

SPARCL: A Logic Programming Language Built on Sets

Lindsey Spratt and Allen Ambler

Department of Electrical Engineering and Computer Science
University of Kansas,
Lawrence, KS, 66045

Abstract

*This paper presents a novel approach to the basic organization of a logic programming language. We use finite sets as the fundamental method of collecting together terms, as opposed to “structures” or “lists”. This commitment to use finite sets exclusively has many repercussions throughout the design of the programming language, in its semantics, its syntax, its representation, and its programming techniques. We have implemented SPARCL (Sets with **P**ARTitioning **C**onstraints in **L**ogic) to investigate this set-based approach.*

Introduction

Every programming language must have one (or more) basic method for collecting together data. Such tools include lists, structures, records, and arrays. In this paper we describe a language, SPARCL, in which finite sets are used as the fundamental method of collecting together terms. The basic choice of how to organize data influences many aspects of a programming language’s design and usage. Thus, the commitment to use finite sets exclusively has many repercussions throughout the design of the programming language, in its semantics, its syntax, its representation, and in its programming techniques. Earlier works on SPARCL include an initial presentation of the language [1] and a discussion of a particular three-dimensional display technique used in the language [2].

Our goal in following the set-based, visual, and declarative approaches to programming language design is to provide a powerful tool for “exploratory” programming, by which we mean using the programming process to help one to understand a complex problem. The attraction of sets is that they are the simplest construct for specifying collections of data. A set does not imply an order on its members, and a term either is or is not a member – a term isn’t a member to some degree (as in fuzzy sets) or a member some number of times (as in multisets or bags).

Thus, sets allow a programmer to be semantically precise and only include those constraints on data which are appropriate to the problem at hand. More constrained representations of data can be built “on top” of sets, as necessary.

Set-based programming features discussed in this paper include: multiple representations of sets, representation and semantics of programs, and partitioned set constraints in unification pattern matching.

Set Theory

We have adopted as our underlying set theory NBG Set Theory (von Neumann, Bernays, Gödel), as presented in [3], extended with “ur” constants. The ur constants are the numbers and alphanumeric tokens commonly used in writing the contents of sets. They don’t change the mathematical properties of NBG. Mendelson recommends [4] for the minor changes to add constants to NBG. I have adopted the term “ur constant” from [5].

NBG is a first-order theory. In the intended interpretation, variables take *classes* as values. Classes are “the totalities corresponding to some, but not necessarily all, properties.”:

Those properties which actually do determine classes will be partially specified in the axioms. These axioms provide us with the classes we need in mathematics and appear (we hope) modest enough so that contradictions are not derivable from them.¹

1. p. 160 in [3].

In NBG, any class which belongs to some other class is a *set* (this avoids a classic set theory paradox). The set theory we actually need is a very simple subset of NBG, since we work with *finite* sets.

Related Work

There are many instances of researchers combining sets with logic programming as well as other kinds of programming. But all of these efforts view sets as something that are added onto a language, rather than the essential organizing technique. The research in this paper emphasizes sets as the essential organizing technique.

Sets in logic programming

{Log} (read “set log”) [6] adds sets to PROLOG. LDL [7] is a database-query oriented logic programming language similar to PROLOG that has sets as “first class objects”. GÖDEL [8] is a logic programming language that includes a set processing capability as an “add on” module. There is an unnamed language by [9] which combines subset processing in an equational logic programming system.

There are several approaches to extending CLP for sets: a CLP extension by Gervet [10], CLPS [11; 12], CLP(Σ^*) [13], and an unnamed extension by [14].

Multisets are used by some logic programming languages: GAMMA [15] and the language of Hölldobler and Thielscher [16].

The above approaches to sets in logic programming are all based on an axiomatic approach to set theory: extend classical logic with set axioms. George Tsiknis identifies the “logistic approach” that introduces set abstraction terms into arguments by “simple” rules of deduction². He presents SetLog, a logic programming language with set abstraction based on NaDSet* (which Tsiknis says is derived from Gilmore’s NaDSet).


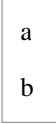

Comparison with set-based approach.


These systems do not rely on sets as the central method of organizing data, they do not use sets in the metalanguages in which these systems are described, they do not provide multiple representations for sets, and they don’t use partitioning as a basic programming mechanism.

Sets for Representing Terms

Sets in a programming language are either extensional or intensional. Both are part of SPARCL. Extensional sets are used as the basic organization for collections of terms. Intensional sets are essentially a notational shorthand for a certain use of the *setof/3* and *multisetof/3* builtin predicates; intensional sets do not have an explicit presence in the semantics of the language.

One of the issues to resolve in representing sets is the uniqueness of elements. A set containing variables may be interpreted in such a way that distinct elements in the pre-interpreted form of the set become the same element in the interpreted form of the set. For instance the set with

