

Seeing the Logic of Programming with Sets

by

Lindsey L. Spratt

S.B., Massachusetts Institute of Technology, 1977

M.S., University of Kansas, 1992

Submitted to the Department of Electrical
Engineering and Computer Science and the
Faculty of the Graduate School of the Uni-
versity of Kansas in partial fulfillment of
the requirements for the degree of Doctor
of Philosophy.

Dr. Allen L. Ambler
Professor in Charge

Dr. Costas Tsatsoulis

Dr. James R. Miller

Dr. J. Michael Ashley

Dr. Arthur Skidmore

Date Defended: _____

Abstract

This dissertation studies the hypothesis that visual logic programming based on sets with partitioning constraints provides a superior basis for *exploratory* programming languages. Our research program is to design and implement such a programming language with an integrated development environment, and to analyze this implementation.

The programming language we have created is named SPARCL (**S**ets and **P**artitioning **C**onstraints in **L**ogic). It has the four elements of the hypothesis: it is a *visual* language, it is a *logic programming* language, it relies entirely on *sets* to organize data (which implies that programs are organized using sets, since programs can be viewed as data in this language), and it supports *partitioning constraints* on the contents of sets. In developing this language, we invented new visual language representation techniques (the automatically laid out “smooth” hyperedge in two and three dimensions) and a new kind of unification for logic programming (the partitioned set unification algorithm).

We have evaluated SPARCL in three different ways. We examinee solutions in SPARCL of several programming problems, and compared these to solutions in LISP and PROLOG. We developed and applied software measurements for objectively analyzing these solutions across the three languages. Finally, we created a small experiment (involving seven participants) for testing the usability of the language and analyzed the data gathered from this experiment.

Our results provide modest support for the usefulness of the approach to programming language design that SPARCL embodies.

Acknowledgements

Many thanks to my wife and counselor Kim Roddis, my advisor Allen Ambler, the people at *Logic Programming Associates Inc.* who provided the MACPROLOG32 system, the *POV-Team* who provide the freeware POV-RAY, and my colleague Jennifer Leopold for reviewing early versions of SPARCL.

Table of Contents

Abstract.....	ii
Acknowledgements.....	iii
Table of Contents.....	iv
Section 1: Introducing the Project.	
Chapter 1: Introduction.....	1
Chapter 2: Related Work.....	13
Section 2: Testing the Hypothesis: Creating SPARCL.	
Chapter 3: Design Elements.....	31
Chapter 4: Partitioned Set Unification.....	92
Chapter 5: Three-dimensional Representation.....	157
Chapter 6: Implementation.....	182
Section 3: Testing the Hypothesis: Evaluating SPARCL.	
Chapter 7: Subjective Analytic Assessment of SPARCL.....	219
Chapter 8: Objective Analytic Assessment of SPARCL.....	273
Chapter 9: Usability Testing.....	306
Section 4: Closure	
Chapter 10: Future Work.....	332
Chapter 11: Conclusions.....	336
Bibliography.....	339
Appendices.	
Appendix 1: A Tutorial Introduction to SPARCL.....	356
Appendix 2: SPARCL Implementation Source Metrics.....	423
Appendix 3: Example Programs.....	427
Appendix 4: Interaction Log Data.....	439
Appendix 5: Response Time Data.....	469

Chapter 1

Introduction

This dissertation studies the hypothesis that visual logic programming based on sets with partitioning constraints provides a superior basis for *exploratory* programming languages. Our research program is to design and implement such a programming language with an integrated development environment, and to analyze this implementation. By “exploratory” programming we mean using the programming process and the programs one creates as a tool in understanding a complex problem.

The programming language we have created is named SPARCL (**S**ets and **P**artitioning **C**onstraints in **L**ogic). It has the four elements of the hypothesis: it is a *visual* language, it is a *logic programming* language, it relies entirely on *sets* to organize data (which implies that programs are organized using sets, since programs can be viewed as data in this language), 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 environment is graphical. 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, it allows the structure of a set to be constrained by a *partitioning* of that set into pairwise disjoint subsets, the union of which is equal to the whole set. The partitioning constraint is a novel programming technique we developed for this thesis.

There are several reasons supporting the plausibility of the hypothesis. We consider each of the four elements. Visual programming languages (VPLs) can be much easier to understand than linear text-based programming languages: 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. There is the additional “obvious” benefit of visual programming that to the extent that a programmer tends to use pictures (particularly diagrams) to describe her problem, the representation of a program “visually” more closely mirrors the programmer’s way of thinking than the textual representation does.

Logic programming is an approach to “declarative” programming. The ideal of

this approach to programming is to relieve the programmer of the necessity of considering *how* the program interpreter achieves the programmer's goal—the programmer need only provide a problem and a logical description of the problem domain. It is not yet possible to achieve this ideal, but strides in this direction have been made.

Sets are the simplest construct for specifying collections of data. A set does not imply an order on its members (as do lists and records), 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.

We speculate that sets are used frequently in the description of problems. Thus, the semantic precision of sets should often be useful, shortening the distance from the original problem conception to the programmatic expression of that problem. This “shorter distance” should make it easier for a person to work with their program (e.g., create, modify, debug).

Partitioning constraints are useful in abstractly specifying the structure of sets. These constraints can be used to build common set operations such as union, intersection, and difference.

We consider some of these plausibility arguments in more detail below.

Greater variety of expression. The greater variety of expression is two-fold. First, the programming language elements can be arbitrary pictures, as opposed to a fairly limited set of graphics from a character set. Second, the programming elements can be related to each other “pictorially”, instead of simply by their proximity in a string. The most common pictorial relationships are: a line drawn to connect elements, and one element's picture encompassing the picture of another element. Other possibilities for expressing relationships include corresponding colors, shapes, and shape elements (such as line thickness or fill patterns). These last techniques are particularly useful for identifying elements as having particular properties. With the great variety of expressive techniques available, a VPL can be semantically very dense. A road map is a common example of a semantically dense visual presentation of data.

Weakly ordered. The weakly ordered nature of a two dimensional presentation

means that if one sees several pictures in a single display (which don't happen to line up in rows and columns), there's no (culturally determined) order which the viewer

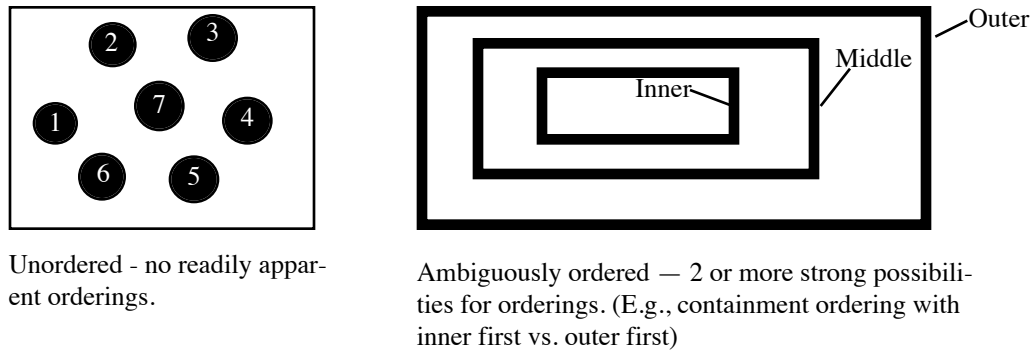


Figure 1. 1: Weak ordering examples.

assumes for these pictures. This is shown in Figure 1. 1. In contrast, when presented with rows of strings of characters, one adopts the ordering which one's culture uses for writing text (e.g., left to right and top to bottom for languages in the Indo-European family). Thus, when one has elements to present which are *not* meant to be ordered, a VPL provides a way to do this and a text-based language does not. In the case of the text based language one needs to achieve the unordering of the elements via the semantics of the language. A VPL can present orderings of elements (although not as compactly as a text-based language) via an appropriate choice of pictorial conventions (such as the use of connecting lines with arrow heads).

Declarative programming. A programming language which allows one to *declare* what a solution “looks like” (what properties a solution has) is potentially much easier to use than one which requires that one specify the *procedure* necessary to go from some initial state to a final solution state. The declarative approach to a programming language is a fundamentally unordered presentation of properties. The procedural approach is a fundamentally ordered (via control flow) presentation of computational steps. Thus, the declarative paradigm is better suited as the underlying semantics of a VPL than is the procedural paradigm, since the declarative paradigm can take advantage of the possibility of unordered displays but the procedural paradigm does not. Some of the major declarative programming paradigms are logic programming, functional programming, and data-flow programming. Logic and functional programming,

viewed as declarative programming paradigms, are essentially mathematical/notation-al variants of each other. Data-flow programming is substantially different in its organization from these other two. Of these paradigms, logic programming has not been used as the underlying paradigm for a solely-visual programming language.

Nondeterminism is closely related to declarative programming. The sense of the term “nondeterminism” is as follows: a logic program which can return multiple answers is nondeterministic - its first answer can be “failed” causing the program to “retry” and produce another answer. This is not the notion of nondeterminism common to formal language theory: A program is “strictly” nondeterministic if the output (answer) from executing that program is not necessarily the same every time that program is executed with the same inputs. If one considers executing a logic program “from the top”, then logic programs are “strictly” deterministic - they give the same results every time they are run with the same inputs. However, if one considers each “retry” of a program as a separate “execution” of that program but with the same input, then one can even say that a logic program can be “strictly” nondeterministic, since it *can* produce different results with each such retry. Nondeterminism is *not* the same thing as unpredictability - a program can be nondeterministic and yet be completely predictable.¹ Michal Walicki and Sigurd Meldal identify a close relationship between sets and nondeterminism:

“A feature shared by sets and nondeterministic operations is their natural capability of abstraction. Sets abstract from the ordering inherent in the syntax of the language and in the most common data structures. Nondeterminism abstracts from the procedural description of a (possibly deterministic) process and allows one to focus on the results produced.

The abstract character of a set makes it an essentially nondeterministic structure: the lack of an ordering of its elements does not, typically, reflect the lack of such an ordering on the concrete representations (implementations) of these elements, but merely the fact that any concrete ordering is equally acceptable. On the other hand, nondeterministic operations are related to sets in that we think of them as capable of returning any element from some set of the possible results. Also, the paradigmatic example of a nondeterministic operation is that of an arbitrary choice among the elements of some set.”²

-
1. There is a use made of nondeterminism in formal language theory which does seem related to unpredictability, though. A nondeterministic machine which has several next states from its current state, all of which use the same transition symbol, can arbitrarily (nondeterministically) choose to which of these next states to move. When doing a complexity analysis, it may be interesting to assume that the machine always chooses the next state which is most “efficient”.
 2. p. 1 of [Walicki&Meldal 1993].

Sets and partitions. 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. The set approach is rarely used in programming languages, SETL adds a set data type to an otherwise normal procedural/imperative style language and REFINE uses a set-theoretic “top-level” as a specification language which the users then “refine” into a C, COBOL, or LISP program (without sets). The data collection which needs an unordered presentation is a *set*, making it an interesting candidate as the basic data organizing tool for a VPL. Semantically, one can still represent N-tuples as sets, so ordering can be expressed in a set-based language. The idea of partitioning a set is a useful and natural way to constrain the structure of a set. This can be viewed as a generalization of sorts 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 and lists. It is often relevant to *order* data items. In most symbolic languages this is done via a list or “structure”. For instance, LISP uses only lists, PROLOG generally implements lists via 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 expression of ordering purely in terms of sets is extremely cumbersome, a special representation for ordering of elements is provided in SPARCL. This special representation is the N-tuple, where a sequence of n elements t_1 through t_n is written “ $\langle t_1, \dots, t_n \rangle$ ”. From this general construction, any order-dependent structure can be defined. A function of two arguments can be described as a 3-tuple, 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 list can be described as being built up out of nested uses of an ordered pair (a 2-tuple). Since n -tuples are recursive in their first argument, as defined above, and lists are usually considered to be recursive in their second argument, the “order” of arguments is exchanged between the list representation and the ordered-pair representation. The first argument of the ordered pair representing a list is another ordered pair

(representing the “rest” of the list), or the empty list represented by the symbol ‘nil’ (indicating that the “rest” of the list is empty). A list “[a, b]” is defined as “⟨⟨nil, b⟩, a⟩”, which can also be written as “⟨nil, b, a⟩”. Since a list always has a known (‘nil’) first argument in its N-tuple representation, relations involving ordered collections of data of indeterminate length are frequently more conveniently defined over lists instead of arbitrary n-tuples. This is due to the terminating condition for a recursive definition being easier to define for lists (since the end of the list is known to always be ‘nil’) than for n-tuples (where *any* term may be the “end” (actually first argument) of an N-tuple).

Criticisms

Each of the three major techniques that we combine faces substantial criticisms. In this section we briefly acknowledge some of these criticisms.

Visual programming languages. The graphical representations associated with visual programming languages are characterized as uniformly harder to use than corresponding textual representations by Marian Petre in [Petre 1995]. The major sense of “harder to use” here seems to be “slower to understand”. This work also identifies “secondary notation”, informal aspects of program representation layout, as crucial to improving the comprehensibility of graphical and textual programs. Jeffrey Nickerson claims that graphical representations are less “informationally dense” than textual ones in [Nickerson 1994a; Nickerson 1994b].

We believe that visual programming languages have potential for improved understandability with respect to textual languages, contrary to Petre’s position. Reasons for this include the greater variety of expression discussed earlier in this chapter and the many situations in which diagrammatic representations, which are strictly speaking only available to visual programming languages, are much more readily understood than the corresponding linguistic representations. An example of this latter point is a genealogy: a diagram of familial relationships is much easier to understand than several sentences of text describing these same relationships.

The informational density difference between graphical and textual representations is less dramatic than one might expect. We measure the sizes of various SPARCL programs and corresponding programs in LISP and PROLOG in chapter 8 (“Objective

Analysis”). The SPARCL programs turn out to be reasonably “dense” (although not as dense as the other languages). In any event, we are unconvinced that this has an important effect on the relative understandability of the two approaches.

Logic programming. Logic programming has been criticized as leading to programs which are slower than corresponding programs in other languages. Another common criticism is that logic programming languages are perceived as providing only a single search technique (e.g. backtracking depth first in PROLOG), and that this is not appropriate for many problems. Logic programming (and especially PROLOG) is criticized as requiring the programmer to use “extra” variables, since it uses a “relational” style of programming instead of “proper” functions (e.g. in PROLOG one must say “sum(X, Y, Total), product(Total, Z, Result)” instead of “product(sum(X, Y), Z)”).

There are many implementations of logic programming languages that produce very efficient programs. Creating such an implementation is a complex and difficult task, but it can be and has been done. The single-search-technique criticism confuses the behavior of a program and the behavior of the runtime package implementing the language in which that program is written. There is no need for a PROLOG program to use depth-first searching; the runtime that executes the PROLOG program may use depth-first searching, but the program itself may implement a breadth-first search or a hash-table. Relational programming does indeed use more variables when compared with functional programming for certain problems, but this is more an issue of syntax than underlying semantic model. Since the conversion from functional syntax to relational syntax is very easy, there are simple syntax preprocessors that accept functional extensions to the basic relational syntax and automatically convert it. The conversion the other direction is more complex, since only some uses of variables are “functional”. Thus, to convert the other direction one needs to analyze a particular relational expression to determine which variables are “functional” (i.e. what the domain and range of the one-to-one and onto mapping are). There may not even *be* a functional re-expression of the relation. In any event, this analysis can be quite complex. Thus, the relational expression is occasionally more awkward than a corresponding functional expression, but is also more general.

Programming with partitioned sets. There are no criticisms of programming with partitioned sets, as this is an approach first presented in this thesis. But there is a long

history of work in (unpartitioned) sets. A criticism of sets is that they are too abstract, and lead to performance problems in the implementation of set-based algorithms.

The abstract nature of sets is both a weakness and a strength. A weakness is that a complex data object can be difficult to decipher when presented as many interrelated elements among nested sets. A strength is that some very general relationships for sets, and various special kinds of sets, can be programmed once and applied to many different specific complex data structures represented using sets. We mitigate the “weakness” by providing a variety of specialized representations of sets: when a programmer sees one of these specialized representations, she immediately knows some of the constraints on the structure of the represented set (as implied by the specialized representation). We maintain the “strength” by making the various representation specializations purely *syntactic* devices: they all are transformed into sets internally, providing a uniform data structure on which programs operate. There are performance problems when working with sets, but these can be reduced by applying various optimizations: unifications can be “pre-compiled” and program transformations can be applied that use knowledge of the specialized representations.

In general. We do not claim that these three techniques are universally appropriate for solving programming problems, individually or in combination. But, we do claim that there are ways in which each approach is valuable, and that these three approaches are particularly well-suited to being used in combination with each other.

Overview of the SPARCL Project.

The SPARCL research project is presented in this thesis in three parts: the hypothesis of the project, the testing of the hypothesis, and conclusions. The testing of the hypothesis has two sub-parts: creating SPARCL and evaluating SPARCL. The sections of the thesis correspond to these parts: section 1, “Introducing the Project”, presents the hypothesis (and related material); section 2, “Testing the Hypothesis: Creating SPARCL”, and section 3, “Testing the Hypothesis: Evaluating SPARCL”, present the testing of the hypothesis; and the conclusion is in section 4.

The research contributions of this project are found in both sections 2 and 3. They include: a new unification problem and its solution, new two- and three-dimensional visual programming language representations, new two- and three-dimensional repre-

sentation techniques for connecting lines, a diagrammatic/linguistic programming language representation evaluation technique, an analytic multi-paradigm programming language comparison technique, and an approach to integrated usability testing.

The Hypothesis.

The first major objective of this research is to determine the feasibility of the SPARCL approach: Can one create a visual logic programming language based on partitioned sets? What factors limit the feasibility of the approach? The answer to this question is primarily a demonstration—show that such a language can be built by doing so. The second major objective of the proposed research is to test the hypothesis presented at the beginning of this chapter: In what way is a visual logic programming language based on sets and partitioning a superior approach to programming? Is there *any* combination of a particular programming problem domain and another approach to programming for which SPARCL is the better tool in that problem domain? There is no universally compelling method for answering such a question, and ultimately the bulk of the answer are in the form of informal arguments and reasoning from examples. A better form of the answer would be the result of testing of various approaches to visual logic programming based on sets and partitioning with various groups of potential users (e.g. professional software engineers, scientists in and out of computer science, students). However, the resources for such testing are not available. This project does do some user testing, but unfortunately not enough to be statistically significant as predicting results for a large population. Achieving the first objective allows us to approach the second one.

Feasibility. SPARCL feasibility requires effective syntax and semantics: a usable representation and a system which interprets that representation and presents the results of the interpretation to the user. The usability of a representation includes not only its static properties such as understandability but also its dynamic properties such as ease of program creation and maintainability. The representation of SPARCL and its associated integrated development environment (IDE) are essential to the feasibility question. They are also central to the assessment of the primary objective of this research project, the value of visual logic programming with partitioned sets. The interpretation of the representation requires an inference mechanism based on partitioned set unification. The most difficult part of this interpreter is the algorithm for unification of sets which honors the partitioning constraints. There is no existing solu-

tion of this unification problem, although there are solutions for closely related problems. Thus one of the contributions of this research is the invention of this unification problem and its solution. There are other aspects of the interpreter's inference mechanism which must be addressed. The inference mechanism uses Horn clause resolution. This approach is chosen because it is a relatively simple type of logic programming semantics. Since our aim is to make the language as declarative as possible, the inference mechanism should *not* introduce any ordering in its execution on which the programmer may rely. (This is in contrast to PROLOG, with its source-defined ordering of goal execution and clause searching.)

Input/Output. The effective semantics of SPARCL as described above include limited input and output capabilities: input in the form of the representation of programs and output in the form of presenting results to the user. A “complete” programming language provides input/output capabilities considerably greater than this, but extensive input/output facilities for a visual programming language in general and SPARCL in particular pose research problems beyond the scope of our primary research objectives. The design of the handling of input and output is more complicated in a visual programming languages than in linear text ones, and also the design of the semantics of input and output is troublesome for logic programming languages. Thus, the design of the handling of input and output in a visual logic programming language is doubly difficult. I/O in a visual language can mean simple linear textual I/O (say to and from an I/O stream), or it can mean reading and writing diagrams or even pictures (i.e. general images). When linear textual languages read and write they are prepared to do fairly sophisticated string parsing and generation. For instance, the `read/1` predicate in PROLOG parses the entire syntax of the PROLOG language. Thus, one can write an interpreter for PROLOG which reads PROLOG source files and prepares them for execution simply by repeated uses of the `read/1` predicate, each use reading the next clause in the file, then adding the just-read clause to the PROLOG clause database by calling `assert/1` with the term just created/read by the `read/1` predicate. Similarly, the `writeln/1` predicate writes a term out to a file in a form which can be used as a source file. LISP has similar capabilities for its I/O system. What does it mean for a visual language to have a read primitive which “parses” a source file for that language? Should a visual language have a write primitive which can be used to mimic saving a program in the editing environment? We provide some initial solutions to these questions in the con-

text of the implementation of SPARCL, but a great deal more work needs to be done in this area to produce fully satisfying solutions.

Performance. For SPARCL to be feasible it must perform reasonably well. There are two levels of performing “reasonably well” which we recognize for this research project: adequate for the user testing, and adequate for applications. In addition to there being two levels of performance, there are also two areas of performance, the performance of the user interface for working with the representation (the IDE), and the performance of the interpretation of that representation. For this project we demonstrate performance feasibility at the level which is adequate for the user testing. There are major performance obstacles to attaining any version of the second level for the interpreter, and overcoming these obstacles will require substantial research and development efforts. The performance of the IDE which is adequate for user testing is also adequate for modest applications. The performance of the interpreter hinges on the implementation of the unification algorithm. Approaches which could go a long way toward achieving the second level of performance include compilation (with emphasis on compiling “away” the unification) and partial evaluation. These are well-researched topics in the constraint logic programming field and we expect eventually to apply this research to SPARCL.

- Nickerson 1994a *Visual Programming* by Jeffrey V. Nickerson. PhD. Dissertation, Dept. of Computer Science, New York University, 1994.
- Nickerson 1994b “Visual Programming: Limits of Graphic Representation” by Jeffrey V. Nickerson. Pages 178-179 in *Proceedings 1994 IEEE Symposium on Visual Languages*, October 4-7, 1994. St. Louis, Missouri. IEEE Computer Society Press: Los Alamitos, CA.
- Petre 1995 “Why looking isn’t always seeing: readership skills and graphical programming.” by Marian Petre. Pages 33-44 in *Communications of the ACM*, **38**(6), June, 1995.
- Walicki&Meldal 1993 “Sets and Nondeterminism” by Michal Walicki and Sigurd Meldal 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.

Chapter 2

Related Work

The work related to developing a visual, partitioned-set, logic programming language is discussed below. There are several major areas: visual languages, human computer interaction, programming language design, sets in programming languages, logic programming, and programming environments. The visual languages section primarily surveys visual programming languages, organizing them by their underlying semantic model. Visual language grammar formalisms are also surveyed in this section. Human computer interaction is discussed briefly as a central issue in the design of any visual programming language. The section on sets in programming languages surveys this topic, especially sets in *logic* programming languages. The logic programming section discusses: a history of logic programming, logic programming languages, unification (especially set unification), logic program compilation with an emphasis on the Warren Abstract Machine, and the special execution control technique of delayed evaluation. The programming environment section focuses mostly on approaches to debugging.

Visual Languages.

In [Chang 1987], there are four categories of visual languages given:

1. *Languages that support visual interaction.*
2. *Visual programming languages.*
3. *Visual information processing languages.*
4. *Iconic visual information processing languages.*

The language proposed here is of the second category, a visual programming language. These four categories are derived from the combinations of two values each in two dimensions. In one dimension, the objects dealt with by a visual language can be inherently visual, or inherently nonvisual but with imposed visual representation. Along the other dimension, the programming language constructs can be visual or linear. In the case of a visual programming language, the objects dealt with are inherently nonvisual with an imposed visual representation and the programming language constructs are visual. The research being proposed has to do with “category 2” visual

languages only: *visual programming languages*.

A generalized icon is defined by [Chang 1987] as either a process or object icon. An object icon is a two part representation of an object, the logical part (the meaning) and the physical part (the image). An object may lack one or the other parts. A process icon represents an action or a computational process. In this terminology, SPARCL uses only object icons. The inference engine of SPARCL provides the implicit computation model.

Another way to categorize visual languages is by the kind of representation technique they employ. Two major approaches are box-and-line (making a graph or network) and box-only representations. The box-only representations rely on containment and relative position to convey relationships between the things represented by the boxes. The box-and-line representations rely additionally on connections made by lines to show relationships. SPARCL is a box-and-line, diagrammatic, visual language.

Visual programming languages can also be categorized according to the kind of underlying programming semantics they employ. These semantic “paradigms” can be broadly divided into procedural and declarative paradigms. The procedural paradigms include the traditional imperative programming languages (such as FORTRAN, COBOL, PL/I, ALGOL, PASCAL, and C), symbolic imperative programming languages (such as LISP), and the object-oriented programming paradigm languages (such as CLOS, SMALLTALK, and C++). The declarative paradigms include data-flow programming, functional programming, logic programming, and constraint programming (closely related to logic programming). The following section is an overview of the development of visual programming languages organized by these kinds of the programming paradigm of the underlying semantics. The procedural visual programming languages are not immediately relevant to SPARCL. The declarative ones are more relevant, and the visual logic programming languages are the most relevant of all.

Visual Programming Languages. For a survey of the history of the field of visual languages, there is [Ambler&Burnett 1989]. The history of visual programming languages begins with SKETCHPAD [Sutherland 1963]. However, SKETCHPAD was an iconic visual information processing language, not a visual programming language.

Procedural Languages. Most of the initial general purpose visual programming languages were visual extensions of preexisting linear languages. These languages were control-flow-charting systems for working with imperative-paradigm languages. Grail [Ellis et al. 1969]¹ compiled from a flow chart, with machine language statements in the boxes of the flow chart. GAL (or GRASE) [Albizuri-Romero 1984] is another control-flow-chart based language. It was represented using Nassi-Shneiderman box diagrams [Nassi&Shneiderman 1973] instead of boxes-and-lines, and was “compiled” to Pascal. PIGS [Pong&Ng 1983] uses Nassi-Shneiderman flowcharts and Pict [Glinert&Tanimoto 1984] uses box-and-line flowcharts. A visual programming language based on state transition diagrams is presented in [Jacob 1985].

Object oriented programming. Programming-by-demonstration is a more recently developed approach to specifying what a program means; the programmer “demonstrates” what the program is to do and the program development environment “observes” the demonstration, making some kind of recording of it as the program. The systems of this kind which have been developed so far have an underlying object-oriented semantics. The Rehearsal World [Finzer&Gould 1984] language is one of the most unusual visual languages. It adopts a “theater” as its metaphor for programming, with troupes of performers acting on stages according to cues. PLAY [Tanimoto&Runyan 1986] is another theater-metaphor, demonstration-oriented system. It is particularly for introducing children to programming. Another unusual programming metaphor is employed by PT (Pictorial Transformations) [Hsia&Ambler 1988], making a film (movie). In this language the programmer “demonstrates” a film - which is a sequence of pictorial transformations. ARK [Smith 1987] uses a physical-world metaphor with an underlying object-oriented execution model. It is oriented toward simulating the physical world. A strictly object-oriented language (an extension of Smalltalk) with the theater metaphor interface is Molière [Borne 1993].

A non-demonstration object oriented system is ObjectWorld [Penz 1991].

Declarative Languages. A remarkable exception to the early interest in procedural (and specifically control-flow-charting) visual languages was AMBIT/G [Christensen 1968] and AMBIT/L [Christensen 1971]. These languages represented

1. Cited in [Myers 1986]. p. 35 of [Glinert 1990].

both data and programs as directed graphs and used pattern matching and transformations on graphs as their operational methods. Many elements of these languages as described by [Rovner&Henderson 1969] would not become widely applied in visual programming environments until much later.

Constraint Languages. ThingLab [Borning 1981; Borning 1986; [Borning et al. 1987] had a constraint-based semantics. It was a conceptual descendant of SKETCHPAD. ThinkPad [Rubin et al. 1985] was a programming-by-demonstration system with a constraint-based semantics which was heavily influenced by ThingLab. It compiled into PROLOG, and debugging and output was done textually (in PROLOG).

Dataflow Languages. PROGRAPH [Pietrzykowski et al. 1983] was an early visual dataflow language.² Other dataflow languages include: Show and Tell [Kimura et al. 1986; Kimura et al. 1990], HI-VISUAL [Hirakawa et al. 1986], LabVIEW [Vose&Williams 1986], Petri Net-like Transaction Nets [Kimura 1988], apE [Ohio Supercomputer Center 1989] for supercomputer network management, Khoros [Williams&Rasure 1990], VIVA [Tanimoto 1990], VPL [Lau-Kee et al. 1991], Hyperflow [Kimura 1992], and PHF [Fukunaga et al. 1993a; Fukunaga et al. 1993b].

A special class of the dataflow languages are the spreadsheet-based “forms” languages. These include: Forms [Ambler 1987; Ambler 1990], NoPumpG [Wilde&Lewis 1990; Lewis 1990], and WYSWIC spreadsheets [Wilde 1993].

Functional Languages. Functional programming is the basis of the FFP based VisaVis [Poswig et al. 1992], and the FP based languages of [Raeder 1984] and [Borges 1990].

Logic Languages. Visual programming languages that have logic programming paradigm semantics include: “Predicates and Pixels” [Ringwood 1989], CUBE [Najork&Kaplan 1991], Pictorial Janus [Kahn&Saraswat 1990], VLP [Ladret&Rueher 1991], VPP [Pau&Olason 1991], Mpl [Yeung 1988] (which is actually a constraint logic language, based on CLP(R) [Heintze, et. al. 1992]), picture

2. PROGRAPH is still under active development and is now a successful commercial development system available on the Apple Macintosh computer.

logic programming [Meyer 1992], Visual Logic [Puigsegur et al. 1996], and PrologSpace [Yazdani&Ford 1996]

Of the various kinds of visual languages, the visual logic languages are the most directly interesting to the SPARCL project. They differ importantly from SPARCL in that only one of these make use of sets (Visual Logic) and none use partitions of sets. Also, they have very limited or nonexistent meta-programming capabilities. Visual Logic is basically a notational system, presented in [Puigsegur et al. 1996] as a “front-end” for pure PROLOG. Only CUBE and Pictorial Janus are completely visual environments, the others rely on linear textual presentations of code in certain situations. Mpl is a combination of a straight linear textual PROLOG plus a two-dimensional presentation of matrices, intermingled in the linear presentation. Mpl introduces novel pattern matching techniques for matrices which may be incorporated into SPARCL in the future.

There has been some work in visualizing logic programs, which is distinct from a visual programming language. The Transparent Logic Machine (TLM) [Eisenstadt&Brayshaw 1988] and the AND/OR graph-based system of Senay and Lazzeri [Senay&Lazzeri 1991] are examples of this. TLM offers another approach to representing logic graphically which may be useful to SPARCL.

In a somewhat different vein, Peirce presented a diagrammatic representation of logic. He proposed this as an alternative to linear textual representation of first order logic. John Sowa proposed a diagrammatic representation for higher order logic, conceptual graphs, in [Sowa 1984]. There is an effort currently under way to implement a system which uses Peirce’s diagrammatic logic in conjunction with Sowa’s. This system is more of a theorem proving environment than a programming language environment.

Production Rule Languages. There are a few visual languages which use production rules. These include ChemTrains [Bell&Lewis 1993] and BITPICT [Furnas 1991]. Bell and Lewis note that ChemTrains was strongly influenced by the OPS family of production languages [Newell 1973; Forgy 1981], these being the origin of linear-textual production rule languages. This approach is not sufficiently logic-based to be of much use to the SPARCL project.

Visual language grammar There has been some work in formal grammar systems

for specifying the syntax of visual programming languages. [Helm&Marriott 1991] is the most formal. Other works are relation grammars [Crimi et al. 1991], unification based grammars [Wittenburg et al. 1991], picture layout grammars [Golin 1991; [Golin&Magliery 1993], visual grammars [Lakin 1987], Positional Grammars [Costagliola et al. 1991; Costagliola et al. 1993], Fringe Relational Grammars [Wittenburg 1992], Conditional Set Rewrite Systems [Najork&Kaplan 1993], and Constraint Hypergraph Grammars [Minas&Viehstaedt 1993; Viehstaedt 1995]. The VAMPIRE system by McIntyre and Glinert [McIntyre&Glinert 1992] is a system for implementing iconic visual language environments, providing both syntax specification and semantics implementation tools.

We have not produced a formal grammar for SPARCL. All of the grammar formalisms for visual languages contain major textual portions, and are difficult to use in ways that express the concrete representation adopted for SPARCL in either its two or three-dimensional version.

However, we plan to pursue formal specification as an area of future work. This is an important step in presenting a formal *semantics* of SPARCL. Also, grammars and parsing may be useful techniques in handling visual input to a visual language. Finally, a simple formal grammar approach can be a very powerful method of programmer-specified extensions of the language representation. An example of this is the use of programmer-specified operator precedence grammars for Edinburgh-style PROLOG programs [Clocksin&Mellish 1992].

The Value of Visual Representations. There is a substantial literature on assessing visual representations versus textual ones, particularly with respect to programming languages.

Some of these assessments largely deprecate the value of a visual representation. The graphical representations associated with visual programming languages are characterized as uniformly harder to use than corresponding textual representations by Marian Petre in [Petre 1995]. The major sense of “harder to use” here seems to be “slower to understand”. Jeffrey Nickerson claims that graphical representations are less “informationally dense” than textual ones in [Nickerson 1994a] and [Nickerson 1994b].

The idea of “secondary notation”, informal aspects of program representation layout, is presented in [Petre 1995] as crucial to improving the comprehensibility of

graphical *and* textual programs. This seems to blur the distinction between a graphical and textual representation, since the secondary notation is generally graphical (e.g. indentation and newlines).

Shin makes a clear distinction between diagrammatic and linguistic representations in [Shin 1994], where diagrammatic representations use spatial relationships (adjacency, containment, linking) to represent semantic relationships. In Shin's approach, the use of icons is generally a linguistic representational technique, while Petre views this a graphical representation technique (using the example of programs written in LabView). Petre is vague about what a graphical representation is, but it is different from Shin's notion of a diagrammatic representation. Petre notes that "text is essentially graphics with a very limited vocabulary."³ So, in Petre's terms graphics *includes* text. This is in strong contrast with Shin's dichotomy of diagrammatic and linguistic representations. Petre occasionally uses the phrase "analog mapping" in ways that are similar to Shin's use of "diagrammatic representation."

Graphical and diagrammatic representations are either two-dimensional or three-dimensional. Much less work has been done on three-dimensional representations, much less comparing them with two-dimensional ones. One such work is [Ware&Franck 1994], which demonstrates empirically that three-dimensional representations of graphs can be much easier to understand than two-dimensional ones.

Human Interface Design.

A concern for good quality human-computer interface design is the driving force behind visual languages. The subject is *defined* by a particular approach ("visual") to human-computer interfaces. *The Art of Human-Computer Interface Design*. [Laurel 1990] is a collection of papers which discuss every aspect of this issue.

Don Norman emphasizes that one must involve the (potential) users from the inception of a programming project, rather than building a system and then trying it out on them afterwards.⁴ If one has a well-defined group of prospective users, members of this group can be brought into the design process. Another approach is to involve people who as a group are presumed to be typical of the eventual actual user group. The involvement of these people can be in different ways, including as formal mem-

3. p. 42 in [Petre 1995]

4. p. 8 in Rheingold???

bers of the design team and as members of test-groups. Norman advocates user testing at every stage of the design process. Kathleen Gomoll sketches techniques for user testing in [Gomoll 1990]. Unfortunately, this good advice on the involvement of users is difficult to act on when one has few resources with which to work.

There is some work which explicitly deals with user interface issues in visual programming. Mouse/palette-based diagram editing versus pen/gesture-based diagram editing is discussed by [Citrin 1993] and [Apte&Kimura 1993]. Matrix manipulation in a visual language versus linear text language is discussed by [Pandey&Burnett 1993].

Programming language design.

There are many issues in programming language design. The focus in this work is on the ease of use of the language, with assumption that it is worth making the computer work harder (e.g., run slower, use more storage), and to make the programming language environment implementor work harder, in order that the programmer may work less hard.

The arguments for declarative programming instead of procedural programming are arguments about the superior usability of declarative programming — that programs written in declarative programming languages are easier to write, understand, and maintain than those written in procedural programming languages, although they might run more slowly and use more storage.

How a programming language addresses mental representations is central to the usability of that language, according to [Fix et al. 1993]. In their work, they compare the performance of novices and experts on various mental-representation-oriented measures and find that the experts have a “stronger” mental representation (with regard to these measures; hierarchically structured representation, explicit mappings, recurring basic patterns, well connected representation, and well grounded). Their work is closely tied to a procedural language (Pascal), so it is difficult to directly apply to the work here. However, it seems to imply that a language which makes it easy to have high marks in these measures will be easier to use than one which does not lend itself to readily achieving such high marks.

Sets in programming languages.

The use of sets and multisets in general-purpose programming languages can be divided into logic programming languages using sets and all other kinds of languages using sets. There are relatively few non-logic programming languages which use sets. SETL [Snyder 1990] adds a set data type to an otherwise normal procedural/imperative paradigm language. STARSET [Gilula 1993] is a SETL-like language which Gilula notes allows the programmer to work with “‘pure’ sets, unlike, say, the SETL language.” REFINE [Smith et al. 1985] uses a set-theoretic “top-level” as a specification language which the users then “refine” into a C or LISP program (without sets). *Higraphs* are introduced in [Harel 1988] as a visual language technique for handling sets (and graphs).

Sets in logic programming languages. {Log} (read “set log”) [Dovier et al. 1991] adds sets to PROLOG. LDL [Beeri et al. 1991] is a database-query oriented logic programming language similar to PROLOG which has sets as “first class objects”. GÖDEL [Hill&Lloyd 1992] is a logic programming language that includes a set processing capability as an “add on” module. There is an unnamed language by [Jayaraman&Nair 1988] which combines subset processing in an equational logic programming system.

There are several approaches to extending CLP for sets: a CLP extension by Gervet [Gervet 1993], CLPS [Legeard&Legros 1992; Legeard et al. 1993], $CLP(\sum^*)$ [Walinsky 1989], and an unnamed extension by [Bruscoli et al. 1993].

Multisets are used by some logic programming languages: GAMMA [Banâtre&Métayer 1993] and the language of Hölldobler and Thielscher [Hölldobler&Thielscher 1993].

The above approaches to sets in logic programming are all based on axiomatic approaches to set theory: extend classical logic with set axioms. George Tsiknis identifies the “logistic approach” which introduces set abstraction terms into arguments by “simple” rules of deduction [Tsiknis 1993]. He presents *SetLog*, a logic programming language with set abstraction based on NaDSet* (which [Tsiknis 1993] says is derived from Gilmore’s NaDSet).

Comparison with SPARCL. In contrast with SPARCL, these systems do not rely on sets as the *central* method of organizing data, and they do not use sets in the metalan-

languages in which these systems are described. This leads these languages to have syntax and semantics that are quite different from those of SPARCL. None of these systems use partitioning as a basic programming mechanism. Also, none of these systems provide a visual language environment (except for HiGraphs).

Logic Programming.

The logic programming paradigm complements many approaches to programming. Of particular interest here are constraints, functional programming, and parallelism. There are many logic programming languages, some of which combine logic programming with some of these complementary approaches.

The main approach in logic programming languages for “implementing” logic is to base the language on a clausal form of first order predicate calculus, Horn Clauses. John Lloyd provides an excellent overview of the genesis of logic programming in [Lloyd 1987b]. He identifies [Herbrand 1930] as being the deep origins of the automated theorem proving work represented by [Prawitz 1960], [Gilmore 1960], and [Davis&Putnam 1960]. Chin-Liang Chang and Richard Char-Tung Lee in [Chang&Lee 1973] describe the development of automated theorem proving, the “ambition to find a general decision procedure to prove theorems”⁵, as originating with Leibniz then being revived by Peano (circa 1900) and Hilbert’s “school” (circa 1920). Herbrand’s paper ([Herbrand 1930]) proposed a mechanical proof procedure, which was implemented by Gilmore ([Gilmore 1960]). Gilmore’s approach was improved by Davis and Putnam ([Davis&Putnam 1960]).

J. Alan Robinson presented the crucial discovery of the *resolution* principle in [Robinson 1965]. This is a single inference rule which is both efficient and easy to implement. The early work in logic programming was largely based on resolution theorem proving. Lloyd points out that the idea of using logic *as a programming language* was presented by several people: Cordell Green ([Green 1969]), Patrick Hayes ([Hayes 1973]), Alain Colmerauer, H. Kanoui, P. Roussel, and R. Pasero ([Colmerauer et al. 1973]), and Robert Kowalski ([Kowalski 1974]). The PLANNER language of Carl Hewitt ([Hewitt 1972]) is a conceptual precursor to PROLOG in certain respects⁶. The work of Colmerauer (and associates) and Kowalski led directly

5. page xi in [Chang&Lee 1973].

6. PLANNER was never implemented. A restricted version, micro-PLANNER, was implemented.

to Prolog. Roussel, an associate of Colmerauer, wrote the first PROLOG interpreter in ALGOL-W at Marseille in 1972. The Marseille group wrote an improved version in FORTRAN in 1973 ([Battani&Meloni 1973; Roussel 1975]). The first widely used and reasonably efficient implementation of a compiling PROLOG was the DEC10 PROLOG by David H. D. Warren [Warren 1977].

There are non-resolution-based approaches to logic programming, as well as the many resolution-based approaches. Lloyd mentions the work of Kenneth Bowen in full first order logic ([Bowen 1982]), Hansson, Haridi, and Tärnlund in various logics ([Hansson et al. 1982]), and Haridi and Sahlin in natural deduction ([Haridi&Sahlin 1983]) as examples. O'Donnell uses equational logic [O'Donnell 1985].

There has been some work in logic programming languages which use some logic other than first-order predicate calculus, particularly as restricted to Horn clauses. A major area of this has been work in “disjunctive” logic programming [Lobo, et. al. 1992]. This is primarily distinguished from the Horn clause languages by a more sophisticated treatment of negation. Since “pure” PROLOG relies on negation-as-failure [Clark 1978] and the Closed World Assumption [Reiter 1978], it is actually implementing a non-monotonic logic rather than classical first order predicate calculus. The disjunctive logic programming languages pursue this notion of non-monotonic logic more thoroughly.

PROLOG is the most widely used and extensively studied logic programming language. For a good presentation of the logic programming paradigm and the PROLOG language there is [Sterling&Shapiro 1986]. The formal semantics of logic programming (with an emphasis on PROLOG) are set forth in [Lloyd 1987b]. There are many varieties of PROLOG. The PROLOG as described in [Clocksin&Mellish 1992] is the de facto standard syntax and semantics, sometimes referred to as “Edinburgh” PROLOG since it closely follows the DEC10 PROLOG implemented at the University of Edinburgh by David H. D. Warren [Warren 1977]. Major PROLOG implementations include Quintus PROLOG [Bowen et al. 1985], SICSTUS PROLOG [Carlsson&Widen 1988], NU-PROLOG [Naish 1985], and XSB PROLOG by David S. Warren at Stony Brook, SUNY.

There are many languages which are resolution-based, but not quite PROLOG. There are several concurrent execution-oriented logic programming languages (what Lloyd calls “system” languages). There are languages which are a mixture of con-

straint satisfaction and logic programming. There are languages which are a mixture of functional programming and logic programming. Some of these are: Gödel [Hill&Lloyd 1992], PARLOG [Clark&Gregory 1986], LDL [Beerli et al. 1987; Beerli et al. 1991], Datalog, CLP(R) [Jaffar&Lassez 1987], CHIP [Van Hentenryck 1989], GHC [Ueda 1986], CP [Shapiro 1983], FCP [Codognet et. al. 1990], {log} [Dovier et al. 1991], subset-logic programming [Jayaraman&Nair 1988; Jayaraman&Plaisted 1989], logic programming with sets [Kuper 1987], VLP [Ladret&Rueher 1991], VPP [Pau&Olason 1991], Mpl [Yeung 1988], and CUBE [Najork&Kaplan 1991].

Unification. There are many different approaches to and uses of unification. An extensive survey of unification is given in [Knight 1989]. Unification is a central technique in most approaches to logic programming. The basic idea of unification is to find a *substitution* which makes two *terms* equivalent. Although Herbrand presented an algorithm for unification in his 1930 doctoral thesis [Herbrand 1930], the modern work in unification largely stems from J. Alan Robinson. In [Robinson 1965], he presented unification as a central part of a new approach to automated theorem proving based on the resolution inference rule. This laid the foundations on which logic programming was later built by Colmerauer, Kowalski, and others (as described above).

A more precise statement of the unification problem is: given a term s and another term t , find a substitution σ for the variables in s and the variables in t such that the substituted version of s is equivalent to the substituted version of t . This is written as $\sigma(s) = \sigma(t)$. A substitution is a function which maps variables into terms. The *application* of a substitution σ to a term s is written $\sigma(s)$, as though σ is a function which maps terms into something. The meaning of application is: If the argument is a variable, then the application returns the mapping of that variable defined by σ . If the argument is a term which has no subterms and is not a variable, then the application returns the term. If the argument is a term which has subterms, then the application returns the term with each of its subterms “replaced” by the application of the substitution function σ to that subterm.

The notion of term has many different possible definitions. The definition of term most interesting in this thesis uses sets, constants (the empty set and ur elements (arbitrary “named” constants, including numbers)), and variables. The most common

definitions use functions, constants, and variables. Unification is closely related to equation solving, where finding a substitution for two terms is equivalent to solving for the variables in two equations.

A substitution which makes two terms equivalent is a *unifier* for those two terms. A central problem in using unification is to find a *most general unifier* (MGU). The unifier σ of two terms s and t is an MGU for those terms if for any other unifier θ there exists a substitution τ such that $\tau\sigma = \theta$. An MGU for two terms is the “simplest” unifier of those terms. For unification of normal first-order logic terms (i.e. not sets) there is always a unique MGU. However, for set unification there may be many distinct MGU’s. In this case one is interested in a set of MGU’s such that they are all mutually “independent” (i.e. for any two of the substitutions in the MGU set σ and θ there does *not* exist a substitution τ such that $\tau\sigma = \theta$ or $\sigma = \tau\theta$) and such that the set is “complete” (i.e. for any unifier θ of s and t there exists a substitution σ in the MGU set for s and t such that there exists a substitution τ such that $\tau\sigma = \theta$).

The major papers in unification algorithms consider *sound* algorithms, ones which only return true if the terms being unified actually have a unifier. Robinson’s original paper was such an algorithm. Other important work in these sound algorithms are given by [Boyer&Moore 1972; Venturini-Zilli 1975; Huet 1976; [Paterson&Wegman 1976; de Champeaux 1986; Martelli&Montanari 1976; [Martelli&Montanari 1982; Corbin&Bidoit 1983]. These algorithms are various attempts to improve the speed or space-efficiency or both with respect to [Robinson 1965].

The performance of unification-based logic programming languages is heavily influenced by the implementation of the unification algorithm. Due to this influence, there has been a lot of attention to improving the performance of unification. The approaches to improving the performance of unification have focused on three areas: finding better algorithms (this primarily has involved finding better data structures), weakening the requirements for unification, and using partial evaluation as part of compiling a logic program. The search for better algorithms is described in the previous paragraph. The notion of a “better” algorithm has largely focussed on reducing the worst case complexity of the algorithm, with little attention being paid to the size of the constant factor. Unfortunately, the linear complexity algorithms all have large constant factors which make them unattractive in practice since most uses of unifica-

tion are simple.

The weakening of the requirements for unification has been achieved by removing the “occurs check” from the algorithm. Removing the occurs check means that the algorithm is no longer sound, but it is unsound in a way which a careful programmer can avoid by properly structuring her programs. The gain in performance is potentially quite large—a unification algorithm of quadratic complexity can be reduced to a linear algorithm, without an increase in the constant factor.

Compiling Logic Programs - The Warren Abstract Machine (WAM) D. H. D. Warren’s “discovery” of a way to compile PROLOG programs (and logic programs in general) was the crucial step in making PROLOG implementations which are sufficiently efficient to be used for general purpose programming. The central element of the compilation technique is an abstract machine called the WAM (Warren Abstract Machine) which provides an instruction set tailored to the needs of executing PROLOG, but which is a fairly normal sequential, imperative language. The initial explanation of the compilation technique is given in [Warren 1977]. A somewhat refined version of the WAM is explained in [Warren 1983]. Hassan Aït-Kaci provides a gentler, yet complete, presentation of the WAM in [Aït-Kaci 1991]. One of the efficiencies of compilation of PROLOG is in the ability to “compile away” invocation of fully-general unification.⁷ The general unification algorithm of the WAM is UNION/FIND method of [Aho et al. 1974]. The “compiling away” is done by partially evaluating the unification algorithm at compile-time for each argument in the head of each clause in the program, instead of waiting to run the general unification at run-time. This also makes it possible to use clause-indexing when the clauses have mutually exclusive patterns in their head arguments. Typically, clause-indexing is only done for the first argument of the predicate, and the only exclusivity which is recognized is if the clauses contain different atoms in their first arguments.

There have been many extensions of the basic WAM. The ANLWAM (Argonne National Laboratory WAM) [Butler et al. 1986] which provided and-parallelism, or-parallelism, occurs-check, compiling units on the fly, and indexing to handle large sets of units. The Aquarius PROLOG system is an effort to produce a very high performance WAM, the Berkeley Abstract Machine (BAM) [Van Roy&Despain 1992].

7. Other efficiencies are the elimination of search of the clause base in certain circumstances and the elegant handling of storage.

Micha Meier in [Meier 1993] mentioned that his group had actually found that they got better performance implementing their logic programming language in PROLOG, rather than implementing their own (modified) WAM. His reasoning was that by using PROLOG as their target language, they benefited from ongoing research to improve the performance of PROLOG without making any changes in the implementation of their language. When they built their own WAM they did not benefit from the research of others on WAM technology, unless they made design changes to their implementation of the WAM. Thus, he advocated implementing logic programming languages in PROLOG, rather than building special purpose versions of the WAM.

Certainly to start with, the implementation of SPARCL will (continue to) be in PROLOG. Meier's comments indicate that a very compelling analysis should be done prior to implementing a WAM.

Set Unification A summary of the literature on the unification of sets ("associative-commutative-idempotent-identity matching" or "ACI") is found in [Siekmann 1984]. Huet discusses the mathematically related problem of generating the basis of solutions to homogeneous linear diophantine equations in [Huet 1978]. Unification for associative-commutative functions is presented in [Stickel 1975]. An algorithm for general set unification is presented in PROLOG in [Stolzenburg 1993]. This algorithm has a worst case exponential complexity. Stolzenburg mentions work in ACI unification in [Baader&Büttner 1988] and [Kapur&Narendran 1993], and AC unification in [Lincoln&Christian 1988] and [Kapur&Narendran 1992]. Eric Domenjoud analyzes the minimal number of unifiers of an interesting special case of AC unification in [Domenjoud 1992].

There is another approach to set theory which eliminates the Axiom of Foundation - that uses an Anti-foundation Axiom. Peter Aczel presents such a theory in [Aczel 1988]. A unification algorithm for such sets is discussed in [Aliffi et al. 1993].

Stolzenburg's algorithm for set unification did not prove useful as a basis for the unification algorithm for SPARCL. Although it is relatively fast and concise, the performance cost of doing the analysis for the special case where it could be applied more than consumed the performance advantage it offered for the special case of "simple set unification". The set theory of SPARCL is that of von Neumann, Bernays, and Gödel (NBG) plus the Axiom of Choice as presented in [Mendelson 1964], but there are some intriguing possibilities in working with Aczel's non-well-founded sets

(which allow a finite representation of a certain class of infinite sets).

Delayed evaluation/Coroutining. The technique of delaying evaluation of some predicates according to a *delay* or *wait* declaration can greatly increase performance over the simple source-ordering of goals, and is essential with an inference engine which does not recognize any programmer-specified ordering of goals. Lee Naish presents the *wait* mechanism for MU-Prolog in [Naish 1985]. The main thrust of Naish's paper is the *automation* of adding "wait" declarations to a program, so as to increase its efficiency and termination. Since the wait declarations are automatically added to the program, the programmer does not have to be aware of this mechanism, much less understand the rationale for when these wait declarations should be used.

Programming Development Environments

There are many aspects of developing programs which are collectively the programming development environment. These aspects can be divided into the static and dynamic aspects. Primary among the static aspects is the presentation of the program being developed. Dynamic aspects include everything having to do with the running of the program under development, such as handling input and output and debugging. The presentation of the program is an essential part of this research effort - diagrammatic (visual) representation of a program. Some of the issues around visual programming languages are touched on in the "Visual Languages" section above.

Debugging Debugging in logic programming languages is done in various ways. The major two approaches are execution-model oriented debugging and abstract algorithmic debugging. In PROLOG, the most common approach to debugging is an execution-model oriented debugger known as an interactive stepper. The interactive stepper implemented within the Edinburgh DEC10 PROLOG [Bowen et al. 1982] "has formed the basis for all subsequent PROLOG environments."⁸ This interactive stepper is based on the *Byrd box model* of PROLOG execution as presented by Lawrence Byrd in [Byrd 1980]. Dave Plummer extends this box model with Coda (Clause Oriented Debugging Aid) in [Plummer 1988]. An early alternative to the interactive stepper box model was part of the Transparent Prolog Machine of

8. p. 497 of [Plummer 1988]

[Eisenstadt&Brayshaw 1986; Eisenstadt&Brayshaw 1988]. Plummer claims that Coda is a more declarative debugger than the Byrd box model because the program trace is presented at the level of the clause being executed instead of the goal being proved. The TPM provides a completely declarative debugger by modeling the execution of a PROLOG program as “a search for a proof by growing a search tree”⁹ and the debugger displays this tree (at user-controlled level of detail).

Execution tracing in non-dataflow declarative visual languages (described as “term rewriting languages” by [Poswig et al. 1993]) is done for the logical Pictorial Janus [Kahn&Saraswat 1990] and functional VisaVis [Poswig et al. 1993] with a “re-play” or post-runtime approach. Execution tracing in dataflow and control flow visual languages:

...are very fond of live animation. To depict dataflow they use highlighting of the currently executed functions or the connections between functions. They pop up new windows or modify sub-structures to show control flow.¹⁰

Poswig et alii insist that live animation is inappropriate for concurrently executing languages. Their reasons for this belief seem related to the fact that systems which use live animation highlight a single part (the currently executing part) at a time, and contrary to this belief it seems that a system can highlight multiple parts of a running program nearly as easily as it can highlight a single element (thus displaying concurrent execution of the multiple highlighted elements).

Ehud Shapiro introduced abstract algorithmic debugging in [Shapiro 1982]. This is defined as:

Algorithmic Debugging is a theory of debugging that uses queries on the compositional semantics of a program in order to localize bugs. It uses the following principle: if a computation of a program’s component gives an incorrect result, while all the subcomputations it invokes compute correct results, then the code of this component is erroneous.¹¹

This work was applied to (sequential) PROLOG, as were several subsequent works ([Plaisted 1986; Pereira 1986; Sterling&Shapiro 1986; Dershowitz&Lee 1987; [Lloyd 1987a]). There are also several works applying abstract algorithmic debugging to concurrent languages (Flat Guarded Horn Clauses (FGHC) [Lloyd&Takeuchi 1986; Takeuchi 1986], PARLOG [Huntbach 1987], Flat Concurrent Prolog (FCP) [Lichtenstein&Shapiro 1988a; Lichtenstein&Shapiro 1988b]). The work in [Lichtenstein&Shapiro 1988b] is applicable to concurrent languages in the functional programming paradigm as well as the logic programming paradigm.

9. p. 504 of [Plummer 1988].

10. p.181 in [Poswig et al. 1993].

11. p. 513 of [Lichtenstein&Shapiro 1988b].

Additional References:

Walicki&Meldal 1993 (Ch. 1) [Walicki&Meldal 1993]
Mendelson 1964 (Ch. 3) [Mendelson 1964]
Spratt&Ambler 1993 (Ch 3) [Spratt&Ambler 1993]
Hanus 1995 (Ch. 4) [Hanus 1995]
Apple Computer, Inc. 1995 (Ch. 5) [Apple Computer, Inc. 1995]
Foley et al. 1990 (Ch. 5) [Foley et al. 1990]
Quinlan 1982 (Ch. 5) [Quinlan 1982]
Spratt&Ambler 1994 (Ch 5) [Spratt&Ambler 1994]
Wernecke 1994 (Ch. 5) [Wernecke 1994]
Cohen&Feigenbaum 1982 (Ch. 7, pt 2) [Cohen&Feigenbaum 1982]
Hunt et al. 1966 (Ch. 7, pt 2) [Hunt et al. 1966]
Ernst&Newell 1969 (Ch. 7, pt 3) [Ernst&Newell 1969]
Fikes&Nilsson 1971 (Ch. 7, pt 3) [Fikes&Nilsson 1971]
Green 1969 (Ch. 7, pt 3) [Green 1969]
Rich&Knight 1991 (Ch. 7, pt 3) [Rich&Knight 1991]
Sussman 1975 (Ch. 7, pt 3) [Sussman 1975]
Warren 1974 (Ch. 7, pt 3) [Warren 1974]
Abelson et al. 1985 (Ch. 7, pt 4) [Abelson et al. 1985]
van Harmelen 1989 (Ch. 7, pt 4) [van Harmelen 1989]
Bieman et al. 1991 (Ch. 8) [Bieman et al. 1991]
Bollman&Cherniavsky 1981 (Ch. 8) [Bollman&Cherniavsky 1981]
Briand et al. (Ch. 8) [Briand et al. 1996]
Conte et al. 1986 (Ch. 8) [Conte et al. 1986]
Fenton 1991a (Ch. 8) [Fenton 1991a]
Fenton 1991b (Ch. 8) [Fenton 1991b]
Finkelstein 1984 (Ch. 8) [Finkelstein 1984]
Kitchenham et al. 1990 (Ch. 8) [Kitchenham et al. 1990]
Krantz et al. 1971 (Ch. 8) [Krantz et al. 1971]
Lambert 1990 (Ch. 8) [Lambert 1990]
Sethi 1989 (Ch. 8) [Sethi 1989]
Zuse 1996 (Ch. 8) [Zuse 1996]
Zuse&Bollman 1987 (Ch. 8) [Zuse&Bollman 1987]

Bratko 1990 (Ch. 9) [Bratko 1990]
Carroll&Rosson 1995 (Ch. 9) [Carroll&Rosson 1995]
Compeau at al. 1995 (Ch. 9) [Compeau et al. 1995]
Kay&Thomas 1995 (Ch. 9) [Kay&Thomas 1995]
O'Keefe 1990 (Ch. 10) [O'Keefe 1990]
Steele 1990 (Ch. 10) [Steele 1990]

Bibliography

- Abelson et al. 1985 *Structure and Interpretation of Computer Programs* by Harold Abelson and Gerald Jay Sussman with Julie Sussman. The MIT Press, Cambridge, Massachusetts. 1985.
- Aczel 1988 *Non-well-founded sets* by Peter Aczel. Vol. 14, Lecture Notes, Center for the Study of Language and Information, Stanford, 1988.
- Aho et al. 1974 *The Design and Analysis of Computer Algorithms* by Alfred Aho, John Hopcroft, and Jeffrey Ullmann. Addison-Wesley, Reading, MA, 1974.
- Aït-Kaci 1991 *Warren's Abstract Machine: a Tutorial Reconstruction*. By Hassan Aït-Kaci. The MIT Press:Cambridge, Massachusetts. 1991.
- Albizuri-Romero 1984 "GRASE—A Graphical Syntax-Directed Editor for Structured Programming" by Miren B. Albizuri-Romero in *SIGPlan Notices*. **19**(2) February. 1984. pp. 28-37.
- Aliffi et al. 1993 "Unification of hyperset terms" by Davide Aliffi, Gianfranco Rossi, Agostino Dovier, and Eugenio G. Omodeo. Pages 27-30 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.
- Ambler 1987 "Forms: Expanding the Visualness of Sheet Languages" by Allen Ambler in *Proceedings of the 1987 Workshop on Visual Languages*, TryckCenter, Linkoping, Sweden, Aug. 1987. pp. 105-117.
- Ambler 1990 "Generalizing the Sheet Language Paradigm" by Allen Ambler in *Visual Languages and Applications*, T. Ichikawa, E. Jungert, and R. R. Korfhage, Eds., Plenum Press. 1990.
- Ambler&Burnett 1989 "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.
- Apple Computer, Inc. 1995 *3D Graphics Programming With QuickDraw 3D*, Apple Computer, Inc. Addison-Wesley Publishing Company:Reading, Massachusetts. 1995.
- Apte&Kimura 1993 "A Comparison Study of the Pen and the Mouse in Editing Graphic Diagrams" by A. Apte and T. D. Kimura. Pages 352-359 in:
Proceedings 1993 IEEE Symposium on Visual Languages, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- Baader&Büttner 1988 "Unification in Commutative Idempotent Monoids" by Franz Baader and Wolfram Büttner in *Theoretical Computer Science*, **56** (1988). pp. 345-353.
- Banâtre&Métayer 1993 "Programming by Multiset Transformation" by Jean-Pierre Banâtre and Daniel Le Métayer in *Communications of the ACM*, January 1993, Vol. 36, No. 1.
- Battani&Meloni 1973 "Intrepreteur du Language de Programmation PROLOG" by G. Battani and H. Meloni. Report of Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, 1973.
- Beeri et al. 1987 "Set and negation in a Logic Database Language (LDL1)" by C. Beeri, S. Naqvi et alia., *Proceedings 6th ACM SIGMOD Symposium*, 1987.
- Beeri et al. 1991 "Set constructors in a logic database language" by Catriel Beeri, Shamim Naqvi, Oded Shmueli, and Shalom Tsur. In *Journal of Logic Programming* 1991:10:181-232. Elsevier Science Publishing Co. 1991.

- Bell&Lewis 1993 “ChemTrains: A Language for Creating Behaving Pictures” by Brigham Bell and Clayton Lewis. Pages 188-195 in:
Proceedings 1993 IEEE Symposium on Visual Languages, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- Bieman et al. 1991 “Moving from Philosophy to Practice in Software Measurement” by James Bieman, Norman Fenton, David Gustafson, Austim Melton, and Robin Whitty. Pages 38-59 in *Formal Aspects of Measurement: Proceedings of the BCS-FACS Workshop on Formal Aspects of Measurement, South Bank University, London, 5 May 1991* edited by Tim Denvir, Ros Herman, and R. W. Whitty. Springer-Verlag: London. 1991. Austin Austin
- Bollman&Cherniavsky 1981 “Measurement-Theoretical Investigation of the MZ-Metric.” by P. Bollmann and V.S. Cherniavsky. In *Information Retrieval Research*, R.N. Oddy, S.E. Robertson, C.J. van Rijsbergen, P.W. Williams (ed.), Butterworth, 1981.
- Borges 1990 *Multiparadigm Visual Programming Languages* by J. A. Borges, Ph. D. Thesis, Univ. of Illinois at Urbana-Champaign, 1990.
- Borne 1993 “A Visual Programming Environment for Smalltalk” by Isabelle Borne. Pages 214-218 in:
Proceedings 1993 IEEE Symposium on Visual Languages, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- Borning 1981 “The Programming Languages Aspects of ThingLab, A Constraint Oriented Simulation Laboratory” by Alan Borning in *ACM Transactions on Programming Languages and Systems*, **3** (4), October, 1981. pp. 353-387. Reprinted on pages 416-449 in:
Visual Programming Environments: Paradigms and Systems, edited by Ephraim P. Glinert. IEEE Computer Society Press: Los Alamitos, CA. 1990.
- Borning 1986 “Graphically Defining New Building Blocks in ThingLab” by Alan Borning in *Human Computer Interaction*, **2**(4), 1986, pp. 269-295. Reprinted on pages 450-469 in:
Visual Programming Environments: Paradigms and Systems, edited by Ephraim P. Glinert. IEEE Computer Society Press: Los Alamitos, CA. 1990.
- Borning et al. 1987 “Constraint Hierarchies” by Alan H. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, and M. Woolf from *ACM Proceedings OOPSLA*, 1987, pp. 48-60. Reprinted on pages 470-482 in:
Visual Programming Environments: Paradigms and Systems, edited by Ephraim P. Glinert. IEEE Computer Society Press: Los Alamitos, CA. 1990.
- Bowen 1982 Programming with Full First-Order Logic by Kenneth A. Bowen in *Machine Intelligence 10*, Ellis Horwood, Chichester, 1982, pp. 421-440.
- Bowen et al. 1982 *DEC-10 PROLOG Users Manual* by David L. Bowen, Lawrence Byrd, Fernando C. N. Pereira, Luiz M. Pereira, and David H. D. Warren. Occasional Paper 27, DAI, University of Edinburgh, November 1982.
- Bowen et al. 1985 *Quintus Prolog Reference Manual* by David L. Bowen, Lawrence Byrd, D. Ferguson, and W. Kornfeld. Quintus Computer Systems, Inc., May, 1985.
- Boyer&Moore 1972 “The sharing of structure in theorem proving programs” by R. S. Boyer and J. S. Moore. *Machine Intelligence 7*.
- Bratko 1990 *PROLOG programming for Artificial Intelligence, 2nd Edition* by Ivan Bratko. Addison-Wesley Publishing Company: Reading, Massachusetts. 1990.

- Briand et al. 1996 “On the Application of Measurement Theory in Software Engineering” by Lionel Briand, Khaled El Emam, and Sandro Morasca. International Software Engineering Research Network technical report #ISERN-95-04. To appear in *Empirical Software Engineering: An International Journal*, 1(1), Kluwer Academic Publishers, 1996.
- Bruscoli et al. 1993 “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.
- Butler et al. 1986 “Paths to high-performance automated theorem proving” by R. Butler, E. L. Lusk, W. McCune, and R. A. Overbeek. In *Proceedings of the 8th International Conference on Automated Deduction, Lecture Notes in Computer Science, Vol. 230*, J. Siekmann, ed., pages 588-597, New York, 1986. Springer-Verlag.
- Byrd 1980 “Understanding the control flow of PROLOG programs” by Lawrence Byrd. Pages 127-138 in *Proceedings of the Logic Programming Workshop* by S. Tärnlund, editor. 1980.
- Carlsson&Widen 1988 *SICStus Prolog user's manual* by Mats Carlsson and Johan Widen. Research Report 88007B, Swedish Institute of Computer Science, Kista, Sweden, 1988.
- Carroll&Rosson 1995 “Managing evaluation goals for training” by John M. Carroll and Mary Beth Rosson, pages 40-48 in *Communications of the ACM* 38(7), July 1995.
- Chang 1987 “Visual Languages: A Tutorial and Survey” by Shi-Kuo Chang. In *IEEE Software*, 4(1):29-39, January, 1987.
- Chang&Lee 1973 *Symbolic Logic and Mechanical Theorem Proving* by Chin-Liang Chang and Richard Char-Tung Lee. Academic Press, Inc.: Orlando, Florida. 1973.
- Chang&Lee 1973 *Symbolic Logic and Mechanical Theorem Proving* by Chin-Liang Chang and Richard Char-Tung Lee. Academic Press, Inc.: Orlando, Florida. 1973.
- Christensen 1968 “An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language” by Carlos Christensen in *Interactive Systems for Experimental Applied Mathematics*, Melvin Klerer and Juris Reinfelds, eds. New York: Academic Press, 1968. pp 423-435.
- Christensen 1971 “An Introduction to AMBIT/L, A Diagrammatic Language for List Processing” by Carlos Christensen in *Proceedings of the 2nd Symposium on Symbolic and Algebraic Manipulation*. Los Angeles, CA. Mar 23-25, 1971. pp. 248-260.
- Citrin 1993 “Requirements for Graphical Front Ends for Visual Languages” by Wayne Citrin. Pages 142-151 in:
Proceedings 1993 IEEE Symposium on Visual Languages, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- Clark 1978 “Negation as failure” by Kenneth Clark in *Logic and Data Bases* by H. Gallaire and J. Minker (eds). Plenum Press, New York, 1978. pp. 293-322.
- Clark&Gregory 1986 “PARLOG: A Parallel Logic Programming Language” by Kenneth Clark and S. Gregory, in *ACM Trans. on Prog. Lang. and Systems* 8, 1 (Jan. 1986), pp. 1-49.
- Clocksin&Mellish 1992 *Programming in Prolog* by Clocksin and Mellish. Third edition. 1992.

- Codognet et. al. 1990 "Abstract Interpretation for Concurrent Logic Languages" by Christian Codognet, Philippe Codognet, and Marc-Michel Corsini. pages 215 - 232 in *Logic Programming: Proceedings of the 1990 North American Conference*, edited by Saumya Debray and Manuel Hermenegildo. MIT Press : Cambridge, MA. 1990.
- Cohen&Feigenbaum 1982 *Handbook of Artificial Intelligence, Vol. III* edited by Paul R. Cohen and Edward A. Feigenbaum. William Kaufmann, Inc: Los Altos, California. 1982.
- Colmerauer et al. 1973 *Un Systeme de Communication Homme-Machine en Francais* by Alain Colmerauer, H. Kanoui, P. Roussel, and R. Pasero. A report? of the Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille, 1973.
- Compeau et al. 1995 "End-user training and learning" by Deborah Compeau, Lorne Olfman, Maung Sei, and Jame Webster in *Communications of the ACM* **38**(7), July, 1995.
- Conte et al. 1986 *Software Engineering Metrics and Models* by S. D. Conte, H. E. Dunsmore, and V. Y. Shen. The Benjamin/Cumming Publishing Company, Inc.:Menlo Park, California. 1986.
- Corbin&Bidoit 1983 "A rehabilitation of Robinson's unification algorithm" by J. Corbin and M. Bidoit. In *Information Processing 83*, pp. 73-79.
- Costagliola et al. 1991 "A Generalized Parser for 2-D Languages" by Gennaro Costagliola, Masaru Tomita, and Shi-Kuo Chang. Pages 98-104 in: *1991 IEEE Workshop on Visual Languages*. 1991.
- Costagliola et al. 1993 "Automatic Parser Generation for Pictorial Languages" by G. Costagliola, S. Orefice, G. Polese, G. Tortora, and M. Tucci. Pages 306-313 in: *Proceedings 1993 IEEE Symposium on Visual Languages*, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- Crimi et al. 1991 "Relation grammars and their application to multidimensional languages" by C. Crimi, A. Guercio, G. Nota, G. Pacini, G. Tortora, and M. Tucci in *Journal of Visual Languages and Computing* (1991) **2**, 333-346.
- Davis&Putnam 1960 "A Computing Procedure for Quantification Theory" by M. Davis and Hilary Putnam in *J. ACM* **7** (1960), pp. 201-215.
- de Champeaux 1986 "About the Paterson-Wegman linear unification algorithm." by D. de Champeaux. In *Journal of Computing System Science* **32**, pp. 79-90. 1986.
- Dershowitz&Lee 1987 "Logical Debugging" by N. Dershowitz and Y.-J. Lee in *Proc. 4th Symposium on Logic Programming*, San Francisco, September 1987.
- Domenjoud 1992 "A Technical Note on AC-Unification. The Number of Minimal Unifiers of the equation $\alpha x_1 + \dots + \alpha x_p =_{AC} \beta y_1 + \dots + \beta y_q$ " by Eric Domenjoud in *Journal of Automated Reasoning* **8** (1992), pp. 39-44.
- Dovier et al. 1991 "{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.
- Eisenstadt&Brayshaw 1986 The Transparent PROLOG Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming by M. Eisenstadt and M. Brayshaw. Technical Report TR-21, Human Cognition Research Laboratory, Open University, Milton Keynes, UK, 1986. To appear in the Journal of Logic Programming.
- Eisenstadt&Brayshaw 1988 "The Transparent Prolog Machine (TPM): an execution model and

- graphical debugger for logic programming” by M. Eisenstadt and M. Brayshaw. In *Journal of Logic Programming*, **5** (4), 1988.
- Ellis et al. 1969 “The Grail Project: An Experiment in Man-Machine Communication” by T. O. Ellis, J. F. Heafner, and W. L. Sibley. RAND Report RM-5999-Arpa. 1969.
- Ernst&Newell 1969 *GPS: A case study in generality and problem solving*. by G. Ernst and A. Newell. New York: Academic Press, 1969.
- Fenton 1991a “Software Measurement: Why a Formal Approach?” by Norman Fenton. Pages 3-27 in *Formal Aspects of Measurement: Proceedings of the BCS-FACS Workshop on Formal Aspects of Measurement, South Bank University, London, 5 May 1991* edited by Tim Denvir, Ros Herman, and R. W. Whitty. Springer-Verlag: London. 1991.
- Fenton 1991b *Software Metrics: A Rigorous Approach* by Norman E. Fenton. Chapman & Hall. 1991.
- Fikes&Nilsson 1971 “STRIPS: A new approach to the application of theorem proving to problem solving” by R. E. Fikes and Nils J. Nilsson. Pages 189-208 in *Artificial Intelligence*, **2**(3-4), 1971.
- Finkelstein 1984 “A review of the fundamental concepts of measurement” by L. Finkelstein in *Measurement*, Vol. 2(1) 1984, 25-34.
- Finzer&Gould 1984 “Programming by Rehearsal” by William Finzer and Laura Gould in *BYTE Magazine*, June 1984. Reprinted on pages 356-366 in: *Visual Programming Environments: Paradigms and Systems*, edited by Ephraim P. Glinert. IEEE Computer Society Press: Los Alamitos, CA. 1990.
- Fix et al. 1993 “Mental representations of programs by novices and experts” by Vikki Fix, Susan Wiedenbeck, and Jean Scholtz. Pages 74-79 in *Interchi '93: Conference on Human Factors in Computing Systems (INTERACT'93 and CHI'93)*, Amsterdam, The Netherlands, 24-29 April 1993. Stacey Ashland, Kevin Mullet, Austin Henderson, Erik Hollnagel, Ted White, Editors.
- Foley et al. 1990 *Computer Graphics Principles and Practice* by James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. Addison-Wesley, Reading, Massachusetts, 1990.
- Forgy 1981 “OPS5 Users Manual” by C. Forgy. Technical Report CMU-CS-81-135. Computer Science Department, Carnegie Mellon University, Pittsburgh. 1981.
- Fukunaga et al. 1993a “Object -Oriented Development of a Data Flow Visual Language System” by Alex S. Fukunaga, Takayuke D. Kimura, and Wolfgang Pree. Pages 134-141 in: *Proceedings 1993 IEEE Symposium on Visual Languages*, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- Fukunaga et al. 1993b “Functions as Data Objects in a Data Flow Based Visual Programming Language” by Alex S. Fukunaga, Takayuke D. Kimura, and Wolfgang Pree in *Proceedings of the ACM Computer Science Conference*, Indianapolis, IN. 1993.
- Furnas 1991 “New graphical reasoning models for understanding graphical interfaces” by G. Furnas in *Proceedings of CHI'91*. 1991. pp. 71-78.
- Gervet 1993 “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.

- Gilmore 1960 “A Proof Method for Quantification Theory” by P. C. Gilmore in *IBM J. Res. Develop.* **4** (1960), pp. 28-35.
- Gilula 1993 “Dealing with ‘Pure’ Sets in STARSET Language” by Mikhail M. Gilula. Pages 43-46 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.
- Glinert 1990 *Visual Programming Environments: Paradigms and Systems* by Ephraim P. Glinert (editor). IEEE Computer Society Press, Los Alamitos, CA. 1990.
- Glinert&Tanimoto 1984 “Pict: An Interactive Graphical Programming Environment” by Ephraim P. Glinert and Steven L. Tanimoto in *IEEE Computer*. **17**(11) Nov. 1984. pp. 7-25.
- Golin 1991 “Parsing visual languages with picture layout grammars” by E. J. Golin in *Journal of Visual Languages and Computing* (1991) **2**, 371-394.
- Golin&Magliery 1993 “A Compiler Generator for Visual Languages” by Eric J. Golin and Tom Magliery. Pages 314-321 in:
 Proceedings 1993 IEEE Symposium on Visual Languages, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- Gomoll 1990 “Some Techniques for Observing Users” by Kathleen Gomoll. Pages 85-90 in:
 The Art of Human-Computer Interface Design edited by Brenda Laurel. Addison-Wesley Publishing Company, Inc. Reading, MA. 1990.
- Green 1969 “Applications of Theorem Proving to Problem Solving” by Cordell Green in *IJCAI-69*, Washington, D.C., 1969, pp. 219-239.
- Green 1969 “Application of Theorem Proving to Problem Solving” by Cordell Green. Originally published as pages 219-239 in *Proceedings of the First International Joint Conference on Artificial Intelligence*, Washington, DC, 1969. Also published as pages 202-222 in *Readings in Artificial Intelligence*, edited by B. L. Webber and N. J. Nilsson, Los Altos, California: Morgan Kaufmann, 1981.
- Hansson et al. 1982 “Properties of a Logic Programming Language” by Å. Hansson, S. Haridi, and S.-Å. Tärnlund, in *Logic Programming*, Kenneth L. Clark and S.-Å. Tärnlund (eds), Academic Press, New York, 1982, pp. 267-280.
- Hanus 1995 “Analysis of Residuating Logic Programs” by Michael Hanus in *The Journal of Logic Programming*, vol. 24, n. 3, September 1995.
- Harel 1988 “On Visual Formalisms” by David Harel. In *Communications of the ACM*, **31**(5):514-530, May 1988.
- Haridi&Sahlin 1983 “Evaluation of Logic Programs based on Natural Deduction” by S. Haridi and D. Sahlin, TRITA-CS-8305 B, Royal Institute of Technology, Stockholm, 1983.
- Hayes 1973 “Computation and Deduction” by Patrick J. Hayes in *Proc. MFCS Conference.*, Czechoslovak Academy of Sciences, 1973, pp. 105-118.
- Heintze, et. al. 1992 *The CLP(R) Programmer’s Manual, Version 1.2* by Nevin C. Heintze, Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. This manual is distributed with version 1.2 of CLP(R) by Joxan Jaffar at the IBM Thomas J. Watson Research Center.

This manual is dated September 1992.

- Helm&Marriott 1991 “A Declarative Specification and Semantics for Visual Languages” by Richard Helm and Kim Marriott in *Journal of Visual Languages and Computing* (1991) **2**, 311-331.
- Herbrand 1930 “Investigations in Proof Theory” by J. Herbrand, originally published in 1930, in *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, van Heijenoort, J. (ed.), Harvard University Press, Cambridge, Mass., 1967, pp. 525-581.
- Hewitt 1972 “Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot” by Carl Hewitt, A.I. Memo 251, MIT, 1972.
- Hill&Lloyd 1992 *The Gödel Programming Language* by P. M. Hill and J. W. Lloyd. CSTR-92-27, Dept. of Computer Science, University of Bristol. 245 pages.
- Hirakawa et al. 1986 “HI-VISUAL Iconic Programming” by M. Hirakawa, I. Hoshimoto, M. Tanaka, T. Ichikawa, and S. Iwata in *Proc. IEEE Workshop on Visual Languages*, pp. 34-43, 1986.
- Hölldobler&Thielscher 1993 “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.
- Hsia&Ambler 1988 “Programming through Pictorial Transformations” by Y.-T. Hsia and Allen Ambler in *Proceedings of the 1988 IEEE International Conference on Computer Languages*, October. 1988, IEEE CS Press, Los Alamitos, CA, Order No. FJ874. pp. 10-16.
- Huet 1976 “Resolution d’équations dans les langages d’ordre 1, 2, ..., ω .” by G. Huet. Ph.D. dissertation. Université de Paris VII, France. 1976.
- Huet 1978 “An Algorithm to Generate the Basis of Solutions to Homogeneous Linear Diophantine Equations” by Gérard Huet, *Rapport de Recherche* 274, January 1978. Institut de Recherche d’Informatique et d’Automatique, Le Chesnay, France.
- Hunt et al. 1966 *Experiments in Induction* by E. B. Hunt, J. Marin, and P. J. Stone. New York: Academic Press. 1966.
- Huntbach 1987 “Algorithmic Parlog Debugging” by M. Huntbach in *Proc. 4th Symposium on Logic Programming*, San Francisco, September 1987.
- Jacob 1985 “A state transition diagram language for visual programming” by R. J. K. Jacob in *IEEE Computer* **18**(8), 1985. pp. 51-59.
- Jaffar&Lassez 1987 “Constraint logic programming.” by Joxan Jaffar and Jean-Louis Lassez, in *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111-119. ACM, January 1987.
- Jayaraman&Nair 1988 “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.
- Jayaraman&Plaisted 1989 “Programming with Equations, Subsets and Relations.” by B. Jayara-

- man and D.A. Plaisted. *Proceedings of NACLPL89 (North American Conference on Logic Programming, 1989)*, Cleveland, 1989.
- Kahn&Saraswat 1990 “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.
- Kapur&Narendran 1992 “Double-Exponential Complexity of Computing a Complete Set of AC-Unifiers” by Deepak Kapur and Paliath Narendran in *Proceedings of the 7th Annual Symposium on Logic in Computer Science*, Santa Cruz, CA, 1992, Ewing Lusk and Ross Overbeek, editors. pp. 11-21.
- Kapur&Narendran 1993 “A Unification Algorithm for Associative-Commutative-Idempotent Theories” by Deepak Kapur and Paliath Narendran. (Preliminary Report) Albany NY: State University of New York at Albany, Department of Computer Science, Institute for Programming and Logics, April 1993.
- Kay&Thomas 1995 “Studying long-term system use.” by Judy Kay and Richard C. Thomas. Pages 61-69 in *Communications of the ACM* **38**(7), July 1995.
- Kimura 1988 “Visual Programming by Transaction Network” by Takayuki Dan Kimura in *IEEE Proceedings of the 21st Hawaii International Conference On System Sciences (HICSS-21)*, Volume 2: Software Track, 1988, pp. 648-654.
- Kimura 1992 “Hyperflow: A Visual Programming Language for Pen Computers” by T. D. Kimura. In: *1992 IEEE Workshop on Visual Languages*. 1992.
- Kimura et al. 1986 “A Visual Language for Keyboardless Programming” by Takayuki Dan Kimura, J. W. Choi, and J. M. Mack. Technical Report WUCS-86-6, Department of Computer Science, Washington University, St. Louis, MO., March 1986.
- Kimura et al. 1990 “Show and Tell: A Visual Programming Language” by Takayuki Dan Kimura, Julie W. Choi, and Jane M. Mack. Pages 397-404 in: *Visual Programming Environments: Paradigms and Systems*, edited by Ephraim P. Glinert. IEEE Computer Society Press: Los Alamitos, CA. 1990.
- Kitchenham et al. 1990 “An evaluation of some design metrics” by B. Kitchenham, L. Pickard, and S. J. Linkman in *Software Engineering Journal* Vol 5(1), 1990, 50-58.
- Knight 1989 “Unification: A Multidisciplinary Survey” by Kevin Knight. Pages 93-124 in *ACM Computing Surveys*, Vol. 21, No. 1, March 1989.
- Kowalski 1974 “Predicate Logic as a Programming Language” by Robert A. Kowalski in *Information Processing 74*, Stockholm, North Holland, 1974, pp. 569-574.
- Krantz et al. 1971 *Foundations of Measurement, Vol. 1* by D. H. Krantz, R. D. Luce, P. Suppes, and A. Tversky. Academic Press. 1971.
- Kuper 1987 “Logic Programming with Sets.” by G. M. Kuper. *Proceedings 6th ACM SIGMOD Symposium*, 1987.
- Ladret&Rueher 1991 “VLP: A Visual Logic Programming Language” by Dider Ladret and Michel Rueher in *Journal of Visual Languages and Computing* (1991) **2**, 163-188.
- Lakin 1987 “Visual grammars for visual languages” by F. Lakin. In: *Proceedings of AAAI-87*, pp.

683-688.

- Lambert 1990 *Neues Organon I* by Johann Heinrich Lamber. Berlin: Akademie Verlag, 1990.
- Lau-Kee et al. 1991 “VPL: An Active, Declarative Visual Programming System” by David Lau-Kee, Adam Billyard, Robin Faichney, Yasuo Kozato, Paul Otto, Mark Smith, and Ian Wilkinson. Pages 40-46 in:
1991 IEEE Workshop on Visual Languages. 1991.
- Laurel 1990 *The Art of Human-Computer Interface Design* edited by Brenda Laurel. Addison-Wesley Publishing Company, Inc. Reading, MA. 1990.
- Legeard et al. 1993 “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.
- Legeard&Legros 1992 “Test de satisfaisabilité dans le lagage de programmation en logique avec contraintes ensemblistes: CLPS” by B. Legeard and E. Legros. pp. 18-34. *Actes de JFPL*, France, May 1992.
- Lewis 1990 “NoPumpG: Creating Interactive Graphics with Spreadsheet Machinery” by Clayton Lewis. Pages 526-546 in:
Visual Programming Environments: Paradigms and Systems, edited by Ephraim P. Glinert. IEEE Computer Society Press: Los Alamitos, CA. 1990.
- Lichtenstein&Shapiro 1988a “Concurrent Algorithmic Debugging” by Y. Lichtenstein and Ehud Shapiro in *Proceeding of ACM SIGPOS/SIGPLAN Workshop on Distributed and Parallel Debugging*, Madison, May 1988.
- Lichtenstein&Shapiro 1988b “Abstract Algorithmic Debugging” by Yossi Lichtenstein and Ehud Shapiro in *Logic Programming: Proceedings of the Fifth International Conference and Symposium* (University of Washington, Seattle) by Robert A. Kowalski and Kenneth A. Bowen, editors. Two volumes. The MIT Press, Cambridge, MA. 1988.
- Lichtenstein&Shapiro 1988b “Abstract Algorithmic Debugging” by Yossi Lichtenstein and Ehud Shapiro in *Logic Programming: Proceedings of the Fifth International Conference and Symposium* (University of Washington, Seattle) by Robert A. Kowalski and Kenneth A. Bowen, editors. Two volumes. The MIT Press, Cambridge, MA. 1988.
- Lincoln&Christian 1988 “Adventures in Associative-Commutative Unification (A Summary)” by Patrick Lincoln and Jim Christian in *Proceedings of the 9th International Conference on Automated Deduction*, Argonne, Illinois, USA, May 1988. pp. 358-367. (LNCS 310) New York: Springer, 1988.
- Lloyd 1987a “Declarative Error Diagnosis “ by John W. Lloyd in *New Generation Computing* **5**, pp. 133-154, 1987.
- Lloyd 1987b *Foundations of Logic Programming* by J. W. Lloyd. Springer-Verlag, 2nd edition, 1987.
- Lloyd&Takeuchi 1986 “A Framework for Debugging GHC” by John W. Lloyd and A. Takeuchi. Technical Report TR-186, Institute for New Generation Computer Technology, Tokyo, 1986.
- Lobo, et. al. 1992 *Foundations of Disjunctive Logic Programming* by Jorge Lobo, Jack Minker,

- and Arcot Rajasekar. Cambridge, MA: MIT Press. 1992.
- Martelli&Montanari 1976 “Unification in linear time and space: A structured presentation” by A. Martelli and U. Montanari. in *Internal Rep. No. B76-16*, Ist di Elaborazione delle Informazioni, Consiglio Nazionale delle Ricerche, Pisa, Italy. 1976.
- Martelli&Montanari 1982 “An efficient unification algorithm” by A. Martelli and U. Montanari. In *ACM Transactions on Programming Language Systems 4*.
- McIntyre&Glinert 1992 “Visual Tools for Generating Iconic Programming Environments” by David W. McIntyre and Ephraim P. Glinert in ???[maybe *Journal of Visual Languages and Computing*] (IEEE). 1992. pp. 162-168.
- Meier 1993 “Subject: Re: WAM modifications” by Micha Meier (“From: micha@ecrc.de (Micha Meier)”). USENET news message sent to the comp.lang.prolog newsgroup, dated Mon, 29 Nov 1993 10:04:15 GMT.
- Mendelson 1964 *Introduction to Mathematical Logic* by Elliott Mendelson. New York, NY: D. Van Nostrand Company. 1964.
- Meyer 1992 “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.
- Minas&Viehstaedt 1993 “Specification of Diagram Editors Providing Layout Adjustment with Minimal Change” by M. Minas and G. Viehstaedt. Pages 324-329 in: *Proceedings 1993 IEEE Symposium on Visual Languages*, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- Myers 1986 “Visual Programming, Programming by Example, and Program Visualization: A Taxonomy” by Brad A. Myers in *Conference Proceedings, CHI '86: Human Factors in Computing Systems*, 1986, pp. 59-66. Reprinted on pp. 33-40 of *Visual Programming Environments: Paradigms and Systems* by Ephraim P. Glinert (editor). IEEE Computer Society Press, Los Alamitos, CA. 1990.
- Naish 1985 “Automating control for logic programs” by Lee Naish. In *The Journal of Logic Programming* vol. 2, no. 3, October 1985.
- Najork&Kaplan 1991 “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.
- Najork&Kaplan 1993 “Specifying Visual Languages with Conditional Set Rewrite Systems” by Marc A. Najork and Simon M. Kaplan, pages 12-18 in: *Proceedings 1993 IEEE Symposium on Visual Languages*, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- Nassi&Shneiderman 1973 “Flowchart Techniques for Structured Programming” by I. Nassi and B. Shneiderman in *SIGPLAN Notices*. 8(8) August. 1973. pp. 12-26.
- Newell 1973 “Production Systems: Models of Control Structures” by Alan Newell in *Visual Information Processing*, W. Chase, editor., pp. 463-526. Academic Press, New York. 1973.
- Nickerson 1994a *Visual Programming* by Jeffrey V. Nickerson. PhD. Dissertation, Dept. of Computer Science, New York University, 1994.

- Nickerson 1994b “Visual Programming: Limits of Graphic Representation” by Jeffrey V. Nickerson. Pages 178-179 in *Proceedings 1994 IEEE Symposium on Visual Languages*, October 4-7, 1994. St. Louis, Missouri. IEEE Computer Society Press: Los Alamitos, CA.
- O'Donnell 1985 *Equational Logic as a Programming Language* by Michael J. O'Donnell. Cambridge, MA: MIT Press. 1985.
- O'Keefe 1990 *The Craft of Prolog* by Richard A. O'Keefe. MIT Press, Cambridge, MA. 1990.
- Ohio Supercomputer Center 1989 “apE: Providing Visualization Tools for a Statewide Supercomputing Network” by the Ohio Supercomputer Center in *Proc. 24th Cray Users Group Meeting*, pp. 237-241, Trondheim, 1989.
- Pandey&Burnett 1993 “Is it Easier to Write Matrix Manipulation Programs Visually or Textually? An Empirical Study” by R. K. Pandey and Margaret M. Burnett. pages 344-351 in: *Proceedings 1993 IEEE Symposium on Visual Languages*, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- Paterson&Wegman 1976 “Linear unification” by M. S. Paterson and M. N. Wegman. In *Proceedings of the Symposium on the Theory of Computing*. ACM Special Interest Group for Automata and Computability (SIGACT). 1976. Also published (with changes?) in *Journal of Computing System Science* 16, pp. 158-167. 1978.
- Pau&Olason 1991 “Visual Logic Programming” by L. F. Pau and H. Olason in *Journal of Visual Languages and Computing* (1991) 2, 3-15.
- Penz 1991 “Visual Programming in the Object World” by F. Penz in *Journal of Visual Languages and Computing*, 2 (1). pages 17-41. 1991.
- Pereira 1986 “Rational Debugging in Logic Programming” by Luiz M. Pereira in *Proc. 3rd International Conference on Logic Programming*, LNCS 225, pp. 203-210, Springer-Verlag, 1986.
- Petre 1995 “Why looking isn't always seeing: readership skills and graphical programming.” by Marian Petre. Pages 33-44 in *Communications of the ACM*, 38(6), June, 1995.
- Pietrzykowski et al. 1983 “The Programming Language PROGRAPH: Yet Another Application of Graphics” by T. Pietrzykowski, Stanislaw Matwin, and Tomasz Muldner in *Graphics Interface '83*, Edmonton, Alberta, May 9-13, 1983. pp. 143-145.
- Plaisted 1986 “An Efficient Bug Location Algorithm” by D. A. Plaisted in *2nd International Logic Programming Conference*, pp. 151-157, Uppsala, 1984.
- Plummer 1988 “Coda: An Extended Debugger for PROLOG” by Dave Plummer, pages 496-511. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium* (University of Washington, Seattle) by Robert A. Kowalski and Kenneth A. Bowen, editors. Two volumes. The MIT Press, Cambridge, MA. 1988.
- Plummer 1988 “Coda: An Extended Debugger for PROLOG” by Dave Plummer, pages 496-511. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium* (University of Washington, Seattle) by Robert A. Kowalski and Kenneth A. Bowen, editors. Two volumes. The MIT Press, Cambridge, MA. 1988.
- Plummer 1988 “Coda: An Extended Debugger for PROLOG” by Dave Plummer, pages 496-511. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium* (University of Washington, Seattle) by Robert A. Kowalski and Kenneth A. Bowen, editors. Two volumes. The MIT Press, Cambridge, MA. 1988.

- Pong&Ng 1983 “PIGS--A System for Programming with Interactive Graphical Support” by M. C. Pong and N. Ng in *Software--Practice and Experience*. **13**(9) September. 1983. pp. 847-855.
- Poswig et al. 1992 “VisaVis - Contributions to Practice and Theory of Highly Interactive Visual Languages” by Jörg Poswig, Klaus Teves, Guido Vrankar, and Claudio Moraga in ??? [maybe *Journal of Visual Languages and Computing*] (IEEE). 1992. pp. 155-161.
- Poswig et al. 1993 “Interactive Animation of Visual Program Execution” by Jörg Poswig, Guido Vrankar, and Claudio Moraga. Pages 180-187 in:
Proceedings 1993 IEEE Symposium on Visual Languages, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- Poswig et al. 1993 “Interactive Animation of Visual Program Execution” by Jörg Poswig, Guido Vrankar, and Claudio Moraga. Pages 180-187 in:
Proceedings 1993 IEEE Symposium on Visual Languages, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- Prawitz 1960 “An Improved Proof Procedure” by D. Prawitz in *Theoria* **26** (1960), pp. 102-139.
- Puigsegur et al. 1996 “A Visual Logic Programming Language” by Jordi Puigsegur, Jaume Agustí, and Dave Robertson. Pages 214-221 in *Proceedings 1996 IEEE Symposium on Visual Languages*, September 3-6, 1996, Boulder, Colorado.
- Quinlan 1982 "Induction of Decision Trees" by J. R. Quinlan. Pages 81-106 in *Machine Learning*, vol. 1, no. 1, edited by R. S. Michalski, T. M. Mitchell, and J. Carbonell. Palo Alto, California: Tioga. 1982.
- Raeder 1984 *Programming in Pictures* by G. Raeder, Ph. D. Thesis, Univ. of Southern California, L. A., California. 1984.
- Reiter 1978 “On closed world data bases.” by Raymond Reiter in *Logic and Data Bases* by H. Gallaire and J. Minker (eds). Plenum Press, New York, 1978. pp. 55-76.
- Rich&Knight 1991 *Artificial Intelligence, Second Edition* by Elaine Rich and Kevin Knight. McGraw-Hill, Inc.:New York. 1991.
- Ringwood 1989 “Predicates and Pixels” by G. Ringwood in *New Generation Computing*, **7**, 1989, pp. 59-70.
- Robinson 1965 “A Machine-oriented logic based on the resolution principle.” by J. A. Robinson in *Journal of the ACM* **12**, pp. 23-41.
- Roussel 1975 *PROLOG: Manuel de Reference et d'Utilization* by P. Roussel. Report of the Groupe d'Intelligence Artificielle, Université Aix-Marseille, 1975.
- Rovner&Henderson 1969 “On the Implementation of AMBIT/G: A Graphical Programming Language” by P. D. Rovner and D. A. Henderson, Jr., in *Proceedings of the International Joint Conference on Artificial Intelligence*. Washington, D. C. May 7-9, 1969.
- Rubin et al. 1985 “Think Pad: A Graphical System for Programming by Demonstration” by R. V. Rubin, E. J. Golin, and S. P. Reiss in *IEEE Software*, **2**(2), March 1985. pp. 73-79.
- Senay&Lazzeri 1991 “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.

- Sethi 1989 *Programming Languages: Concepts and Constructs* by Ravi Sethi. 478 pages. Reading, Massachusetts: Addison-Wesley Publishing Co. 1989.
- Shapiro 1982 *Algorithmic Debugging* by Ehud Shapiro. The MIT Press, Cambridge, MA. 1982.
- Shapiro 1983 “A Subset of Concurrent Prolog and its Interpreter” by Ehud Shapiro. Technical Report TR-003, ICOT, Tokyo, 1983.
- Shin 1994 *The Logical Status of Diagrams* by Sun-Joo Shin. 197 pages. Cambridge, England:Cambridge University Press. 1994.
- Siekmann 1984 “Universal Unification” by Jörg H. Siekmann in *Proceedings of the 7th International Conference on Automated Deduction*. Springer-Verlag:New York. 1984. (Newer version was to appear in the *Journal of Symbolic Computing* special issue on unification, C. Kirchner, Ed., 1988.)
- Smith 1987 “Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic” by Randall B. Smith in *IEEE Computer Graphics and Applications*, September 1987. pp. 42-50. Reprinted on pages 388-396 of: *Visual Programming Environments: Paradigms and Systems*, edited by Ephraim P. Glinert. IEEE Computer Society Press:Los Alamitos, CA. 1990.
- Smith et al. 1985 “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.
- Snyder 1990 *The SETL2 Programming Language* by W. Kirk Snyder. Sept. 9, 1990.
- Sowa 1984 *Conceptual Structures - Information Processing in Mind and Machine* by John A. Sowa. Reading MA: Addison-Wesley. 1984.
- Spratt&Ambler 1993 “A Visual Logic Programming Language Based on Sets and Partitioning Constraints” by Lindsey L. Spratt and Allen L. 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.
- Spratt&Ambler 1994 “Using 3D tubes to solve the intersecting line representation problem” by Lindsey L. Spratt and Allen L. Ambler. Pages 254-263 in *Proceedings 1994 IEEE Symposium on Visual Languages*, October 4-7, 1994. St. Louis, Missouri. IEEE Computer Society Press: Los Alamitos, CA.
- Steele 1990 *Common LISP* by Guy Steele, Second edition. Digital Press. 1990.
- Sterling&Shapiro 1986 *The Art of Prolog* by Leon Sterling and Ehud Shapiro. MIT Press:Cambridge, MA. 1986.
- Stickel 1975 “A Complete Unification Algorithm for Associative-Commutative Functions” by M. E. Stickel in *Proceedings of the IJCAI*, Tblisi. pp. 71-76, 1975.
- Stolzenburg 1993 “An Algorithm for General Set Unification and its Complexity” by Frieder Stolzenburg 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.
- Sussman 1975 “A Computer Model of Skill Acquisition” by G. J. Sussman. Cambridge, MA:MIT Press. 1975.

- Sutherland 1963 “Sketchpad: a man-machine graphical communication system” by I.E. Sutherland from AFIPS Conference Proceedings, Spring Joint Computer Conference, 1963, pages 2-19. Reprinted on pages 198-215 of *Visual Programming Environments: Paradigms and Systems*, edited by Ephraim P. Glinert. IEEE Computer Society Press: Los Alamitos, California. 1990.
- Takeuchi 1986 “Algorithmic Debugging of GHC Programs and its implementation in GHC” by A. Takeuchi. Technical Report TR-185, Institute for New Generation Computer Technology, Tokyo, 1986. Also, Chapter 26 in *Concurrent Prolog: Collected Papers* by Ehud Shapiro (ed), MIT Press. 1987.
- Tanimoto 1990 “Towards a Theory of Progressive Operators for Live Visual Programming Environments” by Steven Tanimoto in *Proc. IEEE Workshop on Visual Languages*, October, 1990.
- Tanimoto&Runyan 1986 “PLAY: An Iconic Programming System for Children” by Steven L. Tanimoto and Marcia S. Runyan in *Visual Languages*, S. K. Chang, T. Ichikawa, and P. A. Ligomenides (eds), 1986. pp. 191-205. Reprinted on pages 367-377 in: *Visual Programming Environments: Paradigms and Systems*, edited by Ephraim P. Glinert. IEEE Computer Society Press: Los Alamitos, CA. 1990.
- Tsiknis 1993 “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.
- Ueda 1986 “Guarded Horn Clauses” by K. Ueda. Ph.D. Thesis, University of Tokyo, 1986.
- van Harmelen 1989 “A classification of meta-level architectures” by Frank van Harmelen. Pages 105-122 in *Meta-Programming in Logic Programming* edited by Harvey Abramson and M. H. Rogers. The MIT Press: Cambridge, Massachusetts. 1989.
- Van Hentenryck 1989 *Constraint Satisfaction in Logic Programming* by Pascal Van Hentenryck. The MIT Press, Cambridge, Massachusetts. 1989.
- Van Roy&Despain 1992 “High Performance logic programming with the Aquarius Prolog compiler.” by Peter Van Roy and Alvin M. Despain. In *Computer* **25**, 1 (Jan. 1992), 54-68.
- Venturini-Zilli 1975 “Complexity of the unification algorithm for first-order expressions.” by M. Venturini-Zilli. Research Report: Consiglio Nazionale Delle Ricerche Istituto per le applicazioni del calcolo, Rome, Italy. 1975.
- Viehstaedt 1995 *A Generator for Diagram Editors* by Gerhard Viehstaedt. Ph. D. Dissertation. Universität Erlangen-Nürnberg. 1995.
- Vose&Williams 1986 “LabVIEW: Laboratory Virtual Instrument Engineering Workbench” G. M. Vose and G. Williams in *BYTE*, pp. 84-92, September 1986.
- Walicki&Meldal 1993 “Sets and Nondeterminism” by Michal Walicki and Sigurd Meldal 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.
- Walinsky 1989 “CLP(Σ *): Constraint Logic Programming with Regular Sets.” by Clifford Walinsky in *ICLP’89*. 1989. pp. 181-190.
- Ware&Franck 1994 “Viewing a graph in a virtual reality display is three times as good as a 2D

- diagram” by Colin Ware and Glenn Franck. Pages 182-183 in *Proceedings 1994 IEEE Symposium on Visual Languages*, October 4-7, 1994. St. Louis, Missouri. IEEE Computer Society Press: Los Alamitos, CA. Colin
- Warren 1974 “WARPLAN - a system for generating plans.” by David H. D. Warren. DAI, Memo 76. University of Edinburgh. 1974.
- Warren 1977 “Implementing Prolog—Compiling Logic Programs 1 and 2” by David H. D. Warren. *DAI Research Report 39* and *DAI Research Report 40*, University of Edinburgh, 1977.
- Warren 1983 “An abstract Prolog instruction set.” by David H. D. Warren. *Technical Note 309*, SRI International, Menlo Park, CA, October 1983.
- Wernecke 1994 *The Inventor Mentor* by Josie Wernecke (Open Inventor Architecture Group). Addison-Wesley Publishing Company: Reading, Massachusetts. 1994.
- Wilde 1993 “A WYSIWYC (What You See Is What You Compute) Spreadsheet” by Nicholas Wilde. Pages 72-76 in: *Proceedings 1993 IEEE Symposium on Visual Languages*, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- Wilde&Lewis 1990 “Spreadsheet-Based Interactive Graphics: From Prototype to Tool” by N. Wilde and C. Lewis in *Proceedings of CHI’90, Human Factors in Computing Systems*, ACM, New York, 1990. pp. 153-159.
- Williams&Rasure 1990 “A Visual Language for Image Processing” by C. S. Williams and J. R. Rasure in *Proc. IEEE Workshop on Visual Languages*, October 1990.
- Wittenburg 1992 “Early-style Parsing for Relational Grammars” by Kent Wittenberg, pp. 192-199 in: *1992 IEEE Workshop on Visual Languages*. 1992.
- Wittenburg et al. 1991 “Unification-based grammars and tabular parsing for graphical languages” by K. Wittenberg, L. Weitzmann, and J. Talley in *Journal of Visual Languages and Computing* (1991) **2**, 347-370.
- Yazdani&Ford 1996 “Reducing the cognitive requirements of visual programming” by Masoud Yazdani and Lindsey Ford. Pages 255-262 in *Proceedings 1996 IEEE Symposium on Visual Languages*, September 3-6, 1996, Boulder, Colorado.
- Yeung 1988 “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.
- Zuse 1996 “History of Software Measurement” by Horst Zuse. This is a world-wide-web page at <http://www.cs.tu-berlin.de/~zuse/3-hist.html> (as of May 29, 1996).
- Zuse&Bollman 1987 “Using Measurement Theory to Describe the Properties and Scales of Static Software Complexity Metrics.” by Horst Zuse and Peter Bollmann. Originally published as RC 13504, IBM Thomas Watson Research Center Yorktown Heights, 1987. Republished in *SIGPLAN Notices*, Volume 24, No. 8, pp.23-33, August 1989.

Chapter 3

Design Elements

This chapter presents a definitions of the syntax and semantics of the SPARCL language and relates the various aspects of the design of the SPARCL language to the basic principles presented in the hypothesis of the research project. A detailed tutorial introduction to SPARCL is presented in appendix 1 (“Tutorial Introduction to SPARCL”).

Definition of SPARCL.

We present informal definitions of the syntax and semantics of SPARCL. More formal definitions of the semantics are presented in chapter 6 (“Implementation”), where we give the procedural semantics as implemented by the interpreter, and chapter 4 (“Partitioned Structured Unification”), where we give a formal definition and analysis of the unification algorithm.

Diagrammatic Grammar. The diagrammatic grammar for SPARCL shows the elements of the language and the approximate layout of these elements in two-dimensions. This is basically a BNF-style grammar with diagrams on what would be the right-hand-side of the BNF productions. It is an adaptation of the Hyperedge Replacement Grammar (HRG) formalism presented in [Viehstaedt 1995]. The name of the nonterminal of the grammar that the production defines is in **bold** in the upper left hand corner of the “production box.” The rest of the production box (the “body”) shows the concrete graphical elements used to represent the nonterminal being defined, as well as showing the layout of other nonterminals with respect to those graphical elements. A nonterminal in the body of a production is represented with a box with concave sides containing the name of the nonterminal in *italics*. There is a special body symbol, an empty concave box, that represents the “empty” production.

Fact tables, clauses, arguments, literals, and terms. The first portions of the grammar define the “programmatic” portions: program elements and fact tables in Figure 3. 1, clauses and arguments in Figure 3. 2, and literals in Figure 3. 3.

The **fact_table** production shows a fact table as having a *name*, at least one *fact*,

and zero, one, or more additional *facts*. The *facts* are arranged vertically, in a “stack”. The fact table has a single line box around it.

The **args** production shows that the arguments of a clause or literal “stack”. The **arg** production shows the argument box with possibly empty **contents**. The **literals** production shows the literals being laid out such that they are in columns of three literals, except for the last column which may be one, two, or three literals. A **literal** is shown as being a name and some arguments on a special background. Of the nonterminals mentioned so far, three

have backgrounds: **clause**, **arg**, and **literal**. The clause and arg backgrounds differ in that the coloration of their outer edges are reversed. The literal background differs from the two others in that it is darker. Also, its edges are thicker. These various distinctions are easily observed in the example clause shown in Figure 3. 4.

The next portion of the grammar presents *terms*. The **term** production is shown in Figure 3. 5. The four basic term types are **variable**, **ur**, **empty_set**, and **partitioned_set**. The other three types being special forms of sets: **ntuple**, **table**, and **intensional set**. The productions for these term types are in Figure 3. 6.

Partitioned sets. The partitioned-set-related productions are **parts**, **part**, and **part_term**. The **parts** and **part** productions are in Figure 3. 6 and the **part_term** production is shown in Figure 3. 7. The **partitioned_set** and **part** productions introduce two more background graphics: a solid-colored and shadowed rectangle for the

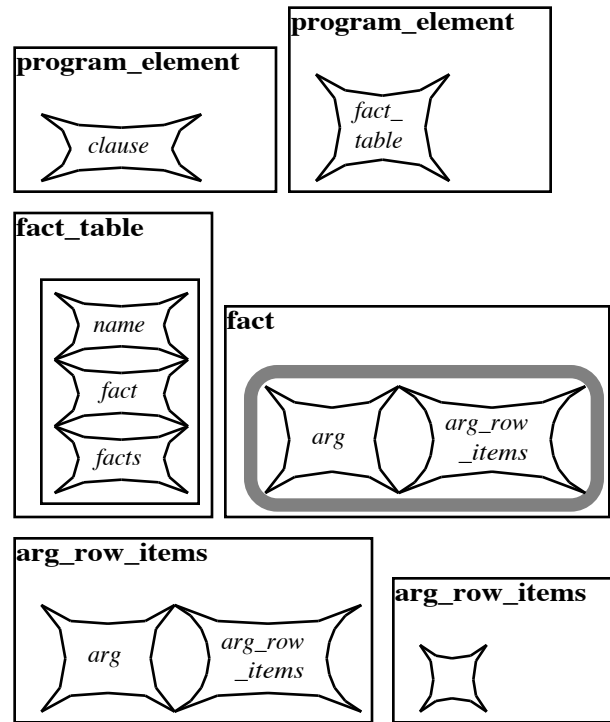


Figure 3. 1: Grammar productions for “program_element”, “fact_table”, “fact”, and “arg_row_items”.

set being partitioned, and a dotted line “open” rectangle for each part of the partitioning. The **parts** productions show that parts in a partitioned set stack one above the other in the same fashion as arguments. The **part_terms** productions show that a part may have many terms as contents, and that these terms are laid out in the same three-to-a-column fashion as for literals in the body of a clause. This is a simplification of the layout algorithm used by SPARCL. The actual layout algorithms are discussed in 6 (“Implementation”).

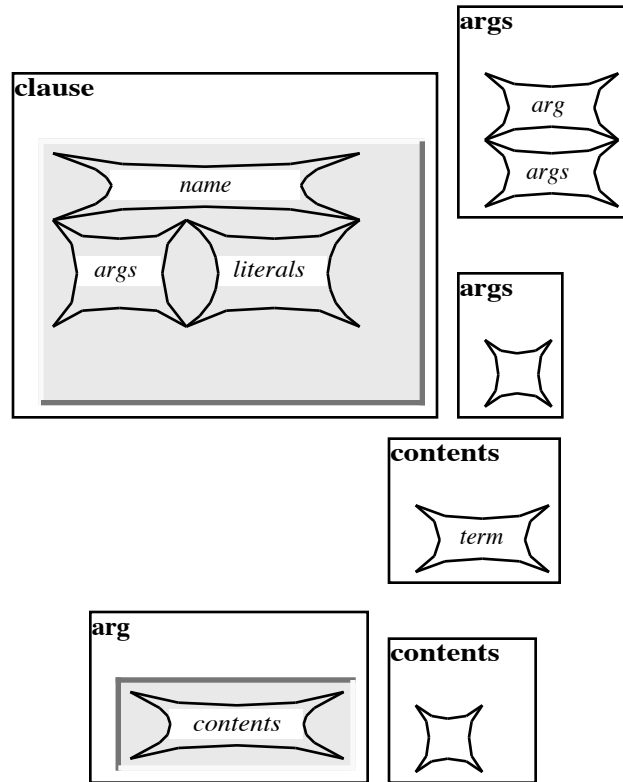


Figure 3.2: Grammar productions for “clause”, “args”, “arg”, and “contents”.

N-tuples. The productions for N-tuples are shown in Figure 3.8. These productions are **ntuple**, **ntuple_items**, **ntuple_item**, and **ntuple_divider**. The **ntuple** production requires that an N-tuple have at least two terms. The rightmost nonterminal, *term*, is one of these terms. The middle nonterminal, *ntuple_item*, is defined as a *term* and an *ntu-*

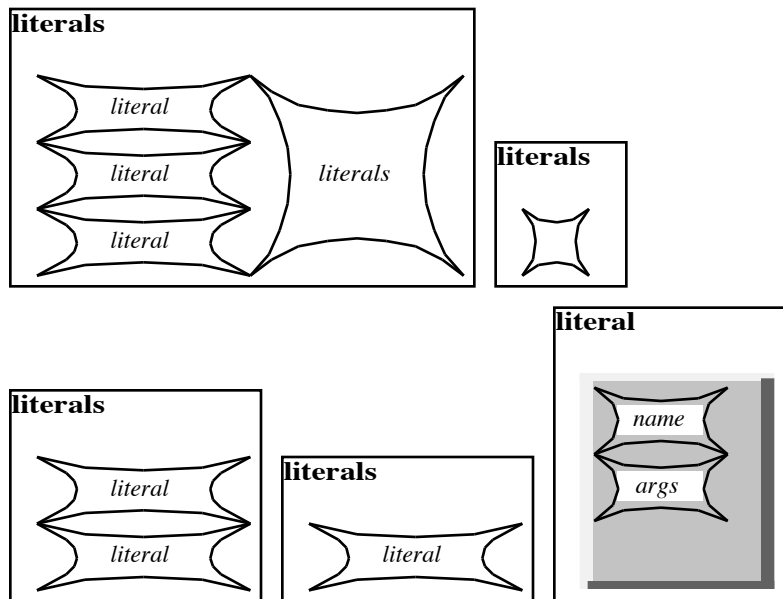


Figure 3.3: Grammar productions for “literals” and “literal”.

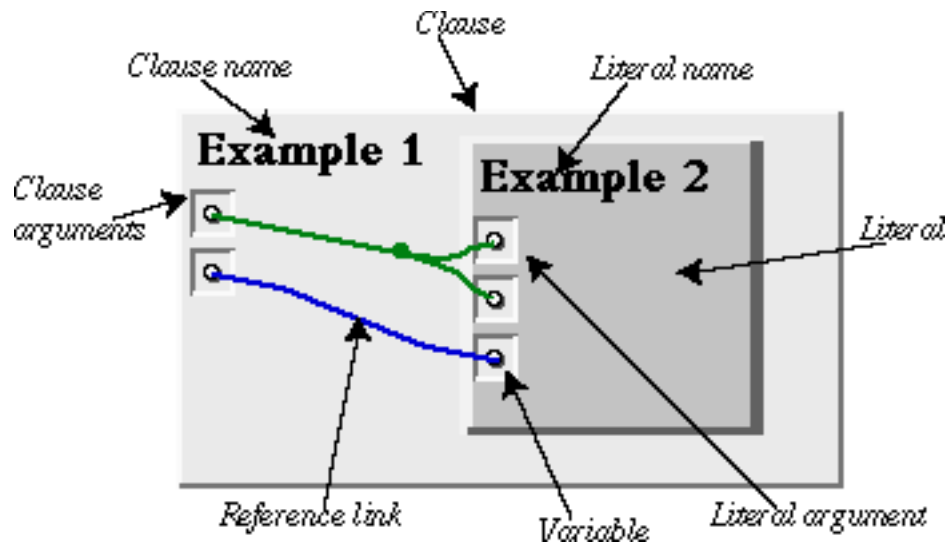


Figure 3.4: Clause and Literal Representations in SPARCL.

ple_divider. This gives the second term. The leftmost nonterminal, *ntuple_items*, is a possibly empty construction of *ntuple_item*. Thus, it may not contribute anything to the **ntuple** production. The **ntuple_item** production is used to insert an *ntuple_divider* (a right arrow) between adjacent elements of an N-tuple.

Tables. The next term-type we define is *table*. The top-level productions for tables are shown in Figure 3.9: **table**, **table_body**, **direct_table_body**, and **function_table_body**. The last two of these productions introduce a new “graphic”, a shadowed rectangle. This shadowed rectangle is nearly white, making it easy to distinguish from the other graphic elements.

