrences of $\cup_3$ and $||$ that are not in solved form. union and disj, however, are capable of generating new constraints of the form $\neq$, $=$, $\notin$. The not_member procedure simplifies $\notin$ constraints, possibly generating new constraints of the form $\neq$, while the not_equal reduces the $\neq$ constraints, possibly introducing new constraints of the form $\in$ and $\notin$. Finally, the equal procedure simplifies the $=$ constraints, generating only new equations in solved form and, possibly, new set constraints. Substitutions computed by equal are immediately applied to the constraint C. These substitutions can possibly transform constraints from solved form to non-solved form.

Note that if C is rewritten to either false or error by any of the rewriting procedures within STEP, then all the subsequent calls to the other rewriting procedures will leave it unchanged.

At the end of the execution of the STEP procedure some non-solved primitive constraint may still occur in C. Either these constraints are introduced by a different rewriting procedure (e.g., not_equal generates $\in$ constraints) or they are the result of applying a substitution to a solved form constraint. Therefore the execution of STEP has to be iterated until a fixed-point is reached—i.e., any new rewritings do not further simplify the constraint. This happens exactly when the constraint is in solved form or it is false or error. The complete definition of the $SAT_{\mathcal{SET}}$ procedure is shown in Figure 11.

```
SAT_SET(C) :
    set_infer(C);
    repeat
        C' := C;
        set_check(C);
        STEP(C)
    until C = C'
```

Figure 11: The $SAT_{\mathcal{SET}}$ procedure

Before calling the STEP procedure, within the rewriting loop, $SAT_{\mathcal{SET}}$ calls the set_check procedure to check the set constraints. set_check removes all occurrences of set not in solved form, without generating any new constraints. If the result of this rewriting is error then it means that a sort violation has occurred. In this case $SAT_{\mathcal{SET}}$ immediately terminates.

Before entering the rewriting loop, $SAT_{\mathcal{SET}}$ calls the procedure set_infer, whose definition is shown in Figure 12. set_infer has the task of determining which variables in the constraint C are required to belong to the sort Set. For any such variable X, set_infer adds a new primitive constraint set(X) to C.

Within set_infer, the function find_set is used to find set terms, possibly occurring inside other terms, and to generate the corresponding set constraints. The definition of find_set is shown in Figure 13. As an

```
set_infer(C) :
    for each c ≡ p(t_1, ..., t_n) or c ≡ p̸(t_1, ..., t_n) in C do
        C := C ∧ find_set(t_1) ∧ ··· ∧ find_set(t_n);
        case c of
            t_1 ∈ t_2, t_1 ∉ t_2 : C := C ∧ set(t_2);
            t_1||t_2, t_1 ∦t_2 : C := C ∧ set(t_1) ∧ set(t_2);
            ∪_3 (t_1, t_2, t_3), ∅̸_3(t_1, t_2, t_3) : C := C ∧ set(t_1) ∧ set(t_2) ∧ set(t_3)
        endcase
    endfor
```

Figure 12: Inference of set constraints

example, the call find_set$(f(a, \{b, \{c \mid Y\} \mid X\}))$ will return the constraint set$(Y) \wedge$ set(X). We assume that all the true constraints possibly generated by find_set are immediately removed via a trivial pre-processing.

```
find_set(t) :
    if t ≡ X ot t is a constant symbol then return true;
    if t ≡ f(t_1, ..., t_n), n > 0, and f ≢ {· | ·} then return find_set(t_1) ∧ ··· ∧ find_set(t_n);
    if t ≡ {t_1, ..., t_n | t} then return find_set(t_1) ∧ ··· ∧ find_set(t_n) ∧ set(t);
```

Figure 13: Finding set terms

Example 9.9 *Consider the execution of $SAT_{\mathcal{SET}}$ on the constraint*

$$X \in \{A, B\}, \{X\} \neq \{A, B\}.$$

The first iteration of $SAT_{\mathcal{SET}}$ will either produce the constraint $X = A, \{A\} \neq \{A, B\}$ or $X = B, \{B\} \neq \{A, B\}$. The first constraint, $\{A\} \neq \{A, B\}$, can be reduced either to $X = A, Z \in \{A\}, Z \notin \{A, B\}$ or $X = A, Z \in \{A, B\}, Z \notin \{A\}$, where Z is a new variable. The former can be only rewritten to false*, whereas the latter is transformed to the constraint in solved form*

$$X = A, A \neq B.$$

Similarly, the second constraint—$X = B, \{B\} \neq \{A, B\}$—will lead to the solved form constraint

$$X = B, A \neq B.$$

Theorem 9.10 (Termination)
The $SAT_{\mathcal{SET}}$ procedure can be implemented in such a way to terminate for every input constraint C. Moreover, each formula returned by $SAT_{\mathcal{SET}}$ is either false*, or* error*, or a constraint in solved form.*

A complete proof of the theorem in presented in the Appendix.

The termination of $SAT_{\mathcal{SET}}$ and the finiteness of the number of non-deterministic choices generated during its computation, guarantee the finiteness of the number of constraints non-deterministically returned by $SAT_{\mathcal{SET}}$. Furthermore, in Section 11 we will formally prove that the disjunction of the constraints returned by $SAT_{\mathcal{SET}}(C)$ is equi-satisfiable with C—i.e., $SAT_{\mathcal{SET}}$ is a sound and complete solver with respect to the selected set theory $\mathcal{SET}$.

To implement the *logical derivation* [35] of a goal $:- G$ with respect to a CLP($\mathcal{SET}$) program P, it is sufficient to choose one of the C' constraints returned by $SAT_{\mathcal{SET}}$, and consider the substitution induced by $(C'_{=})|_{vars(G)}$. If all the C' constraints are either false or error then the derivation can only fail.

10 Programming examples

CLP($\mathcal{SET}$) is not intended for a specific application domain. Indeed, it has been designed as a *general-purpose* language with sets, particularly well-suited for a rapid software prototyping approach to program specification and development.

Nevertheless, there seem to be certain application areas in which the use of sets fits more naturally, thus allowing some definite improvements both in the quality and in the development time of the final software product. These areas include database applications (see for instance [41]), combinatorial problems, graph-related applications and operational research in general (e.g., resource allocation problems), as pointed out for instance in [31, 42, 14].

In particular, set unification can be conveniently exploited in those problems—e.g., resource allocation problems—whose solution can be naturally expressed by a *generate & test* approach. Set unification allows one to non-deterministically generate all possible combinations of values for the given problem, and set constraints will select only those combinations which are satisfactory.

The use of powerful set data abstractions and set constraints encourages a more *declarative* programming style, and it allows simpler and more readable programs to be obtained. In this section we present a number of simple programming examples which are intended to give the flavor of the set-oriented programming style supported by the set facilities provided in CLP($\mathcal{SET}$).

10.1 Restricted Universal Quantifiers and Intensional Sets

A common way of dealing with sets is proving that a given property holds for all the elements of the set. This fact can be easily expressed through the use of *Restricted Universal Quantifiers* (*RUQs*), i.e., formulae of the form

$$\mathsf{forall}(X \in s, \varphi[X])$$

where s denotes a set and φ is an arbitrary formula containing X. φ represents the property that all elements of s are required to satisfy. A RUQ actually represents the quantified implication

$$\forall X (X \in s \rightarrow \varphi).$$

RUQs can be easily implemented in CLP($\mathcal{SET}$) using recursion and constraints over sets. For example, the following RUQ

$$\mathsf{forall}(X \in s, p(X, Y))$$

where p is some (binary) predicate defined elsewhere in the program, can be always replaced by the equivalent atom $\mathtt{forall_p}(s, Y)$ where $\mathtt{forall_p}$ is defined by the following CLP($\mathcal{SET}$) clauses:

$$\mathtt{forall_p}(\emptyset, Y).$$
$$\mathtt{forall_p}(\{A \mid R\}, Y) \text{ :- } A \notin R, p(A, Y), \mathtt{forall_p}(R, Y).$$

This simple form of RUQ can be generalized to the more complex form

$$\mathsf{forall}(t \in s, \exists \bar{Z} \varphi[\bar{Y}, \bar{Z}])$$

where t is a term and $\bar{Y} = vars(t)$, which can also be directly implemented in CLP($\mathcal{SET}$). The logical meaning of this general form of RUQs is

$$\forall X (X \in s \rightarrow \exists \bar{Z} \bar{Y} (X = t \wedge \varphi)).$$

Another very common usage of sets is represented by the *intensional definition* of sets. Intensional sets are defined by providing a condition φ that is necessary and sufficient for an element X to belong to the set itself:

$$\{\, X \,:\, \varphi[X] \,\}$$

where X is a variable and φ is a first-order formula containing X. The logical meaning of the intensional definition of a set S is

$$\forall X(X \in S \leftrightarrow \varphi[X])$$

which can be written as

$$\forall X(X \in S \rightarrow \varphi[X]) \wedge \neg\exists X(X \notin S \wedge \varphi[X]). \tag{7}$$

Similarly to the case of RUQs, also these formulae can be implemented using CLP($\mathcal{SET}$) clauses. For example the following atom

$$S = \{\, X \,:\, p(X,Y) \,\}$$

occurring in a CLP($\mathcal{SET}$) goal can be always replaced by the equivalent atom $\mathtt{setof_p}(S,Y)$ where $\mathtt{setof_p}$ is defined by the following clauses (directly derived from the formula (7)):

$$\mathtt{setof_p(S,Y)} \text{ :- } \mathtt{forall(X \in S,\ p(X,Y)), \neg partial_p(S,Y)}.$$
$$\mathtt{partial_p(S,Y)} \text{ :- } \mathtt{Z \notin S, p(S,Y)}.$$

In this case, however, the effectiveness of the implementation depends on the expressive power of the rules used to handle *negation*. For instance, when there are no free variables occurring in the formula φ, then the CLP($\mathcal{SET}$) implementation works correctly, relying only on the traditional Negation as Failure approach to handle negative literals. However, problems can arise in connection with the use of negation in the general case. An in-depth analysis of the problems connected with intensional sets and negation (in particular *constructive negation*) can be found in [27, 10].

Like RUQs, also intensional set formers are easily generalized to the more complex form

$$\{\, t \,:\, \exists \bar{Z}\varphi[\bar{Y},\bar{Z}] \,\}$$

where t is a term and $\bar{Y} = vars(t)$, whose logical meaning is

$$\forall X(X \in S \leftrightarrow \exists \bar{Z}\bar{Y}(X = t \wedge \varphi)).$$

RUQs and intensional sets can be viewed as simple *syntactic extensions* of the CLP($\mathcal{SET}$) language. Indeed, the current CLP($\mathcal{SET}$) implementation supports these extensions, providing suitable syntactic forms and straightforward translations of them into the corresponding CLP($\mathcal{SET}$) clauses and goals. The translation is performed at compile-time and completely removes RUQs and intensional sets from the CLP($\mathcal{SET}$) code which is actually executed.

Hereafter we will assume our language is endowed with such syntactic extensions. The following example shows a few simple CLP($\mathcal{SET}$) programs using RUQs and intensional sets.

Example 10.1 *(Simple CLP($\mathcal{SET}$) programs)*

- `min(S,X)`*: true if X is the minimum of the set of numbers S.*

$$\mathtt{min(S,X)} \text{ :-}$$
$$\quad \mathtt{X \in S, forall(Z \in S, X \leq Z)}.$$

- `cross_product(A,B,CP)`*: true if* `CP` *is the Cartesian product of the sets* `A` *and* `B`*.*

$$\mathtt{cross_product(A,B,CP)} \text{ :-}$$
$$\quad \mathtt{CP = \{[X,Y] : X \in A, Y \in B\}}.$$

- `power_set(S,PS)`*: true if* `PS` *is the powerset of the set* `S`*.*

$$\mathtt{power_set(S,PS)} \text{ :-}$$
$$\quad \mathtt{PS = \{SS : subset(SS,S)\}}.$$

where the predicate `subset(S,R)` *can be simply defined as:*

$$\mathtt{subset(S,R)} \text{ :- } \cup_3 \mathtt{(S,R,R)}.$$

10.2 Graph Applications

Sets can be naturally employed to describe graphs and algorithms over graphs. A *directed labeled graph* can be represented by the set N of nodes and a finite set $E = \{(\mu_1, \nu_1), (\mu_2, \nu_2), \ldots\}$, $\mu_i, \nu_i \in N$ of directed edges. Similarly, an *undirected labeled graph* can be represented by the set N of nodes and a finite set of edges $E = \{\{\mu_1, \nu_1\}, \{\mu_2, \nu_2\}, \ldots\}$, $\mu, \nu \in N$ (i.e., a set of nested sets).

In this section we show three "classical" graph applications written in CLP($\mathcal{SET}$) using the various set representation and manipulation facilities provided by the language.

Stable Partition

In the first example we consider the notion of *stable partition* of the nodes of a graph G. This property has been defined in [52] to develop an algorithm for finding the coarsest partition induced by a binary relation on a set N. A partition P of a set N is *stable* with respect to a set of nodes $S \subseteq N$ if, for any class B of P, one of the following two conditions holds:

- $B \subseteq pred(S)$, or
- $B \cap pred(S) = \emptyset$

where $pred(S)$ is the set of predecessors of S in the graph.

In order to implement this property we need to define first a predicate that allows us to compute the successors of a given set of nodes. In this setting we can define the predicate `successor` that, given a graph G, determines whether the node Y is a successor of the node X in the graph. The same predicate can be used to verify X is an predecessor of Y.

```
successor(Nodes, Edges , X , Y ) :-
   ( X , Y ) ∈ Edges.
```

Then, using `successor`, we can easily define a predicate `successors(Nodes,Edges,B,Suc)` that is true if `Suc` is the set of all successors of a given set of nodes `B`:

```
successors( Nodes, Edges , B , Suc ) :-
            Suc = { Y : ∃ X ( X ∈ B, successor( Nodes, Edges , X , Y )) } ).
```

These predicates allows us to give the following implementation of the property of a partition P to be *stable* (resp., *unstable*):

```
stablewrt( Nodes, Edges , P , S ) :-
   successors( Nodes, Edges , E , S ),
   forall(B ∈ P, B || E or ∪3( B , E , E )).
unstablewrt( Nodes, Edges , P , S ) :-
   successors( Nodes, Edges , E , S ),
   B ∈ P,
   B ∦ E,
   ∪̸3( B , E , E ).
```

where `or` can be simply implemented using a new predicate defined by two clauses. A partition P is said to be *stable* if for all classes $S \in P$, P is stable with respect to S:

```
stable( Nodes, Edges , P ) :-
   forall(S ∈ P, stablewrt( Nodes, Edges , P , S )).
```

Traveling salesman problem

Another interesting example of the expressive power of programming with sets is given by the encoding of the *traveling salesman problem (TSP)*. Let $G = \langle N, E \rangle$ be a directed graph with weighted edges. We assume that E in G is represented as a set of triples of the form $\langle n_1, c, n_2 \rangle$ with $n_1, n_2 \in N$, and c a cost. The considered problem is to determine whether there is a path in G starting from a source node, passing exactly once for every other node, and returning in the initial node, of global cost less than a constant k.