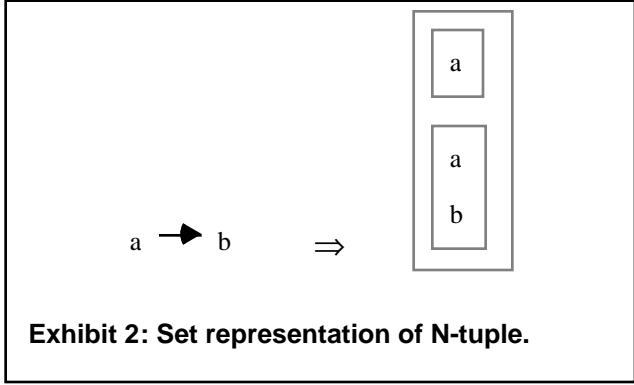
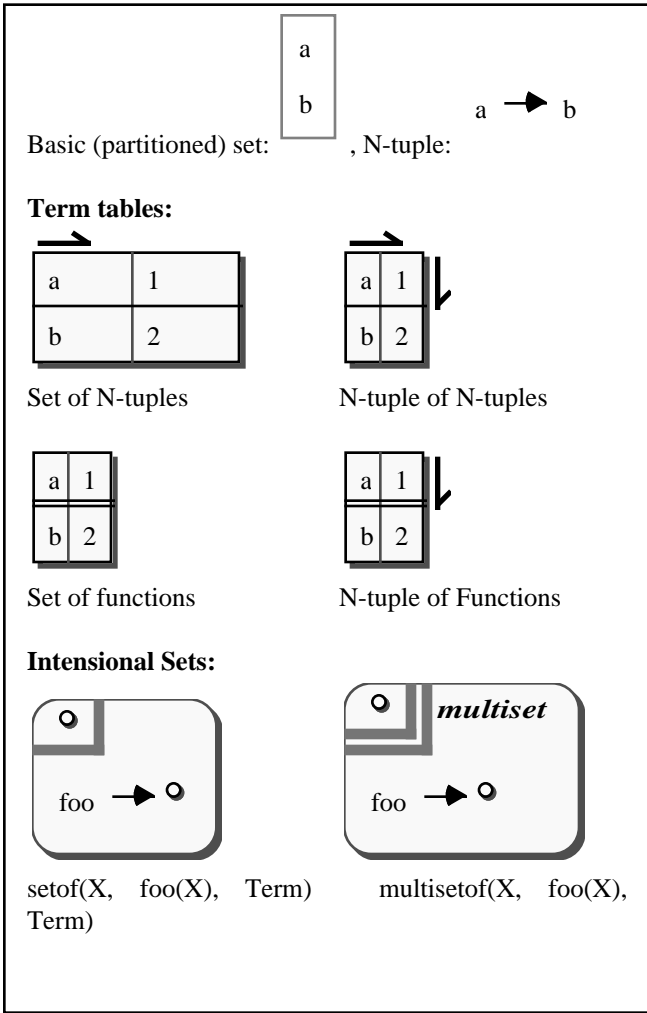
two variables,  , can be interpreted as either  

or  . An interesting feature of this situation is that the cardinality of a non-ground set may be unknown prior to interpretation, even though all of the elements of the set have an explicit representation. There is no similar situation for a language based on lists - if all of the elements of the list have a representation, even if those representations are all uninterpreted variables, then the length of the list is known.

Representing other types of collections as sets.

Other types of collections, such as N-tuples, multisets, lists, tables, and matrices, are defined as special types of extensional sets. Also, program structures such as clauses and literals are mapped onto (set-based) terms. This facilitates meta-programming (as is usual for logic programming languages), and provides an opportunity for unusual semantics. Other approaches to combining sets with logic programming introduce sets into a representation that has some other more primitive data-organization tool. For example, CUBE[18] has a traditional PROLOG-like basic structure, “on top” of which is added the *scons/2* function that is used to build (finite) extensional sets. In contrast, using sets as the fundamental data organization tool contributes to the novelty of a logic programming language in several ways. One of these novelties is the use of set partitioning to work with the structure of sets. This is used in a fashion loosely analogous to the way one uses *functor/arg*, *head/tail* or *car/cdr* facilities for structures or lists.

2. Tsiknis, [17].



N-Tuples

An N-tuple (N > 1) is built recursively from the definition of an ordered pair³:

$$\langle a, b \rangle = \{\{a\}, \{a, b\}\}$$

A 1-tuple has the “degenerate” definition:

$$\langle x \rangle = x.$$

The N-tuple is defined as:

$$\langle x_1, \dots, x_n \rangle = \langle \langle x_1, \dots, x_{n-1} \rangle, x_n \rangle.$$

A list is an N-tuple with the ‘empty_list’ constant as its first element:

$$[x_1, \dots, x_n] = \langle \text{empty_list}, x_1, \dots, x_n \rangle.$$

The “degenerate” 1-tuple definition yields for an empty list is:

$$[] = \langle \text{empty_list} \rangle = \text{empty_list}.$$

Mappings, functions, and multisets.

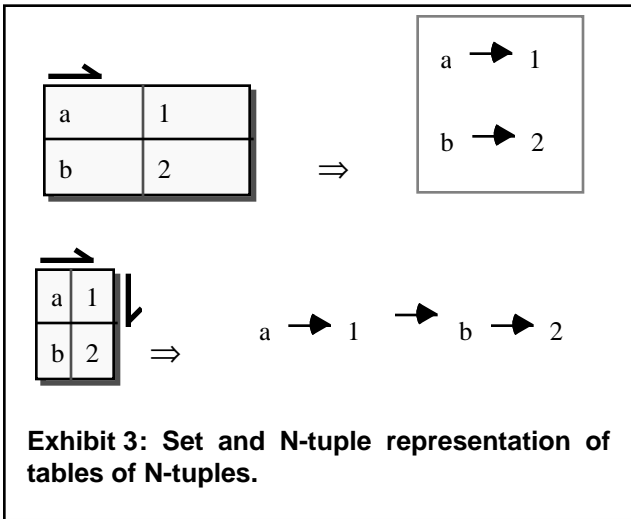
A *mapping* is a set of ordered pairs, where the first elements of the ordered pairs are the domain of the function, and the second elements of the ordered pairs are the

3. This definition is from [3], page 162. Mendelson credits Kuratowski with discovering this definition of an ordered pair in terms of sets. He notes that it

“does not have any intrinsic intuitive meaning. It is just a convenient way ... to define ordered pairs so that one can prove the characteristic property of ordered pairs... (x)(y)(u)(v)(\langle x, y \rangle = \langle u, v \rangle \rightarrow x = u \text{ and } y = v).”