There are two types of tables, “direct” tables and “function” tables. These two types differ in the syntax of the *table_body*. The direct table body is rows of items. There are some additional syntactic constraints on tables

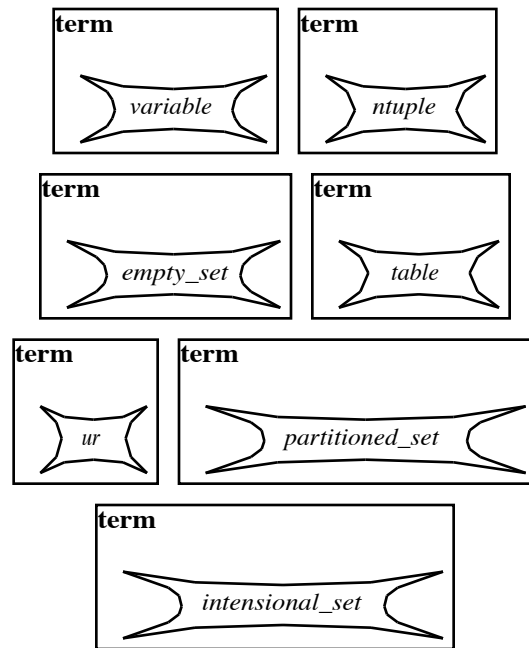


Figure 3.5: Grammar productions for “term”.

that are not shown in the grammar productions. The rows in a table must all have the same number of items and all of the items in a column must have the same width. We have not shown these constraints because they can not be described formally without significantly complicating the grammar formalism.

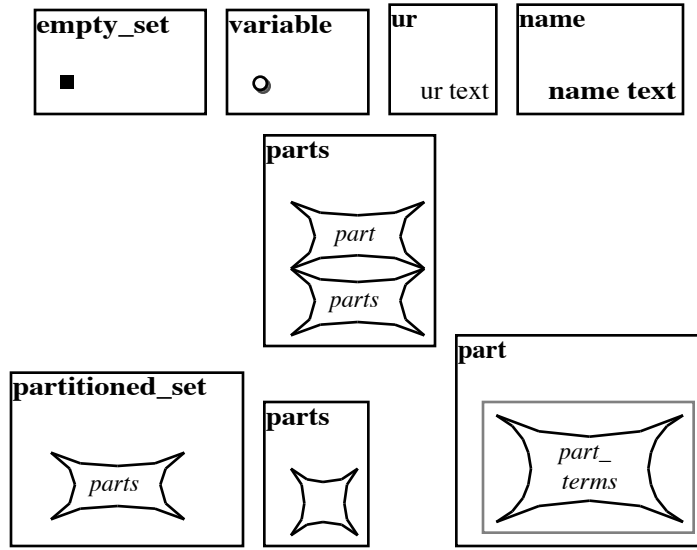


Figure 3.6: Grammar productions for “empty_set”, “variable”, “ur”, “name”, “partitioned_set”, “parts”, and “part”.

Productions for the rows and columns of tables are in Figure 3.10: **term_rows**, **term_row_items**, **term_row**, and **function_table_header**. These productions show that term rows are stacked (similarly to *arguments* and *parts*) and that a term row is a horizontal sequence of *contents* items.

Two new graphic elements are introduced in these productions, a horizontal line and a vertical line. The term rows that are the body of a table are separated by horizontal lines. This is shown in the **term_rows** production by a horizontal line above the *term_row* nonterminal. Similarly, the columns in a table are separated by vertical lines and this is shown in the **term_row_items** production by a vertical line on the right of the *contents* nonterminal. The top row of the *function_table_body* is the header that contains the “names” of the columns. This

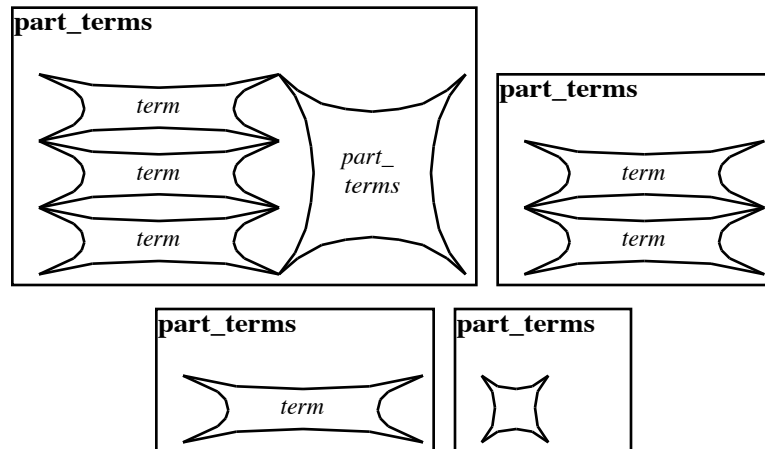


Figure 3.7: Grammar productions for part_terms.

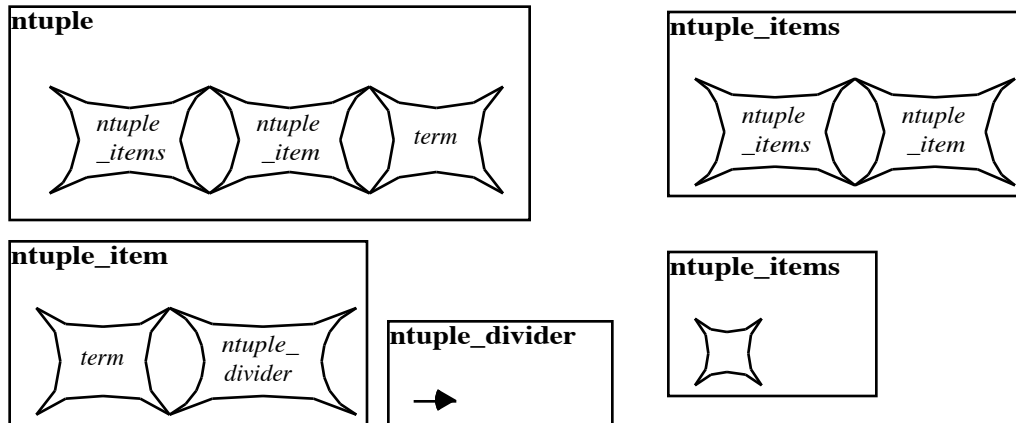


Figure 3. 8: Grammar productions for “ntuple”, “ntuple_items”, “ntuple_item”, and “ntuple_divider”.

row is set off from the rest of the table by a double line. This is indicated in the combination of the **function_table_header** and **term_rows** productions: the **function_table_header** is a *term_row* with a horizontal line under it, and **term_rows** has a horizontal line above each *term_row* in it.

Figure 3. 11 shows the **prefix**, **root**, **column_order_flag**, and **row_order_flag** productions. These define the nonterminals that surround the

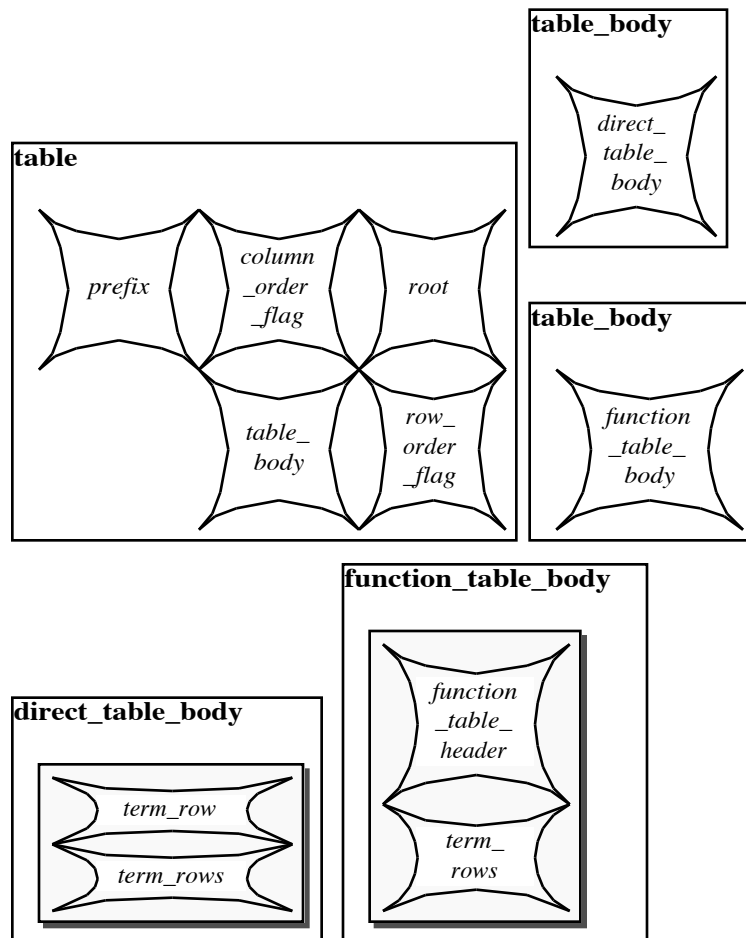


Figure 3. 9: Grammar productions for “table”, “table_body”, “direct_table_body”, and “function_table_body”.

table_body in the **table** production. The *prefix* nonterminal holds a term that is the common “prefix” of all of the rows of the table. This prefix may be an K-tuple, in which the elements of this K-tuple are the first K columns of each row of the table. The *root* nonterminal holds a term that precedes the rows of the table. There can only be a *root* if the

rows are ordered (i.e. they form an N-tuple, where the *root* becomes the leftmost or first element). The *column_order_flag* indicates if the columns of the table are ordered (i.e. if each row of the table makes an N-tuple). The *row_order_flag* indicates if the rows are ordered (i.e. if the table is an N-tuple of its rows).

Intensional sets. The last term-type we define is *intensional_set*. This type of term defines a set or multiset of elements where these elements are all of the terms that have some “property” in common. The idea of having a property in common is expressed by specifying a set of literals that must all be true.

The **intensional_set** and **inten-**

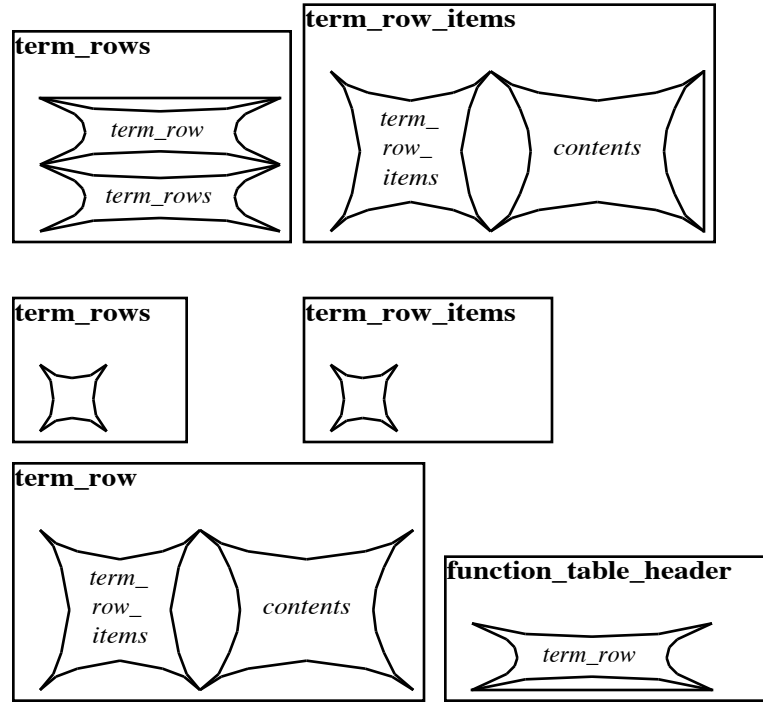


Figure 3.10: Grammar productions for “function_table_header”, “term_rows”, “term_row_items”, and “term_row”.

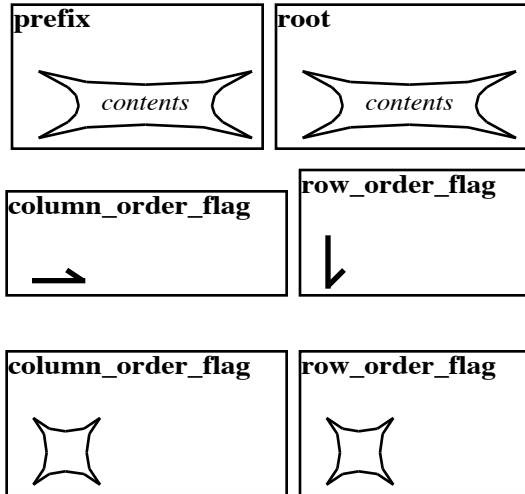
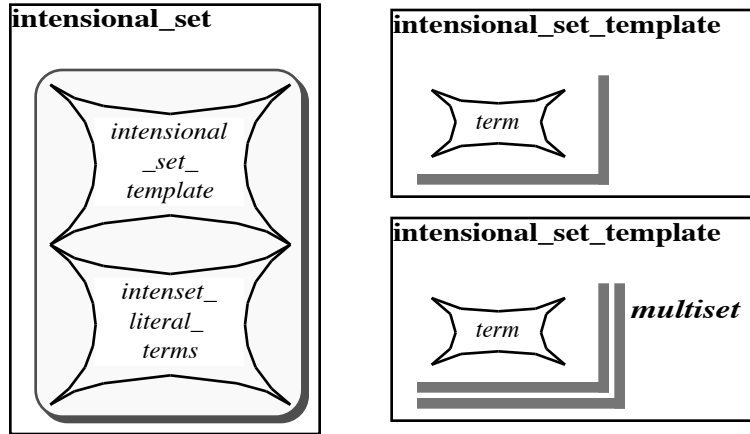


Figure 3.11: Grammar productions for “prefix”, “root”, “column_order_flag”, and “row_order_flag”.

intensional_set_template productions are shown in Figure 3. 12. These productions introduce a new graphic: a light-gray solid-colored shadowed rectangle with rounded corners for



the **intensional_set**; two line segments making a dark gray “corner” for the **intensional_set_template** for a basic intensional set; and, a bold italicized *multiset* and four line segments making two nested dark gray corners for the **intensional_set_template** for an intensional multiset.

The literals defining the common property of the intensional set are in the *intensional_literal_terms* nonterminal. There are several productions used to define *intensional_literal_terms*, but the basic idea is very similar to the *part_terms* nonterminal. The difference is that each *intensional_literal_term* must be a term of a special form, instead of any term as is the case in *part_terms*. The several productions related to *intensional_literal_terms* are needed to specify this special form.

The **intensional_literal_terms** productions are shown in Figure 3. 13. These layout a collection of *intensional_literal_term* nonterminals in columns of three, from left to right. This is the same pattern used to layout *literals* and *part_terms*.

The **intensional_literal_term** and **literal_term** productions are shown in Figure 3. 14. These productions specify the subset of terms that can serve as *intensional_literal_terms*: a *variable* or a *literal_term*, where a *literal_term* is either an *ur* or a *literal_ntuple*. Of these term types, only the *literal_ntuple* is new.

The **literal_ntuple**, **literal_ntuple_items**, and **final_literal_ntuple_item** productions are shown in Figure 3. 15. These productions define a *literal_ntuple* in nearly the same fashion as an *ntuple*. The difference is that the last element of a *literal_ntuple*, i.e. the element furthest to the left, is limited to being an *ur* or a *variable*, where the last element of an *ntuple* is a *term*.

Coreference

links. This grammar does not define the coreference links. A coreference link connects two or more “linkable items” by a hyper-edge. A linkable item is a *term* or a *part*. In this one respect a *part* has the same status in the syntax of SPARCL as a *term*. This allows a SPARCL program to place coreference restrictions on subsets (parts).

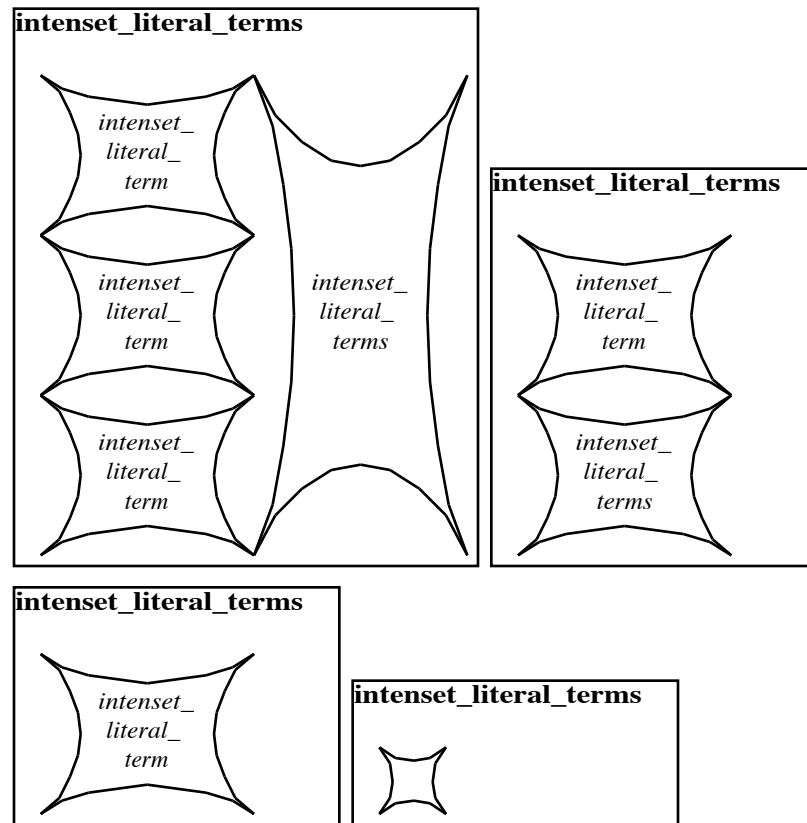


Figure 3.13: Grammar productions for “intenset_literal_terms”.

Design Elements

The design of SPARCL is based on three major aspects: visual programming, logic programming, and sets with partitioning. The various parts of the design derive from one or more of these aspects. We discuss the design elements grouped according to this derivation. Figure 3.16 shows this grouping.

1. Visual & Logic Design Elements.

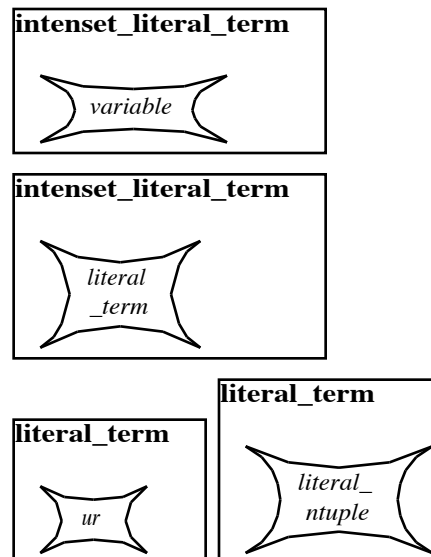


Figure 3.14: Grammar productions for “intenset_literal_term” and “literal_term”.

The visual and logic design elements are the visual representation of the logic programming

semantics of SPARCL. Both the two-dimensional and the three-dimensional visual representation must make clear the various relationships of the semantics of a program. There are four basic ways to do this that are *diagrammatic* (as

opposed to *linguistic*): spatial containment, adjointment/adjacency, connecting lines, and similar appearance. This distinction between diagrammatic and linguistic representations is one which Shin makes in *The Logical Status of Diagrams* [Shin 1994].

Spatial containment is used in the representation of SPARCL for all of the containment relations in a program such as: name of a predicate contained in the representation of a clause for that predicate, literals in the body of a clause contained in the representation of that clause, and elements of a part of a partitioned set contained in the representation of that part. Adjoinment is used for the arguments of a clause or literal: they are placed next to each other (one above the next) to form a sequence where the first one is at the top. Adjoinment is also used for the elements

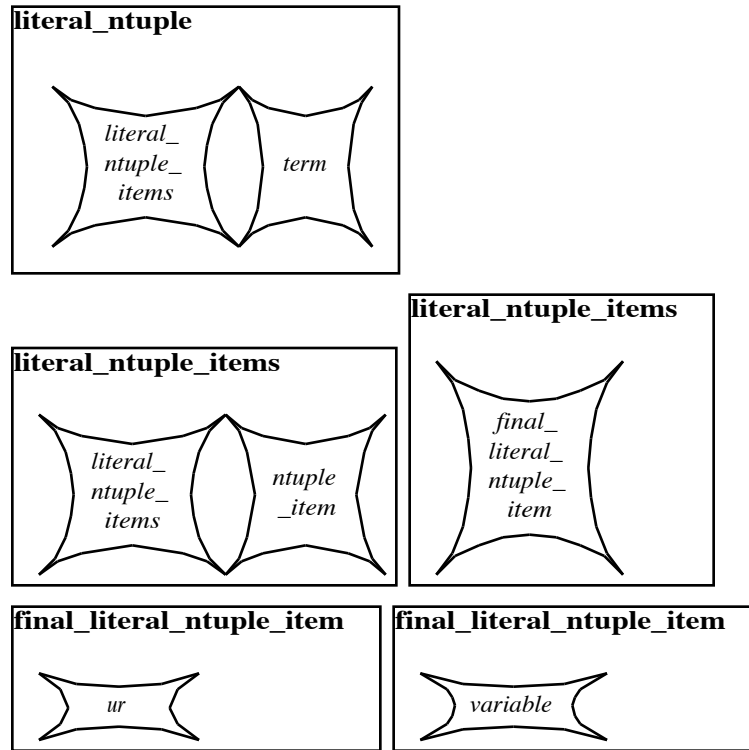


Figure 3.15: Grammar productions for “literal_ntuple”, “literal_ntuple_items”, and “final_literal_ntuple_item”.

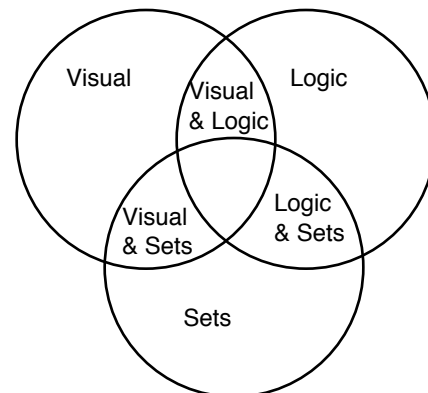


Figure 3.16: Derivation of Design Elements from Major Design Aspects.

of an N-tuple: they are placed next to each other (one to the left of the next) to form a sequence where the first one is at the left end. Connecting lines are used to represent coreference hyperedges. Similar appearances are used to indicate term types. For example, all of the variables are small circles and all of the partitioned sets are solid rectangles with a dashed rectangle just inside and a background of a certain color. We discuss the distinction between diagrammatic and linguistic representations further in chapter 8 (“Objective Analysis”), where we analyze the extent to which SPARCL takes advantage of the opportunity to be diagrammatic.

Program elements. The basic program elements of a SPARCL program are derived from its logic programming aspect: program, clause, and literal. A program is a set of clauses. It is represented by a window in the programming environment (which is stored in a single document or file). The parts of the clause and literal are shown in Figure 3.4. The outermost light gray rectangle at the top of the figure encloses a clause. In the upper left corner is the “clause name” and on the left hand side of the clause rectangle are two clause argument rectangles. The clause name and number of arguments identifies the predicate to which the clause belongs, in this case that is the predicate ‘Example 1’/2. The arguments of the clause are ordered, the first argument is at the top, the second argument is below that, and so on. The darker gray rectangle on the right side of the clause rectangle encloses the representation of a literal. The “literal name” is in the upper left corner of the literal rectangle, and the literal argument rectangles are on the left hand side of the literal rectangle. The literal name and the number of its arguments identify the predicate whose defining clauses are to be used in solving the literal. In this case, this is a literal of the ‘Example 2’/3 predicate. Literal arguments are ordered in the same way as for clause arguments, with the first argument at the top, the second argument immediately below the first, and so forth.

The argument rectangles for the clause and literal contain terms to be unified. In this figure, the terms are all variables.

Coreference links. Terms in a SPARCL clause may be specified as referring to the same “underlying” term. Such terms are said to “corefer”. This common underlying term may be identical to one of the terms as represented, or it may have some nonvariable elements in places where one or both of the coreferring terms have variables.

Technically, coreferring terms are *unified* as part of interpreting the clause which contains these terms. Coreference of terms is shown by curved lines that connect those terms. In Figure 3. 4, the variable in the first argument of the clause corefers with the variables in the first and second arguments of the literal, and the second argument of the clause corefers with the third argument of the literal. This example shows how SPARCL uses coreference links instead of variable *names* to identify term instances in the representation of a pro-

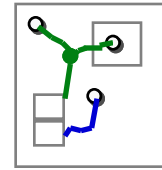


Figure 3.17:
Example of two
coreference
links in SPARCL.

gram which corefer. Coreference links provide more than the facility of variable names because any kind of linkable items may be linked, not just variables. This is shown in Figure 3.17. In this figure there is a link between three items and another link between two items. The three-way link links two variables and a part of a partitioned set. The other link links a variable and the other part of a partitioned set. The choice of explicit linking to express coreference instead of names is taking advantage of the diagrammatic possibilities of a *visual* representation in representing the *logical* concept of coreference (implemented as unification).

This approach to coreference provides a facility with some of the expressiveness of feature structures. Feature structures are a technique used in some grammar formalisms which allow arbitrary substructures to be identified (usually by an integer) and any two substructures which have the same identifier must “unify”, essentially they are to be considered as being the same substructure. This allows the representation of containment relationships between parts of feature structures to be a graph instead of a tree. Since arbitrary terms in SPARCL may be made to corefer, the representation of containment relationships between terms in SPARCL may also be a graph instead of a tree. Feature structures are not completely supported by SPARCL because it does not support the special kind of unification needed for feature-structures. In feature structure unification, the result of unifying two feature structures is a feature structure that contains all of the features in both of the structures being unified, and where the same-named feature appears in both of these structures, their values must feature-structure-unify. Thus, feature structure unification is a mixture of set unification and set union.

Delay specification. Figure 3. 18 shows a ‘*DELAY*/2 clause. Clauses of this form are used to specify to the SPARCL interpreter when to delay the interpretation of a lit-

eral. Before SPARCL interprets a literal, it checks to see if there is a delay clause that matches that literal. If so, that literal will not be considered for interpretation until some other literal has been interpreted, i.e. the literal's interpretation is "delayed". The example in the figure delays the interpretation of an 'Example 1'/2 literal that has an unbound variable as its first argument. The interpretation of delay specifications is discussed in more detail in section 6, "Logic Programming Design Elements."

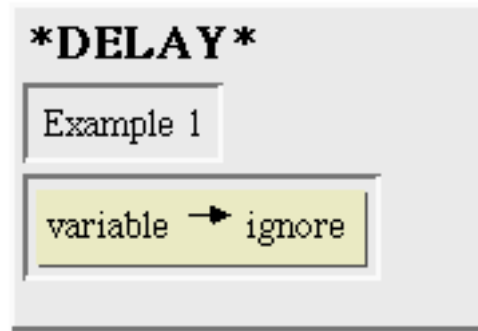


Figure 3.18: "***DELAY***" clause for the 'Example 1'/2 clause of Figure 3.4.

Since delay specifications are very stylized kinds of clauses, a special representation for them can probably be devised that is much more concise than using a regular clause with arguments and N-tuples. We have not attempted such a design since this one was sufficient to allow delays to be specified and required no additional development work in the editing system of SPARCL. However, we found in our analysis of SPARCL programs that the delay specifications are significant part of the size of a SPARCL program. This is discussed in chapter 8 ("Objective Analysis"). For this reason, we expect to redesign the representation of delay specifications in the future.

Built-in predicates. The bottom right corner of Figure 3.4 shows a list of the built-in predicates defined for SPARCL. These have interpretations built in to the SPARCL interpreter.

2. Visual & Sets Design Elements.

There are several different representations of sets in SPARCL. These representations are of two types, basic and specialized. The basic representation is semantically complete. Anything which can be expressed using sets in SPARCL can be expressed using only the basic representation. However, the basic representation is awkward for representing some common special forms of sets. These special forms are of three types, N-tuples, tables, and intensional sets. The N-tuple is the basic representation for ordered collections of terms, for sequences. A list is a special kind of N-tuple. A

table is a set or N-tuple of collections. The elements of a table, the rows, are either N-tuples (all of the same length) or functions (all of the same domain). An intensional set is a specification of a set by giving the property which all of the members of that set must have, rather than explicitly giving the elements of the set (an extensional set, which is the basic representation of a set).

Basic set representation. The representation of sets is visual. More precisely it is diagrammatic rather than linguistic. There are two forms of the basic set

representation, the empty set and the partitioned set. These are shown in Figure 3.19. The parts of a partitioned set represent subsets of the set being partitioned. The elements in a part of a partitioned set have no particular order. The layout mechanism places them so as to keep the representation fairly compact (within a “minimal” rectangle). The layout has no ordering implications. As discussed earlier, this lack of ordering is easier to convey in a nonlinear representation than in a linear one. The parts of a partitioned set are simply “stacked” vertically by the layout mechanism. Again, there is no particular order to the parts. The stacking is less than ideal in this case since a viewer might infer some order (top to bottom or vice versa) where none is intended. However, the stacking has the virtue of being easy to layout in a readable fashion. It is unusual to have partitioned sets of more than two parts, and very unusual to have more than three parts. A part which is shown with nothing in it is considered to be a variable part, one with unspecified contents, rather than an empty part¹. There is generally no need to explicitly represent an empty part, one can simply omit an empty part from the representation altogether. If an explicitly empty part is needed, then one can create a hollow part and

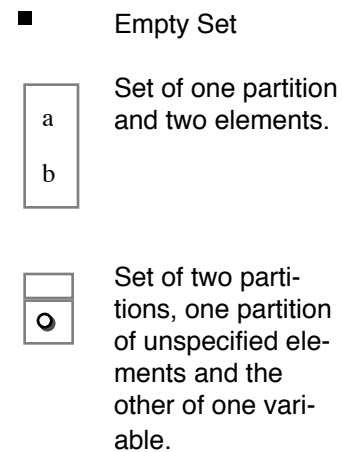


Figure 3.19: Example SPARCL Basic Sets.

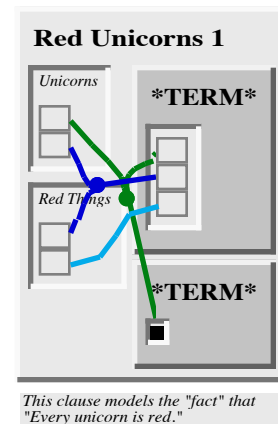


Figure 3.20: Example of having a part of a partitioned set corefer with an empty set in a “*TERM*/1 literal. (From Figure 8.4.)

1. This representation of a variable part is an aspect of SPARCL’S representation being diagrammatic

$a \rightarrow b$ create an empty set elsewhere in the clause (using a `*TERM*/1` literal to hold the empty set, if there is no other logical place to put it), and make the hollow part and the empty set corefer. An example of this is shown in Figure 3. 20.

Figure 3.21:
N-tuple of two
elements in
SPARCL.

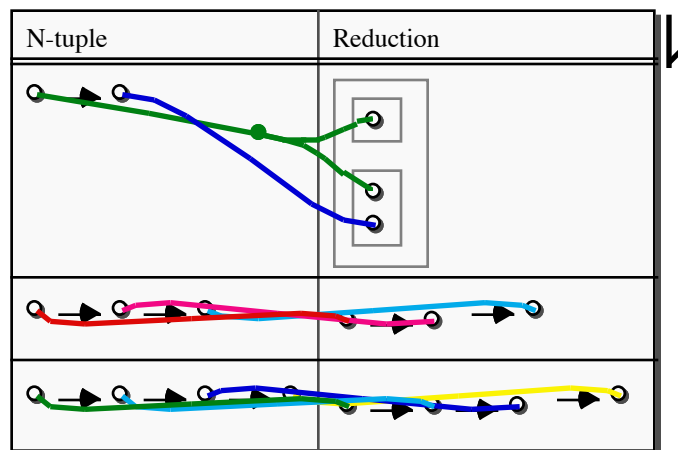


Figure 3.22: N-tuple reductions.

The top part of the partitioned set in the first argument of the ‘Red Unicorns 1’/2 clause corefers with the top part of the partitioned set in one of the `*TERM*/1` literals and with the empty set in the other `*TERM*/1` literal. (This example is taken from a larger example presented in Figure 8.4.)

Specialized set representation for N-tuples. The N-tuple is an ordered set of 2 or more terms. An example with two elements is shown in Figure 3.21. Some reductions of N-tuples are shown in Figure 3.22. The first line of the table in Figure 3.22 shows the reduction of an ordered pair (a 2-tuple) to a set containing two sets. The top contained set contains one element, the first element of the ordered pair. The bottom contained set contains two elements, the first and second elements of the ordered pair.

The second line of the table of Figure 3.22 shows the reduction of an ordered triple (a 3-tuple) to an ordered pair. The ordered pair has an ordered pair as its first element, and its second element is the same as the third element of the ordered triple. The ordered pair of the reduction ordered pair’s first element consists of the first two elements of the ordered triple.

The third line of the table of Figure 3.22 shows the reduction of a 4-tuple to an ordered pair. The first three elements of the 4-tuple map into an ordered triple which is the first element of the reduction ordered pair. The last element of the 4-tuple maps into the second element of the reduction ordered pair.

Other ordered collections such as lists and matrices can be built using N-tuples.

instead of linguistic.

Special representations for these other ordered collections would be useful in SPARCL, in addition to the N-tuple representation, but these have not been designed or implemented yet.

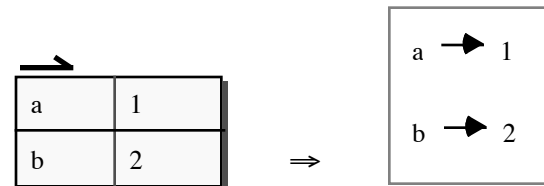
Mappings, functions, and multisets. A *mapping* is a set of ordered pairs, where the first elements of the ordered pairs are the domain of the function, and the second elements of the ordered pairs are the range. A *function* is a mapping where no two ordered pairs have the same domain element.

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 ‘+’ (plus) and ‘*’ (times) as used by the is/2 built-in predicate both accept multisets (with numbers as their domain).

Specialized set representation for tables. A term table represents a set or an N-tuple of rows, where each row is an N-tuple or a function. 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.

Examples of tables of N-tuples are shown in Figure 3.23. The left-hand side of the double-arrow at the top of the figure shows a table of two rows and two columns. The horizontal half-arrow at the top of the table indicates that the columns are ordered. The lack of any “decoration” (a vertical half-arrow) on the side of the table indicates that the rows are unordered. This table on the left-hand side of the double arrow is interpreted by SPARCL as though it were the term represented on the right-hand side of the double arrow. Thus, these two

terms are equivalent; they unify. The term to the right-hand side of the double arrow is a set of two N-tuples: $\{(a \Rightarrow 1), (b \Rightarrow 2)\}$.



The bottom of Figure 3.23 shows a table on the left-hand side of the double arrow with half arrows on its top and side. These half arrows indicate that the columns are ordered and that the rows are

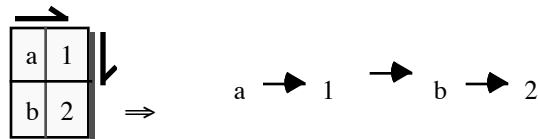


Figure 3.23: Set and N-tuple representation of tables of N-tuples. (The \Rightarrow means “is a representation of”).

ordered. The interpreted version of this term is shown on the right-hand side. It is an N-tuple of N-tuples: $((a \Rightarrow 1) \Rightarrow (b \Rightarrow 2))$.

There are some additional styles of representing tables not shown in the figure. If the term table is an N-tuple of rows, then there may be a “root” 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. Also, there may be a common “factor” extracted from each of the rows and placed next to the table. If the table is a collection of N-tuples, then the common factor is a prefix for each of the rows of the table. If the table is a collection of functions (see below), then the common factor is a set of domain value/range value pairs common to each of the rows. This factoring can significantly reduce the number of columns of a table.

Examples of tables of functions are shown in Figure 3.24. The left-hand side of the top pair of terms shows a function table with unordered rows. The names of the columns are “a” and “1”. This is the domain of the functions that this table represents. The range values into which the domain values are mapped are in the bottom two rows of the table. The right-hand side shows an equivalent term, a set of two N-tuple tables. Each of these N-tuple tables corresponds to one of the functions specified by the bottom two rows of the function table. For the first range row, the corresponding function is $\{(a \Rightarrow b), (1 \Rightarrow 2)\}$. The function for the second row is $\{(a \Rightarrow c), (1 \Rightarrow 3)\}$. The left-hand side of the bottom pair of terms in Figure 3.24 shows a function table with ordered rows. The corresponding right-hand side term is an N-tuple of the two functions for the bottom two rows of the left-hand side. A function table can be considered as representing a “relation”. For instance, one can represent a phone book “relation” where the domain values are “name”, “phone number”, and “address”. The rows of this function table are the entries in a phone book.

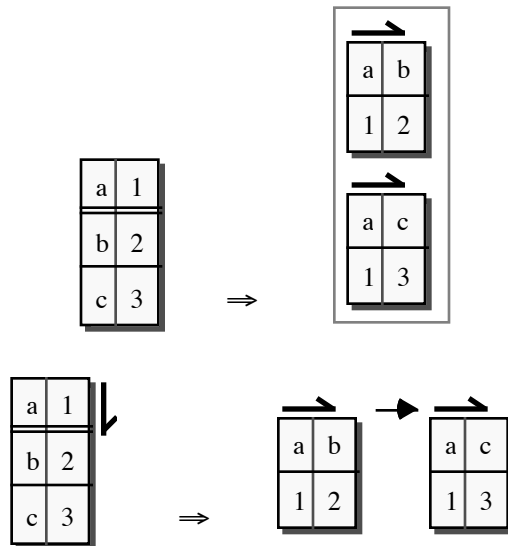


Figure 3.24: Set and N-tuple representation of tables of N-tuples.

Specialized set representation for intensional sets. Intensional sets in SPARCL are a shorthand notation for the use of the “setof” built-in predicate. Beyond simply providing a compact representation of a “setof” literal, this representation also allows one to handle the entire “setof” expression as a term, nesting it in other terms and placing it in argument positions. An example of an intensional set is shown in Figure 3.25.

Figure 3.25 shows two semantically equivalent forms of a clause for the ‘bar’/1 predicate. The upper form uses an intensional set term in the single argument of the clause. This set is the set of all terms such that these terms are solutions of the argument for the ‘foo’/1 predicate. Suppose the ‘foo’/1 predicate is true only for the even numbers between 1 and 9 (i.e. 2, 4, 6, and 8). This makes the value of the intensional set ‘{2, 4, 6, 8}’.

The lower form of the ‘bar’/1 clause uses the ‘setof’/3 built-in meta-predicate. The ‘setof’/3 predicate has an interpretation related to the intensional set term. The first argument of the ‘setof’/3 predicate is the form or pattern of terms to be collected into a set, just as the term in the upper left corner of the intensional set box is the pattern of terms in the intensional set. The second argument of the ‘setof’/3 predicate is the set of literals that when satisfied provide bindings for one or more variables in the set element pattern of the first argument. This set of literals is found in the “body” of the corresponding intensional set box. The third argument is the set produced by all of the values of the pattern of the first argument from all of the solutions of the literal set of the second argument. This set is the term that the intensional set term represents. Thus the lower form of the ‘bar’/1 clause in Figure 3.25 has the same solution as the upper form, the set ‘{2, 4, 6, 8}’.

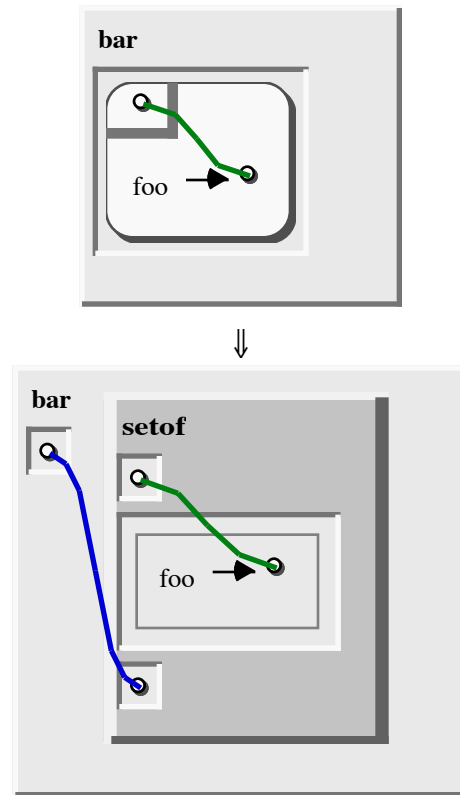


Figure 3.25: Set and literal representation of an intensional set (in an argument of a clause).

Using the intensional set term frequently allows one to use fewer coreference links, as is the case in the example in Figure 3.25. Nesting intensional set terms extends this benefit, as well as being substantially more compact. The nesting occurs when one needs to express something like “the set of all pairs (X, Y) such that p(X) and Y is the set of all B such that q(X, B)”. In this example the nested intensional set is “Y is the set of all B such that q(X, B)”. If p/1 is “p(a); p(b); p(c)” and q/2 is “q(a,1); q(a,2); q(b,3); q(c,4);q(c,5)”, then this example would yield the set: ‘{(a, {1,2}), (b, {3}), (c, {4,5})}’. We used a form of this kind of nesting in our SPARCL solution of the *Classify Examples* sub-problem of the *ID3* problem, discussed in sections 1 and 2 of chapter 7 (“Subjective Analysis”).

3. Logic & Set Design Elements.

The combination of sets with partitioning constraints and logic programming leads to several design elements of SPARCL. Sets are the fundamental organizational tool of SPARCL programs, instead of structures as in PROLOG or lists as in LISP. Ordered collections of terms, N-tuples, are handled as special organizations of sets, as indicated above. The unification algorithm and inference mechanism are specialized in SPARCL to handle partitioned sets and their associated constraints. Unification is optimized to recognize N-tuples and handle them directly when possible instead of always converting them to their pure set representation. The partitioned set unification algorithm is defined and analyzed in chapter 4 (“Partitioned Structured Unification”). The implementation is discussed in chapter 6 (“Implementation”).

Clause and literal ordering. Programs are sets of clauses in SPARCL and therefore unordered; they are usually ordered in other logic programming languages. Similarly, the body of a clause is a set of literals and is also therefore unordered. Since SPARCL is a programming language and not a theorem prover, we have accepted some limitations on SPARCL’s ability to “solve” collections of Horn clauses. Also, we want to provide the programmer some ability to direct the interpreter’s actions, for improved performance. These are perhaps the central distinctions between logic programming and theorem proving. We have provided the “delay” mechanism to mitigate the unorderedness of literals in a clause. This mechanism allows the programmer to specify certain conditions on the state of the interpretation under which interpreting a particu-

lar literal should not be attempted. These conditions involve the presence of unbound variables in one or more arguments of the literal in question. This is a non-sequential-ordering mechanism; one can specify (in effect) that literal B can't be solved until literal A has been. For example, suppose literals A and B have coreferring variables in an argument of each and literal B has a delay specification requiring that literal B be delayed if the argument with the variable coreferring with the variable in A is non-ground. Then solving A such that the variable of interest is grounded is the only way to make the delay specification of B fail to apply, thus making B available for interpretation. This is a non-sequential ordering in that literal A is required to be solve before literal B, but it need not be solved immediately before solving B. The solution of other literals could be done between solving A and B. More complex ordering relationships are possible. For instance, a literal Z could be delayed until *either* literal X *or* literal Y has been solved, with the ordering of X and Y unspecified.

The ordering of choices among clauses (as opposed to the ordering of choices among literals) is also sometimes necessary. Since SPARCL views a predicate definition as a set of clauses, this ordering would normally be unavailable to the programmer. This poses a serious problem when the programmer wants to implement a specific algorithm for solving a problem (as opposed to developing an algorithm “native” to SPARCL). We encountered this difficulty in implementing a solution for the *WARP-LAN* problem presented in chapter 7 (“Subjective Analysis”). SPARCL provides clause-ordering via two built-in meta-predicates, `ordered_disjunction/2` and `if/3`. Declaratively, the `ordered_disjunction/2` predicate is true if either of the (sets of literals) in its first argument or in its second argument is true. Procedurally, `ordered_disjunction/2` tries its first argument first and when it backtracks it tries its second argument. The `if/3` predicate takes three sets of literals, the “condition”, “then”, and “else” sets. Declaratively, the `if/3` predicate is true if either the condition set is true and the “then” set is true, or if the condition set is false and the “else” set is true. Procedurally, the `if/3` predicate solves the condition first. It “remembers” if any solution to the condition was found. If the condition was true, it then solves the “then” set of literals. On backtracking it will find other solutions of the condition and “then” sets of literals, if any. It will *not* backtrack into the “else” set of literals if any solution of the condition set was found. The `if/3` predicate only attempts to solve the “else” set of literals if *no* solution to the condition set was found.

Multisets. Multisets are introduced in the “Mappings, functions, and multisets” subsection of section 2 (“Visual & Sets Design Elements”) above. Multisets are present in SPARCL as a response to the desire to use intensional sets in “counting” situations, particularly as operands to arithmetic operators. This comes up naturally in many situations: one frequently wants the sum or product of all of the values such that some predicate holds for each value. For instance, one might want the sum of all of the daily revenue of a business to determine the revenue for the week (the predicate these values have in common is “being the daily revenue of the business”). Normally, one finds that statements of the form “all of the values such that some predicate holds for each value” are correctly interpreted as “the set of all X such that $p(X)$.” This, in turn, is represented concisely in SPARCL as an intensional set with X as the template variable and $p(X)$ as the body literal. However, in the example given, this statement should be interpreted as “the *multiset* of all X such that $p(X)$.” If the revenue of two different days happened to be the same, then that value should appear twice in the sum. However, a value only “appears” once in a set so a sum over the *set* of values for the week would miss a day’s revenue (the duplicated value). In the multiset of these revenues the duplicated value has a count of 2 (where the nonduplicated values have counts of 1). A properly constructed sum over this multiset produces the correct result by multiplying each value by its count.

4. Visual Programming Design Elements.

The visual representation of SPARCL embodies many design elements. There are two distinct representations of SPARCL, a *two-dimensional* one and a *three-dimensional* one. The two-dimensional representation is fully developed and was used for most of the research on which this dissertation reports. The three-dimensional representation is incomplete in various ways, but it is interesting even in this partial implementation. Both the two-dimensional and three-dimensional representations are constructed from the same internal canonical representation. This is feasible because SPARCL relies almost entirely on automated layout to arrange the visual details of the concrete representation, be it two- or three-dimensional. Automated layout in SPARCL supports a variety of concrete representations and it also supports a “semantically-oriented” structured editing environment. Various techniques are used to visually differentiate elements of the representation in a coordinated and principled way: straight lines for

terms versus smoothly curved lines for coreference links; greys and light pastels for terms versus strong colors for links. The different kinds of spatial relationships (containment, adjoinment, linking) are each used in visually consistent way: Elements that abstractly contain other elements are represented by rectangular shapes with the representations of the abstractly contained elements concretely nested within the shape; ordering is represented by adjoinment (vertical or horizontal), sometimes including an appropriate decoration (such as an arrow); and coreference (a kind of equivalence) is represented by connecting lines.

The three-dimensional representation has only a static version, there is no direct user editing of the representation. The user can interactively *view* the representation. Also the three-dimensional representation is partial in that not all of the syntax of SPARCL can be represented three-dimensionally.

Program editing approaches. A problem common to visual programming languages is the need to display and edit relationships among elements within pictures or "scenes" that are the visual representation of some part of a program. Using nonlinear representations, the programmer may edit scenes in four ways:

- (1) the programmer may draw the program representation "free hand" and have the system infer what the intended program is;
- (2) the programmer may draw the program representation using system-provided elements (box, circle, line, text, etc.) and have the system infer what the intended program is;
- (3) the programmer may draw the program representation using system-provided semantically-specific elements (statement boxes, variable ovals, etc.), positioning them and sizing them explicitly; or,
- (4) the programmer may indicate the program representation by specifying semantic modifications ("add a statement", "change variable reference", "reference a function") with automatic layout, an extreme version of a syntax-directed editor.

SPARCL uses the fourth approach, a "semantic modification" editing environment. Examples of this approach are given in chapter 3 ("Tutorial Introduction to SPARCL"). This approach is most strongly motivated by the desire to simplify the editing of a

two- or three-dimensional representation. It also has the advantage of elimination syntactic programming errors, since the program representation is syntactically correct at all times. The primary disadvantage is that the automated layout system is not as good at creating an easily understood layout as the programmer is. However, a counteracting advantage is that the layout style of programs does not vary among programmer's, making it easier to understand other programmer's code. Within the context of automated layout, one can offer parametric control of the layout algorithm so that each programmer can tailor the appearance of programs to their taste. This tailoring is not arbitrary, it is limited by the layout algorithm's capabilities. Thus, although programs are entirely automatically laid out, the way in which a particular program looks can be different for different programmers.

The need to simplify the editing task is discussed in the following sub-sections.

Representing relationships. There are several reasons that a layout may be incomprehensible. A common problem is simply fitting all of the program elements of interest on the screen. If a program is sufficiently complex such that it has many “tightly” interacting elements, there just may not be room to show all of the elements and their interactions at one time. Another common problem is showing relationships among elements.

These two problems are related. A representation which must show that two items are related has relatively few general purpose options. The major ones are connecting related items with a line of some type, having the representation of one item visually contained within the representation of another item, and using same appearance for the same visual or textural appearance for related items (e.g., same color, same shape, same text). All of these options have their problems.

Connecting lines create a graph layout problem. Line crossings are inevitable unless only planar graphs are allowed and even then the need for a representation that is flexible about the relative positioning of the “nodes” is a problem. Directed planar graphs can be laid out without line crossings, but the directed nature of the graph is generally obscured by such a layout.

Simple containment (for representing “tree” relationships) is easy to layout if screen space is not an issue. However, it uses up screen space faster than connecting lines. This is in part due to the necessity of leaving part of a container blank when a regular layout of that container's contents don't use up the available space in the

container. For readability one wants to make containers of the same kind be obviously similar in appearance (making it easy to “read” the similarity of kind), thus they should have very similar shapes. This pushes one to avoid changing the shape of a container simply to more efficiently use screen space. Generalized containment allows overlapping containers, which can represent directed-acyclic-graphs instead of only trees. Generalized containment is much harder to layout; it is much like the graph layout problem and it consumes screen space more rapidly than do connecting lines.

The “same appearance” approach requires that relations between items have the same appearance. It is limited to representations of equivalence classes with relatively few classifications of items and relatively few items overall, wherein each classification is readily distinguished from all of the other classifications. The relatively few classifications requirement springs from the difficulty of distinguishing very many different “appearances”. Also, there should be relatively few items overall as the viewer must search through all representations to locate all of the items having the same appearance. If there are many items and/or the distinguishing aspect of the appearance is difficult to see (e.g. very small shapes, small variation in color), then this search becomes tedious.

Many representations use all three of the techniques for different aspects of the programming language. In PHF [Fukunaga et al. 1993a], “same text” and “same icon” is used to classify particular operators such as “plus” or “times”, containment shows “subroutine” relationships, and connecting lines show data flows. For the editing environment, the programmer draws program elements using system-provided, semantically-tagged shapes, explicitly positioning and sizing them.

Overall, the attractiveness of lines is that:

- (1) they don't require searching as does "same appearance",
- (2) they are more flexible for displaying various relations than is containment, and
- (3) they don't use up screen space as rapidly as does containment.

Thus, using lines to show relatedness is clearly an interesting approach in spite of the difficulty of generating a layout which avoids line crossings. This provides one of the motivations for developing a 3D representation of SPARCL. Since the main objection to lines is their crossings, and the main problem with crossings is the visual confusion created by a lack of depth in 2D representations, a 3D representation may

reduce or eliminate this visual confusion.

Requirements for a 3D Line Representation. Since crossing lines are inevitable in many representations, some technique to make them less confusing is needed. We propose to use a 3D representation that will clearly delineate line crossings, thereby removing the visual confusion associated with 2D representations of lines.

One of the challenges of providing such a 3D representation is providing the programmer with a convenient way to edit scenes. With 3D representation, scenes can be enormously complex to define, even when there are relatively few (10 or 20) elements. For 3D representation, either the scenes must be extremely simple, or the system must provide substantial aid in generating them. A “semantic-modification” environment, which does all of the layout, provides the most such aid. This approach is feasible if there is an appropriate modification scheme. A modification scheme is “appropriate” if it is usable, and if at any distinguished part in a representation there are only a “small” number of modifications possible, and if this scheme can be used to create all valid programs. Not all programming language representations lend themselves to such a modification scheme.

Fully automated layout is feasible if an algorithm can be implemented which can find a comprehensible layout for most valid programs in a reasonable amount of time and space. For those valid programs which the algorithm cannot find a comprehensible layout there must be an alternative program which achieves the same ends as the difficult program and for which the algorithm can find a comprehensible layout. This last condition allows the system to “require” the programmer to use some kind of modularization to achieve a readable program representation. The “most valid programs” phrase means “most of the valid programs a programmer is likely to write”. This means that the modularization requirement of the layout algorithm should at most infrequently force a programmer to divide something into modules only to aid in layout comprehensibility.

This provides several design elements for the representation of SPARCL: three-dimensional representation, “semantic-modification” structured editing, a modification scheme “appropriate” in the above sense to “semantic-modification” structured editing, and fully automated layout (which requires a representation which supports a layout algorithm which can find a comprehensible layout for most valid programs in a reasonable amount of time and space). A more detailed discussion of the issues

addressed in SPARCL in three-dimensional representation are presented in chapter 5 (“Three-dimensional Representation”).

5. Sets With Partitioning Design Elements.

The decision to use sets with partitioning as the basic organization of data leads to several other design elements of SPARCL. Sets are unordered and this is convenient for those situations where one has unordered data. But, some way of expressing order is also useful and for this SPARCL uses N-tuples. The representation of N-tuples in SPARCL is shown in Figure 3.21 and Figure 3. 22. N-tuples are defined in SPARCL in terms of sets. An N-tuple ($N > 1$) is built recursively from the definition of an ordered pair²:

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

A 1-tuple has the “degenerate” definition:

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

The N-tuple is defined as:

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

Partitioning provides the necessary expressivity for all of the common set operations such as union, intersection, and difference. The SPARCL ‘Union’/3 predicate demonstrates this. This example is discussed in appendix 1 (“Tutorial Introduction to SPARCL”). We reproduce that discussion here.

The SPARCL predicate for Union/3 is shown in Figure 3. 26. 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

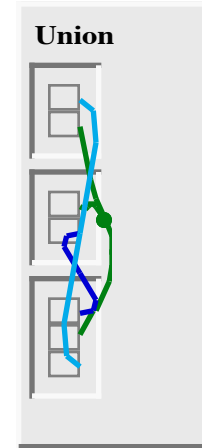


Figure 3.26: SPARCL Union/3 program.

2. This definition is from [Mendelson 1964], 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... $(x)(y)(u)(v)(\langle x, y \rangle = \langle u, v \rangle \rightarrow x = u \text{ and } y = v)$.”

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:

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

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

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 .

One can reason about how Union/3 specifies that $C = A \cup B$ as follows: Let the two parts of A be sets X and Y , the two parts of B be sets Y and Z , and the three parts of C by X , Y , and Z . Since X and Y form a partitioning of A , everything in A is in one or the other of them and they have no elements in common. Similarly for Y and Z with respect to B and X , Y , and Z with respect to C .

We can show that Y must be the intersection of A and B by contradiction. The two possible cases are that there is an element in the intersection which is not in Y , or that there is an element in Y which is not in the intersection. Suppose that there is some element E which is in the intersection of A and B but is not in Y . Because E is in A and B , then E must be in A . Because X and Y cover A , E is in A , and E is not in Y , then E must be in X . A similar line of reasoning shows that E must be in Z . Thus, the intersection of X and Z contains E . However, this contradicts the given fact that X , Y and Z are pairwise disjoint, i.e. that X and Z have an empty intersection. Thus, there can not be an element of the intersection of A and B which is not in Y .

Having handled the first case, suppose that there is a element F which is in Y but which is not in the intersection of A and B . Since Y is part of the cover of A , F must be in A . Similarly, F must be in B . Since F is in A and B , it must be in the intersection

of A and B , but this contradicts our assumption. Thus, there can not be an element of Y which is not in the intersection of A and B . This completes our proof that Y is the intersection of A and B .

Since X and Y disjointly cover A , then X must be A minus Y . Since Y is the intersection of A and B , it contains all of B which is in A , so A minus $Y = A$ minus B . Thus, X is A minus B . Similarly, Z is B minus A . This gives the desired equation:

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

In summary, the SPARCL Union/3 clause asserts:

$$\begin{aligned} (A = X \cup Y) \wedge (X \cap Y = \emptyset) \\ (B = Y \cup Z) \wedge (Y \cap Z = \emptyset) \\ (C = X \cup Y \cup Z) \wedge (X \cap Y = \emptyset) \wedge (X \cap Z = \emptyset) \wedge (Y \cap Z = \emptyset) \end{aligned}$$

The only solution of these three equations is:

$$\begin{aligned} X &= (A - B) \\ Y &= (A \cap B) \\ Z &= (B - A) \\ C &= (A - B) \cup (A \cap B) \cup (B - A) = A \cup B \end{aligned}$$

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

6. Logic Programming Design Elements.

The logic programming aspect of SPARCL leads to several design elements. SPARCL relies on Horn clauses (based on sets) in which to express the logic of problem solutions. The interpretation of a query and a set of clauses uses resolution theorem proving with a depth-first backtracking search (with delaying of literals). The resolu-

tion theorem proving relies on unification and logic variables. The coreference mechanism described in section 1, “Visual & Logic Design Elements”, is not part of the interpreter. The coreference links for a clause are translated into unification literals which are added to the body of that clause before being passed to the interpreter.

It uses negation as failure and the Closed World Assumption to express negation. The inference engine is a resolution theorem prover extended with constraint handling for the partitioning constraints and delay handling to support delaying the resolution of dynamically selected literals.

The delay mechanism is the major explicit ordering of execution which the SPARCL programmer can specify. The programmer provides delay specifications (discussed in section 1, “Visual & Logic Design Elements”) which the interpreter/inference engine uses to determine whether to delay evaluating/resolving a particular literal. The delay specification for a predicate P/K has the name P of the predicate in its first argument and a K-tuple of term-binding types in its second argument, a term-binding type for each argument of the predicate. These can be “variable”, “nonground”, “ignore”, or “subgoal”. The argument of a literal always matches the specification for that argument if the specified binding type is “ignore”. If the literal argument’s term is an unbound variable, then it matches all four types. A term is nonground if it is an unbound variable or if it is a set which contains a nonground term. If the literal argument’s term is nonground, then it matches “nonground” and “ignore” (but not “variable”). The “subgoal” type can be used with meta-predicates in one or more of their literal-containing arguments (e.g. “*DELAY*=>fails=>subgoal” delays a ‘fails’/1 literal that has a sub-literal that must be delayed). If the type is “subgoal”, then the literal term T matches if it is a variable. If T is not a variable, then it is considered as an N-tuple specifying a literal. The first element of T is the predicate name of this “sub-literal”, and N-1 is the arity of the sub-literal. If the sub-literal is delayed (applying the same process recursively), then the argument containing the sub-literal matches the “subgoal” type. There are many examples of the use of the delay mechanism in the ID3, WARPLAN, and Self Interpreter programs in chapter 7 (“Subjective Analysis”).

The other execution-ordering mechanisms are two built-in (meta)predicates ‘if’/3 and ordered_disjunction/2. The ‘if’/3 predicate evaluates the Test literal first, then it evaluates the Then or Else literals as appropriate. The ‘ordered_disjunction’/2 predicate evaluates its first argument first. On failure of the first argument, then the second

argument is evaluated. There are uses of these predicates in the *WARPLAN* example in chapter 7.

The Interpretation Procedure. The interpreter for SPARCL defines the procedural meaning of SPARCL programs. The interpreter is a procedure for executing a set of goals with respect to a given program. To

“execute goals” means try to satisfy them. We present here the discussion of the interpretation procedure given in section 2.4 of the tutorial (Appendix 1 (“Tutorial Introduction to SPARCL”)).

Let us call this procedure “execute set”. The inputs are a program, a goal set, and a set of partitioning constraints. The outputs are a success/failure indicator and an instantiation of variables. The success/failure indicator is “yes” if the goals are satisfiable and “no” otherwise. We say that “yes” signals a successful termination and “no” a failure. An instantiation of variables is only produced in the case of a successful termination; in the case of failure there is no instantiation. A discursive version of the procedure is shown in Figure 3. 27. This procedure can be written in a Pascal-like notation as shown in Figure 3. 28.

Several additional remarks are in order here regarding the procedures "execute_set" and “execute_list” as presented. First, we don’t explicitly describe how the final resulting instantiation of variables is produced. It is the instantiation *S* which led to a successful terminate, and was possibly further refined by additional instantiations that were done in the nested recursive calls to "execute_list".

To execute a set of goals $\{G1, \dots, Gm\}$ with partitioning constraints *P* the procedure “execute” does the following:

- Order the goal set to create a list $\{G1', \dots, Gm'\}$
- invoke "execute list".

To execute a LIST of goals $[G1', \dots, Gm']$ with partitioning constraints *P* the procedure "execute list" does the following:

- If the goal list is empty then if the delayed goals list is empty terminate with "success", else terminate with "failure".
- If the goal list is not empty then divide the list into the first goal, *G1'*, and the OtherGoals.
- If *G1'* is the special "marker" goal 'end_body', then replace it with the DelayedGoalsList ($[DG1, \dots, DGk]$) to create $[DG1, \dots, DGk, G2', \dots, Gm']$ and recursively invoke execute_list with this new goal list and an empty delayed goals list. The result of the recursive invocation is the result of this invocation.
- Else, if *G1'* is a goal which should be delayed according to the *DELAY* definitions in the program then add *G1'* to the DelayedGoals and recursively invoke execute_list with the OtherGoals ($[G2', \dots, Gm']$) and the extended DelayedGoals. The result of the recursive invocation is the result of this invocation.
- Otherwise (i.e. if GoalList is not empty, *G1'* is not 'end_body', and *G1'* does not need to be delayed) continue with (the following) operation called "SCANNING".

Figure 3. 27 (part 1 of 2): Interpreter procedure, discursive presentation.

- **SCANNING:** Scan through the clauses in the program in any order until a clause, C , is found such that the head of C matches the first goal $G1'$ without violating the current partitioning constraints P . If there is no such clause then terminate with "failure".

If there is such a clause C with head H and body goals $\{B1, \dots, Bn\}$, then replace the variables of C with new variables (essentially, rename the variables of C) to obtain a variable C' of C , such that C' and the list $G1', \dots, Gm'$ have no common variables. Let C' have head H' and body $\{B1', \dots, Bn'\}$. Let $G1'$ match H' ; let the resulting instantiation of variables be S and extend partitioning constraints P to be P' .

Order the goal set $\{B1', \dots, Bn'\}$ to create the goal list

$[B1'', \dots, Bn'']$.

In the goal list $[G1', \dots, Gm']$, replace $G1'$ with the list $[B1'', \dots, Bn'']$, obtaining a new list

$[B1'', \dots, Bn'', G2', \dots, Gm']$.

(Note that if C is a fact then $n = 0$ and the new goal list is shorter than the original one; such shrinking of the goal list may eventually lead to the empty list and thereby a successful termination.)

Substitute the variables in this new goal list with new values as specified in the instantiation S , obtaining another goal list

$[B1''', \dots, Bn''', G2'', \dots, Gm'']$

- Execute (recursively with procedure "execute list") this new goal list. If the execution of this new goal list terminates with success then terminate the execution of the original goal list also with success. If the execution of the new goal list is not successful then abandon this new goal list and go back to SCANNING through the program. Continue the scanning with any untried clause and try to find a successful termination using some other clause.

Figure 3. 27 (part 2 of 2): Interpreter procedure, discursive presentation.

Whenever the recursive call within SCANNING to "execute_list" fails, the execution returns to SCANNING, continuing at the program that had been last used before. As the application of the clause C did not lead to a successful termination SPARCL has to try an alternative clause to proceed. What effectively happens is that SPARCL abandons this whole part of the unsuccessful execution and backtracks to the point (clause C) where this failed branch of the execution was started. When the procedure backtracks to a certain point, all of the variable instantiations that were done after that point are undone. This ensures that SPARCL systematically examines all of the possible alternative paths of execution until one is found that eventually succeeds, or until all of them have been shown to fail.

The implementation of the interpreter used in SPARCL adds many refinements to the execute procedures in Figure 3. 27 or Figure 3. 28. One of these refinements reduces the amount of scanning through the program clauses so SPARCL will only examine the clauses about the relation in the current goal. We discuss the implementation of the interpreter and unification in chapter 6 ("Implementation").

7. Visual Programming, Logic Programming, and Set Design Elements.

The design of the handling of input and output for SPARCL is derived from all three basic principles. The constraints we have adopted for SPARCL's IO are: the concrete representation that is written or read must be non-linear, reading and writing must be backtrackable, and what is read or written is a set of terms. That the concrete representation that is written or read must be non-linear derives from visual programming, that reading and writing must be backtrackable derives from logic programming, and that what is read or written is a set of terms derives from the fun-

damental nature of sets in SPARCL. Our design and implementation of this aspect of SPARCL is not complete.

The basic concepts for SPARCL's IO are the “persistent term” and “term sets”. A persistent term is a SPARCL term that “persists” across invocations of SPARCL. We store persistent terms in persistent term sets (or simply “term sets”). A term set has an identifier that allows SPARCL to find it. In this view, a SPARCL program is simply a special kind of persistent term set, a set of N-tuples that is viewed by the SPARCL system as defining a set of clauses. The name of the program is the term set identifier.

Persistent terms are referenced during interpretation of a query by the persis-

```

procedure execute_set (Program, GoalSet, Constraints, Success)
begin
  OrderedGoalsList := order_goal_set(GoalSet);
  execute_list(Program, OrderedGoalsList, [], Constraints, Success);
end;

procedure execute_list (Program, GoalList, DelayedGoals,
                       Constraints, Success)
begin
  if empty(GoalList) then
    begin
      if empty(DelayedGoals) then
        Success := true
      else Success := false
      end
    end
  else
    begin
      Goal := head(GoalList);
      OtherGoals := tail(GoalList);
      if Goal = end_body then
        begin
          NewGoals := append(DelayedGoals, OtherGoals);
          execute_list(Program, NewGoals, [], Constraints, Success);
        end
      else if delay(Goal, Program) then
        begin
          NewDelayedGoals := append(DelayedGoals, [Goal]);
          execute_list(Program, OtherGoals, NewDelayedGoals,
                       Constraints, Success);
        end
      else
        begin
          Satisfied := false;
          while not Satisfied and "more clauses in program" do
            begin
              Let next clause in Program be
              head H and body {B1, ..., Bn}.
              Construct a variant of this clause
              head H' and body {B1', ..., Bn'}.
              match(Goal, H', Constraints, MatchOK, Instant,
                   MatchConstraints);
              if MatchOK then
                begin
                  OrderedBodyGoals := order_goal_set({B1', ..., Bn'});
                  ExtendedBodyGoals := append(OrderedBodyGoals,
                                              [end_body]);
                  NewGoals := append(ExtendedBodyGoals, OtherGoals);
                  NewGoals := substitute(Instant, NewGoals);
                  execute_list(Program, NewGoals, MatchConstraints,
                              Satisfied);
                end
              end;
            end
            Success := Satisfied
          end
        end
      end;
    end
  end;
end;

```

Figure 3.28: execute_set and execute_list procedures.

tent_term/2 built-in. If a persistent term with the given identifier has already been referenced in the current query processing, then the second argument of persistent_term/2 must unify with that already referenced term. If the given identifier has not been used in the current query processing, then a new entry is added to the persistent term table (kept internally by the interpreter). If there is a term set program in the SPARCL environment with the given identifier, then the contents of that program become the “value” of the persistent term in the table, otherwise the initial value is an unbound variable. In either event, the value of the new entry is unified with the second argument of persistent_term/2. When the query processing terminates successfully, persistent terms in the persistent term table are written to term set programs in the SPARCL environment. If a term set program’s contents were modified by the query, then that term set program is replaced by the new value. If there isn’t already a term set program corresponding to a persistent term’s identifier in the persistent term table of the query, then one is created with that persistent term’s corresponding value. The term set programs may be saved or discarded at the user’s discretion. The result of a query (the head of the clause being queried as bound by successful termination of the query) is written to a term set program with an identifier that is built from the name of the program containing the clause being queried.

The construction of the persistent term table is backtrackable, so the construction of a persistent term only reflects the successful proof path used by the query and none of the unsuccessful branches. The construction of persistent terms across queries is *not* backtrackable. This is analogous to delayed output in transaction processing, where no output is made visible until the transaction commits. As in the transaction case, this kind of output is not always useful. For instance, one cannot see “progress reports” that give the user an idea where the interpreter is in the process of solving a complex query. To deal with this latter case, we implemented two textual writing predicates in SPARCL, write/1 and grounded_write/1. We need two predicates because of the nature of ordering of execution in SPARCL. A write/1 literal will execute whenever the interpreter selects it. If one wants to write out some interim result, the grounded_write/1 predicate might be more appropriate. It has a built-in delay specification that prevents a grounded_write/1 literal from being interpreted until its argument is ground. Thus, it won’t write anything out until it has a fully grounded term to write. In either case, they write a readable linear representation of the argument followed by a newline to the ‘*Output*’ window of the SPARCL environment.

We have no solution for reading terms interactively during the interpretation of a query. Thus, the basic interactive cycle of prompting the user for some input, reading the input, processing it, reporting the result, and starting the cycle over, is not possible in interpreting a single SPARCL query. This is an area we hope to address in the future.

Programming in SPARCL

This section is an introduction to programming in SPARCL.³ Several key concepts are introduced in this section: clauses, facts, rules and running queries.

Facts. Consider the ‘Parent’/2 predicate defined by the six clauses in Figure 3. 29. This is a simple "database"-style

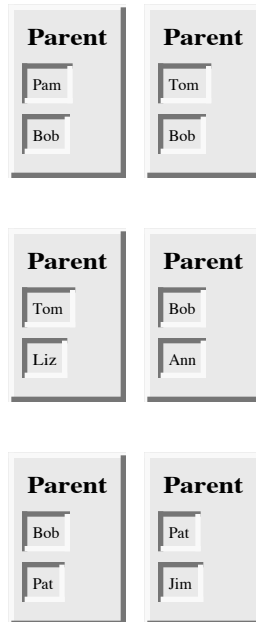


Figure 3. 29: ‘Parent’/2 predicate definition.

program in that all of the clause are facts about the ‘Parent’ relationship: Pam is Bob’s parent, Tom is Liz’s parent, Bob is Pat’s parent, and so forth.

We use this program by “querying” it. A query is written in SPARCL as a clause, generally with one or more literals in it. One such query is shown in Figure 3. 30. This query poses the question

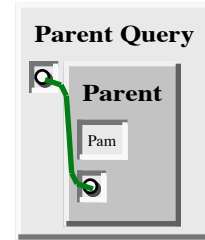


Figure 3. 30: ‘Parent Query’/1 clause: “Who is Pam’s child?”

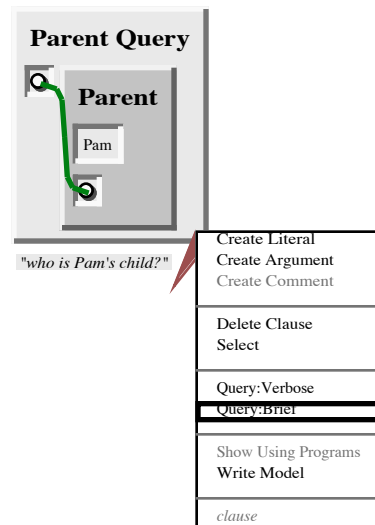


Figure 3. 31: Interaction to query SPARCL “Who is Pam’s child?”

“Who is Pam’s child?”. This ‘Parent Query’/1 clause is true if the ‘Parent’/2 predicate is true with a first argument of ‘Pam’. The interpreter for SPARCL not only determines that ‘Parent’/2 is true, but also keeps track of the set of variable bindings used to make that determination. When it succeeds with the ‘Parent Query’/1 clause of Figure 3. 30 as its query, then it displays an N-tuple of ‘Parent Query’/1 with the appropriate bindings. The interaction to request SPARCL to evaluate a clause as a query is shown in Figure 3. 31. The user has selected the “Query:Brief” option (to suppress trace information). The result of this query is displayed by SPARCL in a special

3. This material is discussed in greater detail in the Appendix 1 (“Tutorial Introduction to SPARCL”).

Parent Query → Bob

Figure 3. 32: Query result for “Who is Pam’s child?”

window. It is the ordered pair (2-tuple) shown in Figure 3. 32. If there were several possible answers to this query, SPARCL would

only report the first one it encountered. To get a result that shows all of the possible answers, one needs to construct a query using a ‘setof’/3 predicate or an intensional set. The clause for the query “Who is Liz’s parent” and the result N-tuple are shown in Figure 3. 33.

This clause is very similar in structure to the previous example, and interpreter answers that ‘Tom’ is a parent of ‘Liz’.

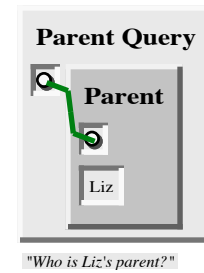
A more complex example query, “Who is the grandparent of Jim?”, is shown in Figure 3. 34. This query asks for Jim's grandparent "X". Since our program does not directly know the "Grandparent" predicate, this query has to be broken down into two parts:

- (1) Who is a parent of Jim? Assume that this is some Y.
- (2) Who is a parent of Y? Assume that this is some X.

We ask a query of two parts by putting a literal for each part in the body of the same query clause and connecting the variables of these literals appropriately.

Some major points from the preceding discussion:

- It is easy in SPARCL to define a predicate, such as the ‘Parent’/2 predicate, by stating the N-tuples of objects that satisfy the predicate.
- The user can easily query the SPARCL system about predicates defined in the program.
- A SPARCL program consists of 'clauses'.
- The arguments of predicates can (among other things) be: concrete objects (such as "Tom" and "Ann"), or general objects which are represented by small circles. Objects of the first kind are called "atoms" and objects of the second kind are called "variables".
- Questions to the system consist of a clause, which may contain any number of 'literals'. Several literals in the body of a single clause means that clause is true when the conjunction of the literals is true.
- An answer can be either positive or negative: for a positive answer we say that



Parent Query → Tom

Figure 3. 33: Query clause and result N-tuple for “Who is Liz’s parent?”

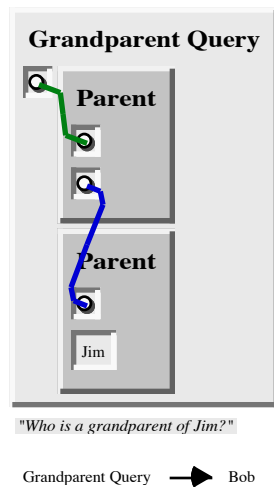


Figure 3.34: Query clause and result for "Who is the grandparent of Jim?"

the query was "satisfiable" and it "succeeded"; for a negative answer we say that the query was "unsatisfiable" and it "failed".

- If a query has several possible answers then SPARCL will find one of them.

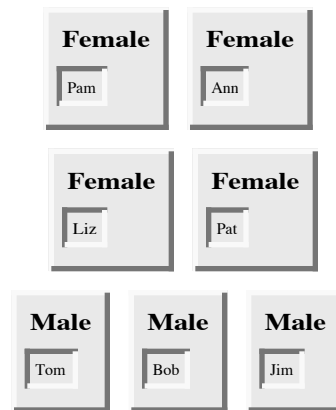


Figure 3.35: Definitions for the 'Female'/1 and 'Male'/1 predicates. Separate clauses version.

Rules. The "Parents" program can be extended in many interesting ways. Let us first add the information on the sex of the people that occur in the 'Parent'/2 predicate. This information has already been entered and saved in a program file, so we can simply open that program.

Definitions for the 'Male'/1 and 'Female'/1 predicates are shown in Figure 3.35. These predicates are unary (or one-place) predicates. A binary predicate like 'Parent'/2 defines a predicate between *pairs* of objects; on the other hand, unary predicates can be used to declare simple yes/no properties of objects.

These 'Female'/1 and 'Male'/1 predicates can be displayed in a different, more compact, way. This alternative representation uses SPARCL's "fact tables". Figure 3.36 has "fact table" versions of the 'Female'/1 and 'Male'/1 predicates. These tables have the same meaning as the collection of shown the in Figure 3.35. The fact table has the predicate name for all of the facts of the table placed in the upper left-hand corner of the table. Each row of the table is a single "fact" - a clause with an empty "body".

The 'Offspring'/2 predicate is the inverse of the 'Parent'/2 predicate. We could define 'Offspring'/2 in a similar way as the 'Parent'/2 predicate; that is, by simply providing a list of simple facts about the 'Offspring'/2 predicate, each fact

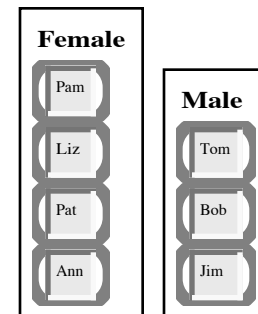


Figure 3.36: 'Female'/1 and 'Male'/1 predicates.

mentioning one pair of people such that one is an offspring of the other.

However, the offspring predicate can be defined much more elegantly making use of the fact that it is the inverse of 'Parent', and that 'Parent' has already been defined. This alternative way can be based on the following logical

statement: "For all X and Y, Y is an offspring of X if X is a parent of Y." The clause in Figure 3.37 represents the preceding "logical statement". This kind of clause is called a "rule".

There is an important difference between facts and rules. A fact like those shown in the 'Parent' clauses is something that is always, unconditionally, true. On the other hand, rules specify things that are true if some condition is satisfied. Therefore we say rules have:

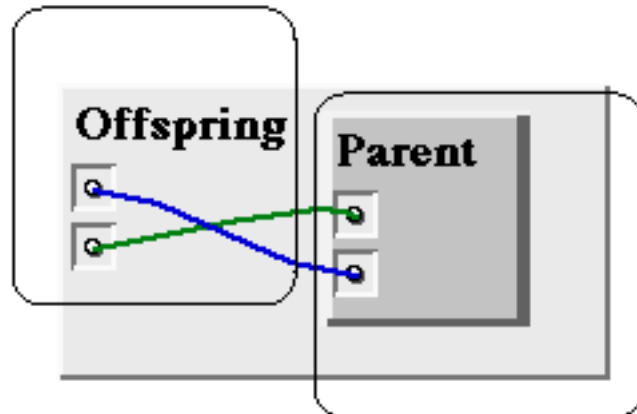
- a condition part (the right-hand side of the clause)
- a conclusion part (the predicate name and arguments on the left-hand side of the clause).

The conclusion part is also called the "head" of a clause and the condition part the "body" of the clause.

How rules are actually used by SPARCL is illustrated by the following example. Let us ask our program whether Liz is an offspring of Tom. The "query" clause in Figure 3.38 represents our question.

There is no fact about offsprings in the program, therefore the only way to consider this question is to apply the rule about offsprings. The rule is general in the sense that it is applicable to any two objects; therefore it can also be applied to such particular objects as "Liz" and "Tom". We say that the variables become instantiated.

The conclusion part of the rule.



The condition part of the rule.

Figure 3.37: Clause with labeled parts defining the 'Offspring' predicate.

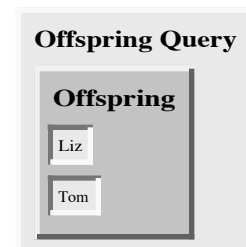


Figure 3.38: Clause for the query "Is Liz an offspring of Tom?"

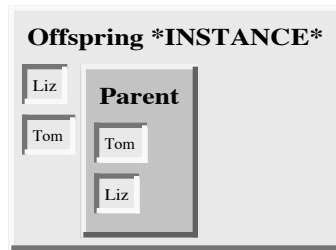


Figure 3.39: Instantiation of Offspring rule clause. Created in response to the query literal of Figure 3.38.

Figure 3.39 is the special case of the Offspring rule clause after instantiation of the general rule in Figure 3.37 in satisfying the query literal from

Figure 3.38.

The single literal of the "Offspring *INSTANCE*" body becomes the new goal for SPARCL to solve. It is trivial to solve as it can be found as a fact in the 'Parent'/2 program. This means that the conclusion part of the rule is also true, and SPARCL will succeed in executing the original query.

In Figure 3.40 there are two similar clauses defining two different predicates. The "Mother" clause, which defines the 'Mother'/2 predicate, shows a rule with two literals in its body, 'Parent'/2 and 'Female'/2. The "Grandparent" clause, which defines the 'Grandparent'/2 predicate, shows a rule which uses the same predicate, 'Parent'/2, twice in the literals of its body.

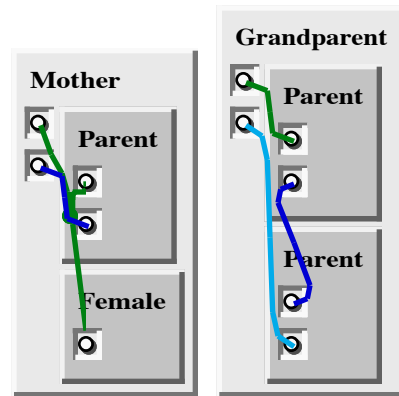


Figure 3.40: The clauses defining the 'Mother'/2 and 'Grandparent'/2 predicates.

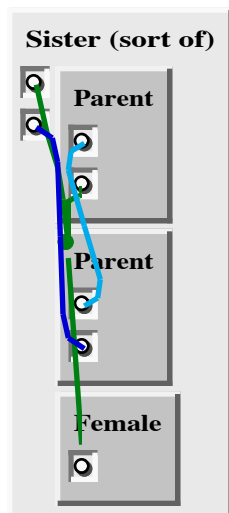


Figure 3.41: Clause defining the 'Sister (sort of)'/2 predicate.

The "Sister (sort of)" clause in Figure 3.41 defines the relationship of someone as the sister of someone if these two people have the same parent. This clause actually is slightly flawed - it allows for someone to be sister to herself. The "Sister" clause in Figure 3.42 fixes this using the 'Different'/2 predicate.

The 'Different'/2 predicate, shown in Figure 3.43, relies on a partitioned set constraint to specify the the terms

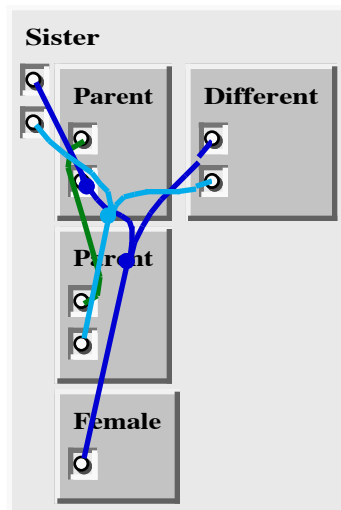


Figure 3.42: Clause defining 'Sister'/2 predicate.

in its two arguments are different. The `*TERM*/1` predicate is a built-in predicate of the SPARCL system. It is always true. It is used here to introduce the partitioned set constraint. Since the two "sisters" are in different parts of a partition, they must be different (since parts of a partition are disjoint sets). This is a kind of "not equal" constraint. We will discuss partitioned sets in more detail in a later section.

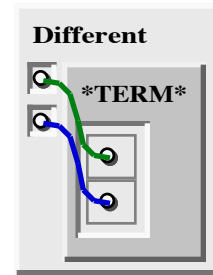


Figure 3.43:
'Different'/2 predicate definition.

The `'Predecessor'/2` predicate is defined by the two `'Predecessor'/2` clauses and a `*DELAY*/2` clause, as shown in

Figure 3.44. This predicate is an example of *recursion*, multiple clauses to defining a single predicate, and specifying a *delay* condition. A recursive definition of a predicate uses the predicate being defined in the body of one or more of the defining clauses of that predicate. Some person X is the Predecessor of some other person Z if X is a Parent of Z (this is one of the clauses), OR if X is the parent of some person Y, and Y is a predecessor of Z (this is the other one of the clauses).

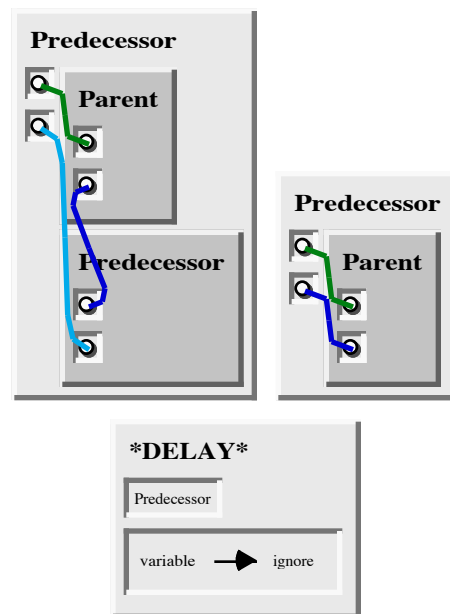


Figure 3.44: Clauses defining the not-quite-satisfactory version of the `'Predecessor'/2` predicate.

The `*DELAY*/2` predicate is used by the SPARCL interpreter in deciding when to evaluate goal literals. This `*DELAY*/2` clause instructs the interpreter to “delay” the evaluation of a `'Predecessor'/2` goal if the first argument is a variable (i.e. an unbound variable term). This is necessary in the case of the `'Predecessor'/2` predicate to prevent the interpreter from trying to solve the `'Predecessor'/2` literal before it has solved the `'Parent'/2` literal. This prevents the interpreter from recursing forever. Unfortunately, this is too restrictive. This restricts us to using the `'Predecessor'/2` predicate to ask the question "Who is X the predecessor of?". We would like to also be able to use `'Predecessor'/2` to answer the question "Who is X's predecessor?".

Another version of the `'Predecessor'/2` program is shown in Figure 3.45 that allows us to ask both of these questions. This version of the `'Predecessor'/2` program

allows us to ask both of the questions mentioned before. The difference between this version and the earlier version small: There is a new predicate, "Predecessor Recursion" which is simply a "wrapper" for 'Predecessor'/2; this predicate is used in the recursive clause of 'Predecessor'/2; and the '*DELAY*'/2 clause now refers to this new predicate 'Predecessor Recursion'/2 instead of the 'Predecessor'/2 predicate. Now, the inter-

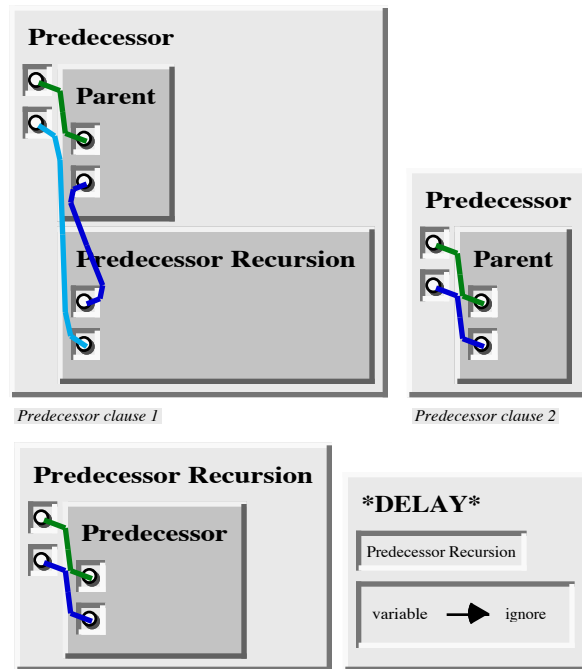


Figure 3.45: Clauses defining the correct version of the 'Predecessor'/2 predicate.

preter will attempt to solve a 'Predecessor'/2 goal which has an unbound variable first argument (which would have been delayed in the definition in Figure 3.44), but it will not recurse infinitely in the attempt since the 'Predecessor Recursion'/2 literal will be delayed when its first argument is an unbound variable.

The '*DELAY*'/2 clauses are the primary method the SPARCL programmer has to control the order in which the SPARCL interpreter attempts to solve goals. This is a particularly important facility when writing recursive predicates to ensure that the recursion terminates.

Important points of this section are:

- SPARCL programs can be extended by simply adding new clauses.
- SPARCL clauses are of three types: facts, rules, and questions.
- Facts declare things that are always, unconditionally true.
- Rules declare things that are true depending on a given condition.
- By means of questions the user can ask the program what things are true.
- SPARCL clauses consist of the "head" and the "body". The head is the name of the predicate and the arguments placed on the left side of the clause. The body is a set of "literals". These literals are understood to be joined by conjunctions.

- Facts are clauses that have a head and the empty body. Rules have the head and the (non-empty) body. A question is a "rule" clause which the programmer chooses to query.
- In the course of computation, a variable can be substituted by another object. We say that a variable becomes "instantiated".
- Variables are assumed to be universally quantified and are read as "for all". Alternative readings are, however, possible for variables that appear only in the body. These can be read as "some" (existential) variables.
- Recursion may be used in defining SPARCL predicates.
- The "***DELAY***" clause may be used to control the order in which the SPARCL interpreter attempts to solve goals.

How SPARCL works.⁴ A question to SPARCL is always a set of one or more goal literals. To answer a question, SPARCL tries to satisfy all of the goals. What does it mean to *satisfy* a goal? To satisfy a goal means to demonstrate that the goal is true, assuming that the predicates in the program are true. In other words, to satisfy a goal means to demonstrate that the goal *logically follows* from the facts and rules in the program. If the question contains variables, SPARCL also has to find what are the particular objects (in place of variables) for which the goals are satisfied. The particular instantiation of variables to these objects is displayed to the user. If SPARCL cannot demonstrate for some instantiation of variables that the goals logically follow from the program, then SPARCL's answer to the question will be "no".

An appropriate view of the interpretation of a SPARCL program in mathematical terms is then as follows: SPARCL accepts facts and rules as a set of axioms, and the user's question as a *conjectured theorem*; then it tries to prove this theorem—that is, to demonstrate that it can be logically derived from the axioms.

We will illustrate this view by a classical example. Let the axioms be:

All men are fallible.

Socrates is a man.

A theorem that logically follows from these two axioms is:

Socrates is fallible.

The first axiom above can be rewritten as:

For all X, if X is a man the X is fallible.

The example can be translated into SPARCL as shown in Figure 3. 46.

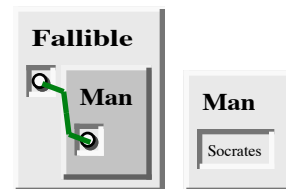


Figure 3. 46: ‘Fallible’/1 and ‘Man’/1 predicates definitions.

Now we ask SPARCL the question of Socrates' fallibility by querying the ‘Fallible Socrates Query’/0 clause in Figure 3. 47, which succeeds.

To discuss how SPARCL works, we use the concept of a *proof sequence*. Given some program (a set of clauses), a sequence of facts can be constructed starting with any *fact* in the program, then successively



Figure 3. 47: Query clause for “Is Socrates fallible?”

4. The text of this section is adapted for SPARCL from section 1.4 of Ivan Bratko's "Prolog Programming for Artificial Intelligence", 2nd edition.

adding other facts from the program or any fact derived using a *rule* of the program and any facts already in the sequence. A goal (or ‘literal’) is *satisfied* if such a sequence of facts can be found which ends with that goal. Let us call such a sequence of facts a *proof sequence*. SPARCL finds an appropriate proof sequence to satisfy a query.

SPARCL searches for a proof sequence satisfying a given goal by starting with that goal and working “backward” to facts of the program. Instead of starting with simple facts given in the program, SPARCL starts with the given goals and, using rules, substitutes the current goals with new goals, until new goals happen to be simple facts (or unify with program facts). Let’s look at an example using ‘Predecessor’/2 and ‘Parent’/2 programs.

How does SPARCL “solve” the query asking if Tom is Pat's predecessor? We start with the initial query shown in Figure 3. 48. This initial query clause provides a single goal literal, “Predecessor(Tom, Pat)”. There are two rule clauses which have heads (consequents) which unify with this goal literal. These rules are labeled "Predecessor clause 1" and "Predecessor clause 2". SPARCL may try either of these clauses first—let’s suppose SPARCL tries the rule of "Predecessor clause 2". It unifies the goal literal with the head of the rule, which binds the variables in the head. Since these variables corefer with variables in the body (antecedent) of the rule, these coreferring variables are also bound. This instantiates the rule clause as shown in Figure 3. 49.

For this instantiation of ‘Predecessor’/2 clause 2 to be applicable in building a proof sequence, the literal in its body must precede the application of this rule in the proof sequence. Thus, SPARCL determines that it must find a proof sequence for this literal. This yields the "query 2" version of the initial query shown in Figure 3. 50. The original goal of "Is Tom Pat's predecessor?" has been replaced by the goal of "Is Tom Pat's parent?".

There is no clause (fact or rule) with a head that matches the ‘Parent’/2 literal in the body of query 2 in Figure 3. 50, so SPARCL fails to solve this goal and it *backtracks*

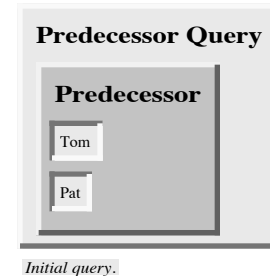


Figure 3. 48: Query clause for “Is Tom Pat's predecessor?”

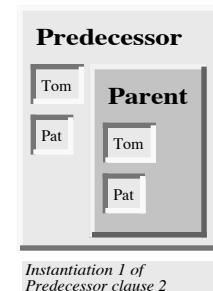


Figure 3. 49: First instantiation of ‘Predecessor’/2 clause 2.

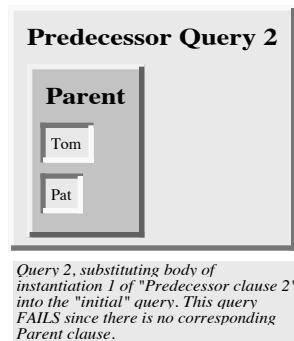


Figure 3.50: Revised version of 'Predecessor Query'/0 (Figure 3.48) based on substituting the body of instantiation 1 of 'Predecessor'/2 clause 2 (Figure 3.49) for the literal in the body of the original 'Predecessor Query'/0.

to try an alternative way to derive the top goal ("Is Tom Pat's predecessor?").

There were originally two ways to solve this top goal, and having failed to solve it using the rule "Predecessor clause 2" SPARCL

now tries rule "Predecessor clause 1". As was done when using "Predecessor clause 2", SPARCL unifies the

goal literal with the head of the rule ("Predecessor clause 1"), which binds the variables in the head. Again, since these variables corefer with variables in the body (antecedent) of the rule, these coreferring variables are also bound. This instantiates the rule clause as shown in Figure 3.51.

Substituting the body of instantiation 1 of "Predecessor clause 1" for the literal of the original query yields the new version of the query (query 3) shown in Figure 3.52. The goal literals, in query 3, share an uninstantiated variable.

SPARCL must now solve the two literals in the body of this revised query. SPARCL is free to satisfy them in any order. Suppose SPARCL tries the 'Predecessor Recursion'/2 literal first. It checks the *DELAY* clauses and finds that 'Predecessor Recursion'/2 is to be delayed if the first argument is an unbound variable. This is the case in query 3, so SPARCL delays solving the 'Predecessor Recursion'/2 literal and tries the 'Parent'/2 literal. This one is easily satisfied by matching the fact that "Tom is Bob's parent." This matching binds the shared

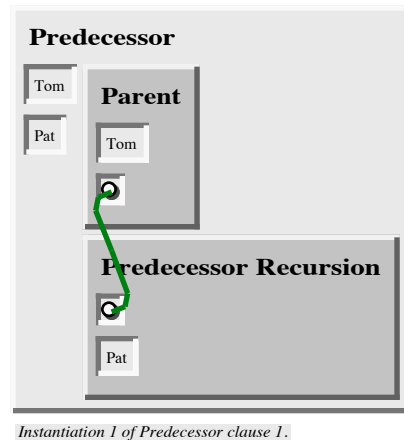


Figure 3.51: First instantiation of 'Predecessor'/2 clause 1.

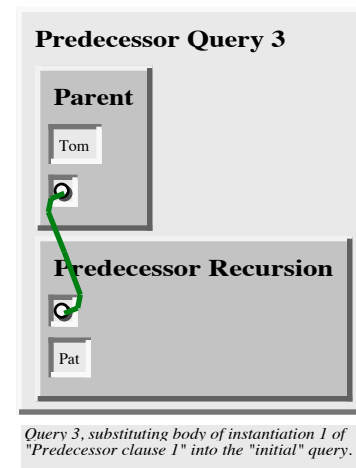


Figure 3.52: Revised version of 'Predecessor Query'/0 (Figure 3.50) based on substituting the body of instantiation 1 of 'Predecessor'/2 clause 1 (Figure 3.51) for the literal in the body of the original 'Predecessor Query'/0.

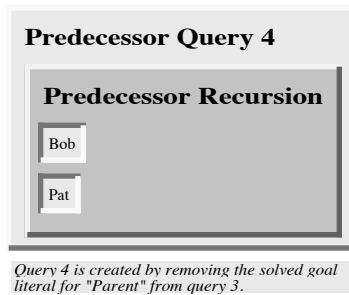


Figure 3. 53: Revised version of 'Predecessor Query 3'/0 (Figure 3. 52) based on removing the solved literal and replacing the remaining variable with its binding ("Bob").

variable to be "Bob". Removing the "solved" literal "Tom is Bob's parent" from query 3 leaves us with query 4 (

Figure 3. 53).

SPARCL solves

the literal of query 4 by instantiating 'Predecessor Recursion'/2 as shown in Figure 3. 54. It had

"delayed" attempting the solution of this literal earlier because the goal had a variable first argument and there is a *DELAY* clause which specifies this situation as requiring such a goal to be delayed. However, the goal literal no longer has a variable first argument, it is now "Bob". So, SPARCL need not delay solving this literal.

Query 5 in Figure 3. 55 is created by substituting the instantiation of 'Predecessor Recursion'/2 in Figure 3. 54 into query 4 in Figure 3. 53.

SPARCL solves the literal of query 5 by instantiating "Predecessor clause 2" in Figure 3. 56, similar to the first attempt at solving the initial query in Figure 3. 49.

Query 6 in Figure 3. 57 is created by substituting instantiation 2 of 'Predecessor'/2

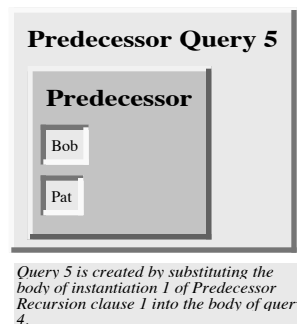


Figure 3. 55: Revised version of 'Predecessor Query 4'/0 (Figure 3. 53) based on substituting the body of instantiation 1 of 'Predecessor Recursion'/2 clause (Figure 3. 54) for the literal in the body of 'Predecessor Query 4'/0.

clause 2 in Figure 3. 56 into query 5. The 'Parent'/2 literal of query 6 matches a 'Parent'/2 fact, and thus is true.

This completes the search for a proof sequence.

The sequence being: the literal of query 6 and instance 2 of Predecessor clause 2 derives the literal of query 5; the literal of query 5 and instance 1 of "Predecessor Recursion" derives the literal of query 4; the literal of query 4 and fact "Tom is Bob's parent" derive

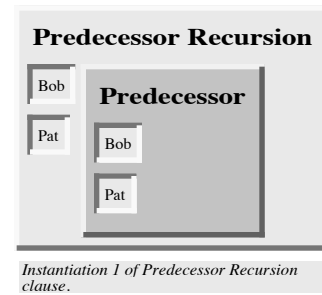


Figure 3. 54: Instantiation 1 of 'Predecessor Recursion'/2.

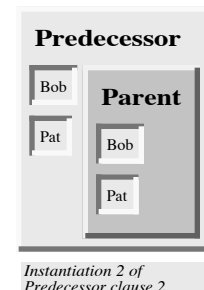


Figure 3. 56: Second instantiation of 'Predecessor'/2 clause 2.

the literals of query 3; the literals of query 3 and instance 1 of "Predecessor clause 1" derive the the literal of initial query.

The trace of the execution of SPARCL can be pictured as a "tree". The nodes of the tree correspond to goal literals, or lists of goal literals, that are to be satisfied. The arcs between the nodes correspond to the application of (alternative) program clauses that transform the goals at one node into the goals at another node. The top goal is satisfied when a path is found from the root node (top goal) to a leaf node labelled "success". A leaf is labelled "success" if it is a simple fact. The execution of SPARCL programs is the searching for such paths. During the search SPARCL may enter an unsuccessful branch. When SPARCL discovers that a branch fails it automatically backtracks to the previous node and tries to apply an alternative clause at that node.

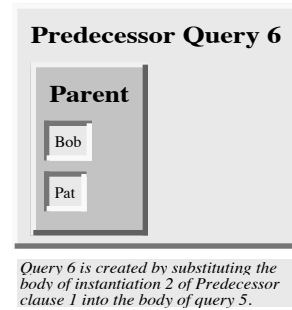


Figure 3. 57: Revised version of 'Predecessor Query 5'/0 (Figure 3. 55) based on substituting the body of instantiation 2 of 'Predecessor'/2 clause (Figure 3. 56) for the literal in the body of 'Predecessor Query 5'/0.

What SPARCL programs mean. There are two different ways to think about the meaning of SPARCL programs.⁵ In the examples so far it has always been possible to understand the results of the program without exactly knowing *how* the system actually found the results. It therefore makes sense to distinguish between two levels of meaning of SPARCL programs; namely,

- the *declarative meaning* and
- the *procedural meaning*.

The declarative meaning is concerned only with the *predicates* defined by the program. The declarative meaning thus determines *what* will be the output of the program. On the other hand, the procedural meaning also determines *how* this output is obtained; that is, how are the predicates actually evaluated by the SPARCL system.

The ability of SPARCL to work out many procedural details on its own is considered to be one of its specific advantages. It encourages the programmer to consider the declarative meaning of programs relatively independently of their

5. The text of this section is largely drawn from section 1.5 of Ivan Bratko's "Prolog Programming for Artificial Intelligence", 2nd edition.

procedural meaning. Since the results of the program are, in principle, determined by its declarative meaning, this should be (in principle) sufficient for writing programs. This is of practical importance because the declarative aspects of programs are usually easier to understand than the procedural details. To take full advantage of this, the programmer should concentrate mainly on the declarative meaning and, whenever possible, avoid being distracted by the executional details. These should be left to the greatest possible extent to the SPARCL system itself.

This declarative approach indeed often makes programming in SPARCL easier than in typical procedurally oriented programming languages such as Pascal. Unfortunately, however, the declarative approach is not always sufficient. It will later become clear that, especially in large programs, the procedural aspects cannot be completely ignored by the programmer for practical reasons of executional efficiency. Nevertheless, the declarative style of thinking about SPARCL programs should be encouraged and the procedural aspects ignored to the extent that is permitted by practical constraints.

Summary of facts and rules section:

- SPARCL programming consists of defining predicates and querying about predicates.
- A program consists of *clauses*. These are of two types: *facts* and *rules*. A clause can be used to ask a *question*.
- A predicate can be specified by *facts*, simply stating the N-tuples of objects that satisfy the predicate, or by stating *rules* about the predicate.
- A *procedure* (also called a “predicate”) is a set of clauses about the same predicate.
- Querying about predicates, by means of *questions*, resembles querying a database. SPARCL's answer to a question consists of a set of objects that satisfy the question.
- In SPARCL, to establish whether an object satisfies a query is often a complicated process that involves logical inference, exploring among alternatives and possibly *backtracking*. All this is done automatically by the SPARCL system and is, in principle, hidden from the user.
- Two types of meaning of SPARCL programs are distinguished: declarative and procedural. The declarative view is advantageous from the programming point of view. Nevertheless, the procedural details often have to be considered by the programmer as well.

- The following concepts have been introduced in this section: clause, fact, rule, question; the head of a clause, the body of a clause; recursive rule, recursive definition; procedure; constant, variable; instantiation of a variable; goal (literal); goal is satisfiable, goal succeeds; goal is unsatisfiable, goal fails; backtracking; declarative meaning, procedural meaning.

Programming with N-tuples.

In this section of our introduction to programming with SPARCL we look at programming with N-tuples. We discussed the N-tuples above using Figure 3.21 and Figure 3.22. Here we discuss some simple uses of N-tuples.

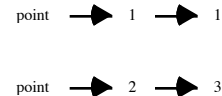


Figure 3.58: Two triples representing the points (1,1) and (2,3).

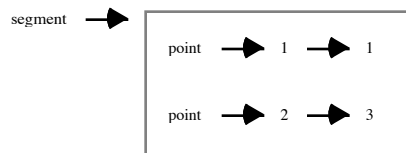


Figure 3.59: An ordered pair representing a line segment with endpoints at (1,1) and (2,3).

Simple geometry representation. The two most widely used set representations are partitioned sets and N-tuples. We can use these to conveniently represent many different kinds of structures. In this section we see how some simple geometric objects can be represented.

The two 3-tuples in Figure 3.58 show a way to represent geometric points, one with $X = 1$ and $Y = 1$ and the other with $X = 2$ and $Y = 3$. An N-tuple is used here to provide a concise “naming” via element position to distinguish the X and the Y value.

The ordered pair (2-tuple) in Figure 3.59 represents a line segment with endpoints at (1,1) and (2,3). The 2-tuple is used here to “name” the endpoint data (telling us it defines a “segment”). The endpoint data is placed in a set, rather than in more elements of a N-tuple. This reflects the fact that there is no distinction between the two endpoints (this isn't a directed line).

The ordered pair in Figure 3.60 represents a triangle with three “corners” at (4,2), (6,4), and (7,1). The ordered pair is used to “name” the collection of points as describing a “triangle”. The points are in a set since no ordering of the points is

needed.

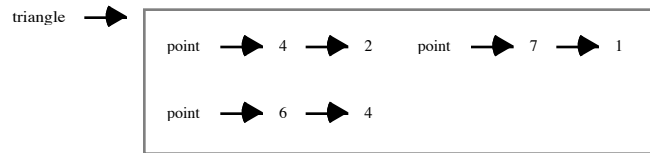


Figure 3.60: An ordered pair representing a triangle with corners at (4,2), (7,1), and (6,4).

Term matching. The most important operation on terms is *matching*. A special kind of matching is used in SPARCL called *unification*. Matching alone can produce some interesting computation.

Given two terms, we say they *match* if:

- (1) they are identical
- (2) the variables in both terms can

be instantiated to objects in such a way that after the substitution of variables by these objects the terms become identical, and these instantiations don't violate any "current" partitioning constraints.

Using example predicates which define properties of line segments, we illustrate how matching alone can be used for interesting computation.

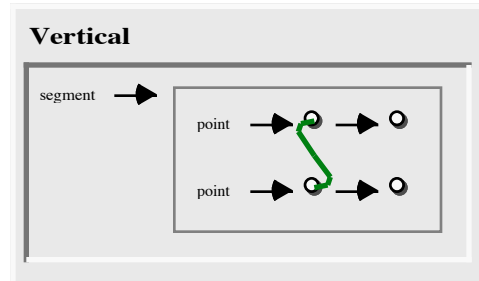


Figure 3.61: Clause defining the 'Vertical'/1 predicate.

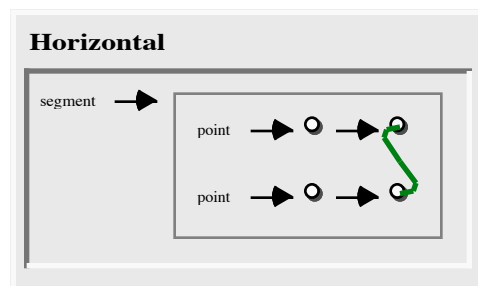


Figure 3.62: Clause defining the 'Horizontal'/1 predicate.

Let us return to the simple geometric objects of the previous example and define a piece of program for recognizing horizontal and vertical line segments. The 'Vertical'/1 predicate in Figure 3.61 is a property of segments, so it can be formalized in SPARCL as a unary relation. A segment is vertical if the x-coordinates of its end-points are equal, otherwise there is no other restriction on the segment. The property 'Horizontal'/1 is similarly formulated in Figure 3.62, with only the x and y interchanged.

The query in Figure 3.63 asks for the y value of a point such that there is a horizontal segment from (1,1) to that point with x = 2. The result in Figure 3.64 shows the desired y value to be 1.

Programming with Intensional Sets.

Next we discuss the ‘Column Sum’/3 predicate shown in Figure 3. 65. It finds the sum of the values in a column of a table. This predicate uses an intensional multiset to express its central idea: gathering together a multiset of column values. Abstractly, it takes a function term table and a domain value and produces the sum of all of the range values for that domain value in the function term table. This is analogous to finding the sum of the values of a given column of a spreadsheet. This predicate’s implementation demonstrates the use of several aspects of SPARCL including: term tables, intensional sets, multisets, and arithmetic.

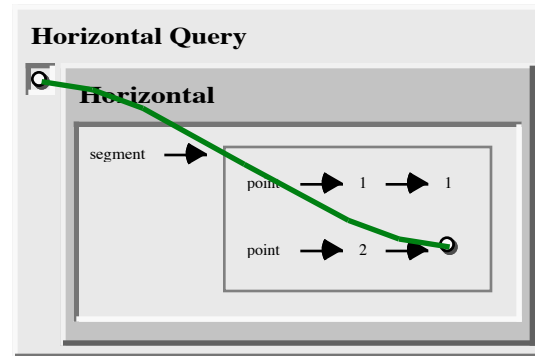


Figure 3.63: Clause for posing the query “What is the value of Y such that there is a horizontal segment from (1,1) to (2,Y).”

Horizontal Query → 1

Figure 3.64: Result of querying the clause in Figure 3.63.

The ‘Column Sum’/3 predicate. This predicate has three arguments. The single clause in its definition has descriptive comments for each argument: the first argument is a “Function Table”; the second argument is the “Domain Value Identifying Range Values to Sum”; the third argument is the “Sum of Identified Range Values.” Each of the predicate argument terms is a variable.

The body of this predicate’s defining clause contains a single literal. This literal refers to the ‘is’/2 predicate. The ‘is’/2 predicate is the general arithmetic evaluation predicate. It takes an arithmetic expression as its second argument and unifies the evaluation of this expression with its first argument. In this case, it does the arithmetic to determine the sum. Expressions are represented as N-tuples where the first (left-most) argument is the operator and the other arguments are the operands. There are both unary and binary operators; an expression using a unary operator is represented using an ordered pair and an expression using a binary operator is represented using an ordered triple. The expression in the ‘Column Sum’/3 clause is the unary ‘+’ with a single operand of the multiset of values of the desired column. The unary ‘+’ with a

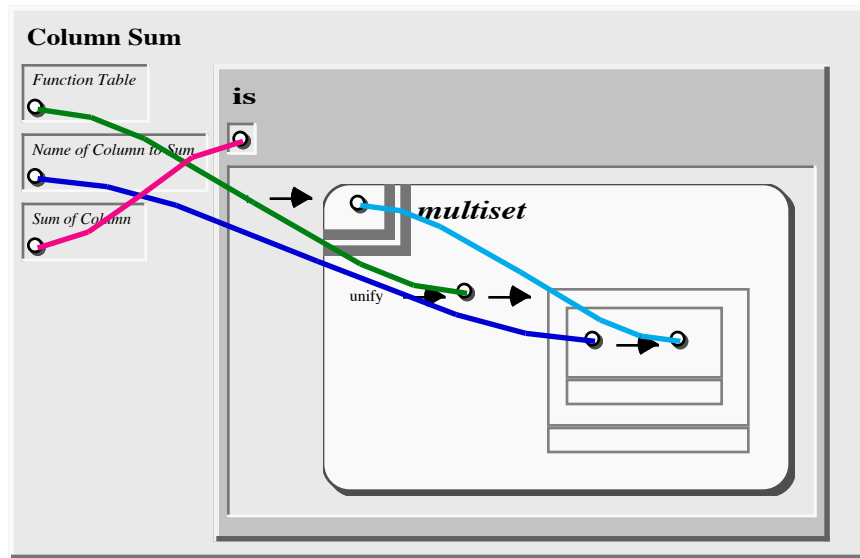


Figure 3.65: Complete clause defining the 'Column Sum'/3 predicate.

set or multiset argument returns the summation of that set. A multiset is represented in SPARCL as a finite function with an integer range. The range of the function gives the “repetition count” for each domain element. A function is represented in SPARCL as a set of ordered pairs where no two ordered pairs have the same first element. The multiset in this 'is'/2 literal is not specified explicitly by showing a particular multiset, but is instead specified using an *intensional multiset* term.

An intensional set is a term that specifies the set of all terms that have the given property. The type of the intensional set may be either "set" or "multiset". A multiset intensional set (or *intensional multiset*) is the multiset of all terms which have the given property. The representation of an intensional (multi)set has two parts, the template and the body. The body is a set of literals. The template contains a term (which almost always contains at least one variable) that is instantiated once for every solution SPARCL finds for the conjunction of the literals of the body of the intensional set. The “result” of the intensional (multi)set is the (multi)set of all of these instantiations of the template. We are using the intensional multiset to collect together all of the values of the given domain of the given function table. Since we want to sum all of these values, we want to keep the duplicates. Thus, we want the result type of the intensional set to be “multiset.” The representation of the intensional set indicates that it has a multiset result by having the double gray lines around the template (instead of the single gray line) and the presence of the bold italic word *multiset*.

Although the first argument of the clause is a function table, there is no function table representation in this clause. There is instead a partitioned set term (inside the second argument of the literal of the clause) that is used to access each of the values of interest in this first argument. A brief review of function tables will help explain this situation. A function table is a table with column headings that represents a set of same-finite-domain functions. The column headings are the values of this common finite domain. Each row of the function table represents one of the finite-domain functions. The value in a column of a row is the range value of that row's corresponding function for the domain value in the header of that column. In this clause we need to deal with a part of the table, so we use the partitioned set representation directly, instead of the function table representation. Thus, the absence of the function table representation from the clause defining 'Column Sum'/3 is an example of this limitation of the table representations: there is no way to work with a table of indefinite size using the table representation, we must use the partitioned set representation instead. Thus, predicates that work with tables of indefinite size (e.g. the predicate works with tables of one row and one column, or two rows and one column, or one row and two columns, etc.) must use the partitioned set representation. This limitation could be removed with appropriate modification of the table representation syntax and semantics.

The structure of the intensional set term has two parts, the template and the body. The template in this case is a single variable. This variable (by suitable coreference) identifies the value of a row of the given function table in the column with the given "name". Collecting all possible such values gives the desired multiset.

The body contains a single literal which refers to the built-in 'unify'/2 predicate. This literal is true for any unification between its first argument, which corefers with the given function table, and its second argument. The construction of the second argument is such that it unifies in as many different ways with the first argument as there are rows in the table (or functions in the set). One might expect that the variable of the function table argument of the clause would be made to corefer with the function table construct of the intensional multiset, instead of indirectly connecting them through a 'unify'/2 literal. The coreference approach does not have the desired result since it is implemented (in the program transformation that creates the internal form) by unifying all of the coreferring terms in the outermost scope. In this case, the outermost scope is that of the clause. This would effectively move the nested set term out

of the intensional multiset, allowing it to be bound only once per evaluation of the clause, instead of potentially many times (to “create” the multiset).

The second argument of the ‘unify’/2 literal is a partitioned set of two parts, where the upper part contains another partitioned set and the lower part is hollow. This “outer” partitioned set unifies with the given table (a set) by partitioning that set into two subsets, one a set of one element of the given set and the other a set of the other elements of the given set. The “inner” partitioned set unifies with the element of the singleton subset. This element is a function of the given table, thus the element is a set. The unification of the inner partitioned set divides this function into two subsets, a set of one element of the function set and a set of the rest of the function set. Finally, the ordered pair in the upper part of the inner partitioned set unifies with the element of the singleton subset of the function set. There are many possible unifications of the inner partitioned set, one for each pair of the function. However, since the first element of the ordered pair of the inner partitioned set corefers with the given domain name, only one such unification succeeds: the one that selects the ordered pair with the first element matching the given domain name. Thus, for each different way in which the outer partitioned set is unified with the given table, there is only one way in which the inner partitioned set and its ordered pair can be unified. The outer partitioned set can be unified with the table to select each of the rows of the table. So, the range value of the ordered pair of the inner partitioned set can be unified once to each row’s value in the given column. Since the template term is a variable that corefers with the range value of the ordered pair of the inner partitioned set, this gives the desired multiset of values.