The problem for a graph $\langle$`Nodes,Edges`$\rangle$ and source node `Source` can be encoded in CLP($\mathcal{SET}$) as follows:

```
tsp(Nodes,Edges,Source,K) :-
   \3(Nodes,{Source},To_visit),
   path(To_visit,Edges,Source,Target,Cost1),
   ⟨ Target,Cost2,Source ⟩ ∈ Edges,
   Cost1 + Cost2 < K.

path({T},Edges,S,T,Cost) :-
   ⟨ S,Cost,T ⟩ ∈ Edges.
path( { I | To_visit },Edges,S,T,Cost) :-
   ⟨ S,Cost1,I ⟩ ∈ Edges,
   I ∉ To_visit,
   path(To_visit,Edges,I,T,Cost2),
   Cost = Cost1 + Cost2.
```

Observe that the predicate `path` is able to find all possible acyclic paths starting from `Source` and involving all nodes. It is easy to provide as output the path computed, when it exists—it is sufficient to collect in a list the variables `I` of `path`.

Map coloring

As the third example, we consider the well-known problem of coloring a geographical map. Given a map of n regions $r_1,\ldots,r_n$, and a set $\{c_1, \ldots, c_m\}$ of colors, the map coloring problem consists of finding an assignment of colors to the regions of the map such that no two neighboring regions have the same color.

A map can be represented as an undirected graph where the nodes are the regions and the arcs are the pairs of neighboring regions. An *assignment* of colors to regions is represented as a set of ordered pairs (r, c) where r is a region and c is the color assigned to it.

The following is a CLP($\mathcal{SET}$) program for a generate & test solution of the map coloring problem.

```
coloring(Regions,Map,Colors,Assignments) :-
    assign(Regions,Colors,Assignments),
    forall({R1,R2} ∈ Map,∃ C ((R1,C) ∈ Assignments,(R2,C) ∉ Assignments).
```

The first subgoal of `coloring` is used to generate a possible assignment of colors to regions, whereas the second subgoal tests whether this assignment is an admissible one or not, i.e., no two adjacent regions in the map have the same color.

The predicate `assign(S1,S2,A)` is true if `A` is an assignment that assigns an element of the set `S2` (selected non-deterministically) to each element of the set `S1`. Its CLP($\mathcal{SET}$) definition is shown below.

```
assign(∅,_,∅).
assign({R | Regions},Colors,{(R,C) | Assignments}) :-
     R ∉ Regions,
     C ∈ Colors,
     assign(Regions,Colors,Assignments).
```

Remark 10.2 *The set of nodes of a graph can be easily computed from the set of edges of the graph provided the graph is completely connected. For example, the set of regions in the* `coloring` *predicate can be obtained as*

$$\texttt{Regions = \{R : } \exists \texttt{ S (\{R,S\} } \in \texttt{ Map or \{S,R\} } \in \texttt{ Map)\}}$$

or, equivalently, as

$$\texttt{Regions = \{R : } \exists \texttt{ Pair (Pair } \in \texttt{ Map, R } \in \texttt{ Pair)\}.}$$

Therefore, under the completely connected hypothesis, the set of nodes of a graph is not strictly required to be passed as an argument to predicates dealing with the graph. In all the examples of this section, however, we prefer to adopt the more general solution which uses the set of nodes explicitly.

A sample goal for the `coloring` predicate is:

$$\begin{array}{ll} :- & \texttt{coloring(\{r1,r2,r3\}, \{\{r1,r2\},\{r1,r3\}\}, \{c1,c2\},R).} \\ (i) & \texttt{R} = \{(\texttt{r1},\texttt{c1}),(\texttt{r2},\texttt{c2}),(\texttt{r3},\texttt{c2})\} \\ (ii) & \texttt{R} = \{(\texttt{r1},\texttt{c2}),(\texttt{r2},\texttt{c1}),(\texttt{r3},\texttt{c1})\}. \end{array}$$

Thanks to the general and flexible use of sets supported by CLP($\mathcal{SET}$) the same program can be used to solve related problems. For example, a different problem could be: given a map and a partially specified set of colors find which constraints the unknown colors must obey to in order to obtain an admissible coloring of the graph. A sample goal for this problem is:

$$\begin{array}{ll} :- & \texttt{coloring(\{r1,r2,r3\}, \{\{r1,r2\},\{r1,r3\}\}, \{X,c2\},R).} \\ (i) & \texttt{R} = \{(\texttt{r1},\texttt{X}),(\texttt{r2},\texttt{c2}),(\texttt{r3},\texttt{c2})\}, \texttt{X} \neq \texttt{c2} \\ (ii) & \texttt{R} = \{(\texttt{r1},\texttt{c2}),(\texttt{r2},\texttt{X}),(\texttt{r3},\texttt{X})\}, \texttt{X} \neq \texttt{c2}. \end{array}$$

The same program could be used unaltered to solve also a more complex problem: finding the minimum number of colors which is required to color a map of n regions. In fact, the number N of colors used to obtain an admissible coloring of a map can be easily computed by calling the predicate `coloring` with the set `Colors` containing exactly n variables and then calling the predicate `size`—which is part of a collection of basic operations implemented in CLP($\mathcal{SET}$) and provided with the CLP($\mathcal{SET}$) interpreter—to compute the cardinality N of the set `Colors`. By taking the set of all such N and computing its minimum value (see the predicate `min` in the previous subsection) finally we get the desired result.

As mentioned in Section 1, the ability to deal with partially specified sets is a distinguishing feature of our language which—at our knowledge—is not present in other related works.

An alternative approach to solve the coloring problem—as well as other related graph management problems—assumes that the nodes of the graph (viz., the regions in the coloring problem) are represented themselves by unbound variables, and an assignment of values (viz., colors) to the nodes is represented by the assignments of values to the variables representing the different nodes. As a consequence, the arcs will be sets consisting of two variable elements.

The new definition of the predicate `coloring` is:

```
coloring(Regions,Map,Colors) :-
   subset(Regions,Colors),
   forall(C ∈ Regions, {C} ∉ Map).
```

The first subgoal is used to generate a possible assignment of colors to regions, whereas the second subgoal tests whether this assignment is an admissible one by requiring that no set of variables $\{R1, R2\}$ in the map has got the same color c for both its variables (thus reducing $\{R1, R2\}$ to the singleton set $\{c\}$). Note that the test that the generated assignment is an admissible one can be implemented in various alternative ways. For instance, an alternative definition of this test is

$$\texttt{\{\{C\} : C } \in \texttt{ Colors\}} \,||\, \texttt{Map}$$

A sample goal for the new definition of the `coloring` predicate is:

$$\begin{array}{ll} :- & \texttt{coloring(\{R1,R2,R3\}, \{\{R1,R2\},\{R1,R3\}\}, \{c1,c2\}).} \\ (i) & \texttt{R1} = \texttt{c1}, \texttt{R2} = \texttt{c2}, \texttt{R3} = \texttt{c2} \\ (ii) & \texttt{R1} = \texttt{c2}, \texttt{R2} = \texttt{c1}, \texttt{R3} = \texttt{c1}. \end{array}$$

The alternative approach of using variables to represent nodes allows a more coincise solution. However, the use of variables to represent the nodes of the graph (instead of ground constant names), as well as the fact that the `Regions` variable set becomes a (ground) set of colors after execution, may turn out to be unnatural. Thus, the negative counterpart of enhanced coinciseness of this solution may be a decreasing of program readability.

11 Formal results

In this section we provide an axiomatic first-order characterization of the hybrid set theory we are dealing with. This axiomatization is proved to *correspond* to the interpretation structure $\mathcal{SET}$. We provide the soundness and completeness theorems associated with the rewriting procedures and, finally, we briefly discuss the computational complexity of the problem and of our solutions. Complete proofs of all the results presented in this section are given in the Appendix.

11.1 An axiomatic characterization

In this section we present a first-order naive set theory which captures the semantics of the constraints of CLP($\mathcal{SET}$). We prove that this theory corresponds with the structure $\mathcal{SET}$ on the class of admissible constraints. Equational theories are not sufficiently expressive to model the membership predicate symbol $\in$, or to force the interpretation of negative properties. Thus, we propose a richer first-order theory, coherent with the desired equational properties.

Different proposals for axiomatic semantics of terms denoting sets, suitable for CLP languages, have been recently presented [22, 29, 20, 25]. In this paper we mix the approach of [25]—which is well-suited for a parametric approach to the design of CLP languages with different kinds of aggregates (namely, sets, multi-sets, lists, and compact-lists)—with that of [22].

The set theory, named *Set*, is presented in Figure 14. *Set* provides the formal semantics for the predicate symbols $=, \in, \cup_3, ||$, and set over *hereditarily finite hybrid sets*, that is sets whose elements are uninterpreted Herbrand terms as well as other (finite) hybrid sets. Observe that, since only well-formed literals can be built (and accepted) starting from these predicate symbols, the axioms only state properties over well-formed literals. This also means that, if one were allowed to write ill-formed literals, then there would exist models of *Set* in which, for instance, $a \in b$ or $a \in \{a \,|\, b\}$ hold, and other models in which they are false.

The following is an informal justification of the axioms in the *Set* theory.

- Membership is described by axioms (N) and (W).
- Equality between sets is regulated by standard extensionality axiom (E). In [25] the axiom (E) is replaced by a different axiom (called (E^s_k)). The introduction of (E^s_k) was motivated by the presence of colored sets, which are not allowed in CLP($\mathcal{SET}$). Nevertheless, [25] proves that (E^s_k), $(Ab)(C\ell)$, as well as (E) extended with kernels, are equivalent criteria for testing set equality.

  Equality between non-sets is regulated by standard equality axioms, and by the Clark's Equality Theory [17, 47] axiom schemes $(F'_1), (F_2)$ for the non-set terms, along with a weak form of the foundation axiom (F^s_3). (F^s_3) can be proved [25] to be strong enough to avoid finite cycles of membership of the form $x_1 \in x_2, x_2 \in x_3, \ldots, x_n \in x_1$. For instance, if $x_1 \in x_2 \wedge x_2 \in x_1$, then $x_2 = \{x_1 \,|\, N\}, x_1 = \{x_2 \,|\, N'\}$, thus $x_1 = \{\{x_1 \,|\, N\} \,|\, N'\}$. (F^s_3) guarantees that this can not happen.
- The semantics of union is modeled by the standard axiom $(\cup)$.
- The disjointness of two sets is governed by axiom $(||)$ which states that two sets are disjoint when there are no common elements.
- Axioms $(S_0), (S_1)$, and axiom scheme (S_2) model the sort constraints set in first-order logic. The set literals are in fact fundamental to guarantee the correctness of the rewriting rules presented. For instance, the literal $X \notin X$ is ill-formed if X is not a set. On the contrary, if X is a set then

it is always satisfiable . Thus, it can be safely removed from the constraint only if the constraint $\mathsf{set}(X)$ has been previously asserted.

for all m, n and for all $x, x_, y, y_, v, w, z$ v, w, z are variables of sort Set		
(N)	$x \notin \emptyset$	
(W)	$x \in \{y \mid v\} \leftrightarrow x \in v \vee x = y$	
(F_1')	$f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \rightarrow x_1 = y_1 \wedge \cdots \wedge x_n = y_n$	$f \in \Sigma, f \not\equiv \{\cdot \mid \cdot\}$
(F_2)	$f(x_1, \ldots, x_m) \neq g(y_1, \ldots, y_n)$	$f, g \in \Sigma, f \not\equiv g$
(F_3^s)	$x \neq t[x]$ Unless t is of the form $\{t_1, \ldots, t_n \mid x\}$ and x does not occur in $t_1, \ldots, t_n$	
(E)	$v = w \rightarrow \forall x\, (x \in v \leftrightarrow x \in w)$	
$(\cup)$	$\cup_3(v, w, z) \quad \leftrightarrow \quad \forall x (x \in z \leftrightarrow (x \in v \vee x \in w))$	
$(\|\|)$	$v \| \| w \leftrightarrow \forall x\, (x \in v \rightarrow x \notin w)$	
(S_0)	$\mathsf{set}(\emptyset)$	
(S_1)	$\mathsf{set}(v) \leftrightarrow \mathsf{set}(\{y \mid v\})$	
(S_2)	$\neg\mathsf{set}(f(t_1, \ldots, t_n))$	$f \in \Sigma, f \not\equiv \{\cdot \mid \cdot\}, f \not\equiv \emptyset$

Figure 14: The theory *Set*

11.2 Correspondence between structure and theory

Given a first-order theory T on $\mathcal{L}$ and a structure $\mathcal{A}$ which is a model of T, T and $\mathcal{A}$ *correspond* on the set of admissible constraints *Adm* [35] if, for each constraint $C \in Adm$, we have that $T \models \vec{\exists}(C)$ if and only if $\mathcal{A} \models \vec{\exists}(C)$. If $\mathcal{A}$ is a model of T then the $(\Rightarrow)$ direction is always fulfilled. This property guarantees that $\mathcal{A}$ is a special model: if we know that C is satisfiable in $\mathcal{A}$ then it will be satisfiable in all the models of T. If *Adm* is the set of all the first-order formulae, then this concept reduces to Robinson's *model-completeness* [16].

The interpretation structure $\mathcal{SET}$ defined in Section 8.2 can be proved to be a model of the simple theory of sets *Set* considered above. In particular:

Theorem 11.1 (Correspondence) *The axiomatic theory Set and the structure $\mathcal{SET}$ correspond [35] on the class of admissible constraints of CLP($\mathcal{SET}$).*

An important property for CLP structures is *solution compactness*: a constraint domain $\mathcal{A} = \langle A, (\cdot)^{\mathcal{A}} \rangle$ is solution compact on *Adm* [35] if:

1. for all $d \in A$ there is a possibly infinite conjunction of constraints $\varphi(X) \equiv \bigwedge_i C_i$ such that the unique solution of $\varphi(X)$ is $X = d$;
2. for each admissible constraint C there exists a (possibly infinite) collection $\{C_1, C_2, \ldots\}$ of admissible constraints such that

$$\mathcal{A} \models \forall \bar{X} \left(\neg C[\bar{X}] \leftrightarrow \bigvee_i \exists \bar{Y}\, C_i[\bar{X}, \bar{Y}] \right)$$

Actually, the variables $\bar{Y}$ in condition 2 are not made explicit in [35]. However, what we need is the ability to effectively negate a constraint while preserving satisfiability and the same valuations for the common variables. New variables do not affect this property.