Specialized Set Definitions and Representations

Of the special types of extensional sets mentioned above, N-tuples and tables have specialized representations in SPARCL. We anticipate that lists and matrices will eventually also have specialized representations. Examples of the implemented representations are shown in Exhibit 1. The example N-tuple is expanded in Exhibit 2, term tables of N-tuples in Exhibit 3, term tables of functions in Exhibit 4, and intensional sets in Exhibit 5.

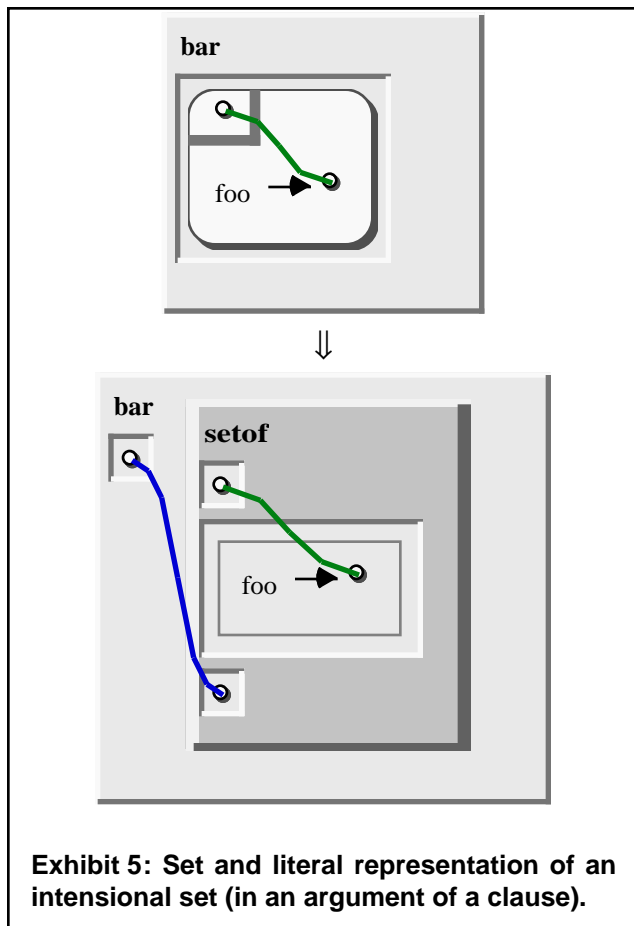
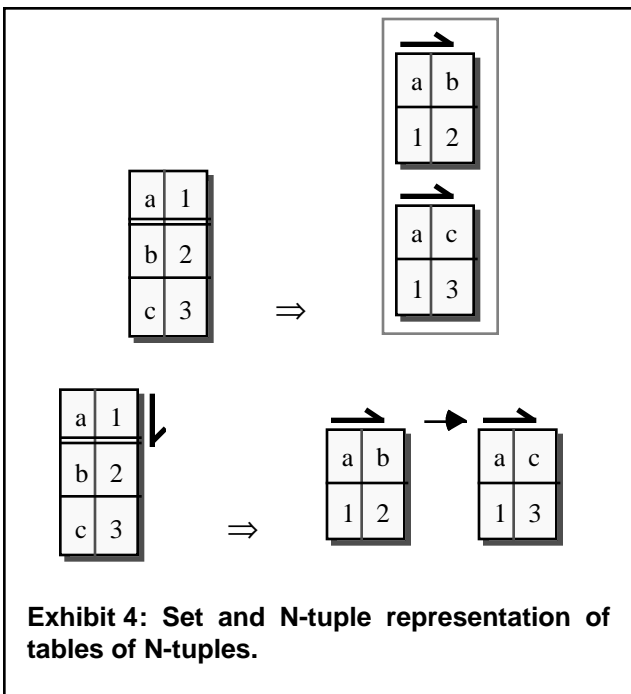


range. A *function* is a mapping where no two ordered pairs have the same domain element.

A *multiset* is a function that has the positive integers as its range. The intensional set representation can be specified to be an intensional *multiset*, and the arithmetic operators '+' and '*' as used by the is/2 builtin predicate both accept multisets (with numbers as their domain).

Tables and matrices.

A term table is a set or an N-tuple of rows, where each row is an N-tuple or a function. If the term table is an N-tuple of rows, then there may be a "prefix" element (a "0-



th row") that may be any term. If the value for such a prefix is 'empty_list' (this being the "root" element of an N-tuple that represents a list), then the term table a list of rows. If the rows of the term table are N-tuples, then they must all be of the same length. If the rows of the term table are functions, then they must all have the same domain.

A matrix is a list of k lists of length n:

$$\begin{bmatrix} x_{11} & \mathbf{L} & x_{1n} \\ \mathbf{M} & \mathbf{L} & \mathbf{M} \\ x_{k1} & \mathbf{L} & x_{kn} \end{bmatrix} = \left[[x_{11} \mathbf{L} x_{1n}], \mathbf{K}, [x_{k1} \mathbf{L} x_{kn}] \right].$$

Sets for Representing Clauses

Clause representation.

