PROGRAM SPECIFICATION AND PROGRAMMING WITH SETS IN LOGIC

G.F.ROSSI
Dipartimento di Matematica, Università di Parma, Via M.D'Azeglio 85/A, 43100 Parma, Italy

We describe the main facilities for set designation and manipulation offered by programming languages and, in particular, by logic programming languages with sets. We show, by means of simple examples, how various forms of set data objects and powerful operations on them (e.g., set unification) can be conveniently exploited in order to obtain a more declarative style of programming and software specification in general.

1 Set data abstractions

Sets are a well-established conceptual tool in mathematics. In the practice of computer science, on the other hand, the use of sets as a data structure is not so common as it might be. Actually, it is widely agreed that the availability of set data abstractions, and of powerful operations upon them, would represent a valuable feature of high-level programming languages and specification languages. According to [1], it would *"[...] improve programmer speed and productivity significantly, and also enhance program clarity and readability"*.

The price for the added expressive power is a certain loss of efficiency. Certainly, this is one of the reasons sets are relatively uncommon as data objects. And this is also the reason for which such kind of languages *"[...] should be regarded, not as a tool for production-efficiency programming, but as a vehicle for rapid experimentation with algorithms and program design"* [1], where efficiency is not a primary requirement.

Among the (few) real-world languages which provide sets as built-in objects we can mention: Pascal and Modula-2 which, however, allow some very primitive and limited support for sets; the specification language Z [2] and the procedural language SETL [1], which, on the contrary, consider sets are their primary data.

2 Sets in logic-based languages

More recently, a number of proposals have been put forward in the area of the integration between logic-based programming paradigms and set theory.

Attention to this area has come first from the field of *deductive databases* (e.g., [3, 4, 5, 6]). Recently, a number of papers have addressed the problem also in a wider setting. General-purpose set constructs and basic operations on sets are inserted into some general logic-based framework: pure logic programming languages ([7]), equational logic languages ([8]), and Constraint Logic Programming (CLP) languages ([9, 10, 11, 12]). The importance of sets as a data structure has been also recognized in `Gödel` [13], which supplies the user with a few basic facilities to define and manipulate sets.

As a matter of fact, logic programming languages seem to be good candidates for hosting set data abstractions, thanks to their potentially high declarative nature, which well combines with the high level of abstraction guaranteed by set constructs.

The development of such kind of extensions raises several interesting theoretical as well as practical problems some of which will be briefly pointed out in the rest of this paper. In particular, we will mainly refer to the techniques, solutions, and notations adopted in $\{log\}$ (read "set-log") [7], an extended logic programming language that embodies the fundamental forms of set designations, along with the basic set-theoretic operations on them.

3 Which kind of sets?

Formal set-theory considers sets as the only objects of the universe of discourse. In our context, however, it is convenient to assume the existence also of non-set objects, often called *atoms*. An atom can belong to a set, but nothing can belong to an atom.

A set can be designated either by simply enumerating all its elements (*extensional sets*) or by stating a property that must be satisfied by all its elements (*intensional sets*). We consider extensional sets first. Then, from Sect. 8, we enlarge the discussion to intensional sets, too.

Sets can contain not only atoms as their elements, but also other sets (*nested sets*). For the sake of generality, we do not put any restriction over set nesting.

Basically, we restrict our attention to *finite sets* only (precisely, *hereditarily finite hybrid sets*, that is sets containing a *finite* number of elements each one of which can be either a non-set element or another *finite* set). Some hints on dealing with infinite sets and other data aggregates will be provided in Sections 11 and 12.

4 Set representation

Hereafter we will assume that the host language where sets are to be embedded is a *(constraint) logic programming language* (cf. [14]). Moreover, we will freely exploit standard Prolog syntactic conventions and notation (in particular, variables begin with an upper-case letter, while constants, functional and predicate symbols begin with a lower-case letter).

In a logic language, sets are naturally represented as *terms*, built out of other terms, starting with atoms and using a suitable set constructor operator (such as union, element insertion, etc.). For example,

- $\{\mathtt{mary}, \mathtt{ann}, \mathtt{peter}\}$
- $\{\mathtt{person}(\mathtt{mary}, \mathtt{white}), \mathtt{person}(\mathtt{peter}, \mathtt{brown})\}$
- $\{\{\mathtt{mary}, \mathtt{ann}\}, \{\mathtt{peter}\}, \{\}\}$

are three set terms. Informally, the first set term denotes the set whose members are (the atoms) `mary`, `ann`, and `peter`. Notice that set elements can be any term including set terms.