Proposition 11.2 (Solution Compactness) *$\mathcal{SET}$ is solution compact on the class of admissible constraints of CLP($\mathcal{SET}$).*

PROOF. Property (1) trivially holds since each element d of the domain of $\mathcal{S}$ is also a term. Thus, we can write the admissible constraint $X = d$. As far as property (2) is concerned, let $C = \ell_1 \wedge \cdots \wedge \ell_n$ be an admissible constraint, and let $vars(C) = \{\bar{X}\}$. Clearly, $\neg C$ is equivalent to $\neg\ell_1 \vee \cdots \vee \neg\ell_n$. Now, for $i = 1$ to n:

- If ℓ_i is a primitive constraint or the negation of a primitive constraint built on one of the predicate symbols $=, \in, \cup_3, ||$, then $\neg\ell_i$ is an admissible constraint, as well.
- If ℓ_i is $\mathsf{set}(s)$ for some term s, consider the set $\mathcal{F} = \{\emptyset, \{\cdot \mid \cdot\}, f_1, f_2, f_3, \ldots\}$. It holds that

$$\neg\mathsf{set}(s) \leftrightarrow \bigvee_{i>0} \exists Y_1 \cdots Y_{ar(f_i)}\, s = f_i(Y_1, \ldots, Y_{ar(f_i)})$$

□

Property (2) becomes more effective when Σ is finite.

11.3 Constraint Satisfiability

The notion of solved form plays a fundamental role in the definition of the constraint satisfiability procedure, as ensues from Theorem 11.4. Basically, given a constraint in solved form C, we are able to guarantee the existence of a successful valuation for all the variables of C using pure sets only. In particular, it is sufficient to restrict the search for possible solutions to sets of the form:

$$\underbrace{\{\cdots\{}_{n}\emptyset\}\cdots\}$$

for all the variables, with the only exception of the variables X occurring as $X = t$ in C.

Example 11.3 *Consider the constraint in solved form:*

$$X = f(Y), Y \neq \emptyset, Y \neq \{Z\}, Y||Z, \cup_3(Z, W, R)$$

It admits the solution:

$$Z = W = R = \emptyset, Y = \{\{\emptyset\}\}, X = f(\{\{\emptyset\}\})$$

Theorem 11.4 (Satisfiability) *Let C be a constraint in solved form. Then C is satisfiable in $\mathcal{SET}$.*

The following theorem proves the correctness and completeness of the non-deterministic rewriting procedure $SAT_{\mathcal{SET}}$. Complete proof of this result is provided in the Appendix.

Theorem 11.5 (Correctness and Completeness) *Let C be a constraint and $C_1, \ldots, C_n$ be the constraints obtained from the application of* set_infer *and by each successive non-deterministic computation of* STEP. *Then,*

1. *if σ is a valuation of C_i and $\mathcal{SET} \models \sigma(C_i)$ then $\mathcal{SET} \models \sigma(C)$ for all $i = 1, \ldots, n$,*
2. *if σ is a valuation of C and $\mathcal{SET} \models \sigma(C)$ then, for all $i = 1, \ldots, n$, σ can be expanded to the variables of $vars(C_i) \setminus vars(C)$ so that it fulfills $\mathcal{SET} \models \sigma(C_i)$.*

Corollary 11.6 *Given a constraint C, the following result holds: Set $\models \vec{\exists} C$ if and only if there is a non-deterministic choice such that $SAT_{\mathcal{SET}}(C)$ returns a constraint in solved form (i.e., different from* false *and* error*).*

Another important property of CLP theories is *satisfaction completeness*. A theory T is *satisfaction complete* [35] if for each admissible constraint C either $T \models \vec{\exists} C$ or $T \models \neg\vec{\exists} C$. Observe that if the class *Adm* is the set of all first-order formulae, then this notion reduces to the usual notion of *completeness* of a theory [16].

Corollary 11.7 *Set is satisfaction complete on the class of admissible constraints of CLP($\mathcal{SET}$). Moreover, for each admissible constraint c, the test Set $\models \vec{\exists} c$ is decidable.*

PROOF. Given an admissible constraint, Theorem 11.1 ensures that $Set \models \vec{\exists} C$ if and only if $\mathcal{SET} \models \vec{\exists} C$. Thus, it is sufficient to prove the property: for each admissible constraint c either $\mathcal{SET} \models \vec{\exists} c$ or $\mathcal{SET} \models \neg\vec{\exists} c$. This, together with the decidability property, is an immediate consequence of termination Theorem 9.10 and of Corollary 11.6. □

11.4 Complexity

The problem we tackle here extends the satisfiability problem for set unification, shown to be NP-complete in [40]. A different (and maybe simpler) proof of NP-hardness of the same problem has been given in [22]. In [51], a methodology to guess a solution of a conjunction of literals where $\mathcal{F} = \{\emptyset, \{\cdot \mid \cdot\}\}$ and $\Pi_c = \{=, \in, \cup, \cap, \setminus\}$ is proposed. Any *guess* is represented by a graph containing a number of nodes polynomially bounded by the number of variables in the original problem. Verification of whether a guess is a solution of the constraint can be done in polynomial time. In [51], it is also shown how this technique can be extended to the hybrid problem—the one we deal with in this paper.

The algorithms we propose here clearly do not belong to NP since they apply syntactic substitutions. However, one could devise implementations of the algorithms adopting standard techniques, such as those in [48], to avoid problems originated by substitutions, in order to achieve better complexity results (that, however, can not be any better than NP).

12 Related Work

In the context of CLP-based solutions, the only other schemes which embed the notion of set are CLP(Σ^*) [65], Conjunto [31], and CLPS [42].

The CLP(Σ^*) proposal considers substantially different classes of sets and it is mostly aimed at handling regular languages over a given alphabet.

In [31], Gervet presents a language, called *Conjunto*, which incorporates a constraint solver over boolean lattices built from (flat) set intervals. The constraints can be more complex (e.g., boolean constraints) than those considered in CLP($\mathcal{SET}$), but the domain is less general. In particular, the simulation of nested sets is not possible—which prevents the direct encoding of many interesting problems. Conjunto has been embedded in the recent releases of the ECL^{i}PS^e system.

A solution based on a CLP-scheme which, on the contrary, shares much with our work is the one described in CLPS [42]. The main difference with our proposal is that no precise definition of the set theory nor of the interpretation domain is given in [42]. This prevents that proposal from providing formal correctness and completeness results. An efficient implementation of the language in [42] has been developed.

Other logic-based languages with sets (which do not fall in the CLP class) have also been proposed, such as $\mathcal{LDL}$ [9], LPS [41], and SEL [39]; the interested reader is referred to [22] for a brief overview of these languages.

Considerable research effort has also been devoted to the analysis and definition of *set-constraints*, originally introduced in the field of Program Analysis [32]. Set-constraints are formulae built with first-order terms and set operators. The original intended use of these formulae is to interpret the solution of a set-constraint as the approximation of the set of possible outputs of a program. The techniques developed for testing satisfiability of set-constraints are very interesting (mostly based on the use of *tree-automata*), although their effective use is made difficult by intrinsic complexity limits (NEXP-time

completeness [61]). However, these results can not be used directly for the kind of constraints over sets we consider in this paper. As a matter of fact, sets involved in set-constraints are flat (subsets of the standard Herbrand Universe) and therefore a real implementation of nested membership is not allowed. Moreover, the interpretation of the function symbols involves cartesian products and projections, which in turn implies that the satisfiability can not be checked over hereditarily *finite* sets.

13 Conclusions

In this paper we have compared existing proposals for handling finite sets in CLP languages, and proposed a novel technique, that captures the benefits of the existing ones and generalizes the existing literature. The new representation scheme uses $\{\cdot \mid \cdot\}$ as set-constructor and $\in, =, \cup_3$, and $||$ as primitive constraint predicates. The use of the $\cup_3$ and $||$ predicate as constraints allows us to obtain effective definitions of various set operations, such as $\subseteq, \cap$, and $\setminus$.

We have provided a formal description of the syntax, logical semantics, and operational semantics of a CLP language relying on these constraints, called CLP($\mathcal{SET}$). The operational semantics have been described through a number of constraint simplification procedures, capable of transforming arbitrary conjunctions of constraints to trivially decidable solved forms. Correctness and termination proofs for this operational semantics have also been presented. The correspondence between structure and theory allows us to guarantee that the satisfiability test holds when infinite sets are allowed in the domain, as well.

Few remarks are due as far as intensionally defined sets are concerned. As shown in [10], intensional set definitions are implementable in the language extended with negation. The rules for handling negation in (constraint) logic programming (e.g., negation as failure) typically introduce restrictions on the set of admissible programs. These restrictions can be weakened using a more general negation rule, such as Constructive Negation [64] instead of Negation as Failure. Nevertheless, the undecidability of any set theory prevents one to solve all the possible problems opened [26]. Alternatively, one can face directly intensional sets, for instance developing a suitable algebra over them: for example, $a \in \{X : nat(X)\}$ can be rewritten simply as $nat(a)$; $\{X : nat(X)\} \cap \{X : odd(X)\}$ can be rewritten as $\{X : nat(X), odd(X)\}$, and so on. Preliminary work on this direction is described in [13].

Although we have considered only "conventional" sets, there are a number of other data aggregate abstractions, such as multisets and lists, which may turn out to fit more naturally the problem requirements than sets in many interesting applications. The solutions and techniques described in this paper are general and flexible enough to be quite easily adapted to these other data structures. A work in this direction is [25], where a formal characterization of various kinds of data aggregates and the definition of suitable (parametric) unification algorithms dealing with them in a logic programming framework are presented.

The interpreter of the language CLP($\mathcal{SET}$), written in SICStus Prolog, is available at

http://www.math.unipr.it/~gianfr/setlog.Home.html.

Work is in progress to improve the existing implementation, in particular by providing better user interfaces and static analysis tools to capture and efficiently handle special cases (e.g., distinguishing between ground and non-ground cases, delay techniques).

References

[1] Abiteboul, S., and Grumbach., S. A rule-based language with functions and sets. *ACM Trans. on Database Systems 16*, 1 (1991), 1–30.

[2] Abrial, J.-R. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[3] Aczel., P. *Non-well-founded sets.*, vol. 14 of *Lecture Notes, Center for the Study of Language and Information.* Stanford, 1988.

[4] Aliffi, D., Dovier, A., and Rossi, G. From Set to Hyperset Unification. *Journal of Functional and Logic Programming 1999*, 11 (1999), 1–48.

[5] Apt, K., and Bol, R. Logic Programming and Negation: A Survey. *Journal of Logic Programming 19/20* (1994).

[6] Arenas-Sánchez, P., and Dovier, A. A minimality study for set unification. *Journal of Functional and Logic Programming 1997*, 7 (December 1997), 1–49.

[7] Baader, F., and Büttner, W. Unification in commutative and idempotent monoids. *Theoretical Computer Science 56* (1988), 345–352.

[8] Baader, F., and Schulz, K. U. Unification in the union of disjoint equational theories: Combining decision procedures. *Journal of Symbolic Computation 21* (1996), 211–243.

[9] Beeri, C., Naqvi, S., Shmueli, O., and Tsur., S. Set Constructors in a Logic Database Language. *Journal of Logic Programming 10*, 3 (1991), 181–232.

[10] Bruscoli, P., Dovier, A., Pontelli, E., and Rossi., G. Compiling Intensional Sets in CLP. In *Proc. Eleventh International Conference on Logic Programming* (June 1994), P. Van Entenryck, Ed., The MIT Press, Cambridge, Mass., pp. 647–661. S. Margherita Ligure, Italy.

[11] Büttner, W. Unification in the Data Structure Sets. In *Proc. of the Eight International Conference on Automated Deduction* (1986), J. K. Siekmann, Ed., vol. 230, Springer-Verlag, Berlin, pp. 470–488.

[12] Cantone, D., Ferro, A., and Omodeo., E. G. *Computable Set Theory, Vol. 1.* No. 6 in International Series of Monographs on Computer Science. Clarendon Press, Oxford, 1989.

[13] Carmona, R., Dovier, A., and Rossi, G. Dealing with infinite intensional sets in clp. In *APPIA-GULP-PRODE'97. Joint Conf. on Declarative Programming* (June 1997), M. Falaschi, Ed., pp. 467–477. Grado, Italy.

[14] Caseau, Y., and Laburthe, F. Introduction to the CLAIRE programming language. Technical report, LIENS, 1996.

[15] Chan, D. Constructive Negation Based on the Completed Database. In *Proc. Fifth International Conference and Symposium on Logic Programming* (1988), R. Kowalski and K. Bowen, Eds., The MIT Press, Cambridge, Mass., pp. 111–125.

[16] Chang, C. C., and Keisler, H. J. *Model Theory.* Studies in Logic. North Holland, 1973.

[17] Clark, K. L. Negation as Failure. In *Logic and Databases*, H. Gallaire and J. Minker, Eds. Plenum Press, 1978, pp. 293–321.

[18] Cooke, D. E. An introduction to SequenceL: A language to experiment with constructs for processing nonscalars. *Software—Practice and Experience 26*, 11 (Nov. 1996), 1205–1246.

[19] Dershowitz, N., and Manna, Z. Proving Termination with Multiset Ordering. *Communication of the ACM 22*, 8 (1979), 465–476.

[20] Dovier, A. *Computable Set Theory and Logic Programming.* PhD thesis, Università degli Studi di Pisa, March 1996. TD–1/96.

[21] Dovier, A., Omodeo, E. G., Pontelli, E., and Rossi., G. {log}: A Logic Programming Language with Finite Sets. In *Proc. Eighth International Conf. on Logic Programming* (June 1991), K. Furukawa, Ed., The MIT Press, Cambridge, Mass., pp. 111–124. Paris.

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

[23] Dovier, A., Piazza, C., and Policriti, A. Comparing expressiveness of set constructor symbols. In *APPIA-GULP-PRODE'99. Joint Conf. on Declarative Programming* (September 1999), M. C. Meo, Ed., pp. 151–165. L'Aquila, Italy.

[24] Dovier, A., Piazza, C., Pontelli, E., and Rossi, G. On the Representation and Management of Finite Sets in CLP-languages. In *Proc. of 1998 Joint International Conference and Symposium on Logic Programming* (June 1998), J. Jaffar, Ed., The MIT Press, Cambridge, Mass., pp. 40–54. Manchester, UK.