A literal in SPARCL is represented as an N-tuple, with the first element of the N-tuple being the predicate name. The rest of the elements of the N-tuple are the arguments of the predicate. The predicate name element must be an ur constant (text) *by the time the literal is evaluated*. A clause is an ordered pair of a head and a body, where the head is a literal and the body is a set of literals. These clauses are basically Horn clauses.

Clause interpretation.

Since a clause body is a set of literals, there is no ordering of these literals. Also, the “same” literal cannot appear in a body more than once, regardless of the number of instances of a literal in the concrete representation of a clause body. That is, the entire clause body has been evaluated once each of its members has been evaluated. For instance, if one wrote a clause body as ‘{a, a}’, the literal ‘a’ is evaluated once, since this representation describes a set that has as its only member ‘a’. Having evaluated ‘a’, one has evaluated all of the members of the represented set.

Clause database.

A clause database is a set of clauses, and thus is unordered. The inference engine is free to inspect the clauses and literals in any order when attempting to solve a query. The engine does solve queries sequentially, but the sequence is not predetermined by the programmer. An exception to this is that the programmer may specify “delay” conditions. As long as a literal satisfies a delay condition its evaluation will not be attempted.

Programming Techniques in a Set-based Language

Some programming techniques that are part of the basic toolkit in non-set-based languages can be dramatically inappropriate in certain circumstances when writing programs in a set-based language. This is because other techniques lead to more concise programs, and also because these other techniques are enormously more efficient.

SPARCL programs.

Three example SPARCL programs illustrate various points made below about programming techniques. We very briefly explain the representation of SPARCL programs to aid in the reader’s understanding of these examples.

A SPARCL program is a collection of Horn clauses. Each outermost box is a clause. The predicate name of the head of the clause is in the upper left corner of the clause. The arguments of the head of the clause are along the left edge of the clause, with the first argument at the top (underneath the predicate name) and the subsequent arguments beneath it. Literals are darker boxes inside the clause box. Each literal has a predicate name in its upper left corner and arguments along its left side.

The basic terms of SPARCL are:

- 1) variable represented by a small circle,
- 2) ur constants represented by text,
- 3) partitioned set represented by one or more dashed boxes (parts) inside of a solid box (the set), and
- 4) N-tuple represented by a horizontal sequence of terms inside a box and separated by rightward pointing arrows.

Term “coreference” is represented by a curved connecting line between the coreferenced terms (this incorporates the variable coreference achieved in a textual language by same-named variables).

Recursion, iteration, and intensional sets: efficiency.

There are many circumstances in programming in a set-based language where neither recursion nor iteration is the appropriate method for processing a collection of elements. This is surprising to a programmer not accustomed to working with sets, as nearly all programming paradigms rely on recursion or iteration, or both. The “set of” intensional set operation is available in several languages, but it is not very widely used. Intensional sets are frequently much the most appropriate representation in SPARCL.

The essential problem is that both recursion and iteration process elements *in order*. Even when the elements being processed have no inherent order, one must be imposed. In languages that do not have sets as a fundamental type, sets are implemented using an ordered representation (such as a sorted list). Thus, there is an obvious

unique order in which to process the elements of a set when the problem being solved does not itself impose an order. In a language such as SPARCL with sets as the fundamental organization, order is neither required nor implied.

Since many different orderings of a set of elements are possible, an iteration or a recursion over the elements of the set must make a number of *choices* about which element to process next. The number of choices is equal to the cardinality of the set). When these choices do not matter (any ordering of the elements will do), then these choices significantly and needlessly slow down the proof process. Worse, if it is necessary to backtrack over the entire processing of the set to some earlier point in the proof process, then it will be necessary to pointlessly try *all possible permutations* of choices, which is the factorial of the cardinality. The underlying inference mechanism can be optimized to handle intensional sets directly with some extensions to the unification and constraint mechanisms, rather than converting intensional sets to an extensional representation as we have done in SPARCL. In this case, intensional sets offer an additional performance benefit. Thus, when considering even the most rudimentary efficiency issues, predicates should use intensional sets whenever they are appropriate, instead of recursively or iteratively processing extensional sets.

Recursion, iteration, and intensional sets: understandability.

An intensional set can be specified in a very compact form. Suppose the programming problem is to find the set of all y such that x is a member of some given set S and $\mathbf{P}(x, y)$ holds. A predicate that contains a solution to this problem is $\mathbf{F}(S, R)$, where R is the desired “result” set. This is specified using an intensional set as:

$$\mathbf{F}(S, \{y | x \in S \wedge \mathbf{P}(x, y)\})$$

A recursive logic programming style version of this is:

$$\begin{aligned} & \mathbf{F}(S, \{y\} \cup R) \\ &= \begin{cases} x \in S \wedge \mathbf{P}(x, y) \wedge \mathbf{F}(S - \{x\}, R) \\ \forall x \neg (x \in S \wedge \mathbf{P}(x, y)) \wedge R = \emptyset \end{cases} \end{aligned}$$

The following equation defines $\mathbf{P}'(x)$ in a convenient alternative form of $\mathbf{P}(x, y)$ for defining the recursive and iterative equivalents to the above.

$$\mathbf{P}'(x) = \begin{cases} \{y\} & \text{if } \mathbf{P}(x, y) \\ \emptyset & \text{otherwise} \end{cases}$$

A corresponding recursive (and functional) definition of this set might be:

$$\mathbf{F}(S) = \begin{cases} \mathbf{P}'(x) \cup \mathbf{F}(S - \{x\}) & \text{if } \exists x \in S \\ \emptyset & \text{if } S = \emptyset \end{cases}$$

An iterative version is:

$$\mathbf{F}(S) : \begin{cases} R = \emptyset; \\ \text{until}(S = \emptyset) \left\{ \begin{array}{l} x = \mathbf{member_of}(S); \\ R = R \cup \mathbf{P}'(x); \\ S = S - \{x\} \end{array} \right. \\ \text{return}(R) \end{cases}$$

These examples can be further simplified if one has special builtins for processing sets (or lists, if sets are implemented as lists), such as the map functions of LISP. But this is essentially using a limited form of an intensional-set like construct to avoid a recursive or iterative specification. This is limited in that it produces one output item for each input item - which may not be the appropriate result. This is due to duplication of resulting items or “filtering” of input items for which no output item is to be produced.