The ‘Column Sum Query’/1 predicate. The single clause of the ‘Column Sum Query’/3 predicate shown in Figure 3. 66 is used to query the ‘Column Sum’/3 predicate. The result of this query is shown in Figure 3. 67. The term in the argument of the query clause gives shape to the result of the query. The query clause provides a table of data and the name of a column to be summed in that table.

The result of the query (in the argument of the clause) is a table of two rows: one row shows the test data and the other row shows the name of the column being summed and the sum for that column. We ensure that the “data” row of the result table is displayed before (above) the “total” row by making the result table an ordered table—one which is an N-tuple of rows instead of a set of rows. The first column of

this table is used to label the parts of the answer.

The first row of the result table holds the test table. The label in the first column is “data” and the variable in the second column corefers with the test data table in the first argument of the ‘Column Sum’/3 literal. The calculation from the ‘Column Sum’/3 predicate is shown in the second

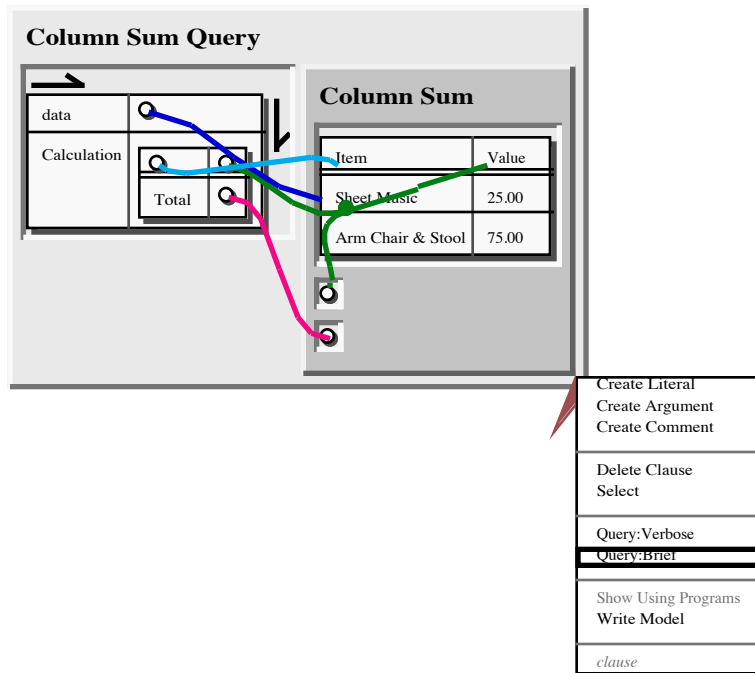


Figure 3.66: Clause defining ‘Column Sum Query’/1 predicate, with interaction to query it.

row of the result table. The label of this row (in its first column) is “Calculation”. The second column “formats” the sum using a function term table with the column titles from the input data. This function table has one row with the first column being the constant “Total” and the second column being the calculated total.

The test data table is in the first argument of the ‘Column Sum’/3 literal. It has two columns titled “Item” and “Value”. There are two rows of values: item “Arm Chair & Stool” with value “75.00”, and item “Sheet Music” with value “25.00”. The other two arguments of the ‘Column Sum’/3 literal contain variables.

Various ground terms are made to corefer with variables in this clause: the entire test data table is linked to a variable in the result table; the “Item” constant heading the first column of the test data table is linked to a variable in the header of the function table in the result table; and, the “Value” constant heading the second column of the test data table is linked to two other variables, one in the header of the function table in the result table and the other in the second argument of the ‘Column Sum’/3 literal (giving the name of the column to be summed). This technique avoids duplicating ground terms within a clause, making the relationship among various parts of the clause more immediately clear than if the ground terms were duplicated. Also, this

eases modification of these ground terms since to change a ground term only requires modifying the one instance of it, instead of modifying multiple copies. To achieve these same advantages in languages where variables corefer by having the same name generally involves

Column Sum Query →

data	Item	Value
	Arm Chair & Stool	75.00
	Sheet Music	25.00
Calculation	Item	Value
	Total	100

↓

Figure 3. 67: Result of the query of the clause for 'Column Sum Query'/1.

some additional complications. Minimally, one must add a statement associating the variable with the ground term. Further, if the language is procedural instead of declarative, then one must be sure to add this additional statement before any of the “uses” of the variable name. If one modifies such a procedural program by adding another use of this common variable, then one must check to be sure that the new use occurs after the “definition” of the variable’s value.

The result of evaluating the query of Figure 3. 67 shows the sum of the items in the test data is 100. The various items are filled in to show the “input” data.

Discussion

In this chapter we have presented a formal syntax of SPARCL, the various aspects of SPARCL in terms of how those aspects relate to the principles in the hypothesis of chapter 1 (“Introduction”), and an introduction to programming in SPARCL. In the formal syntax we presented a diagrammatic grammar that describes the two-dimensional representation of SPARCL. In discussing the design elements of SPARCL, we have shown how many of the elements of the design of the representation, semantics, and development environment spring from these basic ideas. Finally, in the introduction to programming in SPARCL we presented several programs of increasing complexity and explained their design and use.

Formal syntax. The grammar formalism we used to present the syntax of SPARCL is a simplified version of the Hyperedge Replacement Grammar (HRG). This grammar provides productions for the two aspects of SPARCL: programmatic elements and data

elements. It does *not* represent coreference links. Except for the coreference links, all of the concrete graphical elements of the two-dimensional representation of SPARCL are present in the productions of the grammar. The programmatic elements of SPARCL include: clauses, literals, names, and arguments. The data elements are the *linkable* elements: terms and parts (of partitioned sets). The linkable elements are those elements of the syntax that can be linked to by a coreference link. The terms are: variable, ur, empty set, partitioned set, N-tuple, table, and intensional set. The last four of these terms are those that can contain other terms.

Design elements. Section 1 on design elements based on visual programming and logic programming presents several aspects of SPARCL. These elements are those that take advantage of the diagrammatic possibilities of a visual representation to represent the various relationships among the logic programming concepts. These diagrammatic possibilities include: spatial containment, which is used in the representation of relationships such as: name of a predicate a clause for that predicate, a clause's body's literals in that clause, and a partitioned set's part's elements in that part; adjoinment, which is used for the ordering of arguments of a clause or literal and the ordering of the elements of an N-tuple; connecting lines, which are used to represent coreference hyperedges; and similar appearances, which are used to indicate term types.

Using explicit linking to express coreference instead of using naming is taking advantage of the diagrammatic possibilities of a *visual* representation in representing the *logical* concept of coreference (implemented as unification). Variable naming is more of a linguistic representation technique (although it could be considered a “similar appearance” diagrammatic technique). This design choice allowed us to separate the ideas of variables and coreference that are so often combined (conflated) in programming languages. Separating these ideas allows us to specify arbitrary terms as coreferring, not just variables. This in turn simplifies some programs. As we note in chapter 9 (“Usability Testing”), using links instead of variable naming for coreference was one of the things that the participants of the usability study specifically commented on as being an attractive aspect of SPARCL and an improvement on other languages with which they were familiar (such as PROLOG and LISP).

The design of the delay specifications was based on a “minimal effort” approach: the representation is sufficient to allow delays to be specified and requires no addi-

tional development work in the editing system of SPARCL since it represents delay specifications by clauses. However we discovered in our later analysis of SPARCL programs that the delay specifications are a major contributor to their size. Thus it now appears that a greater effort on representing delay specifications would be rewarded by smaller and more comprehensible programs.

In section 2, “Visual & Sets Design Elements”, we show how the variety of representations of sets made possible by the nonlinear/diagrammatic nature of visual programming allows us to provide more compact representations that are at the same time more informative. This is an area that we have only begun to exploit. We provide special representations of N-tuples and various forms of term tables. These representations are directly editable in SPARCL. One can see from the examples (Figure 3. 21, Figure 3. 22, Figure 3. 23, and Figure 3. 24) that these representations are much more compact than the corresponding partitioned set representations. As we mentioned in the “Specialized set representation for N-tuples” section, we would like to provide specialized representations for lists (a special kind of N-tuple) and matrices (lists of same-length lists). Matrices can also be considered as a special kind of ordered table of N-tuples, with “empty_list” as the zeroth row and a common prefix (first element) of “empty_list” for all of the other rows.

Graphs are another style of representing sets that we have not explored yet. A mapping-type set where the domain and range have a nonempty intersection can be represented as a directed graph where the nodes are the domain and range values and there is an edge between two nodes if and only if those two node values are in an ordered pair of the set. A special representation of the directed graph could be used when it was found to acyclic, or a tree. A set of properly constructed ordered triples could be interpreted as a labeled directed graph, where two of the elements of the 3-tuple (say the 2nd and 3rd) are the “source” and “target” node values and the 1st element of the 3-tuple is the label of the edge. If there are a “small” number of different labels in the graph, then a legend could be provided as part of the representation of the graph where the labels are mapped onto different appearances of the edges (e.g., different colors or different thicknesses). All of these graph representations would be directly editable, just as tables are now. Also, as we mentioned in chapter 1 (“Introduction”), we are interested in providing a programmer-adaptable representation in the manner of the programmer-specified operator-precedence grammar of Edinburgh syntax PROLOG. These programmer-defined representations would be built on the

mechanisms used to implement the partitioned set, N-tuple, table, matrix, and graph representations.

Intensional set terms were also presented in section 2. The intensional set term is more compact than the semantically equivalent construction using ‘setof’/3 literals, and it allows one to use fewer coreference links, as is the case in the example in Figure 3.25. Since they are terms, they can be nested (literals are not so conveniently nested). Also, using intensional set terms increases the number of clauses that are “facts”, thus increasing the opportunities to use the fact table representation. This further decreases the size of the overall representation of a predicate.

In section 3 we presented the design elements derived from both logic programming and set-based programming. The most pervasive aspect of SPARCL derived from these two sources is the partitioned set and its implied constraints. The unification algorithm and inference mechanism are specialized in SPARCL to handle them. The need for ordered collections of terms, N-tuples, in the set-based system lead us to handle N-tuples as special organizations of sets.

Programs are sets of clauses in SPARCL and therefore unordered; they are usually ordered in other logic programming languages. Similarly, the body of a clause is a set of literals and is also therefore unordered. We have provided the “delay” mechanism to address the selection of which literal in a clause to interpret and two built-in meta-predicates, `ordered_disjunction/2` and `if/3`, to address ordering the selection of which clause to interpret. The representation of the “delay” mechanism was simply done and needs to be reconsidered to reduce program size.

In the “Visual Programming Design Elements” section we present a categorization of possible editing environments:

- (1) program representation drawn “free hand” and resulting unstructured picture parsed by system;
- (2) program representation drawn using system-provided elements and resulting structured picture parsed by system;
- (3) program representation drawn using system-provided semantically-specific elements with manual layout, no parsing required; or,
- (4) program representation indicated by specifying semantic modifications with automatic layout.

SPARCL uses the fourth approach, the semantic modification/automated layout environment. This approach fits well with diagrammatic visual representation,

declarative semantics of logic programming, and unordered relationship among elements of sets. It is particularly valuable for the three-dimensional representation, since it relieves the programmer of a very complex layout problem.

In the “Sets With Partitioning Design Elements” section we showed the multiple uses of partitioned sets. This single mechanism provides convenient ways to express union, intersection, and difference operations on sets. We presented a detailed examination of the ‘Union’/3 predicate to show how these set operations relate to partitioned sets. Partitioned sets also provide a mechanism to specify the logical structure of a set.

In the “Logic Programming Design Elements” section we explained the general semantics of SPARCL as represented by the SPARCL interpreter. SPARCL is a Horn clause logic programming system that is basically an SDNF resolution theorem prover with partitioned set unification, partitioning constraints, and delayed literals. Negation is provided as “failure” under the Closed World Assumption.

In the “Visual, Logic, and Set Design Elements” section we discussed the Input/Output system of SPARCL. This is a difficult area which has not been thoroughly addressed yet. The basic concepts of the current approach are the persistent term and the term set program. These deal with output in a consistent logical fashion, but don’t provide all of the services a programmer needs. To fill some of the output “gaps”, we also implemented two built-in predicates that write readable linear representations of terms to the ‘*Output*’ window. We don’t yet have any solution to interactive input during the interpretation of a query.

Programming in SPARCL. Our presentation of programming in SPARCL discusses the implementation of several predicates and various ways in which these predicates can be queried. The predicates defined are: ‘Parent’/2, ‘Female’/1, ‘Male’/1, ‘Sister (sort of)’/2, ‘Different’/2, ‘Sister’/2, ‘Offspring’/2, ‘Mother’/2, ‘Grandmother’/2, ‘Predecessor’/2, ‘Vertical’/1, ‘Horizontal’/1, and ‘Column Sum’/3. We introduced the concepts of facts, rules, and questions. Then, we examined in detail how SPARCL solved a particular query of the ‘Predecessor’/2 predicate. The second and third parts of this programming introduction presented handling of complex data: first N-tuples, then partitioned sets, tables, and intensional sets.

Assessment. This chapter shows that SPARCL is a syntactically well-formed language

that successfully embodies programming language design aspects of the hypothesis of this thesis.

- Fukunaga et al. 1993a “Object -Oriented Development of a Data Flow Visual Language System”
by Alex S. Fukunaga, Takayuke D. Kimura, and Wolfgang Pree. Pages 134-141 in:
Proceedings 1993 IEEE Symposium on Visual Languages, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- Mendelson 1964 *Introduction to Mathematical Logic* by Elliott Mendelson. New York, NY: D. Van Nostrand Company. 1964.
- Shin 1994 *The Logical Status of Diagrams* by Sun-Joo Shin. 197 pages. Cambridge, England:Cambridge University Press. 1994.
- Viehstaedt 1995 “A Generator for Diagram Editors” by Gerhard Viehstaedt. Doktor-Ingenieur Dissertation at Universität Erlangen-Nürnberg. 1995.

Chapter 4

Partitioned Set Unification

This set unification algorithm was developed as part of the SPARCL visual logic programming language. SPARCL makes extensive use of finite sets, particularly partitioned finite sets. Thus, partitioned finite set unification is a core element of the semantics of SPARCL. (All of the sets discussed in this paper are finite, so the “finite” is implicit in the rest of the paper.) Languages which work with sets generally provide support for at least three basic operations on sets: union, intersection, and difference. In SPARCL, these operations are expressed using the single mechanism of partitioned set unification. In this chapter we present a formalization of the partitioned set unification algorithm with examples of its operation. We prove the soundness and completeness of the “atomization” portion of this algorithm, but the full proof of these properties for the entire unification algorithm remains to be done.

The related work is all in the area of set (or subset) unification and matching. This is the first work of which we know where partitioned sets are directly unified (or matched). There is limited form of partitioned set matching done in SEL. SEL as presented in [Jayaraman&Nair 1988] is one of the few declarative programming languages in which sets are “...‘first class’ objects, i.e., not simulated by lists.” SEL is a functional language based on term-rewriting. It uses associative-commutative matching of sets, instead of associative-commutative-idempotent matching or unification. The set representation used in SEL is $\{X|T\}$ which is equivalent to $\{X\} \cup T, \text{ if } X \notin T$. This can be represented by a partitioned set: $\text{ptn}(\{\{X\}, T\})$. Partitioned set unification is associative-commutative, and not idempotent. This is similar to the (partitioned) set matching of SEL.

Notation. Here we present a summary of the notations and their meanings as they are used in this chapter.

A *unification* “term” is represented by $S \doteq T$. This can be read “S and T unify” or “S unifies with T”.

Generally, lowercase letters are used for *constants* and uppercase for *variables* of the object language.

A *set* is represented using braces: $\{a, b\}$ is the set of two constants, a and b .

A *multiset* is represented as a set with a *multi* subscript: $\{A, B, C\}_{multi}$ is a multiset of three elements, each with one occurrence. Multiple occurrences are indicated by a prefix superscript: $\{^2A, B, ^5C\}_{multi}$ is a multiset of three elements of 2, 1, and 5 occurrences.

A *multiset union* is indicated by the operator \bigcup_{multi} . For instance: $A \bigcup_{multi} B$. The multiset union is a union of multisets and keeps track of the occurrence counts.

U means the union across the elements of S (which must be sets). For instance: $U\{A, B, C\} = A \cup B \cup C$.

$\{x | p(x)\}$ is the (intensional) set of all x such that predicate $p(x)$ is true.

A *partitioned set* is represented by a “term” with ‘ptn’ as its functor and one argument, which is a set of “parts” of the partitioned set: ‘ $\text{ptn}(\{A, B, C\})$ ’ is a partitioned set with three variable parts.

A *partitioning constraint* is represented by a term with ‘ptncon’ as its functor and one argument, which is a *multiset* of parts: ‘ $\text{ptncon}(\{A, B, C\}_{multi})$ ’ is the partitioning constraint implied by the partitioned set ‘ $\text{ptn}(\{A, B, C\})$ ’. The partitioning constraint uses a multiset instead of a set so that if two variables that represent different parts are bound to the same term, then the resulting multiset of parts has an element with a count greater than 1. If it were a set, then the duplicate element would simply disappear and the partition constraint check would have no way to tell that two parts had a nonempty intersection (because they had become equal). Since it is a multiset, the constraint check requires that all elements of the multiset have an occurrence count of (no more than) 1.

‘ $S = \text{ptn}(T)$ ’ is a shorthand notation for ‘ $S = UT \wedge \forall p, q \in T (p \cap q = \emptyset)$ ’

An *atomization formula* has the form: $S \Rightarrow \text{ptn}(T); \gamma$, where S is the set to be atomized, T is a set of variables and singleton sets, and γ is a substitution that has been applied to S and T .

$\text{var}(x)$ is a predicate that is true when x is a variable of the object language.

A *binding* is a term of the form: X / t where X is a variable and t is any term of the object language. This binds X to t .

A *substitution* is a set of bindings.

σ / t is the *substitution application* of σ to term t . The result of the application has every binding in σ applied to the variables in t . If the variable to which the bind-

ing is being applied matches the left-hand-side of the binding, then the variable is replaced by the right-hand-side of the binding. Otherwise, the variable is unchanged.

$\sigma \circ \tau$ is the composition of two substitutions. If one treats a substitution like a function, then this is normal function composition. The substitution equivalent to the composition is determined by applying σ to the right-hand-side of the elements of τ , and then union this modified τ with σ .

A *unification formula* has the form: $U; \Gamma; V; \Sigma$ where U is a set of unifications, Γ is a set of partitioning constraints, V is a set of variables, and Σ is a substitution.

We represent parts of the unification algorithm using an inference rule format:

$$\frac{Ante}{Cons} \text{ where } Cond$$

Ante is the antecedent of the rule, *Cons* is the consequent, and *Cond* is a condition (frequently a junction of several propositions) that must hold for the rule to be used. We consider that *Ante* is a pattern that is mapped onto a selected term, so that the use of these inference rules is something like the application of axiom schema. *Cons* is general a set of formulas.

For the atomization rules, the formulas of *Ante* and *Cons* are atomization formulas.

For the unification rules, the formulas of *Ante* and *Cons* are unification formulas.

Overview of the algorithm. The set unification algorithm is built on the atomized partitioned set unification algorithm. There are a series of transformations which turn sets into atomized partitioned sets, then one proceeds with the atomized partitioned set unification algorithm. This algorithm converts a set of general unification equations into variable-binding unification equations (an equation with a variable on the left-hand side and any term on the right-hand side). For example, the set of unification equations $\{\{a, b\} \doteq \{b, X\}\}$ is converted to $\{X \doteq a\}$, where a and b are constants and X is a variable. Note that in this document ‘ \doteq ’ is used as the “unifies” operator, which is distinct from ‘ $=$ ’ (equality).

A partitioned set is represented by ‘ $\text{ptn}(X)$ ’ where X is a set of “parts” of the partitioning. These parts are either sets or variables. The partitioned set implies the constraint that all of its parts are pairwise disjoint. Further, a partitioned set is equivalent to a “general” set which is the union of the parts of that partitioned set:

$$\text{ptn}(\{X_1, \dots, X_n\}) = X_1 \cup X_2 \cup \dots \cup X_n$$

The SPARCL program for ‘Union’/3 and a query clause are shown in Figure 4. 1. The ‘Union’/3 predicate is discussed in section 5 of chapter 3 and in appendix 1. This example uses partitioned sets to specify a “union” relation. The example query clause leads to the following unification equations:

$$\{\{a,b\} \doteq \text{ptn}(\{X,Y\}), \{b,c\} \doteq \text{ptn}(\{Y,Z\}), R \doteq \text{ptn}(\{X,Y,Z\})\}$$

The solution for this example is:

$$\{X \doteq \{a\}, Y \doteq \{b\}, Z \doteq \{c\}, R \doteq \{a,b,c\}\}$$

The variable R is unified with the union of $\{a,b\}$ and $\{b,c\}$. This is the only solution because of the disjointness and union constraints implied by the partitioned sets.

A basic aspect of partitioned set unification is that one must unify arbitrary subsets represented by variables. For instance, consider the unification:

$$\{\text{ptn}(\{A,B,C\}) \doteq \text{ptn}(\{X,Y\})\}$$

There are many different ways in which the sets represented by the variables A , B , C , X , and Y may be instantiated to satisfy this problem, including: A is part of X and part of Y , B is the other part of X , and C is the other part of Y ; A , B , and C each contain parts of both X and Y ; A is the union of X and Y , B and C are empty; or, X is the union of A , B , and C and Y is empty. Many other solutions are possible. Our approach to partitioned set unification allows for all possible solutions without committing to any of them. By not committing to any subset of the possible solutions when unifying two partitioned sets, it is never necessary to backtrack and redo this part of the unification process. We do commit to particular possible solutions with

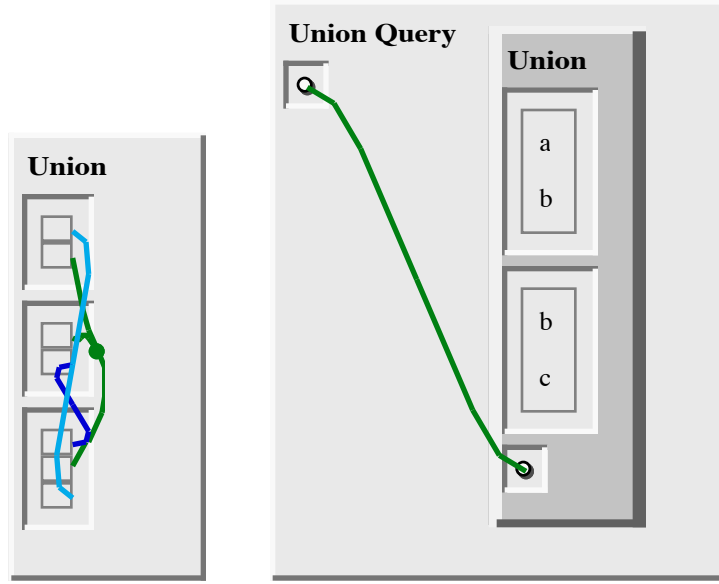


Figure 4. 1: Clause defining the ‘Union’/3 predicate and a clause defining the ‘Union Query’/1 predicate.

regard to the nonvariable parts of a partitioned set; this choice can be backtracked and redone if necessary.

This is achieved in the unification algorithm by the atomization then the “hollowing” of variable parts of partitioned sets. The variable parts introduced in the hollowing process provide the appropriate flexibility to account for all of the possible subset unifications.

The fundamental partitioned set unification algorithm for unifying partitioned sets P and Q consists of several major steps:

Remove Identical Elements.

Introduce Hollow Partitioned Sets.

Connect Hollow Partitioned Set Parts

Unify Unconnected Hollow Partitioned Set Parts

Unify Processed Partitioned Sets

Additional steps used to “clean up” and validate the unification equations:

Delete Empty Partition Element

Occur Check

Instantiate

Commute

These steps are specified using an inference-rule-like format.¹

Converting sets to atomized partitioned sets.

Before the atomized partitioned set unification algorithm can be applied to a set of unification equations, these equations must be “atomized”. There may be several atomizations of the set of unification equations. Each such atomization must be analyzed by the atomized partitioned set unification algorithm, which may produce many solutions for each atomization of the original set of unification equations. Thus, this algorithm may produce many distinct “maximal general unifiers” (MGUs).²

1. This approach to specifying unification is based on the unification-with-residuation specification given in [Hanus 1995], p. 219.

2. Normally MGU is an abbreviation for “*most* general unifier”. But since there may be several maximal general unifiers there usually isn’t a single most general unifier.

What atomization produces. We define an *atomized partitioned set* as a partitioned set where each of its parts is either a singleton set or is a variable. We use the atomized form of partitioned sets since it is simpler to work with this structure than with “general” partitioned sets. We will sometimes speak of atomizing a partitioned set instead atomizing a set; the relationship between these terms is simple. A set ‘S’ is logically equivalent to the partitioned set ‘ $\text{ptn}(\{S\})$ ’, i.e. the partitioned set of one part where that one part is the set ‘S’.

Atomizing partitioned sets does introduce a complexity to the unification process, however. There may be several different atomizations of a partitioned set, depending on how the variables in that set are bound. Consider the partitioned set ‘ $\text{ptn}(\{\{X,a\}\})$ ’, where ‘X’ is a variable and ‘a’ is a constant. This partitioned set has one part, ‘ $\{X,a\}$ ’. It is not atomized since ‘ $\{X,a\}$ ’ has a cardinality of 2. There are two interesting possibilities for the bindings of ‘X’, either it is bound to ‘a’ or it is bound to something else. In the first case the partitioned set collapses to ‘ $\text{ptn}(\{a\})$ ’. In the second case, the partitioned set becomes ‘ $\text{ptn}(\{\{X\}, \{a\}\})$ ’. This second case exactly expresses the idea that “‘X’ is bound to something else”, since the disjointness constraint implied by the partitioned set notation requires that ‘ $\{X\} \cap \{a\} = \emptyset$ ’, and the only way for this to be true is if ‘ $X \neq a$ ’.

We now address the issue of exactly what the atomization produces and how the result of the atomization relates to the partitioned set being atomized. For this discussion we need the ideas of “substitution” and “bound version”. A *substitution* is a mapping of variables to terms. It is important to realize that the term to which a variable is mapped may be another variable. A *bound version* of a term is the result of applying a substitution to that term. Clearly, applying a substitution that has variables in its range to a term may result in a bound version that may still contain variables. The atomization process produces a set of atomizations that satisfies three properties: all of the resulting terms are atomized partitioned sets; for any substitution, the atomization of the bound version of the original set is equivalent to the bound version of one of these atomizations of the original set; and, for each atomized version ‘T’ of the original, there is a substitution such that the bound version of ‘T’ equals the bound version of the original set. We can now summarize the simple set atomization example given above. Since a particular atomized version of a set has an accompanying (possibly empty) substitution, we represent the atomized partitioned set in an ordered pair with

the associated substitution as the second element. Thus, the set of possible atomized partitioned set versions of ‘ $\text{ptn}(\{\{X, a\}\})$ ’ is:

$$\{\langle \text{ptn}(\{\{a\}\}), \{X / a\} \rangle, \langle \text{ptn}(\{\{X\}, \{a\}\}), \emptyset \rangle\}.$$

To completely express the results of atomization, we need to distinguish between atomization of a set and atomization of a partitioned set. The atomization of a partitioned set is constructed using the atomization of sets. Let ‘ $\text{ptn}(P)$ ’ be the partitioned set to be atomized:

$$\begin{aligned} & \text{partitioned_set_atomization}(P, C) \\ &= \bigcup \left\{ AP \left| \begin{array}{l} Q \in P \\ \wedge (\text{var}(Q) \rightarrow AP = \langle Q, \emptyset \rangle) \\ \wedge (\neg \text{var}(Q) \rightarrow AP = \text{atomization}(Q, \{\text{ptn}(P)\} \cup C)) \end{array} \right. \right\} \end{aligned}$$

To atomize a partitioned set (subject to some given partitioning constraints ‘ C ’), we atomize each of that partitioned set’s parts (subject to the partitioning constraint implied by the original partitioned set plus the given partitioning constraints ‘ C ’) and union the results. If a part is a variable, then it is already atomized. If it is not a variable, then it must be a set and we atomize it using the ‘ $\text{atomization}(S, C)$ ’ function described next. The partitioning constraints may be non-empty due to previous unifications.

We now characterize the results of atomizing a set, given some (possibly empty) partitioning constraints. Let ‘ S ’ be a set, ‘ C ’ be a set of partitioning constraints, and ‘ AS ’ be the result of atomizing ‘ S ’:

$$\begin{aligned} AS &= \text{atomization}(S, C) \\ &\rightarrow \left(\begin{array}{l} \forall R, \gamma \left(\begin{array}{l} \langle \text{ptn}(R), \gamma \rangle \in AS \\ \rightarrow \left(\forall p (p \in R \rightarrow (\text{var}(p) \vee \text{cardinality}(p) = 1)) \right) \end{array} \right) \\ \wedge \forall R, \gamma \left(\begin{array}{l} \langle \text{ptn}(R), \gamma \rangle \in AS \\ \rightarrow (\text{valid_partitioning}(\text{ptn}(R)) \wedge \cup R = \gamma / S) \end{array} \right) \\ \wedge \forall \sigma \left(\begin{array}{l} \text{valid_constraints}(\sigma / C) \\ \rightarrow \exists R, \gamma \left(\begin{array}{l} \langle \text{ptn}(R), \gamma \rangle \in AS \\ \wedge \text{valid_partitioning}(\sigma / \text{ptn}(R)) \wedge \cup(\sigma / R) = (\sigma \circ \gamma) / S \end{array} \right) \end{array} \right) \end{array} \right) \end{aligned}$$

The first proposition in this property of atomization states that all of the elements of the atomization are *atomized* partitioned sets, where each part of an atomized partitioned set must be either a variable or a singleton set. The second proposition states that for every pair of partitioned set and substitution in the atomization that partitioned set is equal to the bound version of the original set (i.e. the union of the parts of that partitioned set is equal to the original set). The third proposition states that for any substitution ‘ σ ’ that retains the validity of the given constraints ‘ C ’, there is a pair of a partitioned set and substitution ‘ γ ’ in the atomization such that the version of that partitioned set bound with ‘ s ’ is equal to the version of the original set bound using the composition of the substitutions ‘ σ ’ and ‘ γ ’. The second proposition is a *soundness* constraint: the atomization process only creates atomized partitioned sets that validly derive from the original partitioned set. The third proposition is a *completeness* constraint: Any substitution that validly binds the original set, creating some term T , can be used to validly bind some atomization of the original set to create the same term T .

In the example atomization above, the two resulting partitioned sets in :

$$\left\{ \left\langle ptn(\{\{a\}\}), \{X/a\} \right\rangle, \left\langle ptn(\{\{X\}, \{a\}\}), \emptyset \right\rangle \right\}$$

have the desired properties. The first proposition requires atomized parts, and they are both atomized (they contain only singleton sets, in this case). The second proposition requires that for each atomized partitioned set, the associated substitution makes the union of its parts equal to the union of the parts of the bound version of the original partitioned set. The substitution ‘ $\{X/a\}$ ’ makes ‘ $\{a\}$ ’ equal to ‘ $\{X,a\}$ ’, by converting ‘ $\{X,a\}$ ’ to ‘ $\{a,a\}$ ’, which is identical to ‘ $\{a\}$ ’. The union of ‘ $\{\{X\}, \{a\}\}$ ’ is equal to ‘ $\{X,a\}$ ’, which is the same as the original partitioned set’s only part. Thus, the identity substitution (‘ $\{\}$ ’) “makes” these two equal by doing nothing. The third proposition requires that for any substitution ‘ s ’ for which the bound version of ‘ S ’ is a valid partitioning (i.e. the parts of ‘ S ’ remain pairwise disjoint), there must be some atomized partitioned set ‘ R ’ for which the union of its bound version’s parts is equal to the union of the parts of the bound version of ‘ S ’. This brings us back to the “two interesting possibilities” discussed in the initial presentation of this example above: either ‘ s ’ binds ‘ X ’ to ‘ a ’, or it binds ‘ X ’ to something else. In the first case (‘ $s=\{X/a\}$ ’), the bound version of ‘ S ’ becomes ‘ $ptn(\{\{a\}\})$ ’, which is identically one of the atomized partitioned sets. In the second case where ‘ s ’ binds ‘ X ’ to something

other than ‘a’ ($s=\{X/Y\}$), where ‘Y’ is any term such that $Y \neq a$, ‘S’ is bound to $\text{ptn}(\{\{Y,a\}\})$. The other atomized partitioned set handles this case. It is bound to $\text{ptn}(\{\{Y\}, \{a\}\})$. The union of the parts of the bound ‘S’, $\{\{Y,a\}\}$, must be the same as the union of the parts of this bound atomized partitioned set, $\{\{Y\},\{a\}\}$. Also, the disjointness constraint must hold for the bound version of the atomized partitioned set ($\{Y\} \cap \{a\} = \emptyset$) since we are assuming $Y \neq a$.

The atomization algorithm. The atomization algorithm finds a set of atomizations of a given set as defined above. There can be many possible sets of atomizations that satisfy the above atomization property. These sets differ in the extent to which they contain unbound variables.³ We conjecture that there is a minimal set of atomizations of a given set, and that our atomization algorithm finds that minimal set. Note that since set atomization is a step in the unification process and atomization relies on unification, the atomization algorithm may recurse.

A set S is transformed into an atomized partitioned set by repeated application of the following inference rules to the formula $S \Rightarrow \text{ptn}(\emptyset); \emptyset$. The result of the application of these inference rules is a set of formulas where the left hand-side of each formula is an empty set and the partitioned set on the right-hand of the formula is a transformation of the initial set S . Associated with each such transformation is a (possibly empty) set of unification equations.

Rule 1 and Rule 2 are the inference rules at the heart of the atomization algorithm. They are used to manipulate sets of formulas of the form $S \Rightarrow \text{ptn}(T); \gamma$, where S is the unatomized portion of the original set, $\text{ptn}(T)$ is the atomized partitioned portion of the original set, and γ is the set of variable bindings that have been applied to S and T . The atomization process creates a sequence of sets of these formulas. Given a sequence of formula sets, this sequence is extended by transforming the final formula set of the sequence (the “current” formula set) by applying Rule 1 or Rule 2 to a formula in that set and creating the next set is the current set minus the selected formula plus the consequent formulas from the rule application. Eventually this process converges on a set of formulas where all of the left-hand sides are empty, i.e. where the set represented by the formula is entirely represented by a partitioned set. This final formula set contains all possible atomized partitioned set representa-

3. If the original set contains unbound variables then at least one of the elements of the set of atomizations must also contain unbound variables for that set to be finite.

tions of the original set.

We define ‘ $\text{atomize_step}(\phi)$ ’ as the function that given a formula (‘ ϕ ’) returns the set of formulas produced by transforming ‘ ϕ ’ according to whichever rule applies to ‘ ϕ ’. Let ‘ Φ_i ’ be a formula set in a sequence, where ‘ Φ_{i+1} ’ is the next formula set in that sequence. Then the equation defining a step is:

$$\Phi_{i+1} = \bigcup \{ \text{atomize_step}(\phi) \mid \phi \in \Phi_i \}.$$

The $\text{atomize_step}(\phi)$ function returns a singleton set of the given formula ϕ when that formula’s left-hand side is empty, otherwise it creates a set of formulas each of which has as its left-hand side a set that is a proper subset of the left-hand side of ϕ . Thus, the sequence must converge at some step k such that $\Phi_k = \Phi_j, \forall j \geq k$. This Φ_k is the final formula set of the sequence. The initial formula set for the atomization of a set A is ‘ $\{A \Rightarrow \text{ptn}(\emptyset); \emptyset\}$ ’. The final formula set in an “atomization sequence” is of the form ‘ $\{(\emptyset \Rightarrow \text{ptn}(B_1); \gamma_1), \mathbf{L}, (\emptyset \Rightarrow \text{ptn}(B_n); \gamma_n)\}$ ’, where γ_i is a set of variable bindings such that:

$$\begin{aligned} \gamma_i / A &= \bigcup B_i \\ \wedge \forall p, q \in B_i & \left((\neg \text{var}(p) \wedge \neg \text{var}(q)) \rightarrow p \cap q = \emptyset \right). \\ \wedge \forall p \in B_i & \left(\text{var}(p) \vee \text{cardinality}(p) = 1 \right) \end{aligned}$$

The first of the three propositions of this property of the final formula set says that the set resulting from applying γ_i to the original set A is equal to the union of all the elements of the set B_i . The second of these propositions requires that all of the elements of the set B_i that are not variables must be non-intersecting sets. These first two propositions are the expansion of ‘ $\gamma_i / A = \text{ptn}(B_i)$ ’. The third proposition states that all of the elements of B_i are either variables or singleton sets.

Rule 1 and Rule 2 are the two possible cases: either there is a pair of elements X and Y of S such that X and Y unify and the partitioning T remains valid, or there is not such a pair of elements of S .

$$\begin{array}{c}
\frac{\{X, Y\} \cup S \Rightarrow \text{ptn}(T); \gamma}{\left\{ \{Y\} \cup S \Rightarrow \text{ptn}(\{\{X\}\} \cup T); \gamma \right\} \cup R} \\
\text{if } R \neq \emptyset \\
\wedge R = \left\{ P \left| \begin{array}{l} P = [\{Y\} \cup \sigma / S \Rightarrow \text{ptn}(\sigma / T); \gamma \circ \sigma] \\ \wedge \text{unify}(X, Y, \sigma) \wedge \text{valid_partitioning}(\sigma / T) \end{array} \right. \right\} \quad (\text{Rule.1})
\end{array}$$

(where σ is a set of bindings)

Rule 1 produces a set of formulas from a matched formula, where the set of formulas are the results of all possible unifications of a pair of elements, including the “null”unification—specifying that the two elements must not unify.

$$\begin{array}{c}
\frac{S \Rightarrow \text{ptn}(T); \gamma}{\left\{ \emptyset \Rightarrow \text{ptn}(\{\{U\} | U \in S\} \cup T); \gamma \right\}} \\
\text{if } \neg \exists X, Y, \sigma \left(\begin{array}{l} X, Y \in S \wedge \sigma \in \text{unifiers}(X, Y) \\ \wedge \text{valid_partitioning}(\sigma / T) \end{array} \right) \quad (\text{Rule.2})
\end{array}$$

Rule 2 is the case where there are no unifications among elements of S possible (i.e. the elements of S are pairwise nonunifiable), so each element is placed in its own part of the resulting partitioning.

Example atomization. To help explain how the atomization process works, we present the example of the transformation of $S = \{a, B, C\}$, where ‘a’ is not a variable and B and C are variables. This is solved in four steps, with the following result:

$$\text{atomization}\left(\text{ptn}(\{\{a, B, C\}\}), \emptyset\right) = \left\{ \begin{array}{l} \langle \text{ptn}(\{\{a\}, \{B\}, \{C\}\}), \emptyset \rangle \\ \langle \text{ptn}(\{\{a\}, \{C\}\}), \{B/a\} \rangle \\ \langle \text{ptn}(\{\{a\}, \{B\}\}), \{C/a\} \rangle \\ \langle \text{ptn}(\{\{a\}\}), \{B/a, C/a\} \rangle \\ \langle \text{ptn}(\{\{a\}, \{B\}\}), \{C/B\} \rangle \end{array} \right\}$$

To start the atomization, we create the initial formula set:

$$\Phi_1 = \left\{ \left(\{a, B, C\} \Rightarrow \text{ptn}(\emptyset); \emptyset \right) \right\}.$$

To find the next formula set, we must apply the appropriate rule the element of the current formula set. We apply a rule to a formula by finding all possible mappings of the antecedent onto that formula, then calculating the expanded form of the consequent of the rule for each such mapping. Thus, to start the rule application process, we determine which of the two rules applies to the formula in ' Φ_1 '. Since there are elements on the left-hand-side of the arrow in that formula that unify with each other, we apply Rule 1. There are six mappings of the antecedent of Rule 1 to the single formula of this set:

$$\begin{aligned} X = a, Y = B, S = \{C\}, T = \emptyset; \\ X = a, Y = C, S = \{B\}, T = \emptyset; \\ X = B, Y = a, S = \{C\}, T = \emptyset; \\ X = B, Y = C, S = \{a\}, T = \emptyset; \\ X = C, Y = a, S = \{B\}, T = \emptyset; \\ X = C, Y = B, S = \{a\}, T = \emptyset. \end{aligned}$$

The first of these leads to the following expanded consequent:

$$\begin{aligned} & \left\{ \left(\{B\} \cup \{C\} \Rightarrow \text{ptn}(\{\{a\}\} \cup \emptyset); \emptyset \right), \left(\{a\} \cup \{C\} \Rightarrow \text{ptn}(\emptyset); \{B/a\} \right) \right\} \\ &= \left\{ \left(\{B, C\} \Rightarrow \text{ptn}(\{\{a\}\}); \emptyset \right), \left(\{a, C\} \Rightarrow \text{ptn}(\emptyset); \{B/a\} \right) \right\} \end{aligned}$$

The second of the antecedent mappings gives:

$$\begin{aligned} & \left\{ \left(\{C\} \cup \{B\} \Rightarrow \text{ptn}(\{\{a\}\} \cup \emptyset); \emptyset \right), \left(\{a\} \cup \{B\} \Rightarrow \text{ptn}(\emptyset); \{C/a\} \right) \right\} \\ &= \left\{ \left(\{B, C\} \Rightarrow \text{ptn}(\{\{a\}\}); \emptyset \right), \left(\{a, B\} \Rightarrow \text{ptn}(\emptyset); \{C/a\} \right) \right\} \end{aligned}$$

The third gives:

$$\begin{aligned} & \left\{ \left(\{a\} \cup \{C\} \Rightarrow \text{ptn}(\{\{B\}\} \cup \emptyset); \emptyset \right), \left(\{a\} \cup \{C\} \Rightarrow \text{ptn}(\emptyset); \{B/a\} \right) \right\} \\ &= \left\{ \left(\{a, C\} \Rightarrow \text{ptn}(\{\{B\}\}); \emptyset \right), \left(\{a, C\} \Rightarrow \text{ptn}(\emptyset); \{B/a\} \right) \right\} \end{aligned}$$

The fourth gives:

$$\left\{ \left(\{a, C\} \Rightarrow \text{ptn}(\{\{B\}\}); \emptyset \right), \left(\{a, B\} \Rightarrow \text{ptn}(\emptyset); \{C/B\} \right) \right\}$$

The fifth gives:

$$\left\{ \left(\{a, B\} \Rightarrow \text{ptn}(\{\{C\}\}); \emptyset \right), \left(\{a, B\} \Rightarrow \text{ptn}(\emptyset); \{C/a\} \right) \right\}$$

The sixth gives:

$$\left\{ \left(\{a, B\} \Rightarrow \text{ptn}(\{\{C\}\}; \emptyset), \{a, B\} \Rightarrow \text{ptn}(\emptyset); \{C / B\} \right) \right\}$$

Combining all of these expanded consequents and eliminating duplicate formulas gives:

$$\Phi_2 = \left\{ \begin{array}{l} \left(\{B, C\} \Rightarrow \text{ptn}(\{\{a\}\}; \emptyset), \{a, C\} \Rightarrow \text{ptn}(\emptyset); \{B / a\} \right), \\ \left(\{a, B\} \Rightarrow \text{ptn}(\emptyset); \{C / a\} \right), \left(\{a, C\} \Rightarrow \text{ptn}(\{\{B\}\}; \emptyset), \right. \\ \left. \left(\{a, B\} \Rightarrow \text{ptn}(\emptyset); \{C / B\} \right), \left(\{a, B\} \Rightarrow \text{ptn}(\{\{C\}\}; \emptyset) \right) \right\}$$

This is the result of the first atomization step. We repeat the atomization process on this formula set to find the next one. Again, Rule 1 applies to all of the formulas in this set. The mappings of the antecedent for the first formula are:

$$X = B, Y = C, S = \emptyset, T = \{\{a\}\}, \gamma = \emptyset;$$

$$X = C, Y = B, S = \emptyset, T = \{\{a\}\}, \gamma = \emptyset.$$

The first of these leads to the following consequent:

$$\begin{aligned} & \left\{ \left(\{C\} \cup \emptyset \Rightarrow \text{ptn}(\{\{B\}\} \cup \{\{a\}\}; \emptyset), \{B\} \cup \emptyset \Rightarrow \text{ptn}(\{\{a\}\}; \{C / B\}) \right) \right\} \\ & = \left\{ \left(\{C\} \Rightarrow \text{ptn}(\{\{a\}, \{B\}\}; \emptyset), \{B\} \Rightarrow \text{ptn}(\{\{a\}\}; \{C / B\}) \right) \right\} \end{aligned}$$

The second mapping leads to:

$$\begin{aligned} & \left\{ \left(\{B\} \cup \emptyset \Rightarrow \text{ptn}(\{\{C\}\} \cup \{\{a\}\}; \emptyset), \{B\} \cup \emptyset \Rightarrow \text{ptn}(\{\{a\}\}; \{C / B\}) \right) \right\} \\ & = \left\{ \left(\{B\} \Rightarrow \text{ptn}(\{\{a\}, \{C\}\}; \emptyset), \{B\} \Rightarrow \text{ptn}(\{\{a\}\}; \{C / B\}) \right) \right\} \end{aligned}$$

The simplified combination of these results is:

$$\left\{ \left(\{C\} \Rightarrow \text{ptn}(\{\{a\}, \{B\}\}); \emptyset \right), \left(\{B\} \Rightarrow \text{ptn}(\{\{a\}\}); \{C/B\} \right), \right. \\ \left. \left(\{B\} \Rightarrow \text{ptn}(\{\{a\}, \{C\}\}); \emptyset \right) \right\}$$

The second formula yields the following antecedent mappings:

$$X = a, Y = C, S = \emptyset, T = \emptyset, \gamma = \{B/a\};$$

$$X = C, Y = a, S = \emptyset, T = \emptyset, \gamma = \{B/a\}.$$

The first mapping leads to the following consequent:

$$\left\{ \left(\{C\} \cup \emptyset \Rightarrow \text{ptn}(\{\{a\}\} \cup \emptyset); \{B/a\} \right), \left(\{a\} \cup \emptyset \Rightarrow \text{ptn}(\emptyset); \{B/a, C/a\} \right) \right\} \\ = \left\{ \left(\{C\} \Rightarrow \text{ptn}(\{\{a\}\}); \{B/a\} \right), \left(\{a\} \Rightarrow \text{ptn}(\emptyset); \{B/a, C/a\} \right) \right\}$$

The second mapping gives:

$$\left\{ \left(\{a\} \cup \emptyset \Rightarrow \text{ptn}(\{\{C\}\} \cup \emptyset); \{B/a\} \right), \left(\{a\} \cup \emptyset \Rightarrow \text{ptn}(\emptyset); \{B/a, C/a\} \right) \right\} \\ = \left\{ \left(\{a\} \Rightarrow \text{ptn}(\{\{C\}\}); \{B/a\} \right), \left(\{a\} \Rightarrow \text{ptn}(\emptyset); \{B/a, C/a\} \right) \right\}$$

The combination of these results is:

$$\left\{ \left(\{C\} \Rightarrow \text{ptn}(\{\{a\}\}); \{B/a\} \right), \left(\{a\} \Rightarrow \text{ptn}(\{\{C\}\}); \{B/a\} \right), \right. \\ \left. \left(\{a\} \Rightarrow \text{ptn}(\emptyset); \{B/a, C/a\} \right) \right\}$$

The third formula results are produced in a fashion similar to that shown for the second formula (swapping 'B' and 'C'):

$$\left\{ \left(\{B\} \Rightarrow \text{ptn}(\{\{a\}\}); \{C/a\} \right), \left(\{a\} \Rightarrow \text{ptn}(\{\{B\}\}); \{C/a\} \right), \right. \\ \left. \left(\{a\} \Rightarrow \text{ptn}(\emptyset); \{B/a, C/a\} \right) \right\}$$

The fourth formula gives the combined results:

$$\left\{ \left(\{C\} \Rightarrow \text{ptn}(\{\{a\}, \{B\}\}); \emptyset \right), \left(\{a\} \Rightarrow \text{ptn}(\{\{B\}\}); \{C/a\} \right), \right. \\ \left. \left(\{a\} \Rightarrow \text{ptn}(\{\{B\}, \{C\}\}); \emptyset \right) \right\}$$

The fifth formula gives the combined results:

$$\left\{ \left(\{B\} \Rightarrow \text{ptn}(\{\{a\}\}); \{C/B\} \right), \left(\{a\} \Rightarrow \text{ptn}(\{\{B\}\}); \{C/B\} \right), \right. \\ \left. \left(\{a\} \Rightarrow \text{ptn}(\emptyset); \{B/a, C/a\} \right) \right\}$$

The sixth formula in F2 uses Rule 1 giving as its combined results:

$$\left\{ \left(\{B\} \Rightarrow \text{ptn}(\{\{a\}, \{C\}\}); \emptyset \right), \left(\{a\} \Rightarrow \text{ptn}(\{\{C\}\}); \{B/a\} \right), \right. \\ \left. \left(\{a\} \Rightarrow \text{ptn}(\{\{B\}, \{C\}\}); \emptyset \right) \right\}$$

Combing all of these results for each formula gives formula set 3:

$$\Phi_3 = \left\{ \begin{array}{l} \left(\{C\} \Rightarrow ptn(\{\{a\}, \{B\}\}; \emptyset), \{B\} \Rightarrow ptn(\{\{a\}\}; \{C/B\}) \right), \\ \left(\{B\} \Rightarrow ptn(\{\{a\}, \{C\}\}; \emptyset), \{C\} \Rightarrow ptn(\{\{a\}\}; \{B/a\}) \right), \\ \left(\{a\} \Rightarrow ptn(\emptyset; \{B/a, C/a\}), \{a\} \Rightarrow ptn(\{\{C\}\}; \{B/a\}) \right), \\ \left(\{B\} \Rightarrow ptn(\{\{a\}\}; \{C/a\}), \{a\} \Rightarrow ptn(\{\{B\}\}; \{C/a\}) \right), \\ \left(\{a\} \Rightarrow ptn(\{\{B\}, \{C\}\}; \emptyset), \{a\} \Rightarrow ptn(\{\{B\}\}; \{C/B\}) \right) \end{array} \right\}$$

All of the formulas in ' Φ_3 ' have singleton sets on the left-hand-side of the arrow, so Rule 1 cannot apply to any of them. Thus, they are all processed by Rule 2. This rule moves the contents of the left-hand-side set to the right-hand-side, putting each element in its own part within the partitioned set:

$$\Phi_4 = \left\{ \begin{array}{l} \left(\emptyset \Rightarrow ptn(\{\{a\}, \{B\}, \{C\}\}; \emptyset), \emptyset \Rightarrow ptn(\{\{a\}, \{B\}\}; \{C/B\}) \right), \\ \left(\emptyset \Rightarrow ptn(\{\{a\}, \{C\}\}; \{B/a\}), \emptyset \Rightarrow ptn(\{\{a\}\}; \{B/a, C/a\}) \right), \\ \left(\emptyset \Rightarrow ptn(\{\{a\}, \{B\}\}; \{C/a\}) \right) \end{array} \right\}$$

Since all of the formulas in ' Φ_4 ' have empty left-hand-sides of the arrow, no more processing needs to be done. The final set of atomized partitioned set/substitution pairs is as shown at the beginning of this example.

General Unification.

The general unification algorithm takes a unification formula and transforms it into a set of bindings. This transformation applies the a set of rules in a certain order. The variable unification rules, Rule 20 through Rule 22, are applied at every opportunity to process simple variable unifications. First the set unification rules, Rule 3 through Rule 5, are applied to eliminate set unifications either by replacing them with unification of their elements (for singleton sets) or by converting them into partitioned sets. Then, the partitioned set unification rules are applied. This may introduce new set unifications, which recursively invokes the unification process.

Within the partitioned set unification rules, there is an ordering into five steps: Rule 8, Rule 9, Rule 10, Rule 11, then Rule 12 through Rule 19. At each step in this ordering, the rules of that step are applied repeatedly until they can no longer be

applied, then the process moves on to the next step.

A fully specified rule has the form:

$$\frac{U; \Gamma; V; \Sigma}{\left\{ \begin{array}{l} \langle U_1; \Gamma_1; V_1; \Sigma_1 \rangle, \\ \vdots \\ \langle U_n; \Gamma_n; V_n; \Sigma_n \rangle \end{array} \right\}}$$

where U is a set of unification equations, Γ is a set of constraints (the accumulated pairwise disjointness constraints from the partitioned sets present at any time in the process of “solving” the unification problem), V is the set of all of the variables in U and Γ , and Σ is the set of bindings developed for the unification problem. In this form, a rule may convert a single unification problem in the antecedent into a set of unification problems. This transformation of a single unification problem into multiple unification problems reflects the underlying multiplicity of possible solutions when unifying sets.

Set Unification.

Singleton set unification is equivalent to the unification of their elements:

$$\frac{\{\{A\} \doteq \{B\}\} \cup \Lambda; \Gamma; V; \Sigma}{\{A \doteq B\} \cup \Lambda; \Gamma; V; \Sigma} \quad (Rule.3)$$

where $(A \doteq B) \notin \Lambda$

This rule converts a unification such as ‘ $\{a\} = \{b\}$ ’ to the simpler ‘ $a = b$ ’. All of the other parts of the “mapped” unification formula are unchanged. The condition of this rule simply ensures that the mapping to the given set of unifications divides that set into two disjoint parts, the unification of interest (‘ $\{A\} \doteq \{B\}$ ’) and all of the other unifications (‘ Λ ’). This condition is present in some form for all of the unification rules.

Unifications which involve sets of more than one element are converted to atomized partitioned set unifications by Rule 4:

$$\begin{array}{c}
\frac{\{S \doteq T\} \cup \Lambda; \Gamma; V; \Sigma}{\left\{ \left(\{P \doteq \sigma / T\}; \sigma / (\Gamma \cup \gamma); V \cup \alpha; \sigma \circ \Sigma \right) \middle| \langle P, \sigma, \alpha, \gamma \rangle \in AS \right\}} \\
\text{if } \text{cardinality}(S) > 1 \wedge AS = \text{atomization}(S, \Gamma, V) \\
\wedge (S \doteq T) \notin \Lambda
\end{array} \quad (\text{Rule.4})$$

This rule's consequent generates a set of formulas. The atomization process may create a set of possible atomizations and we need a consequent formula for each possible atomization. A possible atomization is a 4-tuple of an atomized partitioned set, the substitution used to create that partitioned set, the new variables created for that atomized partitioned set, and the new partitioning constraints created for that atomized partitioned set.

This rule converts the unification ' $\{a, b\} = \{X, Y\}$ ' to ' $\text{ptn}(\{\{a\}, \{b\}\}) = \{X, Y\}$ '. More complicated structures involving variables in sets inside sets on the left hand side of the unification problem would make interesting use of the per-atomized-partitioned-set substitution, variable set, and constraint set.

We handle the case of a set appearing on the right-hand-side of a unification by moving it over to the left-hand-side, so that Rule 4 can then be applied to it. This set-commuting is specified by Rule 5:

$$\begin{array}{c}
\frac{\{T \doteq S\} \cup \Lambda; \Gamma; V; \Sigma}{\{S \doteq T\} \cup \Lambda; \Gamma; V; \Sigma} \\
\text{if } \text{cardinality}(S) > 1 \wedge \neg \text{var}(T) \wedge \neg \text{set}(T) \\
\wedge (T \doteq S) \notin \Lambda
\end{array} \quad (\text{Rule.5})$$

The not-variable condition in Rule 5 avoids “undoing” the work of the variable commuting rule (Rule 21). The requirement that it is not a set keeps this rule from commuting formula to which Rule 3 or Rule 4 might be applied. This rule would flip ' $\text{ptn}(X) = \{a, b\}$ ' to ' $\{a, b\} = \text{ptn}(X)$ ', but would leave ' $\{a\} = \{X\}$ ' alone.

Partitioned Set Unification.

Partitioned set unification proceeds in two phases. The first phase is a preparation phase and the second phase is the core partitioned set unification process.

Ground unification. At each step of the unification process, all unification formulas are checked for ground unifications. Each ground unification formula (one that contains no variables) is checked for validity. If it is valid, then it is removed from the set of unifications in its containing formula, otherwise the unification formula to which it belongs is invalid and that formula is removed from the unification formula set.

The two rules that specify this ground unification are defined using the ‘ground’ predicate, ‘ $\text{canon}(T)$ ’ function, and ‘ $\text{setequal}(S, T)$ ’ predicate. The ground predicate is true if its argument is a term that does not contain any variables. The canon function takes a term and replaces partitioned sets by simple sets. Since the partitioned sets are ground, the pairwise disjoint constraint is no longer needed (there are no variables to constrain). Thus we can simplify them to be just sets. The canon function uses two predicates: ‘ $\text{is_constant}(T)$ ’ and ‘ $\text{is_nonempty_set}(T)$ ’. The is_constant predicate is true if its argument is an ur term or an empty set. The is_nonempty_set predicate is true when its argument is nonempty set term (which does not include partitioned set terms). An example canonicalization is:

$$\text{canon}\left(\text{ptn}\left(\left\{\text{ptn}\left(\left\{\{a\}, \{b\}\right\}\right), \{c\}\right\}\right)\right) = \{a, b, c\}$$

$$\text{canon}(T) = \begin{cases} \bigcup \{\text{canon}(p) \mid p \in P\} & \text{if } T = \text{ptn}(P) \\ T & \text{if } \text{is_constant}(T) \\ \{\text{canon}(s) \mid s \in T\} & \text{if } \text{is_nonempty_set}(T) \end{cases}$$

The setequal predicate is true when its two arguments are equivalent sets. This assumes a simple “identical symbols” equality theory for ‘=’, such as that specified in Clarke’s Equality Theory. An example true proposition is:

$$\text{setequal}\left(\left\{\{a, b\}, c\right\}, \left\{c, \{a, b\}\right\}\right)$$

$$\text{setequal}(T_1, T_2) \equiv \left(\begin{array}{l} (\text{is_nonempty_set}(T_1) \wedge \text{is_nonempty_set}(T_2)) \\ \wedge \forall s (s \in T_1 \rightarrow \exists t \in T_2 (\text{setequal}(s, t))) \\ \wedge \forall s (s \in T_2 \rightarrow \exists t \in T_1 (\text{setequal}(s, t))) \\ \vee \left(\begin{array}{l} \text{is_constant}(T_1) \wedge \text{is_constant}(T_2) \\ \wedge T_1 = T_2 \end{array} \right) \end{array} \right)$$

The rules handle the two possibilities for a ground unification, either the unification succeeds or fails. The form of these rules is a slight extension of the general form of the unification rules. In these rules we have made the antecedent be the entire set of unification formulas, instead of being a single formula. This allows us to express the failure case. Rule 6 is the success case. The canonical forms of the two ground terms are found to be setequal. The consequent modifies the selected unification formula so that the valid unification is removed from it. The consequent set of unification formulas is the same as the antecedent set with only the selected unification formula trimmed in this way.

$$\frac{\{(\{X \doteq Y\} \cup \Lambda; \Gamma; V; \Sigma)\} \cup \Pi}{\{(\Lambda; \Gamma; V; \Sigma)\} \cup \Pi} \quad \text{where } \text{ground}(X) \wedge \text{ground}(Y) \quad (\text{Rule.6})$$

$$\wedge \text{setequal}(\text{canon}(X), \text{canon}(Y))$$

$$\wedge (\{X \doteq Y\} \cup \Lambda; \Gamma; V; \Sigma) \notin \Pi \wedge (X \doteq Y) \notin \Lambda$$

Rule 7 is the failure case. The canonical forms of the two ground terms are not “setequal.” The set of unification formulas of the consequent is the set of the antecedent minus the unification formula containing the offending unification.

$$\frac{\{(\{X \doteq Y\} \cup \Lambda; \Gamma; V; \Sigma)\} \cup \Pi}{\Pi} \quad \text{where } \text{ground}(X) \wedge \text{ground}(Y) \quad (\text{Rule.7})$$

$$\wedge \neg \text{setequal}(\text{canon}(X), \text{canon}(Y))$$

$$\wedge (\{X \doteq Y\} \cup \Lambda; \Gamma; V; \Sigma) \notin \Pi \wedge (X \doteq Y) \notin \Lambda$$

Preparation. The preparation phase uses three rules: remove identical elements (Rule 8), introduce hollow partitioned sets (Rule 9), and connect hollow partitioned set parts (Rule 10).

Remove Identical Elements. The two atomized partitioned sets being unified may contain identical parts. These identical parts must be unified with each other and will not contribute to the substitution resulting from the unification of the two partitioned sets. So, we simply remove the identical parts from each of the partitioned sets being unified.

$$\frac{\{\text{ptn}(X \cup A) \doteq \text{ptn}(X \cup B)\} \cup \Lambda; \Gamma; V; \Sigma}{\{\text{ptn}(A) \doteq \text{ptn}(B)\} \cup \Lambda; \Gamma; V; \Sigma} \quad (\text{Rule.8})$$

where $A \cap B = \emptyset \wedge (\text{ptn}(X \cup A) \doteq \text{ptn}(X \cup B)) \notin \Lambda$

The $A \cap B = \emptyset$ constraint ensures that the resulting unification equation has all identical elements removed. This notion of identity is not simply syntactic, but model-based. That is, a subset of the elements of P and a subset of the elements of Q are identical if for all interpretations I, they are interpreted to the same element of the model of I. This eliminates syntactic inequalities such as that between the sets {a, b} and {b, a}. This rule would simplify ‘ $\text{ptn}(\{\{a, Z\}\}, \{X\}, \{b\})$ ’ = $\text{ptn}(\{\{a\}, Y, \{\{Z, a\}\}\}$ ’ to ‘ $\text{ptn}(\{\{X\}, \{b\}\}) = \text{ptn}(\{\{a\}, Y\})$ ’.

Introduce Hollow Partitioned Sets. Rule 9 implements the “hollowing” of variable parts of (atomized) partitioned sets. Each part of an atomized partitioned set is either a singleton set or a variable. This rule transforms the variable parts; it leaves the singleton set parts unchanged. In the introduction of this chapter we briefly presented the need to unify arbitrary subsets of partitioned sets, where the subsets are represented by variable parts. We allow for this by replacing a variable part with a partitioned set that consists only of variable parts, where these variables are not used anywhere else in the unification problem (the original variable is unified with this new “hollow” partitioned set). Suppose one is unifying the two atomized partitioned sets ‘ $\text{ptn}(S) = \text{ptn}(T)$ ’, and that we are processing the variable parts of ‘S’. A new hollow partitioned set replacing a variable part of ‘S’ has as many parts as there are in the other (atomized) partitioned set, ‘T’. The variable parts of the new hollow partitioned set are either null or they are connected to ‘T’. If a variable part of a hollow parti-

tioned set is connected to 'T', then it is bound to a singleton set part of 'T' or a variable part of a hollowed partitioned set part of 'T'. This approach to hollowing and connecting parts of 'S' and 'T' allows for all possible subset relationships between them.

This rule introduces hollow partitioned sets to 'S' and 'T', the next rule connects the hollow partitioned set parts of 'S' to the hollow partitioned set parts of 'T'. The conditional part of this rule partitions 'S' into 'X' and 'A', where 'X' contains all of the variable parts of 'S' and 'A' contains everything else (all of the singleton set parts of 'S'). Similarly, 'T' is partitioned into its variables parts 'Y' and singleton set parts 'B'.

The consequent of this rule builds a single new formula. The first portion of the formula is the unification equations. There are three sets of unification equations: 'Ex', the unifications of each variable part of 'X' with its replacement hollowed partitioned set; 'Ey', the unifications of each variable part of 'Y' with its replacement hollowed partitioned set; and the unification with the transformed 'X' and 'Y'.

The second portion of the formula is the set of constraints. This rule creates new partitioned sets, the hollow partitioned sets that replace the variable parts of 'S' and 'T', so there are new partitioning constraints of which to keep track. These partitioning constraints are represented by ' $\text{ptncon}\left(X' \cup_{\text{multi}} A\right)$ ' and ' $\text{ptncon}\left(Y' \cup_{\text{multi}} B\right)$ '. (The 'ptncon' function is used solely for representing partitioning constraints.)

The third portion of the new unification formula is the set of variables. This is extended by this rule with the variables introduced in the new hollow partitioned sets. This rule is the only rule that uses the set of variables, in addition to being the only one that modifies it. The set of variables is used to ensure that the variables used in the new hollow partitioned sets have not already been used.

$$\begin{array}{c}
\frac{\{ \text{ptn}(X \cup A) \doteq \text{ptn}(Y \cup B) \} \cup \Lambda; \Gamma; V}{\left(\begin{array}{l} \{ \text{ptn}(X' \cup A) \doteq \text{ptn}(Y' \cup B) \} \cup \Lambda; \\ (\sigma_X \circ \sigma_Y) / \left(\Gamma \cup \text{ptncon} \left(X' \cup A \right) \cup \text{ptncon} \left(Y' \cup B \right) \right); \\ V \cup \text{varsin}(\sigma_X) \cup \text{varsin}(\sigma_Y); \\ \Sigma \circ \sigma_X \circ \sigma_Y \end{array} \right)} \\
\text{where } \left(\begin{array}{l} \forall v \in X(\text{var}(v)) \\ \wedge \neg \exists v \in A(\text{var}(v)) \end{array} \right) \wedge \left(\begin{array}{l} \forall v \in Y(\text{var}(v)) \\ \wedge \neg \exists v \in B(\text{var}(v)) \end{array} \right) \\
\wedge (\text{ptn}(X \cup A) \doteq \text{ptn}(Y \cup B)) \notin \Lambda
\end{array} \quad (Rule.9)$$

$$\sigma_X = \text{hollow_partitioned_set_subs}(X, B, Y, V)$$

$$\sigma_Y = \text{hollow_partitioned_set_subs}(Y, A, X, V \cup \text{vars_in}(\sigma_X))$$

$$X' = \text{partitioned_sets}(\sigma_X)$$

$$Y' = \text{partitioned_sets}(\sigma_Y)$$

$$\text{hollow_partitioned_set_subs}(P, Q, R, V) = \left\{ (v / \text{ptn}(\{X_{v_1}, \dots, X_{v_k}\})) \mid v \in P \right\}$$

$$\text{if } k = \text{cardinality}(Q \cup R) \wedge (\forall s, t, u, v (X_{st} = X_{uv} \rightarrow s = u \wedge t = v))$$

$$\wedge \forall s, t (\text{var}(X_{st}) \wedge X_{st} \notin V)$$

$$\text{partitioned_sets}(P) = \left\{ Q \mid \exists v ((v / Q) \in P) \right\}$$

The definition of this rule uses four equations based on two special functions, ‘hollow_partitioned_set_subs(P,Q,R,V)’ and ‘partitioned_sets(P)’. The ‘hollow_partitioned_set_subs(P,Q,R,V)’ function does the core work of this rule. It builds the hollow partitioned sets and associates them with the variable parts. It associates a hollow partitioned set, call it ‘H’, with each variable in ‘P’, where there are as many parts in ‘H’ as elements in the union of ‘Q’ and ‘R’. The variables of ‘H’ are not in ‘V’, (i.e. they are “new”). The value of the function is the set of substitutions unifying each variable part with its replacement hollow partitioned set. The equation defining ‘ σ_x ’ sets ‘ σ_x ’ equal to the equations unifying each element of ‘X’ with hollow partitioned

sets each having as many parts as the cardinality of the union of ‘ B ’ and ‘ Y ’, and none of the hollow partitioned set variable parts are in ‘ V ’. Similarly, the equation defining ‘ σ_y ’ sets ‘ σ_y ’ equal to the equations unifying each element of ‘ Y ’ with hollow partitioned sets each having as many parts as the cardinality of the union of ‘ A ’ and ‘ X ’, and none of the hollow partitioned set variable parts are in the union of ‘ V ’ and the variables in ‘ σ_x ’.

The other special function, ‘ $\text{partitioned_sets}(P)$ ’, maps ‘ P ’ to the right-hand-side of the substitutions in ‘ P ’. When ‘ $\text{partitioned_sets}(P)$ ’ is given a ‘ P ’ that contains only partitioned sets on the right-hand-sides of the unification equations, it returns those partitioned sets. This is how it is used in the equations defining X' and Y' . X' is the ‘ partitioned_sets ’ of ‘ σ_x ’, which we have explained above is a set of equations unifying the elements of X with hollow partitioned sets. Thus, X' is the set of hollow partitioned sets associated by ‘ σ_x ’ with the elements of X . Similarly, Y' is the set of hollow partitioned sets associated by ‘ σ_y ’ with the elements of Y .

As an example of the application of this rule, consider the unification:

$$\left\{ \text{ptn}(\{S_1, S_2, S_3, \{a\}, \{D\}\}) \doteq \text{ptn}(\{T_1, T_2, \{b\}, \{E\}\}) \right\}$$

where $S_1, S_2, S_3, D, T_1, T_2, E$ are variables and ‘ a ’ and ‘ b ’ are constants. Mapping the antecedent of this rule onto the example unification equation in a fashion that honors the rule’s conditional yields:

$$'X = \{S_1, S_2, S_3\}, A = \{\{a\}, \{D\}\}' \text{ and } 'Y = \{T_1, T_2\}, B = \{\{b\}, \{E\}\}'.$$

This gives:

$$\begin{aligned} \sigma_x &= \text{hollow_partitioned_set_subs} \left(\begin{array}{l} \{S_1, S_2, S_3\}, \{\{b\}, \{E\}\}, \\ \{T_1, T_2\}, \{S_1, S_2, S_3, T_1, T_2\} \end{array} \right) \\ &= \left\{ \begin{array}{l} S_1 / \text{ptn}(\{S_4, S_5, S_6, S_7\}), S_2 / \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}), \\ S_3 / \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}) \end{array} \right\} \end{aligned}$$

and

$$\begin{aligned} \sigma_y &= \text{hollow_partitioned_set_subs} \left(\begin{array}{l} \{T_1, T_2\}, \{\{a\}, \{D\}\}, \{S_1, S_2, S_3\}, \\ \{S_1, \mathbf{I}, S_{15}, T_1, T_2\} \end{array} \right) \\ &= \left\{ T_1 / \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}) \right\} \end{aligned}$$

The equations for X' and Y' are:

$$\begin{aligned}
X' &= \text{partitioned_sets}(\sigma_X) \\
&= \text{partitioned_sets} \left(\left(\left[\begin{array}{l} S_1 / \text{ptn}(\{S_4, S_5, S_6, S_7\}), S_2 / \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}) \\ S_3 / \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}) \end{array} \right] \right) \right) \\
&= \left\{ \text{ptn}(\{S_4, S_5, S_6, S_7\}), \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}), \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}) \right\}
\end{aligned}$$

$$\begin{aligned}
Y' &= \text{partitioned_sets}(\sigma_Y) \\
&= \text{partitioned_sets} \left(\left(\left[\begin{array}{l} T_1 / \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}) \end{array} \right] \right) \right) \\
&= \left\{ \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}) \right\}
\end{aligned}$$

We can now construct the unifications portion of the consequent of Rule 9:

$$\begin{aligned}
&\left\{ \text{ptn}(X' \cup A) \doteq \text{ptn}(Y' \cup B) \right\} \\
&= \left\{ \text{ptn} \left(\left(\left[\begin{array}{l} \left(\left[\begin{array}{l} \text{ptn}(\{S_4, S_5, S_6, S_7\}), \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}) \\ \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}) \end{array} \right] \right) \\ \cup \{a\}, \{D\} \end{array} \right] \right) \right\} \\
&\quad \left\{ \text{ptn} \left(\left(\left[\begin{array}{l} \left(\left[\begin{array}{l} \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}) \end{array} \right] \right) \\ \cup \{b\}, \{E\} \end{array} \right] \right) \right\} \right\} \\
&= \left\{ \text{ptn} \left(\left(\left[\begin{array}{l} \left(\left[\begin{array}{l} \text{ptn}(\{S_4, S_5, S_6, S_7\}), \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}) \\ \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}), \{a\}, \{D\} \end{array} \right] \right) \\ \left(\left[\begin{array}{l} \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), \\ \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}), \{b\}, \{E\} \end{array} \right] \right) \end{array} \right] \right) \right\}
\end{aligned}$$

The combined substitutions are used in calculating the constraints and total substitutions for the consequent formula. They are:

$$\sigma_X \circ \sigma_Y$$

$$= \left(\left[\begin{array}{l} \{ S_1 / \text{ptn}(\{S_4, S_5, S_6, S_7\}), S_2 / \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}) \} \\ \{ S_3 / \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}) \} \\ \{ T_1 / \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}) \} \end{array} \right] \right)$$

$$= \left[\begin{array}{l} \{ S_1 / \text{ptn}(\{S_4, S_5, S_6, S_7\}), S_2 / \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}), \\ \{ S_3 / \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}), \\ \{ T_1 / \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}) \} \end{array} \right]$$

The constraints portion of the consequent formula is:

$$(\sigma_X \circ \sigma_Y) / \left(\Gamma \cup \text{ptncon} \left(X' \cup A \right) \cup \text{ptncon} \left(Y' \cup B \right) \right)$$

$$= (\sigma_X \circ \sigma_Y) / \left(\begin{array}{l} \left(\emptyset \right. \\ \left. \left[\begin{array}{l} \left[\begin{array}{l} \text{ptncon} \left(\left[\begin{array}{l} \text{ptn}(\{S_4, S_5, S_6, S_7\}), \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}) \end{array} \right] \right) \right] \\ \left[\begin{array}{l} \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}), \{a\}, \{D\} \end{array} \right] \end{array} \right]_{multi} \right] \\ \cup \left[\begin{array}{l} \left[\begin{array}{l} \text{ptncon} \left(\left[\begin{array}{l} \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), \\ \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}), \{b\}, \{E\} \end{array} \right] \right) \right] \\ \left[\begin{array}{l} \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}), \{b\}, \{E\} \end{array} \right] \end{array} \right]_{multi} \right] \end{array} \right) \end{array} \right)$$

$$= (\sigma_X \circ \sigma_Y) / \left\{ \begin{array}{l} \left[\begin{array}{l} \text{ptncon} \left(\left[\begin{array}{l} \text{ptn}(\{S_4, S_5, S_6, S_7\}), \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}) \end{array} \right] \right) \right] \\ \left[\begin{array}{l} \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}), \{a\}, \{D\} \end{array} \right] \end{array} \right]_{multi} \\ \left[\begin{array}{l} \text{ptncon} \left(\left[\begin{array}{l} \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), \\ \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}), \{b\}, \{E\} \end{array} \right] \right) \right] \\ \left[\begin{array}{l} \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}), \{b\}, \{E\} \end{array} \right] \end{array} \right]_{multi} \end{array} \right\}$$

$$= (\sigma_X \circ \sigma_Y) / \left\{ \begin{array}{l} \text{ptncon}(\{S_4, \mathbf{L}, S_{15}, \{a\}, \{D\}\}_{multi}), \\ \text{ptncon}(\{T_3, \mathbf{L}, T_{12}, \{b\}, \{E\}\}_{multi}) \end{array} \right\}$$

$$\begin{aligned}
& \left(\left[\begin{array}{l} S_1 / \text{ptn}(\{S_4, S_5, S_6, S_7\}), S_2 / \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}), \\ S_3 / \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}), \end{array} \right] \right) \\
&= \left[\begin{array}{l} T_1 / \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}) \\ \left[\begin{array}{l} \text{ptncon}(\{S_4, \mathbf{L}, S_{15}, \{a\}, \{D\}\}_{multi}), \\ \text{ptncon}(\{T_3, \mathbf{L}, T_{12}, \{b\}, \{E\}\}_{multi}) \end{array} \right] \end{array} \right] \\
&= \left[\begin{array}{l} \text{ptncon}(\{S_4, \mathbf{L}, S_{15}, \{a\}, \{D\}\}_{multi}), \\ \text{ptncon}(\{T_3, \mathbf{L}, T_{12}, \{b\}, \{E\}\}_{multi}) \end{array} \right]
\end{aligned}$$

For our example, there were no constraints to start with (hence Γ is empty). The final form of the ‘ptncon’ terms is a simplification where a part of a partitioned set that is itself a partitioned set can be replaced by its parts (partitioned set terms can be “flattened” and the result is semantically equivalent).

The variables portion of the consequent formula is:

$$V \cup \text{varsin}(\sigma_x) \cup \text{varsin}(\sigma_y)$$

$$\begin{aligned}
& \left(\{D, E, S_1, S_2, S_3, T_1, T_2\} \right) \\
&= \left(\cup \text{varsin} \left(\left[\begin{array}{l} S_1 / \text{ptn}(\{S_4, S_5, S_6, S_7\}), S_2 / \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}), \\ S_3 / \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}) \end{array} \right] \right) \right) \\
& \quad \left(\cup \text{varsin} \left(\left[\begin{array}{l} T_1 / \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}) \end{array} \right] \right) \right) \\
&= \{D, E, S_1, \mathbf{L}, S_{15}, T_1, \mathbf{L}, T_{12}\}
\end{aligned}$$

The substitutions portion of the consequent formula is:

$$\Sigma \circ \sigma_x \circ \sigma_y$$

$$\begin{aligned}
&= \emptyset \circ \left[\begin{array}{l} S_1 / \text{ptn}(\{S_4, S_5, S_6, S_7\}), S_2 / \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}), \\ S_3 / \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}), \\ T_1 / \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}) \end{array} \right] \\
&= \left[\begin{array}{l} S_1 / \text{ptn}(\{S_4, S_5, S_6, S_7\}), S_2 / \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}), \\ S_3 / \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}), \\ T_1 / \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}) \end{array} \right]
\end{aligned}$$

The result of applying the “hollow partitions” rule to the example unification formula with an empty constraint set can be summarized by:

$$\begin{aligned}
& \text{apply} \left(\text{hollow_partitions}, \left(\left\{ \text{ptn}(\{S_1, S_2, S_3, \{a\}, \{D\}\}) \doteq \text{ptn}(\{T_1, T_2, \{b\}, \{E\}\}) \right\}; \emptyset; \right) \right) \\
& \left(\left\{ D, E, S_1, S_2, S_3, T_1, T_2 \right\} \right) \\
& \left(\left\{ \left(\left\{ \text{ptn}(\{S_4, S_5, S_6, S_7\}), \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}) \right\} \right) \right\} \right) \\
& \left(\left\{ \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}), \{a\}, \{D\} \right\} \right) \\
& \left(\left\{ \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), \right\} \right) \\
& \left(\left\{ \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}), \{b\}, \{E\} \right\} \right) \\
& = \left\{ \text{ptncon}(\{S_4, \mathbf{L}, S_{15}, \{a\}, \{D\}\}), \right\} \\
& \left\{ \text{ptncon}(\{T_3, \mathbf{L}, T_{12}, \{b\}, \{E\}\}) \right\} \\
& \{D, E, S_1, \mathbf{L}, S_{15}, T_1, \mathbf{L}, T_{12}\}; \\
& \left\{ S_1 / \text{ptn}(\{S_4, S_5, S_6, S_7\}), S_2 / \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}), \right. \\
& \left. S_3 / \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}), \right. \\
& \left. T_1 / \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}) \right\}
\end{aligned}$$

We will continue with the processing of this example in the discussion of the next rule.

Connect Hollow Partitioned Set Parts In the preceding discussion of Rule 9 we stated that given a unification ‘ptn(S)=ptn(T)’, the variable parts of the new hollow partitioned set in ‘S’ are either null or they are connected to ‘T’. And further that if a variable part of a hollow partitioned set is connected to ‘T’, then it is bound to a singleton set part of ‘T’ or a variable part of a hollowed partitioned set part of ‘T’. More particularly, there is a variable unification for each pair of hollowed parts of ‘S’ and ‘T’. That is, for each hollow partitioned set introduced into ‘S’, there is one of that partitioned set’s variable parts that is unified with a variable part in each of the hollowed partitioned sets of ‘T’ (without unifying a variable part of a hollow partitioned set of ‘S’ with another variable part of a hollow partitioned set of ‘S’). Thus, the hollow partitioned sets in ‘S’ and ‘T’ are fully cross-linked.

The diagram shows two vertical columns representing code blocks. The left column is labeled 'S' and contains blocks S_1 , S_2 , S_3 , a , and X from top to bottom. The right column is labeled 'T' and contains blocks T_1 , T_2 , b , and Y from top to bottom. Arrows indicate the mapping: S_1 maps to T_1 , S_2 maps to T_2 , S_3 maps to T_2 , a maps to b , and X maps to Y .

Rule 10 continues where Rule 9 left off. The antecedent of Rule 10 maps a unification of partitioned sets 'ptn(S)=ptn(T)'. The conditional for the rule requires that 'S' and 'T' have been hollowed, as specified in Rule 9. X' is the set of hollow partitioned sets introduced into 'S' and 'A' are all of the other parts of 'S'. Similarly for 'T' with respect to Y' and 'B'. The consequent of the rule expresses the cross-linking of 'S' and 'T' and the reduced form of the unifications for 'S' and 'T' that haven't been processed yet. That is, the variables that are unified as a result of the cross-linking process are removed from the unification problem of the antecedent.

119

$$\sigma_{X'Y'} = \left\{ u/v \mid L \in \left\{ P \times Q \mid \langle \text{ptn}(P), \text{ptn}(Q) \rangle \in X' \times Y' \right\} \wedge \langle u, v \rangle = \text{choice}(L) \right\}$$

$$\text{where } \forall (u/v) \in \sigma_{X'Y'} \left(\begin{array}{l} \neg \exists u' (u' \neq u \wedge (u'/v) \in \sigma_{X'Y'}) \\ \wedge \neg \exists v' (v' \neq v \wedge (u/v') \in \sigma_{X'Y'}) \end{array} \right)$$

$$X'' = \text{trimmed_partitioned_sets}(X', \sigma_{X'Y'})$$

$$Y'' = \text{trimmed_partitioned_sets}(Y', \sigma_{X'Y'})$$

$$\text{trimmed_partitioned_sets}(R, S) = \left\{ P \mid \begin{array}{l} \text{ptn}(Q) \in R \\ \wedge P = \text{ptn} \left(\left\{ u \mid \begin{array}{l} u \in Q \\ \wedge \neg \exists v ((u/v) \in S \vee (v/u) \in S) \end{array} \right\} \right) \end{array} \right\}$$

The expression of the consequent of Rule 10 uses four equations and a special function, ‘trimmed_partitioned_sets(R,S)’. The cross-linking unifications are the value of $\sigma_{X'Y'}$. $\sigma_{X'Y'}$ is the set of substitutions such that there is one substitution linking each pair of hollow partitioned sets where the first element of the pair is from X' and the second element is from Y' . The “where” condition on the definition of $\sigma_{X'Y'}$ prevents any variable from being in more than one unification.

The trimmed_partitioned_sets function returns the partitioned sets from R trimmed according to S . A partitioned set ‘ $P=\text{ptn}(Q)$ ’ is trimmed according to S by removing all of the members of Q that are unified in S . X'' is defined as the set X' trimmed according to $\sigma_{X'Y'}$. Thus X'' is the same X' , but with the variables that were used in connecting X' to Y' removed. Y'' is defined similarly.

The consequent set of formulas, ‘ R ’, may be quite large, depending on the sizes of A and B :

$$\text{connecting_consequent_cardinality} = \sum_{i=1}^k \binom{i}{k} \binom{i}{m}$$

$$\text{where } k = \min(\text{card}(A), \text{card}(B)), m = \max(\text{card}(A), \text{card}(B))$$

The structure of this formula is largely the result of the requirement that $\text{card}(B_u) = \text{card}(A_u)$. For every possible cardinality ‘ i ’ of subsets of A and B , there are $(i \cdot \text{card}(A))$ such subsets of A and $(i \cdot \text{card}(B))$ such subsets of B . All possible pairings of these subsets is the cross product of these two sets of subsets. There are $(i \cdot \text{card}(A)) \cdot (i \cdot \text{card}(B))$ such pairings. Since k and m are the minimum and

maximum of the cardinalities of A and B , k will be the cardinality of one of them and m will be the cardinality of the other one. Thus, $(i \cdot \text{card}(A)) \cdot (i \cdot \text{card}(B)) = i \cdot k \cdot i \cdot m$. The total number of pairings of subsets of A and B is the sum over the possible cardinalities of pairs of subsets of the number of pairings at each of these cardinalities. The smallest such possible cardinality is 1 and the greatest is k , the cardinality of the smaller of the two sets (this is the greatest possible “pairing-cardinality” since there cannot be a pair of same-cardinality subsets for any cardinality greater than the cardinality of the smaller of the two sets). Typically most of these formulas will fail: they will not result in successful unifications as they are processed by later transformations that attempt to unify the elements of the various pairings.

We continue processing our example from Rule 9. There is only one formula in the set of unification formulas, so ‘ Λ ’ (the rest of the “input” formulas) is empty. The values of X' , A , Y' , and B are as before. We have new values for ‘ Γ ’ and ‘ V ’:

$$\Lambda = \emptyset$$

$$X' = \left\{ \text{ptn}\left(\{S_4, S_5, S_6, S_7\}\right), \text{ptn}\left(\{S_8, S_9, S_{10}, S_{11}\}\right), \text{ptn}\left(\{S_{12}, S_{13}, S_{14}, S_{15}\}\right) \right\}$$

$$Y' = \left\{ \text{ptn}\left(\{T_3, T_4, T_5, T_6, T_7\}\right), \text{ptn}\left(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}\right) \right\}$$

$$\Gamma = \left\{ \begin{array}{l} \text{ptncon}\left(\{S_4, \mathbf{L}, S_{15}, \{a\}, \{D\}\}\right), \\ \text{ptncon}\left(\{T_3, \mathbf{L}, T_{12}, \{b\}, \{E\}\}\right) \end{array} \right\}$$

$$V = \{D, E, S_1, \mathbf{L}, S_{15}, T_1, \mathbf{L}, T_{12}\}$$

$$\Sigma = \left\{ \begin{array}{l} S_1 / \text{ptn}\left(\{S_4, S_5, S_6, S_7\}\right), S_2 / \text{ptn}\left(\{S_8, S_9, S_{10}, S_{11}\}\right), \\ S_3 / \text{ptn}\left(\{S_{12}, S_{13}, S_{14}, S_{15}\}\right), \\ T_1 / \text{ptn}\left(\{T_3, T_4, T_5, T_6, T_7\}\right), T_2 / \text{ptn}\left(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}\right) \end{array} \right\}$$

Since $\sigma_{XY'}$ relies on making a “choice”, several different results are possible. However, they are all semantically equivalent. The “pool” of possible connections is:

$$\begin{aligned}
X' \times Y' &= \left(\left[\begin{array}{l} \text{ptn}(\{S_4, S_5, S_6, S_7\}), \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}) \\ \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}) \end{array} \right] \times \left[\begin{array}{l} \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}) \end{array} \right] \right) \\
&= \left\{ \begin{array}{l} \langle \text{ptn}(\{S_4, S_5, S_6, S_7\}), \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}) \rangle, \\ \langle \text{ptn}(\{S_4, S_5, S_6, S_7\}), \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}) \rangle, \\ \langle \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}), \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}) \rangle, \\ \langle \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}), \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}) \rangle, \\ \langle \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}), \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}) \rangle, \\ \langle \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}), \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}) \rangle \end{array} \right\} \\
&\{P \times Q \mid \langle \text{ptn}(P), \text{ptn}(Q) \rangle \in X' \times Y'\} \\
&= \left\{ \begin{array}{l} \langle \{S_4, T_3\}, \{S_4, T_4\}, \{S_4, T_5\}, \{S_4, T_6\}, \{S_4, T_7\}, \{S_5, T_3\}, \{S_5, T_4\}, \{S_5, T_5\}, \{S_5, T_6\}, \{S_5, T_7\} \rangle, \\ \langle \{S_6, T_3\}, \{S_6, T_4\}, \{S_6, T_5\}, \{S_6, T_6\}, \{S_6, T_7\}, \{S_7, T_3\}, \{S_7, T_4\}, \{S_7, T_5\}, \{S_7, T_6\}, \{S_7, T_7\} \rangle \end{array} \right\} \\
&= \mathbf{L}, \\
&\left\{ \begin{array}{l} \langle \{S_{12}, T_8\}, \{S_{12}, T_9\}, \{S_{12}, T_{10}\}, \{S_{12}, T_{11}\}, \{S_{12}, T_{12}\}, \{S_{13}, T_8\}, \{S_{13}, T_9\}, \{S_{13}, T_{10}\}, \{S_{13}, T_{11}\}, \{S_{13}, T_{12}\} \rangle, \\ \langle \{S_{14}, T_8\}, \{S_{14}, T_9\}, \{S_{14}, T_{10}\}, \{S_{14}, T_{11}\}, \{S_{14}, T_{12}\}, \{S_{15}, T_8\}, \{S_{15}, T_9\}, \{S_{15}, T_{10}\}, \{S_{15}, T_{11}\}, \{S_{15}, T_{12}\} \rangle \end{array} \right\}
\end{aligned}$$

A possible value for $\sigma_{X'Y'}$ given the above “pool” is:

$$\begin{aligned}
&\sigma_{X'Y'} \\
&= \left\{ u/v \mid L \in \{P \times Q \mid \langle \text{ptn}(P), \text{ptn}(Q) \rangle \in X' \times Y'\} \wedge \langle u, v \rangle = \text{choice}(L) \right\} \\
&= \{S_4 / T_3, S_5 / T_8, S_8 / T_4, S_9 / T_9, S_{12} / T_5, S_{13} / T_{10}\}
\end{aligned}$$

Using the value for $\sigma_{X'Y'}$, we can determine values for X'' and Y'' :

$$\begin{aligned}
X'' &= \text{trimmed_partitioned_sets}(X', \sigma_{X'Y'}) \\
&= \text{trimmed_partitioned_sets} \left(\left(\left[\begin{array}{l} \text{ptn}(\{S_4, S_5, S_6, S_7\}), \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}) \\ \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}) \end{array} \right] \times \left[\begin{array}{l} S_4 / T_3, S_5 / T_8, S_8 / T_4, S_9 / T_9, S_{12} / T_5, S_{13} / T_{10} \end{array} \right] \right) \right) \\
&= \{ \text{ptn}(\{S_6, S_7\}), \text{ptn}(\{S_{10}, S_{11}\}), \text{ptn}(\{S_{14}, S_{15}\}) \}
\end{aligned}$$

$$\begin{aligned}
Y'' &= \text{trimmed_partitioned_sets}(Y', \sigma_{XY'}) \\
&= \text{trimmed_partitioned_sets} \left(\left(\left[\begin{array}{l} \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), \\ \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}) \end{array} \right], \left[\begin{array}{l} S_4 / T_3, S_5 / T_8, S_8 / T_4, S_9 / T_9, \\ S_{12} / T_5, S_{13} / T_{10} \end{array} \right] \right) \right) \\
&= \{ \text{ptn}(\{T_6, T_7\}), \text{ptn}(\{T_{11}, T_{12}\}) \}
\end{aligned}$$

The consequent formula's A and B "target" subsets are:

$$\begin{aligned}
Targets &= \left\{ \langle A_u, A_n, B_u, B_n \rangle \left| \begin{array}{l} A_u \subseteq A \wedge A_n = A - A_u \\ \wedge B_u \subseteq B \wedge B_n = B - B_u \\ \wedge \text{card}(B_u) = \text{card}(A_u) \end{array} \right. \right\} \\
&= \left\{ \langle A_u, A_n, B_u, B_n \rangle \left| \begin{array}{l} A_u \subseteq \{\{a\}, \{D\}\} \wedge A_n = \{\{a\}, \{D\}\} - A_u \\ \wedge B_u \subseteq \{\{b\}, \{E\}\} \wedge B_n = \{\{b\}, \{E\}\} - B_u \\ \wedge \text{card}(B_u) = \text{card}(A_u) \end{array} \right. \right\} \\
&= \left\{ \langle \emptyset, \{\{a\}, \{D\}\}, \emptyset, \{\{b\}, \{E\}\} \rangle, \right. \\
&\quad \left. \langle \{\{a\}\}, \{\{D\}\}, \{\{b\}\}, \{\{E\}\} \rangle, \langle \{\{a\}\}, \{\{D\}\}, \{\{E\}\}, \{\{b\}\} \rangle, \right. \\
&\quad \left. \langle \{\{D\}\}, \{\{a\}\}, \{\{b\}\}, \{\{E\}\} \rangle, \langle \{\{D\}\}, \{\{a\}\}, \{\{E\}\}, \{\{b\}\} \rangle, \right. \\
&\quad \left. \langle \{\{a\}, \{D\}\}, \emptyset, \{\{b\}, \{E\}\}, \emptyset \rangle \right\}
\end{aligned}$$

These A and B subset targets give unifications for the consequent formula of:

$$\begin{aligned}
TargetedUnifs &= \left\{ \left[\begin{array}{l} \text{ptn}(X'') \doteq \text{ptn}(B_n), \\ \text{ptn}(Y'') \doteq \text{ptn}(A_n), \\ \text{ptn}(A_u) \doteq \text{ptn}(B_u) \end{array} \right] \left| \begin{array}{l} \langle A_u, A_n, B_u, B_n \rangle \in Targets \end{array} \right. \right\} \\
&= \left\{ \left[\begin{array}{l} \text{ptn}(\{ \text{ptn}(\{S_6, S_7\}), \text{ptn}(\{S_{10}, S_{11}\}), \text{ptn}(\{S_{14}, S_{15}\}) \}) \\ \doteq \text{ptn}(B_n), \\ \text{ptn}(\{ \text{ptn}(\{T_6, T_7\}), \text{ptn}(\{T_{11}, T_{12}\}) \}) \doteq \text{ptn}(A_n), \\ \text{ptn}(A_u) \doteq \text{ptn}(B_u) \end{array} \right] \left| \begin{array}{l} \langle A_u, A_n, B_u, B_n \rangle \in Targets \end{array} \right. \right\}
\end{aligned}$$

$$\begin{aligned}
& \left[\begin{array}{l} \left\{ \text{ptn}\left(\left\{\text{ptn}\left(\{S_6, S_7\}\right), \text{ptn}\left(\{S_{10}, S_{11}\}\right), \text{ptn}\left(\{S_{14}, S_{15}\}\right)\right\}\right) \doteq \text{ptn}\left(\{\{b\}, \{E\}\}\right), \right. \\ \left. \left\{ \text{ptn}\left(\left\{\text{ptn}\left(\{T_6, T_7\}\right), \text{ptn}\left(\{T_{11}, T_{12}\}\right)\right\}\right) \doteq \text{ptn}\left(\{\{a\}, \{D\}\}\right), \text{ptn}(\emptyset) \doteq \text{ptn}(\emptyset) \right\} \right] \\
& \left[\begin{array}{l} \left\{ \text{ptn}\left(\left\{\text{ptn}\left(\{S_6, S_7\}\right), \text{ptn}\left(\{S_{10}, S_{11}\}\right), \text{ptn}\left(\{S_{14}, S_{15}\}\right)\right\}\right) \doteq \text{ptn}\left(\{\{E\}\}\right), \right. \\ \left. \left\{ \text{ptn}\left(\left\{\text{ptn}\left(\{T_6, T_7\}\right), \text{ptn}\left(\{T_{11}, T_{12}\}\right)\right\}\right) \doteq \text{ptn}\left(\{\{D\}\}\right), \text{ptn}\left(\{\{a\}\}\right) \doteq \text{ptn}\left(\{\{b\}\}\right) \right\} \right] \\
& \left[\begin{array}{l} \left\{ \text{ptn}\left(\left\{\text{ptn}\left(\{S_6, S_7\}\right), \text{ptn}\left(\{S_{10}, S_{11}\}\right), \text{ptn}\left(\{S_{14}, S_{15}\}\right)\right\}\right) \doteq \text{ptn}\left(\{\{E\}\}\right), \right. \\ \left. \left\{ \text{ptn}\left(\left\{\text{ptn}\left(\{T_6, T_7\}\right), \text{ptn}\left(\{T_{11}, T_{12}\}\right)\right\}\right) \doteq \text{ptn}\left(\{\{a\}\}\right), \text{ptn}\left(\{\{D\}\}\right) \doteq \text{ptn}\left(\{\{b\}\}\right) \right\} \right] \\
= & \left[\begin{array}{l} \left\{ \text{ptn}\left(\left\{\text{ptn}\left(\{S_6, S_7\}\right), \text{ptn}\left(\{S_{10}, S_{11}\}\right), \text{ptn}\left(\{S_{14}, S_{15}\}\right)\right\}\right) \doteq \text{ptn}\left(\{\{b\}\}\right), \right. \\ \left. \left\{ \text{ptn}\left(\left\{\text{ptn}\left(\{T_6, T_7\}\right), \text{ptn}\left(\{T_{11}, T_{12}\}\right)\right\}\right) \doteq \text{ptn}\left(\{\{D\}\}\right), \text{ptn}\left(\{\{a\}\}\right) \doteq \text{ptn}\left(\{\{E\}\}\right) \right\} \right] \\
& \left[\begin{array}{l} \left\{ \text{ptn}\left(\left\{\text{ptn}\left(\{S_6, S_7\}\right), \text{ptn}\left(\{S_{10}, S_{11}\}\right), \text{ptn}\left(\{S_{14}, S_{15}\}\right)\right\}\right) \doteq \text{ptn}\left(\{\{b\}\}\right), \right. \\ \left. \left\{ \text{ptn}\left(\left\{\text{ptn}\left(\{T_6, T_7\}\right), \text{ptn}\left(\{T_{11}, T_{12}\}\right)\right\}\right) \doteq \text{ptn}\left(\{\{a\}\}\right), \text{ptn}\left(\{\{D\}\}\right) \doteq \text{ptn}\left(\{\{E\}\}\right) \right\} \right] \\
& \left[\begin{array}{l} \left\{ \text{ptn}\left(\left\{\text{ptn}\left(\{S_6, S_7\}\right), \text{ptn}\left(\{S_{10}, S_{11}\}\right), \text{ptn}\left(\{S_{14}, S_{15}\}\right)\right\}\right) \doteq \text{ptn}(\emptyset), \right. \\ \left. \left\{ \text{ptn}\left(\left\{\text{ptn}\left(\{T_6, T_7\}\right), \text{ptn}\left(\{T_{11}, T_{12}\}\right)\right\}\right) \doteq \text{ptn}(\emptyset), \right. \\ \left. \left\{ \text{ptn}\left(\{\{a\}, \{D\}\}\right) \doteq \text{ptn}\left(\{\{b\}, \{E\}\}\right) \right\} \right]
\end{aligned}$$

The consequent formula's constraints are:

$$\begin{aligned}
\Gamma_2 &= \sigma_{XY'} / \Gamma \\
&= \left\{ \begin{array}{l} S_4 / T_3, S_5 / T_8, S_8 / T_4, \\ S_9 / T_9, S_{12} / T_5, S_{13} / T_{10} \end{array} \right\} / \left\{ \begin{array}{l} \text{ptncon}\left(\{S_4, \mathbf{L}, S_{15}, \{a\}, \{D\}\}\right), \\ \text{ptncon}\left(\{T_3, \mathbf{L}, T_{12}, \{b\}, \{E\}\}\right) \end{array} \right\} \\
&= \left\{ \begin{array}{l} \left(\begin{array}{l} T_3, T_8, S_6, S_7, \\ T_4, T_9, S_{10}, S_{11}, \\ T_5, T_{10}, S_{14}, S_{15}, \\ \{a\}, \{D\} \end{array} \right) \\ \text{ptncon}\left(\{T_3, \mathbf{L}, T_{12}, \{b\}, \{E\}\}\right) \end{array} \right\}
\end{aligned}$$

The substitutions for the consequent formula are:

$$\begin{aligned}
\Sigma_2 &= \Sigma \circ \sigma_{XY'} \\
&= \left(\begin{array}{l} \left[\begin{array}{l} S_1 / \text{ptn}(\{S_4, S_5, S_6, S_7\}), S_2 / \text{ptn}(\{S_8, S_9, S_{10}, S_{11}\}), \\ S_3 / \text{ptn}(\{S_{12}, S_{13}, S_{14}, S_{15}\}), \\ T_1 / \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}) \end{array} \right] \\ \sigma\{S_4 / T_3, S_5 / T_8, S_8 / T_4, S_9 / T_9, S_{12} / T_5, S_{13} / T_{10}\} \end{array} \right) \\
&= \left(\begin{array}{l} \left[\begin{array}{l} S_1 / \text{ptn}(\{T_3, T_8, S_6, S_7\}), S_2 / \text{ptn}(\{T_4, T_9, S_{10}, S_{11}\}), \\ S_3 / \text{ptn}(\{T_5, T_{10}, S_{14}, S_{15}\}), \\ T_1 / \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}), \\ S_4 / T_3, S_5 / T_8, S_8 / T_4, S_9 / T_9, S_{12} / T_5, S_{13} / T_{10} \end{array} \right] \end{array} \right)
\end{aligned}$$

Combining the above results gives the following as the six consequent formulas:

$$\begin{aligned}
&\left\{ \begin{array}{l} \left[\text{ptn}(\{ \text{ptn}(\{S_6, S_7\}), \text{ptn}(\{S_{10}, S_{11}\}), \text{ptn}(\{S_{14}, S_{15}\}) \}) \doteq \text{ptn}(\{\{b\}, \{E\}\}), \right. \\ \left. \text{ptn}(\{ \text{ptn}(\{T_6, T_7\}), \text{ptn}(\{T_{11}, T_{12}\}) \}) \doteq \text{ptn}(\{\{a\}, \{D\}\}), \text{ptn}(\emptyset) \doteq \text{ptn}(\emptyset) \right] \end{array} \right\}; \Gamma_2; V; \Sigma_2 \\
&\left\{ \begin{array}{l} \left[\text{ptn}(\{ \text{ptn}(\{S_6, S_7\}), \text{ptn}(\{S_{10}, S_{11}\}), \text{ptn}(\{S_{14}, S_{15}\}) \}) \doteq \text{ptn}(\{\{E\}\}), \right. \\ \left. \text{ptn}(\{ \text{ptn}(\{T_6, T_7\}), \text{ptn}(\{T_{11}, T_{12}\}) \}) \doteq \text{ptn}(\{\{D\}\}), \right. \\ \left. \text{ptn}(\{\{a\}\}) \doteq \text{ptn}(\{\{b\}\}) \right] \end{array} \right\}; \Gamma_2; V; \Sigma_2 \\
&\left\{ \begin{array}{l} \left[\text{ptn}(\{ \text{ptn}(\{S_6, S_7\}), \text{ptn}(\{S_{10}, S_{11}\}), \text{ptn}(\{S_{14}, S_{15}\}) \}) \doteq \text{ptn}(\{\{E\}\}), \right. \\ \left. \text{ptn}(\{ \text{ptn}(\{T_6, T_7\}), \text{ptn}(\{T_{11}, T_{12}\}) \}) \doteq \text{ptn}(\{\{a\}\}), \right. \\ \left. \text{ptn}(\{\{D\}\}) \doteq \text{ptn}(\{\{b\}\}) \right] \end{array} \right\}; \Gamma_2; V; \Sigma_2 \\
&\left\{ \begin{array}{l} \left[\text{ptn}(\{ \text{ptn}(\{S_6, S_7\}), \text{ptn}(\{S_{10}, S_{11}\}), \text{ptn}(\{S_{14}, S_{15}\}) \}) \doteq \text{ptn}(\{\{b\}\}), \right. \\ \left. \text{ptn}(\{ \text{ptn}(\{T_6, T_7\}), \text{ptn}(\{T_{11}, T_{12}\}) \}) \doteq \text{ptn}(\{\{D\}\}), \right. \\ \left. \text{ptn}(\{\{a\}\}) \doteq \text{ptn}(\{\{E\}\}) \right] \end{array} \right\}; \Gamma_2; V; \Sigma_2
\end{aligned}$$

$$\begin{array}{l}
\left[\begin{array}{l} \text{ptn}\left(\left\{\text{ptn}\left(\{S_6, S_7\}\right), \text{ptn}\left(\{S_{10}, S_{11}\}\right), \text{ptn}\left(\{S_{14}, S_{15}\}\right)\right\}\right) \doteq \text{ptn}\left(\{\{b\}\}\right), \\ \text{ptn}\left(\left\{\text{ptn}\left(\{T_6, T_7\}\right), \text{ptn}\left(\{T_{11}, T_{12}\}\right)\right\}\right) \doteq \text{ptn}\left(\{\{a\}\}\right), \\ \text{ptn}\left(\{\{D\}\}\right) \doteq \text{ptn}\left(\{\{E\}\}\right) \end{array} \right] \\
\left. \begin{array}{l} \text{ptn}\left(\left\{\text{ptn}\left(\{S_6, S_7\}\right), \text{ptn}\left(\{S_{10}, S_{11}\}\right), \text{ptn}\left(\{S_{14}, S_{15}\}\right)\right\}\right) \doteq \text{ptn}(\emptyset), \\ \text{ptn}\left(\left\{\text{ptn}\left(\{T_6, T_7\}\right), \text{ptn}\left(\{T_{11}, T_{12}\}\right)\right\}\right) \doteq \text{ptn}(\emptyset), \\ \text{ptn}\left(\{\{a\}, \{D\}\}\right) \doteq \text{ptn}\left(\{\{b\}, \{E\}\}\right) \end{array} \right] \end{array} \Bigg\}; \Gamma_2; V; \Sigma_2$$

The ground unification rules can be applied to the first and second of these resulting unification formulas. For the first unification formula, the first ground unification rule removes the ‘ $\text{ptn}(\emptyset) \doteq \text{ptn}(\emptyset)$ ’ unification. For the second unification formula, the second ground unification rule removes the entire formula because the ‘ $\text{ptn}(\{\{a\}\}) \doteq \text{ptn}(\{\{b\}\})$ ’ unification fails.

Unify Unconnected Hollow Partitioned Set Parts. The last step in preparing the partitioned sets for unification is to create individual unifications for the parts of the hollowed partitioned sets that are not used in connecting to other hollowed partitioned sets. These are the parts “left over” after the application of the previous rule (Rule 10 “connect hollow partitioned sets”).

$$\frac{\{\text{ptn}(Z) \doteq \text{ptn}(C)\} \cup \Lambda; \Gamma; V; \Sigma}{\left\{ (\tau_Z / \Lambda; \tau_Z / \Gamma; V; \Sigma \circ \tau_Z) \mid \tau_Z \in T_Z \right\}}$$

(Rule.11)

where $\forall t \in Z \exists P (t = \text{ptn}(P) \wedge \forall u \in P (\text{var}(u)))$
 $\wedge (\text{ptn}(Z) \doteq \text{ptn}(C)) \notin \Lambda$

$$\sigma_Z = \{(u/t) \mid \text{ptn}(P) \in Z \wedge (u/t) \in \text{mapping}(P, C)\}$$

$$\text{mapping}(P, C)$$

$$= \begin{cases} \{u/t, u/\emptyset\} \cup \text{mapping}(P - \{u\}, C - \{t\}) & \text{if } \begin{pmatrix} C \neq \emptyset \\ \wedge u = \text{choice}(P) \wedge t = \text{choice}(C) \end{pmatrix} \\ \{u/\emptyset \mid u \in P\} & \text{if } C = \emptyset \end{cases}$$

$$T_Z = \left\{ \tau_Z \mid \begin{array}{l} \tau_Z \subseteq \sigma_Z \\ \wedge \forall t \in C \exists u \left(\begin{array}{l} (u/t) \in \tau_Z \\ \wedge \forall s \in C ((u/s) \in \tau_Z \rightarrow t = s) \\ \wedge \neg \exists v (u \neq v \wedge (v/t) \in \tau_Z) \end{array} \right) \end{array} \right\}$$

This rule relies on two equations and a special function, ‘mapping(P, C)’. The mapping function associates each element of P with an element of C and with the empty set, where no element of P is associated with more than one element of C and every element of C is associated with an element of P . The definition of ‘mapping(P, C)’ is recursive and relies on ‘choice(S)’ to choose an element from ‘ S ’. Using the choice function recursively imposes an order on the elements of P and C . The cardinality of P is guaranteed to be greater than or equal to the cardinality of C by the way in which hollow partitioned sets are constructed.

The ‘ σ_Z ’ term is defined to be all mappings from partitioned set parts in Z to ele-

ments of C . The ‘ T_Z ’ term is defined to be a set of substitutions where each substitution completely maps Z to C and sets to empty parts not used in mapping to C .

We continue the example from the previous rule. Consider the first of the consequent formulas, “simplified” as indicated above by the application of the first of the ground unification rules:

$$\left\{ \begin{array}{l} \text{ptn}\left(\left\{\text{ptn}\left(\{S_6, S_7\}\right), \text{ptn}\left(\{S_{10}, S_{11}\}\right), \text{ptn}\left(\{S_{14}, S_{15}\}\right)\right\}\right) \doteq \text{ptn}\left(\{\{b\}, \{E\}\}\right), \\ \text{ptn}\left(\left\{\text{ptn}\left(\{T_6, T_7\}\right), \text{ptn}\left(\{T_{11}, T_{12}\}\right)\right\}\right) \doteq \text{ptn}\left(\{\{a\}, \{D\}\}\right) \end{array} \right\}; \Gamma_2; V; \Sigma_2$$

The current rule applies to both unifications in this formula. Consider the second unification, ‘ $\text{ptn}\left(\left\{\text{ptn}\left(\{T_6, T_7\}\right), \text{ptn}\left(\{T_{11}, T_{12}\}\right)\right\}\right) \doteq \text{ptn}\left(\{\{a\}, \{D\}\}\right)$ ’.

$$Z = \left\{ \text{ptn}\left(\{T_6, T_7\}\right), \text{ptn}\left(\{T_{11}, T_{12}\}\right) \right\}$$

$$C = \left\{ \{a\}, \{D\} \right\}$$

$$\Lambda = \left\{ \text{ptn}\left(\left\{\text{ptn}\left(\{S_6, S_7\}\right), \text{ptn}\left(\{S_{10}, S_{11}\}\right), \text{ptn}\left(\{S_{14}, S_{15}\}\right)\right\}\right) \doteq \text{ptn}\left(\{\{b\}, \{E\}\}\right) \right\}$$

From these values of Z and C we derive ‘ σ_Z ’:

$$\begin{aligned} \sigma_Z &= \left\{ (u/t) \mid \text{ptn}(P) \in Z \wedge (u/t) \in \text{mapping}(P, C) \right\} \\ &= \left\{ (u/t) \mid \begin{array}{l} \text{ptn}(P) \in \left\{ \text{ptn}\left(\{T_6, T_7\}\right), \text{ptn}\left(\{T_{11}, T_{12}\}\right) \right\} \\ \wedge (u/t) \in \text{mapping}\left(P, \left\{ \{a\}, \{D\} \right\}\right) \end{array} \right\} \\ &= \left\{ \begin{array}{l} T_6 / \{a\}, T_7 / \emptyset, T_{11} / \{D\}, T_{12} / \emptyset, \\ T_{11} / \{a\}, T_{12} / \emptyset, T_{12} / \{D\}, T_{12} / \emptyset \end{array} \right\} \end{aligned}$$

This value of ‘ σ_Z ’ allows us to build ‘ T_Z ’:

$$T_Z = \left\{ \begin{array}{l} \{T_6 / \{a\}, T_7 / \{D\}, T_{11} / \emptyset, T_{12} / \emptyset\}, \\ \{T_6 / \{a\}, T_7 / \emptyset, T_{11} / \emptyset, T_{12} / \{D\}\}, \\ \{T_6 / \emptyset, T_7 / \{D\}, T_{11} / \{b\}, T_{12} / \emptyset\}, \\ \{T_6 / \emptyset, T_7 / \emptyset, T_{11} / \{a\}, T_{12} / \{D\}\} \end{array} \right\}$$

The substitutions in T_Z are each applied to the initial constraints Γ :

$$\begin{aligned}
& \{T_6/\{a\}, T_7/\{D\}, T_{11}/\emptyset, T_{12}/\emptyset\}/\Gamma_2 \\
&= \left\{ \left\{ \text{ptncon} \left[\begin{array}{c} \left(\begin{array}{c} T_3, T_8, S_6, S_7, \end{array} \right) \\ \left\{ \begin{array}{c} T_4, T_9, S_{10}, S_{11}, \\ T_5, T_{10}, S_{14}, S_{15}, \end{array} \right\} \\ \left\{ \begin{array}{c} \{a\}, \{D\} \end{array} \right\} \end{array} \right\} \right\} \\
& \left\{ \text{ptncon} \left[\begin{array}{c} \left(\begin{array}{c} T_3, T_4, T_5, T_6, T_7, \\ T_8, T_9, T_{10}, T_{11}, T_{12}, \end{array} \right) \\ \left\{ \begin{array}{c} \{b\}, \{E\} \end{array} \right\} \end{array} \right\} \right\} \\
&= \left\{ \left\{ \text{ptncon} \left[\begin{array}{c} \left(\begin{array}{c} T_3, T_8, S_6, S_7, \end{array} \right) \\ \left\{ \begin{array}{c} T_4, T_9, S_{10}, S_{11}, \\ T_5, T_{10}, S_{14}, S_{15}, \end{array} \right\} \\ \left\{ \begin{array}{c} \{a\}, \{D\} \end{array} \right\} \end{array} \right\} \right\}, \left\{ \text{ptncon} \left[\begin{array}{c} \left(\begin{array}{c} T_3, T_4, T_5, \{a\}, \{D\}, \end{array} \right) \\ \left\{ \begin{array}{c} T_8, T_9, T_{10}, \\ \{b\}, \{E\} \end{array} \right\} \end{array} \right\} \right\} \\
& \{T_6/\{a\}, T_7/\emptyset, T_{11}/\emptyset, T_{12}/\{D\}\}/\Gamma_2 \\
&= \left\{ \left\{ \text{ptncon} \left[\begin{array}{c} \left(\begin{array}{c} T_3, T_8, S_6, S_7, \end{array} \right) \\ \left\{ \begin{array}{c} T_4, T_9, S_{10}, S_{11}, \\ T_5, T_{10}, S_{14}, S_{15}, \end{array} \right\} \\ \left\{ \begin{array}{c} \{a\}, \{D\} \end{array} \right\} \end{array} \right\} \right\}, \left\{ \text{ptncon} \left[\begin{array}{c} \left(\begin{array}{c} T_3, T_4, T_5, \{a\}, \end{array} \right) \\ \left\{ \begin{array}{c} T_8, T_9, T_{10}, \{D\} \\ \{b\}, \{E\} \end{array} \right\} \end{array} \right\} \right\} \\
& \{T_6/\emptyset, T_7/\{D\}, T_{11}/\{a\}, T_{12}/\emptyset\}/\Gamma_2 \\
&= \left\{ \left\{ \text{ptncon} \left[\begin{array}{c} \left(\begin{array}{c} T_3, T_8, S_6, S_7, \end{array} \right) \\ \left\{ \begin{array}{c} T_4, T_9, S_{10}, S_{11}, \\ T_5, T_{10}, S_{14}, S_{15}, \end{array} \right\} \\ \left\{ \begin{array}{c} \{a\}, \{D\} \end{array} \right\} \end{array} \right\} \right\}, \left\{ \text{ptncon} \left[\begin{array}{c} \left(\begin{array}{c} T_3, T_4, T_5, \{D\}, \end{array} \right) \\ \left\{ \begin{array}{c} T_8, T_9, T_{10}, \{a\} \\ \{b\}, \{E\} \end{array} \right\} \end{array} \right\} \right\} \\
& \{T_6/\emptyset, T_7/\emptyset, T_{11}/\{a\}, T_{12}/\{D\}\}/\Gamma_2 \\
&= \left\{ \left\{ \text{ptncon} \left[\begin{array}{c} \left(\begin{array}{c} T_3, T_8, S_6, S_7, \end{array} \right) \\ \left\{ \begin{array}{c} T_4, T_9, S_{10}, S_{11}, \\ T_5, T_{10}, S_{14}, S_{15}, \end{array} \right\} \\ \left\{ \begin{array}{c} \{a\}, \{D\} \end{array} \right\} \end{array} \right\} \right\}, \left\{ \text{ptncon} \left[\begin{array}{c} \left(\begin{array}{c} T_3, T_4, T_5, \\ T_8, T_9, T_{10}, \{a\}, \{D\}, \end{array} \right) \\ \left\{ \begin{array}{c} \{b\}, \{E\} \end{array} \right\} \end{array} \right\} \right\}
\end{aligned}$$

All of the above results are equivalent. This will always be the case. Let this common partitioning constraint set be ‘ Γ_3 ’.

