A Visual Logic Programming Language Based on Sets and Partitioning Constraints.

Lindsey Spratt and Allen Ambler

Computer Science Department
University of Kansas,
Lawrence, KS, 66045

Abstract

This paper presents a new programming language named SPARCL *that has four major elements: it is a visual language, it is a logic programming language, it relies on sets to organize data, and it supports partitioning constraints on the contents of sets. It is a visual programming language in that the representation of the language depends extensively on non-textual graphics and the programming process relies on graphical manipulation of this representation. It is a logic programming language in that the underlying semantics of the language is the resolution of clauses of a Horn-like subset of first order predicate logic. It uses sets as the* only *method of combining terms to build complex terms. Finally, one may constrain a set's structure by specifying a* partitioning *into pairwise disjoint subsets.*

Rationale

Visual programming languages (VPLs) can be much easier to understand than linear text-based programming languages, since the expression of a programming task visually can more closely mirror the programmer's way of thinking than the textual expression does. This greater felicity of expression is due to the greater variety of expression available in VPLs and the intrinsically unordered nature of a two-dimensional presentation.

Sets and partitioning constraints.

There are two approaches to ways to collect together data. One is an *n-tuple* and the other is a *set*. The n-tuple is present in almost all programming languages in a variety of forms. Common forms are records, lists, arrays, and structures (a functor plus arguments). An n-tuple is an *ordered* collection of elements. A set is an *unordered* collection of elements.[1] The extensional definition of a set needs an unordered presentation of that set's elements, making the set an interesting candidate as the basic data organizing tool for a VPL. One can represent n-tuples as sets (using a particular nesting of sets containing sets), so a set-based language can express ordering.

A partition in SPARCL contains one or more elements. Each element is a subset, and all of the elements of a partition are pairwise disjoint. A partition P unifies with a set S if there exists a partition of S which unifies with P. Partitioning a set is a useful and natural way to *constrain* the structure of a set. This is a generalization of the ability in list-based languages to "abstractly" specify the first element of a list and the rest of that list (e.g., '[H|T]' in Prolog constrains H to be bound to the first element of a list and T to be bound to the rest of that list).

N-Tuples.

It is often relevant to *order* data items. In most symbolic languages programs do this by a list or "structure". For instance, LISP uses only lists, PROLOG generally implements lists by structures with structures being the more elementary construct, and many languages provide arrays and some kind of record structuring. However, it is not necessary to have any additional mechanism beyond sets in order to express ordering. Since the representation of ordering using only sets is extremely cumbersome, SPARCL provides a special representation for an ordered collection of elements. This special representation is the n-tuple, where a sequence of n elements t_1 through t_n is written "$\langle t_1, \ldots, t_n \rangle$". This general construction defines any order-dependent structure. A 3-tuple can describe a function of two arguments, with its first element the name of the function and the second and third elements of the 3-tuple corresponding to the first and second function arguments. A program may represent a list by nesting uses of a 'cons' function of two arguments. The second

1. There are two kinds of definitions of sets, extensional and intensional. An extensional definition simply enumerates the elements of the set. An intensional definition provides a property which is true of an item if and only if that item is an element of the set.

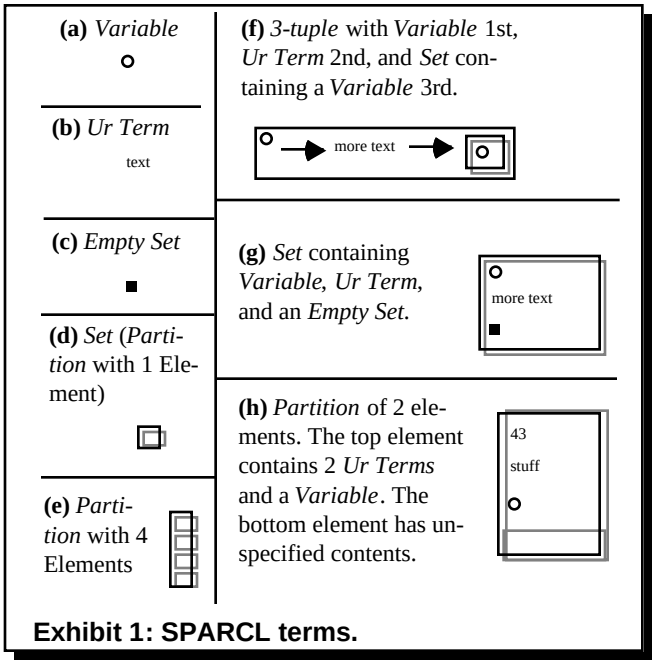

Exhibit 1: SPARCL terms.