A solution commonly adopted to represent sets as terms in a logic language is using a *list-like representation* where a binary function symbol, say $\{\cdot \mid \cdot\}$, is used as the set constructor, and a constant, $\{\}$, is used to denote the empty set. Precisely, $\{\mathtt{t} \mid \mathtt{s}\}$, where `t` and `s` are terms, is interpreted as the set obtained by adding (the object denoted by) `t` as a member to (the set denoted by) `s` (i.e., $\{t\} \cup s$).[a] Notice that, in order to distinguish sets from set terms (i.e., semantics from syntax), we will use italics for the former and teletype font for the latter.

Thus, for example, the set $\{a, b, c\}$ is represented as $\{\mathtt{a} \mid \{\mathtt{b} \mid \{\mathtt{c} \mid \{\}\}\}\}$. The more natural notation $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$—that we will use in all our next examples—is just a convenient abbreviation for this list-like representation of the set $\{a, b, c\}$.

According to the interpretation of set terms as sets, the language should guarantee that order and repetition in a set term don't matter. Precisely, this means that in the underlying set theory the $\{\cdot \mid \cdot\}$ operator must exhibit the *(left) permutativity property*, $\{x, y \mid z\} = \{y, x \mid z\}$, and the *(left) absorption property*, $\{x, x \mid z\} = \{x \mid z\}$.

For example, $\{\mathtt{peter}, \mathtt{mary}, \mathtt{peter}, \mathtt{ann}\}$ should denote the same set as $\{\mathtt{mary}, \mathtt{ann}, \mathtt{peter}\}$.[b]

Being terms, `t` and `s` in the set term $\{\mathtt{t} \mid \mathtt{s}\}$ can be variables, that is, we allow a set to contain variable elements (*non-ground sets*) as well as to be

[a] Such a solution is used for instance in [5], in [4] (where $\{\cdot \mid \cdot\}$ is called *scons*), in [7] (where it is called `with`), and also in the Gödel language [13]. [8] uses the $\cup$ operator, but its use is restricted to allow to implement it as the $\{\cdot \mid \cdot\}$ operator.

[b] Note that duplicates do not occur in the set, of course, but they may occur in the concrete representations of it, i.e., in the set term denoting that set.

partially specified (only some of its elements are explicitly enumerated, while the rest of the set is left unspecified). For example,

- $\{\mathtt{mary},\mathtt{john}\,|\,\mathtt{Others}\}$: a set with two elements and something else, i.e. the set $\{mary, john\} \cup Others$;
- $\{\mathtt{Any}\}$: a singleton set;
- $\{\mathtt{X},\mathtt{Y},\mathtt{Z}\}$: a set with three variable elements, not necessarily all distinct.

5 Set unification

Basic operations on sets include primarily the ability to compare two sets. When considering the more general case where also non-ground and partially specified sets are allowed this operation amounts to solve a non trivial *set unification problem.*

Set unification differs from standard unification in that the former must account for the properties of the set constructor operator, as shown by the following examples (note that, hereafter, $=$ means equal with respect to the underlying set theory T, i.e., $=_T$).

- $\{\mathtt{X},\mathtt{Y}\} = \{\mathtt{mary},\mathtt{john}\}$ has two solutions, namely (i) $\mathtt{X}=\mathtt{a},\mathtt{Y}=\mathtt{b}$ and (ii) $\mathtt{X}=\mathtt{b},\mathtt{Y}=\mathtt{a}$, neither of which is more general than the other (non-uniqueness of mgu's).
- $\{\mathtt{mary}\} = \{\mathtt{mary},\mathtt{mary}\}$ does hold (whereas it doesnot hold using standard unification).
- $\mathtt{X} = \{\mathtt{john}\,|\,\mathtt{X}\}$ has the solution $\mathtt{X} = \{\mathtt{john}\,|\,\mathtt{N}\}$, where $\mathtt{N}$ is a new variable (no solution, instead, using standard unification).