The substitutions in T_z are applied to the initial substitution Σ :

$\Sigma \sigma \tau_Z$

$$= \left\{ \begin{array}{l} S_1 / \text{ptn}(\{T_3, T_8, S_6, S_7\}), S_2 / \text{ptn}(\{T_4, T_9, S_{10}, S_{11}\}), \\ S_3 / \text{ptn}(\{T_5, T_{10}, S_{14}, S_{15}\}), \\ T_1 / \text{ptn}(\{T_3, T_4, T_5, T_6, T_7\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, T_{11}, T_{12}\}), \\ S_4 / T_3, S_5 / T_8, S_8 / T_4, S_9 / T_9, S_{12} / T_5, S_{13} / T_{10} \end{array} \right\} \sigma \tau_Z$$

To make the presentation more compact, we extract the portion of ‘ Σ ’ that no substitution in ‘ T_Z ’ alters and label it ‘ σ_{common} ’:

$$\sigma_{common} = \left\{ \begin{array}{l} S_1 / \text{ptn}(\{T_3, T_8, S_6, S_7\}), S_2 / \text{ptn}(\{T_4, T_9, S_{10}, S_{11}\}), \\ S_3 / \text{ptn}(\{T_5, T_{10}, S_{14}, S_{15}\}), \\ S_4 / T_3, S_5 / T_8, S_8 / T_4, S_9 / T_9, S_{12} / T_5, S_{13} / T_{10}, \end{array} \right\}$$

The four overall substitutions (one for each τ_Z) are:

$$\begin{aligned} & \left\{ \begin{array}{l} T_1 / \text{ptn}(\{T_3, T_4, T_5, \{a\}, \{D\}\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}\}), \\ T_6 / \{a\}, T_7 / \{D\}, T_{11} / \emptyset, T_{12} / \emptyset \end{array} \right\} \cup \sigma_{common} \\ & \left\{ \begin{array}{l} T_1 / \text{ptn}(\{T_3, T_4, T_5, \{a\}\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, \{D\}\}), \\ T_6 / \{a\}, T_7 / \emptyset, T_{11} / \emptyset, T_{12} / \{D\} \end{array} \right\} \cup \sigma_{common} \\ & \left\{ \begin{array}{l} T_1 / \text{ptn}(\{T_3, T_4, T_5, \{D\}\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, \{b\}\}), \\ T_6 / \emptyset, T_7 / \{D\}, T_{11} / \{b\}, T_{12} / \emptyset \end{array} \right\} \cup \sigma_{common} \\ & \left\{ \begin{array}{l} T_1 / \text{ptn}(\{T_3, T_4, T_5\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, \{a\}, \{D\}\}), \\ T_6 / \emptyset, T_7 / \emptyset, T_{11} / \{a\}, T_{12} / \{D\} \end{array} \right\} \cup \sigma_{common} \end{aligned}$$

Since all of the substitutions in ‘ T_Z ’ bind variables not found in ‘ Λ ’, there are no changes in ‘ Λ ’ from the antecedent to the consequent.

The consequent set of unification formulas this yields is:

$$\begin{aligned}
& \{(\tau_Z / \Lambda; \tau_Z / \Gamma; V; \Sigma \circ \tau_Z) \mid \tau_Z \in T_Z\} \\
&= \left\{ \left(\Lambda; \Gamma_3; V; \left\{ \begin{array}{l} T_1 / \text{ptn}(\{T_3, T_4, T_5, \{a\}, \{D\}\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}\}), \\ T_6 / \{a\}, T_7 / \{D\}, T_{11} / \emptyset, T_{12} / \emptyset \end{array} \right\} \cup \sigma_{\text{common}} \right), \right\} \\
& \left\{ \left(\Lambda; \Gamma_3; V; \left\{ \begin{array}{l} T_1 / \text{ptn}(\{T_3, T_4, T_5, \{a\}\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, \{D\}\}), \\ T_6 / \{a\}, T_7 / \emptyset, T_{11} / \emptyset, T_{12} / \{D\} \end{array} \right\} \cup \sigma_{\text{common}} \right), \right\} \\
& \left\{ \left(\Lambda; \Gamma_3; V; \left\{ \begin{array}{l} T_1 / \text{ptn}(\{T_3, T_4, T_5, \{D\}\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, \{b\}\}), \\ T_6 / \emptyset, T_7 / \{D\}, T_{11} / \{b\}, T_{12} / \emptyset \end{array} \right\} \cup \sigma_{\text{common}} \right), \right\} \\
& \left\{ \left(\Lambda; \Gamma_3; V; \left\{ \begin{array}{l} T_1 / \text{ptn}(\{T_3, T_4, T_5\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}, \{a\}, \{D\}\}), \\ T_6 / \emptyset, T_7 / \emptyset, T_{11} / \{a\}, T_{12} / \{D\} \end{array} \right\} \cup \sigma_{\text{common}} \right), \right\}
\end{aligned}$$

The same processing is applied to each of four formulas in this consequent. Consider the first of these four formulas. The antecedent mapping is:

$$\begin{aligned}
Z &= \{\text{ptn}(\{S_6, S_7\}), \text{ptn}(\{S_{10}, S_{11}\}), \text{ptn}(\{S_{14}, S_{15}\})\} \\
C &= \{\{b\}, \{E\}\} \\
\Lambda &= \emptyset \\
\rho_1 &= \left\{ \begin{array}{l} T_1 / \text{ptn}(\{T_3, T_4, T_5, \{a\}, \{D\}\}), T_2 / \text{ptn}(\{T_8, T_9, T_{10}\}), \\ T_6 / \{a\}, T_7 / \{D\}, T_{11} / \emptyset, T_{12} / \emptyset \end{array} \right\} \\
\Sigma &= \rho_1 \cup \sigma_{\text{common}}
\end{aligned}$$

The substitution mappings are:

$$\sigma_Z = \left\{ \begin{array}{l} S_6 / \{b\}, S_6 / \emptyset, S_7 / \{E\}, S_7 / \emptyset, \\ S_{10} / \{b\}, S_{10} / \emptyset, S_{11} / \{E\}, S_{11} / \emptyset, \\ S_{14} / \{b\}, S_{14} / \emptyset, S_{15} / \{E\}, S_{15} / \emptyset \end{array} \right\}$$

This yields a ‘ T_Z ’ of:

$$T_Z = \left\{ \begin{array}{l} \{S_6/\{b\}, S_7/\{E\}, S_{10}/\emptyset, S_{11}/\emptyset, S_{14}/\emptyset, S_{15}/\emptyset\}, \\ \{S_6/\{b\}, S_7/\emptyset, S_{10}/\emptyset, S_{11}/\{E\}, S_{14}/\emptyset, S_{15}/\emptyset\}, \\ \{S_6/\{b\}, S_7/\emptyset, S_{10}/\emptyset, S_{11}/\emptyset, S_{14}/\emptyset, S_{15}/\{E\}\}, \\ \{S_6/\emptyset, S_7/\{E\}, S_{10}/\{b\}, S_{11}/\emptyset, S_{14}/\emptyset, S_{15}/\emptyset\}, \\ \{S_6/\emptyset, S_7/\emptyset, S_{10}/\{b\}, S_{11}/\{E\}, S_{14}/\emptyset, S_{15}/\emptyset\}, \\ \{S_6/\emptyset, S_7/\emptyset, S_{10}/\{b\}, S_{11}/\emptyset, S_{14}/\emptyset, S_{15}/\{E\}\}, \\ \{S_6/\emptyset, S_7/\{E\}, S_{10}/\emptyset, S_{11}/\emptyset, S_{14}/\{b\}, S_{15}/\emptyset\}, \\ \{S_6/\emptyset, S_7/\emptyset, S_{10}/\emptyset, S_{11}/\{E\}, S_{14}/\{b\}, S_{15}/\emptyset\}, \\ \{S_6/\emptyset, S_7/\emptyset, S_{10}/\emptyset, S_{11}/\emptyset, S_{14}/\{b\}, S_{15}/\{E\}\} \end{array} \right\}$$

These various substitutions yield a common constraint set ‘ Γ_4 ’:

$$\Gamma_4 = \tau_Z / \Gamma_3$$

$$\begin{aligned} &= \left\{ \text{ptncon} \left(\left\{ \begin{array}{l} (T_3, T_8, \{b\}, \{E\}), \\ T_4, T_9, \\ T_5, T_{10}, \\ \{a\}, \{D\} \end{array} \right\} \right), \text{ptncon} \left(\left\{ \begin{array}{l} T_3, T_4, T_5, \\ T_8, T_9, T_{10}, \{a\}, \{D\}, \\ \{b\}, \{E\} \end{array} \right\} \right) \right\} \\ &= \left\{ \text{ptncon} \left(\left\{ \begin{array}{l} T_3, T_4, T_5, T_8, T_9, T_{10}, \\ \{a\}, \{D\}, \{b\}, \{E\} \end{array} \right\} \right) \right\} \end{aligned}$$

In this constraint set, the two separate ‘ptncon’ terms of Γ_3 have collapsed to a single term.

The consequent substitutions, one for each ‘ τ_Z ’ in ‘ T_Z ’, differ only in the bindings of certain ‘ S_i ’ variables. These bindings are only in ‘ σ_{common} ’. Further, ‘ σ_{kernel} ’ is the part of ‘ σ_{common} ’ that is unchanged by all of these ‘ τ_Z ’:

$$\begin{aligned} \sigma_{\text{kernel}} &= \{S_4/T_3, S_5/T_8, S_8/T_4, S_9/T_9, S_{12}/T_5, S_{13}/T_{10}\} \\ \sigma_{\text{common}} &= \left\{ \begin{array}{l} S_1/\text{ptn}(\{T_3, T_8, S_6, S_7\}), S_2/\text{ptn}(\{T_4, T_9, S_{10}, S_{11}\}), \\ S_3/\text{ptn}(\{T_5, T_{10}, S_{14}, S_{15}\}) \end{array} \right\} \cup \sigma_{\text{kernel}} \end{aligned}$$

There are nine overall substitutions, one for each unification formula in the consequent. The non-kernel portions of these are:

$$\begin{aligned}
& \left\{ S_1 / \text{ptn}(\{T_3, T_8, \{b\}, \{E\}\}), S_2 / \text{ptn}(\{T_4, T_9\}), S_3 / \text{ptn}(\{T_5, T_{10}\}) \right\} \\
& \left\{ S_1 / \text{ptn}(\{T_3, T_8, \{b\}\}), S_2 / \text{ptn}(\{T_4, T_9, \{E\}\}), S_3 / \text{ptn}(\{T_5, T_{10}\}) \right\} \\
& \left\{ S_1 / \text{ptn}(\{T_3, T_8, \{b\}\}), S_2 / \text{ptn}(\{T_4, T_9\}), S_3 / \text{ptn}(\{T_5, T_{10}, \{E\}\}) \right\} \\
& \left\{ S_1 / \text{ptn}(\{T_3, T_8, \{E\}\}), S_2 / \text{ptn}(\{T_4, T_9, \{b\}\}), S_3 / \text{ptn}(\{T_5, T_{10}\}) \right\} \\
& \left\{ S_1 / \text{ptn}(\{T_3, T_8\}), S_2 / \text{ptn}(\{T_4, T_9, \{b\}, \{E\}\}), S_3 / \text{ptn}(\{T_5, T_{10}\}) \right\} \\
& \left\{ S_1 / \text{ptn}(\{T_3, T_8\}), S_2 / \text{ptn}(\{T_4, T_9, \{b\}\}), S_3 / \text{ptn}(\{T_5, T_{10}, \{E\}\}) \right\} \\
& \left\{ S_1 / \text{ptn}(\{T_3, T_8, \{E\}\}), S_2 / \text{ptn}(\{T_4, T_9\}), S_3 / \text{ptn}(\{T_5, T_{10}, \{b\}\}) \right\} \\
& \left\{ S_1 / \text{ptn}(\{T_3, T_8\}), S_2 / \text{ptn}(\{T_4, T_9, \{E\}\}), S_3 / \text{ptn}(\{T_5, T_{10}, \{b\}\}) \right\} \\
& \left\{ S_1 / \text{ptn}(\{T_3, T_8\}), S_2 / \text{ptn}(\{T_4, T_9\}), S_3 / \text{ptn}(\{T_5, T_{10}, \{b\}, \{E\}\}) \right\}
\end{aligned}$$

The consequent for this second application of Rule 11 is:

$$\begin{aligned}
& \{(\tau_Z / \Lambda; \tau_Z / \Gamma; V; \Sigma \circ \tau_Z) \mid \tau_Z \in T_Z\} \\
& \left\{ \left(\emptyset; \Gamma_4; V; \left(\begin{array}{l} \left\{ S_1 / \text{ptn}(\{T_3, T_8, \{b\}, \{E\}\}) \right\}, S_2 / \text{ptn}(\{T_4, T_9\}), S_3 / \text{ptn}(\{T_5, T_{10}\}) \right\} \right) \right. \right. \\
& \quad \left. \left. \cup \rho_1 \cup \sigma_{\text{kernel}} \right) \right\}, \\
& \left\{ \left(\emptyset; \Gamma_4; V; \left(\begin{array}{l} \left\{ S_1 / \text{ptn}(\{T_3, T_8, \{b\}\}) \right\}, S_2 / \text{ptn}(\{T_4, T_9, \{E\}\}), S_3 / \text{ptn}(\{T_5, T_{10}\}) \right\} \right) \right. \right. \\
& \quad \left. \left. \cup \rho_1 \cup \sigma_{\text{kernel}} \right) \right\}, \\
& \left\{ \left(\emptyset; \Gamma_4; V; \left(\begin{array}{l} \left\{ S_1 / \text{ptn}(\{T_3, T_8, \{b\}\}) \right\}, S_2 / \text{ptn}(\{T_4, T_9\}), S_3 / \text{ptn}(\{T_5, T_{10}, \{E\}\}) \right\} \right) \right. \right. \\
& \quad \left. \left. \cup \rho_1 \cup \sigma_{\text{kernel}} \right) \right\}, \\
& \left\{ \left(\emptyset; \Gamma_4; V; \left(\begin{array}{l} \left\{ S_1 / \text{ptn}(\{T_3, T_8, \{E\}\}) \right\}, S_2 / \text{ptn}(\{T_4, T_9, \{b\}\}), S_3 / \text{ptn}(\{T_5, T_{10}\}) \right\} \right) \right. \right. \\
& \quad \left. \left. \cup \rho_1 \cup \sigma_{\text{kernel}} \right) \right\}, \\
& = \left\{ \left(\emptyset; \Gamma_4; V; \left(\begin{array}{l} \left\{ S_1 / \text{ptn}(\{T_3, T_8\}) \right\}, S_2 / \text{ptn}(\{T_4, T_9, \{b\}, \{E\}\}), S_3 / \text{ptn}(\{T_5, T_{10}\}) \right\} \right) \right. \right. \\
& \quad \left. \left. \cup \rho_1 \cup \sigma_{\text{kernel}} \right) \right\}, \\
& \left\{ \left(\emptyset; \Gamma_4; V; \left(\begin{array}{l} \left\{ S_1 / \text{ptn}(\{T_3, T_8\}) \right\}, S_2 / \text{ptn}(\{T_4, T_9, \{b\}\}), S_3 / \text{ptn}(\{T_5, T_{10}, \{E\}\}) \right\} \right) \right. \right. \\
& \quad \left. \left. \cup \rho_1 \cup \sigma_{\text{kernel}} \right) \right\}, \\
& \left\{ \left(\emptyset; \Gamma_4; V; \left(\begin{array}{l} \left\{ S_1 / \text{ptn}(\{T_3, T_8, \{E\}\}) \right\}, S_2 / \text{ptn}(\{T_4, T_9\}), S_3 / \text{ptn}(\{T_5, T_{10}, \{b\}\}) \right\} \right) \right. \right. \\
& \quad \left. \left. \cup \rho_1 \cup \sigma_{\text{kernel}} \right) \right\}, \\
& \left\{ \left(\emptyset; \Gamma_4; V; \left(\begin{array}{l} \left\{ S_1 / \text{ptn}(\{T_3, T_8\}) \right\}, S_2 / \text{ptn}(\{T_4, T_9, \{E\}\}), S_3 / \text{ptn}(\{T_5, T_{10}, \{b\}\}) \right\} \right) \right. \right. \\
& \quad \left. \left. \cup \rho_1 \cup \sigma_{\text{kernel}} \right) \right\}, \\
& \left\{ \left(\emptyset; \Gamma_4; V; \left(\begin{array}{l} \left\{ S_1 / \text{ptn}(\{T_3, T_8\}) \right\}, S_2 / \text{ptn}(\{T_4, T_9\}), S_3 / \text{ptn}(\{T_5, T_{10}, \{b\}, \{E\}\}) \right\} \right) \right. \right. \\
& \quad \left. \left. \cup \rho_1 \cup \sigma_{\text{kernel}} \right) \right\}
\end{aligned}
\right\}
\end{aligned}$$

The unification formulas in this result are finished, that is there are no more unifications to process. Thus, the partitioning constraints and substitutions in them are final results.

There are nine such final formulas for each of the four unification formulas of the previous step making 36 final results emanating from the first formula in the consequent of the previous rule. The second formula of the consequent of the previous rule “fails”, as mentioned above. The third formula produces an interestingly different result. The third formula of the consequent of the preceding rule is:

$$\left[\begin{array}{l} \text{ptn}(\{\text{ptn}(\{S_6, S_7\}), \text{ptn}(\{S_{10}, S_{11}\}), \text{ptn}(\{S_{14}, S_{15}\})\}) \doteq \text{ptn}(\{\{E\}\}), \\ \text{ptn}(\{\text{ptn}(\{T_6, T_7\}), \text{ptn}(\{T_{11}, T_{12}\})\}) \doteq \text{ptn}(\{\{a\}\}), \\ \text{ptn}(\{\{D\}\}) \doteq \text{ptn}(\{\{b\}\}) \end{array} \right]; \Gamma_2; V; \Sigma_2$$

The first and second unifications in this formula produce three and two different substitutions, respectively. The third formula is not processed by the current rule (Rule 11). The set of unification formulas that processing the first and second unifications produces is:

$$\left[\begin{array}{l} / \{ \text{ptn}(\{\{D\}\}) \doteq \text{ptn}(\{\{b\}\}) \}; \\ (\rho \cup \sigma \cup \tau) / \Gamma_2; \\ V; \\ \Sigma_2 \circ (\rho \cup \sigma \cup \tau) \end{array} \right] \left[\begin{array}{l} \rho = \{S_7 / \emptyset, S_{11} / \emptyset, S_{15} / \emptyset, T_7 / \emptyset, T_{12} / \emptyset\} \\ \wedge \sigma \in \left\{ \begin{array}{l} \{S_6 / \{E\}, S_{10} / \emptyset, S_{14} / \emptyset\}, \\ \{S_6 / \emptyset, S_{10} / \{E\}, S_{14} / \emptyset\}, \\ \{S_6 / \emptyset, S_{10} / \emptyset, S_{14} / \{E\}\} \end{array} \right\} \\ \wedge \tau \in \left\{ \begin{array}{l} \{T_6 / \{a\}, T_{11} / \emptyset\}, \\ \{T_6 / \emptyset, T_{11} / \{a\}\} \end{array} \right\} \end{array} \right]$$

Processing the fourth formula of the preceding rule with the current rule produces a similar result:

$$\left[\begin{array}{l} / \{ \text{ptn}(\{\{a\}\}) \doteq \text{ptn}(\{\{E\}\}) \}; \\ (\rho \cup \sigma \cup \tau) / \Gamma_2; \\ V; \\ \Sigma_2 \circ (\rho \cup \sigma \cup \tau) \end{array} \right] \left[\begin{array}{l} \rho = \{S_7 / \emptyset, S_{11} / \emptyset, S_{15} / \emptyset, T_7 / \emptyset, T_{12} / \emptyset\} \\ \wedge \sigma \in \left\{ \begin{array}{l} \{S_6 / \{b\}, S_{10} / \emptyset, S_{14} / \emptyset\}, \\ \{S_6 / \emptyset, S_{10} / \{b\}, S_{14} / \emptyset\}, \\ \{S_6 / \emptyset, S_{10} / \emptyset, S_{14} / \{b\}\} \end{array} \right\} \\ \wedge \tau \in \left\{ \begin{array}{l} \{T_6 / \{D\}, T_{11} / \emptyset\}, \\ \{T_6 / \emptyset, T_{11} / \{D\}\} \end{array} \right\} \end{array} \right]$$

The result of processing the fifth formula of the preceding rule's consequent is:

$$\left[\begin{array}{l} / \left\{ \text{ptn}(\{\{D\}\}) \doteq \text{ptn}(\{\{E\}\}) \right\}; \\ (\rho \cup \sigma \cup \tau) / \Gamma_2; \\ V; \\ \Sigma_2 \circ (\rho \cup \sigma \cup \tau) \end{array} \right] \left[\begin{array}{l} \rho = \{S_7 / \emptyset, S_{11} / \emptyset, S_{15} / \emptyset, T_7 / \emptyset, T_{12} / \emptyset\} \\ \wedge \sigma \in \left\{ \begin{array}{l} \{S_6 / \{b\}, S_{10} / \emptyset, S_{14} / \emptyset\}, \\ \{S_6 / \emptyset, S_{10} / \{b\}, S_{14} / \emptyset\}, \\ \{S_6 / \emptyset, S_{10} / \emptyset, S_{14} / \{b\}\} \end{array} \right\} \\ \wedge \tau \in \left\{ \begin{array}{l} \{T_6 / \{a\}, T_{11} / \emptyset\}, \\ \{T_6 / \emptyset, T_{11} / \{a\}\} \end{array} \right\} \end{array} \right]$$

Finally, the result of processing the sixth formula of the preceding rule's consequent is the single unification formula:

$$\left(\left\{ \text{ptn}(\{\{a\}, \{D\}\}) \doteq \text{ptn}(\{\{b\}, \{E\}\}) \right\}; \sigma / \Gamma_2; V; \Sigma_2 \circ \sigma \right)$$

$$\text{where } \sigma = \left\{ \begin{array}{l} \left(\tau = (S_i / \emptyset) \wedge i \in \{6, 7, 10, 11, 14, 15\} \right) \\ \vee \left(\tau = (T_i / \emptyset) \wedge i \in \{6, 7, 11, 12\} \right) \end{array} \right\}$$

At this point we have expanded the original single unification formula into 55 unification formulas, 36 of which are “finished” and the other 19 contain a single unification.

Unify Processed Partitioned Sets. The inference rules which follow process the elements of a partition on the left-hand side of a unification equation. They generally reduce a unification that involves partitioned sets into a unification between an element of each of these partitioned sets and the unification of these partitioned sets minus the “extracted” elements. This reduction relies on the “pairwise disjoint” property of partitioned sets in that if a part of a partitioned set unifies with something, then no other part of that same partitioned set can unify with that same thing.

These rules are applied repeatedly to the same unification of a unification formula to produce several different mappings of that unification formula onto the antecedent of the rule.

Unify Partitioned Set Element of Set-type. These three rules are the possibilities for a part on the left-hand-side that is a singleton set. In Rule 12 and Rule 13, the part on the right-hand-side is itself a partitioned set. In Rule 14, the part on the right-hand-side is a singleton set. The part on the right-hand-side is never a variable: the vari-

ables at this “level” have been removed by the hollowing process.

$$\frac{\left\{ \text{ptn}(\{\{R\}\} \cup P) \doteq \text{ptn}(\{\text{ptn}(\{X\} \cup S)\} \cup Q) \right\} \cup \Lambda; \Gamma; V; \Sigma}{\sigma / \left(\left\{ \text{ptn}(P) \doteq \text{ptn}(\{\text{ptn}(S)\} \cup Q) \right\} \cup \Lambda \right); \sigma / \Gamma; V; \Sigma \circ \sigma}$$

where $\text{var}(X) \wedge \sigma = \{X/R\}$ (Rule.12)

$\wedge \left(\text{ptn}(\{\{R\}\} \cup P) \doteq \text{ptn}(\{\text{ptn}(\{X\} \cup S)\} \cup Q) \right) \notin \Lambda$

$$\frac{\left\{ \text{ptn}(\{\{R\}\} \cup P) \doteq \text{ptn}(\{\text{ptn}(\{\{X\}\} \cup S)\} \cup Q) \right\} \cup \Lambda; \Gamma; V; \Sigma}{\left\{ R \doteq X, \text{ptn}(P) \doteq \text{ptn}(\{\text{ptn}(S)\} \cup Q) \right\} \cup \Lambda; \Gamma; V; \Sigma}$$

where $\left(\text{ptn}(\{\{R\}\} \cup P) \doteq \text{ptn}(\{\text{ptn}(\{\{X\}\} \cup S)\} \cup Q) \right) \notin \Lambda$ (Rule.13)

$$\frac{\left\{ \text{ptn}(\{\{R\}\} \cup P) \doteq \text{ptn}(\{\{X\}\} \cup Q) \right\} \cup \Lambda; \Gamma; V; \Sigma}{\left\{ R \doteq X, \text{ptn}(P) \doteq \text{ptn}(Q) \right\} \cup \Lambda; \Gamma; V; \Sigma}$$

where $\left(\text{ptn}(\{\{R\}\} \cup P) \doteq \text{ptn}(\{\{X\}\} \cup Q) \right) \notin \Lambda$ (Rule.14)

We can continue our example from Rule 11. The first group of unification formulas is not processed any further since it contains no unifications. The second group of unification formulas was:

$$\left[\begin{array}{l} \left(\left\{ \text{ptn}(\{\{D\}\}) \doteq \text{ptn}(\{\{b\}\}) \right\}; \right) \\ \left(\rho \cup \sigma \cup \tau \right) / \Gamma_2; \\ V; \\ \Sigma_2 \circ (\rho \cup \sigma \cup \tau) \end{array} \right] \left[\begin{array}{l} \rho = \{S_7 / \emptyset, S_{11} / \emptyset, S_{15} / \emptyset, T_7 / \emptyset, T_{12} / \emptyset\} \\ \wedge \sigma \in \left\{ \begin{array}{l} \{S_6 / \{E\}, S_{10} / \emptyset, S_{14} / \emptyset\}, \\ \{S_6 / \emptyset, S_{10} / \{E\}, S_{14} / \emptyset\}, \\ \{S_6 / \emptyset, S_{10} / \emptyset, S_{14} / \{E\}\} \end{array} \right\} \\ \wedge \tau \in \left\{ \begin{array}{l} \{T_6 / \{a\}, T_{11} / \emptyset\}, \\ \{T_6 / \emptyset, T_{11} / \{a\}\} \end{array} \right\} \end{array} \right]$$

The unification formulas of this group all have the same unification:

$$\text{ptn}(\{\{D\}\}) \doteq \text{ptn}(\{\{b\}\})$$

This unification can only be mapped onto the third of these rules, Rule 14. This mapping gives:

$$\begin{aligned}
R &= D \\
P &= \emptyset \\
X &= b \\
Q &= \emptyset \\
\rho &= \{S_7 / \emptyset, S_{11} / \emptyset, S_{15} / \emptyset, T_7 / \emptyset, T_{12} / \emptyset\} \\
\Theta_S &\in \left\{ \begin{aligned} &\{S_6 / \{E\}, S_{10} / \emptyset, S_{14} / \emptyset\}, \\ &\{S_6 / \emptyset, S_{10} / \{E\}, S_{14} / \emptyset\}, \\ &\{S_6 / \emptyset, S_{10} / \emptyset, S_{14} / \{E\}\} \end{aligned} \right\} \\
\Theta_T &\in \left\{ \begin{aligned} &\{T_6 / \{a\}, T_{11} / \emptyset\}, \\ &\{T_6 / \emptyset, T_{11} / \{a\}\} \end{aligned} \right\} \\
\sigma_j &\in \Theta_S \\
\tau_k &\in \Theta_T \\
\Gamma_{jk} &= (\rho \cup \sigma_j \cup \tau_k) / \Gamma_2 \\
\Sigma_{jk} &= \Sigma_2 \circ (\rho \cup \sigma_j \cup \tau_k) \\
\Lambda &= \emptyset
\end{aligned}$$

The j and k indices are used to select some particular pair of Θ_S and Θ_T substitutions. There are six such pairings possible. This mapping produces the consequent unification formula:

$$\{D \doteq b, \text{ptn}(\emptyset) \doteq \text{ptn}(\emptyset)\}; \Gamma_{jk}; V; \Sigma_{jk}$$

This is simplified by the ground unification rules to:

$$\{D \doteq b\}; \Gamma_{jk}; V; \Sigma_{jk}$$

A similar process is used in unifying the fourth and fifth groups to create:

$$\{a \doteq E\}; \Gamma_{lm}; V; \Sigma_{lm}$$

$$\{D \doteq E\}; \Gamma_{pq}; V; \Sigma_{pq}$$

The sixth group gives two unification formulas, which are the same except for their sets of unifications. These two sets of unifications are:

$$\{a \doteq b, D \doteq E\}$$

$$\{a \doteq E, D \doteq b\}$$

The unification formula containing the first of these is eliminated by the ground unification rules since a does not unify with b . Thus the single unification formula remain-

ing from Rule 14 processing the sixth result of the previous example discussion is:

$$\{a \doteq E, D \doteq b\}; \Gamma_3; V; \Sigma_3$$

where Γ_3 and Σ_3 are the mappings from the sixth result onto the antecedent of Rule 14.

Unify Partitioned Set Element of Partitioned-set-type. These rules handle the case where there is a part on the left-hand-side that is a partitioned set. The antecedents of both rules select a part of a partitioned set that is a part of the left-hand-side of a unification. In both antecedents there are two new unifications created to replace the selected unification: one unification is for the selected part and the other unification is for the original unification minus the selected part. The two rules differ in how they set up these two new unifications: either the selected part is set to the empty set and the remainder of the left-hand-side is set to the original right-hand-side (Rule 15) or the selected part is set to some part of the right-hand-side of the unification and the remainder of the left-hand-side is set to the remainder of the right-hand-side (Rule 16).

$$\frac{\left\{ \text{ptn}\left(\left\{ \text{ptn}\left(\left\{ R \right\} \cup Y\right)\right\} \cup P\right) \doteq \text{ptn}(Q) \right\} \cup \Lambda; \Gamma; V; \Sigma}{\left\{ R \doteq \emptyset, \text{ptn}\left(\left\{ \text{ptn}(Y) \right\} \cup P\right) \doteq \text{ptn}(Q) \right\} \cup \Lambda; \Gamma; V; \Sigma} \quad (\text{Rule.15})$$

where $\left(\text{ptn}\left(\left\{ \text{ptn}\left(\left\{ R \right\} \cup Y\right)\right\} \cup P\right) \doteq \text{ptn}(Q) \right) \notin \Lambda$

$$\frac{\left\{ \text{ptn}\left(\left\{ \text{ptn}\left(\left\{ R \right\} \cup Y\right)\right\} \cup P\right) \doteq \text{ptn}(\{Z\} \cup Q) \right\} \cup \Lambda; \Gamma; V; \Sigma}{\left\{ R \doteq Z, \text{ptn}\left(\left\{ \text{ptn}(Y) \right\} \cup P\right) \doteq \text{ptn}(Q) \right\} \cup \Lambda; \Gamma; V; \Sigma} \quad (\text{Rule.16})$$

where $\left(\text{ptn}\left(\left\{ \text{ptn}\left(\left\{ R \right\} \cup Y\right)\right\} \cup P\right) \doteq \text{ptn}(\{Z\} \cup Q) \right) \notin \Lambda$

Delete Empty Partition Element There are three rules for handling empty sets. Partitioned sets on the left-hand-side that are unified with empty sets must have all of their parts unified with empty sets (Rule 17). Empty sets that appear on the left-hand-side are shifted to the right-hand-side (Rule 18), which makes the previous rule applicable in some circumstances. Finally, a partitioned set with no parts is replaced by the empty set (Rule 19).

$$\frac{\{\text{ptn}(Q) \doteq \emptyset\} \cup \Lambda; \Gamma; V; \Sigma}{\{q \doteq \emptyset \mid q \in Q\} \cup \Lambda; \Gamma; V; \Sigma} \text{ where } \{\text{ptn}(Q) \doteq \emptyset\} \notin \Lambda \quad (\text{Rule.17})$$

$$\frac{\{\emptyset \doteq T\} \cup \Lambda; \Gamma; V; \Sigma}{\{T \doteq \emptyset\} \cup \Lambda; \Gamma; V; \Sigma} \text{ where } (\emptyset \doteq T) \notin \Lambda \quad (\text{Rule.18})$$

$$\frac{\{\text{ptn}(\emptyset) \doteq T\} \cup \Lambda; \Gamma; V; \Sigma}{\{\emptyset \doteq T\} \cup \Lambda; \Gamma; V; \Sigma} \text{ where } (\text{ptn}(\emptyset) \doteq T) \notin \Lambda \quad (\text{Rule.19})$$

$$\frac{\{T = \text{ptn}(\emptyset)\} \cup \Lambda; \Gamma; V; \Sigma}{\{T \doteq \emptyset\} \cup \Lambda; \Gamma; V; \Sigma} \text{ where } (T = \text{ptn}(\emptyset)) \notin \Lambda$$

Variable Unification. These rules are concerned with the unification of variables.

Occur Check: This rule verifies that the variable doesn't occur in the term with which it is being bound. It recognizes a kind of unification that *fails*. Since this rule specifies a failure, it uses the extended form of the rule that is used in Rule 6 and Rule 7. The entire unification formula to which the failing unification belongs is removed from the set of unification formulas to be solved.

$$\frac{\{\{X \doteq T\} \cup \Lambda; \Gamma; V; \Sigma\} \cup \Pi}{\Pi} \quad (\text{Rule.20})$$

where $T \neq X \wedge \text{var}(X) \wedge \text{occurs_in}(X, T)$

Instantiate. This rule “instantiates” a variable unification—applies the binding of the variable to the rest of the unification problem.

$$\frac{\{X \doteq T\} \cup \Lambda; \Gamma; V; \Sigma}{\sigma / \Lambda; \sigma / \Gamma; V; \Sigma \circ \sigma} \quad (\text{Rule.21})$$

where $\text{var}(X) \wedge \neg \text{occurs_in}(X, T) \wedge \sigma = \{X / T\}$
 $\wedge (X \doteq T) \notin \Lambda$

Commute. This rule transposes unifications so that all unifications involving a vari-

able and a non-variable have the variable on the left-hand-side.

$$\frac{\{T \doteq X\} \cup \Lambda; \Gamma; V; \Sigma}{\{X \doteq T\} \cup \Lambda; \Gamma; V; \Sigma} \quad (\text{Rule.22})$$

where $\neg \text{var}(T) \wedge \text{var}(X) \wedge (T \doteq X) \notin \Lambda$

An Example Unification.

The “union” example presented earlier serves as an example for showing the application of the unification algorithm outlined above. The initial unifications are:

$$\{\{a,b\} \doteq \text{ptn}(\{X,Y\}), \{b,c\} \doteq \text{ptn}(\{Y,Z\}), R \doteq \text{ptn}(\{X,Y,Z\})\} \quad (\text{Union.1})$$

The initial variables are X , Y , Z , and R . The constraint and substitution sets are empty. In most of the discussion below, we will suppress references to the variable, constraint, and substitution sets to simplify the presentation.

The first step is to atomize these equations applying Rule 1 and Rule 2 to produce:

$$\left\{ \left\{ \left\{ \text{ptn}(\{\{a\}, \{b\}\}) \doteq \text{ptn}(\{X,Y\}), \right\} \right\} \right\} \quad (\text{Union.2})$$

$$\left\{ \left\{ \left\{ \text{ptn}(\{\{b\}, \{c\}\}) \doteq \text{ptn}(\{Y,Z\}), \right\} \right\} \right\}$$

$$\left\{ \left\{ R \doteq \text{ptn}(\{X,Y,Z\}) \right\} \right\}$$

Rule 8 has no effect on this example, since there are no “identical” elements. Rule 9 can be applied to each of the three unifications in Union 2. We show the application of Rule 9 in detail for the first unification. First, we distinguish the variable parts of partitionings from the nonvariable parts:

$$\begin{aligned} \text{ptn}(\{\{a\}, \{b\}\}) &\doteq \text{ptn}(\{X,Y\}) \\ \hat{X} &= \emptyset \\ \hat{Y} &= \{X,Y\} \\ \hat{A} &= \{\{a\}, \{b\}\} \\ \hat{B} &= \emptyset \end{aligned}$$

Next, we set up the “skeletons” for the variable parts:

$$\begin{aligned}
\sigma_x &= hpss(\hat{X}, \hat{B}, \hat{Y}, \emptyset) = \emptyset \\
\sigma_y &= hpss(\hat{Y}, \hat{A}, \hat{X}, \emptyset) = \left\{ (X/\text{ptn}(\{X_1, X_2\})), (Y/\text{ptn}(\{Y_1, Y_2\})) \right\} \\
\hat{X}' &= ps(\sigma_x) = \emptyset \\
\hat{Y}' &= ps(\sigma_y) = \left\{ \text{ptn}(\{X_1, X_2\}), \text{ptn}(\{Y_1, Y_2\}) \right\}
\end{aligned}$$

Finally, we combine these elements to create the unification which replaces the unification with which we started:

$$\text{ptn}(\{\{a\}, \{b\}\}) \doteq \text{ptn}(\{\text{ptn}(\{X_1, X_2\}), \text{ptn}(\{Y_1, Y_2\})\})$$

Doing this replacement in Union 2 produces a new form of the “union” set of unifications:

$$\left[\begin{array}{l} \left[\left[\text{ptn}(\{\{a\}, \{b\}\}) \doteq \text{ptn}(\{\text{ptn}(\{X_1, X_2\}), \text{ptn}(\{Y_1, Y_2\})\}) \right], \right] \\ \left[\left[\text{ptn}(\{\{b\}, \{c\}\}) \doteq \text{ptn}(\{\text{ptn}(\{Y_1, Y_2\}), Z\}) \right], \right] \\ \left[R \doteq \text{ptn}(\{\text{ptn}(\{X_1, X_2\}), \text{ptn}(\{Y_1, Y_2\}), Z\}) \right] \end{array} \right]$$

Repeating this process for the other original unification yields:

$$\left[\begin{array}{l} \left[\left[\text{ptn}(\{\{a\}, \{b\}\}) \doteq \text{ptn}(\{\text{ptn}(\{X_1, X_2\}), \text{ptn}(\{Y_1, Y_2\})\}) \right], \right] \\ \left[\left[\text{ptn}(\{\{b\}, \{c\}\}) \doteq \text{ptn}(\{\text{ptn}(\{Y_1, Y_2\}), \text{ptn}(\{Z_1, Z_2\})\}) \right], \right] \\ \left[R \doteq \text{ptn}(\{\text{ptn}(\{X_1, X_2\}), \text{ptn}(\{Y_1, Y_2\}), \text{ptn}(\{Z_1, Z_2\})\}) \right] \\ \left[\mathbf{L}; \left\{ (X/\text{ptn}(\{X_1, X_2\})), (Y/\text{ptn}(\{Y_1, Y_2\})), (Z/\text{ptn}(\{Z_1, Z_2\})) \right\} \right] \end{array} \right] \quad (\text{Union.3})$$

In Union 3 we give an elided form of the full unification formula, with just the first term (the unification set) and the last term (the substitutions).

Rule 10 does not apply in this example. If it were to apply, it would apply here. But, there are no unifications with partitionings containing variable parts (“hollow partitionings”) on both sides of the unification and thus there are no hollow parti-

tioned sets to “connect”.

The next phase of the unification process applies Rule 11 to unify the variable parts which were just introduced. We show this process for one of the unifications. First, we distinguish the variable parts from the nonvariable parts, as before:

$$\begin{aligned}
 \text{ptn}(\{\{a\}, \{b\}\}) &\doteq \text{ptn}(\{\text{ptn}(\{X_1, X_2\}), \text{ptn}(\{Y_1, Y_2\})\}) \\
 \hat{X}'' &= \emptyset \\
 \hat{Y}'' &= \{\text{ptn}(\{X_1, X_2\}), \text{ptn}(\{Y_1, Y_2\})\} \\
 \hat{A} &= \{\{a\}, \{b\}\} \\
 \hat{B} &= \emptyset
 \end{aligned}$$

First, we build the sets of possible substitutions:

$$T_Z = \left\{ \begin{array}{l} \left\{ \begin{array}{l} X_1 / \{a\}, X_2 / \{b\}, \\ Y_1 / \emptyset, Y_2 / \emptyset \end{array} \right\}, \\ \left\{ \begin{array}{l} X_1 / \{a\}, X_2 / \emptyset, \\ Y_1 / \emptyset, Y_2 / \{b\} \end{array} \right\}, \\ \left\{ \begin{array}{l} X_1 / \emptyset, X_2 / \{b\}, \\ Y_1 / \{a\}, Y_2 / \emptyset \end{array} \right\}, \\ \left\{ \begin{array}{l} X_1 / \emptyset, X_2 / \emptyset, \\ Y_1 / \{a\}, Y_2 / \{b\} \end{array} \right\} \end{array} \right\}$$

These sets are shown here in a “compressed” form. There are other sets of possible unifications, but they are not logically distinct from those shown here - they can ultimately lead to different answers. This compression relies on the variables on the left hand sides of these unifications being variables introduced in this unification process which therefore cannot be referenced in any other part of the larger “query” of which this unification might be a part.

The modified form of the “union” unification set, with empty set simplification (Rule 17) applied, is:

$$\begin{aligned}
& \left(\left[\begin{array}{l} \text{ptn}(\{\{b\}, \{c\}\}) \doteq \text{ptn}(\{\text{ptn}(\{Z_1, Z_2\})\}) \\ R \doteq \text{ptn}(\{\text{ptn}(\{\{a\}, \{b\}\}), \text{ptn}(\{Z_1, Z_2\})\}) \end{array} \right]; \right. \\
& \quad \left. \mathbf{L}; \left[\begin{array}{l} X/\text{ptn}(\{\{a\}, \{b\}\}), Y/\emptyset, \\ Z/\text{ptn}(\{Z_1, Z_2\}), \\ X_1/\{a\}, X_2/\{b\}, Y_1/\emptyset, Y_2/\emptyset \end{array} \right] \right) \\
& \left(\left[\begin{array}{l} \text{ptn}(\{\{b\}, \{c\}\}) \doteq \text{ptn}(\{\text{ptn}(\{\{b\}\}), \text{ptn}(\{Z_1, Z_2\})\}) \\ R \doteq \text{ptn}(\{\text{ptn}(\{\{a\}\}), \text{ptn}(\{\{b\}\}), \text{ptn}(\{Z_1, Z_2\})\}) \end{array} \right]; \right. \\
& \quad \left. \mathbf{L}; \left[\begin{array}{l} X/\text{ptn}(\{\{a\}\}), Y/\text{ptn}(\{\{b\}\}), Z/\text{ptn}(\{Z_1, Z_2\}), \\ X_1 \doteq \{a\}, X_2 \doteq \emptyset, Y_1 \doteq \emptyset, Y_2 \doteq \{b\} \end{array} \right] \right) \\
& \left(\left[\begin{array}{l} \text{ptn}(\{\{b\}, \{c\}\}) \doteq \text{ptn}(\{\text{ptn}(\{\{a\}\}), \text{ptn}(\{Z_1, Z_2\})\}) \\ R \doteq \text{ptn}(\{\text{ptn}(\{\{b\}\}), \text{ptn}(\{\{a\}\}), \text{ptn}(\{Z_1, Z_2\})\}) \end{array} \right]; \right. \\
& \quad \left. \mathbf{L}; \left[\begin{array}{l} X/\text{ptn}(\{\{b\}\}), Y/\text{ptn}(\{\{a\}\}), Z/\text{ptn}(\{Z_1, Z_2\}), \\ X_1 \doteq \emptyset, X_2 \doteq \{b\}, Y_1 \doteq \{a\}, Y_2 \doteq \emptyset \end{array} \right] \right) \\
& \left(\left[\begin{array}{l} \text{ptn}(\{\{b\}, \{c\}\}) \doteq \text{ptn}(\{\text{ptn}(\{\{a\}, \{b\}\}), \text{ptn}(\{Z_1, Z_2\})\}) \\ R \doteq \text{ptn}(\{\text{ptn}(\{\{a\}, \{b\}\}), \text{ptn}(\{Z_1, Z_2\})\}) \end{array} \right]; \right. \\
& \quad \left. \mathbf{L}; \left[\begin{array}{l} X/\emptyset, Y/\text{ptn}(\{\{a\}, \{b\}\}), Z/\text{ptn}(\{Z_1, Z_2\}), \\ X_1 \doteq \emptyset, X_2 \doteq \emptyset, Y_1 \doteq \{a\}, Y_2 \doteq \{b\}, \end{array} \right] \right)
\end{aligned}$$

(Union.4)

This is the first point in the unification process where we see multiple results. This result is further simplified by replacing partitioning parts which are *ground* partitioned sets by their elements (a kind of “flattening”):

$$\begin{array}{l}
\left(\left[\begin{array}{l} \text{ptn}(\{\{b\}, \{c\}\}) \doteq \text{ptn}(\{\text{ptn}(\{Z_1, Z_2\})\}) \\ R \doteq \text{ptn}(\{\{a\}, \{b\}, \text{ptn}(\{Z_1, Z_2\})\}) \end{array} \right] ; \right. \\
\left. \mathbf{L}; \left[\begin{array}{l} X/\text{ptn}(\{\{a\}, \{b\}\}), Y/\emptyset, \\ Z/\text{ptn}(\{Z_1, Z_2\}), \\ X_1/\{a\}, X_2/\{b\}, Y_1/\emptyset, Y_2/\emptyset \end{array} \right] \right) \\
\left(\left[\begin{array}{l} \text{ptn}(\{\{b\}, \{c\}\}) \doteq \text{ptn}(\{\{b\}, \text{ptn}(\{Z_1, Z_2\})\}) \\ R \doteq \text{ptn}(\{\{a\}, \{b\}, \text{ptn}(\{Z_1, Z_2\})\}) \end{array} \right] ; \right. \\
\left. \mathbf{L}; \left[\begin{array}{l} X/\text{ptn}(\{\{a\}\}), Y/\text{ptn}(\{\{b\}\}), Z/\text{ptn}(\{Z_1, Z_2\}), \\ X_1 \doteq \{a\}, X_2 \doteq \emptyset, Y_1 \doteq \emptyset, Y_2 \doteq \{b\} \end{array} \right] \right) \\
\left(\left[\begin{array}{l} \text{ptn}(\{\{b\}, \{c\}\}) \doteq \text{ptn}(\{\{a\}, \text{ptn}(\{Z_1, Z_2\})\}) \\ R \doteq \text{ptn}(\{\{a\}, \{b\}, \text{ptn}(\{Z_1, Z_2\})\}) \end{array} \right] ; \right. \\
\left. \mathbf{L}; \left[\begin{array}{l} X/\text{ptn}(\{\{b\}\}), Y/\text{ptn}(\{\{a\}\}), Z/\text{ptn}(\{Z_1, Z_2\}), \\ X_1 \doteq \emptyset, X_2 \doteq \{b\}, Y_1 \doteq \{a\}, Y_2 \doteq \emptyset \end{array} \right] \right) \\
\left(\left[\begin{array}{l} \text{ptn}(\{\{b\}, \{c\}\}) \doteq \text{ptn}(\{\{a\}, \{b\}, \text{ptn}(\{Z_1, Z_2\})\}) \\ R \doteq \text{ptn}(\{\{a\}, \{b\}, \text{ptn}(\{Z_1, Z_2\})\}) \end{array} \right] ; \right. \\
\left. \mathbf{L}; \left[\begin{array}{l} X/\emptyset, Y/\text{ptn}(\{\{a\}, \{b\}\}), Z/\text{ptn}(\{Z_1, Z_2\}), \\ X_1 \doteq \emptyset, X_2 \doteq \emptyset, Y_1 \doteq \{a\}, Y_2 \doteq \{b\}, \end{array} \right] \right)
\end{array} \quad (\text{Union.5})$$

The unification process proceeds with each of the four alternatives in the above unification set. The third and fourth alternatives fail directly, due to the partitioning/partitioning unification. In both of these alternatives this unification requires that a set consisting of only b and c be unified with a set containing a and possibly other things. This unification must fail.

This leaves the first two alternatives to explore. The first alternative produces one result (considering both ways of mapping Z_1 and Z_2 to $\{b\}$ and $\{c\}$ as producing essentially the same result):

$$\left\{ R \doteq \text{ptn}\left(\left\{\left\{a\right\},\left\{b\right\},\left\{b\right\},\left\{c\right\}\right\}\right); \mathbf{L}; \left\{ \begin{array}{l} X/\{a,b\}, Y/\emptyset, Z/\{b,c\}, \\ X_1/\{a\}, X_2/\{b\}, \\ Y_1/\emptyset, Y_2/\emptyset, \\ Z_1/\{b\}, Z_2/\{c\}, \end{array} \right\} \right\}$$

invalid

This result fails due to the partitioning constraint implied by the last unification equation. This presents a partitioned set of $\text{ptn}(\{\{a\}, \{b\}, \{b\}, \{c\}\})$. This is invalid due to two of the parts of the partitioning having a nonempty intersection, b .

This leaves only the second alternative of Union 5 as a possible solution of the original *union* unification set. It contains a single partitioned set/partitioned set unification equation. This is processed in the same fashion as was done above, and the result is simplified:

$$\left\{ \emptyset; \mathbf{L}; \left\{ \begin{array}{l} X/\{a\}, Y/\{b\}, Z/\{c\}, \\ X_1/\{a\}, X_2/\emptyset, Y_1/\emptyset, Y_2/\{b\}, \\ Z_1/\emptyset, Z_2/\{c\}, R/\{a,b,c\} \end{array} \right\} \right\} \quad (\text{Union.6})$$

This completes the unification process for the original unification set, the interesting substitutions being $X/\{a\}, Y/\{b\}, Z/\{c\}, R/\{a,b,c\}$.

Completeness and Soundness of Atomization.

The atomization algorithm is valid if it satisfies the atomization definition given above. We repeat this definition here:

$$AS = \text{atomization}(S, C)$$

$$\begin{aligned} & \left(\begin{array}{l} \left(\langle \text{ptn}(R), \gamma \rangle \in AS \right) \\ \forall R, \gamma \left(\rightarrow \left(\forall p \left(p \in R \rightarrow \left(\text{var}(p) \vee \text{cardinality}(p) = 1 \right) \right) \right) \right) \end{array} \right) \\ \rightarrow & \left(\begin{array}{l} \wedge \forall R, \gamma \left(\left(\langle \text{ptn}(R), \gamma \rangle \in AS \right) \right. \\ \left. \rightarrow \left(\text{valid_partitioning}(\text{ptn}(R)) \wedge UR = \gamma / S \right) \right) \\ \left(\text{valid_constraints}(s / C) \right) \\ \wedge \forall s \left(\rightarrow \exists R, \gamma \left(\left(\langle \text{ptn}(R), \gamma \rangle \in AS \right) \right. \right. \\ \left. \left. \wedge \text{valid_partitioning}(s / \text{ptn}(R)) \wedge U(s / R) = (s \circ \gamma) / S \right) \right) \end{array} \right) \end{aligned}$$

As described earlier, the second proposition of this definition is *soundness* and the third proposition is *completeness*. In our description of the atomization algorithm we noted that it produces a sequence of sets of atomization formulas, and that this sequence converges on a set of atomization formulas all of which have empty left-hand-sides.

Soundness of atomization. We claim that if the soundness proposition holds for each step of the sequence, then the final set of atomization formulas is a sound transformation of the initial set. To prove that each step is sound, we need to show that each application of a rule is sound.

The rules have the form:

$$\frac{S \Rightarrow \text{ptn}(T); \gamma}{\{U_1 \Rightarrow \text{ptn}(V_1); \mu_1, \mathbf{L}, U_n \Rightarrow \text{ptn}(V_n); \mu_n\}}$$

where $n \geq 1$. The elements of the original set, R , are found in the union of the sets on the two sides of the arrow. For statement $S \Rightarrow \text{ptn}(T)$, the original set is $R = \mathcal{U}(\{S\} \cup T)$. Also, the sets on the two sides of the arrow are disjoint $S \cap (\mathcal{U}T) = \emptyset$. The rules are valid if they preserve the elements of the original set after applying the current unifiers, the resulting statements are valid (i.e. maintain the disjointness of the two sides of the arrow), the unifier set is valid, and they find all distinct partitionings. The unifier set is valid if no variable is unified with two different terms.

Consider Rule 2:

$$\frac{S \Rightarrow \text{ptn}(T); \gamma}{\{\emptyset \Rightarrow \text{ptn}(\{\{U\} | U \in S\} \cup T); \gamma\}}$$

if $\neg \exists X, Y, \sigma \left(\begin{array}{l} X, Y \in S \wedge \sigma \in \text{unifiers}(X, Y) \\ \wedge \text{valid_partitioning}(\sigma / T) \end{array} \right)$

This rule is the simpler of the two atomization rules. First, we replace the ‘ $\text{ptn}(T)$ ’ in the antecedent of the rule by its definition, so that the rule antecedent is expressed in terms of standard set-theoretic propositions. This expansion is done by replacing the ‘ $\text{ptn}(T)$ ’ by a variable (‘ A ’), defining the value of this variable

(‘ $A = UT$ ’), and adding a proposition describing the pairwise disjointness constraint implicit in the use of ‘ $\text{ptn}(T)$ ’ (‘ $\forall p, q \in T (p \cap q = \emptyset)$ ’).

Reducing the antecedent:

$$S \Rightarrow \text{ptn}(T) \Leftrightarrow \left(\begin{array}{l} S \Rightarrow A, \\ A = UT \wedge \forall p, q \in T (p \cap q = \emptyset) \end{array} \right)$$

$$\text{let } R = S \cup A = S \cup UT = U(\{S\} \cup T)$$

R is the total set represented by both sides of the antecedent formula.

Reducing the consequent:

$$\emptyset \Rightarrow \text{ptn}(\{\{U\} | U \in S\} \cup T) \Leftrightarrow \left(\begin{array}{l} \emptyset \Rightarrow V, \\ V = U(\{\{U\} | U \in S\} \cup T) \\ \wedge \forall p, q \in (\{\{U\} | U \in S\} \cup T) (p \cap q = \emptyset) \end{array} \right)$$

Simplifying the total set represented by the antecedent, V :

$$V = U(\{\{U\} | U \in S\} \cup T) = U(\{S\} \cup T)$$

The total set for the statements is preserved between the antecedent and the consequent ($R = V$).

The remaining question is whether the partitioning constraints introduced by the consequent limit the possible substitutions more than the antecedent limits them. This is not a problem due to the constraint on the application of the rule:

$$\neg \exists X, Y, \sigma \left(\begin{array}{l} X, Y \in S \wedge \sigma \in \text{unifiers}(X, Y) \\ \wedge \text{valid_partitioning}(\sigma / T) \end{array} \right)$$

The elements of the consequent’s new partitioning constraint are all of the singleton sets from S . This constraint can only fail if they fail to be pairwise disjoint, i.e. if two of them intersect. For two singleton sets to intersect, their single members must be equal. It is impossible for any of these singleton sets to be equal, even after applying an arbitrary valid-for-the-antecedent substitution, because the condition on the rule forbids that any two elements of S be unifiable. Therefore, the consequent does not limit the unification possibilities beyond whatever limitations were already present in the antecedent. Further, no partitioning constraints are removed, so none of the antecedent’s constraints are eased, either. Thus, Rule 2 is sound.

Consider Rule 1:

$$\begin{array}{c}
\frac{\{X, Y\} \cup S \Rightarrow \text{ptn}(T); \gamma}{\left\{ \{Y\} \cup S \Rightarrow \text{ptn}(\{\{X\}\} \cup T); \gamma \right\} \cup R} \\
\text{if } R \neq \emptyset \\
\wedge R = \left\{ P \left| \begin{array}{l} P = [\{Y\} \cup \sigma / S \Rightarrow \text{ptn}(\sigma / T); \sigma \circ \gamma] \\ \wedge \text{unify}(X, Y, \sigma) \wedge \text{valid_partitioning}(\sigma / T) \end{array} \right. \right\} \\
\text{(where } \sigma \text{ is a set of bindings)}
\end{array}$$

First, we expand the antecedent:

$$\begin{aligned}
& \{X, Y\} \cup S \Rightarrow \text{ptn}(T) \\
& \Leftrightarrow \{X, Y\} \cup S \Rightarrow A, A = \bigcup T \wedge \forall p, q \in T (p \cap q = \emptyset) \\
& R = \bigcup (\{\{X, Y\} \cup S\} \cup T)
\end{aligned}$$

There are two parts to the analysis of the consequent, the “initial” formula and the ‘ R ’ construct:

$$\left\{ \{Y\} \cup S \Rightarrow \text{ptn}(\{\{X\}\} \cup T); \gamma \right\} \cup R$$

The consequent expansion of the initial formula:

$$\begin{aligned}
& \{Y\} \cup S \Rightarrow \text{ptn}(\{\{X\}\} \cup T) \\
& \Leftrightarrow \{Y\} \cup S \Rightarrow A, A = \bigcup (\{\{X\}\} \cup T) \wedge \forall p, q \in (\{\{X\}\} \cup T) (p \cap q = \emptyset) \\
& \text{Let } \Psi = \bigcup (\{\{X, Y\} \cup S\} \cup T) \\
& \Psi'_1 = \bigcup (\{\{Y\} \cup S\} \cup (\{\{X\}\} \cup T)) = \bigcup ((\{\{X\}\} \cup \{\{Y\} \cup S\}) \cup T) \\
& = \bigcup (\{\{X\} \cup \{Y\} \cup S\} \cup T) = \bigcup (\{\{X, Y\} \cup S\} \cup T) = \Psi
\end{aligned}$$

We set Ψ'_1 to be the “total set” represented by the initial formula of the consequent. The extended equality shows it to be the same as the total set for the antecedent.

Next we analyze the a prototypical element of the R construct. First, we develop an expression for the “total set” of this element, Ψ'_2 :

$$\begin{aligned}
& \sigma / (\{Y\} \cup S) \Rightarrow \text{ptn}(\sigma / T) \\
& \Leftrightarrow \sigma / (\{Y\} \cup S) \Rightarrow A, A = \bigcup (\sigma / T) \wedge \forall p, q \in (\sigma / T) (p \cap q = \emptyset) \\
& \Psi'_2 = \bigcup (\{\sigma / (\{Y\} \cup S)\} \cup (\sigma / T))
\end{aligned}$$

The “prototype” substitution applied to the set of X and Y collapses that set to just the common substituted term:

$$\sigma \in \text{unifiers}(X, Y) \rightarrow ((\sigma/X) = Y) \rightarrow \sigma/\{X, Y\} = \sigma/\{Y\}$$

Now we can verify that applying the combined substitution to the total set of the antecedent produces the same set as the total set of the prototypical element of R . Since the antecedent is presumed to have had the antecedent substitution applied to it,

$$(\sigma \circ \gamma)/\Psi = \sigma/\Psi$$

Using the simplified form of the substitution applied to the total set for the antecedent, we show that this is equal to the total set for the prototypical element of R , derived above:

$$\begin{aligned} \sigma/\Psi &= U\left(\left\{\left\{\sigma/(\{X, Y\} \cup S)\right\}\right\} \cup (\sigma/T)\right) \\ &= U\left(\left\{\left\{\sigma/(\{Y\} \cup S)\right\}\right\} \cup (\sigma/T)\right) = \Psi'_2 \\ \therefore \Psi'_2 &= (\sigma \cup (\sigma/\gamma))/\Psi \end{aligned}$$

This shows that the total set is preserved by all of the atomization formulas produced by this rule. The new atomization formulas preserve disjointness. For the initial new formula, the unifier set is valid since it is unchanged. For the other new formulas, the unifier sets are valid since X was an unbound variable when the antecedent was applied, and thus must not have been previously unified.

These two rules have been shown to be sound; the atomizations they produce are equivalent to the original partitioned set (under some substitution).

Completeness of atomization. To establish that the completeness proposition holds we need to prove that all possible distinct bound versions of the original partitioned set have corresponding atomized versions. To prove this we need the detailed definitions of the atomization functions. The basic recursive relationship defining the atomization sequence was given above as:

$$\Phi_{i+1} = U\left\{\text{atomize_step}(\phi) \mid \phi \in \Phi_i\right\}$$

The `atomize_set`, `atomize_step`, `apply`, and `strict_apply` functions are given below:

$$\text{atomization}(S) = \left\{ \langle R, \gamma \rangle \mid (\emptyset \Rightarrow R; \gamma) \in \text{atomize_step}(\{(S \Rightarrow \emptyset; \emptyset)\}) \right\}$$

$$\begin{aligned} \text{atomize_step}(S) = \\ \text{if } (\forall F \in S (\text{strict_apply}(\text{rule1}, F) = \emptyset \wedge \text{strict_apply}(\text{rule2}, F) = \emptyset)) \\ \text{then } S; \\ \text{else } \text{atomize_step}(\cup \{ \text{apply}(\text{rule1}, F) \cup \text{apply}(\text{rule2}, F) \mid F \in S \}). \end{aligned}$$

The $\text{atomization}(S)$ function returns the pairs of atomized partitioned sets and their associated substitutions that are the result of the atomization process. The $\text{apply}(\text{Rule}, F)$ function returns a set of atomization formulas as the result of applying rule Rule to formula F:

$$\begin{aligned} \text{apply}(\text{Rule}, F) = \\ \text{if } \text{strict_apply}(\text{Rule}, F) = \emptyset \\ \text{then } \{F\}; \\ \text{else } \text{strict_apply}(\text{Rule}, F). \end{aligned}$$

This function is a simple wrapper for the $\text{strict_apply}(\text{Rule}, F)$ function. If the strict_apply function returns an empty set (no applications are possible), the apply returns a singleton set of the given formula.

The $\text{strict_apply}(\text{Rule}, F)$ function returns a set of atomization formulas that are the result of applying rule Rule to formula F. If no applications are possible, then it returns an empty set. There are two forms of this function, one for each of the two atomization rules. Both of these forms return the union of the set of all possible consequent atomization formula sets for all possible mappings of the antecedent onto the given formula that satisfy that rule's constraints.

$$\begin{aligned} & \text{strict_apply}(\text{rule1}, L \Rightarrow \text{ptn}(T); \gamma) \\ &= \cup \left\{ \text{Cons} \left[\begin{array}{l} \{X, Y\} \subseteq L \wedge X \neq Y \wedge S = L - \{X, Y\} \wedge R \neq \emptyset \\ \wedge R = \left\{ P \mid P = [\{Y\} \cup \sigma / S \Rightarrow \text{ptn}(\sigma / T); \sigma \cup (\sigma / \gamma)] \right. \right. \\ \left. \left. \wedge \sigma = \text{unifier}(X, Y) \wedge \text{valid_partitioning}(\sigma / T) \right\} \right] \right\} \\ & \text{Cons} = \left\{ \{Y\} \cup S \Rightarrow \text{ptn}(\{\{X\}\} \cup T); \gamma \right\} \cup R \end{aligned}$$

$$\begin{aligned}
& \text{strict_apply}(\text{rule2}, S \Rightarrow \text{ptn}(T); \gamma) \\
&= \left\{ \left(\emptyset \Rightarrow \text{ptn}(\{\{U\} \mid U \in S\} \cup T); \gamma \right) \mid \neg \exists X, Y, \sigma \left(\begin{array}{l} X, Y \in S \wedge \sigma \in \text{unifiers}(X, Y) \\ \wedge \text{valid_partitioning}(\sigma / T) \end{array} \right) \right\}
\end{aligned}$$

The `atomize_step` function applies itself recursively to the set of formulas. At each step of the recursion it exhaustively expands the set of formulas by applying Rule 1 and Rule 2, replacing each expanded statement by its expansion. The recursion terminates when every statement has an empty left-hand side. Each application of a rule reduces the cardinality of the left-hand-side set. If a left-hand-side set is non-empty, then either Rule 1 or Rule 2 applies to it. Thus, the recursion processes every statement fully and it must terminate.

The atomization function is complete if for any A which is a valid atomization of S there exists an $\langle F, g \rangle$ in $\text{atomization}(S)$ such that there is a substitution h such that $(h \circ g) / F = A$. Consider the set G which is the union of the parts of partitioned set F . Elements of G are in S or are the result of unifying two or more elements of S (by the unification in Rule 1). Rule 2 puts those elements which cannot be unified into the result, and Rule 1 puts atomizations for all possible unifications of two elements and the atomization for not unifying those two elements into the result (i.e. a partitioned set with these two elements in different parts and thus required to not unify together). Repeated applications of Rule 1 finds atomizations for all possible unifications of two or more elements. Thus, an F that can be unified with any possible A will be constructed by $\text{atomization}(S)$.

Implementation.

The implementation of the partitioned set unification algorithm by the *SPARCL Interp Unify* module is discussed in chapter 6. In the worst case the algorithm's essential complexity is exponential: given a partitioned set of two variable parts unifying with a set of n elements, there are 2^n possible subsets of elements and each subset can unify with either of the two parts (the remainder unifying with the other part).

The implemented unification algorithm is substantially more complex than the one outlined in this chapter. There are many special cases of unifications which are recognized in order to speed the unification process. Also, there is a direct representation of N-tuples which is incorporated into the unification process (an N-tuple being a special form of a set), and various special cases introduced to handle N-tuples efficiently.

The handling of the constraints is somewhat complex in that the constraints are simplified periodically. The inference procedure of which the unification algorithm is part trims irrelevant constraints when possible. Irrelevant constraints are those that constrain variables that are not present in the literals to be solved (and thus ones that cannot be unified by subsequent processing of the inference engine). This analysis of irrelevance requires keeping track of which variables are used in which set of or multi-set of meta-predicate scope; a variable is irrelevant outside of its scope. Also, the "hollow" partitioned sets for variable parts of partitioned sets may introduce variables that prove to be irrelevant. Trimming these irrelevant variables proved to be an important storage and performance efficiency because the constraint collection was constantly growing over the course of solving a query. For some queries, the internal limit on the number of variables that can be defined in MACPROLOG32 (the language in which SPARCL is implemented) was reached, preventing the query from being solved. Even when the query solution didn't create too many variables, the large constraint collections slowed down execution in that they required a lot of time to check.

Discussion

We have presented a formalization of the partitioned set unification algorithm, an example of its application, a proof of its soundness, and comments on its implementation. We also proved the generality of atomized partitioned sets as a basic representa-

tion of collections of terms. The unusual aspects of this formalization, compared with “traditional” unification, include the atomization process, the use of the atomized partitioned set as the canonical form of collections of terms (instead of strict sets or union sets), and the introduction and “linking” of hollow partitioned sets. The use of atomized partitioned sets as the basic form simplifies the formalization, analysis, and implementation of the unification algorithm. The hollowing process dramatically improves the performance of the implementation in certain common unification situations.

In the implementation section we pointed out that there are several performance-enhancing complications of the implementation of the unification algorithm compared with the formalization. These involve special cases for common unification situations and careful handling of the constraints. We introduced the idea of irrelevant variables in constraint specifications and pointed out that trimming the constraint collection to eliminate irrelevant variables was a crucial performance enhancement in SPARCL. The constraint handling is an area of SPARCL that can use a great deal more work. We currently use a very simple approach to determining the validity of a collection of constraints: a collection of constraint partitioned sets is acceptable if all of the ground partitioned sets are valid. A ground partitioned set is valid if its parts are pairwise disjoint. Ground partition sets are removed from the constraint partitioned set collection. A more sophisticated analysis would recognize some collections of nonground partitioned sets that could never be simultaneously satisfied, thus failing the validity check of the entire collection. This would trim searches of some unprofitable proof tree branches much earlier, thus speeding up SPARCL.

Another unification implementation improvement we would like to explore is the dynamic specialization of the unification algorithm, perhaps for each clause head. This is part of the basic strategy used by most compilers of logic programming languages. A more general approach is partial evaluation of SPARCL with respect to a particular clause, predicate, or collection of predicates. This partial evaluation would produce a “compiled” form of the SPARCL interpreter specialized for a particular SPARCL program. Such a compilation might include unification specialization.

The formalization is not as concise as we believe it can be. We hope to investigate finding a shorter presentation. Having a formal definition of the partitioned set unification algorithm is an essential element of producing an analysis of the procedural semantics of SPARCL. Other elements that we have not yet addressed include an

equality theory (similar to Clark's Equality Theory, perhaps) that includes sets or an adaptation of the standard Herbrand interpretation that includes finite sets. These are necessary for the model-theoretic connection between a declarative semantics of SPARCL and a procedural semantics of SPARCL, in the manner of the standard approach to logic programming semantics epitomized by John Lloyd's *Foundations of Logic Programming* [Lloyd 1987b].

- Hanus 1995 “Analysis of Residuating Logic Programs” by Michael Hanus in *The Journal of Logic Programming*, vol. 24, n. 3, September 1995.
- Jayaraman&Nair 1988 “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. Also found in *Proceedings International Conference on the Fifth Generation Computer Systems*, Tokyo, 1984.
- Lloyd 1987b *Foundations of Logic Programming* by J. W. Lloyd. Springer-Verlag, 2nd edition, 1987.

Chapter 5

Three-Dimensional Representation of SPARCL

Three-dimensional representation of SPARCL has several motivations. In general, three-dimensional scenes are something people are very adept at understanding. Thus, a three-dimensional representation of a program could be more understandable in ways that take advantage of this natural ability of people. A more specific motivation stems from the “line-crossing” problem discussed in section 4 (“Visual Programming Design Elements”) of chapter 3. Most of the work in this thesis focusses on the 2D representation because we were able to complete the design and implementation of the 2D-based interactive development environment (IDE), but a complete 3D-based IDE was not produced. However, substantial progress was made on the use of a 3D representation. In this chapter we present our approach to three-dimensional representation in SPARCL: the 3D representation in general of SPARCL, our approach to rendering 3D models, the automated layout of 3D models of the abstract representation of a SPARCL program, and a detailed presentation of the modelling of hyperedges. The discussion of the modelling of hyperedges extends material originally presented in [Spratt&Ambler 1994].

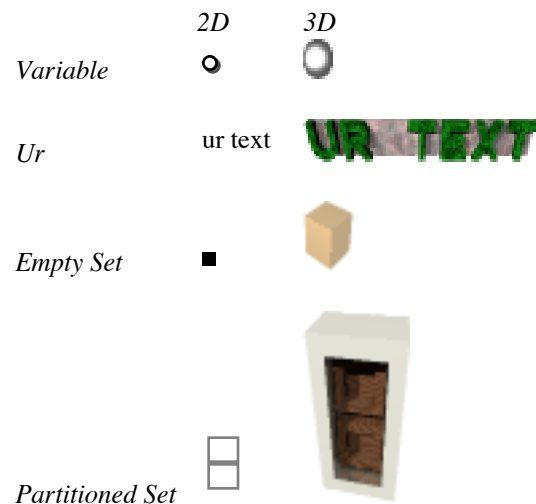


Figure 5. 1: 2D and 3D versions of SPARCL terms.

General 3D representation of SPARCL.

The 3D designs of most of the concrete representations of the display objects in the full, abstract visual representation of SPARCL are simple extensions of their 2D designs. Two-dimensional and three-dimensional versions of various SPARCL terms are shown in Figure 5. 1 and Figure 5. 2. If a display object is represented by a circle in

2D (i.e. a variable), then it is represented by a sphere in 3D (as shown by the *variable* in Figure 5. 1). Text display objects (clause name, literal name, ur constants) are represented in 3D by extruded text; the characters in a text string become 3D objects. If a display object is represented by a rectangle in 2D, then it is represented by a box in 3D (as shown by the *partitioned set* term in Figure 5. 1).

The representation of an N-tuple in 3D differs from

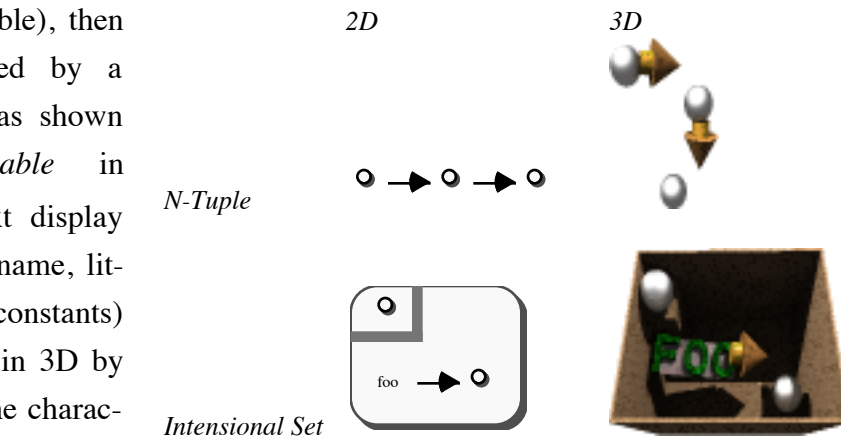


Figure 5. 2: 2D and 3D versions of *N-tuple* and *intensional set* terms.

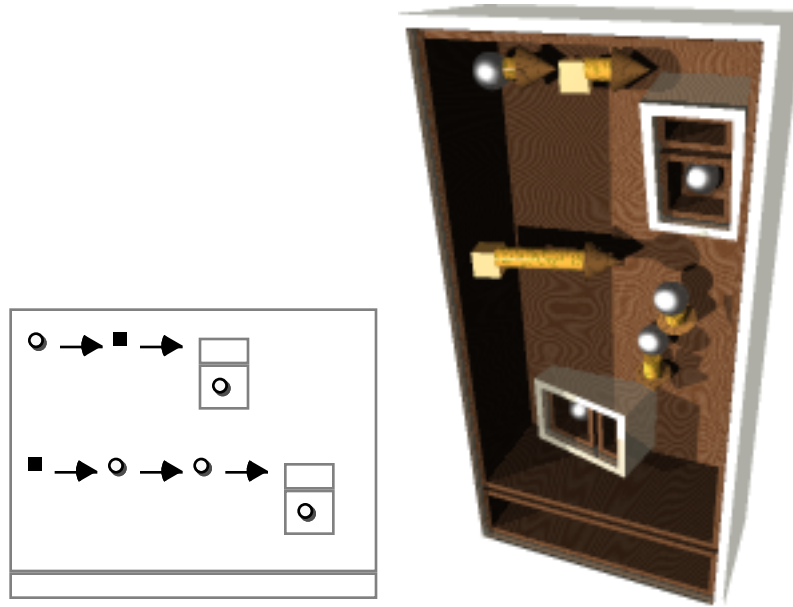


Figure 5. 3: 2D and 3D representations of a partitioned set containing two parts, one of which contains two N-tuples.

that the 3D representation is built around a spiral. This spiral is seen in the example 3-tuple shown in Figure 5. 2. We discuss this in more detail below. Abstract containment between two terms is represented by concrete containment between the projection of the models of the terms to the X-Y plane. Two- and three-dimensional examples of the representation of containment is shown in Figure 5. 3. This shows a partitioned set containing two parts, with one part containing two N-tuples. Each of these N-tuples has as its “final” element a partitioned set containing two parts, where one of these

parts contains a variable.

Four views of a 3D representation of the ‘Union’/3 predicate are shown in Figure 5. 4. These views are rendered by POV-RAY. These scenes are anti-aliased, shaded, and shadowed renderings.

The hyperedges (a) are “biased” to the right so that they won’t be stacked up in this view. Each argument is outlined by a thin rectangle. This clause has a set of two parts in the first and second arguments and a set of three parts in the third argument. In the three-part partitioned set, there is an empty space in

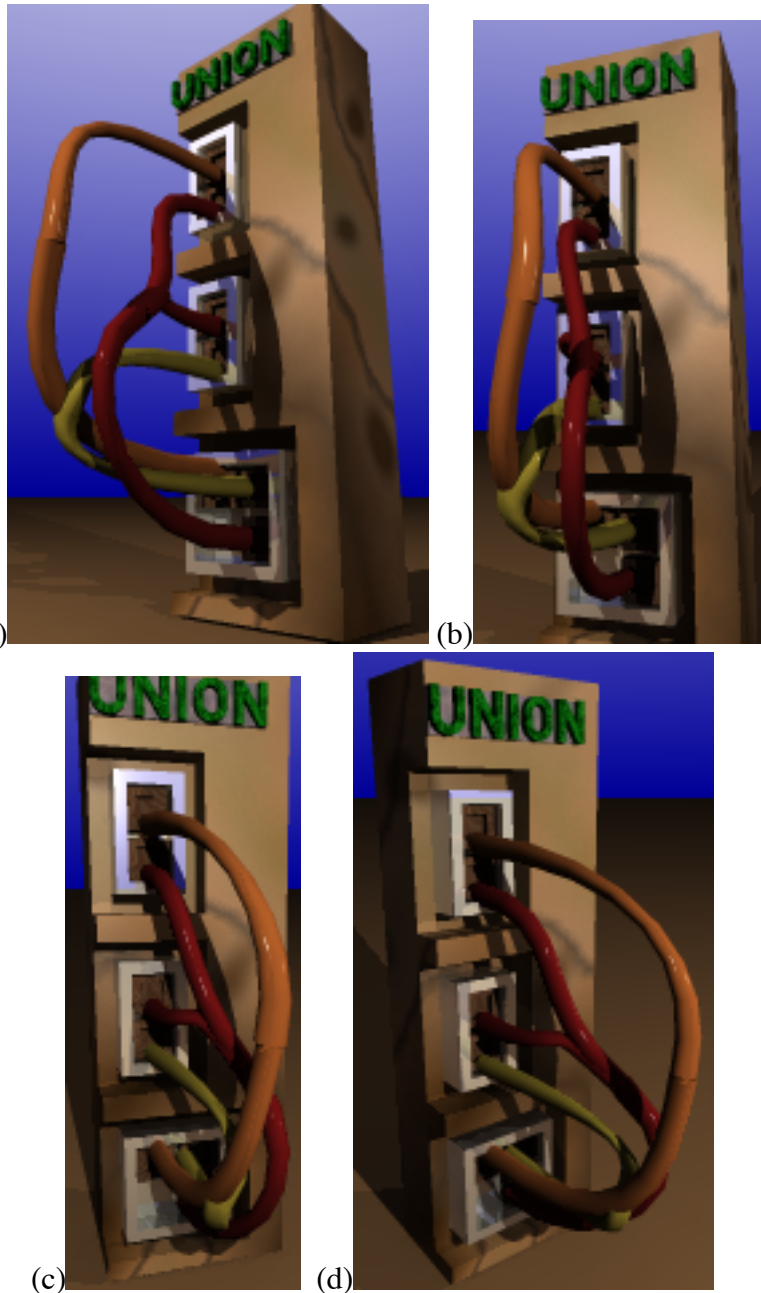
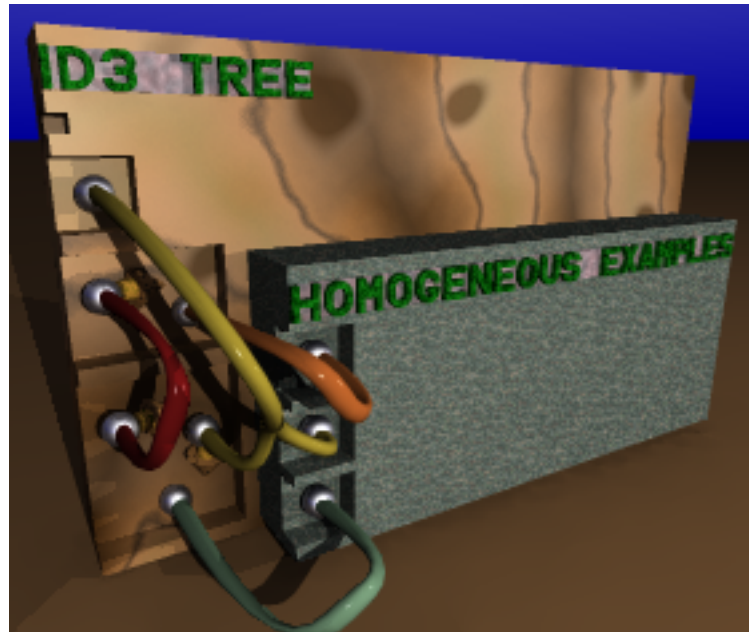


Figure 5. 4: Four views of a 3D representation of the definition of the ‘Union’/3 predicate.

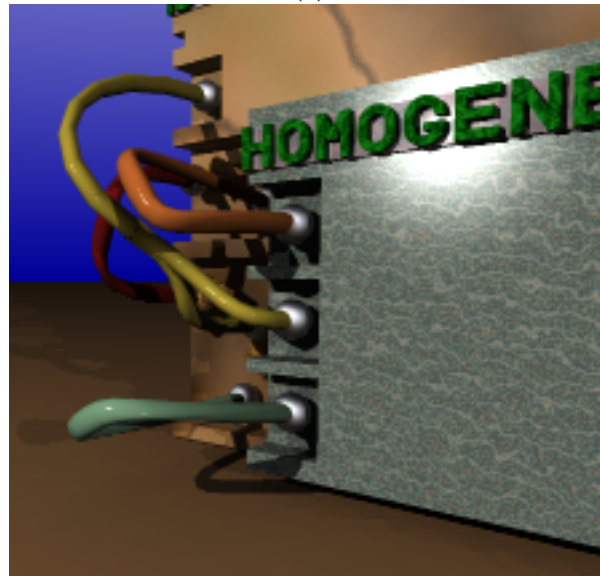
the set (on the lower left of the set) where a fourth part would go. View (c) gives a better idea of the shapes of the parts. They are boxes with open tops (as are the sets). The simple lighting model used by this renderer doesn’t show these part boxes well in the front view (a).

Two views of a 3D representation of the “homogeneous” clause for the ‘ID3

Tree’/4 predicate are shown in Figure 5. 5. The ID3 implementation of which this is a part is presented in chapter 7. (This clause is called “homogeneous” because it uses a ‘Homogeneous Examples’/3 literal as opposed to the other clause for ‘ID3 Tree’/4 shown in Figure 5. 13 and Figure 5. 14 that uses a ‘Heterogeneous Examples’/2 literal.) This clause shows a literal in its body (where the ‘Union’/3 clause in Figure 5. 4 has an empty body), two N-tuples, and several variables. The two N-tuples are a 2-tuple (ordered pair) in the third argument of the clause and a 3-tuple (ordered triple) in the fourth argument the



(a)



(b)

Figure 5. 5: Views (a) and (b) of a 3D concrete representation of the “homogeneous” clause from the ‘ID3 Tree’/4 predicate definition.

clause. The literal name has a thin, light-colored back plate that helps the name stand out by making it higher contrast. This is useful for since the literal “plate” is fairly dark and does not contrast as strongly with the literal name.

Figure 5. 6 was rendered using the Apple Macintosh QuickDraw3D Simple

Viewer. There are various rendering problems with this view; aliasing (i.e. jagged lines that should be smooth), “angular” instead of smoothly curving hyperedges, no shadows, and very simple surface texture modelling (although more complex texture mapping is possible with this renderer). Making these simplifications allows for much faster rendering, so that changes in views are immediate with very little (or no) apparent time for making the changes when running on an Apple Power Macintosh 8500/120. Comparing this with

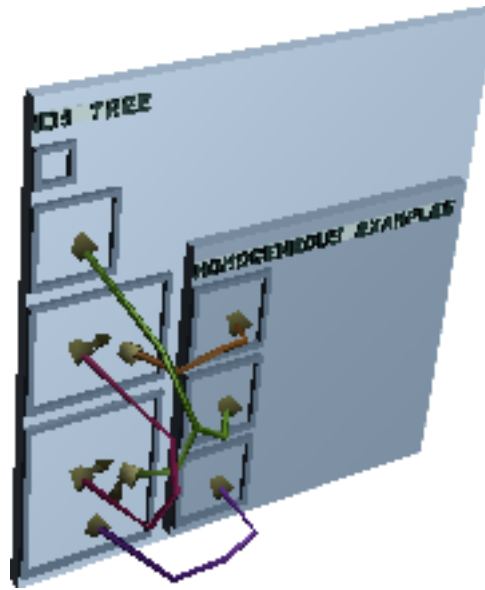


Figure 5. 6: View of the QuickDraw 3D concrete representation of the ‘ID3 Tree’/4 predicate’s “homogeneous” clause.

Figure 5. 5, one can see the advantages of the anti-aliasing in the overall increased legibility and the shadowing in better visual definition of the geometry of the model. The drawback is that the POV-RAY scenes each required about 15 minutes to render on the same machine that rendered the scenes in Figure 5. 5 almost instantaneously.

The representation of text for QD3D POV-RAY is simplified by using only upper case text, although SPARCL uses both cases and is actually case-sensitive. Making the text both upper and lower case is conceptually easy to fix, but time-consuming. The problem is that POV-RAY comes with a library of upper-case only character models. We adapted these models for use in QD3D (which comes with no character modelling). To represent lower-case characters, we will need to create models for them. A less tedious solution is available for QD3D: there is a fairly simple method for generating character models from text strings using the Apple Macintosh QuickDraw GX system. This system is not yet well integrated into the Macintosh and presents us with other problems, however. Eventually, this is probably the approach we will take. Thus, it’s not worth building our own hand-crafted lower-case representations.

The implementation of the 3D representation is incomplete in several respects: certain elements of SPARCL are not representable (term tables, fact tables, and comments), there is no representation for a group of clauses in the same “scene”, all text is represented as upper-case (as discussed above) and the 3D representation is not

interactive—the user cannot directly edit the display (although she can inspect the representation by arbitrary changes of viewpoint). Comments present some interesting possibilities. We have more places to put them in a 3D representation than in a 2D one. The argument comments can be placed on the side of the clause box, making them only visible to certain “oblique” views of the clause. The clause comment can be placed on another side (such as the top) of the clause box, or on the back. There could be two kinds of clause comments: summary comments and full comments. The summary comments could be placed on the top side, thus easily viewed by an appropriate oblique view, which can include viewing the argument side and the top side. The full comments can be placed on the back side of the clause box, keeping it out of the way for general viewing of the clause. We expect to explore some of these possibilities, as well as developing table representations and making the 3D representation interactively editable, in future work on SPARCL.

Rendering

A 3D concrete representation is a rendering of a scene. The scene describes some geometry and associates texture information with the surfaces of this geometry. The rendering uses a lighting model of the scene (positioning and kinds of the lights) and a viewing model (a “camera” pointed at some point in the scene) to produce a 2D image for display. Thus, there are many variables to control in defining a 3D representation, many more variables than are needed to define a 2D representation. The rendering aspect of the representation can be (and probably should be) controlled by the viewer, although the viewer should be given reasonable default behavior by the rendering process. The interested reader should see [Foley et al. 1990] for a detailed discussion of 3D rendering.

There are many different techniques for rendering a scene. Choosing a rendering technique is one of the major challenges in providing a 3D representation. These techniques vary greatly in their speed, in the quality of the image produced, in the kinds of geometry, texturing, and lighting they handle. There is no best technique for all rendering applications; there sometimes isn’t even an obviously best technique for a particular application. The highest quality rendering is called “photo realistic” - it looks just like a photograph of a real scene. This quality of rendering is still unobtainable for scenes in general; but, for scenes which only use certain kinds of

geometries and certain kinds of texturing, photo-realism is achievable.

The choice of rendering technique is dependent on performance requirements, desired geometries, desired texturing, and desired lighting effects (shades or shadows, diffuse and/or specular, reflection, refraction). The performance requirements need to take into account whether the viewer has a movable point of view, or if a single point of view on a scene is sufficient. Generally, applications which render images provide a choice of techniques which differ primarily in their performance. This allows the user to view many rough drafts of a scene quickly, then to very slowly generate a high-quality final version.

This choice of rendering techniques is further complicated by the availability of specialized hardware to speed up certain techniques. Thus, one's choice of technique is also dependent on the hardware platform for the programming system.

Ray tracing and radiosity are two of the most realistic techniques [Foley et al. 1990]. These have different strengths and they are difficult to combine. Ray tracing is good at shiny and transparent surfaces - reflection and refraction. Radiosity is good at diffuse lighting effects. Radiosity has the additional advantage of allowing one to calculate the lighting effects independent of the observer's point of view. Thus, it lends itself to making many different views of the same scene. Radiosity is difficult to apply to general models (i.e. Constructive Solid Geometry (CSG) models, and ones with curved surfaces other than a cylinder), while ray tracing works well with CSG and a variety of curved surfaces (any surface described by a quadratic implicit equation - e.g. cylinders, spheres, and cones). Ray tracing is generally easier to implement than radiosity. Ray tracing is currently the photo-realistic rendering technique most widely used.

Other, less realistic techniques include z-buffer and spanning scan-lines [Foley et al. 1990]. The z-buffer technique is very popular because it is very fast. This is frequently the technique which specialized graphics hardware supports. The spanning scan-lines technique is less fast, but is a little more realistic. Both of these techniques are widely used.

Our implementation of the 3D representation allows the user to choose between three systems for rendering the 3D model of a SPARCL clause: *POV-Ray*, a freeware omni-platform ray tracing and radiosity package with a very powerful modeling language; *OpenInventor* [Wernecke 1994], a multi-platform, general 3D system originated by Silicon Graphics Inc. with a variety of rendering possibilities; and

QuickDraw3D [Apple Computer, Inc. 1995], a multi-platform, general 3D system originated by Apple Computer, Inc. that provides wireframe and simple shading rendering. These three different systems are supported due to the development history of SPARCL. POV-Ray was the only one of the three available when we began work on the 3D representation of SPARCL, so naturally it was adopted as the rendering system. Later OpenInventor became available to the authors (on a Silicon Graphics ONYX machine). This provided a great deal more flexibility, including the opportunity of interacting with the representation, so we extended our rendering system choices to include it. Finally, QD3D became available on the Apple Macintosh. Since this rendering system is also very flexible and the rest of the implementation of SPARCL is on the Macintosh, we extended our rendering system choices again to include it.

Automated layout of SPARCL in three dimensions.

The 3D representation of SPARCL is particularly challenging because we fully automate the modelling and layout of the concrete representation. The 3D concrete representation is derived from the partial abstract visual representation (which is in turn derived from the full abstract visual representation, which is maintained by SPARCL's editing system). The layout of the 3D representation is done in the same two phases as for the layout of the 2D representation: the display objects other than the hyperedges are modelled, then the hyperedges are modelled.

Using the containment relationship, non-hyperedge display objects can be thought of as forming a tree. In modelling a display object in either two or three dimensions, we first model its containment subtree of display objects. This gives the size of the volume of space the object's model must enclose, which in turn allows us to calculate the geometry of the object. Part of the topology of an object's model is a "hollow" space to hold the subtree object models. We translate the subtree of object models into this hollow space. We collect together the subtree models and the object's model into a single model that is used by the parent of that object. If an object may have an indefinite number of children in the same "role" in the containment subtree, then some "sibling layout" algorithm must be used for placing these children. The layout of an object type that has different "roles" for children is handled by a layout algorithm specialized for that object type. For instance, a clause has three "roles" for its non-hyperedge children: clause name, clause arguments, and clause literals. The lay-

out for a clause makes one application of a sibling layout algorithm to the arguments and another application to the literals, and then it places the argument model collection and the literal model collection within the clause model.

Sibling layout. There are several kinds of display objects that have multiple children: a program (window) may contain multiple clauses, a partitioned set may have multiple parts, a part of a partitioned set may have multiple members, an N-tuple has multiple elements, a clause may have multiple arguments or multiple literals, a literal may have multiple arguments, an intensional set may have multiple literals. In 3D layout, we use three different sibling layout algorithms. One is used for clause and literal arguments, another for N-tuple elements, and another for partitioned set parts, partitioned set part members, and clause literals. For 2D layout, we use six sibling layout algorithms: one is used for clause and literal arguments, another for N-tuple elements, another for partitioned set parts, another for partitioned set part members and clauses in a program window, another for clause literals, and another for term and fact tables. These 2D layout algorithms are discussed in the *SPARCL Display* module explanation in chapter 6 (“Implementation”).

(1) The sibling algorithm for arguments places the given argument models in order in a vertical stack (i.e. along the y-axis, which is vertical when seen from the default front view), with the first model at the top of the stack.

(2) The sibling algorithm for N-tuple elements places the given element models in order in a spiral with arrows in between consecutive elements. Example three-dimensional N-tuples are shown in Figure 5. 2 and inside the partitioned set of Figure 5. 3. These are “short” spirals in that they only go around two- or three-fourths of one “loop”. An example of a ten-element N-tuple is shown in Figure 5. 7. The spiral’s axis is parallel to the Z-axis. The center view of Figure 5. 7 looks along this axis. The spiral loosely outlines a cone, with the point of the cone being furthest away from the clause base (and thus nearest the viewer in the default front view). The sides of the cone are nearly parallel to its axis (i.e. it is almost a cylinder). The first element is furthest from the clause base (and thus nearest the viewer in the default front view). We adopted the spiral layout to make the presentation of N-tuples compact in three dimensions. The cone outline (as opposed to a cylindrical outline) was adopted to make all of the elements of the N-tuple visible from the default front view. However, this has not actually made much of a difference in practice. The visibility of the ele-

first → second → third → fourth → fifth → sixth → seventh → eighth → ninth → tenth

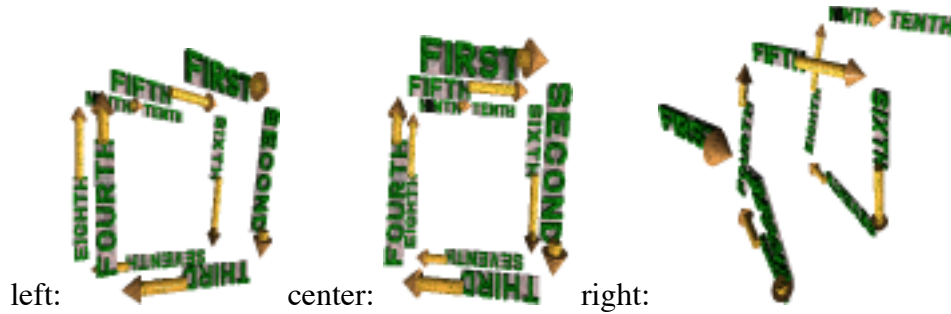


Figure 5. 7: 2D representation of an 10-tuple and three views of a 3D representation of that 10-tuple.

ments of an N-tuple is highly dependent on the distance of the viewpoint from the N-tuple. If the viewpoint is close enough to make *ur* terms legible (as in Figure 5. 7), then the sightlines to the elements of the N-tuple pass *inside* the “cone” of the N-tuple. The design of the layout of N-tuples is an area that needs much more research.

(3) The “compact rectilinear” sibling algorithm is used for laying out partitioned set parts, partitioned set part members, and clause literals. We give examples showing layouts for five, six, and seven elements for each of these three kinds of siblings in Figure 5. 8, Figure 5. 9, and Figure 5. 10. There is no attempt at



Figure 5. 8: 3D Layouts of 5, 6, and 7 terms in a part.



Figure 5. 9: 3D Layouts of 5, 6, and 7 parts in a partitioned set:

providing a layout that reflects a particular order of the models. This algorithm places the given models in an array that when viewed from the default front view has the elements in straight rows and columns, where the models are aligned horizontally (in rows) by their “bottoms” (their smallest X values) and vertically (in columns) by their “left sides” (their smallest Y values). This array is filled in in such a way that it



Figure 5. 10: 3D Layouts of 5, 6, and 7 literals in a clause (for predicate ‘Layout’/0).

is always as near being a filled square of rows and columns as possible. The algorithm for this places models in abstract positions that give the row and column for each model, then translates the abstract position to concrete positions based on the sizes of the models placed in the various rows and columns. It has been designed so that more members can be added to a layout in such a way that the “compactness” of the layout is always preserved but no members already placed need be moved. The “next” abstract position for a model is defined by $\text{abstract_position}/4$ in Figure 5. 11.

The four cases defined in the $\text{abstract_position}/4$ predicate are: If the last member in the array (RowIN,ColumnIN) was at the lower right corner (RowIN == ColumnIN), then the array is square and the next member is placed in the leftmost position of the next row. If the last member in the array is in the last row and the last column (but the array isn’t square), then the next member goes to the top of the next column. If the last member is in the last column but above the last row, then place the next member under the last member. Finally, if the last member is in the last row but before the last column, then

place the next member to the right of the last member. This gives the pattern shown in Figure 5.12. The first member is placed at (1,1), as shown. The “next” position from (1,1) is (2,1) (i.e. a query of ‘ $\text{abstract_position}(1, 1, R, C)$ ’ binds R and C

```

abstract_position(RowIN,          ColumnIN,
RowNEXT, ColumnNEXT) :-
    RowIN == ColumnIN
        -> RowNEXT is RowIN + 1,
            ColumnNEXT is 1
    ; RowIN == ColumnIN + 1
        -> RowNEXT is 1,
            ColumnNEXT is RowIN
    ; RowIN < ColumnIN
        -> RowNEXT is RowIN + 1,
            ColumnNEXT is ColumnIN
    ; RowIN > ColumnIN + 1
        -> RowNEXT is RowIN,
            ColumnNEXT is ColumnIN + 1.

```

Figure 5. 11: $\text{abstract_position}/4$ predicate for “next” position calculation in compact rectilinear sibling algorithm

such that $(R,C)=(2,1)$). Thus the cell (2,1) of Figure 5.12 contains a ‘2’. This placement algorithm guarantees that regardless of the number of members being placed, the absolute difference in the number of columns and rows is never more than one.

Tubes: 3D Connecting Lines.

As we discussed in section 4 (“Visual Programming Design Elements”) of chapter 3, there are three basic ways to show relationships in static nonlinear representations. SPARCL uses all three, but heavily depends on connecting lines. SPARCL uses “same text” to classify predicate names (clauses and literals), “same shape” or “same texture” is to classify term types (variables and constants), containment to show elements in a set, parts of a partitioning, and literals in a clause, and connecting lines to show term coreference (all terms in the same hyperedge unify).

SPARCL employs several 3D representation techniques in its representations of connecting lines. First, items connected by a line are equivalent (i.e., they unify). There may be many items that are intended

to unify with each other (in a textual language these items could be references to the same variable name). SPARCL uses a hyperedge to connect these many items (a hyperedge is an “edge” that connects more than two items). There may be several hyperedges in a single scene, but generally there are fewer than 10. These hyperedges have the same crossing problem common to connecting lines. In SPARCL, these crossings are made less confusing in several ways. Hyperedges are depicted as smoothly curving tubes which curve through all three dimensions. A tube is composed of one or more segments. The segments join each other smoothly (their tangent lines are identical at the join point -- the geometry of these tubes is discussed in more detail below). Each hyperedge is a different color, i.e., all of the segments of

	4	3	2	1
1	13	7	3	1
2	14	8	4	2
3	15	9	6	5
4	16	12	11	10

Figure 5.12: Order of placement for compact rectilinear sibling layout algorithm.

a hyperedge are the same color, segments of different hyperedges are different colors. Hyperedge tubes usually don't intersect. Generally the tubes rise to different heights above the basic plane of the "program". Viewed from some particular angle, such as "in front of" the program, these tubes will appear to cross each other. But, if the viewer changes her view point the crossings will change. The tubes are shaded and shine according to the lighting, and they cast shadows on each other and the other program elements. In the following discussion we differentiate between crossings and intersections: crossings are an artefact of the view used to render a clause, where intersections are intrinsic to the geometry of the model and thus they exist independent of the particular view used to render a clause.

Intersections. There are three different severities of intersections between hyperedges: *centerline intersections* are the most severe where the centerlines of two different hyperedges intersect; *partial centerline intersections* where the centerline of a hyperedge passes through another hyperedge without intersecting that other hyperedge's centerline; and *marginal intersections* where the surfaces of two hyperedges intersect without either of their centerlines intersecting the surface of the other hyperedge. The centerline intersections are potentially the most confusing since there is no vantage point from which the hyperedges appear separated. The marginal intersections are barely noticeable as intersections and thus cause little confusion. The partial centerline intersections are in between these other two kinds in the degree to which they are visually confusing. There is a partial centerline intersections in the 'Union'/3 clause of Figure 5. 4. It is most easily seen in views (c) and (d). It's in front of the third argument of the clause. There are two marginal intersections in the 'ID3 Tree'/4 clause of Figure 5. 14 and Figure 5. 15. One of them is in front of the third argument of the 'Select Attribute'/5 literal. The other one is in front of the top part of the partitioned set in the first argument of the clause. None of these intersections is particularly confusing.

We have not determined a statistical profile for the prevalence of intersections. However, we propose that the higher the ratio of hyperedges to graphical token count (i.e. the higher the "density" of hyperedges), the greater the likelihood of intersections. We define graphical token counts for SPARCL in chapter 8 ("Objective Analysis"). This is based on the observations that the hyperedge endpoints are confined to the volume of space occupied by the clause and that the volume of space occupied by

a clause is roughly proportional to the number of graphical tokens in the clause. The token count for ‘Union’/3 is 17, which gives us a ratio of hyperedges to token count of 3/17, or about 0.18. The hyperedge to token count ratio for the ‘ID3 Tree’/4 “heterogeneous” clause is 8/54, which is about 0.15. This ratio for the ‘ID3 Tree’/4 “homogeneous” clause is 4/27, which is equal to the “heterogeneous” clause ratio (about 0.15). This ratio must always be between 0 and 0.5: there must always be a few more tokens than twice the number of hyperedges. A refinement of this metric is to “weight” the number of hyperedges more heavily: the more hyperedges there are, the more likely that there will be intersections between them. For instance, the adjusted hyperedge-to-size metric P could be:

$$\frac{(H \log_2(H + 1))}{S}$$

where H is the number of hyperedges in the clause and S is the graphical token count “size” of the clause. (The $H+1$ avoids problems with \log_2 of 0). The adjusted values are: ‘Union’/3 = 0.857; heterogeneous clause = 0.469; and homogeneous clause = 0.344. These adjusted values might arguably be in better agreement with the observed severity of intersections of these clauses: we claim that the two marginal intersections of the heterogeneous clause are less an impediment to readability than the partial centerline intersection of the ‘Union’/3 clause. We have not attempted a general analysis of all of our example SPARCL programs to analyze this metric, so we do not have any data on common values to compare with the three values reported here. This is an area for further study.

Also, it is apparent that the more problematic the kind of intersection is the less likely it is (i.e. centerline intersections are less likely than partial centerline intersections, which are less likely than marginal intersections).

Crossings. Figure 5. 13 shows a 2D representation of another clause of our SPARCL program for implementing the ID3 machine learning algorithm [Quinlan 1982]. This clause is shown in Figure 7. 3 of our discussion of the entire program in chapter 7. This is the “heterogeneous” clause of the ‘ID3 Tree’/4 predicate definition. (This clause is called “heterogeneous” because it uses a ‘Heterogeneous Examples’/2 literal as opposed to the other clause for ‘ID3 Tree’/4 shown in Figure 5. 5 that uses a ‘Homogeneous Examples’/3 literal.) Understanding the ‘ID3 Tree’/4 predicate is not

important here; we are only interested in noting the complexity of the line crossings. Figure 5. 14 and Figure 5. 15 show two views of a 3D concrete representation (for POV-RAY) of this “heterogeneous” clause.

There are several aspects of SPARCL’s approach that help the viewer comprehend what items are equivalent in spite of the line crossings. The main problem is to easily identify which apparent intersections of segments are accidental crossings and which are joins. The common colors (obviously, not visible here in black and white) are a simple and effective way to see that two segments do or do not belong to the same hyperedge. The smooth joins within a hyperedge help the viewer to distinguish a crossing that is “accidental” as most accidental crossing are obviously not smooth. The shading, highlights, and shadows all help the viewer discern that two segments accidentally cross since they are not in the same location of the scene. The highlights and shading are different due to a different

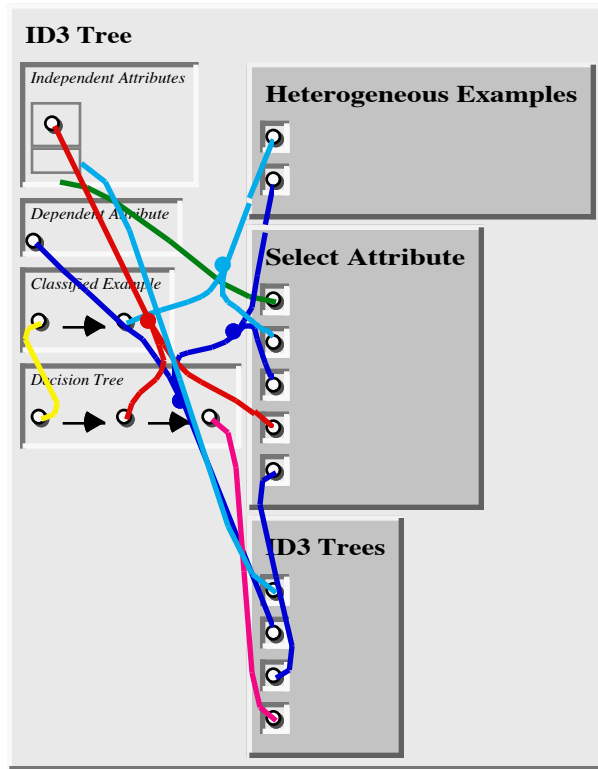


Figure 5. 13: 2D Representation of the “heterogeneous” clause of the ‘ID3 Tree’/4 predicate definition.

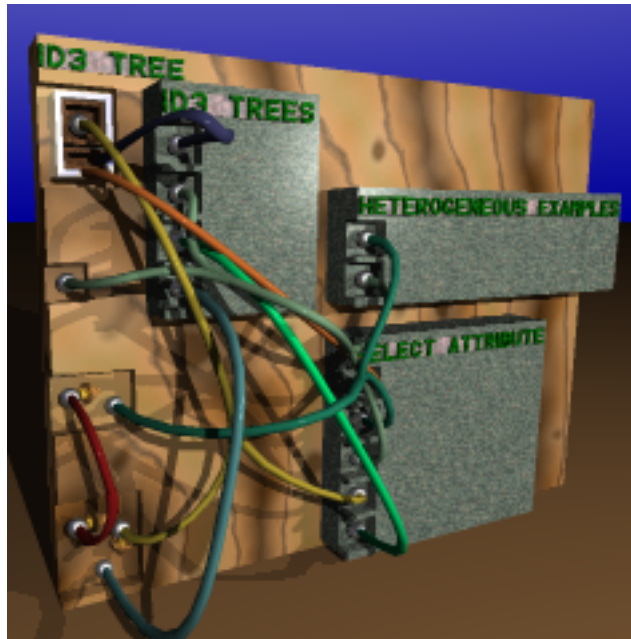


Figure 5. 14: Full view of a 3D concrete representation of the ‘ID3 Tree’/4 “heterogeneous” clause.

viewing angle. If the viewer moves her viewpoint, then many of the accidental crossings will change relative position. Also, the view dependent lighting features (highlights) will also change their relative position. The texturing of



Figure 5. 15: Close view of ‘ID3 Tree’/4 “heterogeneous” clause.

the segments can help further if they are reflective or translucent. If reflective, then one of them may reflect the other in its surface. If translucent, then the one passing behind the other will be dimly visible through the front one.

These aspects of a 3D scene with tubes for connections provide the viewer with many different cues for interpreting the scene. These cues are all ones that people are used to interpreting in dealing with real scenes and, consequently, require no conscious effort on the part of the viewer to utilize them even without explanation. Thus, a very complex collection of hyperedges can be understood more readily in 3D than in 2D.

Since line crossings are not as devastating to comprehension in 3D as they are in 2D, graph layout programs can allow more crossings to occur as a trade-off for other improvements (such as better node placement, or a simpler or faster layout algorithm).

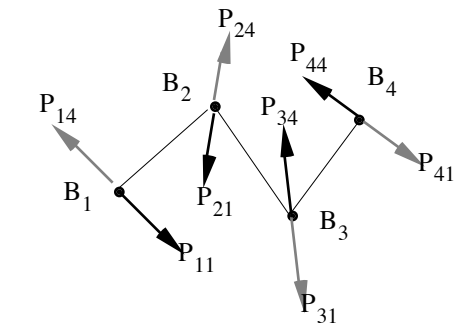
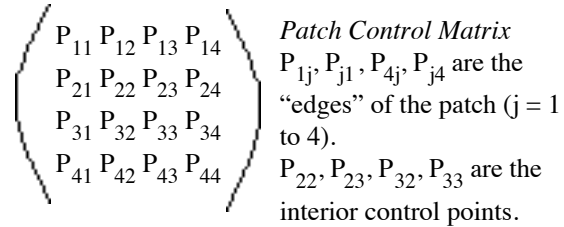
Bezier Tubes.

The segments of a hyperedge are a “Bezier tube”. The 3D model of a Bezier tube is a pair of bicubic Bezier patches [Foley et al. 1990] (for convenience called “top” and “bottom”). A bicubic Bezier patch has 16 “control points” which determine the

shape of the patch. This can be seen in Figure 5.16. If these control points are arranged in a four by four matrix, then the points in the outer rows and columns of this matrix correspond to the four outer edges of the patch: e.g., the four control points in the top row determine the shape of a Bezier curve which is an edge of the patch. The tube is constructed by having the top and bottom patch have the same control points in their outer columns: column 1 of the top patch's control matrix equals column 1 of the bottom patch's control matrix. The bottom patch's "interior" control points are a special "inversion" of the interior control points of the top patch.

The model for a Bezier tube is generated from a cubic Bezier space curve. The cubic Bezier space curve uses only four control points, one at each end and two in the middle. The Bezier curve is defined to start and end at end control points, and to smoothly approximate the middle control points. This provides a simple way to design smoothly curving lines. With careful choice of control point values, Bezier curves can be joined (i.e. share an endpoint) such that they have identical tangent lines at the join point.

A hyperedge connects N items (each item has an attachment point). The layout of the segments which make the hyperedge is a recursive process, starting with the set of connection points being the set of all attachment points. Each segment is a Bezier tube made of two patches. There is one segment in a hyperedge which connects two items, three segments to connect three items, five segments to connect four items. In general, there are $2(N - 2) + 1$ segments in a hyperedge which joins N items. The hyperedges are modeled in the order based on their minimum connection point Z-dimension distance from the clause base. The hyperedge with the least distance from the base is modeled first. The proposed join plane moves



Bezier Control Points & Offset Vectors

Bezier curve = $\langle B_1, B_2, B_3, B_4 \rangle$

The dark vectors are the "main" ones. The gray vectors are the "secondary" ones.

Selected patch control points are shown at the ends of the offset vectors.

Figure 5.16: Patch control matrix and Bezier control points.

out away from the previous actual join plane base by a fixed minimum. The actual join plane is the plane further from the clause between the proposed join plane and the plane a default fixed distance further away from the clause base than the “highest” point the hyperedge must clear. The terms being connected by a hyperedge have a minimal common containing term. This common containing term defines a subtree of the containment tree. For a term being connected by the hyperedge, that term has a “path” of containing terms within the hyperedge subtree, not including the root common containing term. The highest point the hyperedge must clear is the highest point among the models for the terms on the hyperedge containing term paths. For example, suppose two variables are being connected where one variable A is in an N -tuple B in a part C of a partitioned set D in an argument E of a clause X and the other variable F is in a part G of a partitioned set H in a part I of a partitioned set J of an argument K of a literal L in the same clause X . The two variables A and F have hyperedge containment paths: $A B C D E$ and $F G H I J K L$. The two paths do not contain the clause X , since X contains the hyperedge. The highest point the hyperedge must clear is the highest point among the models for $(B C D E G H I J K L)$. This “clearance” analysis provides a simple way to minimize irrelevant intersections of the segments of the hyperedge with the models of terms (e.g. a hyperedge going through the wall of a partitioned set and a part of that set to get to a variable inside the part, instead of going “over” that wall). The entire path must be considered rather than the term highest up the path because the fact that a term T abstractly contains another term S does *not* imply that the model for T extends further away from the clause base than the model for S . (It *does* imply that the projection of the model for T to the X-Y plane contains the projection of the model for S in that plane.)

Determining Bezier space curves for a set of connection points. First, find the two connection points closest together, and calculate the point which is the centroid of the remaining $N-2$ connection points. Calculate a “join” point which is the centroid of the two selected connection points and the $N-2$ -points centroid. Calculate the centroid of the join point and the two selected connection points. This last centroid is the ‘JoinInterior’ point. For each selected connection point, if that connection point was a previous join point then use the associated JoinFollowing point as the ConnectionInterior point, otherwise calculate the average of the (current) join point and that connection point as the ConnectionInterior point. Create two bezier curves,

joining each of the two points to the join point. The control points of the curve for a particular connection point are: the connection point, the ConnectionInterior point, the JoinInterior point, and the join point. The two curves have the same third and fourth control points. This ensures that the two curves will join tangentially at the join point. Finally, calculate a JoinFollowing point which is the JoinInterior point “mirrored” across the join point. This JoinFollowing point is used when defining the curve which starts at the join point. Add the join point (its JoinFollowing point as an annotation) to the set of points to be combined in the hyperedge. Recursively invoke the segment curve generating process on this new set.

This procedure can be described symbolically as follows: Let P be a set of connection points. Let $CLOSEST(X)$ be a function which returns a pair set of points which are two points in the set of points X which are closest together. Let $CENTROID(X)$ be a function which returns the point which is the centroid (average) of the points in X . Let $CARDINALITY(X)$ be a function which returns the cardinality of the set X . Let $THIRDS(X)$ be a function which returns a tuple of four points, $\langle P1, T1, T2, P2 \rangle$, where $P = \{P1, P2\}$ and $T1$ and $T2$ divide the line between $P1$ and $P2$ into thirds. Let $JOINPLANESHIFT(V)$ be a function which returns V with its z component set to $JOINPLANE$ (some value input to this procedure). Let $FOLLOWINGMAP$ be a set of ordered pairs (an association list) with first element a point in P and the second element the “following” point for P . Not all points in P have associations in $FOLLOWINGMAP$. It is initially empty.

The function $CURVES(JOINPLANE, X, FOLLOWINGMAP)$ returns a set of Bezier space curves (specified by four-tuples) given a list of points X and a $FOLLOWINGMAP$. The curves with join plane $JOINPLANE$ for a set of connection points P is found by $CURVES(JOINPLANE, P, \{\})$.

If $CARDINALITY(P) = 2$, then do: If both points in $P = \{P1, P2\}$ have following points in $FOLLOWINGMAP$ ($\langle P1, F1 \rangle$ and $\langle P2, F2 \rangle$) then the Bezier space curve is $\langle P1, F1, F2, P2 \rangle$. Else, if only one point has a following point ($\langle P1, F1 \rangle$), then let $M = CENTROID(\{F1, P2\})$, and $CURVES = \{\langle P1, F1, M, P2 \rangle\}$. Else, neither point has a following point, so let $\langle P1, T1, T2, P2 \rangle = THIRDS(P)$, $J1 = JOINPLANESHIFT(T1)$, and $J2 = JOINPLANESHIFT(T2)$. The Bezier space curve result is $CURVES = \{\langle P1, J1, J2, P2 \rangle\}$.

When $CARDINALITY(P) > 2$, then do: Let $A = \{A1, A2\} = CLOSEST(P)$, $P' = P - A$, $C = CENTROID(P')$, and $M = JOINPLANESHIFT(CENTROID(A \cup \{C\}))$.

The result of the rest of the procedure is to define a Bezier space curve that connects A1 and M and another one which connects A2 and M. Also, a “following point” is calculated for M and added to the FOLLOWINGMAP. To connect A1 to M: Either $\langle A1, F1 \rangle$ is in FOLLOWINGMAP, or else let $F1 = \text{CENTROID}(\{A1, M\})$. The curve from A1 to M is $\text{CURVE1} = \langle A1, F1, C, M \rangle$. Similarly, F2 is determined for A2 and M, and the curve for A2 to M is $\text{CURVE2} = \langle A2, F2, C, M \rangle$. The “following point” for M is $F = (M + (M - C))$. The resulting set is:

$$\begin{aligned} \text{CURVES} &= \{\text{CURVE1}, \text{CURVE2}\} \\ &\cup \text{CURVES}(\text{JOINPLANE}, \\ &\quad P' \cup \{M\}, \\ &\quad (\text{FOLLOWINGMAP} \\ &\quad \quad - \{\langle A1, F1 \rangle, \langle A2, F2 \rangle\}) \\ &\quad \cup \{\langle M, F \rangle\}). \end{aligned}$$

The new connection point set ($P' \cup \{M\}$) has one fewer points in it than does P: P minus the two attachment points, plus the new join point M. Thus, this recursion will eventually halt. The JoinFollowing point (F) in the above construction ensures that the segment “leaving” from the join point will join tangentially with the segments “entering” the join point.