argument is another use of the 'cons' function, or the symbol 'nil' representing the empty list. So, "⟨cons, a, b⟩" defines the function "cons(a, b)", and "⟨cons, a, ⟨cons, b, nil⟩⟩" defines a list "[a, b]".

Declarative programming.

The declarative approach to a programming language is a fundamentally unordered presentation of properties. The procedural approach is a fundamentally ordered (by control flow) presentation of computational steps. Thus, declarative paradigms potentially better suit visual languages as the underlying semantics than do the procedural paradigms. SPARCL has a logic programming semantics, and logic programming is a declarative paradigm.

Related Work

Several research areas are important to the research reported in this paper; visual languages, logic programming, unification, set unification, sets in programming languages, and visual logic programming. Ambler and Burnett [1] survey the field of visual languages. Sterling and Shapiro's *The Art of Prolog* [2] explains the logic programming paradigm in general and PROLOG in specific. Siekmann [3] summarizes the literature of universal unification, including "associative-commutative-idempotent" unification (set unification). The following subsections discuss work on sets in programming languages and on visual logic programming.

Sets in programming languages.

General-purpose programming languages which support sets include: SETL [4], REFINE [5], GAMMA [6], {Log} [7], GÖDEL [8], equational logic programming [9], and *Higraphs* [10]. In contrast with SPARCL, most of these systems do not rely on sets as the *central* method of organizing data and none of them use partitioning as a basic programming mechanism. Also, none of these systems provide a visual language environment.

Visual logic programming.

Visual programming languages that have logic programming paradigm semantics include: VLP [11], VPP [12], Mpl [13], CUBE [14], Pictorial Janus [15], and picture logic programming [16]. None of these make use of sets (or partitions of sets). Only CUBE and Pictorial Janus are completely visual environments, the others rely on linear textual presentations of code in certain situations.

There has been some work in visualizing logic programs, which is distinct from a visual programming language. The Transparent Logic Machine[17] and the AND/OR graph-based system of Senay and Lazzeri [18] are examples of this.

The Language

A SPARCL[2] *program* is a set of clauses. A *clause* is a pair where the first element (the *head*) is a literal and the second element (the *body*) is a set of literals. (Only *positive* literals are directly represented.) A *fact* is a clause with an empty body. A *rule* is a clause with a non-empty body. A *literal* is an n-tuple where the first element (the *predicate name*) is an ur element and the rest of the elements are the arguments of the literal. An *ur element* is any constant except the empty set. An *argument* is any term. A *term* is a constant, a variable, a set, a partition, or an n-tuple. A *constant* is the empty set, an alphanumeric string, or a number.

There are three parts of the concrete syntax; the terms, the programmatic elements, and the tabular extensions. For most of the non-textual elements, the visual program editor adjusts the actual size depending on what the displayed element contains—it is drawn as large as it needs to be so that the representation *visually* contains what it *semantically* contains.

The terms are shown in Exhibit 1. The interpretations of parts (a) through (c), and (f) and (g) are, we hope, self-evident from the exhibit. The "hollow" partition elements in parts (d) and (e) complicates their interpretation.

2. **S**ets and **PAR**tition **C**onstraints in **L**ogic

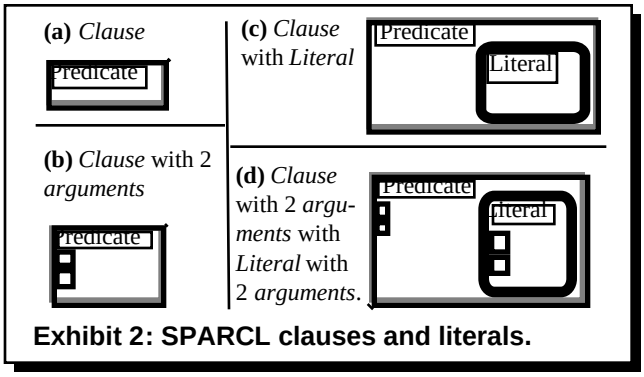

Exhibit 2: SPARCL clauses and literals.