The SPARCL solutions for this abstract programming problem are given in Exhibit 6 and Exhibit 7. The recursive solution is much more complex than the intensional one.

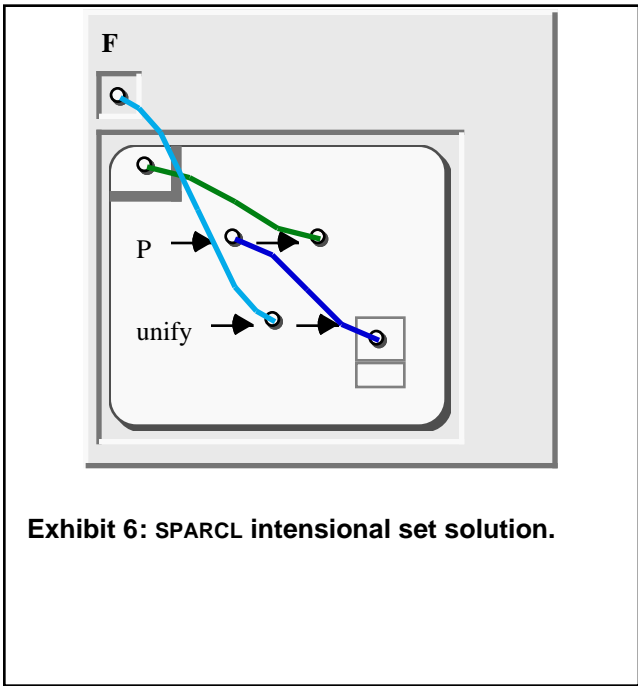


Exhibit 6: SPARCL intensional set solution.

Partitioning constraints: working with the structure of sets.

The partitioned set technique provides an analogous facility for working with sets that the head/tail (car/cdr) or the less familiar “segmenting” technique provides for lists. Typically, one uses these techniques to divide a collection into two (or more) parts and describe different conditions that are true of those parts. The head/tail technique is basic to defining a predicate (in PROLOG) or function (in LISP) which recurses “down” a list, applying some condition to each element of the list. This technique is used to build the very commonly used membership, append, and mapping predicates/functions.

A partitioned set S has one or more “parts” P_i . Parts P_i are themselves sets. The union of these parts is equal to S , and the parts are pairwise disjoint: the P_i form a partition-