Calculating the control points of the Bezier patches for a Bezier tube. The Bezier tube for a given a cubic Bezier space curve has that curve as its (approximate) axis. The simplest way to model this is to “sweep” a circle along the curve, with the center of the circle following the curve and the plane of the circle always perpendicular to the tangent of the curve. The modeling system we used does not have the capability of sweeping a 2D figure along a Bezier space curve, so we approximate this effect by specify two Bezier patches as described earlier, one for the “top” half of the tube and the other for the “bottom” half of the tube. The Bezier patch approach can be preferable to the sweep approach when both are available since the Bezier patches are usually faster to render than the sweep.

Figure 5. 16 gives an indication of the setting of the patch control points. The “top” patch control points are placed at a fixed offset distance from the curve in certain

directions. This fixed offset distance is approximately the radius of the Bezier tube. At the endpoints, the patch control point's offset direction is perpendicular to the curve. For an interior control point, the offset direction is halfway between the directions toward the two adjacent control points. Patch control points are placed at the offset distance in both the positive and negative offset directions. Also, an "up" offset direction is determined at each control point, and two more control points are placed, one each "up" from the first two offset control points. This gives four control points in the "top" patch at each control point of the original curve. Care must be taken in placing the patch's control points in the appropriate columns and rows of the patch control matrix to create a smooth "half tube" appearance. If these control points are placed in the wrong parts of the patch's control matrix the surface becomes twisted. The "bottom" patch's control points are generated from the "top" patch's control points. The exterior columns are the same in both patches. The two "inside" columns of the "bottom" patch are offset twice the negative of the "up" direction from the two inside columns of the "top" patch.

Discussion

In this section we review the major points presented in this chapter and offer some assessments of the 3D representation effort.

Summary. The 3D designs of most of the concrete representations of the display objects in the full, abstract visual representation of SPARCL are simple extensions of their 2D designs. The notable exception to this is the representation of N-tuples. In 3D the representation is built around a spiral (or more accurately a ziggurat), whereas it is a horizontal sequence in the 2D representation. We showed several views of the 3D representations of three clauses (the clauses defining 'Union'/3 and 'ID3 Tree'/4). Certain elements of SPARCL are not currently representable (term tables, fact tables, and comments), there is no representation for a group of clauses in the same "scene", all text is represented as upper-case and the 3D representation is not interactive. We expect to explore some of the possibilities for representing comments, as well as developing table representations and making the 3D representation interactively editable, in future work on SPARCL.

There are many approaches to modelling and rendering 3D scenes. Our imple-

mentation of the 3D representation allows the user to choose between three systems for rendering the 3D model of a SPARCL clause: *POV-Ray*; *OpenInventor* [Wernecke 1994], and *QuickDraw3D* [Apple Computer, Inc. 1995]. The *POV-Ray* system was used for most of the figures of this chapter since it produces a high quality rendering, but it is too slow for interactive viewing. The *OpenInventor* system produces fairly high-quality renderings that can be used interactively (on SGI's multi-processor Onyx and Challenge systems). However, it was awkward for us to use since it only runs on Silicon Graphics systems, and the rest of our tools are on the Apple Macintosh. The *QuickDraw3D* system works well for SPARCL since SPARCL is implemented on the Apple Macintosh and QD3D is a full-featured 3D system native to the Macintosh that is intended for interactive 3D viewing and editing. It provides minimally acceptable rendering, but it is fast enough that a user can change views of a clause in real time (on an Apple Macintosh 8500/120 with 32 Mbytes of RAM).

The 3D representation of SPARCL is particularly challenging because we fully automate the modelling and layout of the concrete representation. The 3D concrete representation is derived from the partial abstract visual representation (which is in turn derived from the full abstract visual representation, which is maintained by SPARCL's editing system). The layout of the 3D representation is done in the same two phases as for the layout of the 2D representation: the display objects other than the hyperedges are modelled, then the hyperedges are modelled. The strong similarity in approaches to automated layout for 2D and 3D representations should make it easier for us to combine these approaches. Such a combination would yield an incrementally updated layout for the 3D representation (such as is currently implemented for the 2D representation). This is essential for interactive editing of the 3D representation.

Two major aspects of the automated layout of the 3D representation are the layout of "siblings" within a container and using Bézier tubes for hyperedges. The sibling layout algorithms implement a few different styles, each designed with an eye toward minimizing the effect of incremental changes.

Hyperedges are depicted as smoothly curving tubes which curve through all three dimensions. A tube is composed of one or more segments. The segments join each other smoothly. Each hyperedge is a different color. Hyperedge tubes can intersect in one of three ways: centerline intersection (the most confusing), partial centerline intersection, and marginal intersection (which causes little confusion). The frequency of these intersections is roughly proportional to the ratio of hyperedges to graphic

token count of the clause and is lower for the more confusing kinds of intersections. In any event, the frequency of intersections is low. We reported this ratio for the three example clauses presented in this chapter, but we have not done a more general study of this hyperedge-to-size metric, but such a study is certainly a possibility for the future. The tubes rise to different heights (join planes) above the basic plane of the “clause”. Viewed from some particular angle, such as “in front of” the program, these tubes will appear to cross each other. But, if the viewer changes her view point the crossings will change, helping the viewer to understand the layout of the tubes. Other visual aids to understanding the layout of the tubes include shading, highlights, and shadows.

The geometry of the segment of a hyperedge tube is similar to that resulting from sweeping a circle along a Bézier space curve, where the circle is centered on the curve and the plane of the circle is kept perpendicular to the tangent of the curve at the point of intersection between the curve and the plane of the circle. The actual construction of the model uses Bézier patches. There are relatively complex algorithms for placing the control points of the underlying Bézier space curve and using these to generate the Bézier patch control points. The placement of the control points of the underlying Bézier space curve for a hyperedge segment relies in part on the determination of the join plane for that hyperedge. Calculating the join plane requires analyzing certain of the models of the terms that “abstractly” contain the terms being connected by the hyperedge. This is necessary to avoid having a segment of the hyperedge pass through the model of one of these abstractly-containing terms.

Assessment. In our work on 3D representations for SPARCL we have succeeded in solving various problems in the design of such representations, particularly in the development of approaches to automated layout in 3D. The hyperedge representation is generally satisfying excepting the “scale” problem discussed below. We simplified the layout problem by making the layout of a hyperedge largely independent of the layout of any other hyperedge (with the exception of the join plane “incrementing”). We speculated that there is enough “room” in three dimensions that it would be relatively rare that two hyperedges would have an intersection. Since actually analyzing for intersections and adjusting to avoid them would make the layout of hyperedges much more complicated, we felt the rare intersection would be acceptable. This speculation seems to have proved true. We have only occasionally seen intersections in

the 3D representations of clauses, as we had expected. It was necessary to “bias” the layout of hyperedges to avoid having them all lined up, one over the other, in the default front view. It is possible to easily understand very complex groupings of hyperedges in 3D that are very hard to understand in 2D. This comparative ease of understanding frequently holds in the default front view, but when it doesn’t, one can easily view the hyperedges from another vantage point that does make the relationships depicted by the hyperedge models clear.

The 3D representation we have presented in this chapter is not as successful as we had hoped. It is not particularly easy to read. There are difficulties in deciding how to size the models for the various terms, particularly hyperedges and partitioned sets. The hyperedges need to be thicker when they are longer as can be seen comparing Figure 5. 4, Figure 5. 5, and Figure 5. 14. The hyperedge models in these three figures are all the same radius. They are somewhat too fat in Figure 5. 4, unnecessarily obscuring parts of the clause and each other. They are better in Figure 5. 5, although perhaps still somewhat fatter than they need to be (and thus tending to a little more visual interference than necessary). They are a good thickness in Figure 5. 14. The partitioned sets and partitioned set parts also have a problem with the thickness of the walls of the open-top boxes modelling these terms, the overall size of these boxes, and the size of the clause in which they are contained. As the overall box size or clause size increase one wants to have the thicknesses of these boxes increase. However, having these changing model sizes makes viewing multiple clauses more awkward in that the models for the same types of terms in these clauses have different appearances (at least different proportions).

We have had little experience with multiple clause views in three dimensions, so the value of that approach largely remains to be investigated.

In this chapter we presented our approach to three-dimensional representation in SPARCL: the 3D representation in general of SPARCL, our approach to rendering 3D models, the automated layout of 3D models of the abstract representation of a SPARCL program, and a detailed presentation of the modelling of hyperedges. We have made significant progress in starting the investigation of the 3D representation of SPARCL and thus the 3D representation portion of the feasibility part of our initial hypothesis. At this point, the feasibility of a usable 3D representation for visual logic programming with partitioned sets is uncertain. Nonetheless, we believe that if the various issues discussed above are addressed then this approach will prove very satisfying for

developing programs.

- Apple Computer, Inc. 1995 *3D Graphics Programming With QuickDraw 3D*, Apple Computer, Inc. Addison-Wesley Publishing Company:Reading, Massachusetts. 1995.
- Foley et al. 1990 *Computer Graphics Principles and Practice* by James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. Addison-Wesley, Reading, Massachusetts, 1990.
- Quinlan 1982 "Induction of Decision Trees" by J. R. Quinlan. Pages 81-106 in *Machine Learning*, vol. 1, no. 1, edited by R. S. Michalski, T. M. Mitchell, and J. Carbonell. Palo Alto, California: Tioga. 1982.
- Spratt&Ambler 1994 "Using 3D tubes to solve the intersecting line representation problem" by Lindsey L. Spratt and Allen L. Ambler. Pages 254-263 in *Proceedings 1994 IEEE Symposium on Visual Languages*, October 4-7, 1994. St. Louis, Missouri. IEEE Computer Society Press: Los Alamitos, CA.
- Wernecke 1994 *The Inventor Mentor* by Josie Wernecke (Open Inventor Architecture Group). Addison-Wesley Publishing Company:Reading, Massachusetts. 1994.

Chapter 6

Implementation

SPARCL has several major parts to its implementation: the interpreter, the display system, and the editing system. Figure 6.1 shows a diagram of the relationships between these subsystems, and lists the major modules implementing each of the subsystems. The interpreter interprets an internal form of SPARCL. The editing system handles all of the interactions with the user and it maintains the canonical display representation of the program. The display system generates a concrete representation from the canonical display representation. To evaluate a query, the editing system converts the canonical display representation into the internal form, invokes the interpreter, converts the resulting internal form back to the canonical display representation, and finally uses the display system to present this result to the user.

The entire system is implemented in *Logic Programming Associates, Ltd.* MACPROLOG32 and runs on Apple Macintosh computers running system 7.0 or greater. Although MacProlog32 does not explicitly support “modules”, we have organized our implementation of SPARCL very similarly to how we would have done it if

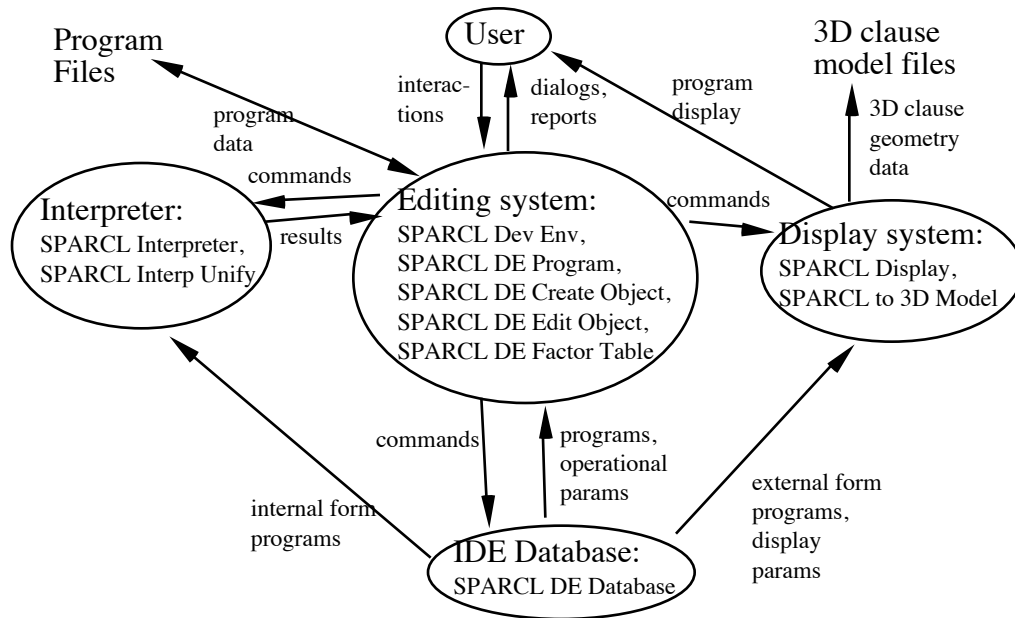


Figure 6. 1: Diagram of major subsystems of SPARCL, with selected modules implementing those subsystems.

modules had been available. Thus, we speak of SPARCL as being implemented by a collection of modules. There are 62 of these modules, all but one (*Browser*) is implemented by us. These are listed alphabetically in Figure 6. 2. Those modules that have names starting with “SPARCL” are specific to the implementation of SPARCL, the other modules are of more general use. We show important modules for the major subsystems in the implementation diagram in Figure 6. 1.

The main module for the SPARCL application is *SPARCL Dev Env*. The editing system’s main modules are: *SPARCL DE Program*, which handles interactions with the user; and, *SPARCL DE Create Object*, *SPARCL DE Edit Object*, and *SPARCL Factor Table*, which together manage the creation and editing of SPARCL programs. The display system’s main modules are: *SPARCL Display*, which presents the two-dimensional representation of SPARCL programs; and, *SPARCL to 3D Model*, which creates a file that defines the three-dimensional representation of SPARCL programs. The interpreter’s main modules are: *SPARCL Interpreter*, which interprets SPARCL programs; and, *SPARCL Interp Unify*, which implements the partitioned set unification algorithm.

In the rest of this chapter we discuss each of the major subsystems (editing system, display system, and interpreter) of SPARCL in detail. Various aspects of the implementation of SPARCL are presented in the course of discussing these subsystems. The

Arithmetic Utilities	SPARCL 3D Model Util	SPARCL Inventor Model
Binary Tree	SPARCL Database Kernel	SPARCL Linear Transform
Browser	SPARCL DE Clause DB	SPARCL Log Analysis
Clause Utilities	SPARCL DE Create Object	SPARCL Menus Kernel
Containment Tree DB	SPARCL DE Database	SPARCL Objects Kernel
curve_to_lines	SPARCL DE Edit Object	SPARCL Output Window
Display Graph	SPARCL DE Factor Table	SPARCL Pgm Window Name
Display Lines	SPARCL DE Menus	SPARCL Picture DB
Error Handling	SPARCL DE Object Util	SPARCL POV Model
File Utilities	SPARCL DE Program	SPARCL Project&Pgm Util
garbage_collecting_call	SPARCL Dev Env	SPARCL QD3D Model
Graph Averaging Layout	SPARCL Dev Env Util	SPARCL Readable Linear
Graph Utilities	SPARCL Dialog Kernel	SPARCL Script
List Utilities	SPARCL Dialogs	SPARCL to 3D Model
My Conversion Support	SPARCL Display	SPARCL Trans Disp Obj
Picture DB	SPARCL Display Util	SPARCL Visual Transform
Picture DB Utilities	SPARCL Halstead	straight
Point-Line Distance	SPARCL Interaction Log	String Utilities
portray	SPARCL Interp Kernel	Term Utilities
Preference Utilities	SPARCL Interp Unify	vectors
Queue Utilities	SPARCL Interpreter	

Figure 6. 2: Modules used in the implementation of SPARCL.

editing system section includes: the interaction style of SPARCL, which relies heavily on popup menus (as opposed to other common interaction techniques such as: command lines, tool palettes, or pull-down menus); the four internal program representations used by SPARCL; the use of program transformation between “higher” and “lower” levels of these representations; and, the SPARCL integrated development environment (IDE) database facility with its support of “undo” using checkpoint and rollback services. The display system section includes: incremental layout; two-dimensional sibling layout algorithms; two-dimensional hyperedge layout; organizing the program window picture database for rapid searching; and, writing 3D model files for the three different rendering systems. The interpreter section includes: details of the implementation of “solving” a goal; details of the implementation of the unification algorithm presented in chapter 4 (“Partitioned Set Unification”).

The Editing System.

The “editing system” refers to all of the services of SPARCL except the program interpreter and the program representation display. It provides the program development environment, it manages the interactions with the user, maintains the canonical visual program representation, and provides various tools to aid in developing and maintaining SPARCL programs. The editing system also provides the interaction logging, the integrated scripting/tutorial facility, and various interaction log analysis tools. The *SPARCL Dev Env* module pulls together all of the services of SPARCL. The non-editing system services (program interpretation and display) are accessed through *SPARCL DE Program*.

The editing system implements the structured semantic editing approach we discussed in chapter 3 (“Design Elements”). There are two kinds of interactions: through the menus on the main menu bar or through the program window object popup menus. Examples of these interactions are given in appendix 1 (“Tutorial Introduction to SPARCL”).

The menus on the main menu bar are: ‘File’, ‘Edit’, ‘Tools’, and ‘Windows’. (The first two are standard menu names used in all Macintosh applications, and the other two are also very common.) The ‘File’ menu provides services for creating, opening, saving, and closing evaluable programs, term set programs, and projects (projects are collections of programs). Other system services are also on this menu, such as record-

ing a system comment (for the SPARCL developer to read), setting and saving preferences, invoking the tutorial system or the help system, and quitting SPARCL. The ‘Edit’ menu provides the standard copy, cut, paste, clear operations for both text and program display objects. The ‘Tools’ menu provides some SPARCL program analysis tools such as size measurement and program overview graph generation, and the interaction log analysis tools. The ‘Windows’ menu provides access to the windows for all of the programs that have been opened (the window for a program may be invisible, in which case the menu is the primary way to make it visible).

The program window object popup menus are the major way to edit a program and they enforce the semantic-structured editing paradigm. When the user depresses the mouse button, the system pops up a menu at the cursor position. There are many different popup menus possible, the one popped up depends on the type of smallest (in area) object beneath the cursor when the mouse button was depressed. The options in the popup menu are the operations that the user may do to the associated display object. Options that are not meaningful in the current state of the system are grayed out (and unavailable as choices). If there is no object beneath the cursor (i.e. the cursor is over the background of the window), then the associated display object is the program for that window. The options in a popup menu are either to modify the associated object, to select it, or to view a related object. The possible modifications are *semantic*, not representational. (One exception to this is that the user may explicitly position a clause in the program window.) For instance, the popup menu for an argument-type object supports inserting a new term in the argument or creating a comment for the argument. The system decides where the new term or comment goes in the argument representation and how to adjust the argument representation to allow for the change. The popup menu for an argument does not support explicit control of the *appearance* of the argument.

A module dependency graph of selected modules used by *SPARCL Dev Env* is shown in Figure 6. 3. In the module dependency graphs shown in this chapter, each node represents a module and if two modules are connected by an edge, then the module to the left “depends on” the module to the right. A module X depends on another module Y if there is at least one clause in X that has a literal in its body that references a predicate defined in Y. This is a logic programming version of a “call” graph. In Figure 6. 3, *SPARCL Dev Env* uses all of the other modules in the graph and *SPARCL DE Program* uses *SPARCL DE Edit Object*, *SPARCL Interaction Log*, and

Picture DB. There are many more modules used by the modules in Figure 6. 3 than are shown in figure. We discuss some of these additional modules and dependencies later in this chapter.

SPARCL Dev Env. The bulk of the implementation of SPARCL is accessed through *SPARCL DE Program*. We will postpone discussion of those modules in Figure 6. 3 that are also depended on by *SPARCL DE Program* to our discussion of the *SPARCL DE Program* module. These postponed modules are *SPARCL DE Edit Object*, *SPARCL Interaction Log*, and *Picture DB*. The other modules of the dependency graph for *SPARCL Dev Env* in Figure 6. 3 implement SPARCL program analysis tools (*SPARCL Halstead* and *Display Graph*), interaction log analysis (*SPARCL Log Analysis*), an interactive tutorial scripting system (*SPARCL Script*), and a help system (*Browser*). The *SPARCL Halstead* module implements the software measurement tools for SPARCL. *Display Graph* is a general purpose module we developed for displaying directed graphs. It was used to generate the display of the module-use graph in Figure 6. 3. The program analysis it supports in SPARCL is in displaying predicate use (“call”) graphs for SPARCL procedures. The *SPARCL Log Analysis* module implements tools for reading, analyzing, and reporting on SPARCL interaction logs and system comment files. We use the *Browser* module (provided with MACPROLOG32) to provide convenient access to “static” information about SPARCL, a simple help system. The *Pic-*

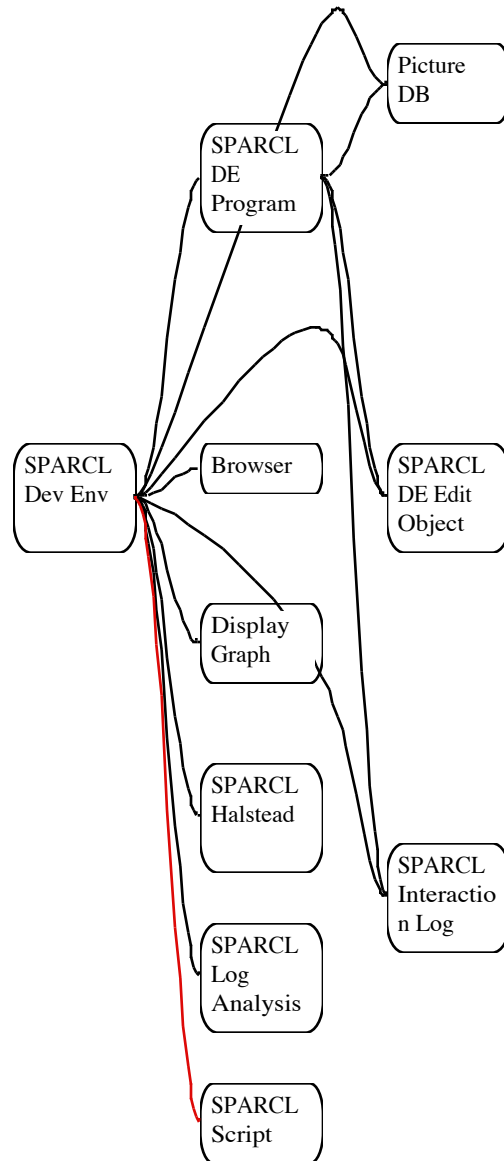


Figure 6. 3: Selected modules used by SPARCL Dev Env.

ture DB module implements a picture database for each program window in the SPARCL environment. This picture database is used when the mouse button is depressed to rapidly determine which display object is the smallest display object beneath the cursor.

SPARCL DE Program. A simplified dependency graph for the modules used by *SPARCL DE Program* is shown in Figure 6. 4. The *SPARCL DE Program* module coordinates the major services of the SPARCL integrated development environment (IDE); the creation and modification of program definitions (using *SPARCL DE Factor Table*, *SPARCL DE Edit Object*, and *SPARCL DE Create Object*), the interpretation of programs (using *SPARCL Interpreter*), the display of programs (using *SPARCL Dis-*

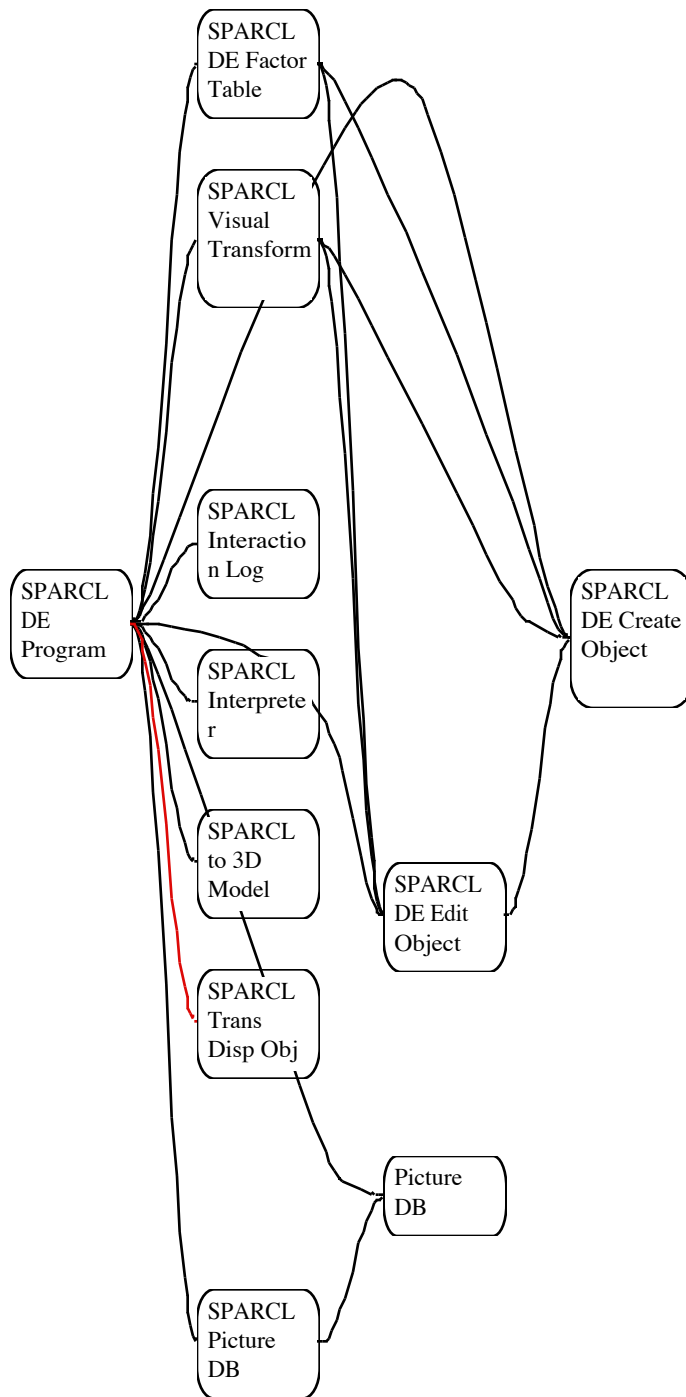


Figure 6. 4: Dependency graph for modules used by *SPARCL DE Program*.

play (not shown) and *SPARCL to 3D Model*), the maintenance of the program picture database (using *SPARCL Picture DB* and *Picture DB*), and converting between pro-

gram representations (*SPARCL Visual Transform* and *SPARCL Trans Disp Obj*).

Program representations. There are four program representations used internally by SPARCL. The primary representation is a collection of “display objects”. This is a geometry-independent definition of the visual representation of a SPARCL program, the “full, abstract” representation. The geometry (and other modelling information such as color and point-of-view) is added when a display mechanism processes the display objects to create a displayable representation. The *SPARCL Display* implements the predicates for creating a two-dimensional representation, and *SPARCL to 3D Model* implements the predicates for creating a three-dimensional representation. These two- and three-dimensional representations are not included among the four “internal” representations since SPARCL does not “read” them. SPARCL does use the two-dimensional representation to guide interactions with the user; that is, the user indicates what object she wants to work with by putting the mouse cursor over the two-dimensional representation of that object. We expect eventually to implement a similar use of the three-dimensional representation.

There is a simplified form of the full abstract representation. This simplified abstract representation is used by *SPARCL to 3D Model* instead of the full abstract representation. The *SPARCL Trans Disp Obj* module implements the translation from the full abstract representation to the simplified abstract representation.

The other two forms of SPARCL programs are the internal linear representation and the readable linear representation. The internal linear representation is used by the *SPARCL Interpreter* module; this is the form of a program that the interpreter evaluates. The readable linear representation is a simple syntactic variation on the internal linear representation that makes the linear representation a little easier for a person to read and write. The *SPARCL Visual Transform* module implements predicates for converting the internal linear representation of SPARCL (as used by the interpreter) into the full abstract representation. The internal linear representation and the full abstract representation are not semantically identical (i.e. they do not differ merely in syntax). The full abstract representation involves concepts that are not present in the internal linear form. Thus, converting between the internal linear representation and the full abstract representation involves program transformations. The internal linear representation does not have term tables and it does not have explicit coreference links. The transformation between these two representations creates semantically equivalent

programs, although the results of evaluating the two representations (if there were an interpreter for the full abstract representation) could differ in the presence of term tables versus equivalently structured sets.

Of these four representations, two are stored (using the *SPARCL DE Database* module) so that they are available across user interactions: the full abstract representation and the internal linear representation. (The concrete display representations are also available across user interactions.)

SPARCL DE Database. Figure 6. 12 shows *SPARCL DE Database* as one of the modules on which *SPARCL Display* depends. The *SPARCL DE Database* module is also used by the three main modules of Figure 6. 5. We have left it out of some dependency graphs (such as Figure 6. 5) to make these graphs easier to read. This *SPARCL DE Database* module manages most of the data that must be remembered/stored across user interactions (this includes data that must be remembered across invocations of the SPARCL application) excepting “file” data such as saved SPARCL programs, interaction logs, and system comments, and “picture” data that is attached to the windows in which the pictures reside. Because most modules use some cross-interaction data, most modules use *SPARCL DE Database*.

SPARCL DE Database stores data with two mechanisms. One is “direct”; the information being stored is “asserted” in special dynamic clauses using the PROLOG `assert/1` built-in. For instance, the display object definitions are stored in ‘`vpsl$object`’/3 clauses. (“vpsl” is a vestige of an earlier version of the SPARCL system; it stands for Visual Partition Structured Logic). The other is as “preferences”; the information is stored using the *Preference Utilities* module so that it can be saved and restored across invocations of SPARCL.

Some of the directly stored data is registered with the checkpoint facility, which is implemented in *SPARCL DE Database*. This provides a general mechanism to restore the state of the database to an earlier state, for the checkpointed part of the database. The checkpoint facility is used to implement the “undo” service. Each user interaction-invoked command that involves checkpointable data starts by making a checkpoint in the database. Thus, that interaction can be undone by rolling back the database to the checkpoint that began the command. The checkpoint facility must “roll back” the display to correspond to the rolled back database. The utilities used to accomplish this rolling back of the display are implemented in *SPARCL Display Util.*

This is why *SPARCL DE Database* uses *SPARCL Display Util*.

The other three modules that Figure 6. 12 shows *SPARCL DE Database* using are *SPARCL Interp Kernel*, *SPARCL Linear Transform*, and *SPARCL Pgm Window Name*. The *SPARCL Interp Kernel* implements some basic utilities related to the *SPARCL Interpreter* module that do not use any predicates in other modules and that are used by other modules that either are themselves used by *SPARCL Interpreter* or that are used by modules that do not otherwise need *SPARCL Interpreter*. There are several “kernel” modules in *SPARCL*. All of these share the property that the clauses in a “kernel” module do not use predicates defined in any other module.

SPARCL Linear Transform. The *SPARCL Linear Transform* module implements the conversion of the full abstract representation to its corresponding internal linear form, suitable for interpretation by *SPARCL Interpreter*. As mentioned above, this translation is a kind of program transformation. The full abstract representation can be considered a higher level program specification that *SPARCL Linear Transform* converts to the lower level internal linear representation. This transformation converts term tables in the full abstract representation into the equivalent sets. It converts the coreference links to uses of logic variables by introducing unification literals in the body of the associated clause. For example, if three sets corefer, then a variable is created for the coreference link and three unification literals are added to the body of the clause, one unifying each of the sets with the newly created variable. Any argument positions that held one of the terms in this coreference link has the new variable placed in it instead of the coreferenced term.

This module is used by *SPARCL DE Database* when storing a “clause” display object. The corresponding linear form of the clause is also stored, so that the clause is “ready” for use by the next `execute_query`.

SPARCL DE Edit Object, SPARCL DE Create Object, and SPARCL DE Factor Table. The editing of the full abstract representation of *SPARCL* programs is supported primarily by three modules, *SPARCL DE Edit Object*, *SPARCL DE Create Object*, and *SPARCL Factor Table*. Some of the dependencies of these modules are shown in Figure 6. 5. The “object” in the names of two of these modules refers to the “display objects” of the full abstract representation of *SPARCL* programs. These objects are stored in an internal database. *SPARCL DE Create Object* implements the creation of

the various kinds of display objects.

SPARCL DE Edit Object implements the selection, modification, deletion, copying, and “pasting” of display objects. These basic editing operations are similar to those the user is familiar with from a text processing application, but with some interesting differences in copying and pasting. The

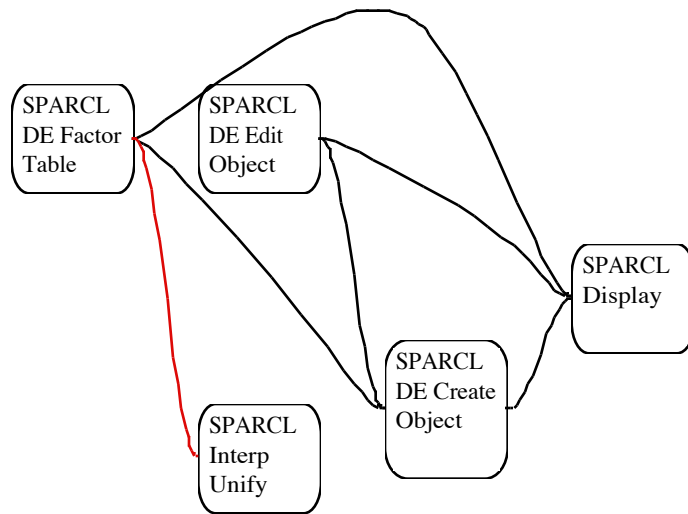


Figure 6.5: Selected module dependencies for *SPARCL DE Edit Object*, *SPARCL DE Create Object*, and *SPARCL DE Factor Table*.

copying of an object must also consider the coreference links that connect to objects inside of the object being copied. The copied object has a hyperedge linking all of the subobjects that are in the same hyperedge in the original object. Since a hyperedge may link subobjects as well as objects outside the object being copied, one may have hyperedge copies that are smaller than their originals. In pasting from “the clipboard” to a text selection, the clipboard text replaces the selected text. Pasting an object to a selected object puts the clipboard object *inside* the selected object. The system considers the types of the clipboard object and the selected object in deciding where the clipboard object goes in the selected object. For instance, if one has selected a clause display object and pastes an argument object into it, then the pasted object is added to the set of arguments. However, if the pasted object is a literal, then the pasted object is added to the set of literals.

SPARCL Factor Table implements the factoring of term tables. Factoring a term table involves finding some elements common to all of the rows of the table and extracting these common elements from the rows. The factored table has shortened rows and a “decoration” that shows the common elements. Factoring term tables is a complex process; it needs to operate in two different ways depending on whether the rows are N-tuples or functions.

The additional major modules on which these “editing” modules in Figure 6.5 rely are the *SPARCL Interp Unify* and *SPARCL Display* modules. The *SPARCL Interp*

Unify implements the “partition structured” unification algorithm of SPARCL, the general unification algorithm for all of the term types of the internal linear representation of SPARCL. The *SPARCL Display* module implements the two-dimensional representation algorithm for SPARCL. This module creates a two-dimensional display representing the “display objects” of the internal SPARCL display object database.

SPARCL Visual Transform. *SPARCL Visual Transform* converts the other way, from the internal linear representation to the full abstract representation. This conversion is similar to inferring a specification in going from the lower level form to the higher level form. This inference treats coreference simply. Multiple uses of the same variable in the linear representation creates a hyperedge among multiple variables in the abstract representation. We would like to make this more sophisticated by replacing some of the unify/2 literals in a clause by coreference links. This would be inverting the process that *SPARCL Linear Transform* uses.

The module dependency graph for selected modules used by *SPARCL Visual Transform* is shown in Figure 6. 6. Since it is this module’s task to create display objects for the full abstract representation from the internal linear representation of SPARCL programs, it is not surprising that this module uses *SPARCL DE Edit Object*, *SPARCL DE Create Object*, and *SPARCL DE Database*. *SPARCL Visual Transform* uses `display_new_program_elements/1` of *SPARCL Display* to show the newly created display objects.

SPARCL Interp Unify is used by the *SPARCL Visual Transform* module (as shown in Figure 6. 6) in the creation of term table display objects. The term that is the collection of rows is either an N-tuple or a set. If it isn’t an N-tuple, then “`partition_structured_unify(Term, set(Contents), c([], []), C1)`” is used to convert the term to a strict set (“`set(Contents)`”). Internally to SPARCL (i.e. not where the SPARCL programmer can see it) there are two representations of sets; strict (“`set(List)`”) and union (“`u(List)`”). Both kinds of sets are represented by lists of items. For a strict set, all of the items in the list are presumed to be distinct (i.e. not mutually unifiable). For a union set, some of the items in the list may be mutually unifiable. The term that is the collection rows for a table may be a union set or a set partitioning (“`p(Parts)`”, where each Part in the Parts list is a set-like term). Since strict sets are much easier to work with, we use `partition_structured_unify/4` to convert the term (if necessary) from a union set or set partitioning to a strict set. Since there can be many different ways to

convert a union set to a strict set, if the union set contains any unbound variables, the conversion to a strict set is backtrackable. The conversion to a term table should fail if there are multiple ways to do the conversion (i.e. due to strategically placed unbound variables), but it currently doesn't check for this. Thus, the term conversion may produce a term table of a particular configuration that requires imposing some constraints on the linear form being converted.

SPARCL Readable Linear contains two utility predicates which are used for testing and debugging in *SPARCL Visual Transform*; `test_sparcl_program/3` and `psl_portray/2`. The `test_sparcl_program/3` predicate defines various SPARCL linear form programs associated with identifiers. Some of these “identified” programs are shown in Figure 6. 7.

The `psl_portray/2` predicate writes an internal linear form program in readable linear form. The internal linear form programs of Figure 6. 7 are shown in Figure 6. 8 as written out by `psl_portray/2`.

SPARCL Interaction

Log. The module dependency graph for the modules used by *SPARCL Interaction*

Log is shown in Figure 6. 9. This module uses *SPARCL DE Database* for determining what kind of interactions are allowed,

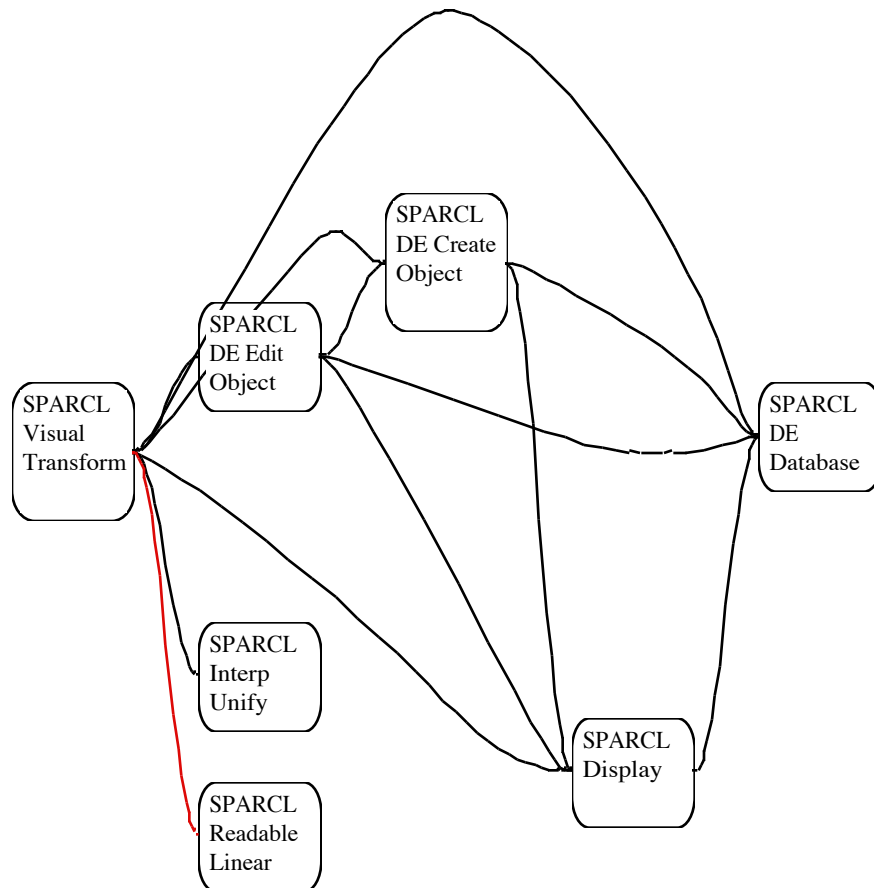


Figure 6. 6: Module dependency graph for selected modules used by *SPARCL Visual Transform*.

what the current “record interaction” mode is, and the current SPARCL user name. The allowed interaction type list is stored directly by *SPARCL DE Database* and the “record interaction” mode and the SPARCL user name are stored as preferences. The main predicate in *SPARCL Interaction Log* is “record_interaction(Action, Argument)”. An interaction is described as an action applied to one or more arguments. There are several specialized versions of record_interaction/2, where these specialized versions simply package information and invoke record_interaction/2:

```
member :
u([nt([nt([ur(m), X, p([u([X]), _])]),
      u([ur(success)])])])])

multiset1:
u([nt([nt([ur(data), X]),
      u([nt([ur(is), X,
            nt([ur(+), ur('1'), ur('4')])
            ])]))],
  nt([nt([ur(data), X]),
      u([nt([ur(is), X,
            nt([ur(+), ur('2'), ur('3')])
            ])])) ]])

multiset2:
u([nt([nt([ur(data), X]),
      u([ur(success)])]),
  nt([nt([ur(data), Y]),
      u([ur(success)])])])

union:
u([nt([nt([ur(u),
      p([X, Y]),
      p([Z, Y]),
      p([X, Y, Z])
      ]),
      u([ur(success)])])])])
```

Figure 6. 7: Some linear form programs defined by test_SPARCL_program/3.

record_menu_selection(MenuName, Option) for a selection of Option from menu MenuName.

record_window_activation(ProgramName) for the activation of the ProgramName window.

record_general_tool_activation(Window) for the activation of the “general tool” in Window.

record_general_tool_close_edit(NewSymbol, ProgramName) for the closure of the text edit box for the window for ProgramName, with NewSymbol as its contents at closure.

record_general_tool_popup(Object, MenuName, Option) for selecting Option from menu MenuName “popped up” over display object Object.

record_connect_tool_activate(Window) for the activation of the “connect tool” in Window.

record_connect_tool_link(Object1, Object2, ProgramName) for the linking of Object1 and Object2 in program ProgramName.

record_script_control_button(Button, NextStep, CurrentScript) for selecting Button of the script control window where NextStep is the next step of the current script CurrentScript.

SPARCL Halstead. The module dependency graph for *SPARCL Halstead* is shown in Figure 6. 10. This module implements various simple software measurements for SPARCL programs. These measurements are described in chapter 8 (“Objective Analy-

sis”). The measurements are done against the partial abstract representation produced by *SPARCL Trans Disp Obj*.

SPARCL Log Analysis. The *SPARCL Log Analysis* module is largely self-contained. It implements several tools for analyzing the interaction log and the system comment file. The results of most of these analyses are used in chapter 9 (“Usability Testing”).

SPARCL Script. The dependency graph for selected modules used by *SPARCL Script* is shown in

Figure 6. 11. This module uses several other modules of SPARCL to implement the script commands and there is a script command for every possible user interaction involving program editing or inspection. The *SPARCL Dialogs* module implements a variety of dialogs used by different parts of the SPARCL IDE. The primary dialog in *SPARCL Dialogs* for *SPARCL Script* is “script control”. This dialog gives the user control over stepping through scripts and choosing among available scripts. It also contains various pieces of information about the state of the current script (what the current script does, what the next step in the current script will be, how it will be accomplished, and a log of the steps taken up to this point). This dialog and the script mechanism in general are presented in chapter 9.

```
member:
U{((m => X => {[ U{X} , _ ]})
=> U{success})}

multiset1:
U{((data => X)
=> U{(is => X => (+ => '1' => '4'))}
),
((data => X)
=> U{(is => X => (+ => '2' => '3'))}
)}

multiset2:
U{((data => X) => U{success}),
((data => Y) => U{success})}

union:
U{((u
=> {[ X , Y ]}
=> {[ Z , Y ]}
=> {[X , Y , Z ]}
)
=> U{success})}
```

Figure 6. 8: Readable linear form of SPARCL programs in Figure 6. 7.

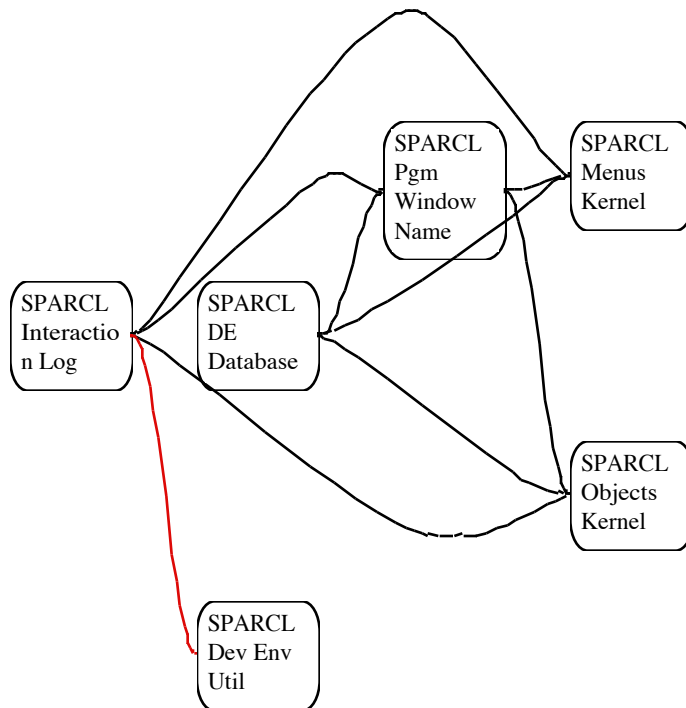


Figure 6. 9: Module dependency graph for the modules used by *SPARCL Interaction Log*



Figure 6. 10: Module dependency graph for *SPARCL Halstead*.

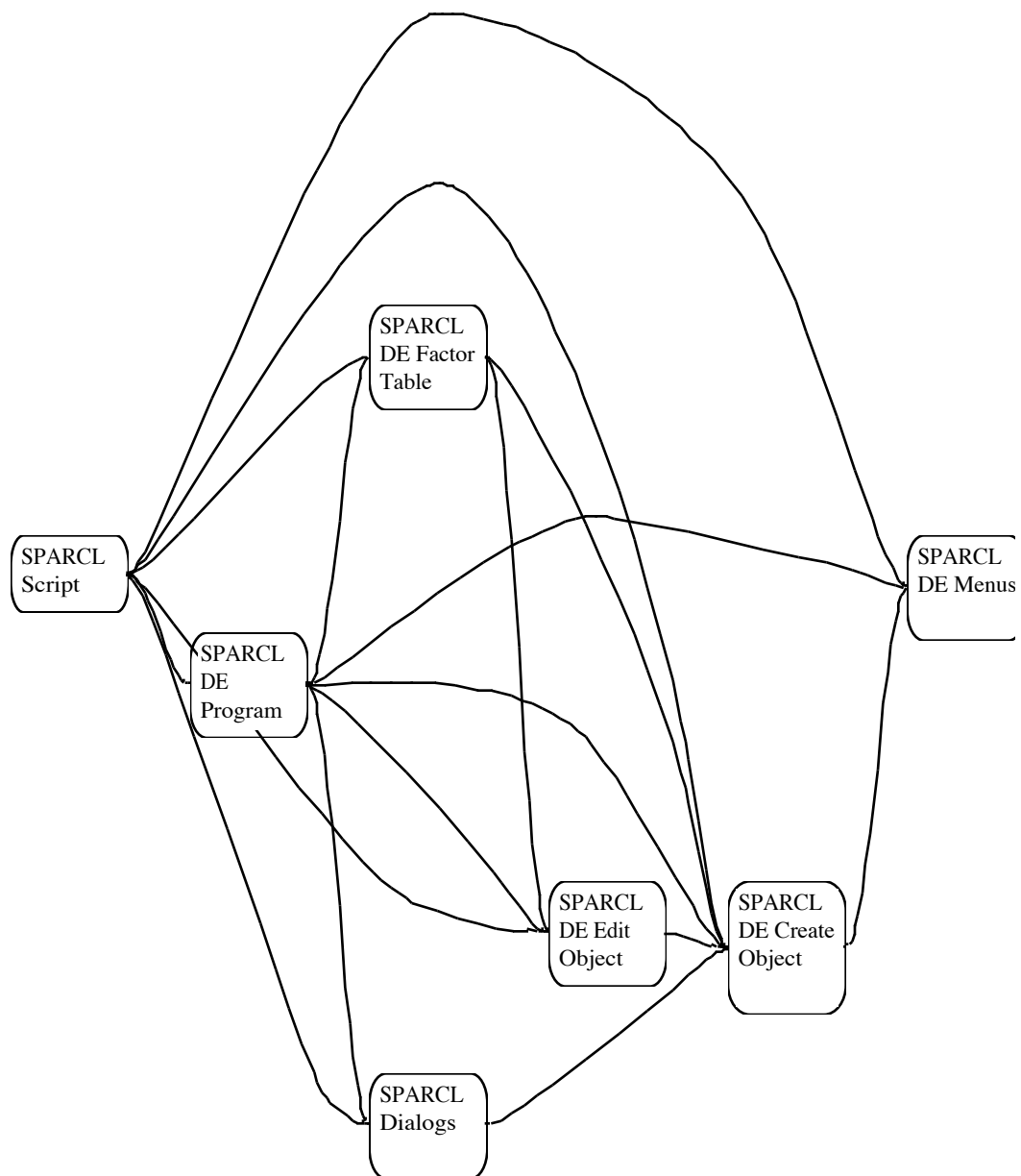


Figure 6. 11: Dependency graph of selected modules used by *SPARCL Script*.

The Display System.

The display system provides both two- and three-dimensional concrete representations of SPARCL programs. There are actually three distinct three-dimensional representations, one for each three-dimensional scene display technology which it supports (Apple Macintosh *QuickDraw3D*, SGI *OpenInventor*, and the POV Team's *POV-Ray*). These three-dimensional representations differ due to differing capabilities of the display technologies. The SPARCL display system implements the automated layout of the canonical representation to create the concrete representation. This layout algorithm is incremental for the two-dimensional representation, it is a batch process for the three-dimensional representation. When an already-displayed program is re-displayed by the two-dimensional algorithm, it limits the portion of the concrete representation which it re-constructs and it has ways in which it makes as small a change as possible to the existing layout to accommodate changes in the canonical display representation.

Aside from the incremental nature of the two-dimensional algorithm, the basic strategy employed by the layout algorithm is the same for all of the concrete representations (the two-dimensional representation and the three three-dimensional representations). A SPARCL program is displayed clause-by-clause. There are two phases to the layout of a clause: laying out the parts of the clause except for the coreference links, then laying out the coreference links. The clause and its parts as related by containment form a tree. The layout process in the first phase is to start at the root of the containment tree with a top/left position for that container (the clause itself), then to do a sizes-only layout (no actual display) of the contents to determine how to position elements of the container, then layout and display the elements of the container, then finally construct the display of the container itself. Each element in the tree of objects to be displayed is either a container type object or not. For container type objects, the above layout process is repeated and for non-container type objects they are directly displayed. There is an obvious inefficiency in the layout algorithm as described here in that it is executed twice, once to get sizes of elements and a second time to actually display them in position. This will be changed eventually to cache the results of the sizes-only layout pass in position-independent terms, allowing the display pass to retrieve the encached terms and simply display them.

The layout of items within a container is referred to as “sibling” layout. There are six different sibling layout algorithms used in the two-dimensional representation of SPARCL and three such algorithms used in the three-dimensional representation. The sibling layout algorithms for the three-dimensional representation are discussed in chapter 5 (“Three-dimensional Representation of SPARCL”). The six sibling layout algorithms for two-dimensional representations are for: (1) clause and literal arguments; (2) N-tuple elements; (3) partitioned set parts; (4) partitioned set part members and clauses in a program window; (5) clause literals; and, (6) term and fact tables.

(1) As is the case for the three-dimensional argument sibling layout algorithm, the two-dimensional sibling algorithm for arguments places the given arguments in order in a vertical stack (i.e. along the y-axis, which is vertical when seen from the default front view), with the first argument at the top of the stack. There is a “gutter” between the arguments.

(2) The sibling layout algorithm for N-tuples places the elements of an N-tuple in a horizontal row, with an “arrow” graphical element between elements.

(3) The partitioned set parts are stacked similarly to the arguments (algorithm 1). Differences are that there is no gutter between parts and all parts are forced to the same width (size in the X direction).

(4) Partitioned set part members and clauses in a program window use the “compact rectilinear” sibling layout algorithm described in chapter 5.

(5) The algorithm for clause literals places them in columns such that all of the columns have three literals, except for the right-most which may contain fewer than three literals.

(6) The table layout algorithm is given rows of items, where there is the same number of items in each row. This algorithm arranges these items such that items in the same row are placed horizontally left-to-right in the order given, and items in the same relative position in different rows are aligned vertically. The algorithm determines how deep (size in the Y direction) each row needs to be and how wide (size in the X direction) each column must be such that each item can be aligned with its row and column siblings.

The coreference link (or “hyperedge”) phase of the layout algorithm proceeds hyperedge-by-hyperedge. The hyperedge layout algorithm is given two kinds of information: a counter of how many hyperedges have been displayed so far in this

phase and the possible “attachment points” for each item which the hyperedge is to join. These attachment points are generated in the container phase of the layout algorithm, each linkable object of the clause has one or more possible attachment points associated with it. The hyperedge layout algorithm generates a collection of segments make up the concrete representation of the link. The display representation of a hyperedge which joins N items is composed of $2(N - 2) + 1$ segments. In the two-dimensional representation, hyperedges are assigned one of six colors. In the three-dimensional representations, there are 20 different colors. The relatively small number of colors in the two-dimensional representation is due to those colors being faster to display. There is no particular meaning to the assignment, it is done only to help the viewer distinguish between them. As discussed earlier, the segments are designed so that they join smoothly to form a single hyperedge representation. Segments which cross and are not part of the same hyperedge are likely to not cross “smoothly”, helping the viewer to recognize that they are parts of different hyperedges.

SPARCL Display. The module dependency graph for *SPARCL Display* is shown in Figure 6. 12. The four modules on which it depends are *SPARCL DE Database*, as mentioned above, *SPARCL Pgm Window Name*, *SPARCL Display Util*, and *curve_to_lines*. *SPARCL Pgm Window Name* implements the utilities for relating the name of a program and the name of the window with which that program is associated. The name of the window for a program includes the program name, program or term set indicator, and a “modified” indicator if the program has been modified since the last save of that program. This is a basic utility module which is used by many other modules. Generally, we do not show it in the dependency graphs displayed in this chapter in order to simplify those graphs. *SPARCL Display Util* implements various basic utilities for working with the display of a program. These utilities are extracted from *SPARCL Display* to allow other modules to use them. The *curve_to_lines* module implements the conversion of a Bézier curve to a polyline (a sequence of line segments defined by a sequence of points). This conversion is used to display coreference hyperedges.

Picture DB. Figure 6. 12 shows some of the dependency structure of the *Picture DB* module. This module relies on the *Containment Tree DB* module, which in turn uses the *Picture DB Utilities* module. The *Picture DB* module is implemented in this three

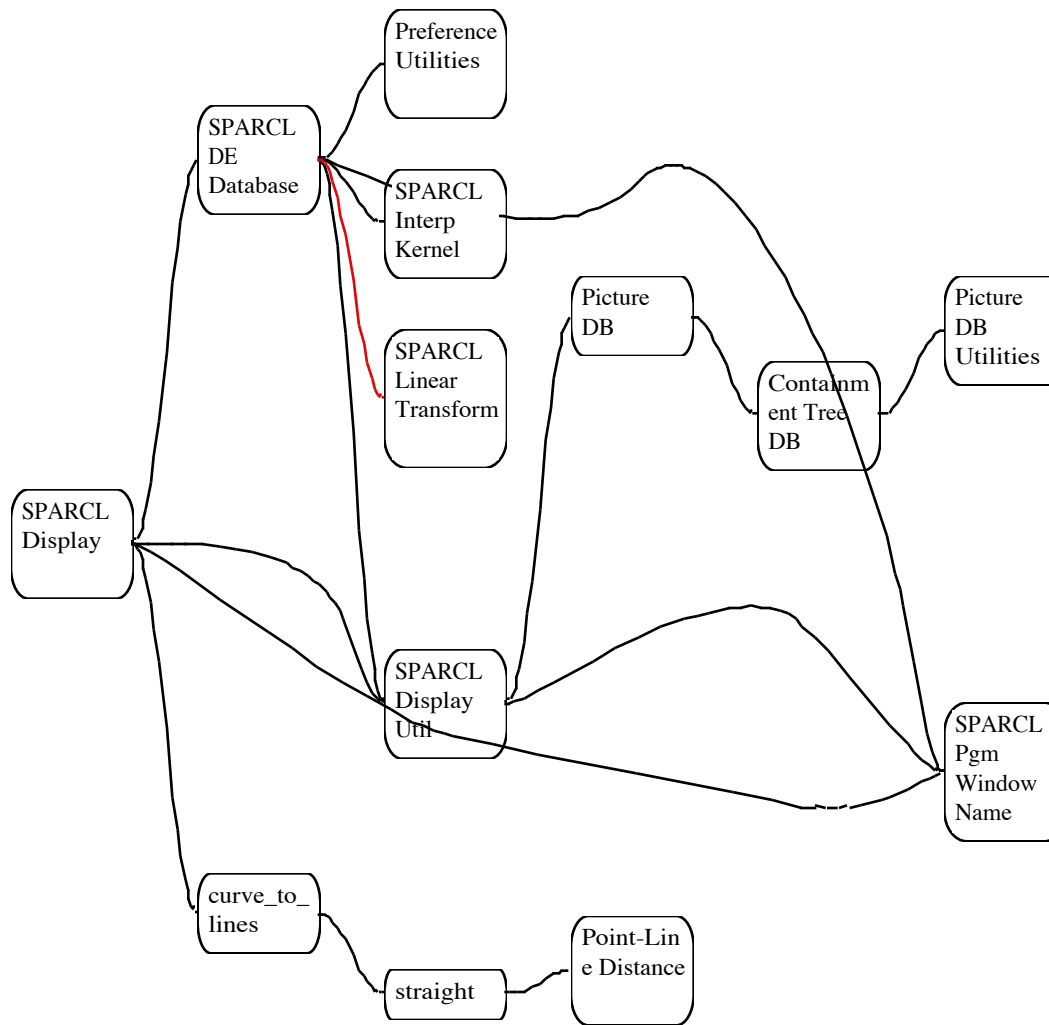


Figure 6. 12: Module dependency graph for *SPARCL Display*.

module form to make it easy to try different organizations of the picture database. Since the picture database is used in a search among potentially hundreds of objects that takes place between the depressing of the mouse button and the popping up of an appropriate menu, it is important that the picture database search mechanism be fast. We studied several different approaches to the organization of the picture database. Our *Containment Tree DB* module was much the fastest. Other picture database organizing modules were: *Four Dimensional X Tree DB*, *List Structured DB*, *Simple Tree Structured DB*, and *Tree Picture DB*.

SPARCL to 3D Model. The module dependency graph for selected modules used by

SPARCL to 3D Model is shown in Figure 6. 13. This module creates a specially formatted file that describes the geometry of a three-dimensional model of a SPARCL clause. The specially formatted file is used by *POV-Ray*, *QuickDraw3D*, or *OpenInventor* to create a rendering of the model. POV, QD3D, and Inventor each has its own model geometry file format. Also, these three systems have different modeling abilities. Thus, the formatting of the geometry file and the details of the specification of the geometry of the model depend on the system used to render to the model. There is a single program that

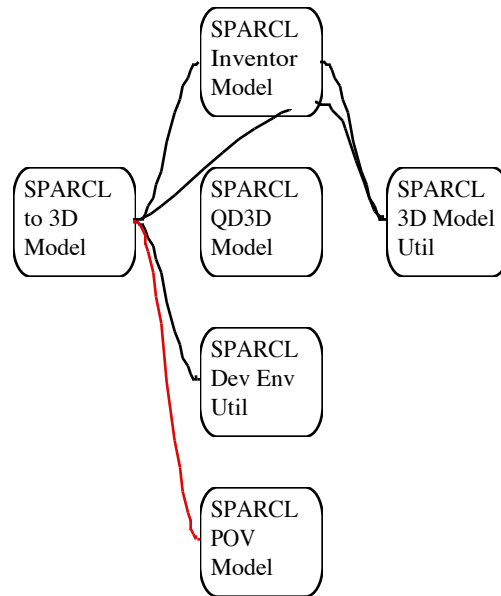


Figure 6. 13: module dependency graph for selected modules used by *SPARCL to 3D Model*.

converts SPARCL display objects to an “abstract” model. The basic abstract-model generating predicate is `model_id/4`, and the model-file-writing predicate is `write_clause_file/4`. These are invoked by the `write_clause_model/2` predicate of the *SPARCL DE Program* module:

```

write_clause_model(ClauseObject, ModelType) :-
    object_id(ClauseObject, ID),
    translate_display_clause_for_modeling(ID, ClauseObject, IOs, []),
    model_id(ID, IOs, ModelType, info(Model, Max, _)),
    clause_display_object_name(ClauseObject, Name),
    write_clause_file(Name, ModelType, Model, Max).

```

The `model_id/4` literal in the definition of `write_clause_model/2` has “`info(Model, Max, _)`” as its fourth argument. The `model_id/4` predicate actually generates an abstract model “info” term that contains three elements: the abstract model, the maximum size in each dimension of the model, and the “attachment points” for the model. (The attachment points are not used in the predicate. They are used when generating the hypertubes representing the coreference links; they are the possible locations where a hypertube segment may be attached to an element of the model.) The `write_clause_model/2` predicate writes the Model to a file with a name starting with

“Name”, using the *ModelType* file-writing predicate. The “Max” value is used in the POV file to position the camera and lights. The camera and light positioning is done automatically by QD3D and Inventor.

The type-specific file-writing predicate for pov is implemented by *SPARCL POV Model*, for qd3d by *SPARCL QD3D Model*, and for inventor by *SPARCL Inventor Model*. The different modeling systems use very different approaches for including common model information (such as the texture of a variable) and *Inventor* provides built-in support for character models while POV and QD3D do not. These differences require major differences in the file-writing predicates. To deal with characters in POV is not too difficult since POV comes with a file defining models for characters. However, QD3D doesn’t even provide this much help. So, we implemented QD3D models for all of the (upper-case) characters using the POV models as guides. This put the QD3D character modelling situation on the same footing as that of POV, we only needed to include references to these models to get characters into our QD3D models.

The Interpreter.

The interpreter implements the semantics of SPARCL. It interprets the internal linear representation of programs. The results of the interpreter are indicated by success or failure of the interpretation, and the binding of variables in the query being evaluated.

SPARCL Interpreter. The module dependency graph for selected modules used by *SPARCL Interpreter* is shown in Figure 6. 14. The bulk of the semantics of SPARCL are implemented in *SPARCL Interpreter* and *SPARCL Interp Unify*. The interpretation procedure, as described in chapter 3 (“Design Elements”), is primarily implemented in *SPARCL Interpreter* with the exception of the unification algorithm and constraint checking. The various SPARCL built-in predicates are implemented in *SPARCL Interpreter*. The unification algorithm and constraint checking is implemented in *SPARCL Interp Unify*. The *SPARCL Readable Linear* module supports an alternative form of the internal “linear” form of SPARCL programs. This “readable” linear form is used in the trace output of the SPARCL interpreter, as the form written by the *write/1* and *grounded_write/1* SPARCL built-in predicates, and in specifying a program “directly”

to the interpreter (rather than defining a program as a collection of display objects). Specifying a program directly to the interpreter is only done when debugging the SPARCL interpreter. The *Binary Tree* module is used by the *SPARCL Interpreter*, *SPARCL Readable Linear*, and *Tree Picture DB* modules. (The *Tree Picture DB* module is one of the picture database organizations compared to containment tree db, and not used.) It supports a binary tree structured term for reasonably fast access to “leaf” terms and fast maintenance of the tree. The *SPARCL DE Clause DB* module provides some simple utilities for working with the database of SPARCL program clauses. The *SPARCL Interpreter* implementation is largely independent of the rest of the SPARCL IDE; the access to the SPARCL program clauses is one of the few points where *SPARCL Interpreter* must rely on other parts of the *SPARCL DE Program* portion of the SPARCL IDE. The other two modules in the *SPARCL Interpreter* dependency graph of Figure 6. 14, *SPARCL Output Window* and *SPARCL Interp Kernel*, have already been discussed.

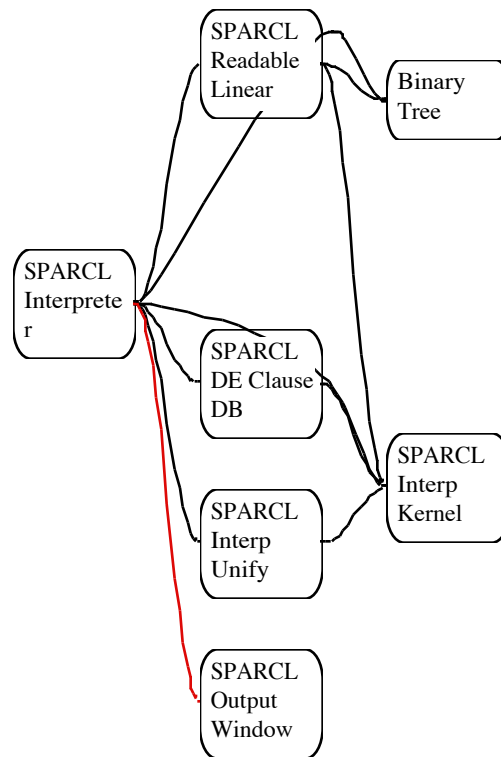


Figure 6. 14: Module dependency graph for selected modules used by *SPARCL Interpreter*.

A summary of the interpretation algorithm implemented in the *SPARCL Interpreter* module is discussed in chapter 3 (“Design Elements”) and in the tutorial in appendix 1 (“Tutorial Introduction to SPARCL”). There is also a metacircular interpreter definition given by the SPARCL program ‘Self Interpreter’/1 in chapter 7 (“Subjective Analysis”). The ‘Self Interpreter’/1 suppresses many of the details of the interpreter since it is written in SPARCL and thus implicitly uses the SPARCL interpreter and unifier. The internal linear form that the interpreter evaluates is built from the terms presented in Figure 6. 15.

A call graph for selected predicates used by *SPARCL_solve_goals/6* is shown in Figure 6. 16. This is the main predicate for the SPARCL interpreter. It takes a list of goals that all must be solved “simultaneously”, takes the first one in the list and processes it and the rest of the goals. This first goal can be delayed, in which case another of the goals is

$set(S)$	a strict set, where S is a list of the elements of the set. S may also be a variable, a list with an unbound tail, or a partition data structure. No two elements of S may be (or become) identical. $s([S1, \dots, Sn])$ is semantically equivalent to $p([s([S1]), s([S2]), \dots, s([Sn])])$.
$u(S)$	a union set, where S is a list of the elements of the set. S may also be a variable, a list with an unbound tail, or a partition data structure. Elements of S may be identical.
$p(P)$	a set partitioning, where P is a list of the parts of the partitioning. The elements of the partitioning may be variables, sets, or set partitionings. P may be an unbound variable. It may not be an “open” list (a list with an unbound variable as its tail).
$nt(NT)$	an N-tuple, where NT is a list of the elements of the N-tuple. An N-tuple is a special kind of set. The $nt(NT)$ data structure is used to improve efficiency and to make the interpreter easier to use (since the $nt(NT)$ structure is much more compact and therefore easier to read than the corresponding $set(S)$ structure).
$ur(U)$	an “ur” element where U is any PROLOG term (generally an atom).
$c(Vs, Ps)$	a constraint set, where Ps is a list of partition structures and Vs is a list of subterms of Ps . The Vs start out as (all of the) unbound variables in Ps . As partition-unifications are performed, some of the Vs may be bound to nonvariable terms. The Vs are used as an indicator that allows the constraint checker to skip some unnecessary checks.

Figure 6. 15: Term structures in the internal linear form.

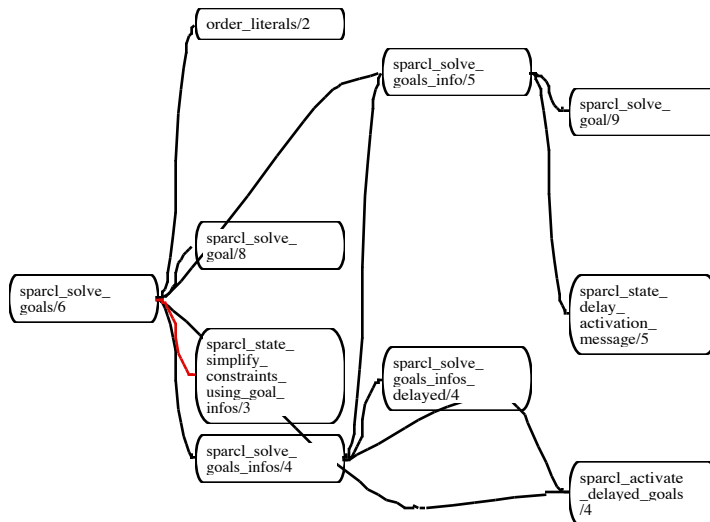


Figure 6. 16: Selective call graph for *SPARCL_solve_goals/6*.

selected for processing. In addition to a list of goals to solve and goals infos identifying all of the other goals to solve, there is a “state” of the interpreter that carries the active (unsolved) constraints and control information for the debugger.

The processing is done using a combination of `SPARCL_solve_goal/8` and `SPARCL_solve_goals_infos/4`. (The edge between `SPARCL_solve_goals_info/5` and `SPARCL_solve_goals/6` in Figure 6.16 is a

“back” edge; it completes a recursion loop.) The “goals info” structure is described in Figure 6.17. `SPARCL_solve_goals_infos/4` processes a list of “active” goals info structures and another list of delayed goals info structures. `SPARCL_solve_goals_infos/4` gives `SPARCL_solve_goal/8` the same delayed goals info list it received and an active goals infos list that is the same as that it received minus the selected goal.

A selective call graph for `SPARCL_solve_goal/8` is shown in Figure 6.18. `SPARCL_solve_goal/8` produces updated active and delayed goals info structure lists in one of three ways based on the goal that it processes. If the goal is solved by finding a matching “fact” (bodiless clause), then the active and delayed goals info structure lists are returned unchanged, effectively removing the goal from the active goals infos. If the goal is solved by finding a matching “rule” (clause with a nonempty

Ancestors-BodyGoals

the Ancestors are the literal (goals) for which attempted solutions lead to instantiating a clause with body *BodyGoals*.

delay(ID, Goal, Ancestors)

the literal *Goal* was delayed at some earlier point in the operation of the interpreter and then “activated” (found to no longer satisfy any delay specification). This literal (*Goal*) was given the unique identifier *ID* when it was delayed and its “proof tree parents” are *Ancestors*.

Figure 6.17: Goals info structure forms.

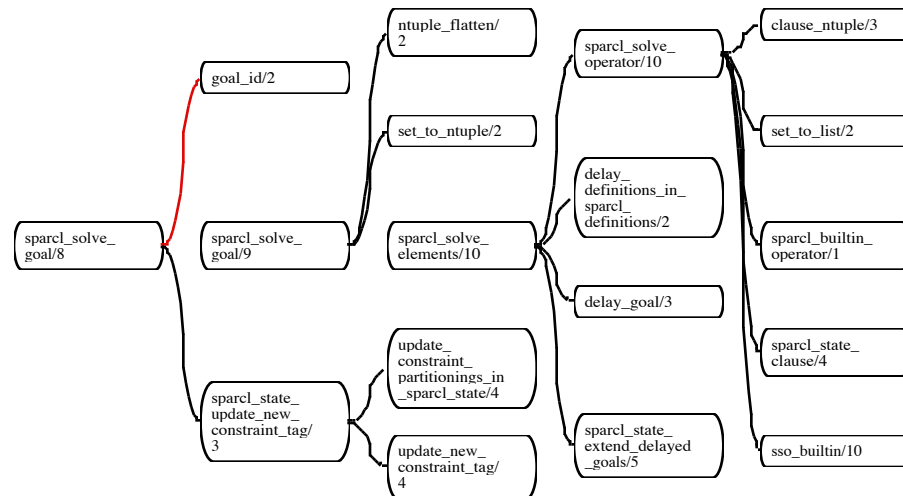


Figure 6.18: Selective call graph for `sparcl_solve_goal/8`.

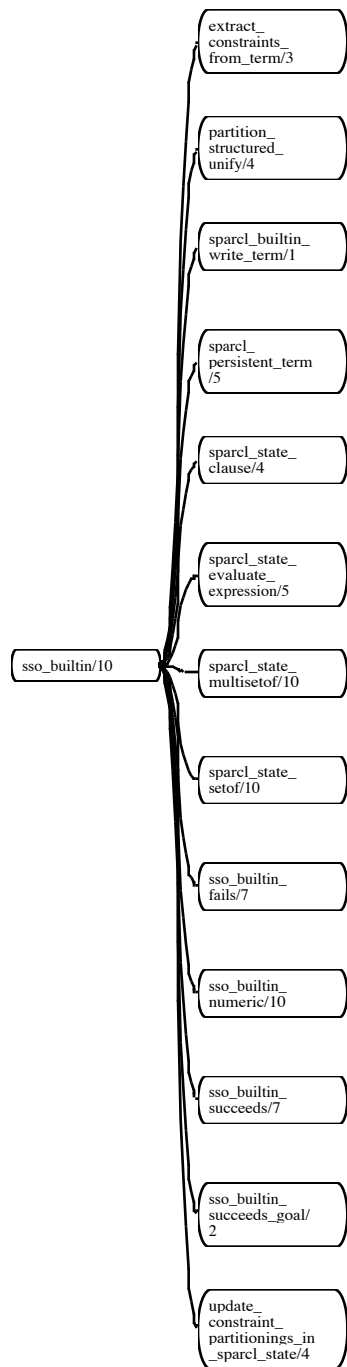


Figure 6.19: selective call graph for sso_builtin/6.

body), then the active goals infos list is extended (at the “front”) with a new goals info structure that holds the matched clause body. Again, the goal given to SPARCL_solve_goal/8 is “removed” from the active goals infos, but the active goals infos is extended with the body goals of the matched clause. Essentially, the given goal is replaced by the matched clause’s body goals. The remaining case is where the given goal is delayed. In this case, a delay-type goals info structure is created for the delayed goal and this structure is added to the delayed goals infos list. Again, as in the other two cases, the given goal is removed from the active goals infos list.

The main sequence of predicates used in the implementation of SPARCL_solve_goal/8 as shown in Figure 6.18 are SPARCL_solve_goal/9, SPARCL_solve_elements/10, and SPARCL_solve_operator/10. These various predicates do “bookkeeping” for the goal, such as assigning it a unique identifier and updating the constraints, finally having SPARCL_solve_operator/10 organize the solution of the goal. SPARCL_solve_operator/10 uses two different approaches to solving a goal, either the goal is determined to be a reference to a built-in predicate (by SPARCL_builtin_operator/1) in which case it is solved by sso_builtin/10, or the goal is solved by using the clause database using SPARCL_state_clause/4.

A selective call graph for sso_builtin/10 is shown in Figure 6.19. The several builtin predicates are generally solved by specialized predicates called by sso_builtin/10. The built-in SPARCL predicates with a

simple relationship to interpreter predicates are: setof/3 is solved by SPARCL_state_setof/10, multisetof/3 is solved by SPARCL_state_multisetof/10, fails/1 is solved by sso_builtin_fails/7, and succeeds/1 is solved by sso_builtin_succeeds/7.

A selective call graph for `SPARCL_state_clause/4` is shown in Figure 6. 20. The main predicate used here is `psli_clause/4`. The clause database is one of two forms, either it is a collection of clauses stored in PROLOG database clauses managed by *SPARCL DE Database* or it is stored as a binary tree term in the interpreter state. `psli_clause_db/2` is used to access the *SPARCL DE Database* form of the SPARCL clauses, given the goal and its database key. `sparcl_clause_key/2` determines the key to use in searching for matching clauses. `partition_structured_unify/4` is used to unify the goal with the head of candidate clauses. This

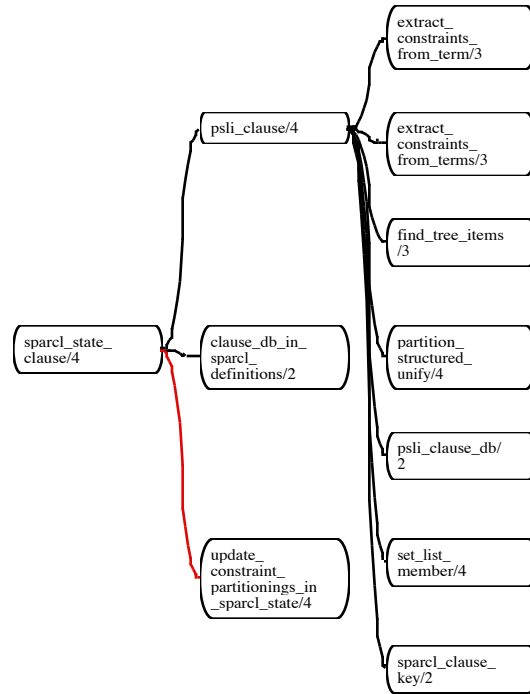


Figure 6. 20: Selective call graph for `SPARCL_state_clause/4`.

database search is an area of the interpreter that can be greatly sped up. It is currently simply a linear search through the clauses that match the key. The key helps restrict the search greatly, but more can be done. The SPARCL clause database is searched by `find_tree_items/3` when it is stored as a binary tree term in the interpreter state. The clause database can be stored in the interpreter state when debugging the SPARCL interpreter, allowing us to simplify the execution environment by getting rid of the editing and display systems of SPARCL.

SPARCL Interp Unify. The *SPARCL Interp Unify* module relies on few other modules, primarily the *SPARCL Interp Kernel* module. The central predicate of *SPARCL Interp Unify* module is `partition_structured_unify/4`. This predicate is used at two points in *SPARCL Interpreter*: it's called by `sso_builtin/10` (in Figure 6. 19) and `psli_clause/4` (in Figure 6. 20). This predicate is also used in *SPARCL DE Factor Table* (in Figure 6. 5) and *SPARCL Visual Transform* (in Figure 6. 6).

A selective call graph for `partition_structured_unify/4` is shown in Figure 6. 21. The major call path through this call graph is through `partition_unify/4`, atom-

ized_partition_unify/4, and unify_atomized_partition_elements/5. There are several special cases that partition_structured_unify/4 handles. The fully general unification algorithm is invoked by calling partition_unify/4. The other major case for unification is the unification of two N-tuples. This is handled by ntuple_elements_unify/4. The partition_unify/4 predicate prepares the terms for unification by “atomizing” them. atomized_partition_unify/4 does constraint processing and invokes the core of the unification algorithm as implemented by unify_atomized_partition_elements/5.

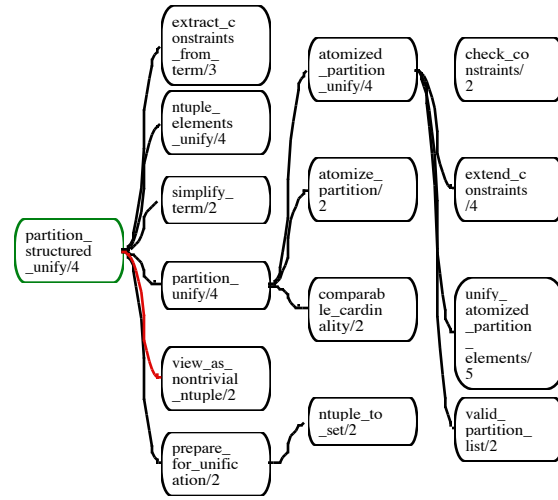


Figure 6.21: Selective call graph for partition_structured_unify/4.

A selective call graph for unify_atomized_partition_elements/5 is shown in Figure 6.22. The main sequence of predicate calls that this predicate uses to unify two atomized set partitioning terms is: unify_apes/5, unify_ape/4, unify_appl/7 or unify_aps/6, and finally a recursive invocation of partition_structured_unify/4.

The unify_apes/5 predicate unifies two atomized partition element lists. The unify_ape/7 predicate unifies two atomized partition elements. unify_ape/7 unifies its two given terms in two different ways, either the first term is a (strict) set or it is a set partitioning. The second argument

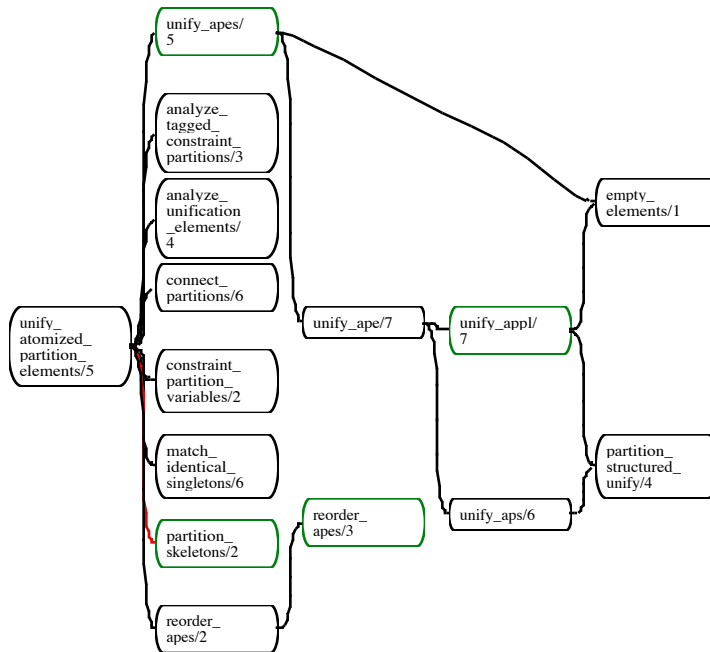


Figure 6.22: Selective call graph for unify_atomized_partition_elements/5.

may be either of these kinds of terms in either case. If the first argument is a set partitioning, then `unify_appl/7` is called. This unifies an **atomized** **partition** element set **partitioning**'s parts list with the second term. If the first argument is a set, then `unify_aps/6` is used. This unifies an **atomized** **partition** **set**'s element list with the second term.

Since the element lists which are used to call `unify_aps/5` have been “atomized” (which flattens sub-partitions), the only partitioned sets present as elements when `unify_aps/6` is called are those which were introduced via `partition_skeletons/2` (which was called by `unify_atomized_partition_elements/5`). These “introduced” partitioned sets only have unbound elements. Further, these partition skeletons have already been connected to all of the partition skeletons in the “other” element list and these connected partition skeleton elements have been removed, so the remaining partition skeleton elements are to be bound with sets in the “other” element list, or bound to empty sets.

So, consider that the first argument of `unify_aps/6` (call it “E1”) is a singleton set. Let the second argument be “E2”. If E2 is a partition of more than one element, let E2a be one element and E2r be the other elements, then E1 is unified with E2a and replace `p(E2)` with `p(E2r)` in the Elements2 list still to be unified. Otherwise, E2 is either a partition of one element or a set (of one element) and E1 is unified with E2. The unification rules that `unify_aps/6` implements as they are given in chapter 4 (“Partitioned Set Unification”) are:

$$\frac{\left\{ \text{ptn}(\{\{R\}\} \cup P) \doteq \text{ptn}(\{\text{ptn}(\{X\} \cup S)\} \cup Q) \right\} \cup \Lambda; \Gamma; V; \Sigma}{\sigma / \left(\left\{ \text{ptn}(P) \doteq \text{ptn}(\{\text{ptn}(S)\} \cup Q) \right\} \cup \Lambda \right); \sigma / \Gamma; V; \Sigma \text{ } \circ \sigma}$$

Rule 12

where $\text{var}(X) \wedge \sigma = \{X/R\}$

$$\wedge \left(\text{ptn}(\{\{R\}\} \cup P) \doteq \text{ptn}(\{\text{ptn}(\{X\} \cup S)\} \cup Q) \right) \notin \Lambda$$

$$\frac{\left\{ \text{ptn}(\{\{R\}\} \cup P) \doteq \text{ptn}(\{\text{ptn}(\{\{X\}\} \cup S)\} \cup Q) \right\} \cup \Lambda; \Gamma; V; \Sigma}{\left\{ R \doteq X, \text{ptn}(P) \doteq \text{ptn}(\{\text{ptn}(S)\} \cup Q) \right\} \cup \Lambda; \Gamma; V; \Sigma}$$

Rule 13

where $\left(\text{ptn}(\{\{R\}\} \cup P) \doteq \text{ptn}(\{\text{ptn}(\{\{X\}\} \cup S)\} \cup Q) \right) \notin \Lambda$

$$\frac{\left\{ \text{ptn}(\{\{R\}\} \cup P) \doteq \text{ptn}(\{\{X\}\} \cup Q) \right\} \cup \Lambda; \Gamma; V; \Sigma}{\left\{ R \doteq X, \text{ptn}(P) \doteq \text{ptn}(Q) \right\} \cup \Lambda; \Gamma; V; \Sigma} \quad \text{Rule 14}$$

where $\left(\text{ptn}(\{\{R\}\} \cup P) \doteq \text{ptn}(\{\{X\}\} \cup Q) \right) \notin \Lambda$

There is an additional clause for `unify_aps/6` that handles a special cases of Rule 12 and Rule 13 where S is empty. $\{R\}$ is our E1 and $\{X\}$ is our E2 or E2a.

There are two cases for `unify_appl/5`, either the elements are empty or they are not. In the second case, the first element is unified with an empty set or with some element of the second list. The unification rules that `unify_appl/5` implements as they are given in chapter 4 (“Partitioned Set Unification”) are:

$$\frac{\left\{ \text{ptn}(\{\text{ptn}(\{\{R\} \cup Y\})\} \cup P) \doteq \text{ptn}(Q) \right\} \cup \Lambda; \Gamma; V; \Sigma}{\left\{ R \doteq \emptyset, \text{ptn}(\{\text{ptn}(Y)\} \cup P) \doteq \text{ptn}(Q) \right\} \cup \Lambda; \Gamma; V; \Sigma} \quad \text{Rule 15}$$

where $\left(\text{ptn}(\{\text{ptn}(\{\{R\} \cup Y\})\} \cup P) \doteq \text{ptn}(Q) \right) \notin \Lambda$

$$\frac{\left\{ \text{ptn}(\{\text{ptn}(\{\{R\} \cup Y\})\} \cup P) \doteq \text{ptn}(\{Z\} \cup Q) \right\} \cup \Lambda; \Gamma; V; \Sigma}{\left\{ R \doteq Z, \text{ptn}(\{\text{ptn}(Y)\} \cup P) \doteq \text{ptn}(Q) \right\} \cup \Lambda; \Gamma; V; \Sigma} \quad \text{Rule 16}$$

where $\left(\text{ptn}(\{\text{ptn}(\{\{R\} \cup Y\})\} \cup P) \doteq \text{ptn}(\{Z\} \cup Q) \right) \notin \Lambda$

A selective call graph for `ntuple_elements_unify/4` is shown in Figure 6. 23. This predicate puts the two N-tuples to be unified into a canonical form and then the corresponding elements of the two canonical form N-tuples are unified.

Discussion.

In this section we provide a summary of the presentation of the implementation and an assessment of the implementation.

Summary. The SPARCL implementation is functionally divided into

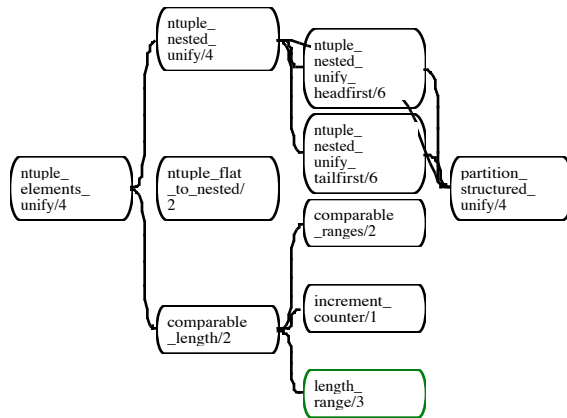


Figure 6. 23: Selected call graph for `ntuple_elements_unify/4`.

three major parts: the interpreter, the display system, and the editing system. The interpreter interprets an internal form of SPARCL. The editing system handles all of the interactions with the user and it maintains the canonical display representation of the program. The display system generates a concrete representation from the canonical display representation. To evaluate a query, the editing system converts the canonical display representation into the internal form, invokes the interpreter, converts the resulting internal form back to the canonical display representation, and finally uses the display system to present this result to the user.

The SPARCL application is implemented in *LPA MacProlog32* to run on an Apple Macintosh with MacOS 7.1 or greater. The source is divided into 62 modules varying in size from 2312 logical source lines (Halstead volume of 154419) to 71 logical source lines (Halstead volume of 210). The median logical source lines is about 350 and the median volume is 13353. The total logical source lines is 28650.

The main module is *SPARCL Dev Env*. Other crucial modules are: *SPARCL Display*, which presents the two-dimensional representation of SPARCL programs; *SPARCL to 3D Model*, which creates a file that defines the three-dimensional representation of SPARCL programs; *SPARCL Interpreter*, which interprets SPARCL programs; *SPARCL Interp Unify*, which implements the partitioned set unification algorithm; and the three modules *SPARCL DE Create Object*, *SPARCL DE Edit Object*, and *SPARCL Factor Table*, which together manage the creation and editing of SPARCL programs.

The Editing System. The “editing system” refers to all of the services of SPARCL except the program interpreter and the program representation display. It provides the program development environment, it manages the interactions with the user, maintains the canonical visual program representation, and provides various tools to aid in developing and maintaining SPARCL programs. The editing system implements the structured semantic editing approach discussed in chapter 3 (“Design Elements”). There are two kinds of interactions: through the menus on the main menu bar or through the program window object popup menus. Examples of these interactions are given in appendix 1 (“Tutorial Introduction to SPARCL”). The popup menus enforce the structured semantic editing paradigm. The editing system also provides the interaction logging, the integrated scripting/tutorial facility, and various interaction log analysis tools. The *SPARCL Dev Env* module pulls together all of the services of

SPARCL. The non-editing system services (program interpretation and display) are accessed through *SPARCL DE Program*.

There are four program representations used internally by SPARCL. The primary representation is a collection of “display objects”. This is a geometry-independent definition of the visual representation of a SPARCL program, the “full abstract” representation. A simplified form of the full abstract representation, the simplified abstract representation, is used when creating the 3D model of a clause instead of the full abstract representation. The other two forms of SPARCL programs are the internal linear representation and the readable linear representation. The internal linear representation is used by the interpreter. The readable linear representation is a linear textual representation of the internal linear representation that we use in debugging the interpreter, tracing the interpretation of SPARCL programs, and writing terms from SPARCL by the `write/1` and `grounded_write/1` builtin predicates. SPARCL programs are stored “across user interactions” using both the full abstract representation and the internal linear representation.

The full abstract representation is converted to the internal linear representation for the interpreter. The internal linear representation of the terms of the persistent term table and the results of a successful query are converted to the full abstract representation for presentation to the user. These two representations are not simply semantically equivalent, so these conversions are program transformations. The full abstract representation is a higher level program specification that *SPARCL Linear Transform* converts to the lower level internal linear representation. The interesting parts of the conversion involve reducing term tables and coreference links to the simpler mechanisms available in the internal linear representation. *SPARCL Visual Transform* converts the other way, from the internal linear representation to the full abstract representation. This conversion is similar to inferring a specification in going from the lower level form to the higher level form. This inference treats coreference simply. Multiple uses of the same variable in the linear representation creates a hyper-edge among multiple variables in the abstract representation. We would like to make this more sophisticated by replacing some of the `unify/2` literals in a clause by coreference links. This would be inverting the process that *SPARCL Linear Transform* uses.

The *SPARCL DE Database* module manages most of the data that must be remembered across user interactions (this includes data that must be remembered

across invocations of the SPARCL application) excepting “file” data such as saved SPARCL programs, interaction logs, and system comments, and “picture” data that is attached to the windows in which the pictures reside. Because most modules use some cross-interaction data, most modules use *SPARCL DE Database*.

SPARCL DE Database stores data with two mechanisms: direct and preferences. Some of the directly stored data is registered with the checkpoint facility, which is implemented in *SPARCL DE Database*. This provides a general mechanism to restore the state of the database to an earlier state, for the checkpointed part of the database. The checkpoint facility is used to implement the “undo” service. Data that is stored as preferences is remembered across invocations of SPARCL.

The editing system includes the basic editing operations of selection, copying, pasting, and deletion for display objects similar to those the user is familiar with from a text processing application, but with some interesting differences in copying and pasting. The copying of an object copies the coreference links that connect to objects inside of the object being copied. This may produce hyperedge copies that are smaller than their originals. When pasting from “the clipboard” to a text selection, the clipboard text replaces the selected text. In contrast, pasting a the clipboard object to a selected object puts the clipboard object *inside* the selected object. SPARCL determines where the pasted object goes inside the selected object based on the types of the two objects, this is not specified by the user.

The editing system also factors (and expands) term tables at the user’s request. This is a complex process; it needs to operate in two different ways depending on whether the rows are N-tuples or functions.

The Display System. The display system provides both two- and three-dimensional concrete representations of SPARCL programs. There are actually three distinct three-dimensional representations, each based on a different display technology. The SPARCL display system implements the automated layout of the canonical representation to create the concrete representation. This layout algorithm is incremental for the two-dimensional representation, it is a batch process for the three-dimensional representation. When an already-displayed program is re-displayed by the two-dimensional algorithm, it limits the portion of the concrete representation which it re-constructs and it has ways in which it makes as small a change as possible to the existing layout to accommodate changes in the canonical display representation.

Aside from the incremental nature of the two-dimensional algorithm, the basic strategy employed by the layout algorithm is the same for all of the concrete representations (the two-dimensional representation and the three three-dimensional representations). However, the implementations of the 2D and 3D layout algorithms are entirely distinct. We would like to combine them in the future.

The *Picture DB* module implements the picture database. The picture database is used by the editing system in deciding which type of object the mouse is over when the user depresses the mouse button. The object type determines the popup menu to present to the user. Since the picture database is frequently used in a search among potentially hundreds of objects that takes place between the depressing of the mouse button and the popping up of an appropriate menu, it is important that the picture database search mechanism be fast. We studied several different approaches to the organization of the picture database. Our *Containment Tree DB* module was much the fastest.

A 3D model of a clause is generated in two steps. First, the “abstract” 3D model (specialized for one of the three modelling systems; POV, QD3D, or Inventor) is generated using a common implementation. This common implementation handles all of the topology and geometry calculations. Second, the a 3D-model-file-writing system is invoked that is specific to the target mechanism. One of the ways in which these file-writing mechanisms most diverge is their handling of references to common or predefined aspects of models. These aspects can be just predefined constants, or they can be complex models such as those for characters.

The Interpreter. The central algorithm implemented by this interpreter is given in chapter 3 (“Design Elements”) and in appendix 1 (“Tutorial Introduction to SPARCL”). The interpreter uses a list of “goals info” structures to keep track of information about the goals to be solved. Goals are grouped in the same regular goals info if they are literal instantiations from the same clause body instantiation. Each goals info structure also records the ancestors of the goals in that goals info structure, the ancestors being goals in the proof tree path leading to the clause instantiation that “created” the goals in that goals info structure. The list of goals infos is managed as a stack, implementing a depth-first search through the proof tree if there are no delayed goals. A delayed goal is recorded in a special goals info structure that records the unique identifier the goal was given, the goal being delayed, and the ancestors of that goal. The goals info

structure for a delayed goal is added to the *end* of the goals infos list when that goal “becomes active” again.

The interpreter solves a goal in one of two ways, either the goal uses a builtin predicate or it doesn’t. There is special code to handle each of the builtin predicates, the most complex of these being *setof/3* and *multisetof/3*. When a goal is not a builtin, then SPARCL solves it using the SPARCL clause database. We divide the search of this database up by clause predicate name, then do a linear search through the clause heads with the same predicate name, trying to unify each head to the given goal. This is an area where substantial performance improvements are possible by an indexing scheme for the clauses based on their arguments, and by specializing the unification procedure for each clause head.

The unification algorithm is formalized and analyzed in chapter 4 (“Partitioned Set Unification”). Unification is a central procedure in SPARCL; it’s used in two places in *SPARCL Interpreter*, and once each in *SPARCL DE Factor Table* (in Figure 6. 5) and *SPARCL Visual Transform* (in Figure 6. 6). The rules given in the formalization of partitioned set unification in chapter 4 have some close correspondences with the implementation of unification. Rules 11 through 14 are discussed in some detail. The implementation of unification deviates from the formalization in providing for special cases of the rules and in implementing N-tuple unification.

Assessment. The implementation of SPARCL is extensive. It provides a wide variety of services: event-driven human-computer interaction management, database management (including checkpoint and rollback), structured-program editing, programming-in-the-large, program transformation, program interpretation, (partitioned set) unification, two-dimensional and three-dimensional representation with automated layout, three-dimensional geometric modelling for multiple modelling/rendering systems, computer-based training, activity logging, and software measurement.

We believe that LPA’s MACPROLOG32 has been a great help in creating SPARCL. The wide variety of services listed above is implemented in 3667 procedures spanning 28650 logical source lines, with a total token count 191012. The large number of (obviously small) procedures is normal for PROLOG, implementations of which are usually optimized for executing procedures such that there is almost no overhead introduced by a procedure call compared to “inline” code. It has an extensive “high level” integrated graphics system that connects with the Apple Macintosh QuickDraw

Toolbox utilities that greatly simplifies many aspects of working with graphics. Similarly, its menu, dialog, and window facilities provide a very convenient “high level” wrapper for the standard Macintosh Toolbox utilities.

One awkwardness we encountered was in the window handling. MacProlog32 identifies a window by its name, which is the same as its title. We associate a window in SPARCL with each SPARCL program. The SPARCL program name is extended to create the window name. The window name is modified with a check mark to indicate if the associated program has been changed since it was last saved. Thus, there is an unfortunately complex bit of code needed to keep track of the relationship between SPARCL programs and their windows. This is the job of the *SPARCL Pgm Window Name* module.

The other drawback to LPA’s MACPROLOG32 is the lack of any development library management tools, and the lack of programming-in-the-large support in general (e.g., there is no “make” facility for creating applications). We implemented our own library system to make up for this lack.

The performance of SPARCL overall is inadequate for use as a real programming language. The editing system is adequate and the display system is mostly adequate, although it can degrade seriously when there are a lot of clauses loaded. This is discussed in chapter 9 (“Usability Testing”) in the section on response time. The major performance problem is the interpreter. There are many things we can do to improve the performance of the interpreter. One common optimization is to compile to a byte-code or even assembler. However, there are several algorithmic improvements that we expect to investigate before we consider moving the run-time of SPARCL out of PROLOG. In a communication in the PROLOG news group (comp.lang.PROLOG), an implementor of a logic programming language noted that they had moved their run-time out of PROLOG and into a byte-code interpreter, and that he later regretted the change. The interpreter became much more complex, and therefore harder to change, and they no longer automatically benefited from improvements to PROLOG implementation performance without copying these improvements explicitly into their own system.

Chapter 7

Subjective Analytic Assessment of SPARCL

1. Introduction and the Classify Examples problem.

This chapter presents SPARCL programs which solve several programming problems. For the *classify_examples* problem we also present solutions in LISP and PROLOG. We explain the implementations, building on the introduction to SPARCL given in chapter 3 (“Design Elements”), and argue that the SPARCL solution is a comparatively more understandable approach. There is no measurement involved in this argument, but rather a reliance on the reader’s subjective assessment when allowed to inspect the program. The next chapter (Chapter 8) presents a more objective analytic assessment of these languages for selected programming problems (which include the *classify_examples* problem). Chapter 9 presents an empirical assessment of SPARCL based on limited user testing.

In this section we present the *classify_examples* problem and our solutions in all three languages. The *classify_examples* problem is part of the *ID3* machine-learning programming problem:

An “example” is a set of attribute name/value pairs. An example set is a set of examples all of which have the same set of attribute names. Given an example set and a classifying attribute name, find a classification of the examples according to their values on that attribute name.

The SPARCL solution of this problem demonstrates both the brevity of the SPARCL language and also the usefulness of its nonlinear representations (particularly tables). There are four different representations of sets in the SPARCL code shown here: partitioned sets, intensional sets, table of set of ntuples, and table of set of “functions”. The table of a set of ntuples shows each N-tuple in a row of the table, the order of the rows in the table isn’t meaningful. The table of a set of functions puts the domain values (which all of the functions share) in a header for the table and each row is the range values for a function. A function is a set of ordered pairs, the domain of the function being the first element of the pairs and the range being the second element of the pairs, and no two ordered pairs have the same first element.

The *Classify Examples/3* predicate for classifying examples is used in the implementation of the ID3 machine learning algorithm. The *Classify Examples Query/1* predicate is used to “query” the *Classify Examples/3* predicate. This query presents a table of examples to *Classify Examples/3*, which (when thus queried) returns a table of tables, one for each value of the ‘color’ attribute. These tables are shown in Figure 7.1. 1.

The *Classify Examples* program shown in Figure 7.1. 2 is implemented by a single predicate of a single clause. The third argument contains an intensional set with an empty body. The template of this intensional

set is a 2-tuple: a variable and another intensional set. The second intensional set is the set of all examples which share the same value on the given attribute. This “same” value is coreferenced by the variable which is the first element of the outer intensional set’s template. This inner intensional set contains a term which is a set within a (part of) a set. The inner set is a little darker than the outer set to make it easier to “read”. We can express this clause formally by:

Query:

color	size
blue	big
red	big
red	small

Classified examples:

blue	color	size
	blue	big
red	color	size
	red	big
	red	small

Figure 7.1. 1: Query and classified examples tables.

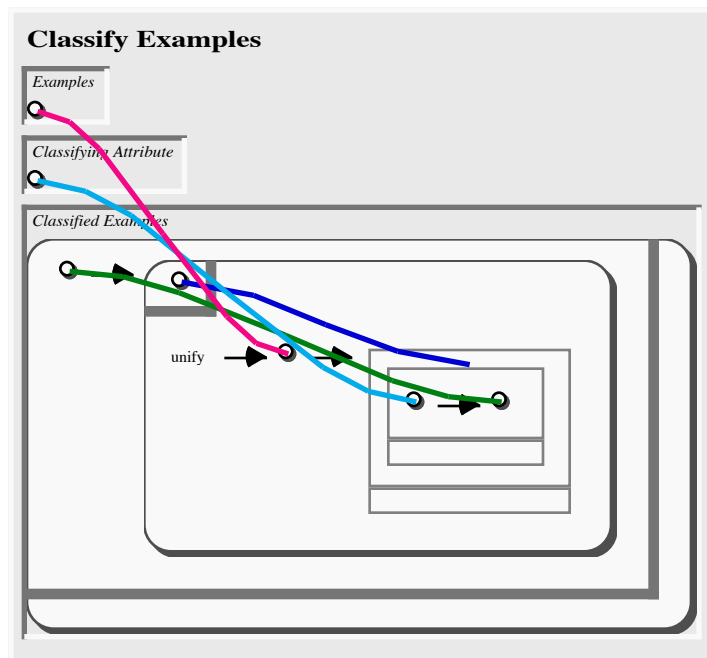


Figure 7.1. 2: *Classify Examples/3* program.

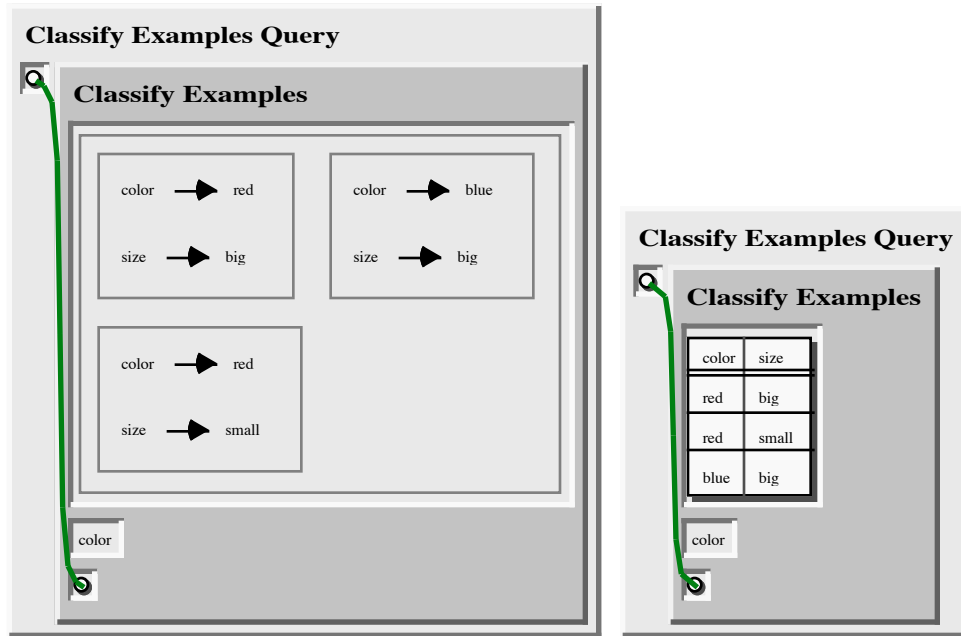


Figure 7.1. 3: Two forms of query for *Classify Examples/3* program.

$classify_example(X, Y, Z)$

$$\Leftrightarrow Z = \left\{ A \mid \exists p, q \left(A = \langle p, q \rangle \wedge q = \left\{ r \mid \langle Y, p \rangle \in r \wedge r \in X \right\} \wedge q \neq \emptyset \right) \right\}$$

This form of classify examples maps onto the SPARCL clause in a simple way: X is the set of examples to be classified, Y is the name of the classifying attribute, Z is set equal to the “outer” intensional set, and q is set equal to the “inner” intensional set. The template of the outer intensional set is A , which is set equal to the ordered pair of p and q , p is the classifying attribute value, and r is an example in the set of examples (X).

Two forms of a query clause are shown in Figure 7.1. 3. They have the same meaning; they differ in that one represents the examples to be classified using a set containing sets of N-tuples and the other represents these same examples using a function table. This clause is used to “query” the *Classify Examples* program with some test data; the table discussed above. The N-tuple (an ordered pair) shown in Figure 7.1. 4 is the result of running the above query and gives the classified examples as a table of tables. The programming environment determined that the classified examples set could (and should) be displayed in this fashion; the programmer made no specification about how to display this “classified examples” set.

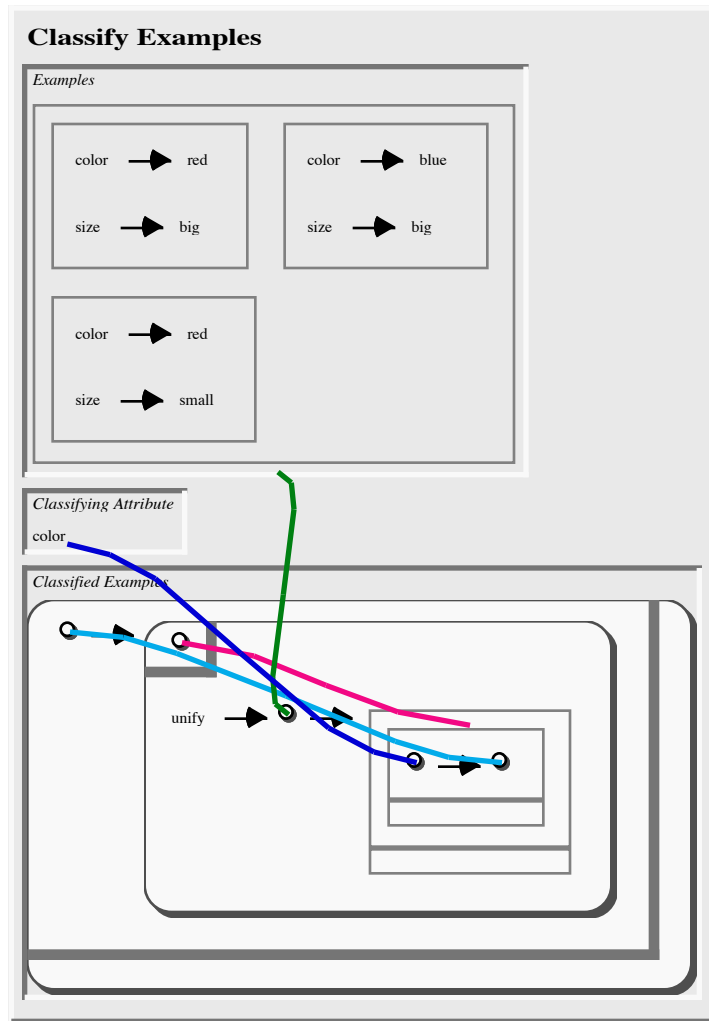


Figure 7.1. 5: ‘Classify Examples’/3 clause with query values in arguments.

There are some “expansions” of the ‘Classify Examples’/3 clause that give an idea of the approach SPARCL uses to solve this query. The ‘Classify Examples’/3 clause with its arguments instantiated to those of the literal in the query is shown in Figure 7.1. 5. This clause is further expanded by replacing the use of intensional sets with ‘setof’/3 and ‘setof’/4 literals. This is shown in Figure 7.1. 6. In this clause there is a

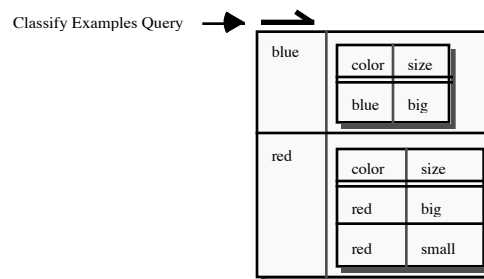


Figure 7.1. 4: Result 2-tuple from evaluating the query for the *Classify Examples*/3 program.

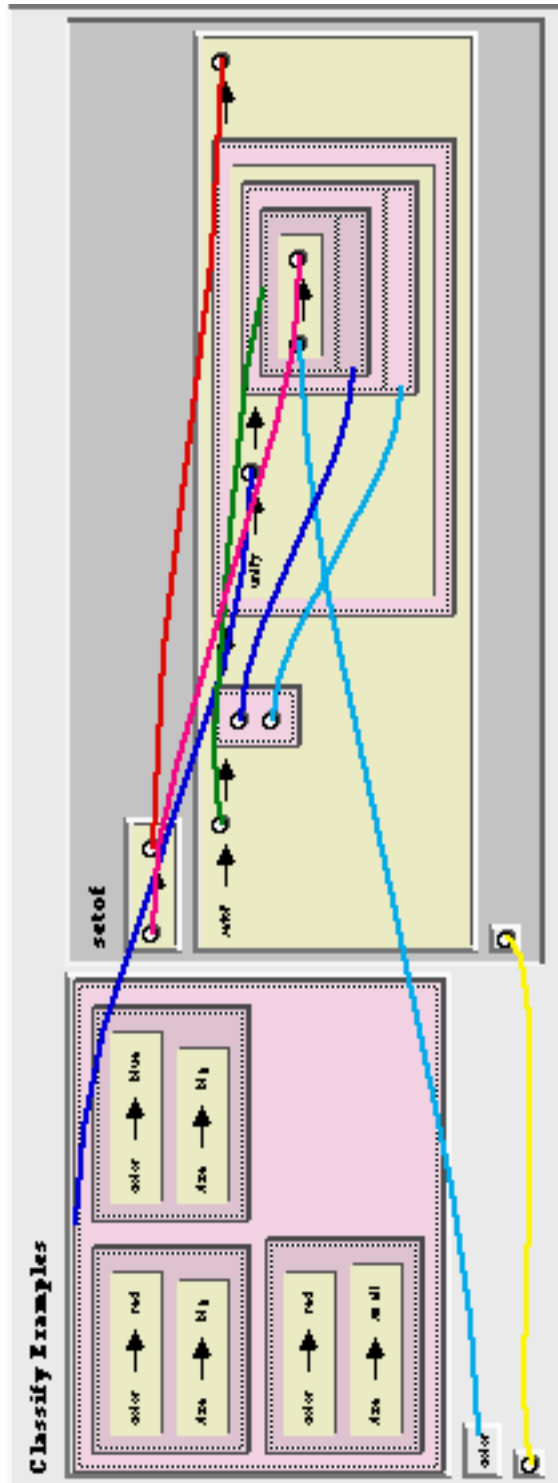


Figure 7.1. 6: ‘Classify Examples’/3 with query values and expanded intensional sets.

‘setof’/3 literal that corresponds to the “outer” intensional set. We expand this literal to a clause as shown in Figure 7.1. 7. This clause has one literal (represented by an N-tuple) in its second argument that corresponds to the “inner” intensional set. This inner ‘setof’ has four arguments. Its second argument (the third element of the N-tuple) is a set of two variables. This argument specifies existentially qualified variables: these variables (and the variables with which they corefer) can be bound differently for each successful evaluation of the body of the setof. In this case these variables are being used to existentially qualify the “rest of” parts of the two partitioned sets. If these parts weren’t existentially qualified then they would have to have the same value for every solution contributing to a set of solutions.

The ‘setof’/4 literal is shown as a clause in Figure 7.1. 8. A subtle but important aspect of this clause is the *unlinked* variable in the second argument of the ‘unify’/2 literal (represented as a 3-tuple) in the third argument of this ‘setof’/4 clause. This variable is unlinked in Figure 7.1. 8 because its coreference is outside of the ‘setof’/4 literal in Figure 7.1. 7. Since this is an unlinked variable and

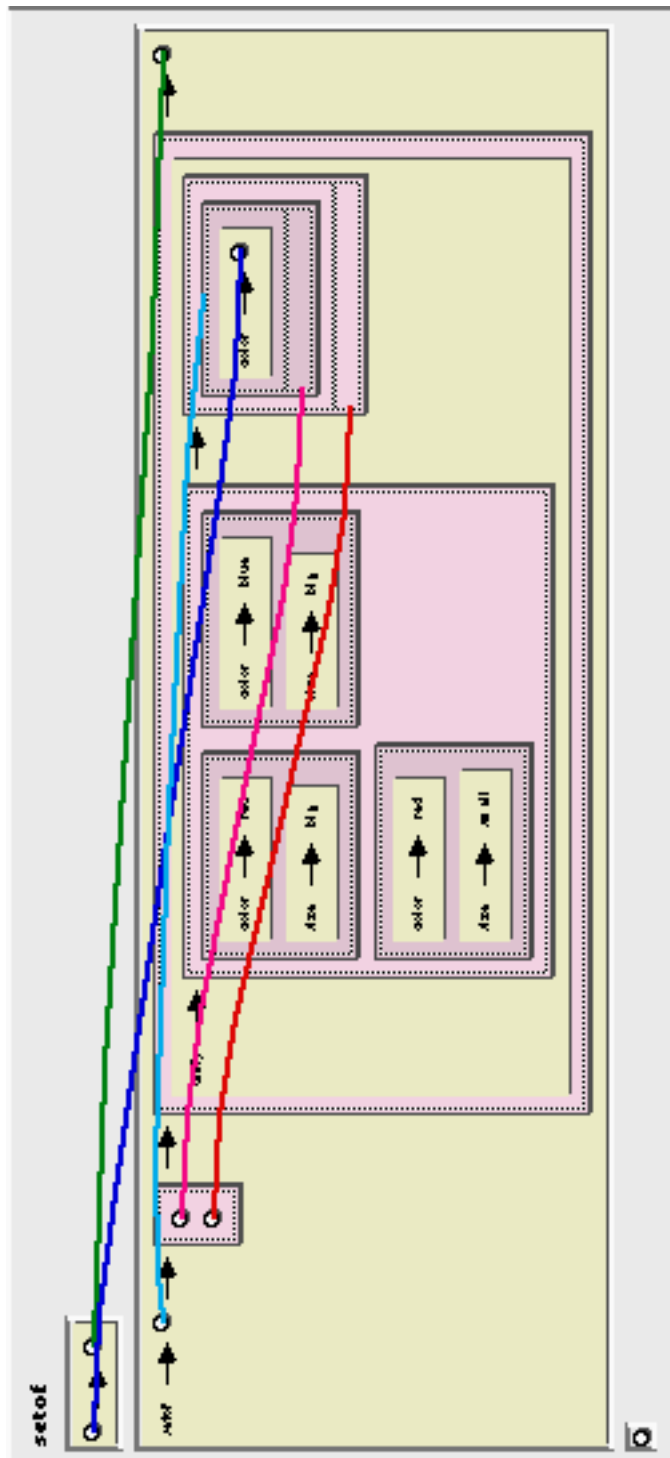


Figure 7.1. 7: 'setof'/3 expansion from 'Classify Examples Query'/1 evaluation.

it is not in the existentially qualified variables set of the second argument, it must have the same value for all of the solutions used to make the set of the fourth argument. In this example there are three choices for its first binding: 'red', 'red', and 'blue'. There is one choice for each of the examples in the example set. Thus, this inner 'setof'/4 produces three result sets when evaluated by the outer 'setof'/3, two of them identical (using the 'red' color value). Since two of them are identical, the outer 'setof'/3's result set has only two elements, one pairing for each color value of color value and examples containing that color value. This is the desired result.