Part (d) represents a partition of a set into one subset. Since the union of a partition's elements equals the whole set, this one subset is identical to the whole set. (The solid box represents the set in its entirety and the offset gray box represents the one partition element of that set.) Part (d) does not specify whether or not there are any members of this partition element. A partition element that is drawn without any contents (a "hollow" partition element) is a partition-element "variable", which can only unify with a set-like term. A set-like term is: a partition element, an entire partition, an empty set, an n-tuple, or a variable. If a single hollow circle were drawn in the element of (d), then the partition element would be a subset containing a single term. The hollow circle identifies this term. Part (e) is a partition with four elements (subsets). Part (e) does not specify whether or not *any* of these partition elements contains any members, since all of them are hollow.

The programmatic elements are shown in Exhibit 2. All four items are clauses. Items (a) and (b) are facts. Items (c) and (d) are rules.

Exhibit 4 illustrates co-reference. Two terms are references to the same term if a solid line connects them. Co-referring terms may look different, but they *must* unify. The same (multiple segment) line may connect any number of terms. All of the representations that a line connects must be within the same clause and they all must simultaneously unify (e.g., for three co-referring terms A, B, and C, it must be true that there exists a Most General Unifier σ such that Aσ = Bσ = Cσ).

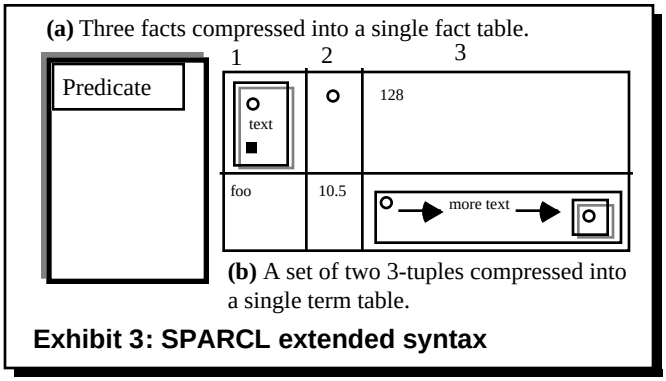

Exhibit 3: SPARCL extended syntax

Extended Syntax.

The extended concrete syntax provides tabular representations for collections of facts or terms. Exhibit 3 shows a tabular representation for a predicate definition that consists of only a set of facts (part a), and a tabular representation for a set of n-tuples (part b). SPARCL also uses a tabular representation to display both an n-tuple of n-tuples and also an n-tuple of sets. (These are not shown.)

Queries and Programs.

The set of clauses in the same window are a *program*. (All of the loaded programs (windows) are a *project*.)

The user can use the head of any clause as a query. The user places the cursor within the clause, presses the mouse button, and selects the "Query" item from the popup menu. Specifying a clause as a query means that a literal with the same structure as the head of the clause becomes the query and SPARCL invokes the interpreter with that query. After "solving" the query, the interpreter may have bound one or more variables in the query to some values. SPARCL displays this bound version of the query in the same window as the clause that was the source of the query literal.

An Example.

Exhibit 4 shows the union program. The co-reference lines or "links" attach to the right sides of the partitions. In the implemented system the links are in different colors (up to 6).

The union clause is true when the third argument is the set that is the union of the sets in the first two arguments. The connections of the partition elements of the three partition terms indicate this. A co-reference link connects the middle partition element of the third argument to an element in each of the other two arguments. Another co-reference link connects the top partition element of the first argument to the bottom partition element of the third argument. Thus, there are co-reference links for both of the partition elements of the first argument to the third argument. So, any set that unifies with the first argument is a subset of the

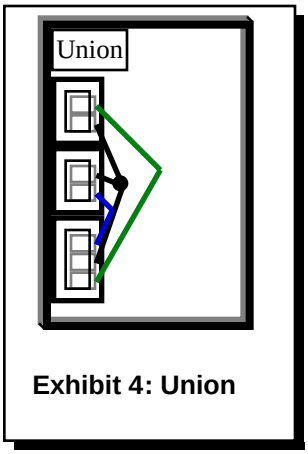

Exhibit 4: Union

third argument. Similarly, any set that unifies with the second argument is a subset of the third argument.