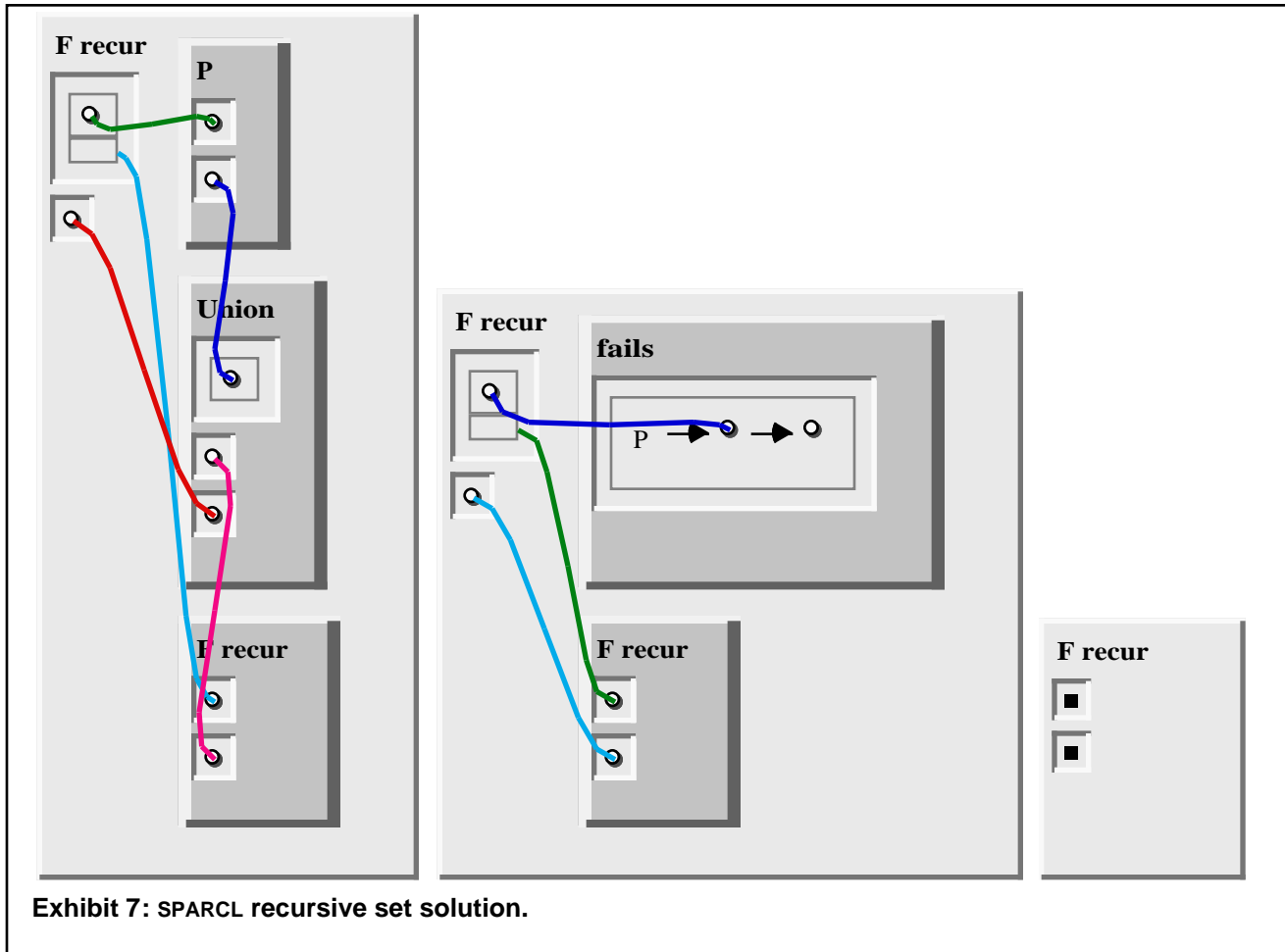


Exhibit 7: SPARCL recursive set solution.

ing of S , a pairwise disjoint cover⁴. In SPARCL the basic set representation is a partitioned set with just one part. An example of a partitioned set with two parts is given in Exhibit 6, the second argument to the unify/2 predicate. One of the parts is a set containing one element, a variable, and the other part is any set (not containing the one variable of the first part). In this example, the two part partitioned set is used to extract an element from a set via unification. This is similar to the previously mentioned use of the head/tail technique. An important difference is that any element of the set may be the one identified by the variable, whereas with the head/tail approach there is exactly one “head” element. Thus, the partitioned set technique (in a logic programming setting) provides the “membership” tool directly. This is similar to the “segment” approach occasionally used in working with lists.

Segments of a list may be specified in a PROLOG fashion by:

$$[...X, Y, ...Z, ...A],$$

where ...X, ...Z, and ...A are segments of the list, and Y is a single element. A segment is a list that fits in the given position. So, [a,b,c,d] can be unified with the segmented list producing any of the following:

X = [],	Y = a,	Z = [],	A = [b,c,d];
X = [],	Y = a,	Z = [b],	A = [c,d];
X = [],	Y = a,	Z = [b,c],	A = [d];
X = [],	Y = a,	Z = [b,c,d],	A = [];
X = [a],	Y = b,	Z = [],	A = [c,d];
X = [a],	Y = b,	Z = [c],	A = [d];
X = [a],	Y = b,	Z = [c,d],	A = [];
X = [a,b],	Y = c,	Z = [],	A = [d];
X = [a,b],	Y = c,	Z = [d],	A = [];
X = [a,b,c],	Y = d,	Z = [],	A = [].

The Logistica⁵ language is a Scheme-based pattern-matching language that provides segmenting in its pattern specification language. Segments differ from partitioning in that they are order-based and parts of a partitioning are not. Also, segments impose no additional constraints on their “contents” where parts of a partitioning must be pairwise disjoint. These differences arise naturally from the nature of the underlying concepts—lists and sets.

```

append([], X, X).

append([Head|Tail], X, [Head|CombinedTail]) :-
    append(Tail, X, CombinedTail).

union([], X, X).

union([Head|Tail], X, [Head|CombinedTail]) :-
    \+ member(Head, X),
    union(Tail, X, CombinedTail).

union([Head|Tail], X, CombinedList) :-
    member(Head, X),
    union(Tail, X, CombinedList).

```

Exhibit 8: append/3 and union/3 PROLOG programs.

Segments are surprisingly difficult to introduce into a logic programming language. One needs a constraint mechanism to fully exploit them, and many logic programming languages (most notably PROLOG) lack such a mechanism. If one doesn’t have a constraint mechanism, one needs to distinguish between “input” and “output” uses of segments (using them to access or construct a list, respectively). Similarly, full use of partitioned sets is only possible with a constraint mechanism, as is done in SPARCL.

Partitioning constraints: unification.

The introduction of sets with partitioning constraints into a logic programming language has required the creation of a new unification mechanism. Set unification is more complex than traditional “structural” unification - the performance issues require a carefully constructed algorithm for a complex problem, and the unification algorithm must be able to find multiple solutions (but no more than a minimal collection of MGUs), since there is not necessarily a single Most General Unifier. Added to these difficulties is the management of the partitioning constraints and the unification of partitioned set parts between partitioned sets. Thus, partitioned set unification has been a difficult part of the implementation of SPARCL. The SPARCL partitioned set unification algorithm and proofs of its completeness and soundness will be presented in a subsequent paper.

4. One might also have a “covering” pattern, in addition to the “partitioning” pattern discussed here. A covering would consist of parts whose union is the entire set being covered, as the partitioning does, but the parts of a covering need not be pairwise disjoint.