This example demonstrates two of the services the intensional set representation provides when setting up the translation to the 'setof'/3 and 'setof'/4 literals: it determines how to nest uses of 'setof'/(3,4); and, it determines which

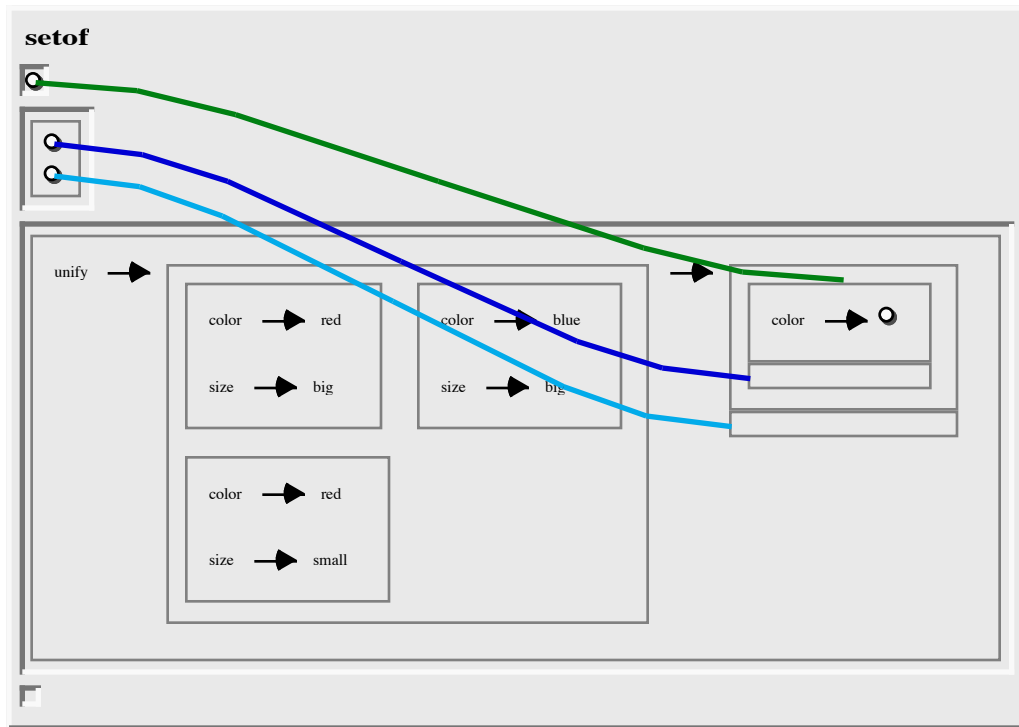


Figure 7.1. 8: ‘setof’/4 expansion from ‘Classify Examples Query’/1 evaluation.

variables need to be existentially qualified and in which ‘setof’ literals “scope”. This is one of the *program transformation* operations mentioned in chapter 6 (“Implementation”) that SPARCL performs in transforming the external form of a clause to its internal form.

The PROLOG implementation of the classify examples program.

A brief solution. The program shown in Figure 7.1. 9 is a maximally concise implementation in PROLOG of a solution to the *Classify Examples* programming problem.

A fast solution. Another implementation of a solution to the *Classify Examples* problem is shown in Figure 7.1. 10 and Figure 7.1. 11. (The *sort/2* and *member/2* procedures are assumed to be provided by the PROLOG implementation.) This implementation is much faster than the “brief” solution.

The *classify_examples/3* procedure uses *classify_examples1/3* to build a list of pairs, where each pair has a value in the `classifyingAttribute` as its first element and

```

classify_examples(Examples, ClassifyingAttribute, ClassifiedExamples) :-
    setof(V-Es,
        setof(XE,
            (member(XE, Examples),
             member(ClassifyingAttribute-V, XE)),
            Es
        ),
        ClassifiedExamples
    ).

```

Figure 7.1. 9: PROLOG “brief” *Classify Examples* solution.

an example from `Examples` which contains that value for the `ClassifyingAttribute`. These pairs are then sorted to put all of the same-valued pairs together. Finally, *classify_examples/3* uses *assemble_value_example_pairs/2* to convert this sorted list of pairs to a list of pairs where each pair has a value in the `ClassifyingAttribute` as its first element and *all* of the examples from `Examples` which contains that value for the `ClassifyingAttribute`.

The *assemble_value_example_pairs/2* procedure uses the *assemble_value_example_pairs/4* procedure, “priming” it with the value of the first input pair and providing the “tail” of the list of examples with that value. The *assemble_value_example_pairs/4* procedure “iterates” over the list of value-example pairs. For each such pair, if the value is the same as the previous value, then it adds the example to the list of examples for that value and passes on the tail of that list of examples to the next step. The tail of the output pair list is passed on unchanged (as it came in). If the value is not the same as the previous value, then it sets the tail of the

```

classify_examples(Examples, ClassifyingAttribute, ClassifiedExamples) :-
    classify_examples1(Examples, ClassifyingAttribute,
                      ValueExamplePairs),
    sort(ValueExamplePairs, SortedPairs),
    assemble_value_example_pairs(SortedPairs, ClassifiedExamples).

classify_examples1([], ClassifyingAttribute, []).

classify_examples1([Example|OtherExamples],
                  ClassifyingAttribute,
                  [Value-Example|OtherValueExamplePairs]) :-
    member(ClassifyingAttribute-Value, Example),
    classify_examples1(OtherExamples, ClassifyingAttribute,
                      OtherValueExamplePairs).

```

Figure 7.1. 10: PROLOG “fast” *Classify Examples* solution, part 1: *classify_examples/3* and *classify_examples1/3* predicates.

```

assemble_value_example_pairs([Value-Example|OtherPairs],
                             [Value-[Example|OtherValueExamples]
                              | OtherClassifiedExamples]
                             ) :-
    assemble_value_example_pairs(OtherPairs, Value,
                                OtherValueExamples,
                                OtherClassifiedExamples).

assemble_value_example_pairs([], _, [], []).

assemble_value_example_pairs([Value-Example|OtherPairs],
                             RefValue, ValueExamples,
                             ClassifiedExamples
                             ) :-
    (Value = RefValue
     -> ValueExamples = [Example|OtherValueExamples],
        ClassifiedExamples = OtherClassifiedExamples
    ;
    % For this case, Value  $\neq$  RefValue
    ValueExamples = [],
    ClassifiedExamples = [Value-[Example|OtherValueExamples]
                        | OtherClassifiedExamples]
    ),
    assemble_value_example_pairs(OtherPairs, Value,
                                OtherValueExamples,
                                OtherClassifiedExamples
                                ).

```

Test query:

```

:- classify_examples([[color-red, size-big],
                    [color-red, size-small],
                    [color-blue, size-big]
                    ],
                    color,
                    ClassifiedExamples).

```

Figure 7.1. 11: PROLOG Classify Examples solution, part 2: assemble_value_example_pairs/3, assemble_value_example_pairs/4, and classify_examples/3 query.

list of examples for the previous value to empty (it “closes” the previous value’s list of examples), it starts a new pair of value and examples list, and passes on the tail of this new examples list and the new tail of the output pair list. This continues until all of the pairs have been processed, when the tail of the previous value’s example list is closed and the tail of the output pair list is closed.

Discussion. The first PROLOG solution closely parallels the SPARCL solution and is very similarly brief. It is an unusual use of PROLOG - the nesting of setof/3 predicates is rare, and the obvious inefficiency of the program (in that it calculates each class

once for each example in the class instead of once for each value defining a class). This same inefficiency is found in the SPARCL implementation, so this solution is an interesting comparison with the SPARCL solution.

The second solution is substantially longer than the first PROLOG solution and the SPARCL solution. Its advantage is that it is a great deal more efficient than both of these solutions. It is less desirable than these other two solutions in ease of understanding. Much more “reading” is required to determine how the terms of the arguments are used in the literals; particularly, one must read and parse the code to find the variable references and to determine the coreferences. Since there are many fewer elements in the SPARCL implementation there is much less to read and parse; fewer procedures to track down, fewer arguments to “connect” (between the head and body literals), and fewer terms overall to interpret. Further, the coreferences are immediately obvious in the SPARCL code due to the coreference link representation.

The PROLOG solutions don’t offer any convenient graphical representation of the data: it is input and output as a list. The “set” nature of the collection of examples is nowhere explicitly apparent in the code, since PROLOG offers no way to work “directly” with sets. One could write additional code to format the data, but this would further complicate the implementation. Some PROLOG implementations have set-handling libraries. Such a library could help with the input/output format issue, but would make little difference in the structure of the rest of the solution.

The LISP implementation of the classify examples program.

An implementation in LISP of a solution to the classify examples programming problem is shown in Figure 7.1. 12. This program was extracted from a LISP implementation of ID3 with a few changes to simplify it for comparison with the other two solutions. In the version used by the ID3 program it actually goes beyond the basic programming problem to include counts of the “decision” attribute (assumed to be the first attribute/value pair in an example) and to “trim” the classifying attribute out of the classified examples.

Discussion. The SPARCL solution of the *classify examples* problem uses an unusual nested construction of intensional sets. We have “short” and “long” versions of the PROLOG solution, where the short version is slower than the long version. The long

```

(defun classify (name examples)
  (classify1 name examples nil))

(defun classify1 (name examples classes)
  (cond ((null examples) classes)
        (t (classify1 name
                       (rest examples)
                       (extend classes
                              name
                              (get-attribute-value name (first examples))
                              (first examples) )))))

(defun extend (classes name value example)
  (cond ((null classes)
        (list (list value
                     (list example) )))
        ((equal value (first (first classes)))
         (cons (list value
                     (cons example (second (first classes))) )
               (rest classes)))
        (t (cons (first classes)
                  (extend (rest classes) name value example) ))))

(defun get-attribute-value (name attributes)
  (attribute-match name attributes))

(defun attribute-match (name attributes)
  (cond ((equal name (first (first attributes))) (second (first attributes)))
        (t (attribute-match name (rest attributes)))))

```

Figure 7.1. 12: LISP *Classify Examples* solution.

version is typical of how a PROLOG programmer would solve this problem in order to get the improved performance. The SPARCL and short PROLOG solutions are structurally similar, but the SPARCL solution is (we claim) more readily understood. This claimed relative ease of understanding of the SPARCL solution depends on the reader having a thorough understanding of the intensional set semantics of SPARCL. Such an understanding may be difficult to acquire “passively” from reading an exposition such as this thesis, we believe it generally requires experience with constructing and using programs that use intensional sets.

The coreference links contribute to the concision of the SPARCL solution by allowing the programmer to refer to various parts of a complex data structure, an example within a set. The “example within a set” is the second argument of the ‘unify’/2 literal in the “inner” intensional set, in Figure 7.1. 2. There are three links to different parts of this structure, making it easy to see what aspects of this structure are referenced else-

where in the clause, and exactly where those references are.

The LISP solution is similar to the long PROLOG solution. It is much more complex than the SPARCL solution. The explicit representation of sets in SPARCL (and the lack of this in the other languages) contributes to its *classify examples* solution being easier to understand than the solutions in the other languages. The representation of the data for input and output is better in SPARCL than in LISP or PROLOG since SPARCL provides a familiar and concise representation for tables that is entirely appropriate to the *classify examples* problem, where LISP and PROLOG don't provide any diagrammatic representations of data. Since SPARCL selects the table representation for the output data without direction from the programmer, this gives the programmer the appropriate representation without any output-representation-specific code in the SPARCL solution, keeping it simpler than otherwise.

2. The ID3 program.

The ID3 problem incorporates the *classify examples* problem as a subproblem. The ID3 algorithm of Michalski is a basic machine learning algorithm. The version of the problem used here is simplified from that presented by Ross Quinlan in [Quinlan 1982]. The ID3 algorithm is based on the CLS (Concept Learning System) of Earl Hunt, presented in [Hunt et al. 1966]. We retain the basic element of inferring a decision tree from a given collection of examples. An example is as described above for the classify examples problem, a collection of attribute-value pairs where one of the attributes is the “decision”. A concise description of the CLS/ID3 algorithm is given in page 407 of [Cohen&Feigenbaum 1982]:

The CLS algorithm starts with an empty decision tree and gradually refines it, by adding decision nodes, until the tree correctly classifies all of the training instances. The algorithm operates over a set of training instances, C , as follows:

- Step 1. If all instances in C are positive, then create a YES node and halt.
 If all instances in C are negative, create a NO node and halt.
 Otherwise, select (using some heuristic criterion) a feature, F , with values $V = \{v_1, \dots, v_n\}$ and create a decision node labeled F with a branch for each of these values.
- Step 2. Partition the training instances in C into subsets C_1, \dots, C_n according to the values of V .
- Step 3. Apply the algorithm recursively to each of the sets C_i .

The ID3 algorithm of Quinlan uses a heuristic criterion for step 1 which chooses the feature that most strongly discriminates between positive and negative instances. He defines “most strongly discriminates” to be the feature that “leads to the greatest reduction in the estimated entropy of information of the training instances in C .”¹ The ID3 algorithm was an extension of the CLS algorithm which was meant to handle very large sets of training instances. This aspect of the algorithm is *not* used in our simplified version of the ID3 problem. We are using the CLS algorithm with ID3’s choice of heuristic criterion as our “ID3 problem.”

The solution to this problem in SPARCL is presented below, the solutions in LISP and PROLOG are given in appendix 3.

SPARCL solution. The SPARCL solution of the ID3 problem uses 12 predicates, which

1. p. 408 in [Cohen&Feigenbaum 1982].

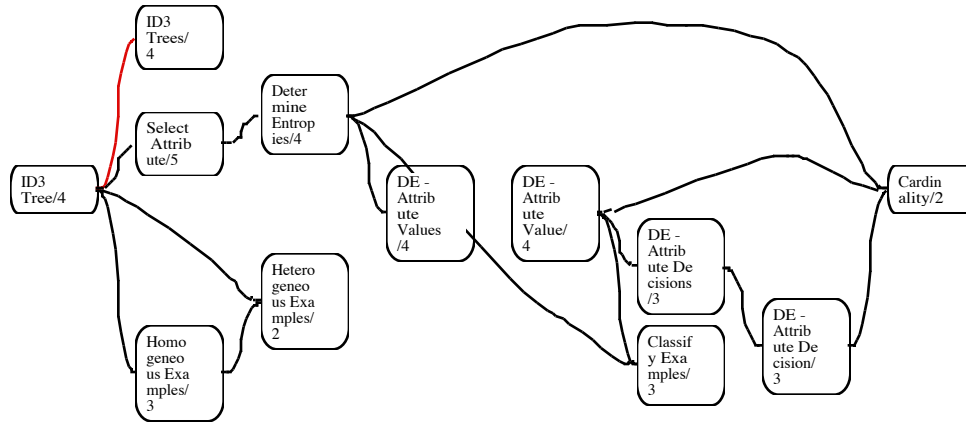


Figure 7.2. 1: Overview of the predicates in the SPARCL solution of ID3 problem. A predicate has a clause in its definition which has a literal for another predicate if two predicates are connected in the graph and the first predicate is “to the left” of the second predicate. For example, ‘ID3 Tree’/4 uses as literals in one or more of the clauses that define it the predicates ‘ID3 Tree’/4, ‘Select Attribute’/5, ‘Heterogeneous Examples’/2, and ‘Homogeneous Examples’/3. Not displayed in this graph: ‘ID3 Tree’/4 and ‘ID3 Trees’/4 are mutually recursive.

relate to each other as shown in Figure 7.2. 1. The definitions of the predicates are presented in the following figures.

The ‘ID3 Trees’/4 predicate is defined in Figure 7.2. 5. This predicate takes a set of independent attribute names, dependent attribute name, a set of example classifications, and returns a set of “decision trees” that summarize the relationship between independent attribute values and dependent attribute values. The set of decision trees is a decision tree determined by ‘ID3 Tree’/4 for each entry in the example classification set. An entry in the example classification set is an ordered pair of a classifying value and the set of examples which share that value in the classifying attribute. (The classifying attribute is not present, or necessary, in this predicate.)

The ‘ID3 Tree’/4 predicate defined in Figure 7.2. 2 finds a “decision tree” given a set of independent attribute names, the dependent attribute name, and an ordered pair associating a classifying value and a set of examples which all have that value in some attribute. This ordered pair is one entry in an example classification for an attribute. If the given examples all have the same value for the dependent attribute (i.e. they are *homogeneous* on that attribute), then a “leaf” decision tree node is constructed. This node is a triple of the classifying value, the dependent attribute name, and the common dependent attribute value.

If the examples do not all have the same dependent attribute value (i.e. they are

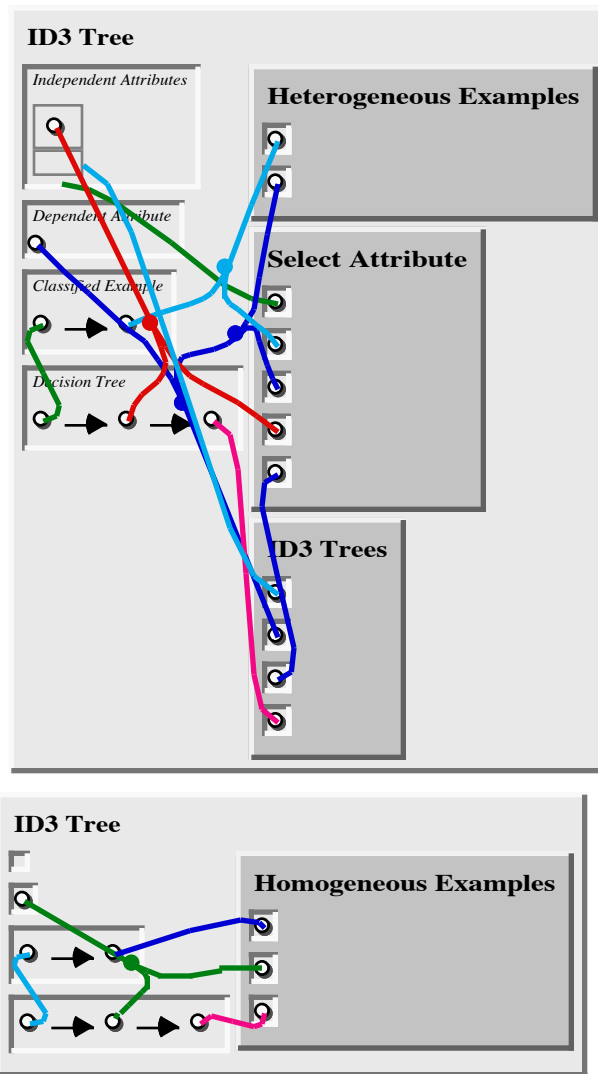


Figure 7.2. 2: 'ID3 Tree'/4 clauses.

heterogeneous on that attribute), then 'Select Attribute'/5 is used to select an independent attribute on which to "split" the examples, and 'ID3 Trees'/4 is used to construct a set of subtrees, one for each of the classifications of the given examples on the values of the splitting attribute.

The first argument of the upper clause for 'ID3 Tree'/4, the set of independent attribute names, contains an interesting use of a partitioned set. Three portions of this complex term are used in three different coreference hyperedges. The variable in the upper part of the partitioned set corefers with the variable which will be bound to the independent attribute chosen by 'Select Attribute'/5. Since this is the only element in the upper part of the independent attribute names set, the other part

of the partitioned set is all of independent attribute names *except* the selected attribute name. A distinct coreference is where this other part of the independent attribute names set corefers with the variable in the first argument of the 'ID3 Trees'/4 literal; this is the set of independent attribute names from which 'ID3 Trees'/4 will split the classification on the selected attribute name of the input example set and produce the subordinate decision trees. The final distinct coreference is where the independent attribute names set as a whole corefers with the variable of the first argument, the independent attribute names set, of the 'Select Attribute'/4 literal.

There is an example of a query of 'ID3 Tree'/4 and the result of that query in Figure 7.2. 3. The classified examples are given to 'ID3 Tree'/4 as an ordered pair in

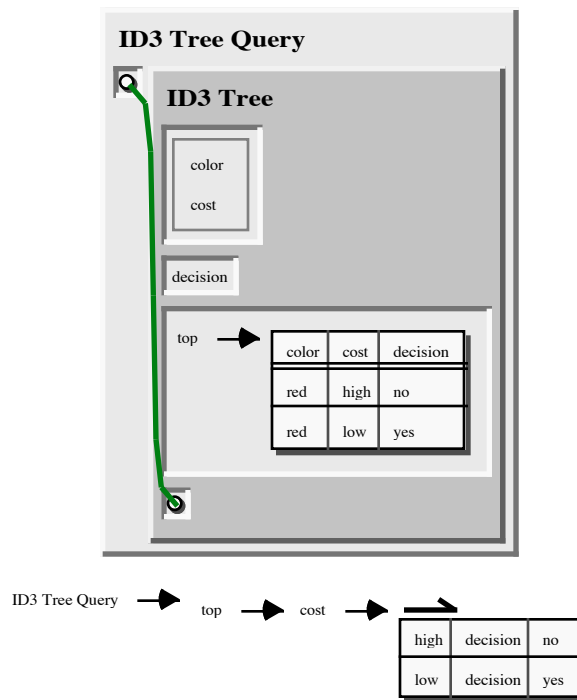


Figure 7.2. 3: 'ID3 Tree Query'/1 clause and the result of evaluating this clause.

the third argument. They have an artificial root value of 'top' in the first element and the example (function) table of interest as the second element. This function table represents a set of two examples, where each example is a set of three ordered pairs. The result ordered pair has as its second element the term built by evaluating 'ID3 Tree'/4 with the given data. The "decision tree" is a triple that has the artificial root value, 'top', as its first element, a selected attribute as the second element ('cost'), and the

subtree for that selected attribute's values as the third element. The subtree was built in evaluating the 'ID3 Trees'/4 literal in the "heterogeneous" clause of 'ID3 Tree'/4. This subtree is displayed as a table of ordered triples, where the first column is a 'cost' value, the second column is the dependent attribute name ('decision'), and the third column is the expected value of this dependent attribute given the independent attribute values encountered in descending the tree. Thus, this result table specifies that a high cost has a decision of 'no' and a low cost has a decision of 'yes'.

The '*DELAY*'/2 clauses in Figure 7.2. 4 cause the interpretation of 'ID3 Tree'/4 literals to be delayed whenever any of the first three arguments are unbound variables. Thus, this predicate is only interpreted when the set of independent attribute names, the dependent attribute name, and the classified examples entry are at least partially instantiated (i.e. non-variable).

The definition of 'ID3 Tree'/4 is shown in Figure 7.2. 5. It classifies a set of examples on some selected attribute's values and uses 'ID3 Trees'/4 to process each of these classified sets of examples, which will recursively invoke 'ID3 Tree'/4. 'Select Attribute'/5 chooses an attribute which has the lowest "entropy", calculated by

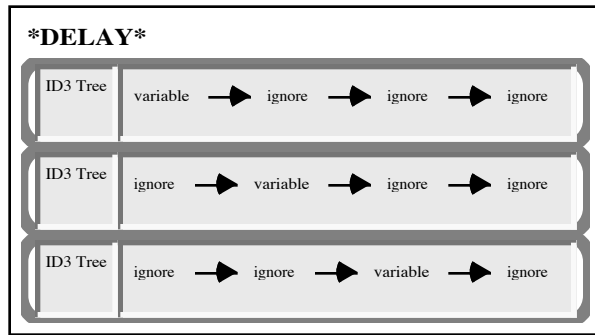


Figure 7.2. 4: “*DELAY*/2 clauses for ‘ID3 Tree’/4

some attribute name and bind the lower portion to the attribute names which are the rest of the set. If this attribute name does not satisfy the ‘Select Attribute’/5 literal when it is interpreted, then SPARCL will backtrack all the way back to the unification of the literal with the clause and make another choice such that some other attribute name is bound to the variable and the subset of the independent attribute names without this newly chosen attribute name is bound to the lower part. This backtracking could happen many times, until an attribute name satisfactory to ‘Select Attribute’/5 is finally chosen. All of this backtracking could be avoided if the unification of the parts of the partitioned set was delayed until some portion of it was needed. Since we have ‘*DELAY*/2 specifications for ‘Select Attribute’/5 and ‘ID3 Trees’/4, we know that the “selected attribute” need not be bound before ‘Select Attribute’/5

‘Determine Entropies’/4.

There is a potentially massive inefficiency in the way SPARCL interprets the upper clause of Figure 7.2. 2. The unification of the first argument of an ‘ID3 Tree’/4 literal with the first argument of this clause will bind the variable in the upper portion to

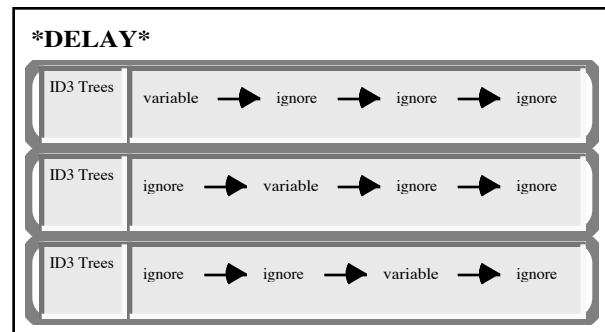
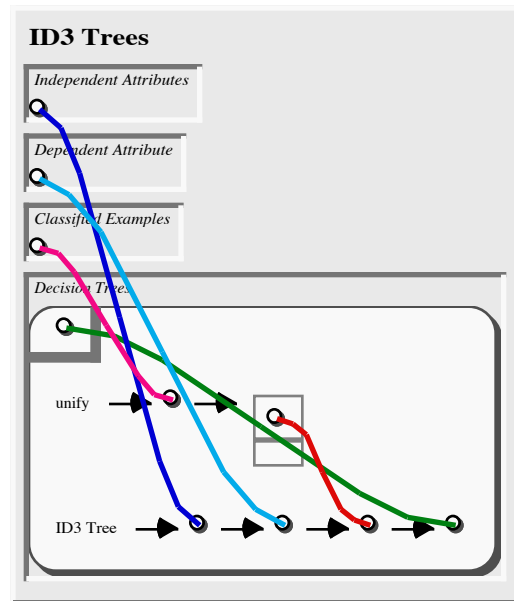


Figure 7.2. 5: ‘ID3 Trees’/4 and its “*DELAY*/2 clauses.

can be interpreted. Also, we know that the “nonselected independent attribute names” set must be bound before the ‘ID3 Trees’/4 literal can be interpreted. Thus, the partitioned set unification should be done after ‘Select Attribute’/5 is interpreted and before ‘ID3 Trees’/4 is interpreted. We imagine that SPARCL could do this kind of analysis as part of its literal ordering, but it does not do so currently.

The definition of ‘Select Attribute’/5 is shown in Figure 7.2. 6. It finds an independent attribute from the given independent attributes that has minimal entropy over the given example set when compared to the all of the other given independent attributes over that example set. It uses ‘Determine Entropies’/4 to develop an “entropy triple” for each independent attribute, then the *fails-unify-less* constrains the specified triple to be one for which there is no other triple with a lower entropy. Each triple consists of an attribute name, the example classification for that attribute, and the entropy for that attribute. There is a potential ordering bug in this clause. The *fails*/2 literal must be interpreted after the ‘Determine Entropies’/4 literal, but there is nothing in the program to enforce this. To fix this, the *fails*/2 literal can be placed in a new predicate’s clause (say ‘Minimal Entropy Triple’/2) and this new predicate have a delay specification that causes the appropriate ordering by delaying interpretation if either argument is an unbound variable (i.e. (variable => ignore) and (ignore => variable)).

This clause demonstrates some of the expressiveness of partitioned sets. The partitioned set in the fourth argument of the ‘Determine Entropies’/4 literal is used to simultaneously *select* an element from a set and to specify the subset of that set *minus* the selected element. The ‘*fails*’/1 literal imposes a constraint on the relationship between the selected element and the remaining subset. SPARCL solves this constraint by searching through all of the possible choices of elements until one is found that is acceptable (and remains prepared to keep looking if the later steps of the interpretation should fail).

The definition of ‘Heterogeneous Examples’/3 is shown in Figure 7.2. 7. It determines if a given set of examples contains at least two examples that have different values for a given attribute.

The definition of ‘Homogeneous Examples’/3 is shown in Figure 7.2. 8. It determines if all of the given examples have the same value for a given attribute and returns that common value. It uses the failure of ‘Heterogeneous Examples’/3 to determine the homogeneity of the examples. Since all of the examples have the same value on the given attribute, one of these examples is selected by the nested sets of

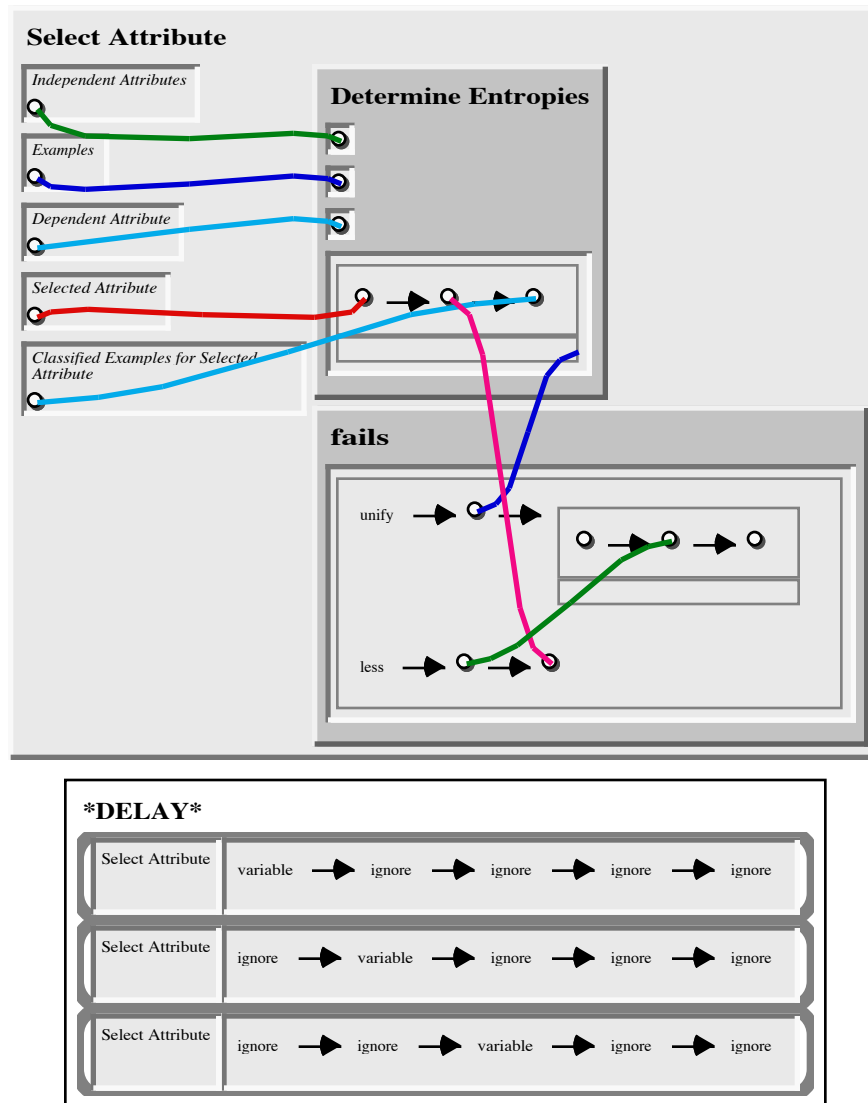


Figure 7.2. 6: 'Select Attribute'/5 predicate definition.

the first argument and the value in this example is used as the returned value.

This predicate could be implemented more efficiently by making it gather the set of all values on the given attribute and unify that set with a singleton set. The single element in the singleton set is the common value. This is more efficient than the implementation shown in that there are no “choice points” left behind in the interpretation of this alternative implementation, but there is a (useless) choice point in the implementation of Figure 7.2. 8, the selected example in the first argument. On backtracking, the interpreter would unify the “selected example” with each of the different

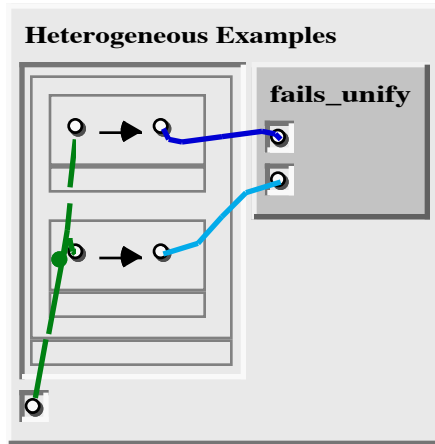


Figure 7.2. 7: 'Heterogeneous Examples'/3 clause.

examples of the given example set, getting the same attribute value each time.

The definition of the 'Determine Entropies'/4 predicate is shown in Figure 7.2. 9 and Figure 7.2. 10. This predicate builds the set "entropy triples" using an intensional set term in its fourth argument. This term is the set of all entropy triples such that the 'unify'/2, 'Classify Examples'/3, and 'DE - Attribute Values'/4 literals are simultaneously satisfied. The 'unify'/2 literal is true once for each element of the given set of attribute names.

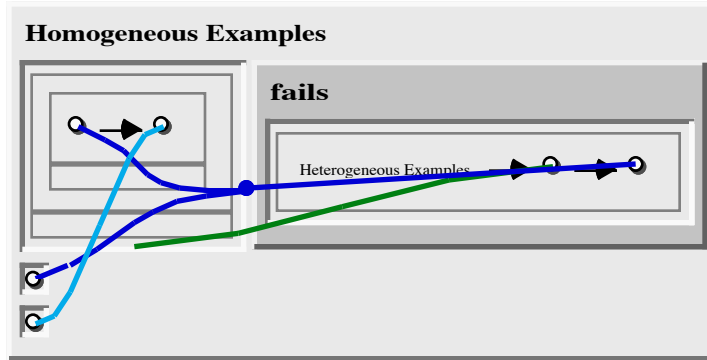


Figure 7.2. 8: 'Homogeneous Examples'/3 clause.

'Classify Examples'/3 gives the example classification for the attribute name "selected" by the 'unify'/2 literal. Finally, 'DE - Attribute Values'/4 the entropy of the selected attribute with respect to the given dependent

attribute name using the example classification for the selected attribute.

The literals in the intensional set are an example of ordering interpretation using the '*DELAY*/2 specifications. Since 'Classify Examples'/3 is delayed if either of its first or second arguments are unbound, it must be interpreted after the 'unify'/2 literal. Since 'DE - Attribute Values'/4 is delayed if any of its first three arguments are unbound, it must be interpreted after 'Classify Examples'/3.

The definition of the 'DE - Attribute Values'/4 predicate is shown in Figure 7.2. 11. The entropy for an attribute is the sum of the sub-entropies. There is one sub-entropy for each entry in the given example classification of that attribute. The sub-entropies are calculated by the 'DE - Attribute Value'/4 predicate. The sub-entropy values are collected in a multiset instead of a simple set so that duplicate values are "preserved" and thus all values calculated contribute to the sum of the sub-entropies.

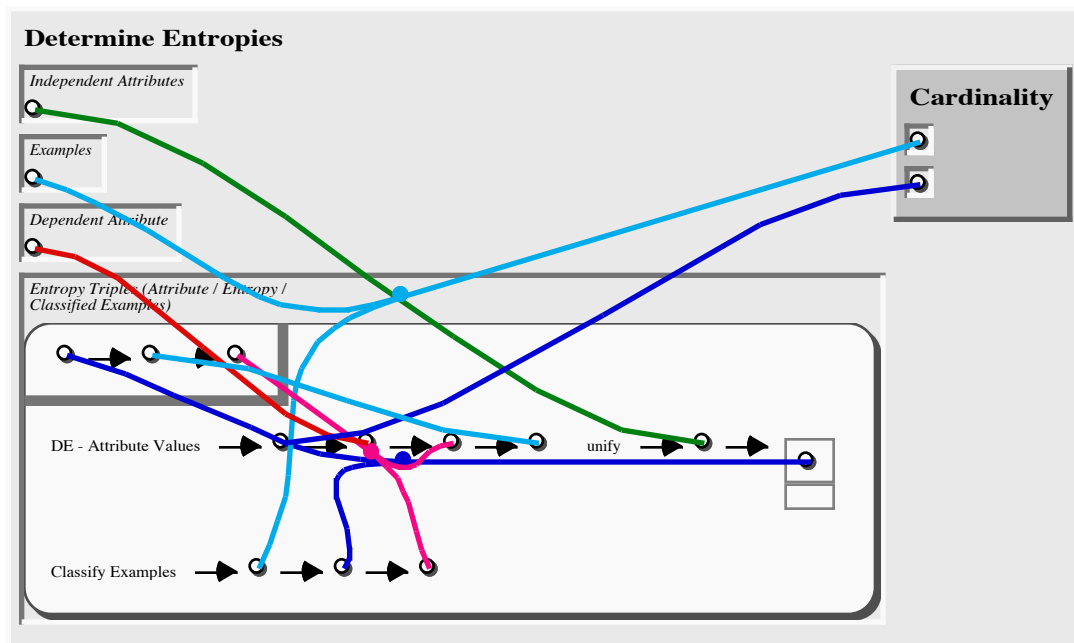


Figure 7.2. 9: 'Determine Entropies'/4 predicate definition (except delays).

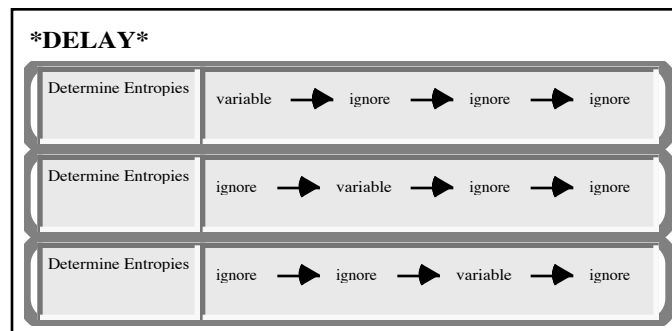


Figure 7.2. 10: ***DELAY*/2** clauses for 'Determine Entropies'/4.

The definition of the 'DE - Attribute Value'/4 predicate is shown in Figure 7.2. 12. This predicate calculates the local (or "sub") entropy for an entry of an example classification. The local entropy is the product of three values; the "decision negative

entropy", -1, and the ratio of the size of this classification entry's example set to the complete example set for which the entropy is being calculate. The "decision negative entropy" calculation relies on the classification of the given example set on the dependent attribute and is calculated by the 'DE - Attribute Decisions'/3 predicate.

There is almost a bug in this clause. The 'is'/2 literal's second argument expression is the product ('*') of a *set* of values. This should be a *multiset* of values. It happens to be impossible for the three values to coincide, but if they did, the replication due to this coincidence would be lost. Thus, if the "decision negative entropy" could be -1, then the set of values becomes just {-1, Ratio} instead of the desired {-1, -1,

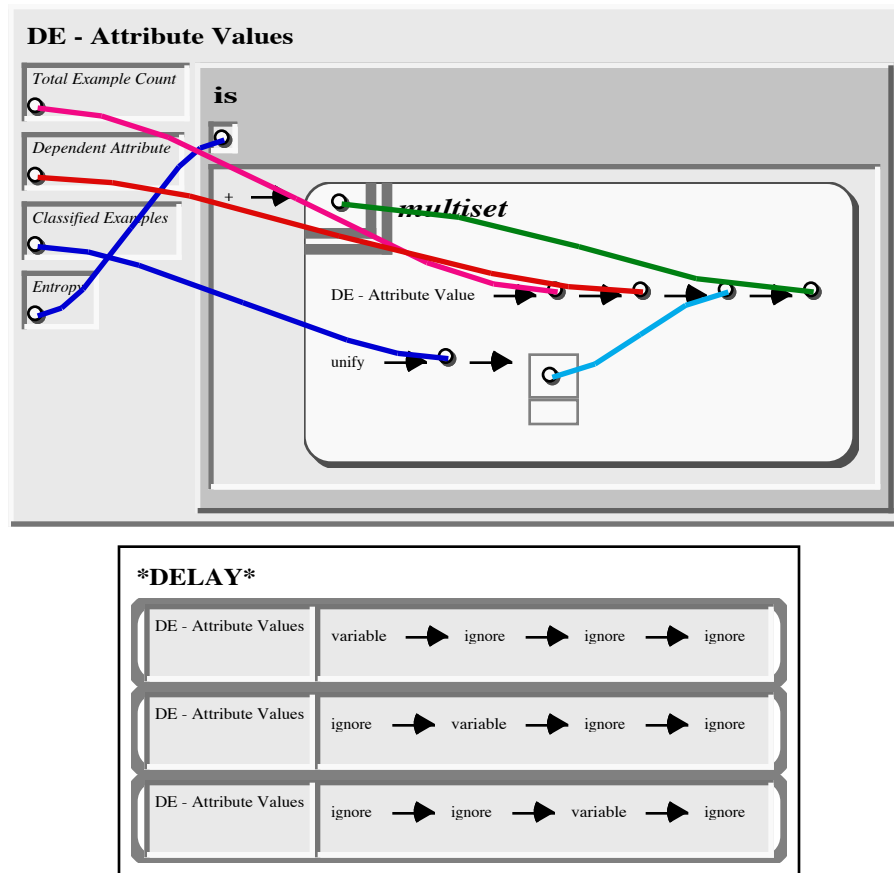


Figure 7.2. 11: ‘DE - Attribute Values’/4 predicate definition.

Ratio}. This produces dramatically different results. This can be fixed by replacing the expression with nested multiplication ordered pairs, such as “* => DNE => (* => -1 => Ratio)” (DNE is the “decision negative entropy”). That the coincidence of these values is impossible can be shown as follows: -1 and the ratio of two positive numbers can never coincide; and, the decision negative entropy must be negative and must be greater than -1, and thus it cannot be the same as either of the other values in the set.

The definition of the ‘DE - Attribute Decisions’/3 predicate is shown in Figure 7.2. 13. The decision negative entropy is the sum of the decision classification negative entropies. The multiset in the expression argument to the ‘is’/2 literal gathers the decision classification negative entropies, one for each classification of the examples on the decision attribute (the first argument of ‘DE - Attribute Decisions’/3).

The definition of the ‘DE - Attribute Decision’/3 predicate is shown in

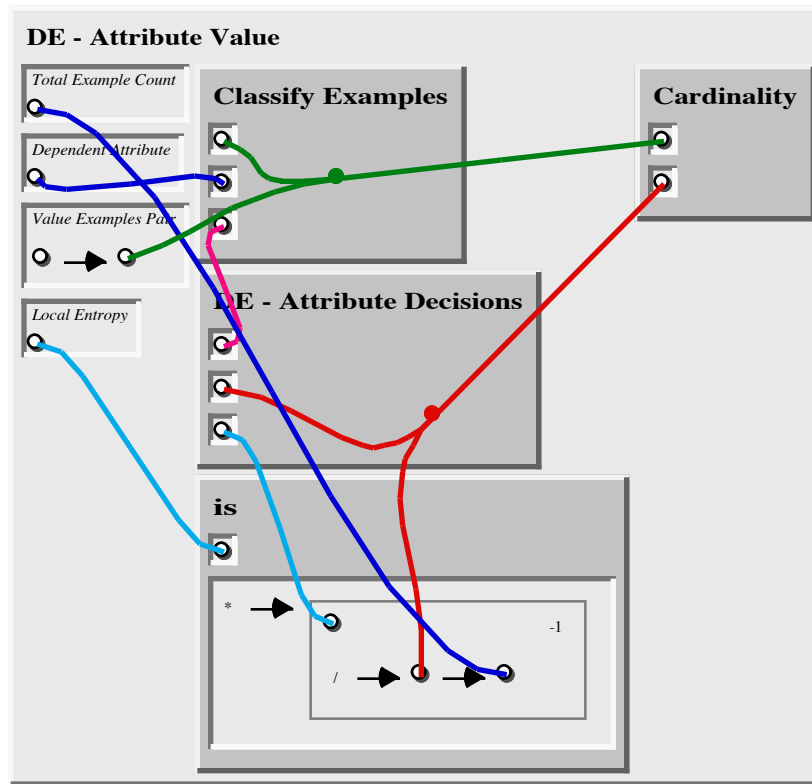


Figure 7.2. 14. This predicate calculates the decision classification negative entropy given a set of examples from a decision (dependent) attribute-based classification of an entry in an independent attribute-based classification and the cardinality of the independent attribute-based classification entry's example set. Let D be the cardinality of the set of examples in the decision attribute-based classification entry and A be the cardinality of the example set of the independent attribute-based classification entry. The decision classification negative entropy (DCNE) is:

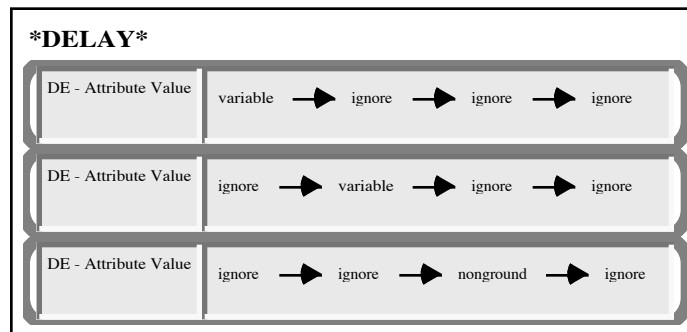


Figure 7.2. 12: 'DE - Attribute Value'/4 predicate definition.

$$DCNE = \left(\frac{D}{A}\right) \log\left(\frac{D}{A}\right)$$

The definition of the 'Cardinality'/2 predicate is shown in Figure 7.2. 15. This predicate determines the cardinality (size) of a given set. The calculation is neither

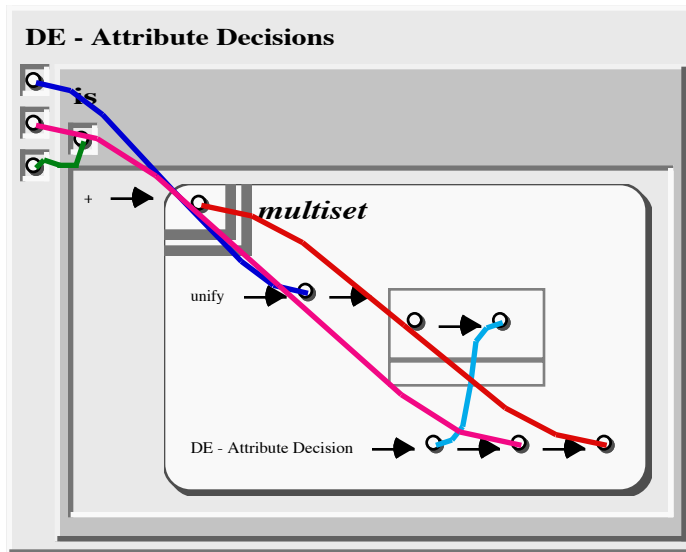


Figure 7.2. 13: 'DE - Attribute Decisions'/3 predicate definition.

iterative nor recursive. An intensional multiset is used in an unusual fashion to count the number of elements in the given set. The body of the intensional multiset is a 'unify'/2 literal which will be satisfied once for each element of the given set. The unusual aspect of the intensional multiset is that there is no connection between the template and the body. There

will a 'marker' ur constant term in the resulting multiset for each solution of the 'unify'/2 literal, thus the repetition Counter of the "'marker'=>Counter" ordered pair in the resulting multiset gives the cardinality of the given set.

This predicate definition is included here because of its unusual use of the intensional multiset and as an example of set processing which is neither iterative nor recursive. SPARCL should have 'Cardinality'/2 as a built-in so that its performance can be optimized, since it is a commonly used application-independent predicate.

Discussion. We defined the ID3 problem and presented the SPARCL solution. The LISP and PROLOG solutions are given in appendix 3. This problem includes the *classify examples* problem as a subproblem. This is substantially larger solution than that for *classify examples* alone: the ID3 solution uses 12 predicates

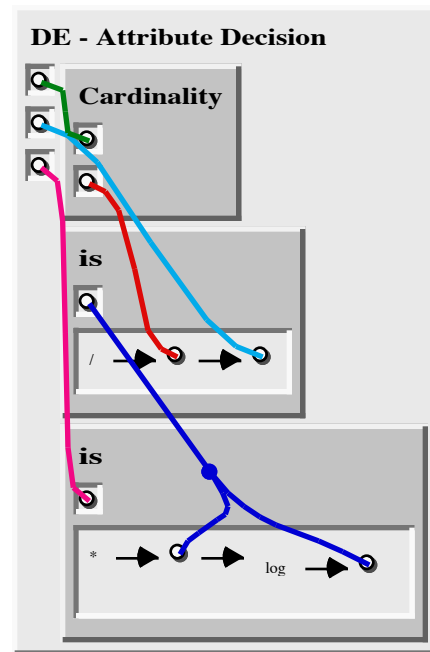


Figure 7.2. 14: 'DE - Attribute Decision'/3 predicate definition.

implemented by 15 regular clauses and 18 ‘*DELAY*’/2 clauses.

In the discussion of the ‘ID3 Tree’/4 clause of Figure 7.2. 2 we note that SPARCL can interpret this clause very inefficiently. This is due to SPARCL unifying parts of a partitioned set before they are “needed”, a kind of eager unification. There is an

opportunity of a substantial performance improvement in some predicates if we develop an appropriate notion of lazy unification for partitioned sets.

This clause demonstrates some of the expressiveness of partitioned sets in that it uses a partitioned set to simultaneously *select* an element from a set and to specify the subset of that set *minus* the selected element. The selection is arbitrary “don’t know”-style nondeterminism—there is some element that satisfies an associated constraint, but the programmer doesn’t know which one it is ‘a priori’. SPARCL applies the constraint to each element until it finds one that is satisfactory.

The literals in the intensional set of ‘Determine Entropies’/4, shown in Figure 7.2. 9, are an example of ordering interpretation using the ‘*DELAY*’/2 specifications. The delay specifications for ‘Classify Examples’/3 and ‘DE - Attribute Values’/4 have the effect of forcing the interpreter to interpret the ‘unify’/2 literal first, then the ‘Classify Examples’/3 literal, and finally the ‘DE - Attribute Values’/4 literal. This ordering in this particular clause need not have been considered by the programmer, the programmer only needed to consider each of the component procedures independently (when implementing those procedures) and make appropriate delay specifications considering only the semantics of the procedure. This is a simplification over languages requiring explicit ordering where the programmer must determine the proper ordering herself in every grouping of literals. In SPARCL, the system uses the programmer’s delay specifications to determine the proper ordering. This provides the programmer a more abstract approach to ordering execution.

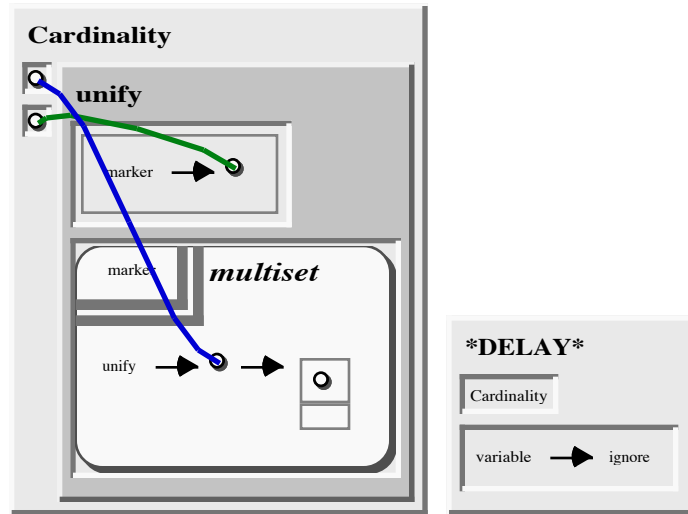


Figure 7.2. 15: ‘Cardinality’/2 predicate definition.

The ‘DE - Attribute Values’/4 predicate definition, in Figure 7.2. 11, shows a use of an intensional multiset in an arithmetic expression. In this use it provides a very compact expression of the sum of all of the values having a particular property (being the result of a ‘DE - Attribute Value’/4 literal). The ‘DE - Attribute Value’/4 predicate definition, in Figure 7.2. 12, shows that a partitioned set can be used as the argument to a commutative operator (‘*’ in this case), but that one must carefully analyze whether the elements of the set might possibly take on the same value (and thus “disappear” from the calculation). In this case we show that the three elements can not take on the same value.

The definition of the ‘Cardinality’/2 predicate is shown in Figure 7.2. 15. This predicate is should be implemented as a built-in in SPARCL, since it is semantically simple, commonly useful (not problem domain specific), and could be enormously speeded up by building it in. We show it here because this definition makes an unusual use of the intensional multiset. It is used to count the elements of a set. It is unusual because there is no connection between the intensional set template term (an ur constant ‘marker’) and the literal in the body of the intensional set. There is one instance of ‘marker’ in the resulting multiset for each solution of the body. Since there is exactly one solution of the for each member of the set for which we want the cardinality, then the count of ‘marker’ in the result multiset is the cardinality of interest.

Coreference links provide various aids to the programmer in the various clauses of the *ID3* solution. As in the *Classify Examples* problem, they make it easy to connect interior portions of complex data structures, and to readily see and understand these connections. An example of this can be seen in Figure 7.2. 6 and the various connections between elements in N-tuples in different sets. Different aspects of the same complex structure can participate in different links, which helps to keep the representation compact. This can be seen in Figure 7.2. 2 in the first argument of the “heterogeneous” clause of ‘ID3 Tree’/4 where the partitioned set as a whole, an element of the partitioned set, and a part of the partitioned set all are used in coreference links. The number of uses and their locations of a term can be easily seen by “following” the coreference link that connects to it, if any. It is apparent at a glance that all of the coreferences in ‘Select Attribute’/5 in Figure 7.2. 6 are “simple” (i.e. only involve two terms). In more complex coreferences such as those of “heterogeneous” clause of

'ID3 Tree'/4 in Figure 7.2. 2, it is still easy to see that the dependent attribute name in the second argument is referenced in exactly four places, and where those four places are.

- Cohen&Feigenbaum 1982 *Handbook of Artificial Intelligence, Vol. III* edited by Paul R. Cohen and Edward A. Feigenbaum. William Kaufmann, Inc: Los Altos, California. 1982.
- Hunt et al. 1966 *Experiments in Induction* by E. B. Hunt, J. Marin, and P. J. Stone. New York: Academic Press. 1966.
- Quinlan 1982 "Induction of Decision Trees" by J. R. Quinlan. Pages 81-106 in *Machine Learning*, vol. 1, no. 1, edited by R. S. Michalski, T. M. Mitchell, and J. Carbonell. Palo Alto, California: Tioga. 1982.

3. The WARPLAN Program.

David H. D. Warren devised the WARPLAN planning algorithm [Warren 1974] specifically to take advantage of the abilities of PROLOG. This is an elegantly logical approach to planning which he designed with the PROLOG programming language in mind. It is readily adaptable to other logic programming languages such as SPARCL. However, since it relies on the backtracking search and unification machinery of logic programming, it is relatively difficult to implement in non-logic programming languages such as LISP. WARPLAN is an extension of the STRIPS approach to planning [Fikes&Nilsson 1971], which in turn is based on Green's resolution-based planner [Green 1969] and makes use of the means-ends analysis pioneered by GPS (General Problem Solving) [Ernst&Newell 1969].

WARPLAN solves planning problems—given a world model, a goal, and some initial conditions, it finds a sequence of actions which achieve the goal from the initial conditions. The world model defines the types of actions possible and the possible world states. A world state is a collection of facts (which are relations between objects). A goal is a set of facts. An action is an instance of an operator (a type of an action). The application of an action changes the world state to a new state. An operator is defined in three parts: the preconditions, the additions, and the deletions. Each of these parts is a collection of (possibly nonground) facts. Typically the facts in these three parts share some variables (e.g., doing an action may delete some fact in its precondition). Instantiating an operator involves binding all of the variables in that operator's definition. This grounded form of the operator defines the action.

The standard example planning problem is the Blocks World. The world model is very simple: there are four objects, blocks A, B, and C, and the floor; there is one operator—*move* a block from one place to another; and there are two kinds of facts—a block is *on* something, and a block is *clear*; there are three impossible types of combinations of facts—a block can't be clear and having something on it, a block can't be on two different things, and a block can't be on itself. A simple interesting initial state places A on the floor, C on A, and B on the floor. A classic goal to achieve from this state is to have A on B, B on C, and C on the floor.

The WARPLAN algorithm builds a plan incrementally which “solves” each fact in the goal. If a fact “holds” in the current plan, then it considers that fact solved. If a goal fact is not solved, then it finds an action which adds that goal fact (i.e. the goal

fact is in the additions part of the action's definition), and then it finds a way to add that action to the current plan such that that actions precondition facts are solved by the portion of the plan preceding the action.

There are two ways to add the action to the current plan, either at the end (to extend the plan), or inserting the action somewhere before the end of the plan. Adding the action at the end is fairly simple (if possible at all); the major complication being to avoid "deleting" any of the goal facts which have already been solved. Inserting the action at an earlier point in the plan is more complicated: the current plan is "retraced", extending the set of solved facts to be protected so that the retraced portion of the plan won't be "broken" by the action(s) being inserted. When trying to add an action, WARPLAN must ensure that the preconditions of that action are all solved by the plan preceding the action. In doing this, WARPLAN may need to recursively invoke the planner to find action sequences which solve some of the precondition facts.

The implementation of WARPLAN is complicated by allowing operator definitions to have variables. These variables should be instantiated as late in the planning process as possible, making the current plan actually represent a set of current plans (all of the different plans which could be created by different instantiations of the variables in the current plan). A plan with variables can be considered a plan *type*. This use of variables can substantially reduce the size of the space WARPLAN must search since it can do much of its searching in the plan type space instead of directly in the plan space.

The solution of the WARPLAN problem in SPARCL is discussed below, the solution in PROLOG is given in appendix 3 ("Example Programs").

SPARCL solution. The SPARCL solution to the WARPLAN problem uses 20 predicates which relate to each other as shown in Figure 7.3. 1 and Figure 7.3. 2. The overview graph is split in two parts, with both parts showing the nodes for 'Achieve'/5, 'Retrace 1'/4, and 'Equivalent'/2. The general WARPLAN solution is all of the predicates shown in the overview except for 'Can', 'Add', 'Del', 'Given', and 'Impossible'. These five predicates define a particular "world". The versions of these predicates for the "blocks world" definition are shown in Figure 7.3. 20, Figure 7.3. 21, Figure 7.3. 22, and Figure 7.3. 23. A query clause which poses a blocks world planning problem is shown in Figure 7.3. 24.

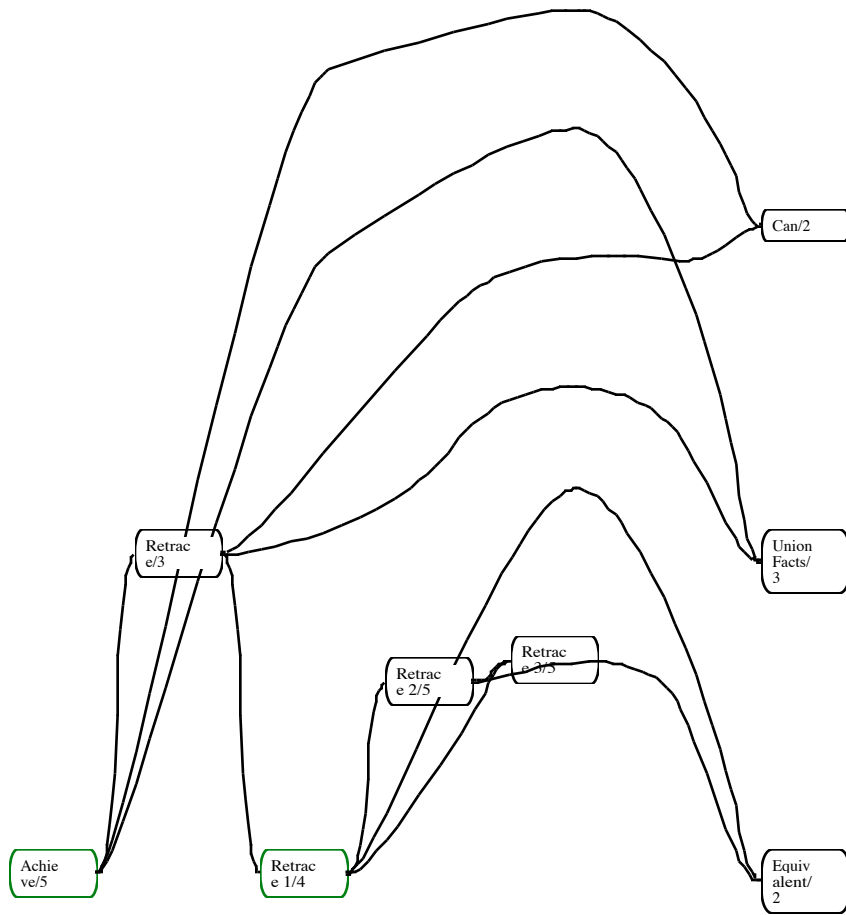


Figure 7.3. 1: Overview of the SPARCL WARPLAN problem solution, part 1.

Figure 7.3. 1 shows most of the ‘Retrace’/3 subgraph of the overview. One predicate used by ‘Retrace’/3, ‘Consistent’/2, is in Figure 7.3. 2 instead. The ‘Achieve’/5 and ‘Retrace 1’/4 predicates use themselves (direct recursion). The ‘Retrace’/3 predicate implements the retracing discussed above, splitting up the task among three other predicates, ‘Retrace 1’/4, ‘Retrace 2’/4, and ‘Retrace 3’/4.

Figure 7.3. 2 shows the ‘Plan’/4 and ‘Preserves’/2 subgraphs of the overview. ‘Plan’/4 and ‘Holds’/2 are directly recursive, in addition ‘Achieve’/5 and ‘Retrace 1’/4 mentioned above. ‘Implied’/2 uses ‘Inconsistent’/2 as well as being used by ‘Inconsistent’/2. ‘Solve’/5 uses ‘Achieve’/5 (instead of the other way around as pictured).

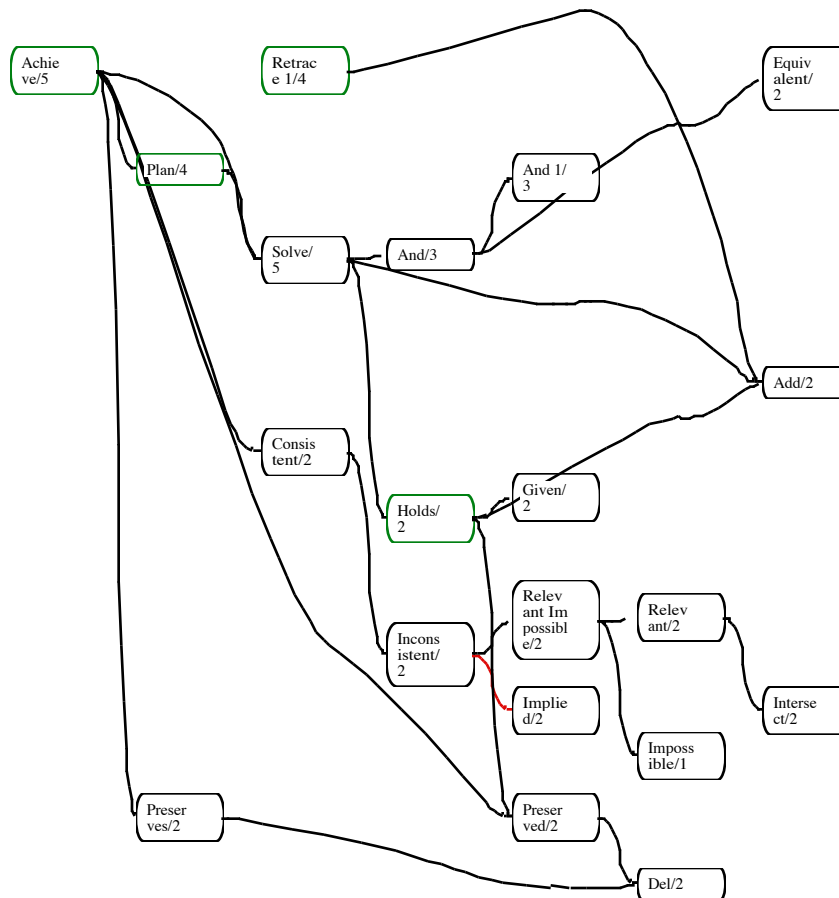


Figure 7.3. 2: Overview of SPARCL WARPLAN problem solution, part 2.

The ‘Achieve’/5 “plan” clause is shown on the left side of Figure 7.3. 3. The ‘Achieve’/5 predicate finds a plan *extending* the given plan which achieves the given action while preserving the given protected facts, where the “achieving” plan supports the given action’s precondition. The given action’s precondition is specified by the ‘Can’/2 literal. The preservation of the given protected facts by the given action is checked by the ‘Preserves’/2 literal. The action-enabling precondition facts and the protected facts must be consistent (i.e. not be an impossible conjunction). This consistency is checked by ‘Consistent’/2. Note that the second argument is the union of the precondition facts and the protected facts, while the first argument is the precondition facts. This separate set of precondition facts is for a performance enhancement of ‘Consistent’/2.

The “retrace” clause of the ‘Achieve’/5 predicate is shown on the right side of Figure 7.3. 3. This clause retraces the given plan, making the last action of the given

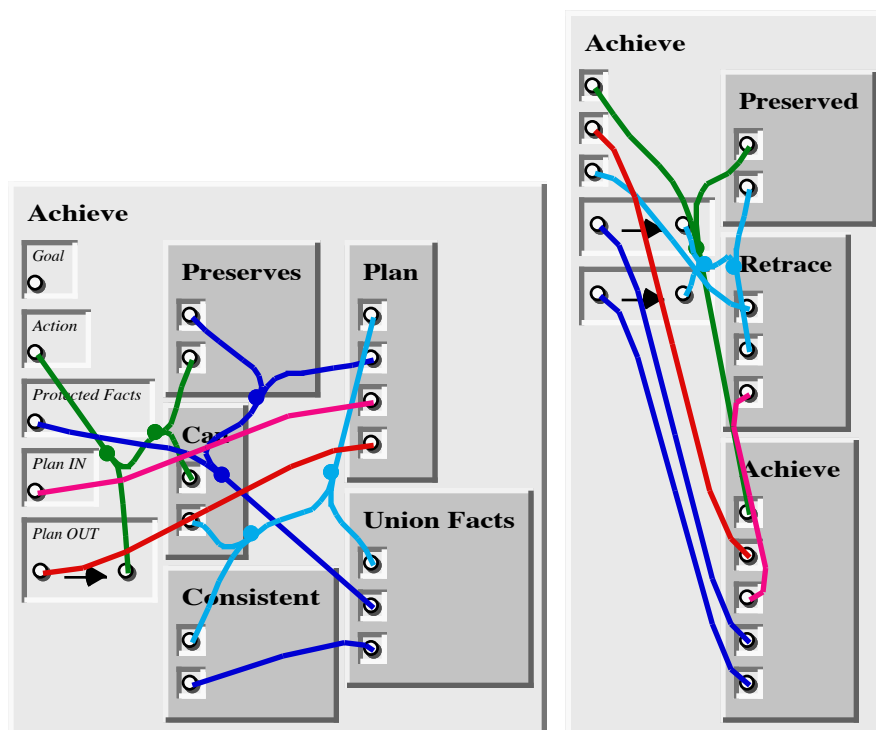
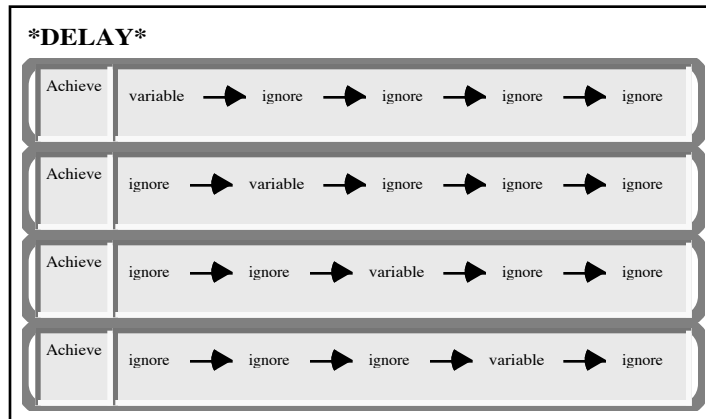


Figure 7.3. 3: 'Achieve'/5 predicate definition.

there is a danger that the retraced action will *undo* the given goal which the given action is supposed to be achieving. The 'Preserved'/2 literal is used to ensure that this doesn't happen. The protected facts of the recursive use of 'Achieve'/5 literal are the given protected facts and precondition facts of the retraced action. Protecting the retraced action's precondition facts ensures that the retraced action can be done after the resulting plan from the recursive 'Achieve'/5 literal. The adjustment of the protected facts is done by the 'Retrace'/3 literal.

plan the last action of the result plan and finding a plan which achieves the given action based on the retraced

plan. The retraced action is the one which is the last action of the given plan and is the last action of the result plan. Since this action will be done *after* the plan which achieves the given action,

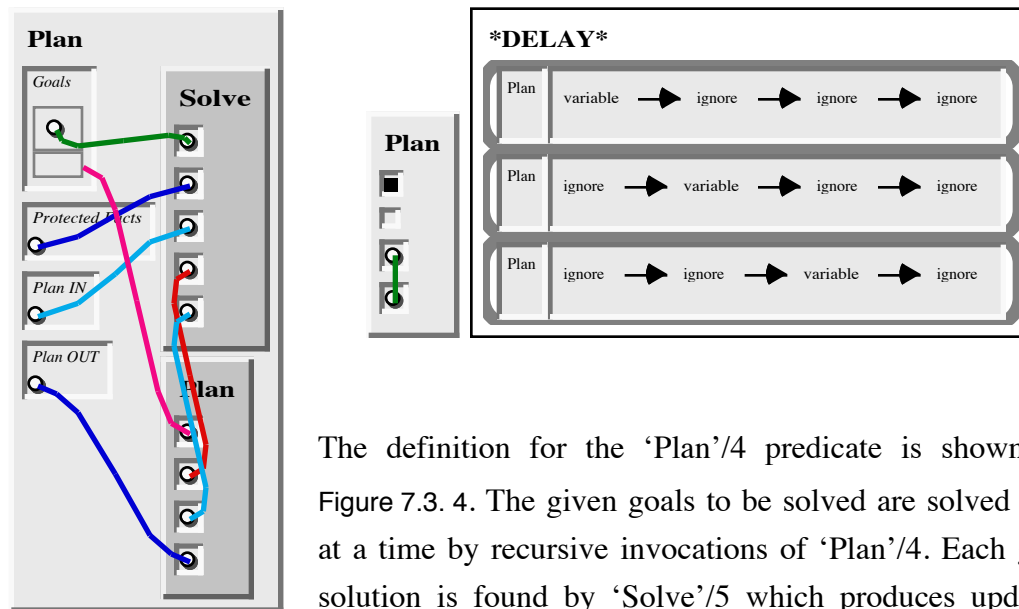


Figure 7.3. 4: 'Plan'/4 predicate definition.

An example query of the 'Plan'/4 predicate and the result of that query are shown in Figure 7.3. 5. This query clause is used to query 'Plan'/4 to find a plan which goes from the "start" initial state to a state which includes "a is on b" and "b is on c". The initial state is "a is on 1", "b is on 2", "c is on a", "c is clear", "b is clear", and "3 is clear". The numbers indicate places on the floor at which blocks may be put. The letters are names of blocks. The state and operation definitions are the "blocks world" defined later in this section. This planning problem is referred to as the "Sussman Anomaly". [Rich&Knight 1991] (p.344) attributes this naming to the extensive analysis of this problem in [Sussman 1975]. The result shows a plan that achieves the goal state in a table that is an N-tuple of N-tuples. The "root" element of the table is 'start' (the "zero-th" element of the N-tuple of rows). The next 3 elements are "move c from a to 3", then "move b from 2 to c", then "move a from 1 to b".

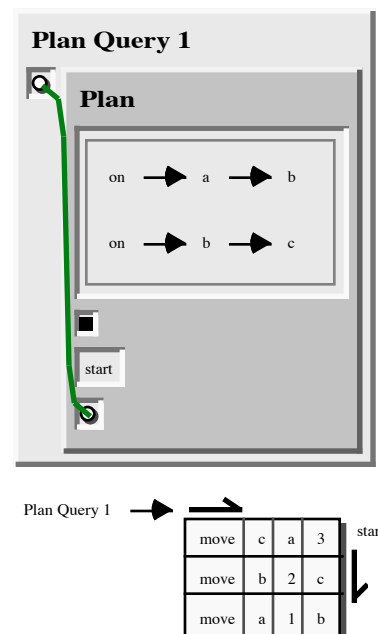


Figure 7.3. 5: Query for the 'Plan'/4 predicate and the result of evaluating this query.

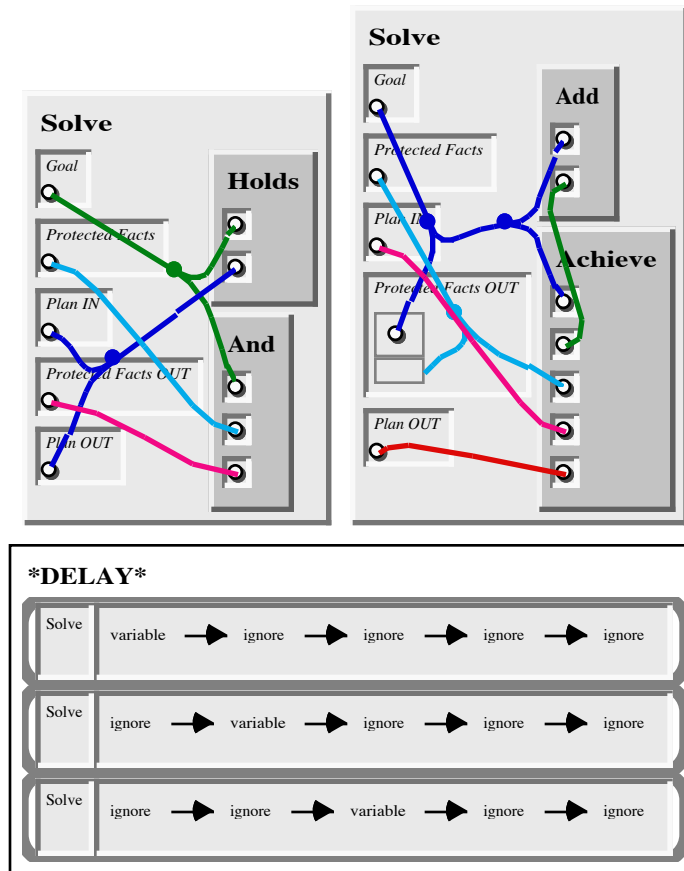


Figure 7.3. 6: 'Solve'/5 predicate definition.

adds the given goal, then uses the 'Achieve'/5 literal to find a modification of the given plan which includes the new action while preserving the given goal and the given protected facts. For both clauses, the resulting protected facts are the given protected facts plus the given goal.

The definition of the 'Holds'/2 predicate is shown in Figure 7.3. 7. The given fact (goal) holds if the last action in the plan "adds" it, if the given goal is preserved by the last action in the plan and the given goal holds for the preceding portion of the plan, or if the given plan is a single item which is the name of a given state and the given goal is in that state.

Retracing is implemented in four predicates. The 'Retrace'/4 predicate definition is shown in Figure 7.3. 8. 'Retrace'/4 produces a set of protected facts from given protected facts and a given action which preserves precondition facts for that given action. The 'Can'/2 literal provides precondition facts for the given action. The 'Consistent'/2 literal ensures that the combination of the reduced preserved facts and

The definition for the 'Solve'/5 predicate is shown in Figure 7.3. 6. 'Solve'/5 is true for a given goal if either that goal "holds" for the given plan or if there is an action which "adds" the given goal and which can be "achieved" with the given plan and protected facts. The "holds" clause uses a 'Holds'/2 literal to determine that the given goal is satisfied by the given plan. The "achieves" clause uses the 'Add'/2 literal to find an action which

the precondition facts is consistent.

The definition of the predicate ‘Retrace 1’/4 is shown in Figure 7.3. 9. ‘Retrace 1’/4 processes each given protected fact recursively (directly or indirectly through ‘Retrace 2’/5). Each given protected fact is checked if it is added by the given action. If it is, then it is not put in the resulting protected facts, and ‘Result 1’/4 is invoked on the remainder of the given protected facts. If the selected given protected fact is *not* added by the given action, then it is processed by ‘Retrace 2’/5, which ultimately recursively invokes ‘Retrace 1’/4 on the remainder of the given protected facts.

Figure 7.3. 10 shows the ‘*DELAY*’/2 facts for the ‘Retrace’/3 and ‘Retrace 1’/4 predicates. The delay facts for ‘Retrace’/3 specify that interpretation of ‘Retrace’/3 literals is to be delayed if either of the first two arguments are unbound variables. The delay facts for ‘Retrace 1’/4 specify

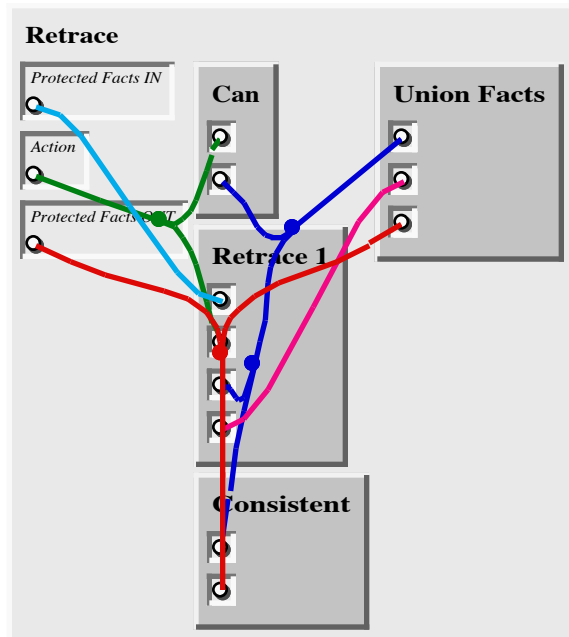


Figure 7.3. 8: ‘Retrace’/4 predicate definition.

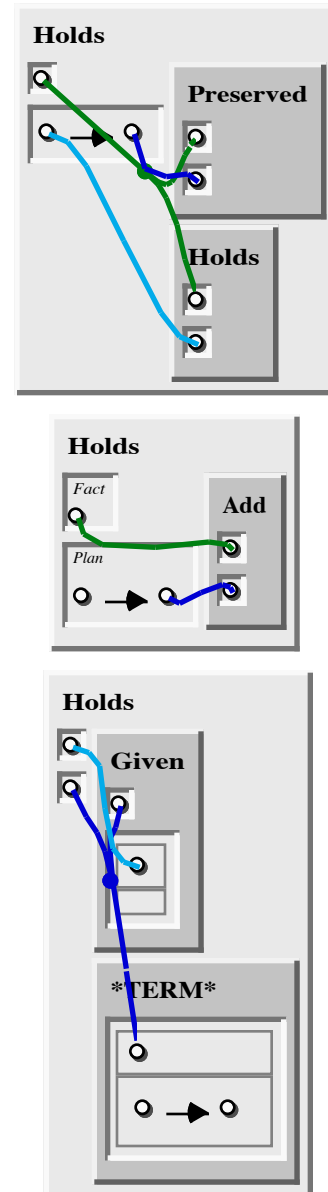


Figure 7.3. 7: ‘Holds’/2 predicate definition.

that interpretation of ‘Retrace 1’/4 literals is to be delayed if any of the first three arguments are unbound variables.

The definition of the ‘Retrace 2’/5 predicate is shown on the right of Figure 7.3. 11. This predicate checks if the given selected protected fact is equivalent to one of the given precondition facts (fourth argument to the

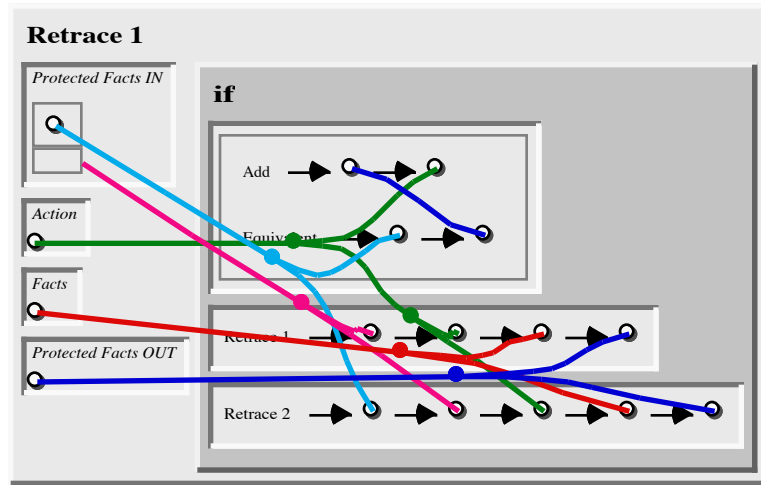


Figure 7.3. 9: ‘Retrace 1’/4 predicate definition.

clause). If it is equivalent, then it is not put in the reduced protected facts and ‘Retrace 1’/4 is used to process the rest of the given protected facts. If the given selected protected fact is *not* equivalent to a precondition fact, then ‘Retrace 3’/5 is used to add this fact to the result reduced protected facts and process the other given protected facts.

The definition of ‘Retrace 3’/5 is shown on the left of Figure 7.3. 11. This predicate simply puts the given selected protected fact in the result reduced protected facts and uses ‘Retrace 1’/4 to process the other given protected facts to produce the rest of the result reduced protected facts.

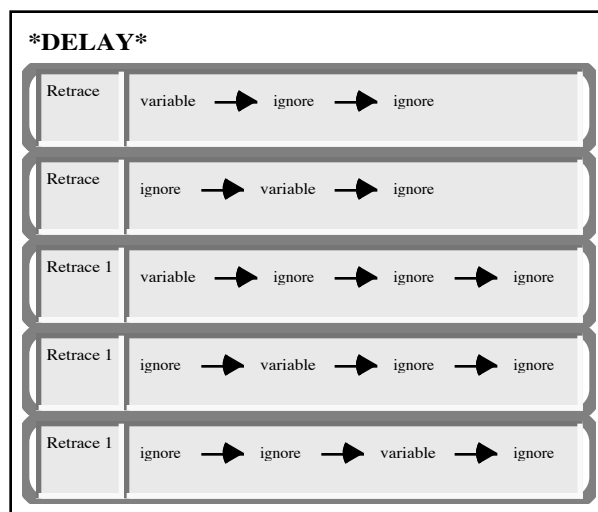


Figure 7.3. 10: ‘*DELAY’/2 facts for ‘Retrace’/3 and ‘Retrace 1’/4 predicates.

The definitions of the ‘Preserved’/2 and ‘Preserves’/2 predicates are shown in Figure 7.3. 12. These two predicates do essentially the same thing, with the difference that ‘Preserved’/2 takes a single fact as its first argument and ‘Preserves’/2 takes a set of facts as its first argument. Both predicates check that the given action does not “delete” the

given fact or facts. Thus, the given fact (or facts) is preserved by the given action (i.e. not deleted). The delay specifications cause literals of either predicate to be delayed if either their first or second arguments are unbound variables.

The definition of the 'And'/'3 and 'And 1'/'3 predicates are shown in Figure 7.3. 11:

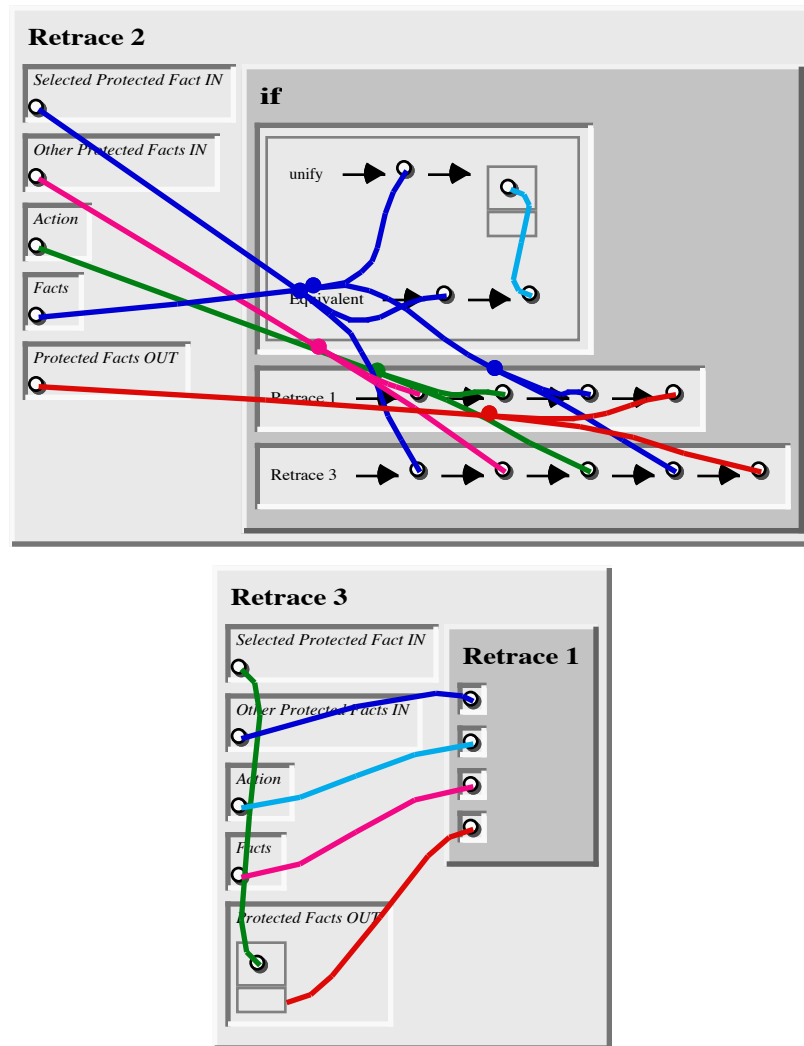


Figure 7.3. 11: 'Retrace 2'/'5 and 'Retrace 3'/'5 predicate definitions.

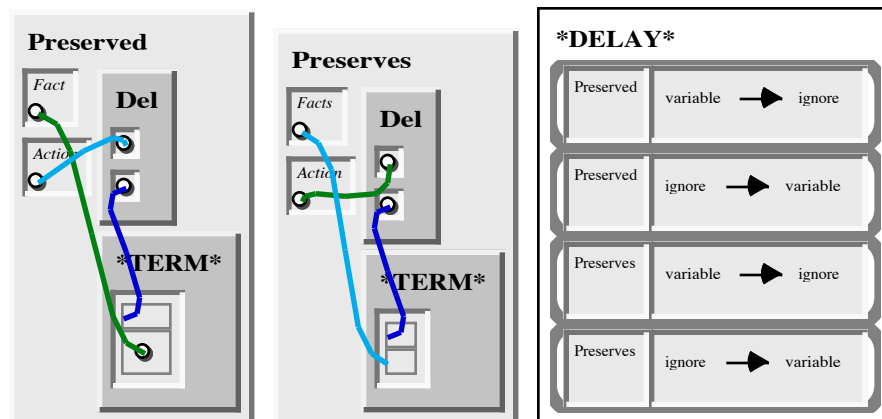


Figure 7.3. 12: 'Preserved'/'2 and 'Preserves'/'2 predicate definitions.

‘And’/3 combines a given goal (or “fact”) and a given set of protected facts to produce a new set of protected facts. The given goal is combined with the given protected facts if it is not equivalent to some fact in the given protected facts. ‘And 1’/3 adds the given goal to the given fact set.

The definition of ‘Consistent’/2 is on the left of Figure 7.3. 14. It checks that there are no impossible combinations in the given combined facts. It determines this by checking if ‘Inconsistent’/2 fails.

The definition of the ‘Inconsistent’/2 predicate is on the right Figure 7.3. 14. It is true when there is an impossible combination of facts in the given combined facts. The given new facts (which are included in the given combined facts) are used to limit the number of impossible combinations which are checked—only impossible combinations which include a new fact are tried. This predicate assumes the facts in the combined facts that are not the new facts have already been checked for consistency and thus that any impossibility which is to be discovered must include one or more new facts. ‘Inconsistent’/2

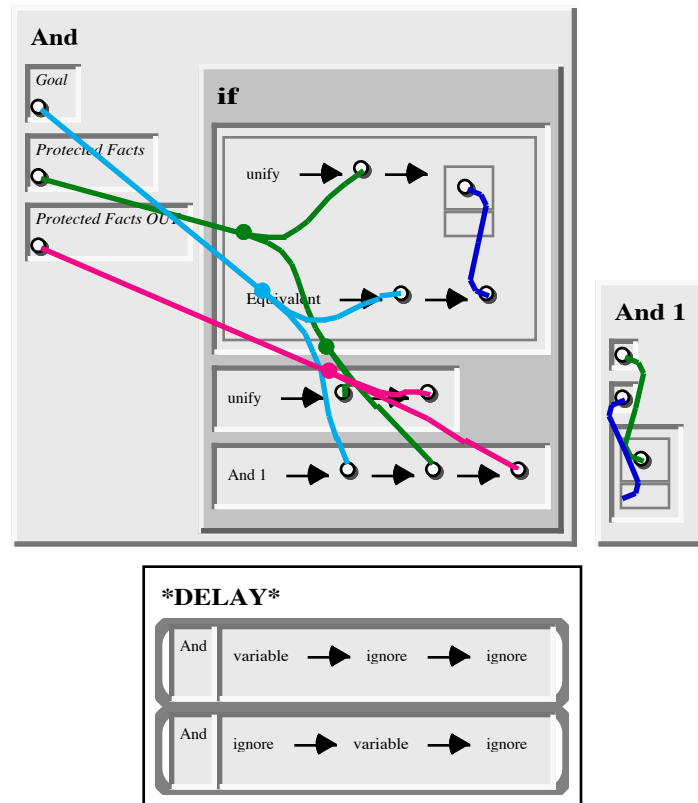


Figure 7.3. 13: ‘And’/3 and ‘And 1’/3 predicate definitions.

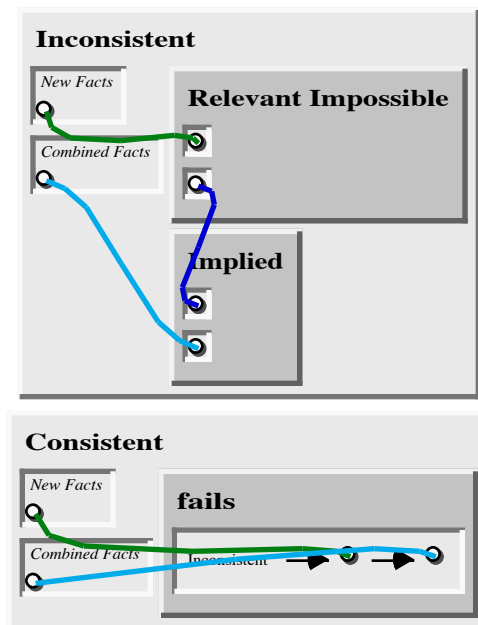


Figure 7.3. 14: ‘Consistent’/2 and ‘Inconsistent’/2 predicate definitions.

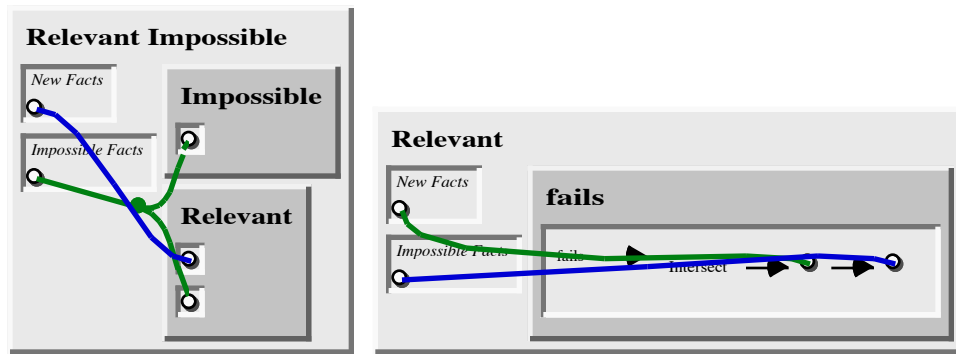


Figure 7.3. 15: ‘Relevant Impossible’/2 and ‘Relevant’/2 predicate definitions.

uses ‘Relevant Impossible’/2 to find an impossible fact “pattern” (set of impossible facts which may contain unbound variables, i.e. be nonground). It uses ‘Implied’/2 to see if the relevant impossible fact set is “implied” by the given combined facts.

The definition of the ‘Relevant Impossible’/2 predicate is shown in Figure 7.3. 15.

DELAY	
Consistent	variable → ignore
Consistent	ignore → variable
Inconsistent	variable → ignore
Inconsistent	ignore → variable
Intersect	variable → ignore
Intersect	ignore → variable
Relevant	variable → ignore
Relevant	ignore → variable

Figure 7.3. 16: “*DELAY*/2 facts for ‘Consistent’/2, ‘Inconsistent’/2, ‘Intersect’/2, ‘Relevant’/2, and ‘Implied’/2.

It finds an impossible fact set which is “relevant” to the given new facts. It uses the ‘Impossible’/1 literal to find impossible fact sets. The ‘Relevant’/2 literal determines if the given new facts are “relevant” to an impossible fact set.

The definition of the ‘Relevant’/2 predicate is shown in Figure 7.3. 16. It is true if the given new fact set has some fact in common with the given impossible fact set. ‘Relevant’/2 determines this using an ‘Intersect’/2 literal. The ‘Intersect’/2 literal is “wrapped” by two ‘fails’/1 predicates. This is logically similar to the ‘Intersect’/2 literal by itself, that is the ‘fails’ of ‘fails’ of ‘Intersect’ is true exactly when ‘Intersect’ alone is true. The difference is that using the double ‘fails’ construction undoes any variable bindings which the interpretation of the ‘Intersect’/2 literal may have made.

This is a performance optimization which avoids the need for a choice point in ‘Intersect’/2. The binding isn’t necessary since the later use of the ‘Implied’/2 predicate will do whatever bindings are required to establish an impossibility. The ‘Intersect’/2 and ‘Implied’/2 predicate definitions are shown in Figure 7.3. 17.

The definition of the ‘Equivalent’/2 predicate is shown in Figure 7.3. 18. This predicate establishes that two terms are equivalent if they cannot be unified with terms which are in different parts of a partitioned set. Since parts of a partitioned set are disjoint, the terms in the different parts must be different. The only way they can not be different is if they are identical. Thus for two terms to be equivalent they must be identical—if they contain variables, the same variable must be in corresponding positions of the terms. Two terms may unify and not be equivalent—they may have some different variables but in positions that make some equivalence-inducing substitution possible. If two terms are equivalent then they are unifiable, but not the converse. The delay specification causes the interpretation of an ‘Equivalent’/2 literal to be delayed if either of its arguments are unbound variables.

The definition of the ‘Union Facts’/3 predicate is shown in

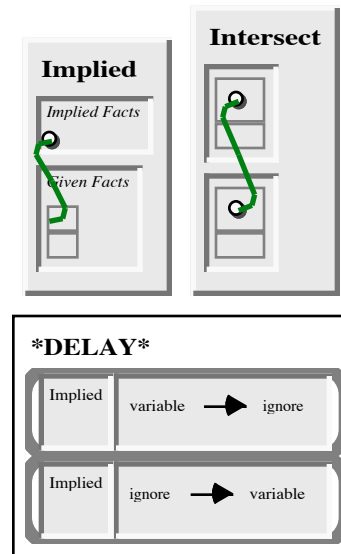


Figure 7.3. 17: ‘Implied’/2 and ‘Intersect’/2 predicate definitions.

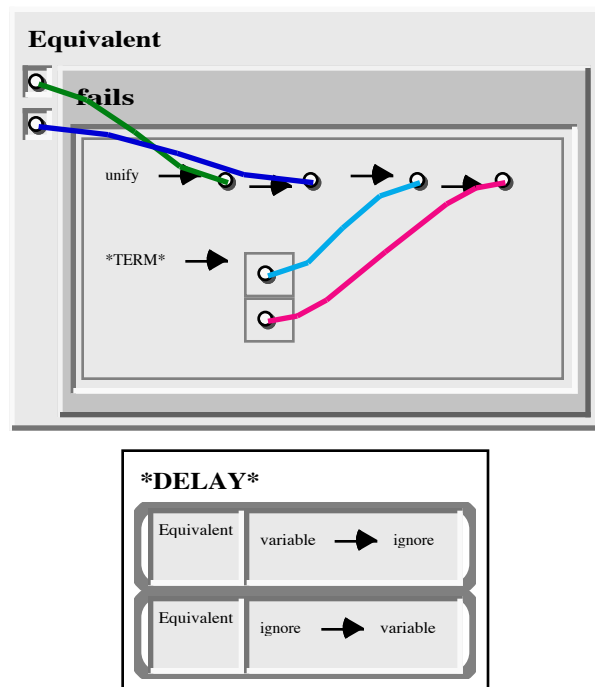


Figure 7.3. 18: ‘Equivalent’/2 predicate definition.

Figure 7.3. 19. The ‘Union Facts’/3 predicate is the same as the ‘Union’/3 predicate discussed in chapter 3 (“Design Elements”), except that ‘Union Facts’/3 has a delay specification. This delay specification causes the interpretation of ‘Union Facts’/3 to be delayed if either of its first two arguments are unbound variables.

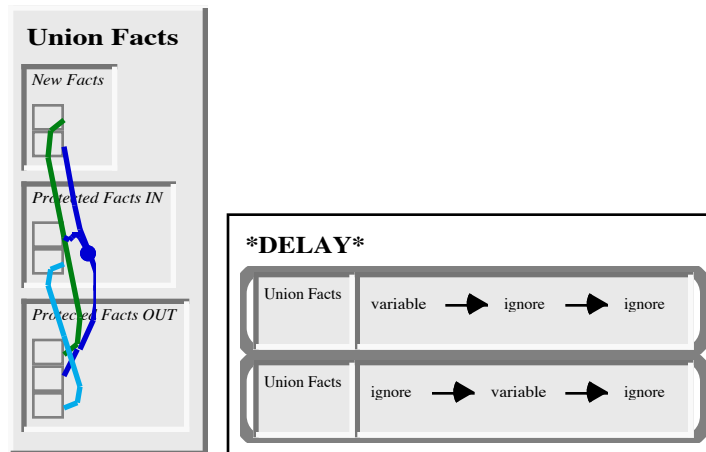


Figure 7.3. 19: ‘Union Facts’/3 predicate definition.

The definition for the ‘Add’/2 predicate of the blocks world is shown in Figure 7.3. 20. These clauses specify that “move A from B to C” adds the facts “A is on C” and “B is clear”. An ‘Add’/2 literal is delayed if both of its arguments are unbound variables.

The definition of the ‘Can’/2 predicate of the blocks world is shown in Figure 7.3. 21. This predicate specifies that we can “move A from B to C” if “A is on B, A is clear, and C is clear”, where B and C must be different.

The definition of the ‘Del’/2 predicate is shown in Figure 7.3. 22. This predicate specifies that “move A from B to C” deletes the facts “A is on B”, and “C is clear”. A ‘Del’/2 literal is delayed if its first argument is an unbound variable.

The definition of ‘Given’/2 for the blocks world is shown in Figure 7.3. 23. This associates an initial state with the name “start”. The initial state is “a is on 1”, “b is on 2”, “c is on a”, “c is clear”, “b is clear”, and “3 is clear”. The numbers indicate places on the

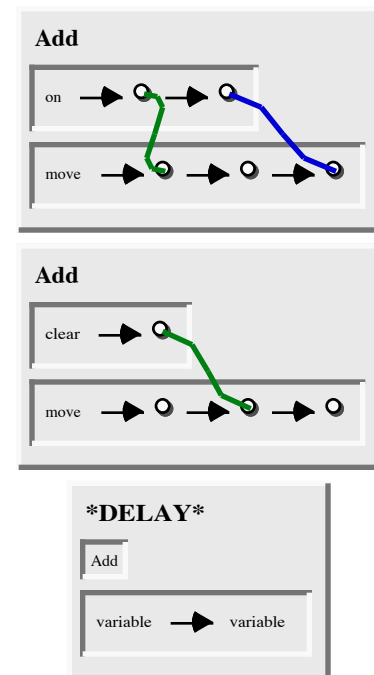


Figure 7.3. 20: ‘Add’/2 predicate definition.

floor at which blocks may be put. The letters are names of blocks. Interpretation of a ‘Given’/2 literal is delayed if the first argument is an unbound variable.

The definition of the ‘Impossible’/1 predicate for the blocks work is shown in Figure 7.3. 24. The clauses for this predicate define three

Figure 7.3. 21: ‘Can’/2 predicate definition.

different impossible fact patterns: “A on B” and “clear B”; “A on X” and “A on Y” where X and Y are different; and “A on A”.

Discussion. In this section we have presented the *WARPLAN* problem and discussed the SPARCL solution of it. This implementation is substantially larger than that for ID3:

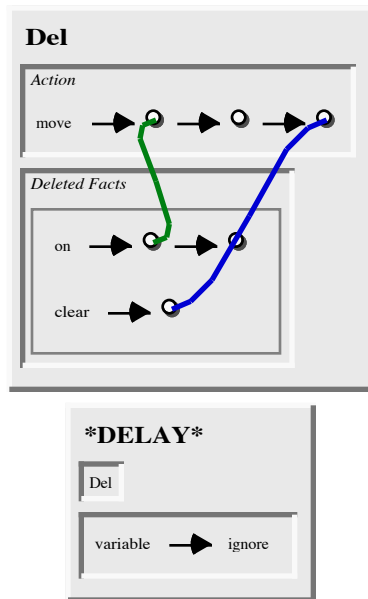


Figure 7.3. 22: ‘Del’/2 predicate definition.

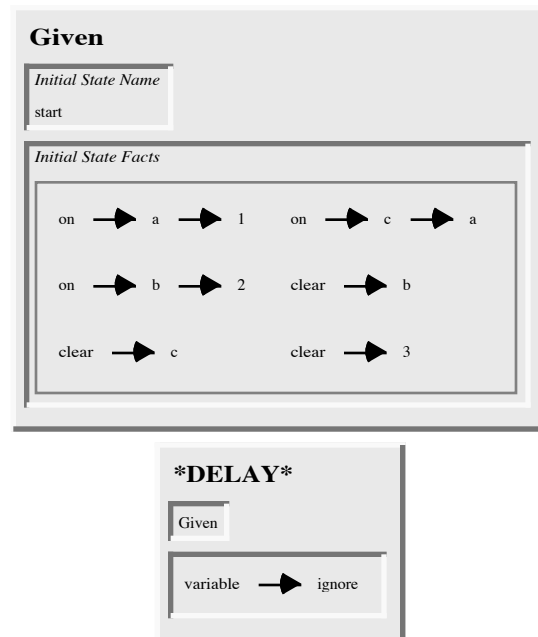


Figure 7.3. 23: ‘Given’/2 predicate definition.

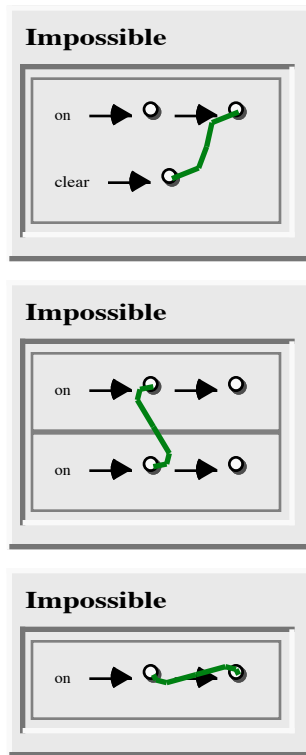


Figure 7.3. 24:
'Impossible'/1 predicate definition.

the WARPLAN “planner” has 15 predicates implemented by 23 regular clauses and 34 ‘*DELAY*’/2 clauses (compared to ID3’s 15 regular clauses and 18 ‘*DELAY*’/2 clauses). There are an additional five predicates defined in specifying the “blocks world” domain for the planner. These are implemented by 7 regular clauses and 4 ‘*DELAY*’/2 clauses.

The implementation of WARPLAN in SPARCL is a meta-interpreter for planning. The SPARCL implementation differs from the PROLOG one in several ways, one is that the SPARCL implementation works with unordered sets of facts, where in the PROLOG implementation the facts are ordered. The SPARCL implementation works with these sets in several places: ‘Solve’/5, ‘Holds’/2, ‘Retrace 1’/4, ‘Retrace 2’/5, ‘Preserved’/2, ‘Preserves’/2, ‘And’/3, ‘And 1’/3, ‘Intersect’/2, ‘Implied’/2, ‘Union Facts’/3, ‘Can’/2, ‘Del’/2, and ‘Given’/2. In each of these places it allows for a simpler expression compared to the ordered representation of facts.

The ‘if’/3 built-in predicate is used in the definitions of ‘Retrace 1’/4 and ‘Retrace 2’/5, shown in Figure 7.3. 9 and Figure 7.3. 11. This allows us to implement a “case” where the cases are tested only once, rather than testing a case and then testing its negation, which a “purely” logical approach in SPARCL would require.

The ‘Preserves’/2 predicate in Figure 7.3. 12 provides an example of using a partitioned set in a ‘*TERM*’/1 literal to specify that two sets must be disjoint. The ‘*TERM*’/1 literal is a “dummy” that is there simply to provide a place to put the constraining partitioned set. The ‘*TERM*’/1 predicate is a built-in that is always true. It was defined for just this purpose, as a holder for terms. The partitioned set is only interesting in this clause for the requirement that its parts be disjoint, the union of the parts is not used. We are considering modifying the representation of ‘*TERM*’/1 literals so that only the term in its argument is displayed “free floating” among the literals of the clause body. This would require various mostly minor adjustments to the editing and display systems.

The ‘Equivalent’/2 predicate in Figure 7.3. 18 shows another use of a partitioned set for the disjointness constraint on its parts. This predicate essentially implements the ==/2 (“identical” terms) predicate of PROLOG. without introducing any new “extra-logical” predicates. This says that two terms are identical if it is not possible to unify them with two terms which could possibly be distinct. The “two terms which could possibly be distinct” portion of the predicate is provided by the partitioned set in the argument to ‘*TERM*’/1.

The ‘Can’/2 predicate in Figure 7.3. 21 is a final example of the use of a partitioned set for its part-disjointness constraint, where we require that the block that is being moved must be different from the place to which it is being moved.

A full use of the partitioned set constraints, both disjointness and union, is seen in the definition of ‘Impossible’/1 in Figure 7.3. 24. We use the union constraint to require that two “on” facts are in the same set of facts. The disjointness constraint is used to require that the third elements of these are different. They must be different since the other two elements are specified to be the same (same name for the first elements and a coreference link for the second elements) and the two facts are in different parts of the partitioned set.

The use of coreference links provides the same services in the *WARPLAN* solution as it did in the *ID3* solution: easy decoding of links involving several terms such as the four term links in the “plan” clause of ‘Achieve’/n in Figure 7.3. 3, easy recognition that all links are simple such as in ‘Plan’/4 in Figure 7.3. 4, and connections between interior portions of complex data structures as in Figure 7.3. 21. The clauses of this solution did not make any use of connecting to multiple aspects of the same complex term.

- Ernst&Newell 1969 *GPS: A case study in generality and problem solving*. by G. Ernst and A. Newell. New York: Academic Press, 1969.
- Fikes&Nilsson 1971 “STRIPS: A new approach to the application of theorem proving to problem solving” by R. E. Fikes and Nils J. Nilsson. Pages 189-208 in *Artificial Intelligence*, 2(3-4), 1971.
- Green 1969 “Application of Theorem Proving to Problem Solving” by Cordell Green. Originally published as pages 219-239 in *Proceedings of the First International Joint Conference on Artificial Intelligence*, Washington, DC, 1969. Also published as pages 202-222 in *Readings in Artificial Intelligence*, edited by B. L. Webber and N. J. Nilsson, Los Altos, California:Morgan Kaufmann, 1981.
- Rich&Knight 1991 *Artificial Intelligence, Second Edition* by Elaine Rich and Kevin Knight. McGraw-Hill, Inc.:New York. 1991.
- Sussman 1975 “A Computer Model of Skill Acquisition” by G. J. Sussman. Cambridge, MA:MIT Press. 1975.
- Warren 1974 “WARPLAN - a system for generating plans.” by David H. D. Warren. DAI, Memo 76. University of Edinburgh. 1974.

4. Self Interpreter.

SPARCL is designed to help in using exploratory programming to find solutions to complex programming problems. A major technique for working on a complex problem is to create a new programming language specialized for that problem. This is called metalinguistic abstraction by Sussman and Abelson. If we have an implementation of a language and write an interpreter for a second language in the first language, then the second language is an *embedded language*. If the interpreter is for the language in which it is written, then it is a *metacircular* interpreter since the language is now embedded in itself.¹

Metacircular interpreters, or simply “meta-interpreters”, are a powerful programming technique particularly appropriate to exploratory programming. Leon Sterling and Ehud Shapiro describe the value of meta-interpretation as follows:

The ability to write a meta-interpreter easily is a very powerful feature a programming language can have. It enables the building of an integrated programming environment and gives access to the computational process of the language.²

A meta-interpreter in a logic programming language can be considered an implementation of a meta-level architecture for a reasoning system. Meta-level architectures are a popular approach to implementing reasoning systems because they separate the domain information from the control information, and because the control information is explicitly represented instead of being implicit in the implementation of the system. Advantages stemming from this separate and explicit representation of control information are summarized by [van Harmelen 1989]. Thus another value of the ability to write a meta-interpreter easily is that it makes implementing meta-level architectures easier. A classification of meta-level architectures is given in [van Harmelen 1989].

A “self” interpreter is the most basic of meta-interpreters, it is an interpreter which interprets the language in which it is written without adding to it or changing it in any way. A debugger written in the language for which it is a debugger is an example of a more complex kind of meta-interpreter, one that interprets the language in which it is written and adds additional services such as printing out execution traces

1. See pages 294-295 of [Abelson et al. 1985].

2. p. 303 in [Sterling&Shapiro 1986].

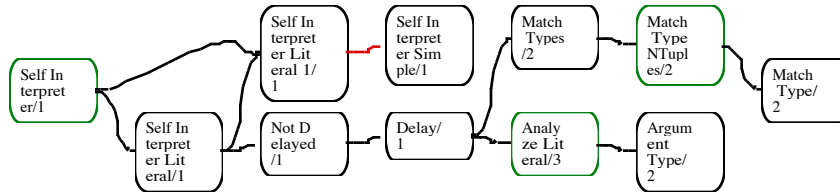


Figure 7.4. 1: Overview of the implementation of the SPARCL self interpreter.

or interactive control of program interpretation. In this section we present a self interpreter for SPARCL. Since self-interpretation poses unique problems for each programming language in which it is solved (and it cannot be solved “easily” in all programming languages), we have not provided solutions in PROLOG or LISP (although self-interpreters are certainly possible in those languages).

The classification of meta-level architectures by van Harmelen divides them into three kinds based on “where” they do inferences: object-level inference, mixed-level inference, and meta-level inference. The meta-level inference category is divided into three sub-categories: monolingual, amalgamated, and bilingual. The ‘Self Interpreter’/1 predicate implements a monolingual meta-level inference system. A *monolingual* implementation uses the object-level language as the language in which the meta-level inference is implemented. In this case no distinction is made between object-level and meta-level expressions. A different approach to Self Interpreter could have used meta-level constants to represent object-level variables and provided some naming scheme to translate between meta-level and object-level expressions. This would be a *bilingual* approach. The *amalgamated* approach is one where the two languages are the same, but there is still a naming distinction.

The overview of the self interpreter is shown in Figure 7.4. 1. This version of the self interpreter does not quite do the whole language, although it is simple to extend it. The extensions are all in the ‘Self Interpreter Simple’/1 predicate.

The self interpreter is mostly contained in the predicates ‘Self Interpreter’/1, ‘Self Interpreter Literal’/1, ‘Self Interpreter Literal 1’/1, and ‘Self Interpreter Simple’/1. The other predicates are all part of the handling of delaying literals. The ‘Self Interpreter’/1 predicate is directly recursive. The ‘Self Interpreter Literal 1’/1 predicate uses ‘Self Interpreter’/1. Other directly recursive predicates are ‘Analyze Literal’/3 and ‘Match Type NTuples’/2.

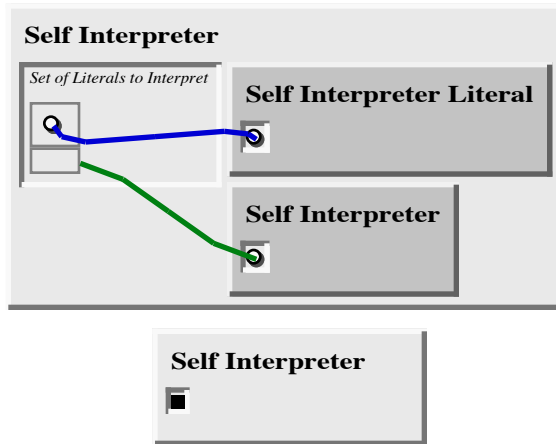


Figure 7.4. 2: ‘Self Interpreter’/1 predicate definition.

The definition of the ‘Self Interpreter’/1 predicate is shown in Figure 7.4. 2. This predicate is true when all of the given literals are solved. It solves a set of literals by solving any one of them with a ‘Self Interpreter Literal’/1 literal and solving the rest of them with a ‘Self Interpreter’/1 literal. If there are no literals to solve, then ‘Self Interpreter’/1 is true.

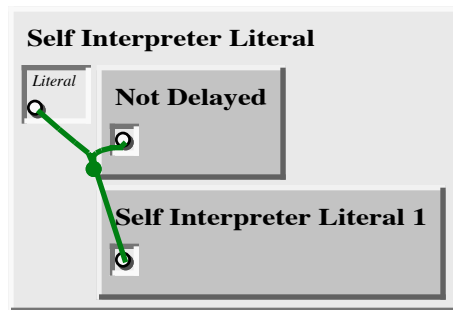


Figure 7.4. 3: ‘Self Interpreter Literal’/1 predicate definition.