The middle partition element of the third argument is the intersection of the first two arguments (i.e., the largest subset consisting of members common to the first two arguments). That this is so relies on both properties of a partition—the elements form a cover of the unifying set, and the elements are pairwise disjoint subsets. The middle element of the third argument consists of members common to the first two arguments, since it co-refers with partition elements of the first two arguments.

It is the largest subset of common members due to the construction of the partition of the third argument. If a common member is *not* in the middle element, then it will be in *both* the first and third elements (since it is common). However, the elements of a partition must be pairwise disjoint, so this would violate the partition. Thus, common members can *only* be in the middle partition. Since the first and second arguments are subsets of the third argument (an earlier paragraph discusses this), then *all of* the common members must be in the third argument. Thus, *all* of the common members are in the middle element.

Semantics

The semantics of the language is basically that of a resolution theorem prover with negation-as-failure, set unification, and partition constraint satisfaction. This is analogous to (but not the same as) the semantics of Prolog.[19]

Delays

The programmer can use delay specifications to force a weak ordering on the execution of the inference procedure. The interpreter delays the attempt to determine if a literal is true if the literal satisfies the delay specification of its predicate. For instance, the system will delay the ‘is’ predicate under certain circumstances. A programmer uses the builtin ‘is’ predicate to evaluate arithmetic expressions. The interpreter unifies the first argument of the ’is’ predicate with the result of the arithmetic evaluation of the second argument. The delay specification for this predicate says to delay a literal that has a nonground second argument, i.e., one for which the arithmetic expression is not completely “filled in”. The interpreter retries delayed literals whenever there are no more undelayed literals to prove. A two-argument clause with the predicate name ‘*DELAY*’ defines a delay specification. The first argument is the predicate name of the predicate to be delayed, and the second argument is an n-tuple of keywords indicating the conditions under which to delay the predicate. There can be any number of delay specifications for the same predicate.

Unification

The interpreter determines whether or not a literal instantiation matches a clause head instantiation by trying to unify them. The process of unification may further instantiate unbound variables in either the literal or the clause head instantiation.

SPARCL has a more complex definition of unification than that which is normally used in logic programming. There are several reasons for this. The unification problem in this language involves sets, which requires associative-commutative-idempotent unification (the usual unification definition assumes none of these properties). Also this language uses partitions, which requires constraint satisfaction, the translation between sets and partition representations, and special treatment for variable partition elements.

Unification in SPARCL is slow. Adapting standard logic programming compilation techniques to handle unification should dramatically improve the speed of SPARCL.

Programming Environment

We implemented SPARCL on a Macintosh using LPA MacPROLOG version 4.5. Both the Apple Macintosh and the PROLOG language are natural choices for implementing SPARCL. Since PROLOG is a logic programming language, many of its facilities are directly useful in implementing the logic programming aspects of SPARCL. The Macintosh has a rich graphical environment that is easy to use in LPA MacPROLOG, thus easing the implementation of the graphical aspects of SPARCL. The environment uses the standard elements of a Macintosh interface; pulldown menus, windows, dialogs, and popup menus.

Conclusion

SPARCL is only a first attempt at exploring the combination of sets, partition constraints, logic programming, and a visual programming environment. The set and partition unification mechanism is very powerful, but it is also very slow. We have done some work to speed up the unification algorithm, but this is clearly an area that can use a great deal more attention.

Visual Programming.

There are some difficulties with the representation of programs in SPARCL. When the structures being repre-

sented become even mildly complex, the many nested boxes can be hard to interpret (for example, an argument box containing a set box, containing a partition element box, containing an n-tuple box, containing a set box as the first element of the n-tuple, ...). This should be a problem that we can mitigate with some careful thought about other ways to represent these different kinds of terms. Perhaps simply greater variety in the appearance of the various kinds of boxes would help. The extended concrete syntax is a step in producing a more comprehensible appearance.