5. This language has been largely the work of Frank Brown. It is an interesting mixture of functional programming and logic programming with several unique features. It is particularly well suited to writing theorem provers.

```

/* union_ordered(+Set1, +Set2, ?Union)
union_ordered(+Set1, +Set2, ?Union, ?Set1Diff,
?Set2Diff)
If Set1 and Set2 are the ordered
representations of two sets, Union is unified
with the ordered representation of their
union. union_ordered/3 is not defined if Set1
or Set2 is insufficiently instantiated or not
in standard order.
Set1Diff is unified with the ordered
representation of the Set1 minus Set2.
Set2Diff is unified with the ordered
representation of the Set2 minus Set1.
*/

union_ordered(Set1, Set2, Union) :-
    union_ordered(Set1, Set2, Union,
        _, _).

union_ordered([], Set2, Set2, [], Set2).
union_ordered([H1|T1], Set2, Union,
    L1Diff, L2Diff) :-
    union_ordered_2(Set2, H1, T1, Union,
        L1Diff, L2Diff).

union_ordered_2([], H1, T1, [H1|T1],
    [H1|T1], []).
union_ordered_2([H2|T2], H1, T1, Union,
    L1Diff, L2Diff) :-
    compare(Order, H1, H2),
    union_ordered_3(Order, H1, T1, H2, T2,
        Union, L1Diff, L2Diff).

/* Note the re-ordering of L2Diff and
OtherL1Diff in the call of union_ordered_2/6,
in the first clause for union_ordered_3/8.
*/

union_ordered_3(<, H1, T1, H2, T2,
    [H1|Union], [H1|OtherL1Diff],
    L2Diff) :-
    union_ordered_2(T1, H2, T2, Union,
        L2Diff, OtherL1Diff).
union_ordered_3(=, H1, T1, _, T2,
    [H1|Union], L1Diff, L2Diff) :-
    union_ordered(T1, T2, Union,
        L1Diff, L2Diff).
union_ordered_3(>, H1, T1, H2, T2, [H2|Union],
    L1Diff, [H2|OtherL2Diff]) :-
    union_ordered_2(T2, H1, T1, Union,
        L1Diff, OtherL2Diff).

```

Exhibit 9: PROLOG union_ordered/3 program.

Partitioning constraints: an example program.

The append/2 predicate in PROLOG is a simple recursive predicate. It is shown in Exhibit 8. The analogous operation for sets is “union”. In a list-oriented language, where one represents sets as ordered lists, something like

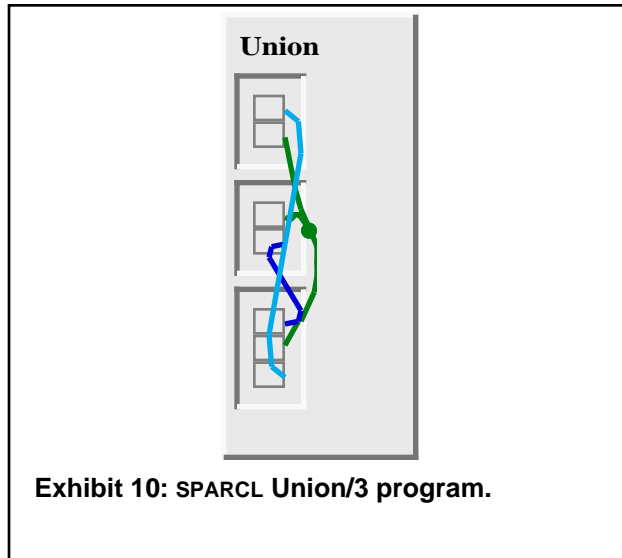


Exhibit 10: SPARCL Union/3 program.

the append/3 predicate can be used to implement union/3. The simplest version is in Exhibit 8.

This is not an efficient implementation. For efficiency one should take advantage of the ordering, and recurse down both “input” lists based on comparisons of their elements. The implementation in Exhibit 9 is based on Richard A. O’Keefe’s ord_union/3⁶.

The SPARCL predicate for Union/3 is shown in Exhibit 10. The single clause of this predicate uses the set partitioning mechanism to impose constraints such that the first two arguments are sets whose union is the third set. This predicate can be used at any point in the evaluation process, and any or none of the three arguments may be instantiated at the time this predicate is instantiated. It relies on the partitioned-set semantics to specify the union:

$$A \cup B = (A - B) \cup (B - A) \cup (A \cap B)$$

The three sets $(A - B)$, $(B - A)$, and $(A \cap B)$ taken together are a partitioning of $A \cup B$ in that they are pairwise disjoint and their union is $A \cup B$. The differences are specified via partitionings also:

$$A = (A - B) \cup (A \cap B)$$

$$B = (B - A) \cup (A \cap B)$$

6. pages 155 to 157 in [19].

The two sets $(A - B)$ and $(A \cap B)$ taken together are a partitioning of A in that they are pairwise disjoint and their union is A . Similarly, $(B - A)$ and $(A \cap B)$ are a partitioning of B .

The SPARCL Union/3 clause asserts:

$A = X \cup Y$, where X and Y are disjoint.

$B = Y \cup Z$, where Y and Z disjoint.

$C = X \cup Y \cup Z$, where X , Y , and Z disjoint.

The only solution of these three equations is:

$$X = (A - B)$$

$$Y = (A \cap B)$$

$$Z = (B - A)$$

$$C = (A - B) \cup (A \cap B) \cup (B - A) = A \cup B$$

Thus C (the set in the third argument) is the union of A and B (the first and second argument sets), as desired.

Summary

We have presented the approach of basing a logic programming language on sets. Several of the issues raised by this fundamental design decision were investigated:

- 1) the representation of sets, both in terms of the “uniqueness” issue and in terms of the variety of specialized representations that are of interest;
- 2) the representation of clauses using sets;
- 3) the approach to programming when one is processing the elements of collection, and how these approaches differ when the collection is a list or a set.