A set unification algorithm using the list-like representation of sets presented above (see Sect. 4) has been described in [7]. Basically, this algorithm extends standard unification by embodying the properties of the set constructor operator $\{\cdot\,|\,\cdot\}$, namely, permutativity and absorption. Another set unification algorithm for a similar simple theory of sets is presented in [15].[c]

[c]An alternative representation of sets, often employed when dealing with the problem of set unification on its own (e.g., [16] and [17]), uses a binary union symbol $\cup$ and a singleton constructor $\{\cdot\}$. In this case, set unification reduces to *ACI-unification* (indeed, $\cup$ must be dealt with as an Associative, Commutative and Idempotent operator). As compared with the $\{\cdot\,|\,\cdot\}$-based representation of sets (cf. Sect. 4), ACI-unification turns out to be more general, but also more complex in the general case, since it must be able to manage complex unification problems, with a large number of distinct solutions, that, on the contrary, cannot

Observe that, while standard unification can be performed in linear time, the set unification problem has been proved to be a *NP-complete problem* (see [18]).

Here are two other (more complex) set unification examples:

- $\{\mathtt{john}\,|\,\mathtt{R}\} = \{\mathtt{mary}\,|\,\mathtt{S}\}$, whose solution is $\mathtt{R} = \{\mathtt{mary}\,|\,\mathtt{N}\}, \mathtt{S} = \{\mathtt{john}\,|\,\mathtt{N}\}$;
- $\{\mathtt{X},\mathtt{Y}\} = \{\mathtt{Z}\,|\,\mathtt{S}\}$, whose solutions are (i) $\mathtt{X} = \mathtt{Y}$, $\mathtt{Y} = \mathtt{Z}$, $\mathtt{S} = \{\}$, (ii) $\mathtt{X} = \mathtt{Y}$, $\mathtt{Y} = \mathtt{Z}$, $\mathtt{S} = \{\mathtt{X}\}$, (iii) $\mathtt{X} = \mathtt{Z}$, $\mathtt{S} = \{\mathtt{Y}\}$, (iv) $\mathtt{X} = \mathtt{Z}$, $\mathtt{S} = \{\mathtt{Y},\mathtt{X}\}$, (v) $\mathtt{Y} = \mathtt{Z}$, $\mathtt{S} = \{\mathtt{X}\}$. (vi) $\mathtt{Y} = \mathtt{Z}$, $\mathtt{S} = \{\mathtt{X},\mathtt{Y}\}$.

6 Operations on sets and set constraints

A number of basic set operations are usually provided by languages that support set data abstractions. Among them: membership ($\in$), inclusion and strict inclusion ($\subseteq$, $\subset$), union ($\cup$), intersection ($\cap$), difference ($\setminus$), cardinality ($\#$) , and so on.[d]

An interesting way of dealing with set operations is viewing them as *constraints*. According to this view, systems of (set) constraints are tested as a whole for satisfiability by suitable ad-hoc *constraint solvers*, in order to control execution. Various proposals have been put forward, in particular, for defining instances of the general CLP scheme [14] for domains consisting of some form of sets. Among them, CLP(Σ^*) [9], CLPS [10], {*log*} [11], CLP(SC) [19].

Thinking of set operations as constraints has a number of advantages. First, one can concentrate more on the set constraint solver. ¿From the language semantics viewpoint, this means that the semantics (both procedural and declarative) of the whole language can be simply inherited by that of the general scheme, provided a suitable constraint solver is made available. ¿From the language implementation viewpoint, more efficient set constraint solvers can be devised, using specialized constraint satisfiability algorithms that exploit the (semantics) properties of the set data objects at hand. For instance, [10] uses a constraint satisfaction mechanism based on partial consistency techniques,

---

be expressed using the $\{\cdot\,|\,\cdot\}$ set constructor. And, as noted in [8], while some expressive convenience is sacrificed by the restriction imposed by the $\{\cdot\,|\,\cdot\}$ constructor, most practical cases are unaffected. Moreover, using the $\{\cdot\,|\,\cdot\}$-based representation of sets allows nested sets to be represented, whereas they are not allowed when using the $\cup$-based representation.

[d] It can be shown (cf., e.g., [11]) that having a logic programming language with set terms and set unification, and adding either $\notin$ or $\neq$ as built-in operations would suffice to define all other basic operations on sets in the language itself (notice that $\notin$ and $\neq$ can be easily defined each one in terms of the other). However, while this is feasible in principle, efficiency and effectiveness considerations usually lead to enrich the language with also (few) other basic operations for set manipulation as built-in operations.

whereas [12] extends the finite domain approach to allow variable domains to vary on the finite powerset of the Herbrand Universe.