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

[26] Dovier, A., Pontelli, E., and Rossi, G. A Necessary Condition for Constructive Negation in Constraint Logic Programming. Technical report, New Mexico State University, 1998.

[27] Dovier, A., Pontelli, E., and Rossi, G. Constructive negation and constraint logic programming with sets. Quaderni del Dipartimento di Matematica 183, Università di Parma, October 1998. Submitted to New Generation of Computing.

[28] Dovier, A., Pontelli, E., and Rossi, G. Set unification revisited. NMSU-CSTR-9817, Dept. of Computer Science, New Mexico State University, USA, October 1998. Submitted to Information and Computation.

[29] Dovier, A., and Rossi, G. Embedding Extensional Finite Sets in CLP. In *Proc. of International Logic Programming Symposium, ILPS'93* (October 1993), D. Miller, Ed., The MIT Press, Cambridge, Mass., pp. 540–556. Vancouver, BC, Canada.

[30] Fages, F. Associative-Commutative Unification. *Journal of Symbolic Computation 3* (1987), 257–275.

[31] Gervet, C. Interval propagation to reason about sets : Definition and implementation of a practical language. *International Journal of Constraints 1*, 1 (1997), 191–246.

[32] Heintze, N., and Jaffar, J. Set Constraints and Set-Based Analysis. Technical report, Carnegie Mellon University, 1994.

[33] Hill, P. M., and Lloyd, J. W. *The Gödel Programming Language*. The MIT Press, Cambridge, Mass., 1994.

[34] IC-PARC. Ecl^{i}ps^e, user manual. Tech. rep., Imperial College, London, August 1999. Available at http://www.icparc.ic.ac.uk/eclipse.

[35] Jaffar, J., and Maher, M. J. Constraint Logic Programming: A Survey. *Journal of Logic Programming 19–20* (1994), 503–581.

[36] Jaffar, J., Maher, M. J., Marriott, K., and Stuckey, P. J. The semantics of constraint logic programs. *Journal of Logic Programming 37*, 1–3 (1998), 1–46.

[37] Jana, D., and Jayaraman, B. Set constructors, finite sets, and logical semantics. *Journal of Logic Programming 38*, 1 (1999), 55–77.

[38] Jayaraman, B., and Moon, K. The SuRE programming framework. In *Algebraic Methodology and Software Technology, 4th International Conference, AMAST '95* (1995), V. S. Alagar and M. Nivat, Eds., vol. 936 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, p. 585. Montreal, Canada, July 3-7.

[39] Jayaraman, B., and Plaisted, D. A. Programming with Equations, Subsets and Relations. In *Proceedings of NACLP89* (1989), E. Lusk and R. Overbeek, Eds., The MIT Press, Cambridge, Mass., pp. 1051–1068. Cleveland.

[40] Kapur, D., and Narendran, P. Complexity of Unification Problems with Associative-Commutative Operators. *Journal of Automated Reasoning 9* (1992), 261–288.

[41] Kuper, G. M. Logic Programming with Sets. *Journal of Computer and System Science 41*, 1 (1990), 66–75.

[42] Legeard, B., and Legros, E. Short overview of the CLPS system. In *Proc. Third International Symposium on Programming Language Implementation and Logic Programming* (August 1991), J. Maluszynsky and M. Wirsing, Eds., vol. 528 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 431–433. Passau, Germany.

[43] Liu, M. Relationlog: a typed extension to Datalog with Sets and Tuples. In *International Logic Programming Symposium* (1995), MIT Press.

[44] Livesey, M., and Siekmann, J. Unification of Sets and Multisets. Technical report, Institut für Informatik I, Universität Karlsruhe, 1976.

[45] Lloyd, J. W. Programming in an integrated functional and logic language. *Journal of Logic Programming 1999*, 3 (1999), 1–49.

[46] Maher, M. J. Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees. In *Proceedings of 3rd Symposium Logic in Computer Science* (Edinburgh, 1988), pp. 349–357.

[47] Mal'cev, A. Axiomatizable Classes of Locally Free Algebras of Various Types. In *The Metamathematics of Algebraic Systems*, Collected Papers. North Holland, 1971, ch. 23.

[48] Martelli, A., and Montanari, U. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst. 4* (1982), 258–282.

[49] Munakata, T. Notes on implementing sets in prolog. *Communications of the ACM 35*, 3 (Mar. 1992), 112–120.

[50] Naish, L. *Negation and control in PROLOG*, vol. 238 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1986.

[51] Omodeo, E. G., and Policriti, A. Solvable set/hyperset contexts: I. some decision procedures for the pure, finite case. *Communication on Pure and Applied Mathematics 9–10* (1995), 1123–1155. A special double issue dedicated to Jack Schwartz.

[52] Paige, R., and Tarjan, R. E. Three partition refinement algorithms. *SIAM Journal on Computing 16*, 6 (Dec. 1987), 973–989.

[53] Paterson, M. S., and Wegman, M. N. Linear unification. *Journal of Computer System Science 16*, 2 (1978), 158–167.

[54] Robinson, A. *Introduction to Model Theory and to the Metamathematics of Algebra*. North Holland, 1963.

[55] Roy, P. V. Logic programming in oz with mozart. In *International Conference on Logic Programming* (1999), MIT Press, pp. 38–51.

[56] Schwartz, J. T., Dewar, R. B. K., Dubinsky, E., and Schonberg., E. *Programming with sets, an introduction to SETL*. Springer-Verlag, Berlin, 1986.

[57] Shmueli, O., Tsur, S., and Zaniolo, C. Compilation of Set Terms in the Logic Data Language (LDL). *Journal of Logic Programming 12*, 1 (1992), 89–120.

[58] Siekmann, J. H. Unification theory. In *Unification*, C. Kirchner, Ed. Academic Press, 1990.

[59] SMOLKA, G. The OZ programming model. In *Computer Science Today: Recent Trends and Developments* (1995), J. van Leeuwen, Ed., vol. 1000 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 324–343.

[60] SPIVEY., J. M. *The Z Notation: A reference Manual, 2nd edition.* International Series in Computer Science. Prentice Hall, 1992.

[61] STEFÁNSSON, K. Systems of set constraints with negative constraints are NEXPTIME-complete. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science* (Paris, France, 4–7 July 1994), IEEE Computer Society Press, pp. 137–141.

[62] STERLING, L., AND SHAPIRO, E. *The Art of Prolog.* MIT Press, 1996.

[63] STOLZENBURG, F. An algorithm for general set unification and its complexity. *Journal of Automated Reasoning 22*, 1 (1999), 45–63.

[64] STUCKEY, P. J. Negation and Constraint Logic Programming. *Information and Computation 1* (1995), 12–33.

[65] WALINKSI, C. CLP(Σ^*): Constraint Logic Programming with Regular Sets. In *Proc. 6th International Conf. on Logic Programming* (1989), G. Levi and M. Martelli, Eds., The MIT Press, Cambridge, Mass., pp. 181–196.

Appendix

Proof of Theorem 11.1 (Correspondence)

In order to prove the correspondence between the structure $\mathcal{SET}$ and the theory *Set* on the class of constraint defined in Def. 8.2, we introduce auxiliary definitions and we state and demonstrate some lemmas.

Given a first-order language $\mathcal{L}$ and two structures $\mathcal{A}$ and $\mathcal{B}$ over $\mathcal{L}$, an *embedding* of $\mathcal{A}$ in $\mathcal{B}$ is an isomorphism from $\mathcal{A}$ into a substructure of $\mathcal{B}$ [54].

Lemma 13.1 *Let $\mathcal{A}$ and $\mathcal{B}$ be two structures of a first-order language $\mathcal{L}$, and let h be an embedding of $\mathcal{A}$ in $\mathcal{B}$. If φ is an quantifier-free (open) formula of $\mathcal{L}$, then $\mathcal{A} \models \varphi[\sigma] \leftrightarrow \mathcal{B} \models \varphi[h \circ \sigma]$.*

PROOF. It can easily be proved by induction on the complexity of φ. In [54] the result is presented as a corollary of the first theorem of Chapter 2. □

Lemma 13.2 *$\mathcal{SET}$ is a model of the theory Set.*

PROOF. We prove the property by showing that each axiom of the theory *Set* is modeled by the structure $\mathcal{SET}$.

(N) The fact that $\mathcal{SET}$ models (N) is immediate from the definition of $\in^{\mathcal{S}}$.

$(S_0), (S_1), (S_2)$ Also in these cases the result trivial.

(W) If σ is a valuation over $\mathcal{SET}$ such that $\mathsf{set}(\sigma(v))^{\mathcal{S}}$ holds, then $\sigma(v) \equiv \{s_1, \ldots, s_n \,|\, \emptyset\}$ and, hence, $\sigma(x) \in^{\mathcal{S}} \{\sigma(y) \,|\, \sigma(v)\}^{\mathcal{S}}$ if and only if $\sigma(x) \equiv \sigma(y)$ or $\sigma(x) \in^{\mathcal{S}} \sigma(v)$. Hence, $\mathcal{SET}$ is a model of (W).

(F_1') Let us prove that if $f^{\mathcal{S}}(t_1, \ldots, t_n) = f^{\mathcal{S}}(s_1, \ldots, s_n)$, $f \not\equiv \{\cdot \,|\, \cdot\}$, then $t_1^{\mathcal{S}} = s_1^{\mathcal{S}}, \ldots, t_n^{\mathcal{S}} = s_n^{\mathcal{S}}$. $f^{\mathcal{S}}(t_1, \ldots, t_n) = f^{\mathcal{S}}(s_1, \ldots, s_n)$ is equivalent to $\tau(f(t_1, \ldots, t_n)) = \tau(f(s_1, \ldots, s_n))$ and this implies $\tau(t_1) = \tau(s_1) \wedge \tau(t_n) = \tau(s_n)$, which is equivalent to our thesis. So, $\mathcal{SET}$ is a model of (F_1').

$(F_2), (F_3^s)$ The proof is similar to that in the previous case.

(E) If σ satisfies $\mathsf{set}(\sigma(v))$ and $\mathsf{set}(\sigma(w))$ and $\sigma(v) = \sigma(w)$, then forall $s \in \mathcal{S}$ we have:

$$s \in^{\mathcal{S}} \sigma(v) \text{ if and only if } s \in^{\mathcal{S}} \sigma(w)$$

On the other hand, if σ satisfies $\mathsf{set}(\sigma(v))$, $\mathsf{set}(\sigma(w))$ and $\forall s \in \mathcal{S}\ s \in^{\mathcal{S}} \sigma(v)$ if and only if $s \in^{\mathcal{S}} \sigma(w)$, then we have to consider two cases. If $\sigma(v) = \emptyset$, then for forall $s \in \mathcal{S}$ we have $s \notin^{\mathcal{S}} \sigma(v)$. Thus, the same must hold for $\sigma(w)$. From this we obtain that $\sigma(w) = \emptyset$. If $\sigma(v) = \{u_1, \ldots, u_n \,|\, \emptyset\}$, then $s \in \sigma(w)$ if and only if $s \equiv u_i$, with $i \leq n$. Hence, we obtain that $\sigma(w) = \{u_1, \ldots, u_n \,|\, \emptyset\}$.

$(\cup), (||)$ The fact that $\mathcal{SET}$ models $(\cup)$ and $(||)$ immediately follows from the definitions of $\cup_3^{\mathcal{S}}$ and $||^{\mathcal{S}}$.

□

In the following proof, as well as in the proof of the termination of the algorithm, we will make use of the measure functions *size* and $|\cdot|$:

Definition 13.3 *The function size* $: T(\mathcal{F}, \mathcal{V}) \cup \mathcal{C} \longrightarrow \mathbb{N}$ *is defined as follows:*

$$size(t) = \begin{cases} 0 & \text{if } t \text{ is a variable} \\ 1 & \text{if } t \text{ is a constant} \\ 1 + \sum_{i=1}^{n} size(t_i) & \text{if } t = f(t_1, \ldots, t_n), f \in \mathcal{F} \\ \sum_{i=1}^{n} size(t_i) & \text{if } t = p(t_1, \ldots, t_n), p \in \Pi \\ size(C_1) + size(C_2) & \text{if } t = C_1 \wedge C_2 \end{cases}$$

With $|C|$ *we denote the number of literals in the constraint* C.

Lemma 13.4 *If* $\mathcal{B}$ *is a model of Set, then the function* $h : domain(\mathcal{SET}) \longrightarrow domain(\mathcal{B})$ *defined as* $h(t) = t^{\mathcal{B}}$ *for all* $t \in \mathcal{S}$*, is an embedding of* $\mathcal{SET}$ *in* $\mathcal{B}$.

PROOF. We will prove the following facts:

1. h is an homomorphism;
2. if $p^{\mathcal{B}}(h(t_1), \ldots, h(t_n))$ holds, then $p^{\mathcal{S}}(t_1, \ldots, t_n)$ holds, for all predicate symbols $p \in \{\in, \cup_3, ||, \mathsf{set}\}$;
3. h is injective.

1. For all $f \in \Sigma$, $f \not\equiv \{\cdot \,|\, \cdot\}$, and for all $t_1, \ldots, t_n \in \mathcal{S}$ it holds that

$$h(f^{\mathcal{S}}(t_1, \ldots, t_n)) = (\tau(f(t_1, \ldots, t_n))^{\mathcal{B}} = (f(t_1, \ldots, t_n))^{\mathcal{B}}$$

Moreover $(f(t_1, \ldots, t_n))^{\mathcal{B}} = f^{\mathcal{B}}(t_1^{\mathcal{B}}, \ldots, t_n^{\mathcal{B}})$, by the definition of structure, and $f^{\mathcal{B}}(t_1^{\mathcal{B}}, \ldots, t_n^{\mathcal{B}}) = f^{\mathcal{B}}(h(t_1), \ldots, h(t_n))$.

Consider now the case of the functional symbol $\{\cdot \,|\, \cdot\}$. It holds that

$$h(\{t_1 \,|\, t_2\}^{\mathcal{S}}) = h(\tau(\{t_1 \,|t_2\})) = h(\{s_1, \ldots, t_1, \ldots, s_n\}) = \{s_1, \ldots, t_1, \ldots, s_n\}^{\mathcal{B}}$$

which, since $\mathcal{B}$ is a model of (E) is equivalent to $\{t_1^{\mathcal{B}} \,|\, t_2^{\mathcal{B}}\}^{\mathcal{B}} = \{h(t_1) \,|\, h(t_2)\}^{\mathcal{B}}$.