In the context of this last issue, we presented the partitioned set constraint concept, the difficulties this poses for a unification algorithm, and an example of its use.

Basing the design of a logic programming language on sets has extensive ramifications on the entire language. These ramifications lead to a programming language which though unusual is both interesting and powerful.

References

- [1] “A Visual Logic Programming Language Based on Sets and Partitioning Constraints” by Lindsey Spratt and Allen Ambler. Pages 204-208 in Proceedings 1993 IEEE Symposium on Visual Languages, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- [2] “Using 3D Tubes to Solve the Intersecting Line Representation Problem” by Lindsey Spratt and Allen Ambler. Pages 254-261 in Proceedings 1994 IEEE Symposium on Visual Languages, October 4-7, 1994. St. Louis, Missouri. IEEE Computer Society Press: Los Alamitos, CA.
- [3] *Introduction to Mathematical Logic* by Elliott Mendelson. New York, NY: D. Van Nostrand Company. 1964.
- [4] “Ueber die Unabhängigkeit des Wohlordnungssatzes vom Ordnungsprinzip” by A. Mostowski. Pages 201-252 in *Fundamenta Mathematicae*, **32**:32. 1939.
- [5] *Non-well-founded sets* by Peter Aczel. Vol. 14, Lecture Notes, Center for the Study of Language and Information, Stanford, 1988.
- [6] “{Log}: a logic programming language with finite sets” by A. Dovier, E.G. Omodeo, E. Pontelli, and G. Rossi. Pages 111-124 in Proceedings of the Eighth International Conference on Logic Programming, edited by K. Furukawa, Paris, 1991.
- [7] “Set constructors in a logic database language” by Catriel Beerli, Shamim Naqvi, Oded Shmueli, and Shalom Tsur. In *Journal of Logic Programming* 1991:10:181-232. Elsevier Science Publishing Co. 1991.
- [8] *The Gödel Programming Language* by P. M. Hill and J. W. Lloyd. 348 pages. The MIT Press:Cambridge, Massachusetts. 1994.
- [9] “Subset-Logic Programming: Application and Implementation” by Bharat Jayaraman and Anil Nair, pp.843-858 in *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, edited by Robert A. Kowalski and Kenneth A. Bowen. Cambridge, Massachusetts:MIT Press. 1988.
- [10] “Set and binary relation variables viewed as constrained objects (abstract)” by Carmen Gervet in: *Workshop on Logic Programming with Sets*, 24 June, 1993, Eugenio G. Omodeo and Gianfranco Rossi, organizers. In conjunction with *ICLP’93 - Tenth International Conference on Logic Programming*, June 21-24, 1993, Budapest, Hungary.
- [11] “Test de satisfaisabilité dans le langage de programmation en logique avec contraintes ensemblistes: CLPS” by B. Legeard and E. Legros. pp. 18-34. *Actes de JFPL*, France, May 1992.

- [12] "Constraints over homogeneous hereditarily finite sets" by Bruno Legeard, Henri Lombardi, Fabrice Ambert, and Mohamed Hibti. Pages 9-12 in:
Workshop on Logic Programming with Sets, 24 June, 1993, Eugenio G. Omodeo and Gianfranco Rossi, organizers. In conjunction with *ICLP'93 - Tenth International Conference on Logic Programming*, June 21-24, 1993, Budapest, Hungary.
- [13] "CLP(Σ *): Constraint Logic Programming with Regular Sets." by Clifford Walinsky in *ICLP'89*. 1989. pp. 181-190.
- [14] "Extensional and Intensional Sets in CLP with Intensional Negation" by Paola Bruscoli, Agostino Dovier, Eugenio G. Omodeo, Enrico Pontelli, and Gianfranco Rossi. Pages 13-16 in:
Workshop on Logic Programming with Sets, 24 June, 1993, Eugenio G. Omodeo and Gianfranco Rossi, organizers. In conjunction with *ICLP'93 - Tenth International Conference on Logic Programming*, June 21-24, 1993, Budapest, Hungary.
- [15] "Programming by Multiset Transformation" by Jean-Pierre Banâtre and Daniel Le Métayer in *Communications of the ACM*, January 1993, Vol. 36, No. 1.
- [16] "On Logic Programming with Multisets" by Steffen Hölldobler and Michael Thielscher in:
Workshop on Logic Programming with Sets, 24 June, 1993, Eugenio G. Omodeo and Gianfranco Rossi, organizers. In conjunction with *ICLP'93 - Tenth International Conference on Logic Programming*, June 21-24, 1993, Budapest, Hungary.
- [17] "Adding Abstraction to Logic Programming. The Logistic Approach" by George Tsiknis. Pages 61-64 in:
Workshop on Logic Programming with Sets, 24 June, 1993, Eugenio G. Omodeo and Gianfranco Rossi, organizers. In conjunction with *ICLP'93 - Tenth International Conference on Logic Programming*, June 21-24, 1993, Budapest, Hungary.
- [18] "The CUBE Language" by Marc A. Najork and Simon M. Kaplan. Pages 218 to 224 in *Proceedings of the 1991 IEEE Workshop on Visual Languages*, October 8-11, 1991, Kobe, Japan. IEEE Computer Society Press: Los Alamitos, CA. 1991.
- [19] *The Craft of Prolog* by Richard A. O'Keefe. MIT Press, Cambridge, MA. 1990.