Second, constraints can be used to specify the query as well as the answers. Moreover, they can be used with no concern about instantiation of variables occurring in them. This definitively enhances declarativeness (the order of execution of set constraints does not matter) and support a very flexible way of dealing with partially specified sets. For example, in $\{log\}$:

- given the goal $:- \cap(\{\mathtt{X}\},\{\mathtt{Y}\},\mathtt{S})$ we get the solutions (i) $\mathtt{S} = \{\}$, $\mathtt{X} \neq \mathtt{Y}$ and (ii) $\mathtt{S} = \{\mathtt{X}\}, \mathtt{X} = \mathtt{Y}$, the first of which contains a negative constraint;

- given the goal $:- \mathtt{mary} \notin \{\mathtt{X},\mathtt{Y} \mid \mathtt{S}\}\ \&\ \mathtt{Y} \in \{\mathtt{mary},\mathtt{john}\}$ we get the solution $\mathtt{Y} = \mathtt{john}, \mathtt{X} \neq \mathtt{mary}, \mathtt{mary} \notin \mathtt{S}$.

Set constraints on their own has been also extensively studied as a natural formalism for many problems that arise in program analysis (e.g., type-checking or optimization). A number of satisfiability checking procedures has been developed for general set constraints (actually, conjunctions of formulae of the form $X \subseteq Y$, where X and Y are set expressions constructed using the set union, intersection and complement operators) and for smaller sub-classes of them (see, for instance, [20]).

7 Restricted Universal Quantifiers

Another feature which strongly enhances the expressive power of the language is the availability of *Restricted Universal Quantifiers* (*RUQ*s). *RUQ*s are formulas of the form $\mathtt{forall}(\mathtt{X} \in \mathtt{s}, \mathtt{G})$, with G an arbitrary goal containing X. This form stands for the quantified implication $\forall X((X \in s) \to G)$. For example, the following clause defines a predicate `subset` that tests whether `S1` is a subset of `S2`

$$\mathtt{subset}(\mathtt{S1},\mathtt{S2}) :- \mathtt{forall}(\mathtt{X} \in \mathtt{S1}, \mathtt{X} \in \mathtt{S2}).$$

Some form of *RUQ*s are provided by most of the languages with sets cited above. However, while in procedural and functional languages with sets (e.g., SETL and Z), *RUQ*s can be used only as a convenient way to iterate over the elements of a set, in logic-based languages with sets, such as $\{log\}$ and the language in [5], *RUQ*s can be used also as a *set-builder* by which one can construct all the sets satisfying a given property G. As an example, the following call to the `subset` predicate defined above

$$:- \mathtt{subset}(\mathtt{S}, \{\mathtt{mary},\mathtt{john}\})$$

will return, in the $\{log\}$ language, the four distinct solutions, (i) $\mathtt{S} = \{\}$, (ii) $\mathtt{S} = \{\mathtt{mary}\}$, (iii) $\mathtt{S} = \{\mathtt{john}\}$, (iv) $\mathtt{S} = \{\mathtt{mary}, \mathtt{john}\}$, i.e., all possible subsets of the given set $\mathtt{S} = \{\mathtt{mary}, \mathtt{john}\}$.

8 Intensional sets

In the practice of mathematics very often a set S is denoted *intensionally*, that is, by providing a condition φ that is necessary and sufficient for an element X to belong to S. This *intensional* definition of a set is achieved by use of an *intensional set term*, whose typical syntactic form is $\{\, X \,:\, \varphi[X] \,\}$, where X is a variable and φ is any first order formula containing X. Intuitively, $\{\, X \,:\, \varphi \,\}$ means "the set of all instances of X corresponding to the instances of φ which are true".