If $s \in^{\mathcal{S}} t$, then $t \equiv \{t_1, \ldots, s, \ldots, t_n \,|\, \emptyset\}$; since $\mathcal{B}$ is a model of (W), this implies $s^{\mathcal{B}} \in^{\mathcal{B}} t^{\mathcal{B}}$. We obtain the same result for the predicates $\cup_3, ||$ and set exploiting the facts that $\mathcal{B}$ is a model of $(\cup), (||)$ and $(S_0), (S_1)$.

2. If $s^{\mathcal{B}} \in^{\mathcal{B}} t^{\mathcal{B}}$ then, since $\mathcal{B}$ is a model of (W), it must be that $\mathsf{set}^{\mathcal{B}}(t^{\mathcal{B}})$ holds. Moreover, since $\mathcal{B}$ is also a model of (N), it can not be $t \equiv \emptyset$; and, since $\mathcal{B}$ is a model of (S_2), it can not be the case that $t \equiv f(\ldots)$, for $f \not\equiv \{\cdot \,|\, \cdot\}$. Thus, it must be $t \equiv \{t_1, \ldots, t_n \,|\, \emptyset\}$. The fact that $\mathcal{B}$ is a model of (W) and this property implies that $s^{\mathcal{B}} \in^{\mathcal{B}} \{t_1, \ldots, t_n\}^{\mathcal{B}}$ if and only if $\exists i \leq n$ such that $s \equiv t_i$. This is equivalent to $s \in^{\mathcal{S}} t$. Using the axioms $(\cup_3), (||), (S_0), (S_1)$, and (S_2) we obtain in a similar way the result for the predicate symbols $\cup_3, ||$ and set.

3. If $h(t_1) = h(t_2)$, then we prove $t_1 \equiv t_2$. We can assume that $size(t_1) \geq size(t_2)$ and obtain the result by induction on $size(t_1) \geq 1$.

   Base. If $size(t_1) = 1$, then t_1 and t_2 are constants, hence, since $\mathcal{B}$ is a model of (F_2), $t_1 \equiv t_2$.

   Step. If $t_1 \equiv f(s_1, \ldots, s_n)$, then, since $\mathcal{B}$ is a model of (F_2), it must be $t_2 \equiv f(r_1, \ldots, r_n)$, moreover since $\mathcal{B}$ is a model of (F_1'), also $s_1^{\mathcal{B}} = r_1^{\mathcal{B}}, \ldots, s_n^{\mathcal{B}} = r_n^{\mathcal{B}}$ must hold. By inductive hypothesis, we obtain that $s_1 \equiv r_1, \ldots, s_n \equiv r_n$, and hence that $t_1 \equiv t_2$. If $t_1 \equiv \{s_1, \ldots, s_n\}$, then from (F_2) we obtain $t_2 \equiv \{r_1, \ldots, r_m\}$. t_1 and t_2 are elements of $\mathcal{S}$, so there are no repetitions between their elements and they are ordered. This implies that, since $\mathcal{B}$ is a model of (E), then $n = m$ and $s_i \equiv r_i$ for all $i \leq n$—i.e., $t_1 \equiv t_2$.

□

Theorem 11.1 *The axiomatic theory Set and the structure $\mathcal{SET}$ correspond on the class of admissible constraints of CLP($\mathcal{SET}$) .*

PROOF. From Lemma 13.2 we know that $\mathcal{SET}$ is a model of *Set*; thus, if C is a first-order formula and $Set \models C$, then $\mathcal{SET} \models C$.

On the other hand if C is a constraint and $\vec{\exists}C$ is its existential closure, then $\mathcal{SET} \models \vec{\exists}C$ if and only if there exists σ such that $\mathcal{SET} \models C[\sigma]$. Hence, from Lemma 13.4 and Lemma 13.1, it holds that $\mathcal{B} \models \vec{\exists}C$ for any model $\mathcal{B}$ of *Set*. This is exactly the definition of $Set \models \vec{\exists}C$. □

Proof of Theorem 11.4 (Satisfiability)

The notion of solved form plays a fundamental role in the definition of the constraint satisfiability procedure, as ensues from the following theorem. To prove it we use the auxiliary function *find*:

$$find(t) \quad = \quad \begin{cases} \emptyset & \text{if } t = \emptyset,\ x \neq \emptyset \\ \{0\} & \text{if } t = x \\ \{1 + n : n \in find(x, y)\} & \text{if } t = \{y \mid \emptyset\} \\ \{1 + n : n \in find(x, y)\} \cup find(x, s) & \text{if } t = \{y \mid s\},\ s \neq \emptyset \end{cases}$$

which returns the set of 'depths' in which a given element x occurs in the set t (there is an exception for the unique case $find(\emptyset, \emptyset)$).

Theorem 11.4 *Let C be a constraint in solved form. Then C is satisfiable in $\mathcal{SET}$.*

PROOF. The proof is basically the construction of a mapping for the variables of C into $\mathcal{S}$. The construction is divided into two parts. In the first part, $C_=$ is not considered. A solution for the other constraints is computed by looking for valuations of the form

$$X_i \mapsto \underbrace{\{\cdots\{}_{n_i}\emptyset\}\cdots\}$$

fulfilling all $\neq, ||, \notin, \cup_3$, and set constraints. We will briefly refer to $\overbrace{\{\cdots\{}^{n_i}\emptyset\}\cdots\}$ as $\{\emptyset\}^{n_i}$. In particular, the variables appearing in $\cup_3$ are mapped into $\emptyset$ ($n_i = 0$) and the numbers n_i for the other variables are computed choosing one possible solution of a system of integer equations and disequations, that trivially admits solutions. Such system is obtained by analyzing the "depth" of the occurrences of the variables in the terms; in this case, $||$ and $\neq$ constraints are treated in the same way. Then, all the variables occurring in the constraint C only in r.h.s. of equations of $C_=$ are bound to $\emptyset$ and the mappings for the variables of the l.h.s. are bound to the uniquely induced valuation.

In detail, let $X_1, \ldots, X_m$ all the variables occurring in C, save those occurring in the l.h.s. of equalities, and let $X_1, \ldots, X_h$, $h \leq m$ be those variables occurring in $\cup_3$-atoms. Let $n_1, \ldots, n_m$ be auxiliary variables ranging over $\mathbb{N}$. We build the system *Syst* as follows:

- For all $i \leq h$, add the equation $n_i = 0$.

- For all $h < i \leq m$, add the following disequations:

$$\begin{array}{ll} n_i \neq n_j + c & \forall X_i \neq t \text{ in } C \text{ and } c \in \mathit{find}(X_j, t) \\ n_i \neq c & \forall X_i \neq t \text{ in } C \text{ and } t \equiv \{\emptyset\}^c \\ n_i \neq n_j + c + 1 & \forall t \notin X_i \text{ in } C \text{ and } c \in \mathit{find}(X_j, t) \\ n_i \neq c + 1 & \forall t \notin X_i \text{ in } C \text{ and } t \equiv \{\emptyset\}^c \\ n_i \neq n_j & \forall X_i || X_j \text{ in } C \end{array}$$

If $m = k$ then $n_i = 0$ for all $i = 1, \ldots, m$ is the unique solution of *Syst*. Otherwise, it is easy to observe that it admits infinitely many solutions. Let:

- $\{n_1 = 0, \ldots, n_h = 0, n_{h+1} = \bar{n}_{h+1}, \ldots, n_m = \bar{n}_m\}$ be one arbitrarily chosen solution of *Syst*,
- θ be the valuation such that $\theta(X_i) = \{\emptyset\}^{n_i}$ for all $i \leq m$.
- $Y_1, \ldots, Y_k$ be all the variables of C which appear only on the l.h.s. of equalities of the form $Y_i = t_i$.
- σ be the valuation where $\sigma(Y_i) = \theta(t_i)$.

We prove that $\mathcal{SET} \models C[\theta\sigma]$, by case analysis on the form of the literals of C:

$Y_i = t_i$**:** It is satisfied, since $\sigma(Y_i)$ has been defined as a ground term and equal to $\theta(t_i)$.

$X_i \neq t$**:** If t is a ground term, then we have two cases: if t is not of the form $\{\emptyset\}^c$, then it is immediate that $\theta(X_i) \neq t$; if t is of the form $\{\emptyset\}^c$, for some c, then we have $n_i \neq c$, by construction, and hence $\theta(X_i) \neq t$.

If t is not ground, then if $\theta(X_i) = \theta(t)$, then there exists a variable X_j in t such that $\bar{n}_i = \bar{n}_j + c$ for some $c \in \mathit{find}(X_j, t)$; this cannot be the case since we started from a solution of *Syst*.

$t \notin X_i$**:** Similar to the case above.

$\cup_3(X_i, X_j, X_k)$**:** This means that $\bar{n}_i = \bar{n}_j = \bar{n}_k = 0$ and $\theta(X_i) = \theta(X_j) = \theta(X_k) = \emptyset$.

$X_i || X_j$ and $i, j \leq h$, then $\theta(X_i) = \theta(X_j) = \emptyset$.

If $i > h$ (the same if $j > h$), then $\bar{n}_i \neq \bar{n}_j$, and hence $\theta(X_i) = \{\emptyset\}^{\bar{n}_i}$ is disjoint from $\theta(X_j) = \{\emptyset\}^{\bar{n}_j}$.

$\mathsf{set}(X_i)$**:** It is trivially satisfied since all the variables have been instantiated to set-terms.

□

Proof of Theorem 11.5 (Correctness and Completeness)

To prove the correctness and completeness of the procedure $SAT_{\mathcal{SET}}$ we first prove correctness and completeness of the procedure set_infer with respect to the set of successful valuations on $\mathcal{SET}$.

Lemma 13.5 *Let C be a constraint and $C' = C \wedge C^{set}$ be the constraint obtained from C using the procedure* set_infer. *Then for all valuations σ of the variables of C respecting the sorts it holds that*

$$\mathcal{SET} \models C[\sigma] \quad \leftrightarrow \quad \mathcal{SET} \models C'[\sigma]$$

PROOF. If σ is a successful valuation of C' then it is trivially a successful valuation of C. In the other direction, let us assume that σ is such that $\mathcal{SET} \models C[\sigma]$. Then all the literals in $C[\sigma]$ are well-formed, and all the new constraints added by set_infer must be fulfilled. □

Definition 13.6 *C is a* sort-complete constraint *if* $\mathsf{set_infer}(C)$ *infers only already known information. In other words, all possible* set *constraints are already present in C.*

To prove the correctness and completeness of the non-deterministic procedure STEP we first prove the result for each individual rule.

Lemma 13.7 *Let C be a sort-complete constraint and $C_1, \ldots, C_n$ be the constraints (hence neither* false *nor* error*) non-deterministically obtained by applying a single rule of one of the rewriting procedures of Section 9. Then*

$$\mathcal{SET} \models \vec{\forall}\left(C \leftrightarrow \exists \bar{N} \bigvee_{i=1}^{n} C_i\right)$$

where $\bar{N}$ are the newly introduced variables. If $n = 0$ then the right-hand side is equivalent to false*. Moreover, all the constraints C_i are sort-complete.*

Proof. We check the property for each single rule. In particular, we prove that the stronger result $Set \models \vec{\forall}\left(C \leftrightarrow \exists \bar{N} \bigvee_{i=1}^{n} C_i\right)$ holds for most of the rewriting rules. For the other rules we will prove the result on $\mathcal{SET}$.

not_union: There is a unique rule (1) and it is exactly the implementation of the complementation of ($\cup$). The various C_i are trivially sort-complete.

not_disj: There is a unique rule (1). It is the implementation of the complementation of axiom ($||$). Again the constraint remains sort-complete.

member: Rule (1) is justified by axiom (N).

Rule (2) respects the semantics of $\{\cdot \mid \cdot\}$, stated by axiom (W).

For rule (3), assume there is a set N such that $X = \{t \mid N\}$ and set(N): axiom (W) ensures that $t \in X$. On the other hand, if $t \in X$, by axioms (E) and (W) it holds that $X = \{t \mid X\}$. Observe that the constraint set(N) is added to type the new constraint.

union: To prove the correctness and completeness result for this procedure, we first observe that the result in $\mathcal{SET}$ for a variable N in a constraint

$$\{t \mid s\} = \{t \mid N\} \wedge t \notin N \qquad (*)$$

is the set $\{t \mid s\} \setminus \{t\}$ (by (E) and (W)). Moreover, if set(s) then N is forced to be a set, as well. Thus, the constraint set(N) is superfluous.

Rules (1), (2), and (3) follow trivially by axioms (N), (E), and ($\cup$).

Rule (4): It is easy (but tedious) to check that if σ is a successful valuation in $\mathcal{SET}$ for one disjunct among (i)—(iii) then it is a successful valuation for $\cup_3(s_1, s_2, \{t_1 \mid t_2\})$.

In the other direction, let us assume that $\mathcal{SET} \models \cup_3(s_1, s_2, \{t_1 \mid t_2\})[\sigma]$. By ($W$), we have that $\sigma(t_1) \in \sigma(\{t_1 \mid t_2\})$. Three cases are possible:

- $\sigma(t_1) \in \sigma(s_1) \wedge \sigma(t_1) \notin \sigma(s_2)$: by ($\cup$), ($W$), and ($E$) it must be that $\mathcal{SET} \models \cup_3(s_1 \setminus \{t_1\}, s_2, \{t_1 \mid t_2\} \setminus \{t_1\})[\sigma]$. As stated by ($*$) above, the new variables N and N_1 are the witnesses of the set difference.
- $\sigma(t_1) \notin \sigma(s_1) \wedge \sigma(t_1) \in \sigma(s_2)$: similar to the previous case.
- $\sigma(t_1) \in \sigma(s_1) \wedge \sigma(t_1) \in \sigma(s_2)$: by ($\cup$), ($W$), and ($E$) it must be that $\mathcal{SET} \models \cup_3(s_1 \setminus \{t_1\}, s_2 \setminus \{t_1\}, \{t_1 \mid t_2\} \setminus \{t_1\})[\sigma]$. As stated by ($*$) above, the new variables N, N_1, and N_2 are the witnesses of the set difference.

For rule (5) the situation is similar to the case (4) above. $\cup_3(\{t_1 \mid t_2\}, t, X)$ and ($\cup_3$) imply that $t_1 \in X$. N_1 is $\{t_1 \mid t_2\} \setminus \{t_1\}$. N is $X \setminus \{t_1\}$. Case (i) is when $t_1 \notin t$; case (ii) is when $t_1 \in t$.