The use of lines for co-referencing can be a problem. It is common in visual languages that use lines to connect things that there becomes a profusion of lines in implementations of interesting problems. This profusion is very hard for the programmer or system to manage so that they remain understandable. An important mitigating factor for SPARCL is that lines only connect elements within a clause, never between clauses. Thus, the "range" of interconnections is fairly limited compared to some visual languages that use lines to connect elements. For instance, a data flow language that connects operators by lines (output from one operator connected to the input of another operator) has no language imposed limit on the possible extent of lines within a complex program. Typically such a language handles this by allowing the programmer to use some kind of "functional" abstraction. SPARCL uses clauses to provide this abstraction.

The Future.

This is a report on work in progress. Further work is planned for all of the various aspects of SPARCL, including the formal semantics, the concrete syntax, and the implementation.

References

[1] "Influence of Visual Technology on the Evolution of Language Environments" by Allen L. Ambler and Margaret M. Burnett. In *IEEE Computer*, 22(10):9-22, October 1989.

[2] *The Art of Prolog* by Leon Sterling and Ehud Shapiro. MIT Press:Cambridge, MA. 1986.

[3] "Universal Unification" by Jörg H. Siekmann in *Proceedings of the 7th International Conference on Automated Deduction*, pp. 1-22. 1985.

[4] *The SETL2 Programming Language* by W. Kirk Snyder. Sept 9, 1990.

[5] "Research on Knowledge-Based Software Environments at Kestrel Institute" by Douglas R. Smith, Gordon B. Kotik, and Stephen J. Westfold. In *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, November 1985.

[6] "Programming by Multiset Transformation" by Jean-Pierre Banâtre and Daniel Le Métayer in *Communications of the ACM*, January 1993, Vol. 36, No. 1.

[7] "{Log}: a logic programming language with finite sets" by A. Dovier, E.G. Omodeo, E. Pontelli, and G. Rossi. In *Proceedings of the Eighth International Conference on Logic Programming*, edited by K. Furukawa, pages 111-124, Paris, 1991.

[8] *The Gödel Programming Language* by P. M. Hill and J. W. Lloyd. CSTR-92-27, Dept. of Computer Science, University of Bristol. 245 pages. 1992.

[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] "On Visual Formalisms" by David Harel. In *Communications of the ACM*, 31(5):514-530, May 1988.

[11] "VLP: A Visual Logic Programming Language" by Dider Ladret and Michel Rueher in *Journal of Visual Languages and Computing* (1991) **2**, 163-188.

[12] "Visual Logic Programming" by L. F. Pau and H. Olason in *Journal of Visual Languages and Computing* (1991) **2**, 3-15.

[13] "Mpl - a graphical programming environment for matrix processing based on logic and constraints" by Ricky Yeung, in *IEEE Workshop of Visual Languages*, pages 137-143. IEEE Computer Society Press, October 1988.

[14] "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.

[15] "Complete Visualizations of Concurrent Programs and their Executions" by Kenneth M. Kahn and Vijay A. Saraswat. Pages 7 to 15 in *Proceedings of the 1990 IEEE Workshop on Visual Languages, October 4-6, 1990, Skokie, Illinois*. IEEE Computer Society Press: Los Alamitos, CA. 1990.

[16] "Pictures Depicting Pictures: On the Specification of Visual Languages by Visual Grammars" by Bernd Meyer. Pages 41-47 in *Proceedings of the 1992 IEEE Workshop on Visual Languages September 15-18, 1992, Seattle, Washington*. IEEE Computer Society Press:Los Alamitos, CA. 1992.

[17] "The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming" by M. Eisenstadt and M. Bradshaw. In *Journal of Logic Programming*, **5** (4), 1988.

[18] "Graphical Representation of Logic Programs and Their Behavior" by Hikmet Senay and Santos G. Lazzeri. Pages 25 to 31 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] *Foundations of Logic Programming* by J. W. Lloyd. Springer-Verlag, 2nd edition, 1987.