The availability of such a construct considerably enhances the expressive power of the language, and provides the user with a powerful tool for supporting a declarative programming style. As an example, the following two clauses define the intersection of two sets S1, S2, and the power set of S, respectively:

$$\mathtt{intersection}(\mathtt{S1}, \mathtt{S2}, \mathtt{S3}) :- \mathtt{S3} = \{\mathtt{X} : \mathtt{X} \in \mathtt{S1} \,\&\, \mathtt{X} \in \mathtt{S2}\}.$$

$$\mathtt{powerset}(\mathtt{S}, \mathtt{P}) :- \mathtt{P} = \{\mathtt{X} : \mathtt{subset}(\mathtt{X}, \mathtt{S})\}.$$

Only few proposals for logic programming with sets have tackled the issue of dealing with intensional sets in a semantically clean way. Among them, $\mathcal{LDL}$ (cf., e.g., [4]), *Subset-assertion Programming* (cf., e.g., [8]), and $\{log\}$ [7] (all of them using different approaches for embedding intensional sets in the language).

Some restrictions are usually put on the form the collected set can have, such as, non-emptiness, finiteness, groundness. Actually, the ability to deal with the more general form of intensional sets would require the ability to deal with negative information in a rather non-trivial way (see [21] for a proposal aiming at removing some of the restrictions on intensional sets by the use of a suitable form of negation for implementing the set collection mechanism). For example, let the relation likes between persons and drinks to be defined as follows:

$$\mathtt{likes}(\mathtt{mary}, \mathtt{coke}).$$
$$\mathtt{likes}(\mathtt{peter}, \mathtt{Drink}) :- \mathtt{Drink} = \mathtt{cocktail}(\{\mathtt{rum} \,|\, \mathtt{R}\}).$$

i.e., "mary likes coke" and "peter likes any cocktail containing rum". The query

$$:- \mathtt{Drinkers} = \{\mathtt{X} : \mathtt{likes}(\mathtt{X}, \mathtt{Drink})\}$$

has the following three answers

(i)	$\texttt{Drinkers} = \{\}, \texttt{Drink} \neq \texttt{coke}, \forall \texttt{Z}(\texttt{Drink} \neq \texttt{cocktail}(\{\texttt{rum}\vert\texttt{Z}\})$
(ii)	$\texttt{Drinkers} = \{\texttt{mary}\}, \texttt{Drink} = \texttt{coke}$
(iii)	$\texttt{Drinkers} = \{\texttt{peter}\}, \texttt{Drink} = \texttt{cocktail}(\{\texttt{rum}\vert\texttt{Z}\})$.

Besides the standard intensional set former $\{X : \varphi\}$, other more articulate set formers can be supplied by the language (like, for instance, in SETL and Z) as a convenient way to define a set by its properties. Here are a few examples:

- $\{1..100\}$ is the set of all natural numbers X such that $1 \geq X \leq 100$;
- $\{[\texttt{P},\texttt{S}] : \texttt{P} \in \texttt{Person}\ \&\ \texttt{S} \in \texttt{Salary}\}$ is the set of all pairs, P, S, where P is a *Person* and S is a *Salary* (i.e., the cartesian product of the sets Person and Salary);[e]
- $\{[\texttt{X},\texttt{Y},\texttt{Z}] : \{\texttt{X},\texttt{Y},\texttt{Z}\} = \{\texttt{a},\texttt{b},\texttt{c}\}\}$ is the set of all the permutations of the three elements a, b, c.

9 Set-oriented programming

Set-oriented programming means thinking and designing programs with set data abstractions in mind, trying to really exploit all the potentialities offered by set constructs. Non-determinism in set unification, set constraints, intensional set formers, are all features that potentially allow one to write programs in a more *declarative* fashion, and definitively to obtain simpler and more readable programs.

In particular, set unification can be conveniently exploited in a number of problems —e.g., resource allocation problems, combinatorial problems in general— whose solution can be naturally expressed by a *generate & test* approach. Set unification will non-deterministicaly generate all possible combinations of values for the given problem, and set constraints will select only those values which are really satisfactory. For example, the following is a $\{log\}$ specification for a generate & test solution of the well-known problem of the coloring of a map.

Given a set $\{c_1,\ldots,c_m\}$ of Colors, a set $\{R_1,\ldots,R_n\}$ of Regions, $m \leq n$, and a set $\texttt{Map} = \{\{R_{i_1},R_{j_1}\},\ldots,\{R_{i_k},R_{j_k}\}\}$, $i_p \neq j_p$ for all $p = 1,\ldots,k$, of pairs of neighboring regions, the predicate coloring returns an assignment of colors to the regions so that no two neighboring regions have the same color. (Note also how *RUQ*s contribute to a declarative readings of programs.)