For rules (6) and (7), we can first observe that ($\leftarrow$) is trivially true. Now, let us assume that $\mathcal{SET} \models \cup_3(X, Y, Z)[\sigma]$. We can identify two cases:

- $\sigma(X) = \emptyset$. In this case, since $X \neq t$, by equality, we have that $\sigma(t) \neq \emptyset$ (case (iii)).
- $\sigma(X) \neq \emptyset$: X must be a (non-empty) set. By (E) there must be an element N_1 that belongs to $\sigma(X)$ and not to $\sigma(t)$ (case (i)) or vice versa (case (ii)).

disj: Rule (1): Axiom $(||)$ is trivially verified by two terms s and t if one of them is the empty set and the other is a set. This is exactly the effect of rule (1).

Given a variable X, such that $\mathsf{set}(X)$, the unique way to prove $X||X$ using axiom $(||)$ is to force $X = \emptyset$. This justifies rule (2).

For rule (3), assume $\{t_1 \mid t_2\}||X$ (or vice versa). This means, by axiom $(||)$, that for all $Z \in \{t_1 \mid t_2\}$, it holds that $Z \notin X$. By axiom (W), $Z \in \{t_1 \mid t_2\}$ if and only if $Z = t_1$ or $Z \in t_2$. Thus, by standard equality axioms, $\{t_1 \mid t_2\}||X$ if and only if $t_1 \notin X$ and for all $Z \in t_2$, it holds that $Z \notin X$, namely $t_1 \notin X$ and $t_2||X$. Observe that, by hypothesis, we already know that $\mathsf{set}(X)$.

For rule (4) the reasoning is similar to the above case: by axioms $(||)$ and $(\not|)$, $\{t_1 \mid s_1\}||\{t_2 \mid s_2\}$ if and only if for all $Z \in \{t_1 \mid s_1\}$ it holds that $Z \notin \{t_2 \mid s_2\}$. This means by (W) that $t_1 \notin \{t_2 \mid s_2\}$ and for all $Z \in s_1$ it holds that $Z \notin \{t_2 \mid s_2\}$. Now, $t_1 \notin \{t_2 \mid s_2\}$ is equivalent, by (W), to $t_1 \neq t_2$ and $t_1 \notin s_2$. It remains to prove that $S_1||\{t_2 \mid s_2\}$ is equivalent to $t_2 \notin s_1 \wedge s_1||s_2$. This can be easily derived by repeating the above reasoning for $\{t_2 \mid s_2\}||s_1$.

not_member: Rule (1) is justified by axiom (N).

Rule (2) fulfills the semantics of $\{\cdot \mid \cdot\}$, stated by axiom (W) and (W').

As proved in (3) of the procedure member, $t \in X$ if and only if there is a set N in $\mathcal{SET}$ such that $X = \{t \mid N\}$. Since X here occurs in t, axiom (F_3^s) ensures that $X \neq \{t \mid N\}$. Thus $t \notin X$ is trivially fulfilled, provided X be a set. But this is known from the initial hypothesis.

not_equal: Rule (1) is justified by axiom (F_2).

For rule (2), the fact that $f(s_1, \ldots, s_n) \neq f(t_1, \ldots, t_n)$ implies $s_i \neq t_i$ for some $i = 1, \ldots, n$ is a consequence of standard equality axioms. The other direction is exactly axiom (F_1').

Rules (3) and (4) are again justified by standard equality axioms.

Rules (5) and (6) are the implementation of the axiom scheme (F_3^s).

Rule (7) is exactly the standard extensionality axiom (E) that can be also written as:

$$X \neq Y \leftrightarrow \exists Z\, ((Z \in X \wedge Z \notin Y) \vee (Z \notin X \wedge Z \in Y))$$

set_check: Rules (1), (2), and (3) are exactly axioms (S_0), (S_1), and (S_2).

equal: Rules (1), (2), and (5) are justified by standard equality axioms. Observe that the test ensures that action (5) can not generate ill-formed terms, although the constraint might become inconsistent.

Rules (3) and (4) are the application of the freeness axiom scheme (F_3^s). For rule (6), assume $X = \{t_0, \ldots, t_n \mid X\}$. Then, choosing $N = X$, there exists N such that $X = \{t_0, \ldots, t_n \mid N\}$ holds. Now, assume there is N such that $X = \{t_0, \ldots, t_n \mid N\}$. Then, using the properties (Ab) and $(C\ell)$ implied by (E) it is immediate to verify that

$$X = \{t_0, \ldots, t_n \mid N\} = \{t_0, \ldots, t_n, t_0, \ldots, t_n \mid N\} = \{t_0, \ldots, t_n \mid X\}\,.$$

Rule (7) is justified by axiom (F_2).

The $(\leftarrow)$ direction of rule (8) is a consequence of equality axioms. The $(\rightarrow)$ direction is axiom (F_1).

Rule (9): Assume there is a successful valuation σ for $\{t \mid s\} = \{t' \mid s'\}$ on $\mathcal{SET}$. Then, we have $\mathsf{set}(s)$ and $\mathsf{set}(s')$. By (E) and (W), and the fact that $\mathcal{SET}$ is a model of *Set*, it holds that σ is a successful valuation of one of the formulae $(i), (ii), (iii)$, or that there is a N such that σ can be expanded with $\sigma(N) = \sigma(s) \setminus \{\sigma(t')\}$ or $\sigma(N) = \sigma(s)$ and it is a successful valuation of (iv).

Conversely, if σ is a successful valuation of one the formulae $(i), (ii), (iii)$, or (iv) on $\mathcal{SET}$, it is immediate to prove using (E) that it is a successful valuation for $\{t \mid s\} = \{t' \mid s'\}$ on $\mathcal{SET}$.

Rule (10): As for rule (9), if σ is a successful valuation of one of the formulae (i)–(iv) on $\mathcal{S}$, it is immediate to prove using (E) that it is a successful valuation for $\{t_0, \ldots, t_m \mid X\} = \{t'_0, \ldots, t'_n \mid X\}$. On the other direction, assume σ is a successful valuation for $\{t_0, \ldots, t_m \mid X\} = \{t'_0, \ldots, t'_n \mid X\}$ and

- $\sigma(t_0) = \sigma(t'_j)$ for some j. Then, by (E) and (W) one of the four disjuncts holds (possibly, with $\sigma(N) = \sigma(X)$ or $\sigma(N) = \sigma(X) \setminus \{\sigma(t_0)\}$).
- $\sigma(t_0) \neq \sigma(t'_j)$ for all j. This means that the disjunct (iv) is satisfied with $\sigma(N) = \sigma(X) \setminus \{\sigma(t_0)\}$.

Rule (11) is justified by the fact that only well-typed successful valuations are accepted.

□

Theorem 11.5 *Let C be a constraint and $C_1, \ldots, C_n$ be the constraints obtained from the application of* set_infer *and from each successive non-deterministic computation of* STEP. *Then,*

1. *if $\mathcal{SET} \models C_i[\sigma]$ then $\mathcal{SET} \models C[\sigma]$ for all $i = 1, \ldots, n$.*
2. *if $\mathcal{SET} \models C[\sigma]$ and $C[\sigma]$ is sort-complete, then for all $i = 1, \ldots, n$, σ can be expanded to the variables of $vars(C_i) \setminus vars(C)$ so that it fulfills $\mathcal{SET} \models C_i[\sigma]$.*

PROOF. The result is immediate from Lemmas 13.7 and 13.5, and from the termination result—Theorem 9.10. □

Proof of Theorem 9.10 (Termination)

Let us start by providing an intuitive description of the way used to prove the termination of $SAT_{\mathcal{SET}}$. We begin by proving that each individual procedure—e.g., equal, not_equal etc.—is locally terminating, i.e., each call to such procedures will stop in a finite number of steps. Local termination of each individual procedure does not guarantee global termination of $SAT_{\mathcal{SET}}$, since the different procedures are dependent on each other—i.e., execution of one procedure may produce constraints which need to be processed by other procedures.

It is common practice to prove termination by developing a *complexity measure* for the system of constraints, and by showing the existence of a well-founded order on the complexity measure. The termination derives from the fact that the steps of the constraint solving algorithm produce constraint systems with a smaller complexity.

There is general agreement that proving termination of various classes of equational unification algorithms (in particular in the case of signatures which include both free and no-free function symbols) is a complex task [8, 30, 28]. Since $SAT_{\mathcal{SET}}$ includes the procedure equal, which indeed belongs to this class of complex problems, we prefer to develop an incremental termination proof, instead of immediately attempting to develop a very complex complexity measure. As mentioned earlier, we start by proving that each procedure locally terminates. In particular, the proof of local termination of equal is an adaptation of the proof presented in [22, 28]. After proving local termination, we show that the algorithm, deprived by the union procedure, always terminate, by making use of a suitable complexity measure. Finally, we insert union in the reasoning and we prove the global termination result.

Local termination

We begin by proving that each individual constraint procedure terminates for all possible input constraints. We will make use of the notion of *size* defined in Def 13.3.

Lemma 13.8 *Each of the individual procedures* member, not_equal, disj, not_member, not_union, not_disj, set_check, set_infer, *and* union *used in $SAT_{\mathcal{SET}}$ terminates.*

PROOF. By case analysis, we show that each rule application decreases a given complexity measure.

member: $size(C_{\in})$ is decreased by each rule application.

not_equal: $size(C_{\neq})$ is decreased by each rule application, except for rule (4), that leaves it unchanged. However, this rule application can only double the number of rule applications performed.

disj: $size(C_{||})$ is decreased by each rule application.

union: $size(C_{\cup_3})$ decreases for rules (1)—(5). It remains unchanged for (6) and (7) but their global execution is bounded by $2|C_{\neq}|$. As a matter of fact, rules (6) and (7)iii introduce the $\neq$-constraint $t \neq \emptyset$, and removes the constraint $Z \neq t$. If t is not a variable this constraint can not fire again any of the rules (6) and (7). If t is a variable, then at most one further application is possible.

not_member: $size(C_{\notin})$ is decreased by each rule application.

not_union: $size(C_{\notin})$ is decreased by each rule application.

not_disj: $size(C_{|\!|})$ is decreased by each rule application.

set_scheck: $size(C_{\mathsf{set}})$ is decreased by each rule application.

set_infer, find_set: are based on recursive calls on smaller size atoms and terms.

□

It remains to prove the local termination of the procedure equal. The intuitive idea behind this part of the proof is that it is possible to determine a bound on the height of the terms which can be generated during the constraint solving process. During the development of the constraint solving process, it is possible to show that the algorithm operates on terms which have progressively decreasing height. Let us start by characterizing the notion of level of a term.

Definition 13.9 *Let C be a collection of $=$ constraints. A* level *is a function*

$$lev : vars(C) \longrightarrow \mathbb{N}$$

The function is extended to terms in $T(\Sigma, \mathcal{V})$ as follows

$$\begin{array}{lcl} lev(f(t_0, \ldots, t_n)) & = & 1 + \max\{lev(t_0), \ldots, lev(t_n)\} \quad f \in \Sigma, f \not\equiv \{\cdot \mid \cdot\} \\ lev(\emptyset) & = & 0 \\ lev(\{s \mid t\}) & = & \max\{1 + lev(s), lev(t)\} \end{array}$$

Given a constraint $s = t$, let us define $lev(s = t) = lev(s) + lev(t)$.

A level function lev is a p-level *if it satisfies the following condition: for each constraint $s = t$ in C we have that $lev(s), lev(t) \leq p$.*

Observe that, given a valuation σ of a constraint, a p-level that also fulfills $lev(\sigma(s)) = lev(\sigma(t))$, for some p depending on the global number of occurrences of function symbols in $\sigma(C)$, must exists However, this property will be a corollary of the proof of termination of equal.

In order to find a suitable bound p for the level functions to be used in the termination result, we develop a graph representation of the terms which appear in the constraints, obtained by generalizing the approach adopted in [53]. Given a system of constraints C we define G_0 to be the initial graph representation of C. G_0 is constructed as follows:

- The graph contains one node for each occurrence of a symbol of Σ in C, and one node for each variable in C. For the sake of simplicity each constant a is replaced with a term $a(X)$ where X is a fixed new variable.

- For each term $f(t_1, \ldots, t_n)$ in C ($f \not\equiv \{\cdot \mid \cdot\}$), if the node μ is associated to the specific occurrence of f and ν_i ($1 \leq i \leq n$) is the node associated to the symbol root of t_i, then the graph contains the edges (μ, ν_i). Each of these edges has a label 1 on it.
- Let r be a term of the form $\{s \mid t\}$ in C. Let μ be the node associated to the outermost occurrence of $\{\cdot \mid \cdot\}$ in r, ν the node associated to the outermost symbol of s and η the node associated to the outermost symbol of t. Then the graph contains one additional node ψ and the following edges: (μ, η) (label 0), (μ, ψ) (label 0), and (ψ, ν) (label 1). This is also illustrated in Figure 15. The node ψ is called the *set enclosure* of s.

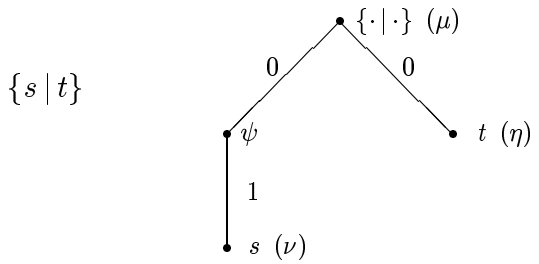

Figure 15: Graph Representation of $\{\cdot \mid \cdot\}$

The equal procedure performs the following transformations to the graph:

- rule (1), (2), (8) leave the graph unchanged;
- rule (5) adds an edge from the node of X to the node of t (with label 0);
- rule (6) replaces an edge (label 0) with another edge to a new node (for N), again with label 0;
- let us assume that the original terms compared are $\{t_0, \ldots, t_m \mid T\}$ and $\{s_0, \ldots, s_n \mid S\}$. Rules (9) and (10) are assumed to be repeated until the set terms that are compared are completely eliminated. This will leave a collection of $=$ constraints of the form $t_j = s_i$ and possibly additional equations of the form $T = \{s_1^T, \ldots, s_h^T \mid N\}$ and $S = \{t_1^S, \ldots, t_k^S \mid N\}$. We assume that the pre-existing nodes introduced for the set enclosures of the elements $s_1^T, \ldots, s_h^T, t_1^S, \ldots, t_k^S$ are used when needed and not repeated. The only new edges created are those that link T and S to the set enclosures of the elements s_i^T and t_j^S (all labeled 0).