[e] Observe that P and S are intended to be (existentially) quantified inside the set former, whereas Person and Salary are free.

```
coloring(Regions, Map, Colors) :-
    Regions = Colors &
    (forall(R ∈ Regions, {R} ∉ Map).
```

A sample goal:

```
:- coloring({England, Wales, Scotland},
            {{England, Wales}, {England, Scotland}}, {green, red}).
```

(*i*) England = green, Wales = red, Scotland = red

(*ii*) England = red, Wales = green, Scotland = green.

Observe that, if one of the colors, say green, is left unspecified by using a variable, say X, in its place, we get the following answers (involving negative constraints): (*i*) England = X, Wales = red, Scotland = red, X $\neq$ red, (*ii*) England = red, Wales = X, Scotland = X, X $\neq$ red.

A decisive step towards a more declarative way of programming is obtained by the use intensional sets. Here is an example of a {*log*} specification using intensional sets (adapted from [1]).

Suppose a set S of integer numbers is given. The problem is finding the *median* of the numbers in S (the median is often used in statistic to represent a "typical" member of a set S). The following is a {*log*} definition of the predicate median:

```
median(S, M) :-
    M ∈ S & #({Z : Z ∈ S & Z < M}, N1) & #({Z : Z ∈ S & Z > M}, N2) &
    N1 = N2.
```

that is, the median is the unique element M of S such that there are as many elements of S which are smaller than M as there are elements of S which are larger than M (if S has an even number of elements median is assumed to be false). For example, given the goal :- median({18, 5, −3, 7, 1, 7, 11, 2}, M) we get the answer M = 5.

10 Which kind of applications?

We are not restricting ourselves to any specific application domain. Indeed, we are thinking of a *general-purpose* language with sets, especially 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, allowing some definite improvements both in the quality and in the development time of the final software product. These "interesting" areas include database applications (see for instance [5]), combinatorial problems, graph-related applications and operational research in general (e.g., resource allocation problems), as pointed out for instance in [12, 10]. Also, in the A.I. field, set-based languages have already been successfully applied in various areas, such as computational linguistics (in particular, constraint based grammars), but further investigation surely deserves to be put forward for evaluating their use in other A.I. related areas, such as knowledge representation (in particular, commonsense reasoning).

11 Finite vs. infinite sets

Through the intensional definition of sets one can easily represent also infinite sets. Although most of the above mentioned proposals do not exclude this possibility, they actually do not provide any support for dealing with infinite sets. Thus, trying to collect an infinite sets would cause an endless computation.

Notwithstanding, there are cases where the operations to be performed on an intensionally defined set do not necessarily require all elements of the set to be explicitly collected. For example, the goal $:- \mathtt{0} \in \{\mathtt{X} : \mathtt{nat(X)}\}$, where $\mathtt{nat(X)}$ is true when $\mathtt{X}$ is a natural number, can be clearly replaced by the simpler goal $:- \mathtt{nat(0)}$. The very same idea could be applied also to other basic operations on sets, such as $=$, $\cup$, $\cap$, etc. A set of program rewriting rules can be provided that eliminate intensional sets when occurring as arguments of these operations by generating equivalent new programs without intensional sets. Although operations involving infinite sets can be undecidable in the general case, there are many special cases for which execution of this simplified program can be carried out successfully.

12 Other data aggregates

Although we have considered only "conventional" sets, there are a number of other data aggregate abstractions, akin to sets, which may turn out to fit more naturally the problem requirements than sets in many interesting applications. Among them, *multisets* (or bags), where the number of occurrences of each element does matter whereas the order of elements does not, and *hypersets*, that is *non-well-founded sets* where, roughly speaking, membership is allowed to form cycles (examples of hypersets arise when trying to model a circular process or structure by means of sets).

Actually, the solutions and techniques described in this paper are general and flexible enough to be quite easily adapted to these other data structures.

A formal characterization of various kinds of data aggregates and the definition of suitable unification algorithms dealing with them in a logic programming framework can be found in [22, 23].

13 Conclusions

We have argued that set-based languages can be an effective and useful vehicle to allow the well-known mathematical notion of set to enter into the real-world computer usage, where they can be exploited to significantly improve programmer speed and productivity. A lot of practical as well as theoretical problems still remain open (e.g., the development of (efficient) set constraint solvers, infinite set handling, etc.), all of which deserve further investigation.

Acknowledgments

$\{log\}$ and most of the ideas about logic programming with sets presented in this paper are the outcome of the joint work with A. Dovier, E. Omodeo, and E. Pontelli, to all of whom I'm deeply indebted.

References

1. J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with sets, an introduction to SETL.* Springer-Verlag, 1986.
2. J. M. Spivey. *The Z Notation: A reference Manual, 2^{nd} edition.* Int'l Series in Computer Science. Prentice Hall, 1992.
3. S. Abiteboul and S. Grumbach. A rule-based language with functions and sets. *ACM Trans. on Database Systems*, 16(1), 1991.
4. C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set Constructors in a Logic Database Language. *J. of Logic Programming*, 10(3):181–232, 1991.
5. G. M. Kuper. Logic Programming with Sets. *J. of Computer and System Science*, 41(1):66–75, 1990.
6. O. Shmueli, S. Tsur, C. Zaniolo. Compilation of Set Terms in the Logic Data Language (LDL). *J. of Logic Programming*, 12(1):89–120, 1992.
7. A. Dovier, E. G. Omodeo, E. Pontelli, and G. Rossi. {log}: A Language for Programming in Logic with Finite Sets. *J. of Logic Programming*, 28(1):1–44, 1996.
8. B. Jayaraman. Implementation of Subset-Equational Programs. *J. of Logic Programming*, 12(4):299–324, 1992.
9. C. Walinski. CLP(Σ^*): Constraint Logic Programming with Regular Sets. In *Proc. 6th Int'l Conf. on Logic Programming.* 181–190, The MIT Press, 1989.
10. B. Legeard and E. Legros. Short overview of the CLPS System. In *Proc. 3rd Int'l Symp. on Programming Language Implementation and Logic Programming*, vol. 528 of LNCS, 431–433. Springer-Verlag, 1991.

11. A. Dovier and G. Rossi. Embedding Extensional Finite Sets in CLP. In *Proc. Int'l Logic Programming Symp., ILPS'93*, 540–556. The MIT Press,1993.
12. C. Gervet. Conjunto: Constraint Logic Programming with Finite Set Domains. In *Proc. Int'l Logic Programming Symp., ILPS'94*, 339–358. The MIT Press, 1994.
13. P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. The MIT Press, 1994.
14. J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *J. of Logic Programming*, 19–20:503–581, 1994.
15. F. Stolzenburg. Membership-Constraint and Complexity in Logic Programming with Sets. In *Proc. 1st Int'l Workshop on Frontiers of Combining Systems*, *Applied Logic*, v. 3. Kluwer Academic Publ., 1996.
16. W. Büttner. Unification in the Data Structure Sets. In *8th Int'l Conf. on Automated Deduction*, *LNCS*, v. 230, 470–488. Springer-Verlag, 1986.
17. M. Livesey and J. Siekmann. Unification of Sets and Multisets. Tech. rep., Institut für Informatik I, Universität Karlsruhe, 1976.
18. D. Kapur and P. Narendran. NP-completeness of the set unification and matching problems. In J. H. Siekmann, ed., *Proc. 8th Int'l Conf. on Automated Deduction*, *LNCS*, v. 230, 489–495. Springer-Verlag, 1986.
19. D. Kozen. Set Constraints and Logic Programming. Tech. rep. 94-1467, Computer Science Dept, Cornell Univ., 1994.
20. A. Aiken and E. Wimmers. Solving systems of set constraints. In *Proc. 7th Symp. on Logic in Computer Science*, 329–340. IEEE, 1992.
21. A. Dovier, E. Pontelli, and G. Rossi. The CLP language {log}, and the relation between Intensional sets and Negation. Tech. rep. NMSU-CSTR-9503, Dept of Computer Science, Las Cruces, N.M., USA, 1995.
22. A. Dovier,E. G. Omodeo, A. Policriti, and G. Rossi. Solving Systems of Equations over Hypersets. In *GULP–PRODE'94 1994 Joint Conf. on Declarative Programming* 403–417, 1994.
23. A. Dovier, A. Policriti, and G. Rossi. Integrating lists, multisets, and sets in a logic programming framework. In *First Int'l Workshop on Frontiers of Combining Systems*, *Applied Logic*, v. 3, 213–229. Kluwer Academic Publishers, 1996.