We can show the following results:

Lemma 13.10 *If we indicate with G_i a graph obtained after i steps of the* equal *procedure, then G_i contains a number of 1 edges which is no greater than the number of 1 edges in G_0.*

PROOF. Obvious from the description of the graph transformations induced by equal. □

Lemma 13.11 *If we indicate with G_i a graph obtained after i steps of the* equal *procedure, then G_i does not contain any cycle with at least 1 edge.*

PROOF. Let us consider the various possible cases:

- rules (1), (2), and (8) do not add edges to the graph, and thus they cannot lead to the creation of cycles;
- rule (6) adds an edges with label 0 and destination a new variable. Since the new variable does not have any outgoing edges no cycles can arise;

- rule (5) creates a new edge (label 0) from the variable X to the term t. If this generates a cycle, then this means that before this step there was already a path (with at least one 1 edge) from the root of t to the node of X—i.e., X is part of the term t. But this is one of the conditions that prevent the application of rule (5);
- the iteration of rules (9) and (10) leads to a collection of equations of the type $s_i = t_j$ (whose creation does not involve generation of new edges) and the binding of the tail variables as result of equations of the type $S = \{r_1, \ldots, r_k \mid N\}$. First of all observe that being N a new variable, the edge from the node of S to the node of N will not create cycles. The binding creates paths (with one 1 edge) from the node of S to the root of r_i. The conditions which allow the application of the substitution (see rule (5)) guarantee that S does not appear within r_i, which in turn guarantees that there is no path from the root of r_i to the node S. This allows us to conclude that also in this case no cycles are generated.

□

In the following theorem we prove the termination of a slightly more deterministic version of the procedure equal. In particular, the determinism is added to rules (6), (9), and (10).

Theorem 13.12 (equal Termination) *There is an implementation of* equal *that terminates for any given input constraint C.*

PROOF. Let us consider a non-failing derivation produced by the equal procedure $C_0, C_1, C_2, \ldots$. We associate to each C_i a complexity measure $Compl(C_i)$ defined as follows:

$$Compl(C_i) = \langle A_i, B_i \rangle$$

where:

- $A_i = \{\![lev_i(c) \ : \ c \in C_i \wedge \ c \textit{ is not in solved form }]\!\}$
- $B_i = \sum_{s=t \in C_i} size(s)$
- $lev_i : vars(C_i) \longrightarrow \mathbb{N}$ defined as follows:
  - if $i = 0$ then for each X in $vars(C_0)$ $lev_0(X) = p$, where $p = size(C_0) + 1$.
  - if $i > 0$ then lev_i is derived from lev_{i-1} as described step by step in the rest of the proof below.
- $\{\![s_1, \ldots, s_n]\!\}$ denotes the multiset containing the elements $s_1, \ldots, s_n$. The relation $\prec$ is defined as $\{\![s_1, \ldots, s_m, t_2, \ldots, t_n]\!\} \prec \{\![t_1, \ldots, t_n]\!\}$ if $s_1 < t_1, \ldots, s_m < t_1$, $m \geq 0$. With a slight abuse of notation let us indicate with $\prec$ the transitive closure of the above relation. $\prec$ is a well-founded ordering [19].

Let us denote with $\lhd$ the usual lexicographic ordering between pairs. We will use $\lhd$ to compare the complexity of two systems of constraints.

Let us examine the effect of the different rules of the procedure equal.

(1) $lev_{i+1} = lev_i$; A_{i+1} is obtained by removing an element $2lev_i(X)$ from A_i.

(2) $lev_{i+1} = lev_i$; $A_{i+1} = A_i$ while B_{i+1} is equal to $B_i - size(t)$.

(5) In this case we modify lev_i to guarantee that $lev_{i+1}(X) = lev_{i+1}(t)$. This is achieved as follows:

1. if $lev_i(X) = lev_i(t)$ then $lev_{i+1} = lev_i$; A_{i+1} is obtained by removing an element $2lev_i(X)$ from A_i.
2. if $lev_i(X) > lev_i(t)$ then $lev_{i+1}(X) = lev_i(t)$, while $lev_{i+1}(Y) = lev_i(Y)$ for all variables $Y \in vars(C_i)$ different from X. A_{i+1} is obtained from A_i by removing an element $lev_i(X) + lev_i(t)$; moreover, for each c in C_i containing X, we have that $lev_{i+1}(c) \leq lev_i(c)$. Thus $A_{i+1} \prec A_i$.

3. if $lev_i(X) < lev_i(t)$, then we need to reduce the level of certain variables in t to obtain the equality $lev_{i+1}(X) = lev_{i+1}(t)$. The function lev_{i+1} is obtained using the following procedure:

```
procedure reduce ( t: term; n: integer);
  if (lev(t) > n) then
    if (t variable) then
      lev(t) := n;
    else if (t = f(s_1, ..., s_m) and f ≢ {· | ·}) then
      for j := 1 to m do
        reduce(s_j, n − 1);
      endfor
    else if (t = {s_1 | s_2}) then
      reduce(s_1, n − 1);
      reduce(s_2, n);
end;
```

The procedure is executed as follows:

```
lev := lev_i;
reduce(t, lev_i(X));
lev_{i+1} := lev;
```

Observe that the procedure reduce will never be called as reduce(s, m) where s is a ground term and $lev(s) > m$. In fact, observe the following facts:

- if $lev_i(X) = p - c$ with $c > 0$, then this means that the graph contains a path of length c from a variable of level p to the node of X.
- if a call of the form reduce(s, m) with s ground and $lev(s) > m$ occurs, then this means that there exists within the term t a path of length $lev(t)$ containing no variables.
- from the two previous points we can conclude that, since $lev(X) < lev(t)$ and $c = p - lev(X)$, $c + lev(t) > p$. This implies that there exists in the graph a path of length strictly greater than p. Since the graph contains only $p-1$ edges of length 1 (from Lemma 13.10), then this means that the graph contains a cycle with at least one 1 edge. Lemma 13.11 asserts that the presence of this sort of cycles leads to an occur check, and this contradicts the initial hypothesis of a non-failing computation.

A_{i+1} is obtained from A_i by removing an element $lev_i(X) + lev_i(t)$; moreover, for each c in C_i containing variables whose level has been modified by this step, we have that $lev_{i+1}(c) \leq lev_i(c)$. Thus $A_{i+1} \prec A_i$.

(6) Let us assume that the application of this rule is immediately followed by rule (5). After the two steps we have that lev_{i+1} is determined as in the previous step. Moreover, for the new variable N we impose $lev_{i+1}(N) = lev_{i+1}(X)$. Similarly to (5), we have that $A_{i+1} \prec A_i$.

(8) $lev_{i+1} = lev_i$. In A_{i+1} an element $2 + \sum_{j=1}^{n} lev_i(s_j) + lev_i(t_j)$ is replaced by the n smaller elements $lev_i(t_j) + lev(s_j)$. Clearly this leads to $A_{i+1} \prec A_i$.

(9)/(10) let us assume that these two steps are iterated until the set equation is completely resolved.[2] This produces a collection of equations of the form $t_j = t'_k$ plus possibly one of the following cases

- $S = \{t'_{j_1}, \ldots, t'_{j_k}\}$ if the first set term has S as tail variable and the second set term has $\emptyset$ as tail
- $T = \{t_{j_1}, \ldots, t_{j_h}\}$ if the second set term has T as tail variable and the first term has $\emptyset$ as tail

[2]This request, as well as the sequence (6), (5) add determinism to the algorithm. As shown in [22], this is needed to ensure termination.

- $T = \{t_{j_1}, \ldots, t_{j_h} \,|\, N\}$ and $S = \{t'_{j_1}, \ldots, t'_{j_k} \,|\, N\}$ if that the first set term has S has tail variable and the second has T as tail variable
- $X = \{t_{j_1}, \ldots, t_{j_h}, t'_{j_1}, \ldots, t'_{j_k} \,|\, N\}$ if we are in case (10)

As in step (5), the application of the substitution for the variables S, T, or X possibly modifies lev_i. For the third case, we additionally require that $lev_{i+1}(N) = \max(lev_{i+1}(S), lev_{i+1}(T))$. The desired properties derive from the discussion made in step (5).

□

Partial termination

To continue in our incremental proof of termination, in this section we consider the execution of the algorithm without the union procedure. The termination result will be extended to the case of union in the next section.

Let us start assuming that one cycle through $SAT_{\mathcal{SET}}$ has been performed. Observe that none of the procedures produce $\not\emptyset_3$ or $\not\parallel$ constraints. Thus, we can safely ignore the two procedures not_union and not_disj from our discussion. Moreover, after the first iteration of $SAT_{\mathcal{SET}}$, the procedure member is activated only by the step (8) of not_equal or by the union procedure. For this reason, we will also ignore the member procedure, and we will instead directly consider its effect on the rest of the constraint system. Similarly, the equal procedure is reactivated only by the same steps and its global effect is only to substitute a variable with a set term. Thus, for the same reason we will not consider the equal procedure any further and we will just consider its global effect on the constraint system.

We consider the definition of STEP' as in Figure 16:

STEP'(C) :	not_equal(C);
	apply_subs(C);

Figure 16: The procedure STEP'

In the procedure not_equal we assume that the step (8) is iterated until the disequation between sets is completely factored out. This means that step (8i) (step (8ii) is symmetrical) is as follows: assume that $\{s \,|\, r\}$ is $\{s_1, \ldots, s_m \,|\, h\}$ and $\{u \,|\, t\}$ is $\{t_1, \ldots, t_n \,|\, k\}$, with h, k variables or $\emptyset$. The global effect of the subcomputation is that of returning a constraint of the form ($1 \leq i \leq n$):

$$N = s_i, s_i \neq t_1, \ldots, s_i \neq t_n, s_i \notin k$$

or one constraint of the form

$$h = \{N \,|\, N'\}, N \neq t_1, \ldots, N \neq t_n, N \notin k$$

if h is a variable. In this last case, we can also safely assume that if t_i contains h, then the disequation $N \neq t_i$ is immediately removed (being obviously true).

The procedure apply_subs performs the following three tasks:

1. It applies a substitution $X = \{N \,|\, N'\}$ to the whole constraint.
2. It reduces constraints of the form $t \notin \{N \,|\, N'\}$ to $t \notin N'$ and $N \neq t$.
3. It reduces constraints of the form $\{N \,|\, N'\}||Y$ to the constraints $N \notin Y$ and $N'||Y$.

Thus, the effect of apply_subs(C) is that of applying *one* of the substitutions $X = \{N \,|\, N'\}$ generated during step (8) of not_equal. The procedure not_member can only be reactivated by applying a substitution which expands the X in a constraint of the form $t \notin X$. For this reason we directly assume that apply_subs itself simplifies $t \notin \{N \,|\, N'\}$ to $t \notin N'$ and $N \neq t$. Similarly, the procedure disj can only be reactivated

by applying a substitution for X in a constraint of the form $X||Y$. For this reason we can safely assume that apply_subs immediately reduces the constraint $\{N \mid N'\}||Y$ to the constraints $N \notin Y$ and $N'||Y$.

The above assumptions lead to a more deterministic version of the different procedures.

Let A be a new variable introduced by step (8) of not_equal in the substitution $X = \{A \mid B\}$. A constraint of the form $A \neq t$ or $A \notin t$ is called *passive*.

Lemma 13.13 *Let A be a new variable introduced during step* (8) *of* not_equal *in the context of a substitution $X = \{A \mid B\}$. The following properties hold:*

1. *a passive constraint of the form $A \neq t$ introduced is immediately in solved form, and it will never be processed again by* not_equal*;*
2. *a passive constraint of the form $A \notin X$ introduced remains inactive until a substitution for X is generated by step* (8) *of* not_equal*. At that point the constraint is replaced by a pair of constraints, one of the form $A \neq A'$ with the same properties listed in point* 1. *and one of the form $A \notin B$ with the same properties as $A \notin X$;*
3. *step* (8) *of* not_equal *will never generate a substitution of the form $A = \{A' \mid B'\}$.*

PROOF. This result can be proved by induction on the number of substitutions performed. The result is obvious if no substitutions are generated.

Let us consider the application of a substitution $X_n = \{A_n \mid B_n\}$, and let us assume the result to hold for the previous $n-1$ substitutions. In particular this implies that X_n is different from A_i with $i < n$. Since the initial constraint was in solved form for $\neq$, $||$, and $\notin$ constraints, then we have the following possible cases:

- a constraint of the form $Y \neq t$ can be affected by the substitution in two ways. If X_n appears in t then the constraint remains in solved form. Otherwise, if $X_n \equiv Y$ and t is a set $\{t_1, \ldots, t_m \mid h\}$, then rule (8) of not_equal is activated. It is straightforward to verify that substitutions for B_n may be generated in the rest of the computation but not for A_n. Also, it is obvious that the passive constraints will never be reactivated.
- a constraint of the form $X||Y$ can be affected by the substitution if $X \equiv X_n$ or $Y \equiv X_n$. Also in this case it is immediate to verify the validity of the result.
- a constraint of the form $t \notin X$ can be reactivated if $X \equiv X_n$. The situation is similar to the one in the previous case.

□

Theorem 13.14 (Partial termination) *Let C be a constraint obtained after executing the first iteration of $SAT_{\mathcal{SET}}$. The repeated application of* STEP' *as in Figure* 16 *eventually terminates.*

PROOF. Let us define the following complexity measure for a system of constraints C: $Compl(C)$ defined as follows:

$$Compl(C) = \langle A, B \rangle$$

where:

- $lev : vars(C) \longrightarrow \mathbb{N}$ is the level function present at the end of the execution of the equal procedure.
- for each constraint $p(t_1, t_2)$ in C we define $lev(c)$ to be $lev(t_1) + lev(t_2)$
- $$\begin{aligned} A \quad = \quad & \{\!\![lev(c) : c \in C \wedge c \text{ is not passive and } c \text{ is not an equation}]\!\!\} \uplus \\ & \{\!\![lev(X) : X = t \text{ is in } C \text{ and it is not in solved form}]\!\!\} \end{aligned}$$
- $\uplus$ is the union between multisets

- $B = \sum_{s=t, s \neq t \in C_i} size(s)$

Let us consider the different rules:

(1)/(5)/(6) A_{i+1} is obtained by removing an element from A_i.

(2) In A_{i+1} an element $2 + \sum_{j=1}^{n} lev(s_j) + lev(t_j)$ is replaced by a smaller element $lev(t_j) + lev(s_j)$. This leads to $A_{i+1} \prec A_i$.

(4) $A_{i+1} = A_i$ while B_{i+1} is equal to $B_i - size(t)$.

(7) A_{i+1} is obtained by removing an element $lev(X) + \max(lev(t_1) + 1, \ldots, lev(t_n) + 1, lev(X))$ and replacing it with an element $lev(t_j) + lev(X)$.

(8) Let us focus on case $(8i)$; the other case is perfectly symmetrical. Assume that $\{s \mid r\}$ is $\{s_1, \ldots, s_m \mid h\}$ and $\{u \mid t\}$ is $\{t_1, \ldots, t_n \mid k\}$, with h, k variables or $\emptyset$. The global effect of the subcomputation is that of returning a constraint of the form $(1 \leq i \leq n)$:

$$N = s_i, s_i \neq t_1, \ldots, s_i \neq t_n, s_i \notin k$$

or one constraint of the form

$$h = \{N \mid N'\}, N \neq t_1, \ldots, N \neq t_n, N \notin k$$

if h is a variable.

- in the first case $A_{i+1} \prec A_i$ since all the new disequations have level smaller than the initial one.
- in the second case, we define $lev(N) = lev(h) - 1$ and $lev(N') = lev(h)$. Observe that $lev_i(h) > 0$: all variables existing at the end of equal have level greater than or equal to 1; if h was introduced by step (8) in a term $\{\cdot \mid h\}$ then it has the same level as one of the preexisting variables (thus ≥ 1); if h was introduced by (8) in a term $\{h \mid \cdot\}$ then by Lemma 13.13 the variable cannot be instantiated to a non-variable term.

  Observe that A_{i+1} is obtained from A_i by removing one element strictly greater than $lev(h)$ and introducing a new element $lev(h)$.

apply_subs: the procedure apply_subs perform the following three tasks:

1. It applies the substitution $X = \{N \mid N'\}$ to the whole constraint. Observe that A_{i+1} is decreased by $lev(X)$ because of the removal of the equation from the constraint. Observe also that, as in the case of equal, the application of the substitution does not raise the level of any other constraint.

2. It reduces the constraints of the form $t \notin \{N \mid N'\}$ to $t \notin N'$ and $N \neq t$. First of all, $N \neq t$ is a passive constraint, that is not accounted for in the complexity measure. Furthermore, the $lev(t \notin \{N \mid N'\}) = lev(t \notin N')$. Thus this step does not change A_{i+1}.

3. It reduces the constraints of the form $\{N \mid N'\}||Y$ to the constraints $N \notin Y$ and $N'||Y$. Note that $N \notin Y$ is a passive constraint, which is not accounted for in the complexity measure. Furthermore, $lev(N'||Y) = lev(\{N \mid N'\}||Y)$. Thus, A_{i+1} is not affected by this step.

□

Collective Termination

Let us consider now the introduction of $\cup_3$ constraints. As in the previous cases, we introduce a deterministic ordering on some of the steps in order to simplify the termination proof. In particular, we assume the following structure of the execution:

1. we start by performing one complete execution of STEP(C);
2. during the successive iterations, we can observe the following:
   - not_union and not_disj are not executed, since these constraints are never regenerated.
   - as discussed in the previous section, we can safely ignore member and treat it directly as a substitution of a variable.
   - in the previous section, the rest of the execution was described as an iteration of the STEP' as in Figure 16. In this case we extend STEP' by replacing apply_subs with a full-blown version of equal, called equal', that can possibly activate a complete execution of union after each rule application in it.
   - We define a modified version not_equal' of the procedure not_equal. When not_equal forces the application of a substitution, we assume that it might activate the procedure union. The two procedures continue interleaved and using application of substitutions when needed, as far as the solved form is reached. We call not_equal' this extended version of the procedure.

STEP''(C) :	not_equal'(C);
	equal'(C);

Figure 17: The procedure STEP"

More in detail, equal' performs the following actions:

1. it behaves exactly in the same way as equal for all the rules except rule (5);
2. rule (5) leads to the following sequence of actions:
   (a) it applies the substitution to the whole constraint and removes the solved form equation;
   (b) it reduces the constraints of the form $t \notin \{N \mid N'\}$ to $t \notin N'$ and $N \neq t$;
   (c) it reduces the constraints of the form $\{N \mid N'\}||Y$ to the constraints $N \notin Y$ and $N'||Y$.
   (d) it performs an *extended execution* of union, namely, the rules in union are repeated until all the $\cup_3$ constraints are in solved form, and each equation of the form $X = t$ generated during this process is immediately applied to the whole constraint.

Lemma 13.15 *Given a constraint C containing only $\cup_3$ constraints in solved form, and given a substitution $W = t$, an extended execution of* union *(point (*2d*) above) terminates.*

PROOF. Let us observe that t can only be either $\emptyset$ or a set term of the form $\{s_1, \ldots, s_k \mid S\}$—the case where S is $\emptyset$ is simpler and not dealt with in this proof.

If t is $\emptyset$ then the proof is obvious. We can also safely avoid to deal with the application of rules (6) and (7). As a matter of fact, these two rules can occur only at the early phases of the computation and are no longer fired later. Consider for instance the case of rule (6i). The initial constraint is $\cup_3(A, B, C) \wedge C \neq r$. The constraint generated is:

$$\cup_3(A, B, C) \wedge N \in C \wedge N \notin r$$

The membership constraint is further rewritten into $C = \{N \mid N'\}$. This means that, after the execution of equal we will have a situation of the kind:

$$\cup_3(A, B, \{N \mid N'\}) \wedge N \notin r$$

Thus, rule (4) can be applied on the first constraint and the termination considerations done for this rule apply.

Otherwise, let us define a measure of complexity for the system of $\cup_3$ and $\notin$ constraints. The complexity is defined as follows:

$$Compl(C) = \{[\eta(t_1) + \eta(t_2) + \eta(t_3) \ : \ \cup_3(t_1, t_2, t_3) \in C]\}$$

where

- $\eta(V) = |\sigma(\{s_1, \ldots, s_k\})| - |\{\sigma(s_i) \ : \ (\sigma(s_i) \notin X) \in C\}|$ for all the variables $V \in vars(C)$, and
- σ is the substitution produced so far by union; σ is set to be empty at the beginning of each execution of union,
- $\eta(\emptyset) = 0$
- $\eta(\{t \,|\, s\}) = \eta(s) + 1$

Basically, we are trying to take advantage of the fact that the only substitutions the extended execution of union generates are of the form $D = \{s_{i_1}, \ldots, s_{i_h} \,|\, N\}$, where the s_{i_ℓ} are among the initial $s_1, \ldots, s_k$ and constraints $s_{i_\ell} \notin N$ are generated. Thus, $\eta(N) < \eta(D)$.

Different s_i's can become equivalent after the application of some substitution σ: this is the reason for considering σ in the definition of η.

We will show that this complexity measure decreases during the execution. We have assumed that the initial constraint is in solved form, namely it contains only constraints of the type $\cup_3(E, F, G)$, in which $\eta(E) = \eta(F) = \eta(G) = |\{s_1, \ldots, s_k\}| \leq k$.

Assume (this is the most general case) that after the application of σ one of the constraints above has the form:

$$\cup_3(\{s_{i_1}, \ldots, s_{i_a} \,|\, X\}, \{s_{j_1}, \ldots, s_{j_b} \,|\, Y\}, \{s_{h_1}, \ldots, s_{h_c} \,|\, Z\}),$$

Moreover, some constraints among

$$\bigwedge_{\ell=1}^{a} s_{i_\ell} \notin X \wedge \bigwedge_{\ell=1}^{b} s_{j_\ell} \notin Y \wedge \bigwedge_{\ell=1}^{c} s_{h_\ell} \notin Z$$

can be present.

We analyze the various constraints introduced:

(4*i*)
- $\{s_{h_1}, \ldots, s_{h_c} \,|\, Z\} = \{s_{h_1} \,|\, N\} \wedge s_{h_1} \notin N$ Any solution of this equation will have $\eta(N) < \eta(\{s_{h_1}, \ldots, s_{h_c} \,|\, Z\})$
- $\{s_{i_1}, \ldots, s_{i_a} \,|\, X\} = \{s_{i_1} \,|\, N_1\} \wedge s_{i_1} \notin N_1$, Any solution of this equation will have $\eta(N_1) < \eta(\{s_{i_1}, \ldots, s_{i_a} \,|\, X\})$
- $\cup_3(N_1, \{s_{j_1}, \ldots, s_{j_b} \,|\, Y\}, N)$

Thus, $Compl(C)$ decreases. Observe that the introduction of $=$ and $\notin$ constraints does not lead to non-terminating computations, since the algorithm on these kinds of constraints has been proved to terminate in Theorem 13.14.

(4*ii*) is perfectly symmetrical.

(4*iii*)
- $\{s_{h_1}, \ldots, s_{h_c} \,|\, Z\} = \{s_{h_1} \,|\, N\} \wedge s_{h_1} \notin N$ Any solution of this equation will have $\eta(N) < \eta(\{s_{h_1}, \ldots, s_{h_c} \,|\, Z\})$
- $\{s_{i_1}, \ldots, s_{i_a} \,|\, X\} = \{s_{i_1} \,|\, N_1\} \wedge s_{i_1} \notin N_1$, Any solution of this equation will have $\eta(N_1) < \eta(\{s_{i_1}, \ldots, s_{i_a} \,|\, X\})$

- $\{s_{j_1}, \ldots, s_{j_b} \mid Y\} = \{s_{j_1} \mid N_2\} \wedge s_{j_1} \notin N_2$. As in the previous step, any solution of this equation will have $\eta(N_2) < \eta(\{s_{j_1}, \ldots, s_{j_b} \mid Y\})$
- $\cup_3(N_1, N_2, N)$.

Thus, $Compl(C)$ decreases.

(5i) In this case, $c = 0$, namely the third argument is simply Z.

- $\{s_{i_1}, \ldots, s_{i_a} \mid X\} = \{s_{i_1} \mid N_1\} \wedge s_{i_1} \notin N_1$. Any solution of this equation will have $\eta(N_1) < \eta(\{s_{i_1}, \ldots, s_{i_a} \mid X\})$
- $Z = \{s_{i_1} \mid N\} \wedge s_{i_1} \notin N$. For the new variable N it holds that $\eta(N) = \eta(Z) - 1$
- $s_{i_1} \notin \{s_{j_1}, \ldots, s_{j_b} \mid Y\}$
- $\cup_3(N_1, \{s_{j_1}, \ldots, s_{j_b} \mid Y\}, N)$.

Thus, $Compl(C)$ decreases.

(5ii) Again, $c = 0$.

- $\{s_{i_1}, \ldots, s_{i_a} \mid X\} = \{s_{i_1} \mid N_1\} \wedge s_{i_1} \notin N_1$, Any solution of this equation will have $\eta(N_1) < \eta(\{s_{i_1}, \ldots, s_{i_a} \mid X\})$
- $Z = \{s_{i_1} \mid N\} \wedge s_{i_1} \notin N$. For the new variable N it holds that $\eta(N) = \eta(Z) - 1$
- $\{s_{i_1} \mid N_2\} = \{s_{j_1}, \ldots, s_{j_b} \mid Y\} \wedge s_{i_1} \notin N_2$ It holds that $\eta(N_2) < \eta(\{s_{j_1}, \ldots, s_{j_b} \mid Y\})$
- $\cup_3(N_1, N_2, N)$.

Thus, $Compl(C)$ is decreased.

Observe that all substitutions that can be generated are of the desired form $N = \{s_{j_1}, \ldots, s_{j_b} \mid Y\}$, with $s_{j_1} \notin Y, \ldots, s_{j_b} \notin Y$. This allows us to conclude the proof. □

Lemma 13.16 *An execution of* non_equal' *(interleaved with the procedure* union *and application of substitution) always terminates.*

PROOF. If rule (8) is not applicable, termination is immediate. Consider application of rule (8i) and consider the substitution of the form $h = \{A \mid B\}$. Without loss of generality (it is immediate to see the logical equivalence) we assume that also the constraint $A \notin B$ is introduced. This substitution can activate the union procedure. For instance, a constraint $\cup_3(X, Y, h)$ can be replaced by $\cup_3(X, Y, \{A \mid B\})$. One possible effect of action (4) is to introduce the constraints

$$\cup_3(X', Y', B) \wedge X = \{A \mid X'\} \wedge Y = \{A \mid Y'\} \wedge A \notin X' \wedge A \notin Y'$$

At this point we apply the two substitutions $X = \{A \mid X'\} \wedge Y = \{A \mid Y'\}$: they can extend terms and require further applications of union. But the process terminates thanks to local termination—Lemma 13.8: each variable is expanded at most once. After applying all these substitutions, the control returns to not_equal. But the first element of the complexity $Compl(C)$ given in Theorem 13.14 (namely, the multiset of levels of constraints) has decreased. □

We are finally ready for the global termination result.

Theorem 9.10 (Termination) *There exists an implementation of $SAT_{\mathcal{SET}}$ that terminates for any input C.*

PROOF. As explained at the beginning of the section, we consider the more deterministic version of $SAT_{\mathcal{SET}}$ that performs one iteration of STEP and then repeats the sequence STEP" until failure or error are detected, or a solved form is reached. As far as the initial execution of STEP, its termination is ensured by Lemma 13.8 and Theorem 13.12.

To prove the global termination, we will point out a complexity measure that decreases from rule application of the extended procedure equal'. The complexity is the same used for the termination of the procedure equal in Theorem 13.12:

$$Compl(C_i) = \langle A_i, B_i \rangle$$

where:

- $A_i = \{[\![lev_i(c) \ : \ c \in C_i \wedge \ c \textit{ equality not in solved form }]\!]\}$
- $B_i = \sum_{s=t \in C_i} size(s)$

Let us analyze the behavior of this complexity during the different steps of the equal' procedure. The novelty of equal', as described before, is the fact that additional actions are performed after rule (5) (in particular an extended execution of union is performed). The complexity decreases after each rule of equal' different from (5) for the same reasons described in the proof of theorem 13.12. If rule (5) is applied then:

- an equation $X = t$ is removed from the constraint, thus removing from A_i an element $lev_i(X) + lev_i(t)$; additionally the levels of other variables may be decreased as a consequence.
- from lemma 13.15 we know that the extended execution of union terminates; additionally, from the proof of the lemma we can see that:
  - some variable substitutions may be generated and immediately applied; from the proof of 13.15 we can see that these substitutions do not modify the level of any term;
  - some equations of the type $s'_m = s_j$ may be generated; these may arise from the equations $\{s_1, \ldots, s_k \,|\, S\} = \{s'_m \,|\, N\}$ produced by union. It is easy to see that the

    $$lev_i(X) + lev_i(t) > lev_{i+1}(s'_m) + lev_{i+1}(s_j)$$

    since s'_m, s_j are proper subterms of t.

From Lemma 13.16 we know that each extended execution of not_equal terminates without introducing equalities—thus, without affecting the complexity. □