The definition of the ‘Self Interpreter Literal’/1 predicate is in Figure 7.4. 3. A literal is solved if the literal is not delayed and ‘Self Interpreter Literal 1’/1 holds for that literal. The ‘Self Interpreter Literal 1’/1 predicate holds in one of seven ways: (1) there is a clause with a head that unifies with the literal, and the body of that clause is solved (in Figure 7.4. 4); (2) the literal is a

“simple” (built-in) literal; (3) the literal is a 2-tuple with ‘fails’ as the first element,

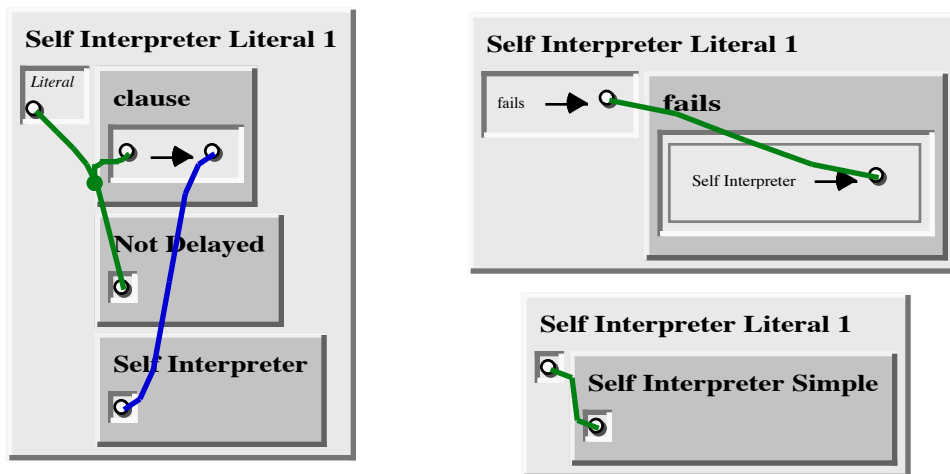


Figure 7.4. 4: ‘Self Interpreter Literal 1’/1 predicate definition, part 1.

and ‘fails’/1 of ‘Self Interpreter’/1 applied to the second element of the literal holds (in

Figure 7.4. 4); (3) the literal is a 5-tuple with ‘setof’ as its first element and ‘setof’/4 holds with a first argument the same as the second element, second argument same as the third element, third argument a set containing an

ordered pair of ‘Self Interpreter’ and the fourth element, and

the fourth argument same as the fifth element (in Figure 7.4. 5); (5) the same approach as for (4), but with ‘multisetof’ instead of ‘setof’ (in Figure 7.4. 5); (6) the literal is a 4-tuple with its first element ‘if’, and ‘if’/3 holds with first argument the a set containing a 2-tuple of ‘Self Interpreter’ and the second element of the literal, the second argument is a set containing a 2-tuple of ‘Self Interpreter’ and the third element of the literal, and the third argument similarly wraps the fourth element of the literal (in Figure 7.4. 6); and, (7) the literal is a 3-tuple with ‘ordered_disjunction’ as its first element and ‘ordered_disjunction’/2 holds with its first argument a set containing a 2-tuple of ‘Self Interpreter’ and the second element of the literal and its second argument a set containing a 2-tuple of ‘Self Interpreter’ and the third element of the literal.

The ‘Self Interpreter Simple’/1 predicate is shown in Figure 7.4. 7. This predicate defines several of the “built-in” predicates of SPARCL. We show a selection of the built-ins; those that are used by the Self Interpreter and two additional common ones

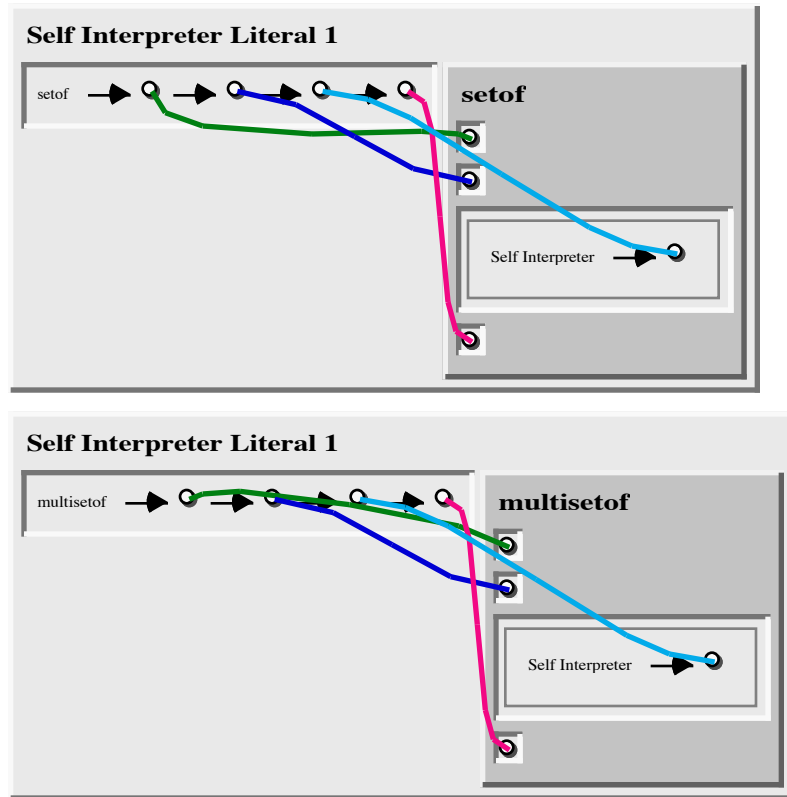


Figure 7.4. 5: ‘Self Interpreter Literal 1’/1 predicate definition, part 2.

(‘unify’/2 and ‘is’/2).

The definition of the ‘Not Delayed’/1 and ‘Delay’/1 predicates are shown in Figure 7.4. 8. A literal is “not delayed” if there is no delay specification that matches it. The ‘Delay’/1 predicate tries to find a delay specification that matches the given literal. It uses ‘Analyze Literal’/3 to extract the predicate name and the literal argument types from the given literal. It uses ‘Match Types’/2 to determine if there is a delay specification for the extracted predicate name with matching types.

The definition of the ‘Analyze Literal’/3 predicate is shown in Figure 7.4. 9. ‘Analyze Literal’/3 processes each of the given literal argument types to determine the appropriate type (“variable”, “nonground”, or “ground”). The result, in the third argument, is a list of argument types corresponding to the literal argument terms (i.e. an N-tuple with first element of “empty_list”).

The definition of the ‘Argument Type’/2 predicate is shown in Figure 7.4. 10. This predicate determines if the given term in the first argument is “variable”, “ground”, or “nonground”. It uses the two built-in predicates ‘is_variable’/1 and ‘is_ground’/1 to do this.

The definition of the ‘Match Types’/2 predicate is shown in Figure 7.4. 11. This predicate is true if there is a delay specification for the given predicate name that has argument type specifications matching the given literal argument types. The delay

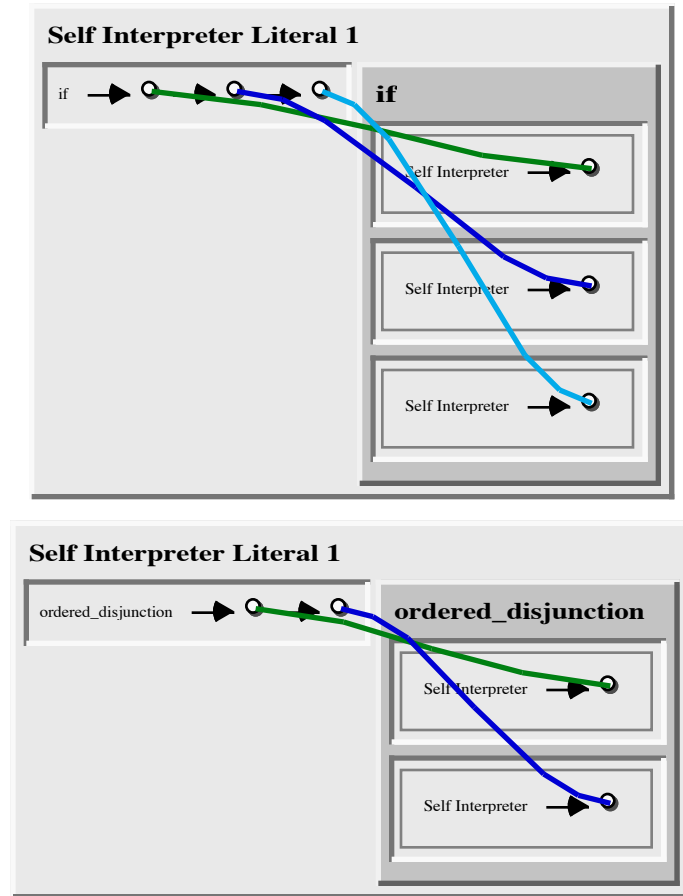


Figure 7.4. 6: ‘Self Interpreter Literal 1’/1 predicate definition, part 3.

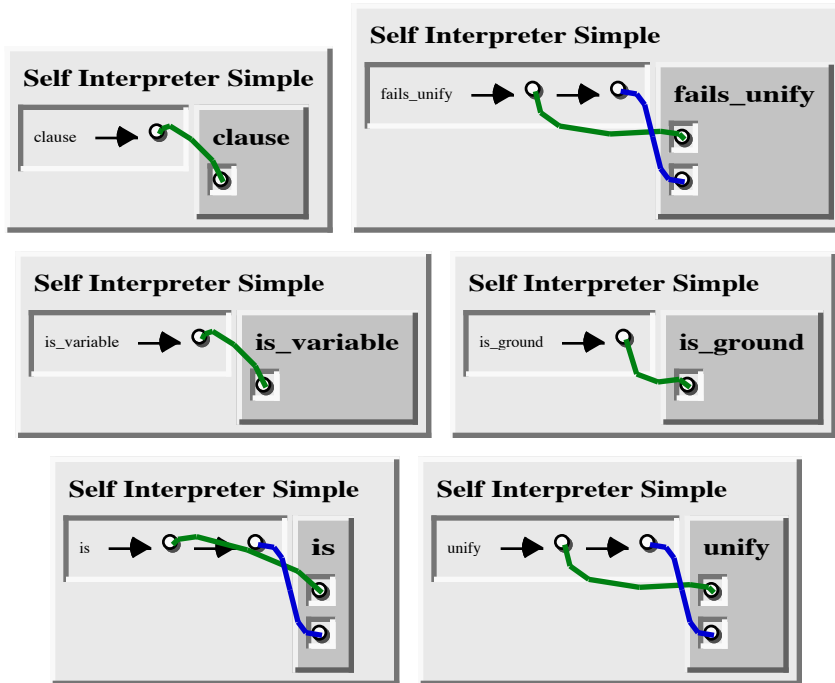


Figure 7.4. 7: 'Self Interpreter Simple'/1 predicate definition.

specification is found by the 'clause'/1 predicate with a suitably constructed clause term. The argument type matching is done by 'Match Type NTuples'/2.

The definition of the 'Match Type NTuples'/2 predicate is shown in Figure 7.4. 12. This predicate is true when each of the corresponding elements of the literal argument types list and the type specifications N-tuple match. A literal argument type and an argument type specification match if 'Match'/2 is true of them.

The definition of the 'Match Type'/2 predicate is shown in Figure 7.4. 13. A literal argument type and a type specification match if they are the same or if the specification is "ignore".

Figure 7.4. 14 shows the '*DELAY*'/2 facts for the predicates of the sparcl self interpreter. 'Self

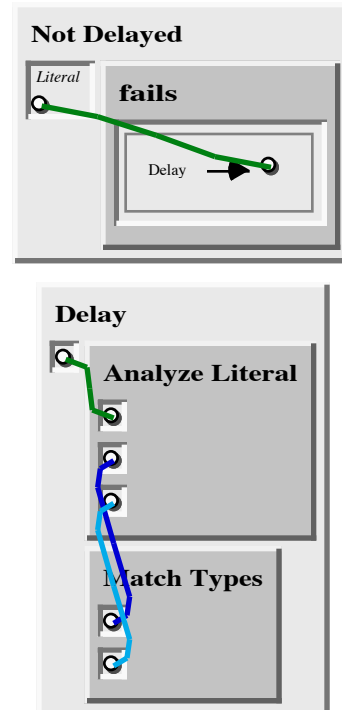


Figure 7.4. 8: 'Not Delayed'/1 and 'Delay'/1 predicate definitions.

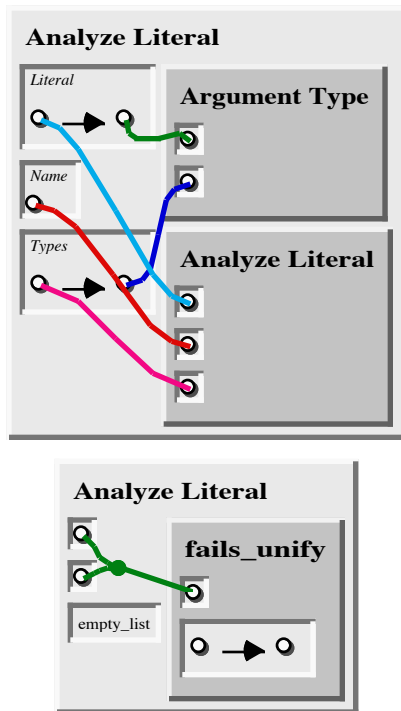


Figure 7.4. 9: 'Analyze Literal'/3 predicate definition.

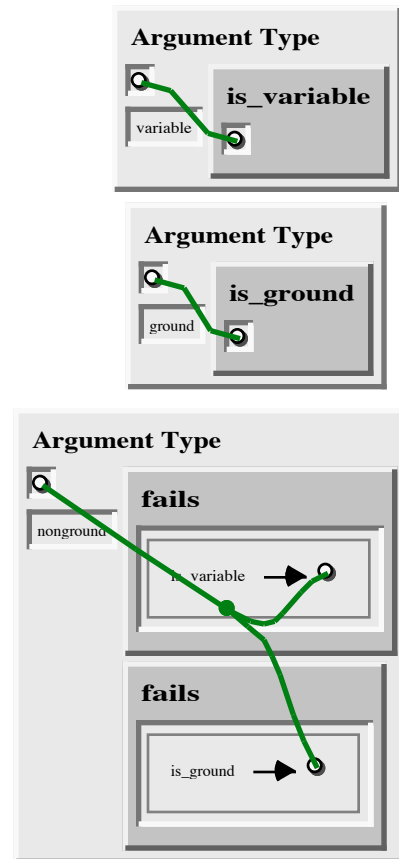


Figure 7.4. 10: 'Argument Type'/2 predicate definition.

Interpreter'/1, 'Self Interpreter Literal'/1, and 'Not Delayed'/1 literals are delayed if their single argument is an unbound variable. An 'Analyze Literal'/3 literal is delayed if its first argument is an unbound variable. A 'Match Type NTuples'/2 literal is delayed if either of its arguments is nonground, i.e. is a term that contains an unbound variable.

Discussion. The SPARCL self interpreter is reasonably short, but perhaps not as short as one might expect. A self-interpreter for "pure" PROLOG is just three short clauses as shown in Figure 7.4. 15.

But a PROLOG interpreter that handles the control predicates

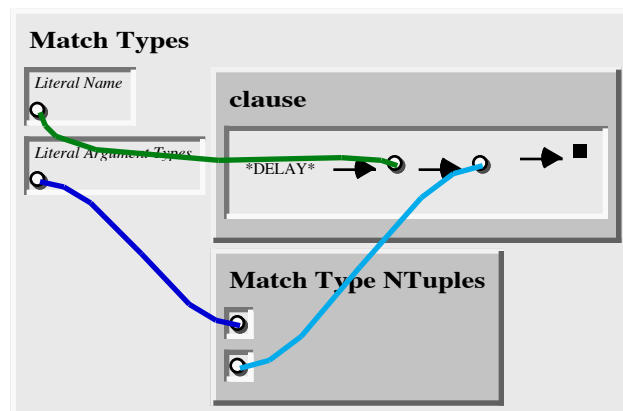


Figure 7.4. 11: 'Match Types'/2 predicate definition.

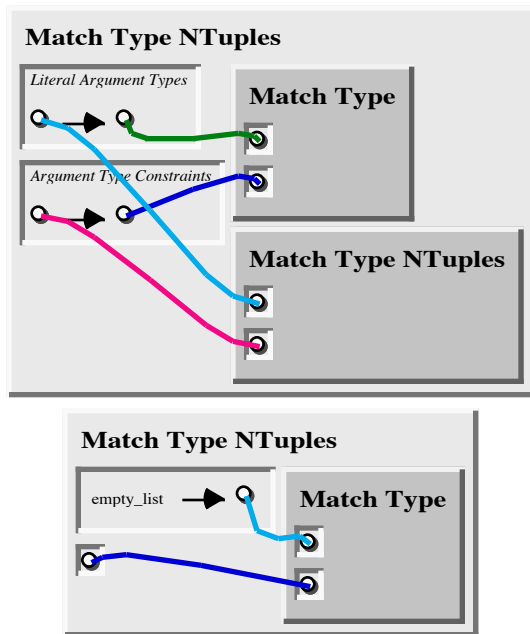


Figure 7.4. 12: ‘Match Type NTuples’/2 predicate definition.

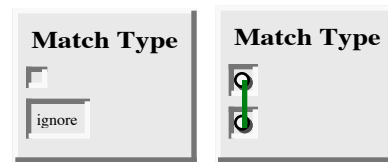


Figure 7.4. 13: ‘Match Type’/2 predicate definition.

of the language, particularly cuts (“!”), is much longer. This is shown in Figure 7.4. 16.

This solution for the PROLOG self interpreter relies on the underlying PROLOG system implementing *ancestor cut* “!(Goal)”, which is not the case in Edinburgh-type PROLOGS. For an Edinburgh-type PROLOG (such as LPA’s MACPROLOG32) the implementation of

cut is substantially more complicated. The additional length of ‘Self Interpreter’/1 is due to the implementation of the SPARCL meta-predicates and delays.

The coreference links help to make it easy to see the “wrapping” of the elements of the meta-predicates in the implementation of ‘Self Interpreter Literal 1’/1 by connecting a variable for an element with another variable buried in an N-tuple in a part of a partitioned set in an argument of a literal. This can be seen in Figure 7.4. 4, Figure 7.4. 5, and Figure 7.4. 6.

The ‘Argument Type’/2 predicate introduces the

DELAY	
Self Interpreter	variable
Self Interpreter Literal	variable
Not Delayed	variable
Analyze Literal	variable → ignore → ignore
Match Type NTuples	nonground → ignore
Match Type NTuples	ignore → nonground
Match Types	nonground → ignore

Figure 7.4. 14: “*DELAY”/2 facts for the ‘Self Interpreter’/1, ‘Self Interpreter Literal’/1, ‘Not Delayed’/1, ‘Analyzed Literal’/3, and ‘Match Type NTuples’/2 predicates.

```

solve(true).
solve((A, B)) :-
    solve(A),
    solve(B).
solve(A) :-
    clause(A, B),
    solve(B).

```

Figure 7.4. 15: Self-interpreter for “pure” PROLOG.

‘is_variable’/1 and ‘is_ground’/1 built-in predicates. These are extra-logical predicates necessary to look at the current “state” of the interpretation for determining whether or not to delay a literal’s interpretation.

5. Assessment

This chapter presents SPARCL programs solving the *ID3*, *WARPLAN*, and *Self Interpreter* problems. Also, solutions in LISP and PROLOG (as well as SPARCL) are presented for the *classify examples* portion of the *ID3* problem.

We argue that comparing the SPARCL solutions to those in LISP and PROLOG shows the SPARCL solution to be more understandable than the solutions in the other two languages. This improved understandability is due to its visual representation of coreference, its specialized representation and handling of sets, and its comparative brevity. The SPARCL solutions of the other example problems provide additional evidence of SPARCL’s brevity and understandability.

We discussed the value of meta-interpretation in the presentation of the ‘Self Interpreter’/1 predicate, which is a metacircular interpreter for SPARCL. This proves the ability of SPARCL to fully describe itself. Since SPARCL is a logic programming language, this shows that a variety of meta-level architectures for reasoning systems can be fairly directly implemented in SPARCL. The *WARPLAN* system is an example of a meta-interpreter that implements a meta-level

```

solve(true).
solve((A, B)) :-
    solve(A),
    solve(B).

solve(!) :-
    !(reduce(A)).
/* '\+' /1 is the “not” or “fails”
predicate in Edinburgh-type PROLOG.
*/
solve(\+ A) :-
    \+ solve(A).
/* Vars is a list of the existentially
qualified vars in A.
*/
solve(setof(X, Vars^A, Xs)) :-
    setof(X, Vars^solve(A), Xs).
/* if-then-else construct. There is an
implicit cut after If.
*/
solve((If -> Then ; Else)) :-
    solve(If)
    -> solve(Then)
    ;
    solve(Else).
solve((A; B)) :-
    solve(A)
    ;
    solve(B).
solve(A) :-
    reduce(A).

reduce(A) :-
    clause(A, B),
    solve(B).

```

Figure 7.4. 16: Self-interpreter for full PROLOG. (Supposing the PROLOG supports ancestral cuts “!(Goal)”.)

architecture for reasoning about problems in planning actions.

- Abelson et al. 1985 *Structure and Interpretation of Computer Programs* by Harold Abelson and Gerald Jay Sussman with Julie Sussman. The MIT Press, Cambridge, Massachusetts. 1985.
- Sterling&Shapiro 1986 *The Art of Prolog* by Leon Sterling and Ehud Shapiro. MIT Press:Cambridge, MA. 1986.
- van Harmelen 1989 “A classification of meta-level architectures” by Frank van Harmelen. Pages 105-122 in *Meta-Programming in Logic Programming* edited by Harvey Abramson and M. H. Rogers. The MIT Press: Cambridge, Massachusetts. 1989.

Chapter 8

Objective Analytic Assessment of SPARCL

There are many programming languages and it is difficult to determine the effectiveness of any one of them, absolutely or with respect to other programming languages. This is a notoriously difficult problem for programming languages with textual representations. The problem is additionally difficult for those visual representations. This chapter develops and applies two techniques for objectively analytically assessing programming languages. These are program size and diagrammatic appropriateness. We first argue for the use of program size for solutions to selected programming problems as a rough indicator of comparative quality of the programming languages in which those solutions are implemented. Then, we argue for the importance of diagrammatic appropriateness in assessing the quality of a visual representation. The rest of the chapter develops these two approaches, applies them, and draws some conclusions about SPARCL based on this application.

Programming language quality and program size. We base our assessment of programming language quality on some very general ideas of what attributes a programming language should have. Ravi Sethi [Sethi 1989] provides a concise description of this:

The language must help us write good programs, where a program is good if it is easy to read, easy to understand, and easy to modify.¹

With this idea of what is “good” in programming languages, we can determine what to measure. Norman Fenton [Fenton 1991a] provides a framework for software measurement which we can use in developing our assessment techniques. Fenton refers to [Finkelstein 1984] and [Krantz et al. 1971] for the mathematical foundations of measurement theory on which his framework is based. Fenton’s framework is as follows²:

-
1. p. 4 of [Sethi 1989].. Performance is not in this list, although one might expect it to be. Performance is primarily an attribute of an *implementation* of a programming language. It is only very indirectly an attribute of a language *definition*.
 2. p.14 of [Fenton 1991a]

...we know that the first obligation in any software measurement activity is to identify the entities and attributes of interest which we wish to measure. In software, there are three classes of entities whose attributes we may wish to measure. These are:

- *Processes* which are any software related activities: these normally have a time factor.
- *Products* which are any artefacts, deliverables or documents which arise out of the processes.
- *Resources* which are the items which are inputs to processes.

Anything that we are ever likely to want to measure or predict in software is an attribute of some entity of the above three classes.

We make a distinction between attributes which are *internal* and *external*:

- *Internal attributes* of a product, process, or resource are those which can be measured purely in terms of the product, process, or resource itself.
- *External attributes* of a product, process, or resource are those which can only be measured with respect to how the product, process, or resource relates to its environment.

Our goal in this chapter is to measure the quality of a programming language. A programming language is neither a process nor a product. However, programs written in a programming language are products, and working with programs written in a programming language is a process. In Fenton's framework, the programming language is a resource, an input to the process of working with programs. A programming language is an abstract notion, so perhaps the programming language definition and program development environment are more accurately resources for the process of working with programs. The programming language definition (understood broadly as including the formal specification of the language as well as the description of the implementation of the language) is used by a programmer when creating a program—this is how the programmer knows what can and cannot be done within the programming language. The program development environment is used by the programmer whenever a program is viewed or modified.

Sethi's definition of what a language should provide provides a definition of a high quality language—the extent to which a language fulfills Sethi's criteria is the quality of that language. We do not know how to measure the quality of a programming language directly, so we use Sethi's definition to provide an indirect measurement of programming language quality. Thus, the measurement of the quality of a language is based on the measurement of the attributes of three processes: ease of reading programs, ease of understanding programs, and ease of modifying programs. Since the quality of a programming language and these three process attributes all involve people interacting with the object being measured, these attributes are all external attributes.

We have identified the external attribute we need to measure (quality) and the object (programming language) and object type (resource) with which the attribute is associated. Also, we have determined that this measurement must be indirect; it is based on measurements of three external process attributes (readability, understandability, and modifiability). Our next step is to consider analytic versus experimental approaches to measurement of these process attributes.

Two main approaches used in assessing programming languages are the analytic and experimental approaches. In the analytic approach, one analyzes descriptions of a programming language and programs written in that language. In the experimental approach one runs empirical studies involving users using (one or more implementations of) the programming language.

Each of these approaches has its strengths and weaknesses. The experimental approach has the advantage that it measures the desired attributes of a programming language in a fairly direct fashion. However, this approach is very sensitive to the particular implementation of the target programming language and the hardware and software environment in which the experiments are conducted. These aspects of the experiments are difficult to factor out to develop a critique of the “underlying” programming language. Also, this approach is expensive in several ways: it is time consuming for the researchers organizing and administering the experiments and for the people who are the subjects of the experiments; also, it requires a highly “polished” implementation to minimize the negative environmental effects.

The analytic approach is much less subject to programming language implementation details than the experimental approach. However, the analytic approach produces only indirect measures. Thus, we must infer assessments, a process fraught with difficulties of its own. However, this approach is appealing for its broad and relatively easy application.

Using Sethi’s definition of a good programming language and Fenton’s framework for software measurement, we identified three process attributes which give us a measurement of the quality of a programming language. Since these are attributes of processes, they can only be measured directly (if at all) by appropriately constructed experiments. There is no static object which can be measured to provide direct assessments of readability, understandability, and modifiability. Further, people’s actions and responses must be involved in any direct assessment of these attributes, and this also prohibits the direct analytic measurement of these attributes. Thus, to have an

analytic measurement of these attributes, we must base the definition of these measurements on some other attribute. The attributes we offer for this purpose are the size of a program and the diagrammatic/linguistic effectiveness.

The attributes which we will measure must have certain qualities: they must be analytic, they must apply to programming languages of either visual or textual representation, they must apply to programming languages of any paradigm (e.g., logic, functional, data-flow, imperative/procedural, concurrent), and they must relate to the three process attributes (readability, understandability, modifiability). There are many measurements of programs which have been proposed. Generally, these are not usable for our purposes. Measurements which rely on particular program abstractions such as control flow graphs, search trees or data flow graphs are not applicable across all programming paradigms: control flow graphs apply to procedural/imperative languages but not to logic programming languages, search trees apply to logic programming languages but not to procedural/imperative programs, data flow graphs and variable set/use analyses are inappropriate for logic programming. Also, measurements that rely on program abstractions and are thus insensitive to the concrete representation of a program do not distinguish between visual and textual representations. Thus, if one has a visual PROLOG and a textual PROLOG such that the visual PROLOG has a simple mapping into the textual PROLOG, then a concrete-representation-insensitive measurement would show no difference between these visual and textual representations of a PROLOG program. The only thing which programs in all programming languages must share is that they have some intended-for-human-manipulation concrete representation and that they are executable. So, an attribute which can be measured for a program (as opposed to the process of that program's execution) in any programming language must be an attribute of the *intended-for-human-manipulation concrete representation* of that program. One of the most basic attributes of the concrete representation of a program is its *size*. There is an approximately inverse monotonic relationship between the size of the concrete representation of a program and the three process attributes of interest: the larger the concrete representation of a program, the less readable (since there is more to read), the less understandable (since there is more to understand), and the less modifiable (since there may be more to change, and it is more difficult to determine the ramifications of a proposed change).

The step from the desired process attribute measurements for some programs to the programming language quality measurement is a large one. This is a generaliza-

tion from some examples to the programming languages as wholes. Any generalizations we draw in this fashion are at best provisional and must be heavily qualified. The best we can hope for in this chapter is to provide evidence for some conclusions about relative qualities of a few programming languages in a few programming problem domains.

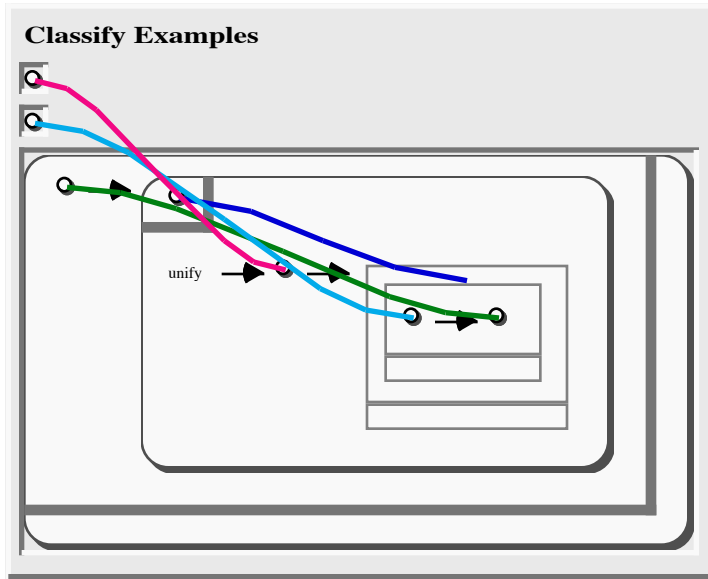


Figure 8. 1: Classify Examples SPARCL program.

An attribute of the programming language *definition* that we measure is the diagrammatic/linguistic appropriateness. This is also concerned with the concrete representation of programs: to what extent are visually concretely represented languages diagrammatic and textually concretely represented languages linguistic? This is an isolated assessment—a programming language is measured in isolation rather than relative to other languages.

Selected programming problems. This chapter presents the various approaches to measurement in detail using the solutions to the example problem of the previous chapter, *classify examples*, and the *union* problem. Also, summary results are presented and discussed for some larger problems, *ID3* and *WARPLAN*. The *classify examples* problem was explained in the previous chapter. The *union* problem is to find the set which is the union of two other sets.

The solutions in the three languages are shown in chapter 7 (“Subjective Analysis”) and appendix 3 (“Example Programs”). First, we analyze the diagrammatic appropriateness of the SPARCL representation; then, we analyze the sizes of the example solutions.

Related Work

Software measurement theory. There are many works on the application of measurement theory to software measurement, such as the work of Norman Fenton [Fenton 1991a; Fenton 1991b], Peter Bollman, V.S. Cherniavsky, and Horst Zuse [Bollman&Cherniavsky 1981; Zuse&Bollman 1987]. Zuse has written an extensive history of software measurement, [Zuse 1996], which includes the application of measurement theory. A recent review and critical analysis of the use of measurement theory in software measurement is [Briand et al. 1996]. Briand *et al.* generally approve of this approach, but caution that the very strict approach to measurement theory advocated by some, such as Zuse and Fenton, eliminates demonstrably useful applications of certain software measures having to do with software complexity.

Visual programming languages. There is little work in assessing a particular approach to representing solutions in programming languages, particularly for comparing linear and nonlinear approaches. What work there is concentrates on user testing—having people use particular programming approaches and evaluating their experiences.

Glinert discusses some approaches to metrics for “nontextual” programming environments in [Glinert 1990]. The development and application of various metrics for comparing visual and textual representations is reported in [Nickerson 1994b]. Nickerson’s work also extends that of Glinert. Work on assessing an aspect of a visual programming language (writing matrix manipulation programs) is reported in [Pandey&Burnett 1993]. The relative merits of two input devices for editing graphic diagrams is reported in [Apte&Kimura 1993].

Sun-Joo Shin presents a unique approach to comparing diagrammatic and linguistic representations of logic in [Shin 1994]. She is concerned with the logical status of diagrams; in particular, can one reason validly using only diagrams? Historically, mathematicians have believed that one cannot. Shin shows that two diagramming systems (based on Venn diagrams) are logically sound and complete, in the process showing how to approach proving soundness and completeness for diagramming systems in general. She also provides an analysis of some essential differences between diagrammatic and linguistic representations.

Linear programming languages. There is considerable work on assessing programs written in linear languages. The vast majority of this work is focused on procedural/imperative languages with destructive assignment, such as PASCAL, FORTRAN, C, and COBOL. A good summary of this work is in [Conte et al. 1986].

Diagrammatic Versus Linguistic Representation

Shin's analysis of the differences between diagrammatic and linguistic representations in [Shin 1994] provides a starting point for an assessment of the extent to which SPARCL's representation takes advantage of its visual nature. That is, to what extent its representation is diagrammatic versus linguistic. This is an important issue in evaluating a visual language. Since a visual approach is more complex to implement and more computationally intensive than a textual approach, there must be some compensatory benefit to the visual approach to justify using it.

Shin provides three ways in which diagrammatic and linguistic representations differ: relations among objects, conjunctive information, and tautologies and contradictions. She is clear that a representation is not necessarily diagrammatic simply because it is visual, neither is a representation necessarily linguistic because it incorporates text.

A linguistic visual representation is essentially no more than an illustration for some textual representation, where the information in the visual representation is presented in essentially the same fashion as in the corresponding textual representation. A rebus puzzle is a common example of a visual representation which is essentially linguistic instead of diagrammatic. In such a puzzle some of the words of a phrase are replaced by pictures which are evocative of the replaced words meaning or pronunciation.

Distinguishing between diagrammatic and linguistic representations. One of Shin's key distinctions between diagrammatic and linguistic texts is diagrammatic texts rely more on the reader's "perceptual inferences" for understanding than do linguistic texts. Linguistic texts rely more instead on the conventions of the associated representation system being known to the reader. Generally a visual representation system is not wholly diagrammatic or linguistic, but is some of each.

Perceptual inference is the information one extracts more or less directly by the act of perception—no symbol system is involved. Perceiving that a picture is of a known person is a perceptual inference; interpreting a string of characters as a message is not a perceptual inference but rather relies on conventions.

One way in which a representation relies on perceptual inference (instead of convention) is by using spatial relations to model relations among objects. The connection between a spatial relation and some other relation is a matter of convention, but the understanding of the spatial relation is a perceptual inference. Shin notes that if the "member of" relation is modeled by putting a dot in a box (the dot being a member of the set represented by the box), then:

‘...this isomorphism between the spatial arrangement and the “member of” relation is more perceptually obvious than any linguistic symbol that a linguistic representation adopts, since no extra convention involving syntactic devices is required.’³

Conjunctive information is Shin’s second way in which a representation may rely more on perceptual inference. In a diagrammatic representation, the conjunction of facts is represented by simply representing the facts to be conjoined in the same text, no additional representational device is required. In a linguistic representation, there must be some kind of device to represent conjunction (such as ‘&’, ‘and’, or ‘^’). Thus, conjunction is a perceptual inference in a diagrammatic representation but is a matter of convention in a linguistic representation.

The third distinction between diagrammatic and linguistic representations is in their handling of tautologies and contradictions. As Shin says: “Tautological information is vacuously true and contradictory information is always false.”⁴ She claims that diagrammatic systems represent tautologies and contradictions in a more perceptually obvious way than linguistic systems do. A diagram which contains no representing fact is tautologous. However, in a linguistic system a tautology must be represented by a nonempty string, and thus is less obviously “vacuously” true. A contradiction is revealed in a diagram by a violation of some basic rule for valid diagrams, such as no two elements may be in the same location in a diagram, or the same element may not be represented in two different places in the same diagram. Such a violation is easy to perceive. In a corresponding linguistic representation it can be fairly subtle to identify a contradiction, requiring reasoning about properties of the represented relations

3. page 162 of [Shin 1994].

4. p. 167 in [Shin 1994].

(such as transitivity and symmetry). The implications of these properties are “perceptually inferred” in the diagrammatic representation.

SPARCL as a Diagrammatic System

To assess SPARCL’s diagrammatic nature, we apply each of Shin’s three distinctions. To simplify this discussion we look only at the “basic” form of SPARCL, without the multiple set representations.

Diagrammatic assessment: relations of objects. Since SPARCL is a thoroughly set-based language, the primary object relationship in SPARCL is that of set membership. We adopt the convention that a set is assigned to a certain kind of basic region, a closed curve in the shape of a rectangle, or a variable represented by a small circle. At this point, the representation is purely conventional and thus linguistic. However, the choice of primitive objects in SPARCL is more constrained than in a linguistic system in that the non-variable representation of sets is limited to closed curves, so that the representation creates an interior and an exterior. However, in a linguistic system any distinguishable symbol may be used.

The membership relationship is represented in SPARCL by the convention that any “term” spatially *in* the representation of a set is a member of that set. This convention connects membership to a spatial relation. Although somewhat arbitrary (other spatial relations can be used for membership, as Shin notes⁵), this is less arbitrary than a purely linguistic device of introducing a symbol/operator such as ‘ \in ’. The use of the spatial relation “appeals to our natural perceptual ability”⁶. Several other membership relationships in SPARCL are represented using the “spatial inside” relationship, including: clauses in programs, literals in clause bodies, predicate name in clause and literal, and arguments in clause and literal.

Another essential relationship in SPARCL is that of the partitioning of a set, such as



, a partitioned set with two parts. A partitioning conveys two kinds of information: that the union of the parts equals (covers) the entire set being partitioned, and that the parts are pairwise disjoint. The representation of sets which are parts of a par-

5. On page 171 of [Shin 1994], Shin mentions a system by Lambert [Lambert 1990] where a set is represented by a line (segment) and all of the points on the line are in the corresponding set..

6. p. 171 in [Shin 1994].

tioning of a set is diagrammatic in SPARCL. The relation “part of a partitioning” is represented by a dotted box (the part) being inside a set (a solid box). All of the parts of a partitioning are next to each other so that they completely cover (“tile”) the area of the representation of the set: No elements of the SPARCL language can be placed outside of all of the parts of a partitioning but inside of the set being partitioned. This represents in a diagrammatic way that the union of the parts of a partitioning cover the partitioned set. It is a matter of convention that the dotted boxes are sets which are parts of a partitioning. Thus, the representation of the pairwise disjointness relationship is conventional.

Aside from the subset relation as embodied by the parts of a partitioning, SPARCL does not provide direct representation of relationships between sets such as union, intersection, and difference. One can use the partitioned set representation to construct these relationships, but it is not as immediately understandable as a direct representation would be, such as one finds in Shin’s VENN-I and VENN-II. For these relationships in SPARCL one must fall back on coreference links and other more linguistic representations: defining a SPARCL predicate which implements the relationship and using a literal to refer to this predicate. However, the use of partitioned set and coreference links does provide a somewhat diagrammatic representation of these “other” set relationships (union, difference, and intersection). One can compare the representations of Figure 8. 2 and Figure 8. 3 to see the set relationships in VENN-I and SPARCL.

The identification of a term as a predicate name is made by the convention of the term being in the upper left corner of the clause or literal box. The connection of a literal to the clauses defining that literal’s predicate is by a convention using the names of the literals and clauses. Variables are represented by convention as small circles. Ur constants are represented by themselves (i.e. text strings). The empty set constant is represented by convention by a small solid black square. This empty set representation is partially diagrammatic in that it is a rectangle, such as any (potentially) non-empty set, but it is filled in so that no term can be placed inside of it—spatially representing that it has/can have nothing in it.

Syntactically distinct terms can “refer” to semantically the same term. This is term coreference. This is represented in SPARCL by coreference links: lines drawn connecting the coreferencing terms. This is a diagrammatic representation that relies on the perceptual inference of seeing the connections. This connecting of arbitrary terms is relatively clumsy in linguistic representations. One might use a convention such as

that done with feature structures⁷ where each part of the structure may be labeled (a linguistic constant followed by a colon (':') followed by the substructure being labeled). Those features with the same label are coreferencing features.

The ordering relationship of elements of an N-tuple and arguments of a literal or a clause is represented diagrammatically. For N-tuples, the elements are horizontally adjacent with intervening arrows. For arguments, the elements are vertically adjacent.

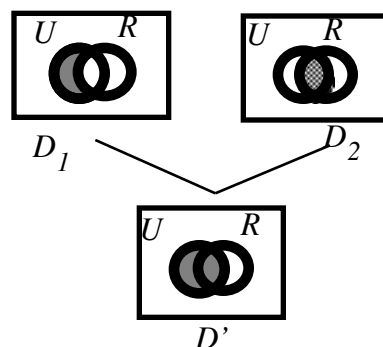


Figure 8.2: The “red unicorns” problem in the VENN-I language. The circles are sets. U = set of unicorns, and R = set of red items. Shaded areas are parts of sets with no members. D' is the union of D_1 and D_2

Diagrammatic assessment: conjunction.

This aspect of diagrammatic representation is not much used by SPARCL. Additional “facts” are added to a representation by adding additional clauses, much as is done in a linguistic representation. No additional syntax is needed, however; the additional clauses are simply visible and thus known to be in conjunction with the existing facts. This is more diagrammatic than the linguistic approach, but it is not a particularly strong use.

Shin gives an example of drawing such inferences using her VENN-I system. As shown in Figure 8.2, the facts “Every unicorn is red” and “No unicorn is red” are conjunctively combined to get a diagram from which it is obvious that there are no unicorns. This problem can be represented in SPARCL using partitioned sets, clauses, and literals as shown in Figure 8.3. In this example, ‘Red Unicorns 1’/2 expresses “Every unicorn is red” and ‘Red Unicorns 2’/2 expresses “No unicorn is red”. The combination of these statements is shown by ‘Red Unicorns’/1.

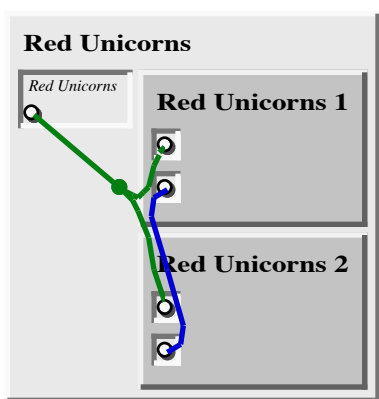
Diagrammatic assessment: tautologies and contradictions. Tautologies in general receive no special handling in SPARCL, and in this regard SPARCL is not specifically diagrammatic. An exception to this regards partitioning. It is vacuously true that a set can be partitioned, since all sets (even empty ones) can be partitioned. This is diagrammatically represented by “vacuous” parts of a partitioning having no contents,

7. pp. 105-108 in [Knight 1989].

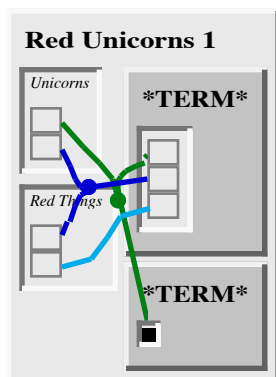
but having room in which to place terms (which is different from the empty set which, being a solid square, has no place to put anything).

Some contradictions receive a diagrammatic representation in SPARCL. Some of them are invalid SPARCL representations. Contradictions with partitioned sets would be either the parts not covering exactly the partitioned set (too large or too small), or a term appearing in more than one part of a partitioning. In the first case, the SPARCL programming environment does not allow such a construction, but it would be easily perceived should it occur. In the latter case, one can easily see that the same term is present in different parts, since the parts are spatially close together, as long as the total number of terms and parts is not large. In a linguistic representation there is no “spatial locality”, and thus the invalid sharing of a term is less readily apparent.

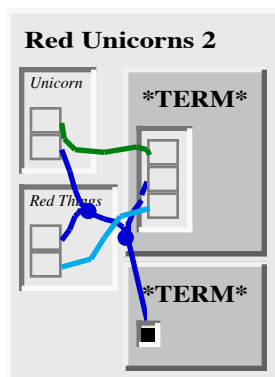
Another contradiction is when two terms which are not mutually unifiable (such as two different constants) are said to corefer. Since coreference is perceptually obvious in SPARCL, due to the use of connecting lines, then this contradiction is fairly easy to see. This is much harder to realize when reading a linguistic representation of coreferencing.



This clause finds the set of red unicorns as specified by the two "facts" modeled by Red Unicorns 1/2 and Red Unicorns 2/2. The correct answer is that the set of Red Unicorns is empty.



This clause models the "fact" that "Every unicorn is red."



The clause models the "fact" that "No unicorn is red."

Figure 8. 3: Clauses defining the 'Red Unicorns'/1 predicate.

ous in SPARCL, due to the use of connecting lines, then this contradiction is fairly easy to see. This is much harder to realize when reading a linguistic representation of coreferencing.

Diagrammatic assessment: summary. SPARCL is appropriately diagrammatic in several important aspects. Thus, the visual representation of SPARCL is beneficial compared to a strictly textual and therefore linguistic representation. Nonetheless, there are areas where more diagrammatic representation is possible; for instance, in the relationships between sets.

Measuring Program Size

Two of the sets of measurements with the best experimental verification for linear programming languages are those based on “lines of code” and the basic Halstead Software Science⁸ (SS) size measurements (vocabulary, size, and volume). Since lines of code are not measurable for visual programming languages, we use an approach based on SS. The key concepts for Software Science (SS) measurements are “token”, “operator”, and “operand”. The tokens are “the basic syntactic units distinguishable by a compiler.”⁹ Halstead’s original differentiation between operators and operands was:

“...based on the fact that all programs can be reduced into a sequence of machine language instructions, each of which contains an operator and a number of operand addresses.”¹⁰

Fenton presents a brief analysis of Halstead’s basic size measurements (size, vocabulary, and volume) and determines that, from a measurement theory point of view, these are “reasonable measures of three internal program attributes which reflect different views of *size*.”¹¹ Program size has been shown to inversely correlate with readability, understandability, and modifiability, as discussed in [Kitchenham et al. 1990] and [Conte et al. 1986], particularly for large programs.

We present two different methods to develop token counts. Both of these are based on the concrete representation: one for visual (or graphical) representations (such as SPARCL) and the other for textual representations (such as LISP and PROLOG). The visual-representation token counting is an extension of the graphic token count of Jeffrey Nickerson as initially presented in [Nickerson 1994a]. We have adapted his approach to produce operator and operand counts. The textual-representation token counting involves two adaptations of the common Halstead approach, one each for LISP and PROLOG.

Another set of token counting techniques was developed in the course of the research on which this chapter reports, but it has been discarded. It was an approach which relied on converting the concrete representation into a canonical “base” representation, then this base representation was analyzed for operator and operand

8. See pages 80-87 in [Conte et al. 1986].

9. p. 37 in [Conte et al. 1986].

10. p. 37 in [Conte et al. 1986].

11. p. 19 of [Fenton 1991a].

tokens. A special base representation was developed for each programming language, and a special volume ('V') formula was developed for each language. We have discarded this approach because it was not based on the concrete representation of the programming language and different concrete representations (particularly visual and textual representations) could be “reduced” to the same base representation. The earlier discussion of measuring the quality of programming languages pointed out that it is important for a measurement to be sensitive to the concrete representation.

In the following discussion we use the standard Software Science notation:

η_1 is the number of unique operators (operator types),

N_1 is the total number of operators,

η_2 is the number of unique operands (operand types),

N_2 is the total number of operands,

η is the vocabulary (the sum of η_1 and η_2),

N is the Software Science “size” (the sum of N_1 and N_2),

V is the volume: $V = N \cdot \log_2(\eta)$

Graphic Token-based Metrics

Jeffrey Nickerson in [Nickerson 1994b] presents four graphical metrics: graphic token count, diagram class complexity, confusion count, and (graphic) token density. We propose the following definitions based on the graphic token count and diagram class complexity:

$$\eta_1 = \begin{pmatrix} \text{number_of_edge_types} + \\ \text{number_of_label_types} + \\ \text{enclosure_present} + \\ \text{adjoinment_present} \end{pmatrix}$$

$$\eta_2 = \begin{pmatrix} \text{number_of_node_types} + \\ \text{number_of_textual_token_types} \end{pmatrix}$$

The “total” counts, N_1 and N_2 , are simple analogs of η_1 and η_2 , respectively. The terms of these equations are based on three basic attributes of graphic representation: *adjoinment* for *number_of_adjoinments*, *linkage* for *number_of_edges* and *number_of_nodes*, and *containment* for *number_of_enclosures*.¹² This approach to graphic token counting only measures the “relationships of objects” aspects of diagrammatic representation identified by Shin. No explicit counting is done for Shin’s other two aspects of diagrams (conjunctive information and tautologies and contradictions).

The following definitions are used to adapt Nickerson’s general approach to SPARCL:

Node—the term which is an endpoint of an edge

Node Type—the type of term which is a node. If the term is an Ur constant then the type of the term is the value of the constant.

Edge—a coreference hyperedge or an arrow connecting terms of an N-tuple. Each coreference hyperedge counts as 1, regardless of the number of segments.

Edge Type—there are two edge types: hyperedge and N-tuple arrow.

Textual Token—any Ur constant, clause name, or literal name.

Textual Token Type—distinct Ur constant, clause name, or literal name.

Enclosure—any display object which contains other display objects, except arguments. A set contains parts; a part may contain any number of terms (if it contains no terms it does not count as an enclosure); an N-tuple contains 2 or more terms (which are ordered by arrows); an intensional set body contains N-tuple literals, an intensional set template contains a term; a clause contains a head (arguments and name) and literals.

Enclosure Type—the type of the enclosing display object (e.g. clause, set, part of partitioned set).

Adjoinment—arguments are adjoined, term table rows/columns are adjoined, intensional set template and body are adjoined.

Adjoinment Type—the type of the display objects being adjoined (e.g. argument or N-tuple elements).

Graphic token counts for the SPARCL examples. The *Union/3* program in Figure 8. 4 has the following basic graphic token counts:

12. Nickerson’s method doesn’t seem to count the use of “similar appearance” (such as same color to indicate a “connection”). Since this is not used in SPARCL, it’s not an issue here.

nodes	7	7 endpoints of hyperedges
node types	1	1 set part
edges	3	3 hyperedges
edge types	1	
textual tokens	1	
textual token types	1	
enclosures	4	3 nonempty sets, 1 clause
enclosure types	2	
adjoinments	2	2 clause arg adj.
adjoinment types	1	

These counts yield the following values for the basic Halstead measures:

$$\begin{aligned}
 \eta_1 &= 1 + 0 + 1 + 2 = 4 & \eta &= 6 \\
 \eta_2 &= 1 + 1 = 2 & N &= 17 \\
 N_1 &= 3 + 4 + 2 = 9 & V &= 17 \log_2(5) = 44 \\
 N_2 &= 7 + 1 = 8
 \end{aligned}$$

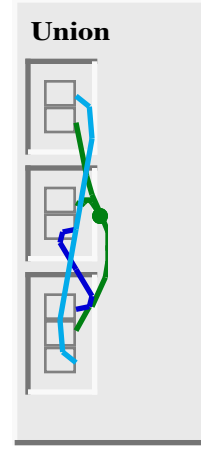


Figure 8. 4: SPARCL Union/3 program.

Abstract syntax token counting for SPARCL. The more traditional approach to defining a token counting scheme is to ignore the concrete syntax and concentrate on elements of the abstract syntax. As discussed above, we consider this approach less attractive for our purposes in this chapter as it is insensitive to details of the concrete representation; whether the concrete representation is diagrammatic or linguistic, for instance. We can define “abstract” token counts for SPARCL as:

$$\begin{aligned}
 \eta_1 &= \left(\begin{aligned} &UniqueNamesCount + SetPresent + PartPresent \\ &+ NTuplePresent + IntenSetPresent \\ &+ FactTablesPresent + TermTablePresent \end{aligned} \right) \\
 \eta_2 &= \left(\begin{aligned} &UniqueReferencesCount + UniqueUrsCount \\ &+ VarPresent + EmptySetPresent \end{aligned} \right) \\
 N_1 &= \left(\begin{aligned} &TotalNames + TotalSets + TotalParts \\ &+ TotalNTuples + TotalIntenSets \\ &+ TotalFactTables + TotalTermTables \end{aligned} \right) \\
 N_2 &= \left(\begin{aligned} &TotalReferences + TotalUrs \\ &+ TotalVars + TotalEmptySets \end{aligned} \right)
 \end{aligned}$$

The **Present* variables are 1 if an item of that kind is present in the clause and 0 otherwise. These definitions give the following counts for the ‘Union’/3 example:

$$\begin{aligned} \eta_1 &= 1 + 1 + 1 + 1 + 0 + 0 + 0 = 4 & \eta &= 7 \\ \eta_2 &= 3 + 0 + 0 + 0 = 3 & N &= 19 \\ N_1 &= 1 + 3 + 7 + 1 + 0 + 0 + 0 = 12 & V &= 19 \log_2(7) = 53 \\ N_2 &= 7 + 0 + 0 + 0 = 7 \end{aligned}$$

Token counting for Prolog. Edinburgh-syntax PROLOG (much the most common syntax for PROLOG) allows for programmer-defined operator-precedence syntax. This notion of “operator” has nothing to do with functional evaluation, it is a purely syntactic notion. The programmer may define that a particular token OP has a particular precedence PRED (an integer in an implementation-dependent range, commonly 0 to 1024), and a particular associativity and position AP. The associativity determines how unparenthesized sequences of OP associate, and the position indicates whether the OP is a prefix, suffix, or infix operator. For example, ‘xfy’ specifies an infix operator (the operator replaces the ‘f’) with right-associativity (‘a OP b OP c’ = ‘a OP (b OP c)’). Operator syntax is used as an alternative to the basic structure syntax—structures written in operator syntax are semantically identical with structures written basic structure syntax. There are several operator definitions which are in the default environment, such as:

```
op(1200, xfx, ':-').
op(1000, xfy, ',').
op(500, yfx, '+').
```

The *member* program in operator syntax is:

```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

This can also be written without operator syntax:

```
member(X, cons(X, _)).
:- (member(X, cons(_, T)), member(X, T)).
```

Since the operator syntax affects the concrete representation, the token count must be sensitive to it. The natural approach is to consider those tokens which are used syntactically as operators as operator tokens. Additional operators are the enclosing syntactic items: ‘()’, ‘[]’, and ‘{ }’. We can construct a table which defines the token counting method for PROLOG using the definitions of $S_1, \eta set_1$,

term	additions to $S_1, \eta set_1$	additions to $S_2, \eta set_2$
–		$t_{\text{underscore}}$
[]		t_{nil}
<i>symbol</i>		t_{symbol}
<i>functor(term-list)</i>	t_0	t_{functor}
<i>(term)</i>	t_0	
<i>term OP term</i>	t_{OP}	
<i>OP term</i>	t_{OP}	
<i>term OP</i>	t_{OP}	
[<i>term-list</i>]	$t_{[]}$	
[<i>term-list</i> <i>term</i>]	$t_{[], t_1}$	
{ <i>term</i> }	$t_{\{ \}}$	
<i>term</i> .	t_{period}	
<i>term</i> , <i>term-list</i>	t_{comma}	

S_2 , and ηset_2 found in **Table 8. 1: Token counting scheme for PROLOG.**

[Bieman et al. 1991]: S_1 is the sequence of operator token types in the program, ηset_1 is the set of operator token types in the program, S_2 is the sequence of operand token types in the program, and ηset_2 is the set of operand token types in the program. A PROLOG program source is a sequence of clauses, where each clause is a term followed by a ‘.’ (period). A term is an atom, number, variable, or structure. A structure has a functor and one or more arguments. The arguments are terms. The table defining the token counting scheme is shown in Table 8. 1.

Using this counting scheme for the *member/2* example:

$$\begin{aligned}
 S_1 &= [t_{\text{period}}, t_0, t_{\text{comma}}, t_{[], t_1}, t_{\text{period}}, t_{:-}, t_0, t_{\text{comma}}, t_{[], t_1}, t_0, t_{\text{comma}}] \\
 \eta set_1 &= \{t_{\text{period}}, t_0, t_{\text{comma}}, t_{[], t_1}, t_{:-}\} \\
 S_2 &= [t_{\text{member}}, t_X, t_X, t_{\text{underscore}}, t_{\text{member}}, t_X, t_{\text{underscore}}, t_T, t_{\text{member}}, t_X, t_T] \\
 \eta set_2 &= \{t_{\text{member}}, t_X, t_{\text{underscore}}, t_T\}
 \end{aligned}$$

Using this token analysis of the *member/2* program, we get the following counts and size measurements:

$$\begin{aligned}
\eta_1 &= 6 & \eta &= 10 \\
N_1 &= 13 & N &= 24 \\
\eta_2 &= 4 & V &= 24 \cdot \log_2(10) = 80 \\
N_2 &= 11
\end{aligned}$$

We apply this counting technique to the *union/3* solution as follows:

Clause 1a:

$$\begin{aligned}
S_1 &= [t_{\text{period}}, t_{:-}, t_{()}, t_{\text{comma}}, \\
&\quad t_{[]}, t_{|}, t_{\text{comma}}, t_{\text{comma}}, \\
&\quad t_{()}, t_{\text{semicolon}}, t_{->}, t_{()}, \\
&\quad t_{\text{comma}}, t_{=}, t_{=}, t_{[]}, t_{|}, t_{()}, \\
&\quad t_{\text{comma}}, t_{\text{comma}}] \\
\eta_{set_1} &= \{t_{\text{period}}, t_{:-}, t_{()}, t_{\text{comma}}, \\
&\quad t_{[]}, t_{|}, t_{\text{semicolon}}, t_{->}, t_{=}\} \\
S_2 &= [t_{\text{union}}, t_H, t_X, t_Y, t_Z, t_{\text{member}}, \\
&\quad t_H, t_Y, t_Z, t_T, t_Z, \\
&\quad t_H, t_T, t_{\text{union}}, t_X, t_Y, t_T] \\
\eta_{set_2} &= \{t_{\text{union}}, t_H, t_X, t_Y, t_Z, t_{\text{member}}, t_T\}
\end{aligned}$$

Clause 2a:

$$\begin{aligned}
S_1 &= [t_{\text{period}}, t_{()}, t_{\text{comma}}, t_{\text{comma}}] \\
\eta_{set_1} &= \{t_{\text{period}}, t_{()}, t_{\text{comma}}\} \\
S_2 &= [t_{\text{union}}, t_{\text{nil}}, t_Y, t_Y] \\
\eta_{set_2} &= \{t_{\text{union}}, t_{\text{nil}}, t_Y\}
\end{aligned}$$

Combined values for these clauses:

$$\begin{aligned}
S_1 &= [t_{\text{period}}, t_{:-}, t_{\text{comma}}, t_{()}, t_{\text{comma}}, t_{[]}, t_{|}, t_{\text{comma}}, t_{\text{comma}}, t_{()}, t_{\text{semicolon}}, \\
&\quad t_{->}, t_{()}, t_{\text{comma}}, t_{=}, t_{=}, t_{[]}, t_{|}, t_{()}, t_{\text{comma}}, t_{\text{comma}}] \\
&\quad + [t_{\text{period}}, t_{()}, t_{\text{comma}}, t_{\text{comma}}] \\
\eta_{set_1} &= \{t_{\text{period}}, t_{:-}, t_{()}, t_{\text{comma}}, t_{[]}, t_{|}, t_{\text{semicolon}}, t_{->}, t_{=}\} \cup \{t_{\text{period}}, t_{()}, t_{\text{comma}}\} \\
&= \{t_{\text{period}}, t_{:-}, t_{()}, t_{\text{comma}}, t_{[]}, t_{|}, t_{\text{semicolon}}, t_{->}, t_{=}\}
\end{aligned}$$

Operator concrete representation:

```

union([H|X], Y, Z) :-
    (member(H, Y)
     -> Z = T
    );
    Z = [H|T]
),
union(X, Y, T). % Clause 1a

```

```

union([], Y, Y). % Clause 2a

```

Base representation:

```

:- (union('.'(H, X), Y, Z),
    '','(';'('->'(member(H, Y),
                    '='(Z, T))),
    '='(Z, '.'(H, T))),
    union(X, Y, Z)). % Clause 1b

```

```

union([], Y, Y). % Clause 2b

```

Figure 8.5: Original and Base representation form of the PROLOG *union/3* program.

$$\begin{aligned}
S_2 &= [t_{\text{union}}, t_H, t_X, t_Y, t_Z, t_{\text{member}}, t_H, t_Y, t_Z, t_T, t_Z, t_H, t_T, t_{\text{union}}, t_X, t_Y, \\
&\quad t_T] \\
&\quad + [t_{\text{union}}, t_{\text{nil}}, t_Y, t_Y] \\
\eta \text{ set}_2 &= \{t_{\text{union}}, t_H, t_X, t_Y, t_Z, t_{\text{member}}, t_T\} \cup \{t_{\text{union}}, t_{\text{nil}}, t_Y\} \\
&= \{t_{\text{union}}, t_{\text{nil}}, t_H, t_X, t_Y, t_Z, t_{\text{member}}, t_T\}
\end{aligned}$$

Using the combined $S_1, \eta \text{ set}_1, S_2$, and $\eta \text{ set}_2$ values determined above, we can calculate the various token counts and size measurements:

$$\begin{aligned}
\eta_1 &= 9 & \eta &= 17 \\
N_1 &= 24 & N &= 45 \\
\eta_2 &= 8 & V &= 45 \cdot \log_2(17) = 184 \\
N_2 &= 21
\end{aligned}$$

We can compare the above measures for the *union/3* program represented using the operator syntax (and the special list syntax) with the measures for the same program written in the “base” syntax, without the operator syntax or the special list syntax. The counts for the base notation version are:

Clause 1b:

$$\begin{aligned}
S_1 &= [t_{\text{period}}, t_(), t_{\text{comma}}, t_(), t_{\text{comma}}, t_{\text{comma}}, t_(), t_{\text{comma}}, t_(), t_{\text{comma}}, t_(), \\
&\quad t_{\text{comma}}, t_(), t_{\text{comma}}, t_(), t_{\text{comma}}, t_(), t_{\text{comma}}, t_(), t_{\text{comma}}, t_{\text{comma}}] \\
\eta \text{ set}_1 &= \{t_{\text{period}}, t_(), t_{\text{comma}}\} \\
S_2 &= [t_{:-}, t_{\text{union}}, t_{.}, t_H, t_X, t_Y, t_Z, t_{,}, t_{,}, t_{,}, t_{->}, t_{\text{member}}, t_H, t_Y, t_{=}, t_Z, t_T, \\
&\quad t_{=}, t_Z, t_{.}, t_H, t_T, \\
&\quad t_{\text{union}}, t_X, t_Y, t_Z] \\
\eta \text{ set}_2 &= \{t_{:-}, t_{\text{union}}, t_{.}, t_H, t_X, t_Y, t_Z, t_{,}, t_{,}, t_{,}, t_{->}, t_{\text{member}}, t_T, t_{=}, \}
\end{aligned}$$

Clause 2b:

$$\begin{aligned}
S_1 &= [t_{\text{period}}, t_(), t_{\text{comma}}, t_{\text{comma}}] \\
\eta \text{ set}_1 &= \{t_{\text{period}}, t_(), t_{\text{comma}}\} \\
S_2 &= [t_{\text{union}}, t_{\text{nil}}, t_Y, t_Y] \\
\eta \text{ set}_2 &= \{t_{\text{union}}, t_{\text{nil}}, t_Y\}
\end{aligned}$$

$$\begin{aligned}
S_1 &= [\mathbf{t_{period}}, t_0, \mathbf{t_{comma}}, t_0, \mathbf{t_{comma}}, \mathbf{t_{comma}}, t_0, \mathbf{t_{comma}}, t_0, \mathbf{t_{comma}}, t_0, \\
&\quad \mathbf{t_{comma}}, t_0, \mathbf{t_{comma}}, t_0, \mathbf{t_{comma}}, t_0, \mathbf{t_{comma}}, \mathbf{t_{comma}}] \quad (21) \\
&+ [\mathbf{t_{period}}, t_0, \mathbf{t_{comma}}, \mathbf{t_{comma}}] \quad (4) \\
\eta_{set_1} &= \{\mathbf{t_{period}}, t_0, \mathbf{t_{comma}}\} \\
&\cup \{\mathbf{t_{period}}, t_0, \mathbf{t_{comma}}\} \\
&= \{\mathbf{t_{period}}, t_0, \mathbf{t_{comma}}\} \\
S_2 &= [t_{:-}, t_{\text{union}}, t_{:,}, t_H, t_X, t_Y, t_Z, t_{:,}, t_{:,}, t_{:-}, t_{\text{member}}, t_H, t_Y, t_{=}, t_Z, t_T, t_{=}, \\
&\quad t_Z, t_{:,}, t_H, t_T, t_{\text{union}}, t_X, t_Y, t_Z] \quad (25) \\
&+ [t_{\text{union}}, \mathbf{t_{nil}}, t_Y, t_Y] \quad (4) \\
\eta_{set_2} &= \{t_{:-}, t_{\text{union}}, t_{:,}, t_H, t_X, t_Y, t_Z, t_{:,}, t_{:,}, t_{:-}, t_{\text{member}}, t_T, t_{=}, \} \\
&\cup \{t_{\text{union}}, \mathbf{t_{nil}}, t_Y\} \\
&= \{t_{:-}, t_{\text{union}}, t_{:,}, t_H, t_X, t_Y, t_Z, t_{:,}, t_{:,}, t_{:-}, t_{\text{member}}, t_T, t_{=}, \mathbf{t_{nil}}\}
\end{aligned}$$
$$\begin{array}{ll} \eta_1 = 3 & \eta = 17 \\ N_1 = 25 & N = 54 \\ \eta_2 = 14 & V = 54 \cdot \log_2(17) = 221 \\ N_2 = 29 & \end{array}$$

In the operator syntax there are fewer parentheses and argument-list commas than in the base syntax. In the transformation from the operator-syntax to the base-syntax, each binary operator term ‘term OP term’ is replaced by ‘OP(term, term)’, one token (the operator OP) in the first case versus three tokens (the operator OP, the parentheses, and the argument list comma) in the second case. Similarly, the unary operator terms go from one token (the OP) to two tokens (the OP and the parentheses). The list syntax also provides a substantial savings: ‘[term1, ..., termN]’ becomes “‘.(term1, ‘.(term2, ... ‘.(termN, [])...)’”’. This changes N tokens in the list syntax for a list of N terms (one for the brackets and N-1 for the commas) into 3N+1 tokens in the base syntax (a ‘.’, a pair of parentheses, and an argument comma for each term, plus the

'[]' (nil) token for the end of the list). Depending on the program, the size difference between the operator/list syntax and the base syntax can be quite large.

Token counting for LISP. We can define the token counting procedure for LISP in a fashion analogous to the approach used for PROLOG. In the syntax of LISP, a program source is a sequence of S-expressions. An S-expression is either an atom or a list. A list is a sequence of S-expressions enclosed by parentheses. There are special kinds of S-expressions, fexprs and lexprs. These are handled specially by the LISP reader. We will base

our operator/operand discrimination on these special S-expressions. A list of elements of LISP syntax and associated counts is shown in Table 8. 2.

A simple LISP program for the *union* programming problem is:

```
(defun union (L1 L2)
  (if (null L1)
      (let ((x (car L1)))
        (if (member x L2)
            (union (cdr L1) L2)
            (cons x (union (cdr L1) L2)))))
      L2))
```

The token sets for this program are:

$$S_1 = [t_0, t_{\text{defun}}, t_0, t_0, t_{\text{if}}, t_0, t_0, t_{\text{let}}, t_0, t_0, t_0, t_0, t_{\text{if}}, t_0, t_0, t_0, t_0, t_0, t_0]$$

$$\eta \text{ set}_1 = \{t_0, t_{\text{defun}}, t_{\text{if}}, t_{\text{let}}\}$$

Element	additions to $S_1, \eta \text{ set}_1$	additions to $S_2, \eta \text{ set}_2$
nil		t_{nil}
symbol		t_{symbol}
(S-expression-sequence)	t_0	
(defun atom S-expr1 S-expr2)	t_0, t_{defun}	
(defmacro atom S-expr1 S-expr2)	t_0, t_{defmacro}	
(let S-expr1 S-expr2)	t_0, t_{let}	
(quote S-expr)	t_0, t_{quote}	
'S-expr	t_{quote}	
(do S-expr-sequence)	t_0, t_{do}	
(while S-expr-sequence)	t_0, t_{while}	
(if S-expr1 S-expr2 S-expr3)	t_0, t_{if}	
(cond S-expr-sequence)	t_0, t_{cond}	
S-expr S-expr-sequence		

Table 8. 2: Token counting scheme for LISP.

$S_2 =$	$[t_{\text{union}}, t_{L1}, t_{L2}, t_{\text{null}},$			
	$t_{L1}, t_x, t_{\text{car}}, t_{L1},$	V	SPARCL	PROLOG
	$t_{\text{member}}, t_x, t_{L2},$	N	39	184
	$t_{\text{union}}, t_{\text{cdr}}, t_{L1}, t_{L2},$	η	17	45
	$t_{\text{cons}}, t_x, t_{\text{union}}, t_{\text{cdr}},$		5	17
	$t_{L1}, t_{L2}, t_{L2}]$			13
$\eta \text{ set}_2 =$	$\{t_{\text{union}}, t_{L1}, t_{L2}, t_{\text{null}},$	Table 8. 3: Size measurements for all solutions to the <i>Union</i> problem.		
	$t_x, t_{\text{car}}, t_{\text{member}}, t_{\text{cdr}}, t_{\text{cons}}\}$			

The token and size measurements for the program are:

$$\begin{aligned}
 \eta_1 &= 4 & \eta &= 13 \\
 N_1 &= 19 & N &= 41 \\
 \eta_2 &= 9 & V &= 41 \cdot \log_2(13) = 152 \\
 N_2 &= 22
 \end{aligned}$$

The summary of the *union* measurements is shown in Table 8. 3. The SPARCL program is smaller than the other two language programs in all measurements (size, vocabulary, and volume).

Programming problem: *Classify Examples*. The *classify_examples* problem has been solved in each of the three languages discussed above. This problem is more complex than the *union* problem discussed above:

An “example” is a set of attribute name/value pairs. An example set is a set of examples all of which have the same set of attribute names. Given an example set and a classifying attribute name, find a classification of the examples according to their values on that attribute name.

This is basically a problem in finding equivalence classes. The three solutions for this problem are presented and discussed in the section “Inspection: Subjective Assessment of Understandability”. The measurements for each of the three implementations of the *classify_examples* problem are presented below.

The SPARCL *Classify Examples/3* program in Figure 8. 1 has the graphic token counts shown in Table 8. 4. These token counts yield the following size measurements:

$\eta_1 = 2 + 0 + 6 + 2 = 10$	nodes	10	8 endpoints of hyperedges, 2 non-ur N-tuple terms not in hyperedges.
$\eta_2 = 3 + 2 = 5$	node types	3	variable, set, interset
$N_1 = 8 + 11 + 4 = 23$	edges	8	4 N-tuple arrows, 4 hyperedges
$N_2 = 10 + 2 = 12$	edge types	2	hyperedge, N-tuple arrow
	textual tokens	2	'unify', 'Classify Examples'
	textual token types	2	"
$\eta = 15$	enclosures	11	4 N-tuples, 2 nonempty parts, 2 non-empty sets, 1 clause, 2 interset
$N = 35$	enclosure types	6	N-tuple, part, set, interset body, interset template, clause
$V = 35 \log_2(15) = 137$	adjoinments	4	2 interset template/body adj., 2 arg adj.
	adjoinment types	2	interset template/body adj., arg adj.

The summary of the counts for both of the *SPARCL Union/3* and the *Classify Examples/3* solutions is shown in Table 8. 5.

Table 8. 4: Graphic token counts, annotated, for the SPARCL *Classify Examples* solution.

The summary of the size measures derived from these counts is shown in Table 8. 6. As one would expect, the *Union/3* program is much smaller than the *Classify Examples/3* program in each of the derived measurements: half the “vocabulary”, less than half the “size”, and less than one third the “volume”.

PROLOG

There are two solutions of this problem in PROLOG, the brief version and the fast version. These are shown in chapter 7 (“Subjective Analysis”). The counts and size measurements for the brief version are:

$$\begin{aligned}\eta_1 &= 5 \\ N_1 &= 18 \\ \eta_2 &= 9 \\ N_2 &= 18 \\ V &= N \log_2(\eta) = 36 \log_2(14) = 137\end{aligned}$$

The counts and size measurements for the fast version are:

$$\begin{aligned}\eta_1 &= 10 \\ N_1 &= 89 \\ \eta_2 &= 21 \\ N_2 &= 79 \\ V &= 168 \cdot \log_2(31) = 832\end{aligned}$$

	<i>Union/3</i>	<i>Classify Examples/3</i>
nodes	7	10
node types	1	3
edges	3	8
edge types	1	2
textual tokens	1	2
textual token types	1	2
enclosures	7	11
enclosure types	2	6
adjoinments	7	4
adjoinment types	1	2

Table 8. 5: Graphical token counts for ‘Union’/3 and ‘Classify Examples’/3.

Clearly, the fast version is much larger than the brief version; more than twice as big in vocabulary, four times as big in token size, and six times as big in volume.

LISP

The counts and volume measurement for the LISP *classify examples* program are:

$$\begin{aligned}\eta_1 &= 3 \\ N_1 &= 60 \\ \eta_2 &= 20 \\ N_2 &= 96\end{aligned}\quad V = 156 \log_2(23) = 706$$

	<i>Union/3</i>	<i>Classify Examples/3</i>
η_1	4	10
N_1	9	23
η_2	2	5
N_2	8	12
η	6	15
N	17	35
V	44	137

Table 8. 6: Summary of ‘Union’/3 and ‘Classify Examples’/3 measurements.

Discussion. The above material develops token counting methods for PROLOG, LISP, and SPARCL. These counting methods were applied to solutions of the *union* and *classify examples* problems in all three languages.

The *Classify Examples* measurements for all of the solutions are shown in Table 8. 7. For this problem, the PROLOG “brief” solution is smaller in size (N) and volume than the other two. The PROLOG “fast” solution is the largest of the four solutions, in all three measures. The brief PROLOG solution and the SPARCL solution are both substantially smaller than the “fast” PROLOG and LISP solutions, in all measures.

Programming problem: ID3. The *ID3* problem has been solved in each of the three languages discussed above.

The counts for the SPARCL *ID3* solution are shown in Table 8. 8. The summary of the size measures for the different “views” of the SPARCL *ID3* solution are shown in Table 8. 9. The three columns correspond to three different counting regimes. The “basic” regime is to count everything present in the SPARCL solution. The “without *DELAY*” regime counts everything in the solution except the ‘*DELAY*’ clauses. The SPARCL language could be modified so as to represent this control information in a more compact fashion, and this column gives a lower bound on how much the size of the program could be reduced by such a change to the language. Conversely, comparing this column and the “basic” column tells us how much of the “basic” size is

devoted to ‘*DELAY*’ control information. The “without *DELAY* and Cardinality” column gives the counts for the program without the ‘*DELAY*’ or ‘Cardinality’ clauses. The Cardinality procedure is a strong candidate for being built in to the SPARCL language, much as length/2 is commonly built in to PROLOG implementations and the length function is commonly built in to LISP implementations.

This table contains three kinds of counts which we have not previously been reporting:

Literals, Clauses, and Procedures. These counts apply to the two logic programming languages, but are do not compare across programming paradigms. For instance, they are not generally applicable to LISP. The ideas of clause and literal are native to the logic programming paradigm and thus do not apply at all outside of that paradigm. The meaning of procedure varies tremendously across paradigms - in logic programming it is a collection of clauses which have heads of a particular functor and arity, and in LISP it is a function (defined by a “defun”, but should a “defmacro” be considered a proce-

	SPARCL	PROLOG.brief	PROLOG.fast	LISP
V	137	137	832	706
N	35	36	168	156
η	15	14	31	23

Table 8. 7: Size measurements for all solutions of the *Classify Examples* problem.

	basic	without *DELAY*	without both *DELAY* and Cardinality
nodes	163	163	158
node types	5	5	5
edges	198	133	128
edge types	2	2	2
textual tokens	176	56	51
textual token types	27	23	22
enclosures	147	115	105
enclosure types	8	7	7
adjoinments	103	66	63
adjoinment types	4	2	2

Table 8. 8: Graphical token counts for the SPARCL solution of the *ID3* problem.

Measurement name	Basic	Without *DELAY*	Without both *DELAY* and Cardinality
η_1	14	11	11
N_1	448	314	296
η_2	32	28	27
N_2	339	219	209
η	46	39	38
N	787	533	515
V	4347	2817	2650
Literals	19	19	18
Clauses	36	13	12
Procedures	13	12	11

Table 8. 9: Size measurements for three “views” of the SPARCL *ID3* solution.

η_1	15	dure?). In some object-oriented languages, pro-	η_1	7
N_1	299	cedures presumably relate in some fashion to	N_1	304
η_2	78	methods, but this relationship is tenuous.	η_2	68
N_2	289	The PROLOG <i>ID3</i> solution uses the brief	N_2	394
η	93	solution of the <i>Classify Examples</i> problem.	η	75
N	588	The size measures are shown in Table 8. 10.	N	698
V	3845	The LISP <i>ID3</i> solution has the size measures	V	4348
Literals	62	shown in Table 8. 11. The results for the solu-	Functions	20
Clauses	17	tions of the <i>ID3</i> problem in the three languages		
Procedures	12	are summarized in Table 8. 12. The SPARCL		

Table 8. 10: Size measurements for the PROLOG *ID3* solution.

Table 8. 11: Size measurements for the LISP *ID3* solution.

LISP solution on nearly all size measurements. The interesting exception is the number of literals. The complexity of the SPARCL solution is in the number of clauses and the complexity of the terms in its arguments more so than in the PROLOG solution. We consider the sum of the numbers of literals and clauses in PROLOG to be the “logical source line count” for PROLOG. Comparing these sums makes the SPARCL solution smaller than the PROLOG solution. When we consider the “without *DELAY*” view of the SPARCL solution, it is substantially smaller than the other two solutions. As discussed in chapter 3 (“Design Elements”) and chapter 7 (“Subjective Analysis”), the delay specifications should be more compactly representable. We estimate that we can reduce the size contribution of the delay specifications by about one-half to two-thirds for this example and the *WARPLAN* example. The benefit of such a more compact representation are demonstrated by the “without *DELAY*” view of the size of the SPARCL solution of the *ID3* problem. The further value of implementing ‘Cardinality’/2 as a built-in predicate is demonstrated by the size reduction shown in the “without *DELAY* & Cardinality” as compared to the other views of the SPARCL solution.

Programming problem: *WARPLAN*. The *WARPLAN* problem is the final programming problem we study in this chapter. The counts and measures for the SPARCL and PROLOG implementations of the solutions to the *WARPLAN* problem (without a “world” specification) are shown in Table 8. 13.

The comparison between the SPARCL and PROLOG solutions is surprising. We had expected the SPARCL implementation to be noticeably larger than the PROLOG implementation, but it is instead much the same depending on which measurement one uses. The volume, which depends

	SPARCL basic	SPARCL without *DELAY*	SPARCL without *DELAY* & Cardinality	PROLOG	LISP
η_1	14	11	11	15	7
N_1	448	314	296	299	304
η_2	32	28	27	78	68
N_2	339	219	209	289	394
η	46	39	38	93	75
N	787	533	515	588	698
V	4347	2817	2650	3845	4348
Literals	19	19	18	62	
Clauses	36	13	12	17	
Procedures/Functions	13	12	11	12	20

Table 8. 12: Size measurements for all solutions of the ID3 problem.

on both vocabulary (η) and size (N), is slightly smaller for SPARCL (the SPARCL volume is 97% of the PROLOG volume). If one considers the delay-less version of the counts, the comparison is strictly favorable for SPARCL.

We found this surprising. The WARPLAN algorithm was designed with PROLOG in mind and is a heuristic search algorithm which requires various steps to be done in a particular order. Ordering literal evaluation in SPARCL requires explicit direction from the programmer, as opposed to the implicit ordering which is present in PROLOG. Thus we expect SPARCL to not compare favorably with an implicit order language such as PROLOG in solving a problem where ordered evaluation is an important part of the solution. The *DELAY* clauses are the major technique in SPARCL for ensuring a particular ordering of evaluation (if/3 and ordered_disjunction/2 being the other techniques). Comparing the two SPARCL columns of the table, we see that the *DELAY* clauses make up 57% of all clauses. The *DELAY* clauses account for 33% of the size and 35% of the volume.

Discussion

This chapter presents methods for analyzing a visual programming language which assesses both the benefit derived from the visual representation and the conciseness of the language compared with other programming languages, both visual

and textual.

Determining what to analyze. We presented an argument for the use of program size of solutions to selected programming problems in objectively comparing different programming languages. The desire to compare programming languages that differ in both representation technique and underlying semantics limits the kinds of comparisons that are meaningful. We used Sethi's definition of a good programming language as our guide in comparing the quality of programming languages. Working through a measurement theory-based

	SPARCL basic	SPARCL without *DELAY*	PROLOG
η_1	11	8	10
N_1	552	373	410
η_2	36	33	84
N_2	460	306	402
η	47	41	94
N	1012	679	812
V	5621	3638	5322
Literals	38	38	118
Clauses	61	26	39
Procedures	21	20	19

Table 8. 13: Size measurements for solutions for the WARPLAN program.

software measurement framework, we determined that: the external attribute we need to measure is quality (as defined by Sethi); the object is a programming language; it is a resource-type object with which the attribute is associated; the measurement must be indirect; and this indirect measurement is based on measurements of three external process attributes, readability, understandability, and modifiability.

We adopt the analytic approach to programming language assessment in this chapter, we investigate an experimental approach to assessment in the next chapter. We determined that an appropriate analytic attribute is program size. An attribute which can be measured for a program (as opposed to the process of that program's execution) in any programming language must be an attribute of the *intended-for-human-manipulation concrete representation* of that program. One of the most basic attributes of the concrete representation of a program is its *size*. There is an approximately inverse monotonic relationship between the size of the concrete representation of a program and the three process attributes of interest: the larger the concrete representation of a program, the less readable, the less understandable, and the less modifiable.

In addition to the measurement of relative sizes discussed above, we measure the diagrammatic/linguistic appropriateness; a direct assessment of how thoroughly SPARCL exploits the possibilities of its visual representation. We first investigated the diagrammatic assessment of SPARCL, then we investigated the sizes of the solutions of

the selected problems.

Diagrammatic versus linguistic analysis. Shin provides three aspects to assess for diagrammatic possibilities: relations of objects, conjunctions, and tautologies and contradictions. Five diagrammatic relations of objects were discussed:

- 1) the membership relationship is represented by the convention that any “term” spatially *in* the representation of a set is a member of that set, connecting membership to a spatial relation;
- 2) the union of the parts of a partitioning cover the partitioned set spatially as well as semantically;
- 3) the empty set representation is partially diagrammatic in that it is a rectangle, such as any (potentially) nonempty set, but it is filled in so that no term can be placed inside of it—spatially representing that it has/can have nothing in it;
- 4) syntactically distinct terms can “refer” to semantically the same term by coreference links—lines drawn connecting the coreferencing terms, this relies on the perceptual inference of seeing the connections; and,
- 5) the ordering relationship of elements of an N-tuple and arguments of a literal or a clause is indicated by their being placed adjacent, horizontally with intervening arrows for N-tuples and vertically for arguments.

Two conventional/linguistic relation representations are the pairwise disjointness of parts in a partitioned set and the connection of a literal to the clauses defining that literal’s predicate is by a convention using the names of the literals and clauses.

SPARCL represents the conjunction of clauses and literals diagrammatically. There are some aids to seeing contradictions in a SPARCL program. If a set is not too large (no more than ten elements, say), then the spatial locality of representation of its elements helps the user to see when the same element is in more than one part (which contradicts the disjointness constraint). The connecting lines for coreference help diagrammatically to see coreferring terms contradict the requirement that all coreferring terms unify (e.g. two different ur constants such as ‘this’ and ‘that’).

Size analysis. We defined size measurements for programs in three languages: SPARCL, PROLOG, and LISP. These size measurements are all based on tokens in the concrete representation. The SPARCL measurement is based on graphical tokens, the other two are based on lexical tokens.

We justify the use of our token-based size measurements based on: they are closely related to Halstead's size measurements (size, vocabulary, and volume); Fenton determined that, from a measurement theory point of view, Halstead's basic size measurements are "reasonable measures of three internal program attributes which reflect different views of *size*"¹³; program size has been shown to inversely correlate with readability, understandability, and modifiability, as discussed in [Kitchenham et al. 1990] and [Conte et al. 1986], particularly for large programs; and, the three external attributes of readability, understandability, and modifiability provide an indication of program quality.

We developed our graphical token counting method from that of Nickerson. This involved developing Halstead operator and operand equations from Nickerson's graphic token count and diagram class complexity equations, and mapping his graphical element concepts onto the concrete (two-dimensional) representation of SPARCL. These graphical element concepts are: node, node type, edge, edge type, textual token, textual token type, enclosure, enclosure present, adjoinment, and adjoinment present.

Textual token counting schemes are defined based on an approach used by [Bieman et al. 1991]. These definitions are presented for PROLOG in Table 8. 1 and LISP in Table 8. 2. The PROLOG syntax has two forms, with and without operators. We showed that the token counting approach is sensitive to differences in concrete representation (as required by our uses of the software measurements based on the token counts) by counting the tokens for an example program written in both of these forms. As one would expect, the operator form of PROLOG is shorter than the form without operators.

The above material develops token counting methods for PROLOG, LISP, and SPARCL. These counting methods were applied to solutions of the *Union*, *Classify Examples*, *ID3*, and *WARPLAN* problems in all three languages. The tables summarizing the measurements for these problems are in Table 8. 3, Table 8. 7, and .

The *Classify Examples* measurements for all of the solutions are shown in Table 8. 7. For this problem, the PROLOG "brief" solution is smaller in size (N) and volume than the other two. The PROLOG "fast" solution is the largest of the four solutions, in all three measures. The brief PROLOG solution and the SPARCL solution are both substantially smaller than the "fast" PROLOG and LISP solutions, in all measures.

13. p. 19 of [Fenton 1991a].

The *ID3* solution measurements in Table 8. 12 shows PROLOG solution the smallest and the SPARCL and LISP solutions a little larger. With a more compact representation of delay specifications, the SPARCL solution could be the smallest of the three in all size measurements.

The *WARPLAN* solution measurements in Table 8. 13 shows the SPARCL solution a little larger than the PROLOG solution (no LISP solution is given). The SPARCL solution is smaller in vocabulary (47 versus 94) and “logical source lines” (99 versus 157). As for the *ID3* solutions, if the delay specifications were more compactly represented in SPARCL then its solution could be much smaller than the PROLOG solution in all respects. This surprised us in that WARPLAN requires a great deal of explicit interpretation ordering. This ordering of the interpretation requires explicit representation in SPARCL but is implicitly represented in PROLOG (using literal order within a clause and clause order within a program file). Even with this explicit/implicit imbalance, the SPARCL solution is about the same size, and would be smaller with an “improved” delay specification representation.

Assessment. SPARCL was shown to exploit many of the diagrammatic possibilities of its visual representation.

Our measurements of the size of SPARCL programs surprised us with how “big” they are; our intuition on looking at these programs was that they were substantially smaller than the corresponding PROLOG and LISP solutions. Our conclusion is that that as SPARCL is represented now, it produces solutions that are about the same size as those in PROLOG and LISP. The second surprise this size analysis afforded us was that the sizes of the *ID3* and *WARPLAN* solutions in SPARCL are very sensitive to the design of the representation of delay specifications. This clearly motivates future work on a more compact representation of delays.

Assessing the size analysis in more detail, SPARCL is somewhat more concise than LISP, but the comparison with PROLOG is complicated. For the two small problems, SPARCL is substantially smaller in one case (union) and the same size in the other case (classify examples). For the two larger problems, the volumes of the SPARCL solution and the PROLOG solution are close; the SPARCL solution volume is $\pm 5\%$ of the PROLOG solution volume. Considering the delay-less measurements of SPARCL, the volume comparison is strongly in SPARCL’s favor: the SPARCL *ID3* delay-less solution is 64% of the volume of the PROLOG solution, the SPARCL *WARPLAN* delay-less solution is

63% of the volume of the PROLOG solution. While the SPARCL solutions must have delay specifications in some form, and thus could not achieve these smaller sizes for valid solutions, significantly more compact representations are possible that we speculate would achieve one-half to two-thirds of these size savings. Additionally, these savings might be realized by having SPARCL determine appropriate delay specifications automatically using a process similar to mode analysis or type inferencing. This would remove much or all of the burden of specifying delays from the programmer, with the corresponding reduction in the size of the SPARCL programs.

- Apte&Kimura 1993 “A comparison study of the pen and the mouse in editing graphic diagrams.” by Ajay Apte and Takayuki Dan Kimura. Pages 352-357 in *Proceedings 1993 IEEE Symposium on Visual Languages*, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- Bieman et al. 1991 “Moving from Philosophy to Practice in Software Measurement” by James Bieman, Norman Fenton, David Gustafson, Austin Melton, and Robin Whitty. Pages 38-59 in *Formal Aspects of Measurement: Proceedings of the BCS-FACS Workshop on Formal Aspects of Measurement, South Bank University, London, 5 May 1991* edited by Tim Denvir, Ros Herman, and R. W. Whitty. Springer-Verlag: London. 1991.
- Bollman&Cherniavsky 1981 “Measurement-Theoretical Investigation of the MZ-Metric.” by P. Bollmann and V.S. Cherniavsky. In *Information Retrieval Research*, R.N. Oddy, S.E. Robertson, C.J. van Rijsbergen, P.W. Williams (ed.), Butterworth, 1981.
- Briand et al. 1996 “On the Application of Measurement Theory in Software Engineering” by Lionel Briand, Khaled El Emam, and Sandro Morasca. International Software Engineering Research Network technical report #ISERN-95-04. To appear in *Empirical Software Engineering: An International Journal*, 1(1), Kluwer Academic Publishers, 1996.
- Conte et al. 1986 *Software Engineering Metrics and Models* by S. D. Conte, H. E. Dunsmore, and V. Y. Shen. The Benjamin/Cumming Publishing Company, Inc.: Menlo Park, California. 1986.
- Fenton 1991a “Software Measurement: Why a Formal Approach?” by Norman Fenton. Pages 3-27 in *Formal Aspects of Measurement: Proceedings of the BCS-FACS Workshop on Formal Aspects of Measurement, South Bank University, London, 5 May 1991* edited by Tim Denvir, Ros Herman, and R. W. Whitty. Springer-Verlag: London. 1991.
- Fenton 1991b *Software Metrics: A Rigorous Approach* by Norman E. Fenton. Chapman & Hall. 1991.
- Finkelstein 1984 “A review of the fundamental concepts of measurement” by L. Finkelstein in *Measurement*, Vol. 2(1) 1984, 25-34.
- Glinert 1990 “Nontextual Programming Environments” by Ephraim P. Glinert. Pages 144-230 in *Principles of Visual Programming Systems* edited by Shi-Kuo Chang. Englewood Cliffs, New Jersey: Prentice Hall. 1990.
- Kitchenham et al. 1990 “An evaluation of some design metrics” by B. Kitchenham, L. Pickard, and S. J. Linkman in *Software Engineering Journal* Vol 5(1), 1990, 50-58.
- Kitchenham et al. 1990 “An evaluation of some design metrics” by B. Kitchenham, L. Pickard, and S. J. Linkman in *Software Engineering Journal* Vol 5(1), 1990, 50-58.
- Knight 1989 “Unification: A Multidisciplinary Survey” by Kevin Knight. Pages 93-124 in *ACM Computing Surveys*, Vol. 21, No. 1, March 1989.
- Krantz et al. 1971 *Foundations of Measurement, Vol. 1* by D. H. Krantz, R. D. Luce, P. Suppes, and A. Tversky. Academic Press. 1971.

- Lambert 1990 *Neues Organon I* by Johann Heinrich Lamber. Berlin: Akademie Verlag, 1990.
- Nickerson 1994a *Visual Programming* by Jeffrey V. Nickerson. PhD. Dissertation, Dept. of Computer Science, New York University, 1994.
- Nickerson 1994b “Visual Programming: Limits of Graphic Representation” by Jeffrey V. Nickerson. Pages 178-179 in *Proceedings 1994 IEEE Symposium on Visual Languages*, October 4-7, 1994. St. Louis, Missouri. IEEE Computer Society Press: Los Alamitos, CA.
- Pandey&Burnett 1993 “Is it easier to write matrix manipulation programs visually or textually?” by Rajeev K. Pandey and Margaret M. Burnett. Pages 344-351 in *Proceedings 1993 IEEE Symposium on Visual Languages*, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- Sethi 1989 *Programming Languages: Concepts and Constructs* by Ravi Sethi. 478 pages. Reading, Massachusetts: Addison-Wesley Publishing Co. 1989.
- Shin 1994 *The Logical Status of Diagrams* by Sun-Joo Shin. 197 pages. Cambridge, England:Cambridge University Press. 1994.
- Zuse 1996 “History of Software Measurement” by Horst Zuse. This is a world-wide-web page at <http://www.cs.tu-berlin.de/~zuse/3-hist.html> (as of May 29, 1996).
- Zuse&Bollman 1987 “Using Measurement Theory to Describe the Properties and Scales of Static Software Complexity Metrics.” by Horst Zuse and Peter Bollmann. Originally published as RC 13504, IBM Thomas Watson Research Center Yorktown Heights, 1987. Republished in *SIGPLAN Notices*, Volume 24, No. 8, pp.23-33, August 1989.

Chapter 9

Usability Testing

This chapter presents the usability testing of SPARCL. Due to resource constraints, this testing was very limited and thus the analysis of it and the conclusions which we may draw from it are correspondingly modest. The testing consisted of having seven graduate students work through an online, integrated tutorial of SPARCL (a computer-based training¹ system that is part of SPARCL). Their interactions were recorded by SPARCL, and they were encouraged to record comments freely during the course of the tutorial. The data from the testing consists of the interaction log files, their recorded comments, and their solutions to the exercises in the tutorial.

Usability testing demands a usable implementation. If the implementation has a poorly implemented interface, or has prominent bugs, then the testers have a hard time testing the underlying concepts. For this reason, we had to wait to do any testing until our implementation of SPARCL was fairly mature. This severely reduces the amount of time in the research project available for the testing and analysis of the test results, much less for responding to such an analysis with changes in SPARCL. Many programming language research projects do user testing after the project has used many researcher-years of effort. In contrast, the SPARCL project has a user testing component, however primitive, as a part of its initial development.

Usability testing can provide information about the processes of learning the SPARCL language and building SPARCL programs, as well as information about the testing process itself. The integrated tutorial, the interaction monitoring, and the system commenting facility are the “instrumentation” for gathering data about these processes. As with any complex scientific instrument, much effort must be devoted to developing the instrument so that the data it provides is useful. We are still in the process of refining our instrumentation, as well as the design of the “experiments” which are to use this instrument. The information in this chapter is a presentation of this work in progress. The conclusions we can draw about learning and using SPARCL from the experiment we have run so far are necessarily provisional and limited in scope.

1. Computer-based training (CBT) or computer-based instruction (CBI) is briefly mentioned in [Compeau et al. 1995]. This has a long history in computer-aided instruction (CAI), notably the PLATO system.

Related Work

An integrated tutorial system for SMALLTALK, MiTTS, is presented in [Carroll&Rosson 1995]. MiTTS interactively introduces the concepts of SMALLTALK, with many exercises. It is in many ways more sophisticated than the tutorial system we provided with SPARCL. The MiTTS tutorial took from 4 to 6 hours to complete. A “commercial” SMALLTALK tutorial was compared to MiTTS. It took about 12 hours to complete. The users of MiTTS did much better on a follow-up test than did the other users.

The monitoring of command data is discussed in [Kay&Thomas 1995]. This is similar to the monitoring of interactions which we used in SPARCL. Kay and Thomas studied the patterns of usage of SAM, a modeless editor with basic commands invoked using the mouse. Their data collection was incomplete in several ways, but they found that interesting conclusions could be drawn from it nonetheless. They determined that the monitoring information was sufficient to provide a fairly accurate profile of the user’s understanding of various aspects of SAM.

The Tutorial

The tutorial is a basic introduction to SPARCL. An adaptation of the entire scripted tutorial is given in appendix 1 (“Tutorial Introduction”). The scripted tutorial assumes no particular knowledge of SPARCL or logic programming, but it does assume a familiarity with the Apple Macintosh user interface. Much of the tutorial is focused on logic programming, as it looks in SPARCL. About half of the tutorial presents concepts which are unique to SPARCL. The tutorial only explains a small portion of the SPARCL language and environment (for instance, nothing is said about projects, program overviews, preferences, or term sets versus evaluable programs). The first portion of the tutorial, which is about logic programming as it appears in SPARCL, is based on the first chapter of [Bratko 1990].

The opening screens of SPARCL are shown in Figure 9. 1 and Figure 9. 2. Figure 9. 1 shows the “splash” screen which the user sees when the SPARCL application is first opened. Figure 9. 2 shows the initial appearance of the tutorial. This is what the user



Figure 9. 1: SPARCL splash screen.

sees directly after the splash screen if she selects the “Tutorial” button on the splash screen.

The tutorial is organized into several parts, a preface and three chapters. Each of the chapters is made up of several sections. The preface provides a basic introduction to the tutorial system. In the course of explaining SPARCL, the tutorial demonstrates the construction and use of programs. The menus which the user would “pop up” are displayed as though the user had invoked them, and there is an “instructional cursor” which shows (roughly) where the user would place the actual cursor were the user doing the action being demonstrated. An example of demonstrating a popup menu is shown in Figure 9. 3. The state of the system after the action of Figure 9. 3 is shown in Figure 9. 4.

Tutorial preface. Information is presented to the user in several places in the course of this tutorial. It may be disorienting at first, but after a little while the user should find it quite usable. There are two scripting windows and there may be one or more additional "regular" SPARCL windows. The simplest way to use the non-exercise portion of the tutorial is to read the "comment" window, the "Next Step" and "How" areas of the "control" window, watch the "regular" SPARCL windows as they are

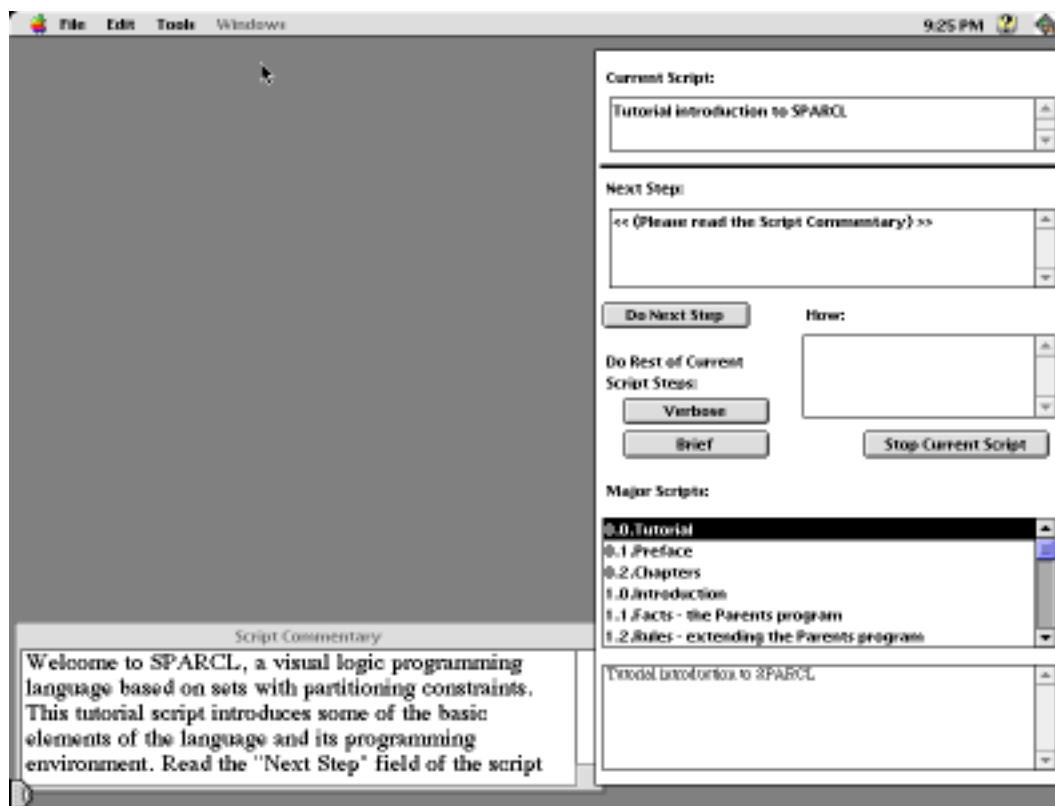


Figure 9. 2: Start of the tutorial.

used in the course of the tutorial, and to click only on the "Do Next Step" button. A more complete explanation of the tutorial script mechanism follows.

The two windows that are just for the script mechanism are the "comment" window in which the user is reading this comment and the "control" window which provides the user control over the execution of the script. Each comment in the comment window is displayed with an "@" character at the end. This lets the user know when she needs to scroll the comment window to get to the end of the text.

There are four buttons in the "control" window: "Do Next Step", "Verbose" and "Brief" under the "Do Rest of Current Script Steps:" header, and "Stop Current Script". The "Do Rest of Current Script:" buttons ("Verbose" and "Brief") tell the system to execute the rest of the current script without stopping, "Verbose" means to display any information that the user would have seen had she "stepped" through the script and "Brief" means to suppress the script-associated information. The user may interrupt the tutorial after she has told it to "Do Rest of Current Script Steps" by clicking in a small window which appears over the "Do Next Step", "Verbose", and

"Brief" buttons. This will put her back into "step" mode at whatever step the tutorial was executing when she clicked in the "*Interrupt*" window.

In the control window, there are five informational areas. The script which is currently being executed is identified as the "Current Script". The effect of the script action which will be executed next (should the user click on the "Do Next Step" button) is described in the "Next Step" area. The way in which the user would do the step herself, were she to do it instead of the script doing it, is described in the "How" section. The fourth informational area is a list of all of the "major" scripts in the tutorial. The current "major" script is highlighted (right now, the highlighted script should be "0.1.Preface"). The current major script may be different than the current script. This happens when the current script is an "internal" or "helper" script which has been invoked in the course of executing the current major script. The fifth informational area is a log of all of the steps executed since starting the tutorial.

There are several tutorial scripts. Each script contains a sequence of comments (such as this) and actions. The kinds of actions which the script makes are the same

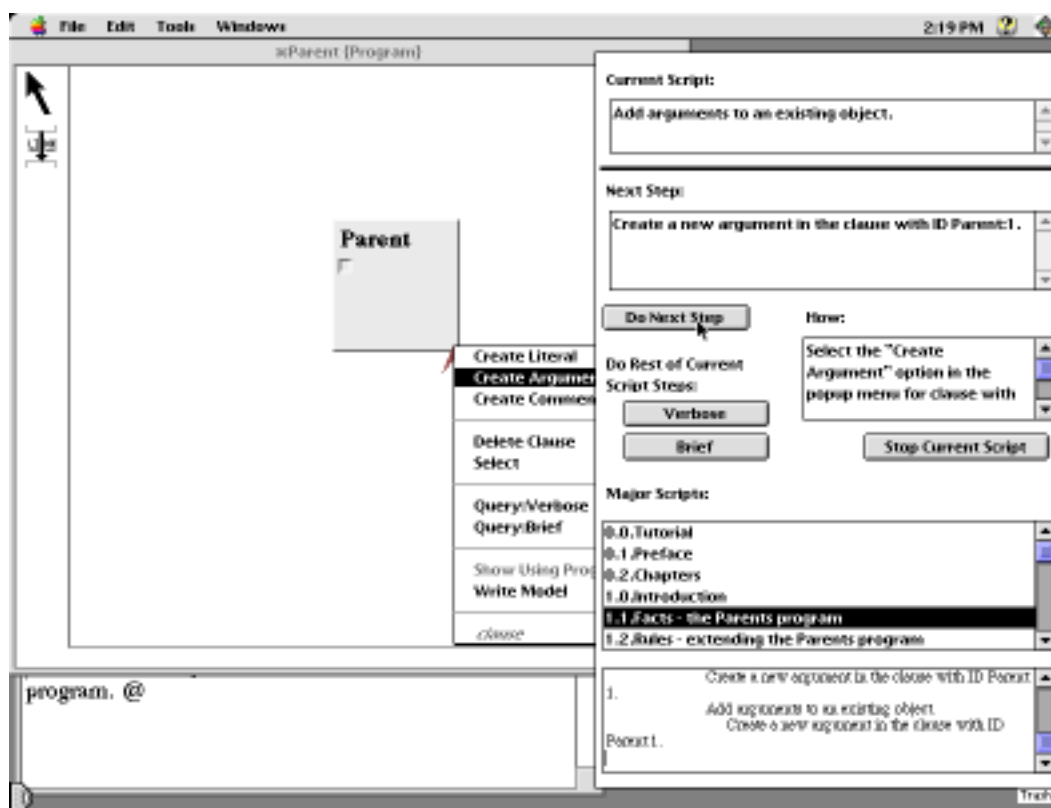


Figure 9. 3: Demonstrating a popup menu.

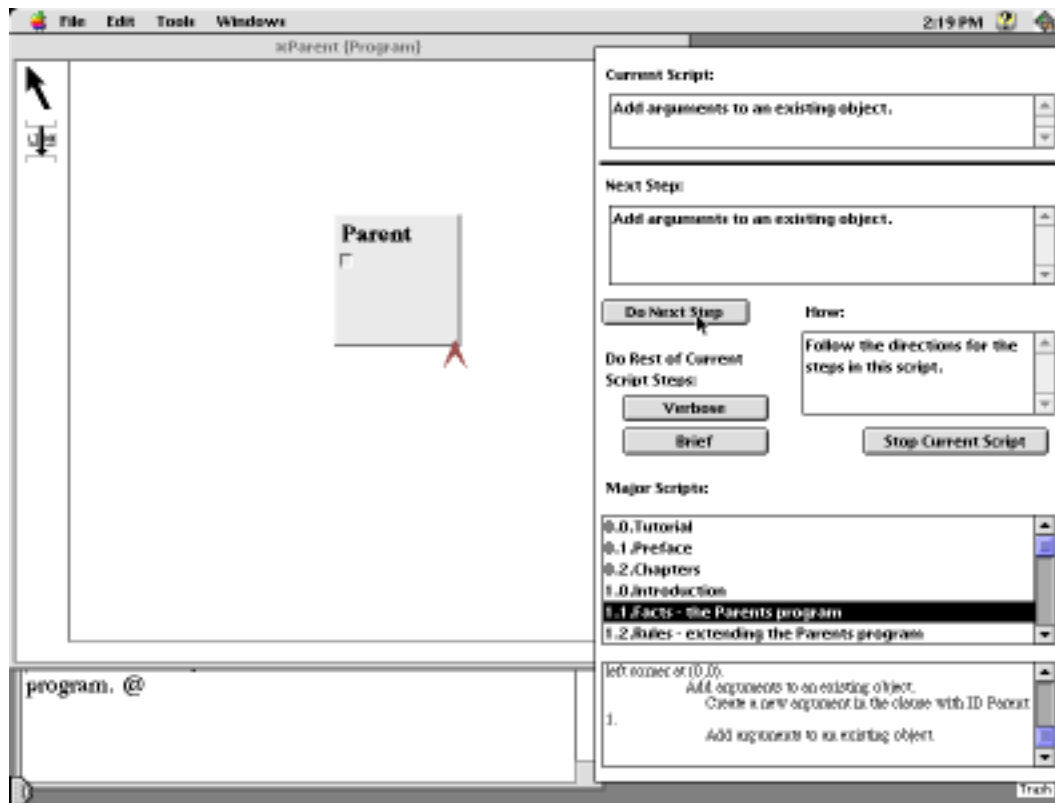


Figure 9. 4: Demonstration of cursor location (the inverted “V” shape at the lower right of the ‘Parent/1’ clause.

kinds of (inter)actions which the user may make with the SPARCL programming environment. Generally, scripts create and execute SPARCL programs, with commentary interspersed to explain the purposes of the actions.

This tutorial "disables" most of the SPARCL operations (to keep the user from accidentally changing the SPARCL environment in such a way that the tutorial will not work). For the exercises, the tutorial will "enable" all of SPARCL.

The scripts are organized into "chapters" and "sections". This script is the "0.1.Preface" section (which comes before the chapters). It's "parent" script is "0.0.Tutorial". This parent script invokes two scripts. This one, the preface, and the "0.2.Chapters" script. The Chapters script invokes three chapters: 1, 2, and 3. Each of the chapters is divided into several sections.

The chapter scripts all have names which have the form "N.0.Chapter" where N is the chapter number and Chapter is the name of the chapter. Section scripts have names of the form "N.K.Section" where K is greater than 0. Chapter and section

scripts may be run "independently", the user does not have to run them in order. She may wish to exit the tutorial before completing all of the tutorial and then return at a later time. The chapters and sections are all "major" scripts and are listed on the "Tutorial Scripts" submenu of the "File " menu.

At various points in this tutorial the user will see references to particular "objects" on the screen in the "How" area of the control window. These are written as "ProgramName:Number" where ProgramName is the name of the program containing the object and Number is an arbitrary identifier.

Tutorial contents summary. The titles of the three chapters and their sections, and the full statements of the exercises, are as follows:

Chapter 1: Introduction

Section 1: Facts - the Parents program

Exercise 1:

In the provided "Exercise 1.1" program window, formulate in SPARCL the following questions about the "Parent" relation (your question formulation should be "query" clauses with names of your choosing (e.g. "Query A", "Query B", "Query C")):

- A) Who is Pat's parent?
- B) Does Liz have a child?
- C) Who is Pat's grandparent?

Section 2: Rules - extending the Parents program

Exercise 1:

In the provided "Exercise 1.2" program window, show translations for the following statements:

- A) Everybody who has a child is happy (introduce a one-argument relation "Happy").
- B) For all X, if X has a child who has a sister then X has two children (introduce a new relation "Has Two Children").

Exercise 2:

Define the relation "Grandchild" using the "Parent" relation. Hint: It will be similar to the "Grandparent" relation.

Exercise 3:

Define the relation "Aunt" of two arguments in terms of the relations "Parent" and "Sister".),

Section 3: How it works

Section 4: Declarative and procedural meaning of programs

Chapter 2: Representation and Meaning of SPARCL Programs

Section 1: Data objects

Exercise 1:

In the provided "Exercise 2.1" program window, develop a representation for rectangles, squares, and circles as structured SPARCL objects. Use an approach similar to that given in the "Geometry Example" program. Write single argument clauses which have example terms of each of these in their argument and use a comment in the argument to explain the elements of the terms (e.g. "a

triangle is represented by a set of three (X,Y) points which are the corners of the triangle"):

- A) write a Rectangle clause of one argument which contains a term for a rectangle with diagonally opposite corners at (1,2) and (34,-5.8)
- B) write a Circle clause of one argument which contains a term for a circle centered at (93,4) with a radius of 16
- C) write a Square clause of one argument which contains a term for a square with upper left corner at (14, 23) and sides of length 123.

Section 2: Matching

Section 3: Declarative meaning of SPARCL programs

Section 4: Procedural meaning of SPARCL programs

Chapter 3: Application of SPARCL

Section 1: "Column Sum" Example

Exercise 1:

This is a complex exercise, you may wish to work on it in multiple attempts. You are asked to implement a "Tournament Scores" predicate which scores players based on their performance in multiple rounds of a tournament. In preparation for implementing this predicate (in part 3, below), you are first asked to implement "Maximal Range Value" in part 1, and then you are asked to implement "Maximal Pairs" in part 2. These predicates will make use of ordered pairs (2-tuples), partitioned sets, intensional sets (and intensional multisets), function tables, "*DELAY*" specifications, and the fails/2 metapredicate (a metapredicate is a predicate which takes a literal as an argument and invokes the given literal).

Write and test a "Maximal Range Value" predicate of two arguments.

- A) Write a program for the predicate "Maximal Range Value" with two arguments. The first argument is a set of pairs, with the second elements (the range values) being numbers. The second argument is a number which is maximal with respect to the range values of this set, i.e. a number such that no range value is greater than it.

HINT:

Use a fails/1 literal and use two literals in its argument, a unify/2 literal and a less/2 literal. The unify/2 literal unifies given the pair set with a partitioned set with an ordered pair of two variables in one part and nothing in the other part. The less/2 literal compares the second argument term with the second element of the ordered pair of the unify/2 set.

- B) Write a "Maximal Range Value Query" predicate of no arguments which tests the "Maximal Range Value/2" predicate by giving it the set "{a=>3,b=>4,c=>5}" and 5 as the maximal range value.

Exercise 2:

Write a "Maximal Pair" predicate of three arguments and a test predicate.

- A) The "Maximal Pairs" predicate has a set of ordered pairs as its first argument, its second argument is the set of maximally range-valued pairs among the first argument's set. Thus, for the first argument set "{a=>2, b=>1, c=>2, d=>0}", the second argument set of maximal pairs is "{a=>2, c=>2}".

HINT:

This is implemented by a single clause, plus a "helper" predicate. The second argument to "Maximal Pairs" should be an intensional set. The

template is an ordered pair (which will become the maximal ordered pairs). The body has two literals. One of these is a unify/2 literal which unifies a 2-part partitioning with a variable. The first part contains a 2-tuple, the second part is hollow. The second literal is the "helper" predicate, "Maximal Range Value" of two arguments as implemented in part 1 of this exercise. For "Maximal Range Value"/2 to work correctly in "Maximal Pairs", there must be two `"*DELAY*"` specifications for the "Maximal Range Value" predicate of two arguments such that "Maximal Range Value" delays if either argument is a variable (i.e. a delay specification for `"variable=>ignore"` and another for `"ignore=>variable"`).

- B) Write a "Maximal Pairs Query" predicate of one argument to test the "Maximal Pairs" predicate. The "Maximal Pairs Query" predicate should return the maximal pairs found by "Maximal Pairs", given the test set `"{a=>2, b=>1, c=>2, d=>0}"`.

Exercise 3:

Write a "Tournament Scores" predicate and a predicate to test it.

- A) The "Tournament Scores" predicate has two arguments. The first argument is the tournament rounds scores and the second argument is a function from player to overall score for that player. The tournament rounds scores are a function table, where each column is a different player in the tournament and each row is a set of scores for a round of the tournament. The overall scoring of a player for the tournament is the number rounds in which that player was among those with the highest score.

HINT:

The second argument of "Tournament Scores" is an intensional multiset, where the template is a variable (which will be the player) and the body contains two literals. One of these literals is a unify/2 literal which extracts a row from the rounds table. The other literal is a "Maximal Pairs"/2 literal which finds the players with maximal scores for that round. The second argument to this "Maximal Pairs" literal should be a partitioned set of two parts. One of these parts is an ordered pair with the first element of this pair being connected to the template variable.

- B) Write a "Tournament Scores Query" predicate of one argument which returns the overall player scores from "Tournament Scores", given a table of:

`"{{a=>1, b=>2, c=>0}, {a=>2, b=>2, c=>1}, {a=>1, b=>0, c=>2}}"`.

The “experiment”.

The user testing experiment involved the seven graduate students enrolled in a course on visual programming languages in the Spring of 1996. Each participant was given a single sheet of instructions, shown in Figure 9. 5. They independently ran the tutorial on one of two Apple Power Macintoshes in the DesignLab facility of the Electrical Engineering and Computer Science Department of the University of Kan-

SPARCL Testing
Tuesday, August 13, 2002

for research project of: Lindsey Spratt
spratt@eecs.ukans.edu

SPARCL has an “active” tutorial built in to it. It is accessed (primarily) from the “about” screen via the “tutorial” button.

For this study, please invoke the tutorial and follow it. There are a few exercises—please attempt them.

SPARCL (currently 2D5) is on the two “big” Macs in the DesignLab at:

HardDisk:Projects:SPARCL

When you start using it, be sure that you set your user name—this helps the data collection for the study.

There are various programs associated with the tutorial, if you would like to look at them outside of the tutorial, they are in a folder named “Tutorial Scripts *f.X*” and their names end in “VPSL”. Other example programs are in a folder named “Examples”. Both of these folders are in the SPARCL application folder.

Please use the “Record Comments...” facility freely, tell me any thoughts you have about SPARCL and the tutorial process as you are working through it. This is on the “File “ menu and can be accessed via command-R.

The interaction data and comments will be more interesting to me if each of you works alone, rather than collaborating.

Finally, relax and keep in mind that you are testing SPARCL and its tutorial—not the other way around.

Thank you for your help,
Lindsey Spratt.

Figure 9. 5: SPARCL study handout.

For each user there were several files: an interaction log, a record of that user’s comments (made through the “Record Comments about SPARCL...” option of the “File “ menu in the SPARCL application), and the exercise programs which that user created for the exercises of the tutorial.

The tutorial was entirely self-explanatory, no other instruction or aid was provided to the study participants. Whenever a user quit the SPARCL application, it would ask that user if she would like to send her files to the researchers running the experiment. If the user said yes, then SPARCL would package the relevant files into an email message and send this message to us. If the user said no, then no files were sent (although they remained on the system so that we could “man-

ually” retrieve them later). This structure of the tutorial, complete self-explanation, automated data gathering, and automated email delivery of the data, was developed so that the SPARCL can be distributed freely over the Internet and a wide variety of individuals may contribute to the user testing. The Internet distribution has not yet been

attempted.

Results: In General.

The study participants each spent two to four hours on the tutorial. Unfortunately, many of them only ran the first chapter of the tutorial. One participant completed all three chapters and the four exercises. The study had 7 participants. There were two versions of the tutorial, 0.5 and 1.0. The early version, 0.2, was used by only one of the participants. There were two versions of the tutorial used in the study, 2.D.5 and 2.D.6. The earlier version, 2.D.5, was used by one participant ('vn') for that person's entire participation and by another participant ('kc') for a very brief session. The tutorial changed substantially from 0.2 to 1.0; the entire second and third chapters were added. Participant *vn* needed to start early, so the tutorial was not quite ready. SPARCL changed in various ways from 2.D.5 to 2.D.6, but these did not greatly affect participants *vn* or *kc*.

Results: Exercises.

There is a program file created by the tutorial and edited by the user associated with each of the four exercises. If the user invokes the tutorial script for a particular exercise multiple times, then there are multiple program files for that exercise. We analyzed these program files to determine how successful the participants were at carrying out the exercise tasks.

All of the results are given in Table 9. 1. Table 9. 2 summarizes these results. There are four "exercises", each of which presented the user with three or more tasks. In the exercise for section 1 of chapter 1 ('1.1') these tasks are labeled 'A', 'B', and 'C'. In the exercise for section 2 of chapter 1 ('1.2'), these tasks are '1 part A', '1 part B', '2', and '3'. In the exercises for section 1 of chapter 2 ('2.1') and section 1 of chapter 3 ('3.1'), these tasks are 'A', 'B', and 'C'. The participants did not even attempt the six tasks of the exercises of chapters 2 and 3, with the notable exception of 'ca' and an unsuccessful attempt at the first of these tasks by 'kc'. Generally, the seven tasks of the first two exercises seemed to be readily achievable given the information in the tutorial. The tasks of the second chapter exercise involved constructing mildly com-

User	1.1	1.2	2.1	3.1
ag	part A correct, didn't attempt parts B and C.	Answered all parts correctly.		
ca	Answered all parts correctly.	1ab correct. 2 nearly correct, missing link for grandparent. 3 correct.	all parts correct. Used N-tuple tables to make complex terms instead of using ntuples directly.	3 attempts, none successful at any part. Only one attempt is nonempty. This contains a single clause of the form: ((("Maximal Range Value"=> _ => _) => {(fails => ((unify => new_ur) => {}}))})
el	Answered all parts correctly.	1A, incorrect, fact table of two "Happy" people (Tom and Pat). 1B, correct. 2 is correct. 3 is nearly correct, arguments to "Sister" are reversed ("Sister" arguments are in the other order for part 2).		
jp	All three parts are correct. two attempts, the first attempt correctly answered parts A and B.	Only attempted 1A, which is correct.		
kc	Answered all parts correctly.	Answered all parts correctly.	No correct answers, only attempted part A. Created heavily nested ntuples: (((({} => {})) => [table, 1 N-tuple row]) => 34) => 1) => 2)	
rv	All parts correct. Part C exactly correct in that it does NOT have an argument.	parts A and B correct, except the clauses are not correctly named.	[file lost]	
vn	A and C not attempted. B is correct.			

Table 9. 1: Exercise assessments.

plex terms, with nesting of N-tuples. These seem to confuse most of the participants. Presumably, the tutorial needs more careful presentation of this information.

Results: Interaction logs.

Use- r	1.1a	1.1b	1.1c	1.2: 1a	1.2: 1b	1.2: 2	1.2: 3	2.1a	2.1b	2.1c	3.1- a	3.- 1b	3.- 1c
ag	+			+	+	+	+						
ca	+	+	+	+	+	±	+	+	+	+	-		
el	+	+	+	-	+	+	±						
jp	+	+	+	+									
kc	+	+	+	+	+	+	+	-					
rv	+	+	+	+	+			?	?	?			
vn		+											

Table 9. 2: Summary of exercise assessments.
('+'=correct, '-'=incorrect, '±'=nearly correct, '?'=missing)

There are one or more interaction logs for each participant. These logs have been analyzed in various ways, including: interaction frequencies by duration and interaction type (e.g. one interaction type is “script_control^step with next step of ‘<< (Please read the script commentary) >>’ and current script of ‘Rules - extending the Parents program’), and interactions identified by type of participant’s larger activity (e.g. “reading section 1.2”). The results of these analyses can be seen in tables in this chapter and in appendix 4 (“Interaction Log Data”) (“Interaction Log Data”). The appendix tables and figures are:

1. Histogram of durations in seconds for all interactions.
2. Duration statistics for all participants organized by action type.
3. Histogram of interaction type frequencies by major action type for all types.
4. Histogram of interaction type frequencies by major^minor action type for all types of interactions.
5. Histogram of interaction type frequencies by major action type for “exercise” interactions
6. Histogram of interaction type frequencies by major^minor action type for “exercise” interactions.
7. Interaction type counts grouped by major action type, minor action type, first argument, and second argument.

Response time and think time. One of our concerns in preparing for the study was the speed of SPARCL’s responses. We were concerned that if SPARCL responded too slowly to common actions, then the users would become frustrated and this would inhibit their use of (and appreciation for) the system. For this reason, this was an area in which we had spent a lot of effort. In reviewing the results of the study, we found

that only one of the participants remarked on the responsiveness of the system. This person complained that the updating of the display was too slow. Thus, apparently, SPARCL was generally responsive enough to meet our goal of not frustrating the user.

One particular response-time problem on which we worked was the presenting of the appropriate popup menu in a program window when a user presses the mouse button with the cursor over some SPARCL display object. This popup menu presentation was initially implemented in a simple fashion which proved to be much too slow. To explain why it was too slow and how the problem was solved, we must first explain the general problem involved of presenting a popup menu. Each kind of display object has a distinct menu, and the interaction processing must determine the appropriate kind of menu to display. This is complicated by the fact that the display objects can be nested, for instance a variable nested in an argument box nested in a clause. Thus, determining the appropriate object requires selecting not simply the object under the cursor, but the object with smallest frame under the cursor.

The process for displaying a popup menu is as follows: When the mouse button is depressed, SPARCL receives a “mouse down” event with the coordinates of the cursor “point”. SPARCL must then determine which display objects “frame” that point and of these display objects determine which one has the smallest frame. (If a user depressed the mouse button while the cursor was over the variable-in-an-argument-in-a-clause, the variable display object would be found to have the smallest frame.) Having selected a display object, SPARCL determines the popup menu associated with the type of the display object. Then the items of that popup menu are processed for enabling/disabling according to the current state of the SPARCL environment. Finally, the popup menu is presented to the user. The problematic steps in this process are those for finding the smallest object framing the cursor point. There may be one or two hundred display objects for even a relatively simple program window. The search among these objects requires comparing the two values (the coordinate) of the cursor point to one to four values (the frame) per object (at least implicitly) to determine if that object’s frame contains the cursor point, and then it is necessary to compare each of the containing frames to each other to determine which frame is smallest. We initially used a simple list and compared items exhaustively, but clearly this was inadequate. It could take several seconds to determine the appropriate object. This was solved by implementing a special index (in PROLOG) which indexes the display objects in a window by their frames (top left corner coordinates and depth and width).

The search of this index returns all of the containing frames, ordered from smallest to largest so that it is trivial to determine the object with the smallest containing frame. After implementing this special index, the popup menu appears nearly instantaneously even in program windows with many hundreds of objects.

The interaction log data duration analysis gives us an idea of how responsive the system was and how much time the users spent thinking. Certain types of interactions should have short durations. These types include most of the program window display object popup menu actions (e.g. “Argument Ops^Insert: Variable”, “Clause Ops^Create Argument”). To compare with the study participants experience, we conducted a simple test of the responsiveness of the system, independent of the usability test. We constructed a program of four clauses, each clause with a single literal. The literals and clauses each had two or three arguments. In our use of SPARCL, we observed a reduced responsiveness when there were many programs loaded. To test this, we ran the simple responsiveness test once with no other programs loaded, and once with a relatively large project loaded (ID3, 766 display objects in SPARCL’s internal database, 34 literals, 36 clauses). The test was run on an Apple Macintosh 8500/120 with 32Meg RAM, with 16Meg application memory for SPARCL. The results of this test are given in appendix 5 (“Response Time Data”).

The durations for the “unloaded” test are summarized in Table 9. 3. This shows that the average and median duration for several simple editing interaction types are two to three seconds. Notable exceptions to this are the ‘Clause Ops^Create Literal’ and ‘Program Ops^Create Clause’ interaction types. The ‘Clause Ops^Create Literal’ type interactions had an average duration of 5.2 seconds and a median of 7 seconds. The ‘Program Ops^Create Clause’ type interactions had an average duration of 3.5 seconds and a median duration of 4 seconds. Creating a literal is more time-consuming due to greater complexity of modifying the representation of a clause to fit the new literal into that clause. Creating a clause generally requires no changes to existing display objects. The “loaded” test (i.e. with the ID3 project loaded) took about twice as long as the “unloaded” test. This is presumably due to time spent searching the display object database, although no careful analysis has been made of this issue.

The tutorial-based study had very few “other” programs loaded when the participants were working on the exercises, and so the responsiveness of the system for them should be compared with the “unloaded” results of the responsiveness test.

There are some types of interactions which we expect were generally part of a

operation	count	total	avg	std dev	median	min	max
Argument Ops	19	38	2	0.9	2	0	4
Insert:Variable	19	38	2	0.9	2	0	4
Clause Ops	12	36	3	2	2	1	7
Create Argument	8	15	1.9	0.3	2	1	2
Create Literal	4	21	5.2	2	7	2	7
link	10	24	2.4	1.2	2	1	5
Literal Ops	7	15	2.1	0.3	2	2	3
Create Argument	7	15	2.1	0.3	2	2	3
window	6	13	2.2	1.1	2	1	4
activate	6	13	2.2	1.1	2	1	4
general_tool	6	10	1.7	0.7	2	0	2
activate	4	6	1.5	0.9	2	0	2
**** SKIP 1 type. ****	2						
Program Ops	4	14	3.5	1.1	4	2	5
Create Clause	4	14	3.5	1.1	4	2	5
connect_tool	4	9	2.2	0.8	3	1	3
activate	4	9	2.2	0.8	3	1	3

Table 9. 3: Selected interaction durations for the “unloaded” responsiveness test grouped by interaction type (major action type/minor action type).

sequence of operations, where the user followed the interaction with another interaction as rapidly as the system allowed. Most of the Argument Ops, Clause Ops, Set Ops, Term Table Cell Ops, Literal Ops, Ur Ops, Partition Ops, Variable Ops, Term Table Ops, and NTuple Ops interactions are of this type, with the exception of the Clause Ops/Create Comment and Clause Ops/Query:Brief interactions. Creat-

ing a comment was generally followed by a significant period of time editing the comment being created. Executing a query could take an arbitrarily long period of time. Table 1 of appendix 4 (“Interaction Log Data”) shows the median durations for these “sequence” interaction types vary between 6 and 10 seconds, with the exception of the NTuple Ops interactions, which have a median of 15 seconds.

The “sequence” type interaction median duration of 6 to 10 seconds for the study is substantially longer than the average (with small standard deviation) durations of 2 to 5 seconds (or median durations of 2 to 7 seconds) in the simple responsiveness test. The study participants took roughly twice as long as necessary to execute these interactions. We speculate that this difference is due to the participants needing to think about what to do, while no “think” time was used in the simple responsiveness test. If

the simple responsiveness test time is taken to be the true system response time, then this can be subtracted from the study interaction duration to determine the think times. The respective think times are shown in Table 9. 4. The interaction types with the lowest common think times are “connect_tool^activate” (clicking on the “connect tool” icon in the program window in preparation for linking two terms in that window), “Literal Ops^Create Argument” (creating an argument in a literal), “Clause Ops^Create Argument” (creating an argument in a clause), and “general_tool^activate” (clicking on the “general tool” icon in the program window in preparation for any nonlinking editing interaction in that window). That these interaction types have shorter think times is what one might expect. Selecting a tool (connect or general) is usually done with some idea of what one will do next, the action which the tool selection enables. Also, creating an argument in a clause or literal is a simple action where the user generally has a definite idea of what to do next (e.g., create another argument, or fill in the one just created with a variable). In the middle between the four interaction types with the shortest think times and the four interaction types with the longest think times is the interaction type “Argument Ops^Insert:Variable”.

The four longest think times are for the interaction types “Program Ops^Create Clause” (create a clause in a program window), “link” (link two terms together), “window^activate” (make a window the front (active) window), and “Clause Ops^Create Literal” (create a literal in a clause). It is reasonable to expect the user to be somewhat more reflective following interactions of these types than for the other interaction types. Having created a clause but before “filling it in” with arguments and literals, the user may need to stop and consider how best to construct the clause

operation	Basic duration			Study duration			Think time (study med - basic avg)
	avg	std dev	median	avg	std dev	median	
cnn.tl^activate	2.2	0.8	3	10.9	28	5	2.8
LtrOp^Create Argument	2.1	0.3	2	7.2	3.9	6	3.9
ClsOp^ Create Argument	1.9	0.3	2	9.5	7.2	6	4.1
gnr.tl^activate	1.5	0.9	2	30.6	166.3	6	4.5
ArgOp^Insert:Variable	2	0.9	2	9.3	14.1	7	5
PrgOp^Create Clause	3.5	1.1	4	12.3	9.7	9	5.5
link	2.4	1.2	2	14.5	18.3	8	5.6
window^activate	2.2	1.1	2	19.1	55.7	9	6.8
ClsOp^Create Literal	5.2	2	7	15.8	13.6	13	7.8

Table 9. 4: Typical “think” times for common editing interactions.

overall—what is the purpose of the clause. A similar situation occurs when creating a literal. The activation of a window requires a few moments for the user to view the window and decide on the next step. An interesting result in this set of values is the relatively long think time (5.6 seconds) for the “link” interactions. Apparently, users typically pause some time after a link to consider the result and how to proceed; notably more so than for argument creation.

Object selection accuracy. Since the primary interaction in the editing environment involves placing the cursor over an object to get that object’s popup menu, it is important that the user be able to easily determine whether the cursor is over the desired object. There are two aids for this in the editing environment. One of these cues is simply the visual cue—the cursor looks to be over the object of interest. The other cue is the object type name at the bottom of the popup menu. This second cue can be ambiguous since there may be more than one object of a given type in the region of the object of interest. Additional cues may be desirable, such as highlighting the object which is under the cursor at each moment.

Unfortunately, the interaction logging is insufficient to shed light on the object selection accuracy question. The logging could be extended to note each popup menu invoked, whether or not a selection was made from the popup menu. We might infer that the wrong popup menu was invoked if a popup menu was “popped up” but no selection was made. Currently, only popup menu selections are recorded. Also, the logging could be extended to track how much time the user spends moving the cursor around. A lot of time spent moving the cursor prior to popping up a menu could indicate that the user was having a hard time positioning the cursor.

Interactions by section and exercise. We analyzed these logs to group sequences of interactions into activity “portions”, and each portion was assigned a category: reading a section, working on an exercise, or “other”. The times associated with these categories are given (in minutes) in the following tables: Table 3 of the appendix 4 (“Interaction Log Data”) and Table 9.5 of this chapter show the durations per participant per section and various statistics by section of these durations per participant, respectively. Tables 4 and 5 of appendix 4 (“Interaction Log Data”) show the durations per participant per exercise and various statistics by exercise of these durations per participant, respectively. Table 6 of appendix 4 (“Interaction Log Data”) and

Statistics	count	total	Sections								
			1.1	1.2	1.3	1.4	2.1	2.2	2.3	2.4	3.1
Count	7	7	7	7	7	6	5	5	4	4	4
Total	49	755.4	273.4	158.6	63.1	11.9	111.1	16	11.7	6.6	103.1
Avg	7	107.9	39.1	22.7	9	2	22.2	3.2	2.9	1.6	25.8
Dev	2.4	46.3	13.4	8	5.6	0.8	14.5	1.4	1	1	11.7
Median	9	81.5	36.2	21.9	9.2	1.6	13.8	3.1	2.9	1.7	29.5
Min	3	46.2	22.4	11.9	1.8	1.4	7.7	0.9	1.7	0.5	10
Max	9	186.4	62.3	33.2	21	3.6	45.2	4.7	4.4	3.2	42.2

Table 9. 5 Participant duration statistics by section. “total” column is statistics for total over all sections per participant. “Count” column is statistics for count of sections per participant.

Table 9. 6 of this chapter show the total durations per participant per category and various statistics by category of these durations per participant, respectively.

Table 9. 6 shows the participants averaged 107.9 minutes working through the sections (with a median of 81.5), 43 minutes working on the exercises (with a median of 32.3), and 17 minutes on “other” activities (with a median of 19.9). The uncategorized activity, “other”, is surprisingly high. They averaged 167.8 minutes overall on the tutorial (with a median of 131). Their individual values in these areas are quite varied as can be seen from the fairly high values for the standard deviations.

Since the participants differed greatly in how much of the tutorial they attempted, it is difficult to compare their activities. There were 9 sections to read. Table 9. 5 shows the average number of sections read was 7 with a median of 9. Four participants read all 9. The sections in which most of the time was spent overall were 1.1, 1.2, 2.1 and 3.1, in order of decreasing total duration. If we consider the average or median times spent in a section as a measure of how demanding that section was, then Table 9. 5 shows the same four sections are the most demanding, but in a slightly changed order: 1.1, 3.1, 1.2, and 2.1 (in decreasing order for either average duration or median duration).

There are two ways we have measured the sizes of the scripts for the sections and exercises. One is simply the number of bytes in the script. The other is the number of steps which the user must execute to see the entire script. The

Statistics	All Sections	All Exercises	Other	Total
Count	7	7	7	7
Total	755.4	300.7	118.8	1174.9
Avg	107.9	43	17	167.8
Dev	46.3	27.7	6.6	65.3
Median	81.5	32.3	19.9	131
Min	46.2	16	8.7	91.3
Max	186.4	101	25.5	263.7

Table 9. 6: Statistics on total times.

sections have the script sizes in bytes and numbers of steps as shown in Table 9. 7.	Section	Bytes	Visible Steps	Bytes/Steps	Median Minutes
	0.0	7100			
	1.0	3147			
	1.1	16446	242	68	36.2
	1.2	10267	44	233	21.9
	1.3	8627	45	192	9.2
	1.4	2479	5	496	1.6
	2.0	3000			
	2.1	9871	37	185	13.8
	2.2	4471	22	203	3.1
	2.3	4760	15	317	2.9
	2.4	8029	7	1147	1.7
	3.1	16978	140	121	29.5

Table 9. 7: Section sizes.

The exercises have the script sizes shown in Table 9. 8. The “number of steps” approach to counting steps does not differentiate between different kinds of steps, a simple action such “open program” is a single step just as is a very long single comment. Since comments can be highly variable in length, some account for how big a comment is should perhaps be included in the “steps” size estimate.

Exercise	Bytes	Visible Steps
1.1ex	1251	5
1.2ex	1222	5
2.1ex	1516	5
3.1ex	5171	5

Table 9. 8: Exercise sizes.

There is no simple relationship between the sizes of the sections and the amount of time the participants spent on them. There is an approximately monotonic increasing relationship between the average times and the size in bytes (this is not honored by sections 2.2 and 2.3). Sections 1.1 and 3.1 are much larger than the other sections because they build SPARCL programs step-by-step, and comment on many of these steps. The other sections simply present SPARCL examples by opening saved programs, instead of building the examples in the script.

Interaction frequencies. Histograms of the frequencies of occurrence of the interactions are shown in the figures 2, 3, 4, and 5 of appendix 4 (“Interaction Log Data”). Much the most common interaction type was “script_control^step” (clicking the “Next Step” button in the script dialog). This occurred 3106 times.

The top 10 interaction types (those accounting for more than 2% of the total of the nonscripting interactions) divide by frequency into several clusters:

- cluster 1— Argument Ops^Insert:Variable (149);
- cluster 2— general_tool^close_edit (125);

cluster 3— Clause Ops^Create Literal (82), link (77), File^Record Comment about System... (74);	User	Comments
cluster 4— Program Ops^Create Clause (53), Clause Ops^Create Argument (51); and	ag	8
cluster 5— Clause Ops^Query:Brief (41), Clause Name Ops^Edit Clause Name (36), Argument Ops^Insert:Ur (31).	ca	14
	el	18
	jp	9
	kc	19
	rv	3
	vn	3

Table 9. 9:
Number of
comments
recorded by
each user.

An interesting absence from this list of interaction types is “Literal Ops^Create Argument”. One might expect it to be present due to the large number of “Argument Ops^Insert:Variable” (149) and “Argument Ops^Insert:Ur” (31), but only 51 “Clause Ops^Create Argument”. At one insertion per argument (and no deletions), this leaves about 130 arguments unaccounted for. These are the literal arguments. They are being created without an explicit interaction by the user. This is due to an optimization in the SPARCL interface. When creating a literal, the user selects a name and arity for the new literal from a list of procedures (names and arities) currently defined by the user. The new literal is then created with the selected name and number of arguments already in place. This both saves the user time and reduces the number of programming errors due to mismatched numbers of arguments.

Results: System comment files.

The numbers of system comments made by the participants are summarized in Table 9. 9. Some of the participants also provided some comments by email directly to the researchers. The comments were of various kinds, including problems with the tutorial and the scripting mechanism, assessments of the tutorial and SPARCL, and questions about the tutorial and SPARCL. There were several requests for the ability to go back to earlier points in a script, at least to be able to go back to the previous step. Several participants commented that they found the coreference link representation to be very understandable and a distinct improvement over variables in PROLOG. The term “ur” confused several people.

Discussion.

Summary. The usability testing consisted of having seven graduate students work through an online, integrated tutorial of SPARCL. Their interactions were recorded by SPARCL, and they were encouraged to record comments freely during the course of the tutorial. The data from the testing consists of the interaction log files, their recorded comments, and their solutions to the exercises in the tutorial.

We summarized the tutorial preface and provided an outline of the tutorial's contents. We gave the exercises in full. An adaptation of the entire tutorial is given in appendix 1 ("Tutorial Introduction"). We provided statistics characterizing the tutorial section sizes in Table 9. 7 and the exercise sizes in Table 9. 8. The sizes of a section are given in number of bytes (of the defining script file) and number of visible steps (to execute the script for that section).

There are three kinds of data from the experiment: the interaction logs, the system comment files, and the exercise files. In analyzing the exercise files, we observed that exercise groups 1.1 and 1.2 were attempted by most participants, that 2.1 was attempted by only two participants, and only the first task of 3.1 was attempted (unsuccessfully) by only one participant. The seven tasks of the first two exercises seemed to be readily achievable given the information in the tutorial. The tasks of the second chapter exercise involved constructing mildly complex terms, with nesting of N-tuples. These seem to confuse most of the participants.

In analyzing the interaction logs we looked at the durations of the interactions, the frequencies of the occurrence of interactions by type of interaction and type of larger activity (e.g. "reading section 1.2", "working on exercise for section 2.1") of which the interaction is a part. The results of these analyses are presented in several tables in this chapter and in appendix 4 ("Interaction Log Data"). We used the duration data to study issues in response time and think time, object selection accuracy, and section and exercise activity durations.

Our analysis of response time was motivated by concern that if SPARCL responded too slowly to common actions, then the users would become frustrated and this would inhibit their use of (and appreciation for) the system. Because of this concern, we devoted substantial effort to making the implementation what we hoped would be adequately responsive. The response of the interaction is dependent on two activities: presenting the appropriate popup menu in reaction to a "mouse down" event over a program object and redisplaying a program to reflect editing changes. According to our analysis, SPARCL's response time for popping up menus was entirely adequate.

The redisplay time was generally acceptable, although we did discover that it degrades seriously as the amount of SPARCL “code” loaded into the environment at one time grows. Since the tutorial does not require large amounts of code to be loaded at one time, this response time degradation was not a problem.

We found that the interaction log data is insufficient to shed light on the object selection accuracy question. The logging could be extended to note each popup menu invoked, whether or not a selection was made from the popup menu. Currently, only popup menu selections are recorded. Also, the logging could be extended to track how much time the user spends moving the cursor around.

The duration data organized by participant and section shows that the sections that had the most total time spent in reading them were 1.1, 1.2, 2.1 and 3.1, in order of decreasing total duration. Considering the average or median times spent in a section as a measure of how demanding that section was, the same four sections are the most demanding, but in a slightly changed order: 1.1, 3.1, 1.2, and 2.1 (in decreasing order for either average duration or median duration).

This notion of “demanding” sections based on durations corresponds well to the size in visible steps for sections 1.1 and 3.1. They are much larger in visible steps than the other sections because they build SPARCL programs step-by-step, and comment on many of these steps. The other sections simply present SPARCL examples by opening saved programs, instead of building the examples in the script.

Overall, participants averaged 107.9 minutes working through the sections (with a median of 81.5), 43 minutes working on the exercises (with a median of 32.3), and 17 minutes on “other” activities (with a median of 19.9). The uncategorized activity, “other”, is surprisingly high. They averaged 167.8 minutes overall on the tutorial (with a median of 131).

Much the most common interaction type was “script_control^step” (clicking the “Next Step” button in the script dialog). This occurred 3106 times. The five most common non-scripting interaction types (with their number of occurrences in the test data) were “Argument Ops^Insert:Variable” (149), “general_tool^close_edit” (125), “Clause Ops^Create Literal” (82), “link” (77), and “File^Record Comment about System...” (74). The “Literal Ops^Create Argument” interaction is not even in the top 10, although one would expect about 130 interactions of this type (given the other interactions) to create literal arguments. Instead they are being created without an explicit interaction by the user due to an optimization in the SPARCL interface.

The system comment file contained comments on a variety of topics. Three recurring comments of interest here are: requests for the ability to go back to earlier points in a script, at least to be able to go back to the previous step; comments that the coreference link representation to be very understandable and a distinct improvement over variables in prolog; and, the term “ur” was confusing.

Assessment. The usability testing was too limited to provide much insight into the usability of SPARCL. It did show that SPARCL is usable in simple ways and that the representation was understandable for very simple programs. The study participants could not understand the one somewhat complex program in the tutorial. We don’t know to what extent this is a failing of the language design or the tutorial. We can make some assessments that are more specific to the implementation of the tutorial and of SPARCL.

The tutorial “instrument” is clearly still in need of development. The tutorial system needs to be able to back up, as several participants requested. Also, several participants would like to have been able to actually do the various steps which the tutorial was demonstrating. The tutorial system could allow the user to “do” the next step, correcting the user if she does the wrong thing, or the user could choose to have the system “do” the next step (as it does now). This is similar to the approach supported by the Apple Guide help system used on the Apple Macintosh.

The interaction logging technique was useful, but was not able to shed light on some questions. As we discussed above, the object selection accuracy question can not be addressed using the interaction logging data. Some extensions to the logging technology were discussed above which could make it useful for this question. These include logging menu “pop ups”, not just menu selections, and logging time spent in cursor movement.

The tutorial content appears to start out well, in that the initial sections were handled well by the participants. But, the later sections are too “aggressive”; they try to cover much too much material and the participants were left confused. The tutorial may need to be about twice as long to adequately present most of the concepts it currently covers. This increased length leaves the tutorial at a still reasonable length, about six to eight hours. This compares reasonably with the two automated SMALL-TALK tutorials at six and 12 hours.

The editing environment may be more difficult to learn to use than we had

expected, although the participant's problems with this may be due to inadequate instruction in the tutorial. The particular editing procedure that seemed confusing was the creation and extension of N-tuples. An N-tuple is created by popping up a menu for at an existing term and selecting a "Create N-Tuple:<term type>" option, where "<term type>" is one of "Variable", "Ur", "Set", "Table", or "IntenSet". This replaces the existing term with an N-tuple that has the existing term as its first element and a term of the chosen type as its second element. An N-tuple is extended by popping up a menu at the N-tuple (*not* above one of the elements of the N-tuple), and selecting "Extend with:<term type>". This adds a term of the chosen type to the end of the N-tuple. None of the participants extended any N-tuples.

We were concerned that the term-linking procedure might prove tedious, but none of the participants appeared to have any trouble with it. We were concerned that it might be tedious due to the necessity to place the cursor very precisely when linking certain kinds of terms, particularly variables. This might have become a problematic feature if the participants had produced programs requiring the linking of more complex terms (e.g. parts of partitioned sets within N-tuples).

The SPARCL response time for simple editing interactions appeared to be generally acceptable. However, this response time degrades substantially when there are many display objects (clauses, literals, terms, etc.) in the environment. This is an area which needs work to prevent the response degradation. Also, edits which cause apparently minor modifications to a complex representation can lead to a slow response due to a long time for redisplaying the modified representation. Better localization of the redisplay could help here.

The SPARCL representation was thought to be good by the participants, judging by their recorded comments. Particularly, there were several positive comments about the representation of variable coreference by connecting lines instead of naming. Contrary evidence on the understandability of SPARCL by the participants is that the final program (Column Sum) may not have been understood by any of the four participants who read that section (3.1).

From this experience the SPARCL environment appears promising, but the construction and use of the complex terms needs careful and extensive explanation. The two SMALLTALK tutorials mentioned above required more time to complete (6 to 12 hours) than we had planned for the SPARCL tutorial (2 to 4 hours). A SPARCL tutorial which is modified to more gradually introduce complex terms and to more fully

explain the execution model should still fit within the 6 to 12 hour range of these other tutorials (and be much closer to the 6 than the 12 hour duration).

- Bratko 1990 *PROLOG programming for Artificial Intelligence, 2nd Edition* by Ivan Bratko.
Addison-Wesley Publishing Company:Reading, Massachusetts. 1990.
- Carroll&Rosson 1995 “Managing evaluation goals for training” by John M. Carroll and Mary
Beth Rosson, pages 40-48 in *Communications of the ACM* **38**(7), July 1995.
- Compeau et al. 1995 “End-user training and learning” by Deborah Compeau, Lorne Olfman,
Maung Sei, and Jame Webster in *Communications of the ACM* **38**(7), July, 1995.
- Kay&Thomas 1995 “Studying long-term system use.” by Judy Kay and Richard C. Thomas. Pages
61-69 in *Communications of the ACM* **38**(7), July 1995.

Chapter 10

Future Work

There are many aspects of SPARCL in that provide an opportunity for further research. We present some of these here.

Design Elements. We noted in chapter 3 (“Design Elements”) that the concrete visual representation of delay specifications needs to be redesigned to be made more compact than it currently is. This could substantially reduce the size of SPARCL programs. There are various extensions to the representation that we hope to investigate. These include: lists, matrices, and directed graphs.

Some linear textual languages have very adaptable syntax that the programmer tailors to the problem at hand. In Edinburgh-style Prolog, the programmer may define the syntax of an operator: its name, its position (prefix, postfix, or infix), its precedence, and its associativity (does or doesn’t associate for unary operators; associates left, associates right, or doesn’t associate for binary operators).¹ There is also a facility for transforming clauses which are being interpreted, called *term_expansion*². Term expansion is a read-only operation - the original form of the clause is unavailable. There is a write “hook” which is the *print/1* predicate. This can be used to arbitrarily reformat system output (such as that produced by the debugger or the top-level interpreter loop). Thus, the programmer can change the *external* (what is read and written) syntax of Prolog in fairly dramatic ways. Lisp has its macro facility, which only affects the reading of s-expressions³. There is an extensive macro preprocessor associated with C which can dramatically alter the syntax of the language. It only applies to reading the language, however. These macro facilities are loosely analogous to the *term_expansion* facility of Prolog. There may interesting analogs in a visual logic programming language for the kinds of flexibility which Prolog provides via operator definitions, and which macro facilities provide. Approaches to this question are closely related to how one addresses the I/O problem identified earlier. Also, we would like to provide an adaptable syntax for SPARCL. This is discussed in more detail below.

1. p. 26 of [Clocksin&Mellish 1992].

2. ff. 303 in [O’Keefe 1990].

3. ff. 193 in [Steele 1990].

We expect to do further research into Input/Output system of sparcl. The immediate problem is to provide for a way for a user to interact with SPARCL during the interpretation of a query. This is necessary to be able to build self-contained interactive applications in SPARCL.

Partition Structured Unification. In chapter 4 (“Partition Structured Unification”), we discussed several points related to the unification algorithm formalization, analysis, and implementation. We would like to create a more concise form of the formalization. The analysis should be completed to include soundness and completeness of the formalization and minimality of the implementation. There are several aspects of the implementation that can be worked on to improve the performance of SPARCL: more sophisticated constraint analysis to recognize some cases of invalid collections of nonground constraint partitioned sets; specialization of the unification algorithm either per clause head or via partial evaluation of the SPARCL interpreter.

Three-dimensional Representation. We expect to explore some of the possibilities for representing comments, as well as developing table representations, modelling multiple clause scenes, and making the 3D representation interactively editable.

The strong similarity in approaches to automated layout for 2D and 3D representations should make it easier for us to combine these approaches. Such a combination would yield an incrementally updated layout for the 3D representation (such as is currently implemented for the 2D representation). This is essential for interactive editing of the 3D representation.

Implementation. Substantial performance improvements are possible in the interpreter’s clause database searching by an indexing scheme for the clauses based on their arguments, and by specializing the unification procedure for each clause head. Another common optimization for logic programming interpreters is to compile to a byte-code (which is interpreted by a heavily optimized run-time written in a low-level language such as C or assembler) or even to compile to “native code”. However, there are several algorithmic improvements that we expect to investigate before we consider moving the run-time of SPARCL out of PROLOG. Changing the interpreter to compilation makes it much more complex, and therefore harder to change, and it will no longer automatically benefit from improvements to PROLOG implementation per-

formance.

Subjective analysis. The “lazy” unification discussed in chapter 7 (“Subjective Analytic Assessment”) may provide another substantial performance improvement, and we hope to investigate it. The analysis to support this “lazy” unification may be subtle and there may be substantial changes needed in the implementation of the interpreter to support it.

We found the ‘*TERM*/1 predicate useful at several points in the SPARCL solutions to the example problems. The ‘*TERM*/1 literals were used to provide a place to put partitioned set terms that didn’t have a “natural” home elsewhere in the clause. This would be reasonably represented by suppressing the display of the ‘*TERM*/1 literal and just displaying the contents of its argument—”free floating” in the clause. There are various relatively minor adjustments required in various parts of the editing and display systems to support this. We plan to investigate this alternative representation.

Objective analysis. We want to investigate making the representation of set relationships (e.g. union, intersection, difference) more diagrammatic.

Usability testing. We would like to modify the interaction logging mechanism. The logging could be extended to note each popup menu invoked, whether or not a selection was made from the popup menu. We might infer that the wrong popup menu was invoked if a popup menu was “popped up” but no selection was made. Currently, only popup menu selections are recorded. Also, the logging could be extended to track how much time the user spends moving the cursor around. A lot of time spent moving the cursor prior to popping up a menu could indicate that the user was having a hard time positioning the cursor.

Further work on the tutorial system will improve its usability (as opposed to the usability of SPARCL): support “backing up” at least one step, preferably more; and, allow the user to “do” the next step, instead of just telling them how it should be done and then doing it for them. The tutorial contents should be changed to more gradually introduce the more advanced concepts of SPARCL, aiming at a total tutorial duration in the range of 6 to 12 hours instead of the current 2.5 hours.

The editing environment interaction model and specific choices of available com-

mands needs more study. Particularly the construction of N-tuples, which appears to be confusing as currently implemented.

Debugging. There is a lot of research in debugging environments for visual and logic programming languages. It should be interesting to investigate a debugging environment for a programming language which is both visual and logic-based.

- Clocksin&Mellish 1992 *Programming in Prolog* by Clocksin and Mellish. Third edition. 1992.
- O'Keefe 1990 *The Craft of Prolog* by Richard A. O'Keefe. MIT Press, Cambridge, MA. 1990.
- Steele 1990 *Common LISP* by Guy Steele, Second edition. Digital Press. 1990.

Chapter 11

Conclusions

At the beginning of chapter 1 we stated that the hypothesis of our research is that visual logic programming based on sets with partitioning constraints provides a superior basis for exploratory programming languages. We further specified that our research program had two major objectives: *feasibility*—test the feasibility of this approach to programming languages by designing and implementing such a programming language with an integrated development environment; and, *desirability*—assess the desirability of this approach to programming languages by analyzing this implementation.

We have fully achieved the first objective by our successful design and implementation of SPARCL. There were many problems to overcome in all aspects of creating SPARCL, its syntax, semantics, and integrated development environment. The second objective is only partially achieved in this thesis. Also, as chapter 10 (“Future Work”) indicates, this research project has many opportunities for additional research.

Demonstrating feasibility. There were a number of problems to overcome in the development of a semantics for SPARCL that followed the hypothesis: logic programming based on sets with partitioning constraints. The most difficult was developing the partitioned set unification algorithm. The idea of partitioned set unification and the accompanying algorithm are perhaps the major single technical contribution of this research project. Another complex area in the semantics was integrating partitioned set unification, partition constraint handling, and the delay mechanism into the interpreter. To simplify the implementation of this interpreter, we simplified the language it interprets compared to the SPARCL language seen by the programmer. This required that we develop a program transformation system to translate between these forms of SPARCL. The main complications addressed in this transformation deal with generating coreferences when going from internal form to external form and with variable scoping analysis when going from external form intensional sets to the internal form.

The design of a suitable representation and the construction of an integrated development environment (IDE) to work with that representation also presented

numerous difficulties. Our use of hyperedges in the representation of SPARCL is a novel and reasonably successful solution of the problem of how to represent networks of connecting lines. To make a fully visual and diagrammatic representation easy to work with requires a great deal of support from the IDE. In particular, the IDE must be able to help with the creation and layout of the representation. We made this almost completely automated: the programmer has very little involvement in layout decisions. Since the layout was done in an editing environment, it was necessary that the layout algorithm be incremental and non-disruptive.

We pushed the design of the representation into three-dimensions, but that work remains incomplete. One of the concrete results of this work is an approach to the layout of hyperedges in three-dimensions. We believe the 3D representation work presented in this thesis is a good foundation for further study.

Assessing desirability. In our analyses we evaluated this language to assess the evidence it provides for the hypothesis. The analyses were of three kinds: subjective, objective, and usability. These analyses provide only an initial step toward achieving the second major objective of the hypothesis. A great deal of work remains to be done to fully achieve this objective.

The subjective analysis comparing the SPARCL solutions to those in LISP and PROLOG shows the SPARCL solution to be arguably, but not undeniably, more understandable than the solutions in the other two languages. This (arguable) improved understandability is due to its visual representation of coreference, its specialized representation and handling of sets, and its comparative brevity. The SPARCL solutions of the other example problems provide additional evidence of SPARCL's brevity and understandability.

The objective analysis demonstrates that SPARCL is comparable to PROLOG and substantially more concise than LISP. Further, the analysis indicates that with certain further work on the SPARCL concrete representation, SPARCL can be improved to be substantially more concise than PROLOG. Based on our connection of size measurements with the quality of the programming language, this indicates that SPARCL may be a better programming language than PROLOG or LISP for some kinds of exploratory programming.

The usability testing indicated several things about the implementation. The SPARCL representation was thought to be good by the participants, judging by their

recorded comments. There was some contrary evidence on the understandability of SPARCL by the participants in that the final example program of the tutorial may not have been understood by any of the four participants who studied it. From this experience the SPARCL environment appears promising, but the construction and use of the complex terms needs careful and extensive explanation.

The analyses showed that the three approaches to programming and the novel set-handling technique work well together, as hypothesized. Each contributed to the reasonable brevity of SPARCL programs. The visual presentation of partitioned sets was successful in that they are easily understood. Intensional sets seem to be the hardest aspect of SPARCL for users to understand. We believe that their visual presentation does not particularly help in making them understandable, although another representation than that we chose might do so. The use of connecting lines for coreference (instead of “co-naming”) was preferred by the participants of the study.

The use of sets is problematic. They are a very abstract data organization that both require and support many other organizations. We addressed this issue in SPARCL by building in representations of sets-as-N-tuples and sets-as-tables. To make SPARCL easier to use, there should probably be a richer suite of built-in data organization alternatives.

SPARCL fully demonstrates the feasibility of visual logic programming with partitioned sets. The more difficult question of the desirability of such an approach to programming language design is not well established. SPARCL demonstrates that such a programming language can be as concise, or more so, as the symbolic programming languages PROLOG and LISP for selected nontrivial programming problems.

Appendix 1

A Tutorial Introduction to SPARCL

This appendix is a tutorial introduction to SPARCL, the language which is built using the principles of the hypothesis: it is semantically a logic programming language which is built around partitioned sets; it is representationally a two- and three-dimensional visual programming language; and it is implemented as an integrated development environment (IDE). The definitions of the syntax of SPARCL and the relationship of the various elements of SPARCL to the design principles of the hypothesis are discussed in chapter 3 ("Design Elements"). The tutorial material in this appendix is extracted from the interactive tutorial that is part of the SPARCL IDE, and which was the basis of the usability testing discussed in chapter 9 ("Usability Testing").

1.Introduction: Introduction to SPARCL

In this section we review the basic mechanisms of SPARCL through an example program. Although the treatment is informal many important concepts are introduced such as: SPARCL clauses, facts, rules and running queries. [Due to the underlying logic programming semantics of SPARCL, there are many similarities between an introduction to SPARCL and introductions to other logic programming languages. Thus, the contents and organization of this script are adaptations to SPARCL of the first chapter of the second edition of Ivan Bratko's "Prolog Programming for Artificial Intelligence", 2nd edition.]

1.1.Facts - the Parents program. In this subsection we create a program which describes the 'Parent'/2 relationship among seven people, then present several queries of this parent relationship.

In this subsection you are shown how to create a simple "database"-style program, 'Parent'/2, and how to query ("run") it.¹ SPARCL is a visual logic programming

1. Due to the underlying logic programming semantics of SPARCL, there are many similarities between an introduction to SPARCL and introductions to other logic programming languages. We adapted section 1.1 of the second edition of Ivan Bratko's "Prolog Programming for Artificial Intelligence" (a discussion of logic programming and PROLOG) to create this subsection of the

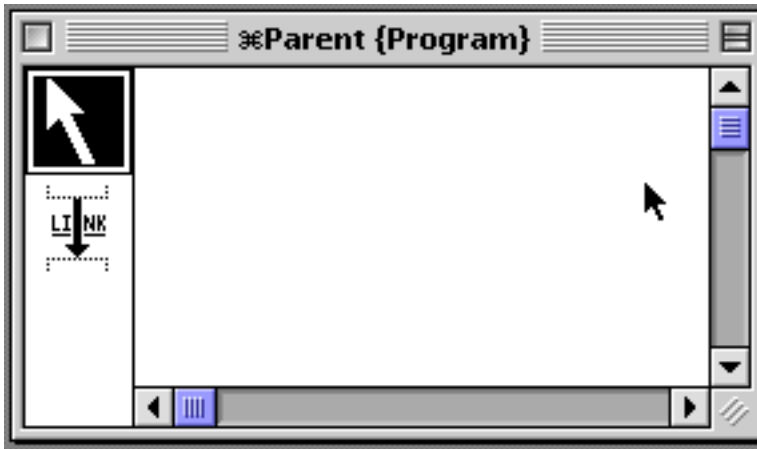


Figure A1-1. 1: The newly created "Parent" program window.

language based on sets with partitioning constraints. This subsection introduces some of the simple visual logic programming aspects of SPARCL.

First we define a simple SPARCL program.

1.1.1.Create the 'Parent'/2 program. In this subsection we create a program of six clauses. These clauses describe a 'Parent'/2 relationship among seven people.

A SPARCL program is represented by a window containing one or more clauses. An empty new program is created via the "New Program..." option of the "File " menu. After we create the program window we will populate it with clauses. The first step is described below, and the results of this step are shown in Figure A1-1. 1.

Step 1.

DO: Create a new program (and window) named "Parent".

BY: Select the "New Program..." option of the "File " menu, enter "Parent" in the program name dialog, and click "OK".

There are three "panes" in a program window (only two of which are visible in Figure A1-1. 1): the drawing pane, the tools pane, and the viewer pane. The drawing pane is the large area on the right side of the window. It has scroll bars on its right and bottom sides. The tools pane is on the left side of the window. There are two tool icons, the top one (an arrow) is the "general" tool and the other tool is the "connect" tool. The viewer pane is optionally visible, it is in the lower left corner of the window. The viewer is used to quickly position the drawing pane. It is left hidden (or "off") to speed up the display operations.

The general tool supports a wide variety of operations, each of which is invoked through a popup menu which is activated by pressing the mouse button while the

tutorial on logic programming and SPARCL.

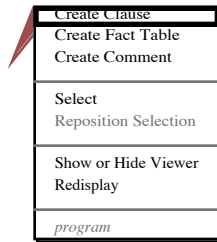


Figure A1-1. 2: Step 2.1 interaction to create a clause. (The small “flag” at the upper left corner of the clause is half of a “demonstration cursor” used by the tutorial system.)

cursor is over some "object" in the program window (the background of the program window is considered to be the "program" object). Each kind of object has a different popup

menu. The connect tool supports connecting two terms together. We discuss this more below.

Now we are ready create the clauses of the ‘Parent’/2 program. Step 2, which creates the first clause, has several sub-steps.



Figure A1-1. 5: Result of argument creation by step 2.2.1.



Figure A1-1. 3: New “Parent” clause resulting from Step 2.1

Step 2: Create a clause with arguments Pam and Bob in program Parent.

Step 2.1 creates an “empty” clause in the program window. The popup menu item selection is shown in Figure A1-1. 2. The result of the interaction described in step 2.1 is shown in Figure A1-1. 3.

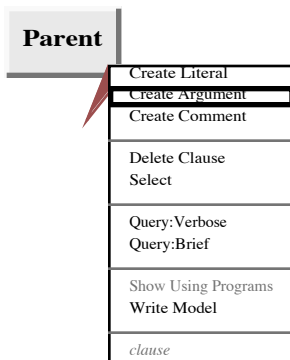


Figure A1-1. 4: Interaction to create an argument for step 2.2.1.

Step 2.1.
DO: Create a new clause in program "Parent".
BY: Select the "Create Clause" option in the popup menu for the program (window).

Now that we have the empty clause, we must add the two arguments positions. This is done in Step 2.2. This step uses two sub-steps.

Step 2.2: Add arguments to an existing object.

The next step creates the first argument position. The popup menu item selection is shown in Figure A1-1. 4. The result of the interaction described in step 2.2.1 is shown in Figure A1-1. 5.

Step 2.2.1.
DO: Create a new argument in the clause.
BY: Select the "Create Argument" option in the popup menu for clause.

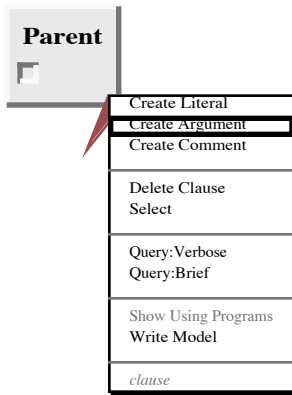


Figure A1-1. 6: Interaction to create an argument for step 2.2.2.

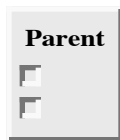


Figure A1-1. 7: Result of argument creation by step 2.2.2.

The next step creates the second argument position. The popup menu item selection is shown in Figure A1-1. 6. The result of the interaction described in step 2.2.2 is shown in Figure A1-1. 7.

Step 2.2.2.(repeat Step 2.2.1).

Having created two argument positions in the 'Parent'/2 clause, we now place the ur constant "Pam" in the first (upper) argument. This is done by step 2.3, which is done in three sub-steps.



Figure A1-1. 9: Result of ur constant creation by step 2.3.1.

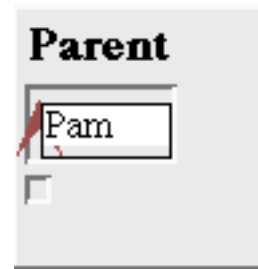


Figure A1-1. 10: Result of ur constant editing by step 2.3.2.

Step 2.3: Create a new ur constant of value Pam in the first argument of the clause.

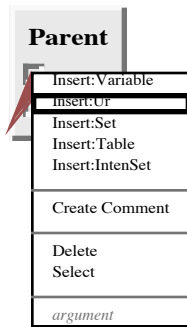


Figure A1-1. 8: Interaction to create a new ur constant for step 2.3.1.

Step 2.3.1.
DO: Create a new ur in the first argument of the clause.
BY: Select the "Insert:Ur" option in the popup menu for the argument.

The step 2.3.1 creates an ur constant in the first argument position with the default value of "new_ur", and leaves an edit item open for that new ur constant. The popup menu item selection is shown in Figure A1-1. 8. The result of the interaction described in step 2.3.1 is shown in Figure A1-1. 9.

Now we enter the text "Pam" into the edit item left open by the previous step. The result of this step is shown in Figure A1-1. 10.

Step 2.3.2.

DO: Edit the value of the ur constant of the "open" ur constant to "Pam".

BY: Enter the characters of the new value.

We have entered the text "Pam", now we close the edit item. The result of this

step is shown in Figure A1–1. 11.

Step 2.3.3.

DO: Close the current edit "box" for program "Parent".

BY: Click in the program window anywhere outside of the box.



Figure A1–1. 11
: Result of
"closing" ur
constant edit
item by step



Figure A1–1. 12
: Result of cre-
ating new ur
constant by
step 2.4.

The next step creates an ur constant “Bob” in the second (lower) argument. It uses the same three sub-steps as did step 2.3. The result of this step is shown in Figure A1–1. 12.

Step 2.4: Create a new ur constant of value Bob in the second argument of the clause.

Step 2.4.1.

DO: Create a new ur in the second argument of the clause.

BY: Select the "Insert:Ur" option in the popup menu for the argument.

Step 2.4.2.

DO: Edit the value of the “open” ur constant to "Bob".

BY: Enter the characters of the new value.

Step 2.4.3.

DO: Close the current edit "box" for program "Parent".

BY: Click in the program window anywhere outside of the box.

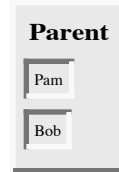
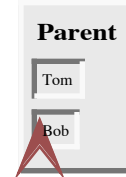


Figure A1–1. 13: Result of creating new clause by step 3.



Step 3 creates a second clause. It uses the same sort of sub-steps as used in creating the first clause, with the exception of the argument position creation steps. When there is just one predicate defined in a program window, as in this case, any “create clause” interaction creates a clause for that predicate with the appropriate number of argument positions. The result of the sub-steps of step 3 are shown in Figure A1–1. 13.

Step 3: Create a clause with arguments” Tom” and “Bob” in program Parent.

Step 3.1.

DO: Create a new clause in program "Parent". (Call this clause 2.)

BY: Select the "Create Clause" option in the popup menu for the program.

(Use existing arguments in an existing object.)

Step 3.2: Create a new ur constant of value “Tom” in the first argument of clause 2.

Step 3.2.1.

DO: Create a new ur in the first argument of clause 2.

BY: Select the "Insert:Ur" option in the popup menu for the argument.

Step 3.2.2.

DO: Edit the value of "open" ur constant to "Tom".

BY: Enter the characters of the new value.

Step 3.2.3.

DO: Close the current edit "box" for program "Parent".

BY: Click in the program window anywhere outside of the box.

Step 3.3: Create a new ur constant of value "Bob" in the second argument of clause 2.

(sub-steps similar to step 3.2)

The remainder of the six clauses are created in the next four steps. These steps achieve their goals by the same process as steps 2 and 3. The resulting set of clauses is shown in .

Step 4: Create a clause with arguments "Tom" and "Liz" in program Parent.

Step 5: Create a clause with arguments "Bob" and "Ann" in program Parent.

Step 6: Create a clause with arguments "Bob" and "Pat" in program Parent.

Step 7: Create a clause with arguments "Pat" and "Jim" in program Parent.

Now that we have defined the "Parent"/2 predicate (a predicate "Name/K" is defined by all of the clauses with the name "Name" and K of arguments), we can query it. A query is written in SPARCL as a clause, generally with one or more "literals" in it. A "literal" is a reference to a predicate.

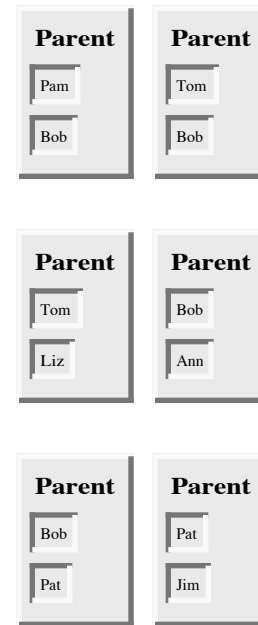


Figure A1-1. 14: Result of steps to create the six 'Parent'/2 clauses.

1.1.2: Querying the Parent program. In this section we create six clauses for querying the parent program and run these queries.

First we create the program which will contain the six query clauses.

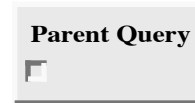


Figure A1-1. 15: Result of step 2.1 to create argument.

Step 1.

DO: Create a new program (and window) named "Parent Query".

BY: Select the "New Program..." option of the "File " menu, enter "Parent Query" in the program name dialog, and click "OK".

For our first example, we ask "Who is Pam's child?". This is done in two phases: first, create a clause which describes the query; second, use the "Query: Brief" option of the "Clause" popup menu to run the query. Step 2 creates the query using several sub-steps.

Step 2: Create a "Parent Query" clause for asking the question "Who is Pam's child?" (Call this clause 1)

Step 2.1 creates the "Parent Query" with one argument. The argument is created in the same fashion as shown in step 2.2.1 of section 1.1.1. The result of the creation is shown in Figure A1-1. 15.

Step 2.1.

DO: Create a new clause named "Parent Query" with 1 argument in program "Parent Query". (Call this clause 1)

BY: Select the "Create Clause" option in the popup menu for the program. Add 1 more argument to the default 0 arguments.

We add a comment to the clause to describe the question it asks in step 2.2. This is done in three sub-steps.

Step 2.2: Create a new comment "Who is Pam's child?" in clause 1.

The next step creates a comment for the clause with the default value "comment". The interaction is shown in Figure A1-1. 16, and the result is shown in Figure A1-1. 17.

Step 2.2.1.

DO: Create a new comment in clause 1.

BY: Select the "Create Comment" option in the popup menu for the clause.

Next the default comment value is replaced with the desired comment "Who is

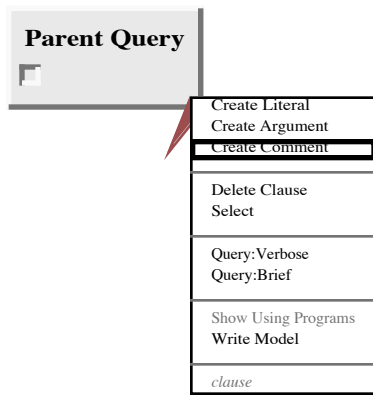


Figure A1-1. 16: Interaction for step 2.2.1 to create argument.

Pam's child?". The result is shown in Figure A1-1. 18.

- Step 2.2.2.
 DO: Edit the value of the "open" comment to "Who is Pam's child?".
 BY: Enter the characters of the new value.

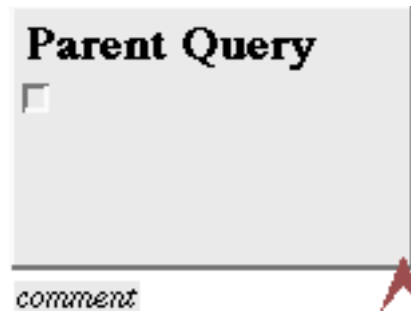


Figure A1-1. 17: Result of step 2.2.1 to create the comment.

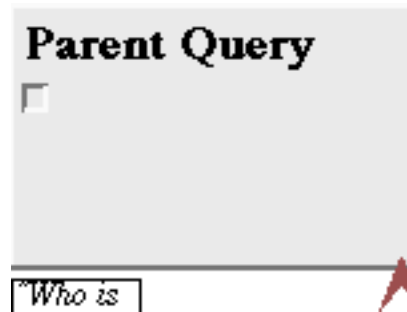


Figure A1-1. 18: Result of step 2.2.2 to set the value of the com-

The next step finishes the creation of the comment by closing the edit item for the comment.

- Step 2.2.3.
 DO: Close the current edit "box" for program "Parent Query".
 BY: Click in the program window anywhere outside of the box.

Next a variable is put in the argument of the clause. The interaction is shown in Figure A1-1. 19 and the result is shown in Figure A1-1. 20.

- Step 2.3.
 DO: Create a new variable in the argument of clause 1.
 BY: Select the "Insert:Variable" option in the popup menu for the argument.

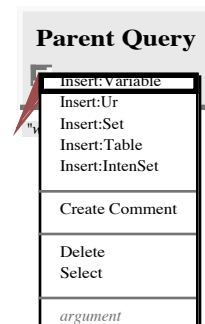


Figure A1-1. 19: Interaction for step 2.3 to create a variable.



Figure A1-1. 20: Result for step 2.3 to create a variable.

The next step creates a literal. The interaction is shown in Figure A1-1. 21, and the result is shown in Figure A1-1. 22.

- Step 2.4.
 DO: Create a new literal with name "Parent" and 2 arguments in clause 1.

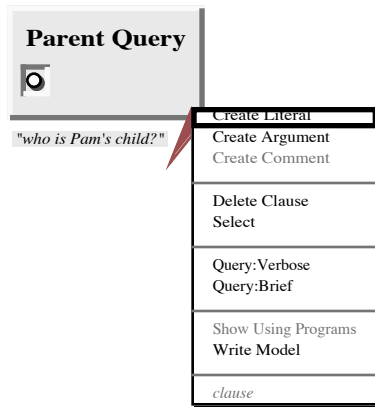


Figure A1-1. 21: Interaction for step 2.4 to create a literal.

Next we put the ur constant valued “Pam” in the first argument of the new literal. This is done by several sub-steps.

Step 2.5: Create a new ur constant of value “Pam” in the argument of the literal of clause 1.

The sub-steps for creating an ur constant value “Pam” start with step 2.5.1. These steps are similar to the ur constant creation steps presented above. The initial interaction is shown in Figure A1-1. 23 and the final result is in Figure A1-1. 24.

Step 2.5.1.

DO: Create a new ur in argument 1 of the literal of clause 1.

BY: Select the "Insert:Ur" option in the popup menu for the argument.

Step 2.5.2.

DO: Edit the value of the “open” ur constant to “Pam”.

BY: Enter the characters of the new value.

Step 2.5.3.

DO: Close the current edit "box" for program "Parent Query".

BY: Click in the program window anywhere outside of the box.

The next step creates a variable in the second argument of the literal, similarly to step 2.3.

Step 2.6.

DO: Create a new variable in the argument 2 of the literal of clause 1.

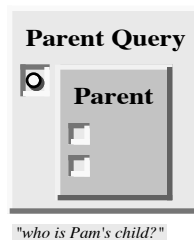


Figure A1-1. 22: Result for step 2.4 to create a literal.

BY: Select the "Create Literal" option in the popup menu for the clause.

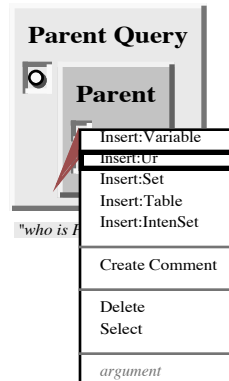


Figure A1-1. 23: Interaction for step 2.5.1 to create an ur constant.

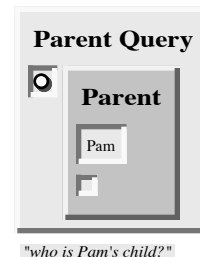


Figure A1-1. 24: Result for step 2.5.3 to close the edit item for the new ur constant.

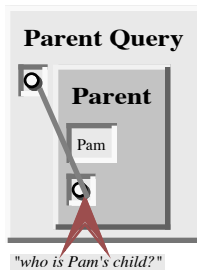


Figure A1-1. 25: Interaction for step 2.7 to link two variables.

BY: Select the "Insert:Variable" option in the popup menu for the argument.

Now we link together the two variables. The interaction is shown in Figure A1-1. 25. The state of the interaction pictured is after doing a click-and-drag from the variable in the argument of the clause to the vari-

able in the second argument of the literal, but before the mouse button is released. The “demonstration cursor” shows where the user’s cursor would be at this point and the grey line shows where the link will be placed. The result is shown in Figure A1-1. 26.

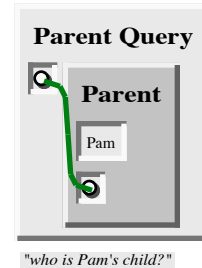


Figure A1-1. 26: Result for step 2.7 to link two variables.

Step 2.7.

DO: Create a coreference link including the variable in the argument of clause 1 and the variable in argument 2 of the literal of clause 1.

BY: With "connect tool" as the current tool, depress the mouse button while the cursor is over one of the variables and drag the cursor until it's over the other variable, then release the mouse button.

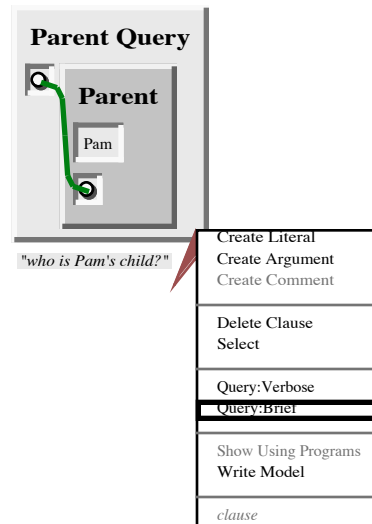


Figure A1-1. 27: Interaction for step 3 to query SPARCL.

This clause has our first uses of SPARCL variables. One is in the argument of the "Parent Query" clause; the other is in the second argument to the 'Parent'/2 literal. These two variables are linked to show that they must refer to the same thing. In a traditional linear language this would be indicated by having two variable instances with the same name.

Now that we have built a “query” clause, we are ready to evaluate it. The next step does this evaluation. The interaction is shown in Figure A1-1. 27 and the result is shown in Figure A1-1. 28.

Parent Query → Bob

Figure A1-1. 28: Result for step 3 to query SPARCL.

Step 3.

DO: Execute a query of clause 1, with tracing information suppressed.

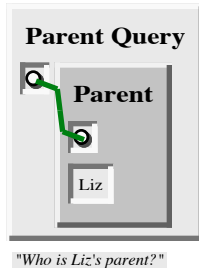


Figure A1-1. 29:
Result for step 4 to
create the “Who is
Liz’s parent?” query

Parent Query → Tom

Figure A1-1. 30:
Result of step 5 pos-
ing the “Who is Liz’s
parent?” clause as a
query.

BY: Select the "Query: Brief" option in the popup menu for the clause.

This query creates output in two windows. It writes a "timing" message in the "*Output*" window telling

how long the query took to be evaluated (this time does not include time spent rendering the result). The query, when successful, also adds an "N-tuple" to the "Parent Query

RESULT" window which is the query clause "head" as it is after the successful query evaluation. In this example, the result in Figure A1-1. 28 shows that Bob is Pam's child.

For our second example, we ask "Who is Liz's parent?". This is done in the same two phases as were used before. The sub-steps used by step 4 to create the query clause are similar to those of step 2 above. The resulting clause is shown in Figure A1-1. 29.

Step 4: Create a "Parent Query" clause for asking the question "Who is Liz's parent?"

Step 5.

DO: Execute a query of clause with ID Parent Query:13, with tracing information suppressed.

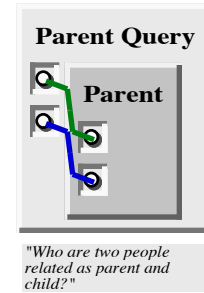
BY: Select the "Query: Brief" option in the popup menu for clause with ID Parent Query:13.

As a result of the preceding query, SPARCL tells us that Tom is a parent of Liz, as shown in Figure A1-1. 30.

For our third example, we ask "Who are two people related as parent and child?". The clause for this question and the result of posing this clause as a query are shown in Figure A1-1. 31.

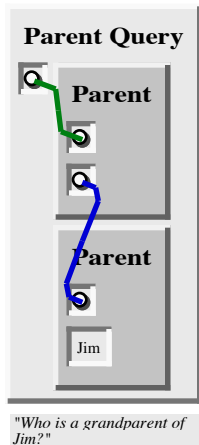
The next query asks for Jim's grandparent "X". Since our program does not directly know the "Grandparent" relation, this query has to be broken down into two parts:

- 1) Who is a parent of Jim? Assume that this is some Y.
- 2) Who is a parent of Y? Assume that this is some X.



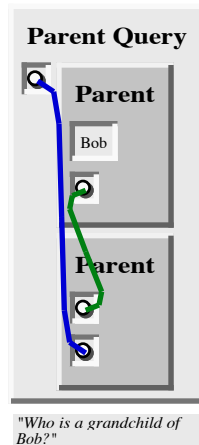
Parent Query → Pat → Jim

Figure A1-1. 31: Clause for
“Who are two people
related as parent and
child?” and the result of
posing this clause as a
query.



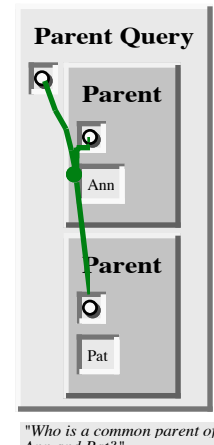
Parent Query → Bob

Figure A1-1. 32: Clause for “Who is the grandparent of Jim?” and the result of posing this clause as a query.



Parent Query → Jim

Figure A1-1. 33: Clause for “Who is a grandchild of Bob?” and the result of posing this clause as a query.



Parent Query → Bob

Figure A1-1. 34: Clause for “Who is a common parent of Ann and Pat?” and the result of posing this clause as a query.

We can ask a query of two parts by putting a literal for each part in the body of the same query clause and connecting the variables of these literals appropriately. The clause for this question and the result of querying SPARCL with it are shown in Figure A1-1. 32. This result shows us that Bob is Jim's grandparent.

The previous query can be "turned around" and we can ask "Who is a grandchild of Bob?". The clause for this question and the result of querying SPARCL are shown in Figure A1-1. 33. This shows us that Jim is a grandchild of Bob.

Another question could be "Who is a common parent of Ann and Pat?". As before, this query has to be broken down into two parts:

- 1) Who is a parent, X, of Ann?
- 2) Is (this same) X a parent of Pat?

The clause for this question and the result of querying SPARCL are shown in Figure A1-1. 34. The result shows us that Bob is a common parent of Ann and Pat.

This section has presented several points:

- It is easy in SPARCL to define a relation, such as the ‘Parent’/2 relation, by stating the N-tuples of objects that satisfy the relation.
- The user can easily query the SPARCL system about relations defined in the

program.

- A SPARCL program consists of 'clauses'.
- The arguments of relations can (among other things) be: concrete objects (such as "Tom" and "Ann"), or general objects which are represented by small circles. Objects of the first kind are called "atoms" and objects of the second kind are called "variables".
- Questions to the system consist of a clause, which may contain any number of 'literals'. Several literals in the body of a single clause means that clause is true when the conjunction of the literals is true.
- An answer can be either positive or negative: for a positive answer we say that the query was "satisfiable" and it "succeeded"; for a negative answer we say that the query was "unsatisfiable" and it "failed".
- If a query has several possible answers then SPARCL will find one of them.

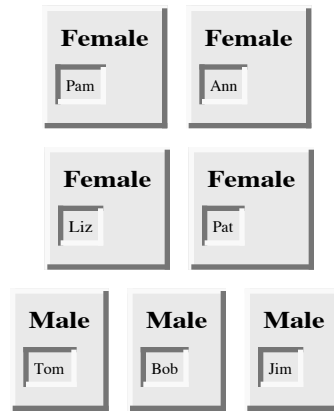


Figure A1-1.35: "Sex" program, separate clauses ver-

1. Exercise for section 1.1:

1. Formulate in SPARCL the following questions about the 'Parent'/2 relation (your question formulation should be "query" clauses with names of your choosing (e.g. "Query A", "Query B", "Query C")):
 1. Who is Pat's parent?
 2. Does Liz have a child?
 3. Who is Pat's grandparent?

1.2: Rules - extending the Parents program. The "Parents" program can be extended in many interesting ways. Let us first add the information on the sex of the people that occur in the 'Parent'/2 relation. This information has already been entered and saved in a program file, so we can simply open that program.

The relations introduced in Figure A1-1.35 are 'Male'/1 and 'Female'/1. These relations are unary (or one-place) relations. A binary relation like 'Parent'/2 defines a relation between *pairs* of objects; on the other hand, unary relations can be used to declare simple yes/no properties of objects.

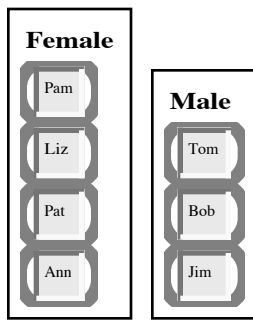


Figure A1-1. 36:
"Sex" program, fact
table version.

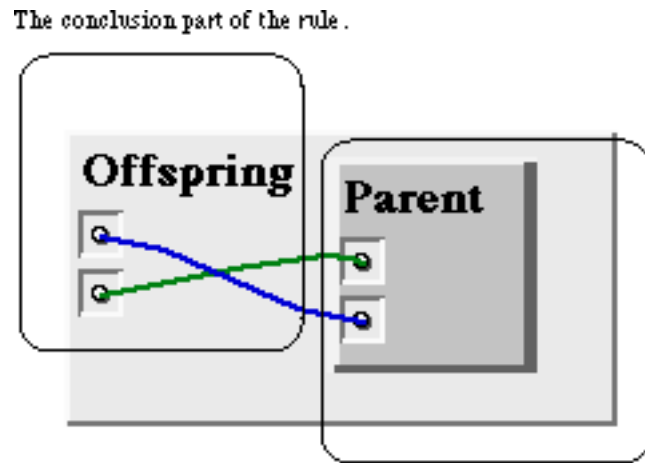


Figure A1-1. 37: Offspring clause, with labeled parts.

These 'Female'/1 and 'Male'/1 relations can be displayed in a different, more compact, way. This alternative representation uses SPARCL's "fact tables". Figure A1-1. 36 has "fact table" versions of the 'Female'/1 and 'Male'/1 relations. These tables have the same meaning as the collection of shown the in Figure A1-1. 35. The fact table has the predicate name for all of the facts of the table placed in the upper left-hand corner of the table. Each row of the table is a single "fact" - a clause with an empty "body".

As our next extension to the program let us introduce the 'Offspring'/2 relation as the inverse of the 'Parent'/2 relation. We could define 'Offspring'/2 in a similar way as the 'Parent'/2 relation; that is, by simply providing a list of simple facts about the 'Offspring'/2 relation, each fact mentioning one pair of people such that one is an offspring of the other.

However, the offspring relation can be defined much more elegantly making use of the fact that it is the inverse of 'Parent'/2, and that 'Parent'/2 has already been defined. This alternative way can be based on the following logical statement: "For all X and Y, Y is an offspring of X if X is a parent of Y." The clause in Figure A1-1. 37 represents the preceding "logical statement". This kind of clause is called a "rule".

There is an important difference between facts and rules. A fact like those shown in the 'Parent'/2 clauses is something that is always, unconditionally, true. On the other hand, rules specify things that are true if some condition is satisfied. Therefore we say rules have:

- a condition part (the right-hand side of the clause)

- a conclusion part (the predicate name and arguments on the left-hand side of the clause).

The conclusion part is also called the "head" of a clause and the condition part the "body" of the clause.

How rules are actually used by SPARCL is illustrated by the following example. Let us ask our program whether Liz is an offspring of Tom. The "query" clause in Figure A1-1. 38 represents our question.

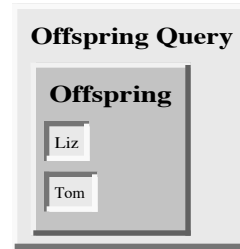


Figure A1-1. 38: Clause for posing the query "Is Liz an offspring of Tom?"

There is no fact about offsprings in the program, therefore the only way to consider this question is to apply the rule about offsprings. The rule is general in the sense that it is applicable to any two objects; therefore it can also be applied to such particular objects as "Liz" and "Tom". We say that the variables become instantiated.

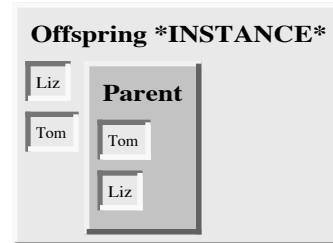


Figure A1-1. 39: Instantiation of Offspring rule clause. Created in response to the query literal of Figure A1-1. 38.

Figure A1-1. 39 is the special case of the Offspring rule clause after instantiation of the general rule in Figure A1-1. 37 in satisfying the query literal from Figure A1-1. 38.

The single literal of the "Offspring *INSTANCE*" body becomes the new goal for SPARCL to solve. It is trivial to solve as it can be found as a fact in the 'Parent'/2 program. This means that the conclusion part of the rule is also true, and SPARCL will succeed in executing the original query.

In Figure A1-1. 40 there are two similar clauses defining two different predicates. The "Mother" clause, which defines the 'Mother'/2 predicate, shows a rule with two literals in its body, 'Parent'/2 and 'Female'/2. The "Grandparent" clause, which defines the 'Grandparent'/2 predicate, shows a rule which uses the same relation, 'Parent'/2, twice in the literals of its body.

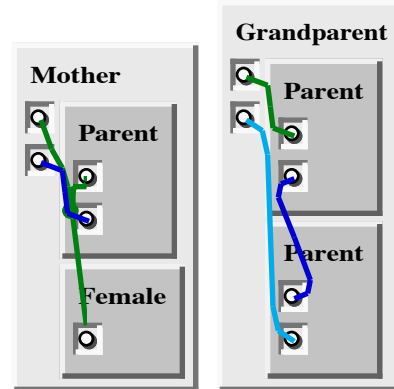


Figure A1-1. 40: The clauses defining the 'Mother'/2 and 'Grandparent'/2 predicates.

The "Sister (sort of)" clause in Figure A1-1. 41 defines the relationship of someone

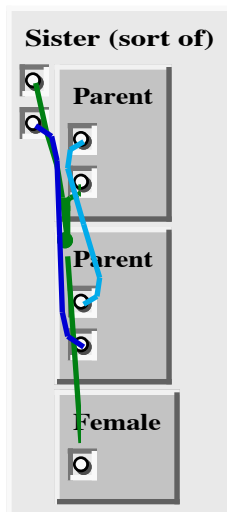


Figure A1-1. 41: Clause defining the 'Sister (sort of)'/2

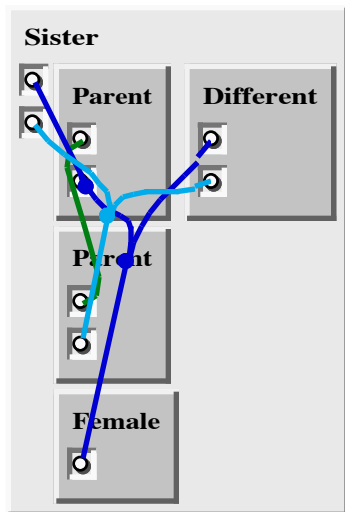


Figure A1-1. 42: Clause defining 'Sister'/2 predicate.

as the sister of someone if these two people have the same parent. This clause actually is slightly flawed - it allows for someone to be sister to herself. The "Sister" clause in Figure A1-1. 42 fixes this using the 'Different'/2 relation.

The 'Different'/2 predicate, shown in Figure A1-1. 43, relies on a partitioned set constraint to specify the the terms in its two arguments are different. The '*TERM*/1

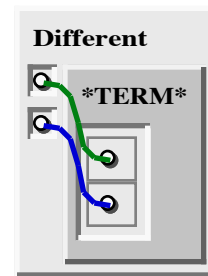


Figure A1-1. 43: Clause defining the 'Different'/2 predicate.

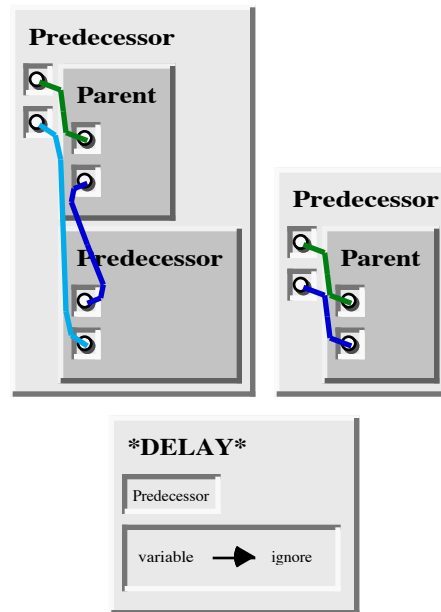


Figure A1-1. 44: Clauses defining the not-quite-satisfactory version of the 'Predecessor'/2 predicate.

predicate is a built-in predicate of the SPARCL system. It is always true. It is used here to introduce the partitioned set constraint. Since the two "sisters" are in different parts of a partition, they must be different (since parts of a partition are disjoint sets). This is a kind of "not equal" constraint. We will discuss partitioned sets in more detail in a later section.

The 'Predecessor'/2 predicate is defined by the two 'Predecessor'/2 clauses and a '*DELAY*/2 clause, as shown in Figure A1-1. 44. This predicate is an example of *recursion*, multiple clauses to defining a single predicate, and specifying a *delay* condition. A recursive definition of a predicate uses the predicate being defined in the body of one or more of the defining clauses of that predicate. Some person X is the

Predecessor of some other person Z if X is a Parent of Z (this is one of the clauses), OR if X is the parent of some person Y, and Y is a predecessor of Z (this is the other one of the clauses).

The ‘*DELAY*/2 predicate is used by the SPARCL interpreter in deciding when to evaluate goal literals. This ‘*DELAY*/2 clause instructs the interpreter to “delay” the evaluation of a ‘Predecessor’/2 goal if the first argument is a variable (i.e. an unbound variable term). This is necessary in the case of the ‘Predecessor’/2 predicate to prevent the interpreter from trying to

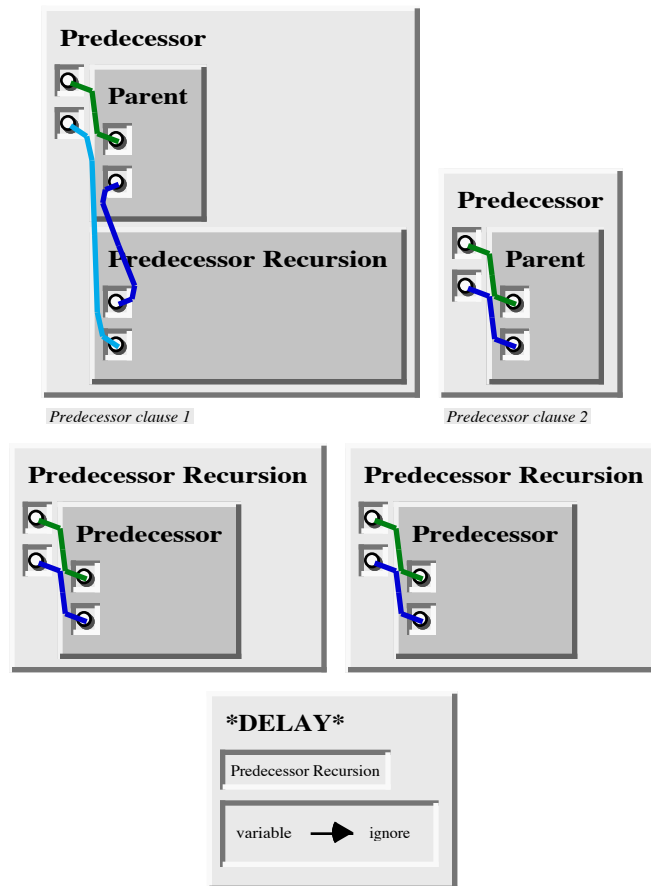


Figure A1–1. 45: Clauses defining the correct version of the ‘Predecessor’/2 predicate.

solve the ‘Predecessor’/2 literal before it has solved the ‘Parent’/2 literal. This prevents the interpreter from recursing forever. Unfortunately, this is too restrictive. This restricts us to using the ‘Predecessor’/2 predicate to ask the question "Who is X the predecessor of?". We would like to also be able to use ‘Predecessor’/2 to answer the question "Who is X's predecessor?".

Another version of the ‘Predecessor’/2 program is shown in Figure A1–1. 45 that allows us to ask both of these questions. This version of the ‘Predecessor’/2 program allows us to ask both of the questions mentioned before. The difference between this version and the earlier version small: There is a new predicate, "Predecessor Recursion" which is simply a "wrapper" for ‘Predecessor’/2; this predicate is used in the recursive clause of ‘Predecessor’/2; and the ‘*DELAY*/2 clause now refers to this new predicate ‘Predecessor Recursion’/2 instead of the ‘Predecessor’/2 predicate. Now, the interpreter will attempt to solve a ‘Predecessor’/2 goal which has an

unbound variable first argument (which would have been delayed in the definition in Figure A1–1. 44), but it will not recurse infinitely in the attempt since the ‘Predecessor Recursion’/2 literal will be delayed when its first argument is an unbound variable.

The ‘*DELAY*’/2 clauses are the primary method the SPARCL programmer has to control the order in which the SPARCL interpreter attempts to solve goals. This is a particularly important facility when writing recursive predicates to ensure that the recursion terminates.

Important points of this section are:

- SPARCL programs can be extended by simply adding new clauses.
- SPARCL clauses are of three types: facts, rules, and questions.
- Facts declare things that are always, unconditionally true.
- Rules declare things that are true depending on a given condition.
- By means of questions the user can ask the program what things are true.
- SPARCL clauses consist of the "head" and the "body". The head is the name of the predicate and the arguments placed on the left side of the clause. The body is a set of "literals". These literals are understood to be joined by conjunctions.
- Facts are clauses that have a head and the empty body. Rules have the head and the (non-empty) body. A question is a "rule" clause which the programmer chooses to query.
- In the course of computation, a variable can be substituted by another object. We say that a variable becomes "instantiated".
- Variables are assumed to be universally quantified and are read as "for all". Alternative readings are, however, possible for variables that appear only in the body. These can be read as "some" (existential) variables.
- Recursion may be used in defining SPARCL predicates.
- The "*DELAY*" clause may be used to control the order in which the SPARCL interpreter attempts to solve goals.

1. Exercise for section 1.2

1. Show translations for the following statements:

1. Everybody who has a child is happy (introduce a one-argument relation "Happy").
2. For all X, if X has a child who has a sister then X has two children

(introduce a new relation "Has Two Children").

2. Define the relation "Grandchild" using the 'Parent'/2 relation. Hint: It will be similar to the "Grandparent" relation.
3. Define the relation "Aunt" of two arguments in terms of the relations 'Parent'/2 and "Sister".),

1.3-How SPARCL works. This section gives an informal explanation of *how* SPARCL answers questions.²

A question to SPARCL is always a set of one or more goal literals. To answer a question, SPARCL tries to satisfy all of the goals. What does it mean to *satisfy* a goal? To satisfy a goal means to demonstrate that the goal is true, assuming that the relations in the program are true. In other words, to satisfy a goal means to demonstrate that the goal *logically follows* from the facts and rules in the program. If the question contains variables, SPARCL also has to find what are the particular objects (in place of variables) for which the goals are satisfied. The particular instantiation of variables to these objects is displayed to the user. If SPARCL cannot demonstrate for some instantiation of variables that the goals logically follow from the program, then SPARCL's answer to the question will be "no".

An appropriate view of the interpretation of a SPARCL program in mathematical terms is then as follows: SPARCL accepts facts and rules as a set of axioms, and the user's question as a *conjectured theorem*; then it tries to prove this theorem—that is, to demonstrate that it can be logically derived from the axioms.

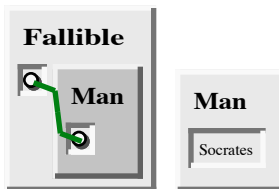


Figure A1-1. 46:
Clauses defining the
'Fallible'/1 and 'Man'/1

We will illustrate this view by a classical example. Let the axioms be:

All men are fallible.

Socrates is a man.

A theorem that logically follows from these two axioms is:

Socrates is fallible.

The first axiom above can be rewritten as:

For all X, if X is a man the X is fallible.

The example can be translated into SPARCL as shown in Figure A1-1. 46.

Now we ask SPARCL the question of Socrates' fallibility by querying the 'Fallible Socrates Query'/0 clause in Figure A1-1. 47, which succeeds.

2. The text of this section is adapted for sparcl from section 1.4 of Ivan Bratko's "Prolog Programming for Artificial Intelligence", 2nd edition.

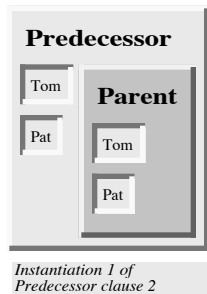


Figure A1–1. 49:
First instantiation
of ‘Predecessor’/2
clause 2.

To discuss how SPARCL works, we the concept of a *proof sequence*. Given some program (a set of clauses), a sequence of facts can be constructed starting with any *fact* in the program, then successively adding other facts from the program or any fact derived using a *rule* of the pro-

gram and any facts already in the sequence. A goal (or ‘literal’) is *satisfied* if such a sequence of facts can be found which ends with that goal. Let us call such a sequence of facts a *proof sequence*. SPARCL finds an appropriate proof sequence to satisfy a query.

SPARCL searches for a proof sequence satisfying a given goal by starting with that goal and working “backward” to facts of the program. Instead of starting with simple facts given in the program, SPARCL starts with the given goals and, using rules, substitutes the current goals with new goals, until new goals happen to be simple facts (or unify with program facts). Let's look at an example using ‘Predecessor’/2 and ‘Parent’/2 programs.

How does SPARCL “solve” the query asking if Tom is Pat's predecessor? We start with the initial query shown in Figure A1–1. 48. This initial query clause provides a single goal literal, “Predecessor(Tom, Pat)”. There are two rule clauses which have heads (consequents) which unify with this goal literal. These rules are labeled "Predecessor clause 1" and "Predecessor clause 2". SPARCL may try either of these clauses first—let's suppose SPARCL tries the rule of "Predecessor clause 2". It unifies the goal literal with the head of the rule, which binds the variables in the head. Since these variables corefer with variables in the body (antecedent) of the rule, these corefering variables are also bound. This instantiates the rule clause as shown in Figure A1–1. 49.

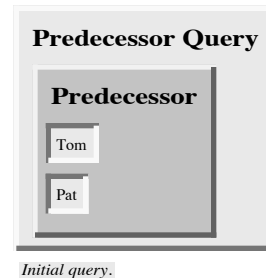


Figure A1–1. 48: Clause
defining initial query for
“Is Tom Pat’s predeces-
sor?”



Figure A1–1. 47: Clause pos-
ing the question “Is
Socrates fallible?”

For this instantiation of 'Predecessor'/2 clause 2 to be applicable in building a proof sequence, the literal in its body must precede the application of this rule in the proof sequence. Thus, SPARCL determines that it must find a proof sequence for this literal. This yields the "query 2" version of the initial query shown in Figure A1-1. 50. The original goal of "Is Tom Pat's predecessor?" has been replaced by the goal of "Is Tom Pat's parent?".

There is no clause (fact or rule) with a head that matches the 'Parent'/2 literal in the body of query 2 in Figure A1-1. 50, so SPARCL fails to solve this goal and it *backtracks* to try an alternative way to derive the top goal ("Is Tom Pat's predecessor?"). There were originally two ways to solve this top goal, and having failed to solve it using the rule "Predecessor clause 2" SPARCL now tries rule "Predecessor clause 1". As was done when using "Predecessor clause 2", SPARCL unifies the goal literal with the head of the rule ("Predecessor clause 1"), which binds the variables in the head. Again, since these variables corefer with variables in the body (antecedent) of the rule, these corefering variables are also bound. This instantiates the rule clause as shown in Figure A1-1. 51.

Substituting the body of instantiation 1 of "Predecessor clause 1" for the literal of the original query yields the new version of the query (query 3) shown in . The goal literals, in query 3, share an uninstantiated variable.

SPARCL must now solve the two literals in the body of this revised query. SPARCL is free to satisfy them in any order. Suppose SPARCL tries the 'Predecessor Recursion'/2 literal first. It checks the *DELAY* clauses and finds that 'Predecessor Recursion'/2 is to be delayed if the first argument is an unbound vari-

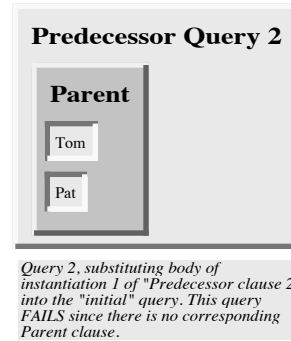


Figure A1-1. 50: Revised version of 'Predecessor Query'/0 (Figure A1-1. 48) based on substituting the body of instantiation 1 of 'Predecessor'/2 clause 2 (Figure A1-1. 49) for the literal in the body of the original 'Predecessor Query'/0.

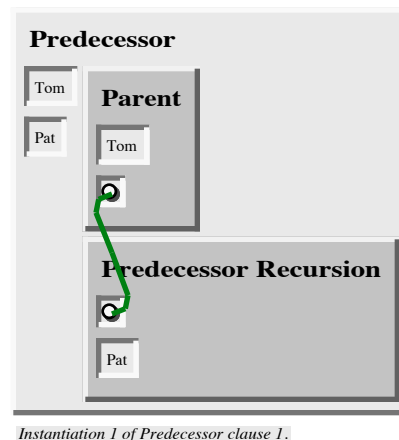
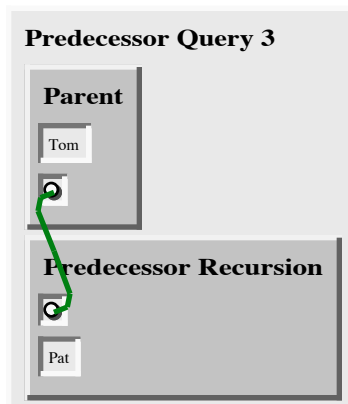


Figure A1-1. 51: Second instantiation of 'Predecessor'/2 clause



Query 3, substituting body of instantiation 1 of "Predecessor clause 1" into the "initial" query.

Figure A1-1. 52: Revised version of 'Predecessor Query' /0 (Figure A1-1. 48) based on substituting the body of instantiation 1 of 'Predecessor' /2 clause 1 (Figure A1-1. 51) for the literal in the body of the original 'Predecessor Query' /0.

er al "Tom is Bob's parent" from query 3 leaves us with query 4 (Figure A1-1. 53).

SPARCL solves the literal of query 4 by instantiating 'Predecessor Recursion' /2 as shown in

Figure A1-1. 54. It had "delayed" attempting the solution of this literal earlier because the goal had a variable first argument and there is a *DELAY* clause which specifies this situation as requiring such a goal to be delayed. However, the goal literal no longer has a variable first argument, it is now "Bob". So, SPARCL need not delay solving this literal.

Query 5 in Figure A1-1. 55 is created by substituting the instantiation of 'Predecessor Recursion' /2 in Figure A1-1. 54 into query 4 in Figure A1-1. 53.

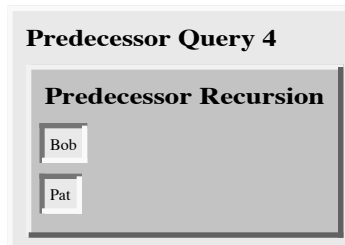
SPARCL solves the literal of query 5 by instantiating "Predecessor clause 2" in Figure A1-1. 56, similar to the first attempt at solving the initial query in Figure A1-1. 48.

able. This is the case in query 3, so SPARCL delays solving the 'Predecessor

Recursion' /2 literal and tries the 'Parent' /2 literal. This one is easily satisfied by

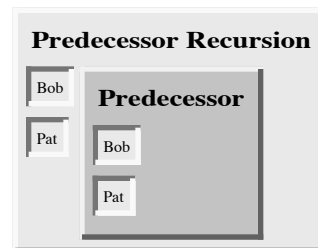
matching the fact that "Tom is Bob's parent." This

matching binds the shared variable to be "Bob". Removing the "solved" lit-



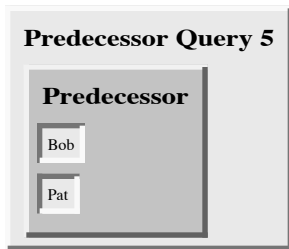
Query 4 is created by removing the solved goal literal for "Parent" from query 3.

Figure A1-1. 53: Revised version of 'Predecessor Query 3' /0 (Figure A1-1. 52) based on removing the solved literal and replacing the remaining variable with its binding ("Bob").



Instantiation 1 of Predecessor Recursion clause.

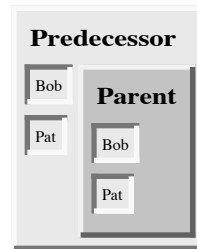
Figure A1-1. 54: Instantiation 1 of 'Predecessor Recursion' /2.



Query 5 is created by substituting the body of instantiation 1 of Predecessor Recursion clause 1 into the body of query 4.

Figure A1-1. 55: Revised version of 'Predecessor Query 4'/0 (Figure A1-1. 53) based on substituting the body of instantiation 1 of 'Predecessor Recursion'/2 clause (Figure A1-1. 54) for the literal in the body of 'Predecessor Query 4'/0.

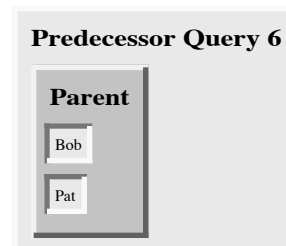
Query 6 in Figure A1-1. 57 is created by substituting instantiation 2 of 'Predecessor clause 2'/0 in Figure A1-1. 56 into query 5. The 'Parent'/2 literal of query 6 matches a 'Parent'/2 fact, and thus is true.



Instantiation 2 of Predecessor clause 2

Figure A1-1. 56: Second instantiation of 'Predecessor'/2

This completes the search for a proof sequence. The sequence being: the literal of query 6 and instance 2 of Predecessor clause 2 derives the literal of



Query 6 is created by substituting the body of instantiation 2 of Predecessor clause 1 into the body of query 5.

Figure A1-1. 57: Revised version of 'Predecessor Query 5'/0 (Figure A1-1. 55) based on substituting the body of instantiation 2 of 'Predecessor'/2 clause (Figure A1-1. 56) for the literal in the body of

query 5; the literal of query 5 and instance 1 of "Predecessor Recursion" derives the literal of query 4; the literal of query 4 and fact "Tom is Bob's parent" derive the literals of query 3; the literals of query 3 and instance 1 of "Predecessor clause 1" derive the the literal of initial query.

The trace of the execution of SPARCL can be pictured as a "tree". The nodes of the tree correspond to goal literals, or lists of goal literals, that are to be satisfied. The arcs between the nodes correspond to the application of (alternative) program clauses that transform the goals at one node into the goals at another node. The top goal is satisfied when a path is found from the root node (top goal) to a leaf node labelled "success". A leaf is labelled "success" if it is a simple fact. The execution of SPARCL programs is the searching for such paths. During the search SPARCL may enter an unsuccessful branch. When SPARCL discovers that a branch fails it automatically backtracks to the previous node and tries to apply an alternative clause at that node.

1.4-What SPARCL programs *mean*. This section discusses ways to think about the meaning of SPARCL programs.³ In the examples so far it has always been possible to understand the results of the program without exactly knowing *how* the system actually found the results. It therefore makes sense to distinguish between two levels of meaning of SPARCL programs; namely,

- the *declarative meaning* and
- the *procedural meaning*.

The declarative meaning is concerned only with the *relations* defined by the program. The declarative meaning thus determines *what* will be the output of the program. On the other hand, the procedural meaning also determines *how* this output is obtained; that is, how are the relations actually evaluated by the SPARCL system.

The ability of SPARCL to work out many procedural details on its own is considered to be one of its specific advantages. It encourages the programmer to consider the declarative meaning of programs relatively independently of their procedural meaning. Since the results of the program are, in principle, determined by its declarative meaning, this should be (in principle) sufficient for writing programs. This is of practical importance because the declarative aspects of programs are usually easier to understand than the procedural details. To take full advantage of this, the programmer should concentrate mainly on the declarative meaning and, whenever possible, avoid being distracted by the executional details. These should be left to the greatest possible extent to the SPARCL system itself.

This declarative approach indeed often makes programming in SPARCL easier than in typical procedurally oriented programming languages such as Pascal. Unfortunately, however, the declarative approach is not always sufficient. It will later become clear that, especially in large programs, the procedural aspects cannot be completely ignored by the programmer for practical reasons of executional efficiency. Nevertheless, the declarative style of thinking about SPARCL programs should be encouraged and the procedural aspects ignored to the extent that is permitted by practical constraints.

3. The text of this section is largely drawn from section 1.5 of Ivan Bratko's "Prolog Programming for Artificial Intelligence", 2nd edition.

Summary:

- SPARCL programming consists of defining relations and querying about relations.
- A program consists of *clauses*. These are of two types: *facts* and *rules*. A clause can be used to ask a *question*.
- A relation can be specified by *facts*, simply stating the N-tuples of objects that satisfy the relation, or by stating *rules* about the relation.
- A *procedure* (also called a “predicate”) is a set of clauses about the same relation.
- Querying about relations, by means of *questions*, resembles querying a database. SPARCL's answer to a question consists of a set of objects that satisfy the question.
- In SPARCL, to establish whether an object satisfies a query is often a complicated process that involves logical inference, exploring among alternatives and possibly *backtracking*. All this is done automatically by the SPARCL system and is, in principle, hidden from the user.
- Two types of meaning of SPARCL programs are distinguished: declarative and procedural. The declarative view is advantageous from the programming point of view. Nevertheless, the procedural details often have to be considered by the programmer as well.
- The following concepts have been introduced in this section: clause, fact, rule, question; the head of a clause, the body of a clause; recursive rule, recursive definition; procedure; constant, variable; instantiation of a variable; goal (literal); goal is satisfiable, goal succeeds; goal is unsatisfiable, goal fails; backtracking; declarative meaning, procedural meaning.

2: SPARCL representation and meaning.

This section gives a systematic treatment of the representation and meaning of basic concepts of SPARCL, and introduces sets.

The topics included are:

- simple data objects (constants and variables)
- sets
- matching as the fundamental operation on objects
- declarative (or nonprocedural) meaning of a program
- procedural meaning of a program
- relation between the declarative and procedural meanings of a program, and
- altering the procedural meaning by "delay" specifications.

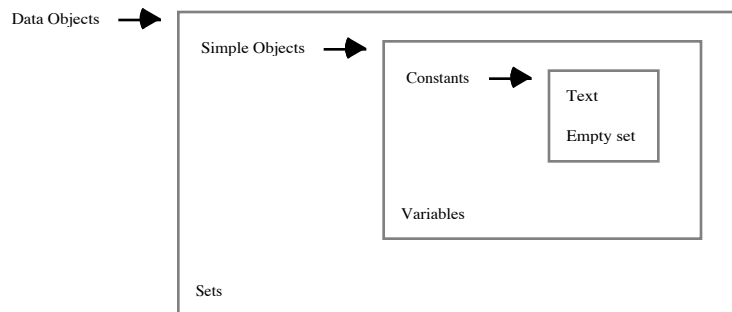


Figure A1-2. 1: Classification of data objects.

Object Type	Example Object
Text Constant	Some text
Empty set	■
Variable	○
Set	

Figure A1-2. 2: Examples of some types of data objects (terms).

Most of these topics have already been reviewed in section 1. Here the treatment is more formal and detailed.

2.1: Data objects section

2.1.1: Term types discussion. Figure A1-2. 1 shows a classification of data objects in SPARCL. The SPARCL system reflects the type of an object in its representation. There is a different kind of visual representation for each type of object. Figure A1-2. 2 shows some examples of these data object (term) types.

We have already seen representations for text constants and variables in Chapter 1: text constants are represented by text, variables are represented by small circles.

This is shown in the first and third entries in Figure A1–2. 2. There are no explicit variable data type declarations in SPARCL programs. In SPARCL, text constants are used to represent numeric and non-numeric data. SPARCL interprets a text constant as a number if a number-handling built-in predicate is being evaluated.

The second entry in Figure A1–2. 2 shows the representation of an empty set. An empty set constant represents a set with no members.

Variables in SPARCL do not have names. Each small circle represents a distinct variable. If two variables are intended to refer to the same thing, then they are connected by a coreference link. Coreference links may connect any number of SPARCL terms (data objects). Most commonly they connect two variables. The terms connected by a single coreference link are all the same - they must unify with each other (unification is a special kind of matching). The terms joined by a coreference link must all be in the same clause. You have already seen some uses of coreference links in the example queries of Chapter 1.

Sets are objects which contain any number of objects. The contained objects can, in turn, be sets. There are several representations of sets in SPARCL. Each representation is associated with a particular special set organization. We have already seen most of these set representations. They have been used to display other information in the examples of Chapter 1 and the earlier examples of this chapter.

Figure A1–2. 3 shows most of the various types of set representations (some types of tables are not shown). The most general of the representations is the partitioned set. Any thing which can be represented using one of the other representations (e.g. N-tuples) can be represented using a partitioned set, but the partitioned set representation may be much less concise.

The partitioned set is a collection of objects which is all in one part, or is divided into several parts. The example in the first entry of Figure A1–2. 3 divides a set into three parts. The division means that these three parts are all subsets of the whole set, that the union of these three sets is the entire set, and that no two parts "overlap" —every pair of parts has an empty intersection. This last constraint is the "pairwise disjoint" constraint. There are many uses of the single-part partitioned set in this section and in section 1.

The top part of the example partitioned set contains two terms, a constant and a variable. The variable could be bound to the same value as the constant, making it so that (after instantiation) this part contained only one distinct member. Or, the variable

may be bound to a distinct term making this part contain two distinct members. The middle part contains no terms. It can be instantiated to a set of any number of terms; none (the empty set), 1 or many. This is, strictly speaking, another representation of a variable, where the variable *must* be bound to a set. The bottom part contains a single variable. Whatever the variable is instantiated to, this part must contain exactly one member. Due to the pairwise disjointness



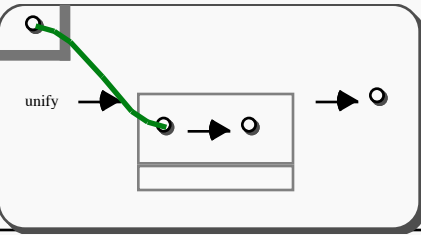

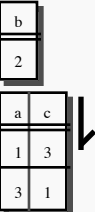
Set Type	Example Set						
Partitioned Set							
N-Tuple							
Intensional Set							
N-Tuple Table							
Function Table	<table border="1" data-bbox="899 909 1089 1031"> <thead> <tr> <th>Column A</th><th>Column B</th></tr> </thead> <tbody> <tr> <td>a</td><td>b</td></tr> <tr> <td>1</td><td>2</td></tr> </tbody> </table>	Column A	Column B	a	b	1	2
Column A	Column B						
a	b						
1	2						
Ordered, Factored Function Table							

Figure A1–2. 3: Examples of some types of sets with specialized representations.

constraint on parts of a partitioned set, the variable in the top part and the variable in the bottom part can not be instantiated to the same term. This is an implicit "inequality" constraint.

There is an example of an N-tuple in the second entry of Figure A1–2. 3. An N-tuple is a set of a particular construction:

- A 2-tuple (ordered pair) of $\langle X, Y \rangle$ is the set $\{\{X\}, \{X, Y\}\}$;
- An N-tuple of $\langle X_1, \dots, X_{(n-1)}, X_n \rangle$ equals $\langle \langle X_1, \dots, X_{(n-1)} \rangle, X_n \rangle$;
- An N-tuple of $\langle X \rangle$ equals X .

N-tuples have been used several times already in figures in this chapter and in Chapter 1.

The Intensional Set, or IntenSet, is a way of expressing "the set of all terms X

such that X has property P"; symbolically this is " $\{XIP(X)\}$ ". An example of an IntenSet is shown in the third entry of Figure A1–2. 3. The intensional set is a shorthand for the use of the "setof" built-in predicate: a way to specify the set of all terms such that some particular property (conjunction of literals) is true of those terms and no others. We will discuss it more in a later chapter.

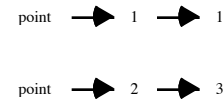


Figure A1–2. 4: Two triples representing the points (1,1) and (2,3).

Tables come in four basic varieties. Three examples are shown in the last three entries of Figure A1–2. 3. Figure A1–2. 3 is itself an example of a table (a “Function Table”). The basic types of tables are developed from combinations of two types of rows, N-tuple and function, and two types of row collections, set and N-tuple. Also, any kind of table may be "factored", where columns that are the same in all rows are extracted from the table and shown in a separate place. Finally, a table may have an initial row (a “0-th” row) which is treated as a single term instead of an N-tuple of terms. This last facility is particularly useful in defining a table which is a list of lists, where a list is an N-tuple with ‘empty list’ as its first element.

The N-tuple Table (shown in entry four of Figure A1–2. 3) is a set of rows, where each row is an N-tuple. All of the rows must be the same length. In this case, the rows are 2-tuples. The Function Table (shown in entry five of Figure A1–2. 3) is a set of rows, where each row is a function. A function is a set of ordered pairs (2-tuples), where the first element of the pair is the domain element and the second element of the pair is the range element. No two pairs in the function set can have the same domain element and different range elements. Such a set defines a function mapping from a domain to a range. All of the rows must have the same domains. In this case, the rows have the domain {"Column A", "Column B"}. The Ordered, Factored Function Table (shown in entry five of Figure A1–2. 3) is an N-tuple of rows, where each row is a function, as for the Function Table. In this example, the ordered pair "<b, 2>" has been factored out of the two rows.

2.1.2: Discussion of simple geometry representation. The two most widely used set representations are partitioned sets and N-tuples. We can use these to conveniently represent many different kinds of structures. In this section we see how some simple geometric objects can be represented.

The two 3-tuples in Figure A1–2. 4 show a way to represent geometric points, one

with $X = 1$ and $Y = 1$ and the other with $X = 2$ and $Y = 3$. An N-tuple is used here to provide a concise "naming" via element position to distinguish the X and the Y value.

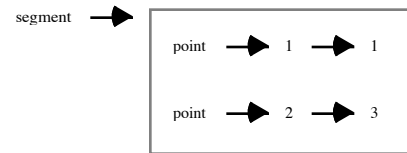


Figure A1–2. 5: An ordered pair representing a line segment with endpoints at (1,1) and (2,3).

The ordered pair (2-tuple) in Figure A1–2. 5 represents a line segment with endpoints at (1,1) and (2,3). The 2-tuple is used here to

"name" the endpoint data (telling us it defines a "segment"). The endpoint data is placed in a set, rather than in more elements of a N-tuple. This reflects the fact that there

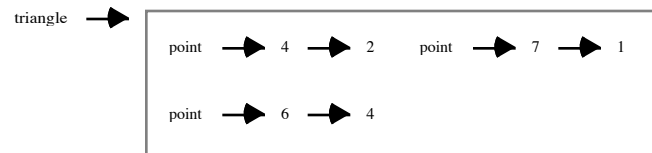


Figure A1–2. 6: An ordered pair representing a triangle with corners at (4,2), (7,1), and (6,4).

is no distinction between the two endpoints (this isn't a directed line).

The ordered pair in Figure A1–2. 6 represents a triangle with three "corners" at (4,2), (6,4), and (7,1). The ordered pair is used to "name" the collection of points as describing a "triangle". The points are in a set since no ordering of the points is needed.

1. Exercise for section 2.1.

Develop a representation for rectangles, squares, and circles as structured SPARCL objects. Use an approach similar to that given in the "Geometry Example" program. Write single argument clauses which have example terms of each of these in their argument and use a comment in the argument to explain the elements of the terms (e.g. "a triangle is represented by a set of three (X,Y) points which are the corners of the triangle"):

1. write a Rectangle clause of one argument which contains a term for a rectangle with diagonally opposite corners at (1,2) and (34,-5.8)
2. write a Circle clause of one argument which contains a term for a circle centered at (93,4) with a radius of 16
3. write a Square clause of one argument which contains a term for a square with upper left corner at (14, 23) and sides of length 123.

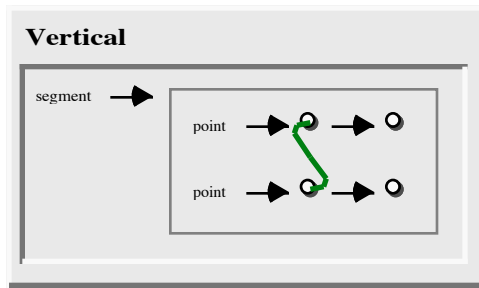


Figure A1-2. 7: Clause defining the 'Vertical'/1 predicate.

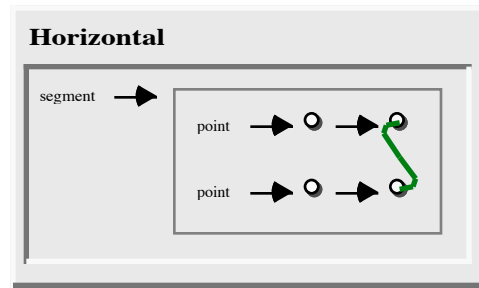


Figure A1-2. 8: Clause defining the 'Horizontal'/1 predicate.

2.2:Term matching. In the previous section we have seen how terms can be used to represent complex data objects. The most important operation on terms is *matching*. A special kind of matching is used in SPARCL called *unification*. Matching alone can produce some interesting computation.

Given two terms, we say they *match* if:

(1) they are identical

(2) the variables in both terms can be instantiated to objects in such a way that after the substitution of variables by these objects the terms become identical, and these instantiations don't violate any "current" partitioning constraints.

Using example predicates which define properties of line segments, we illustrate how matching alone can be used for interesting computation. Let us return to the simple geometric objects of the previous example and define a piece of program for recognizing horizontal and vertical line segments. The 'Vertical'/1 predicate in Figure A1-2. 7 is a property of segments, so it can be formalized in SPARCL as a unary relation. A segment is vertical if the x-coordinates of its end-points are equal, otherwise there is no other restriction on the segment. The property 'Horizontal'/1 is similarly formulated in Figure A1-2. 8, with only the x and y interchanged.

The query in Figure A1-2. 9 asks for the y value of a point such that there is a horizontal segment from (1,1) to that point with x = 2. The result in Figure A1-2. 10 shows the desired y value to be 1.

Matching with the partitioned set term provides a very powerful tool. The 'Union'/3 predicate in Figure A1-2. 11 relies entirely on this mechanism to relate two sets to the set which is their union.

The bottom part of set A, the top part of set B, and the middle part of set C are

connected and thus must be the same (i.e. they must unify with each other)--they must all "refer" to the same set which we'll call X. Since these parts are in sets A and B, X must be a set of terms common to A and B.

The top part of set A and the bottom part of set C are connected and thus must be the same—must refer to the same set which we'll call Y. Since one of these parts is in set A, set Y must be a subset of set A. Since one of these parts is in set A with a part for set X (mentioned earlier), X and Y must have nothing in common (the pairwise disjointness constraint).

The bottom part of set B and the top part of set C are connected and thus must be the same—must refer to the same set which we'll call Z. Since one of these parts is in set B, set Z must be a subset of set B. Since one of these parts is in set B with a part for set X (mentioned earlier), X and Z must have nothing in common (the pairwise disjointness constraint).

Since Y and Z are referred to by these two parts within the same partition, Y and Z must not have any terms in common.

To summarize what we know about the sets A, B, C, X, Y, and Z:

$$A = X \cup Y$$

$$B = X \cup Z$$

$$C = X \cup Y \cup Z$$

$$X \cap Y = \emptyset \left\{ \text{from the partitioning of } A \right\}$$

$$X \cap Z = \emptyset \left\{ \text{from the partitioning of } B \right\}$$

$$Y \cap Z = \emptyset \left\{ \text{from the partitioning of } C \right\}$$

From these equations we can infer:

Horizontal Query → 1

Figure A1–2. 10:
Result of querying
the clause in
Figure A1–2. 9.

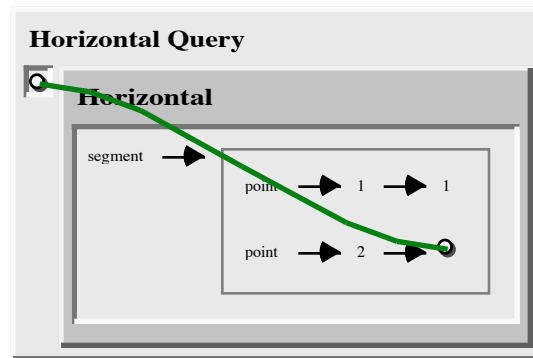


Figure A1–2. 9: Clause for posing the query
“What is the value of Y such that there is a
horizontal segment from (1,1) to (2,Y).”

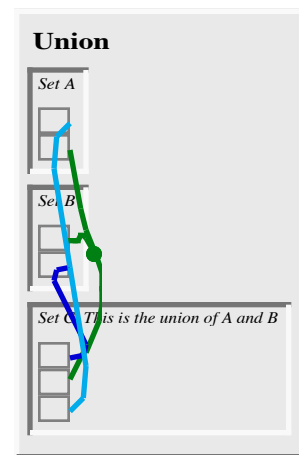


Figure A1–2. 11: Clause
defining the ‘Union’/3
predicate.

$$X = A \cap B$$

$$Y = A - B$$

$$Z = B - A$$

To infer the equation for X, suppose there is some term P which is common to A and B but is not in X, then this element must be in both Y and Z, but this violates the constraint that Y and Z are disjoint. The equation for Y derives from the equation for A as the union of X and Y and the equation for X as the intersection of A and B. The equation for Z is derived similarly to the equation for Y.

Finally, substituting the solutions for X, Y and Z into the equation for C:

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

which is equivalent to:

$$C = A \cup B$$

This is the desired result. The third argument of ‘Union’/3 (“C”) is the union of the first two arguments (“A” and “B”).

2.3: The declarative meaning of SPARCL. We have already seen in section 1 that SPARCL programs can be understood in two ways: declaratively and procedurally. In this and the next section we will consider a more formal definition of the declarative and procedural meanings of programs in basic SPARCL. But first let us look at the difference between the two meanings again.

Consider the clause in Figure A1–2. 12. Some alternative declarative readings of this clause are:

P is true if Q and R are true.

From Q and R follows P.

Alternative procedural readings of this clause are:

To solve problem P, solve the subproblems Q and R.

To satisfy P, satisfy Q and R.



Figure A1–2. 12
: ‘P’/0 example
with two literals.

These procedural readings do not say the order in which to solve or satisfy Q and R. The information given (the lone clause for P) doesn't allow us to infer any particular ordering. However, SPARCL is a fundamentally sequential language. It will choose an order in which to solve Q and R. The programmer may affect this choice using ‘*DELAY*’/2 clauses, the ‘ordered_disjunction’/2 built-in predicate, and the ‘if’/3

built-in predicate.

To see the use of the ‘*DELAY*/2 clause, consider the example in Figure A1–2. 13. Predicates ‘P’/2, ‘Q’/2, and ‘R’/2 each have two arguments and there is a ‘*DELAY*/2 clause.

The ‘P’/2 clause has a similar declarative reading as for the ‘P’/0 clause in Figure A1–2. 12:

P(X,Y) is true if Q(X,A) and R(A,Y) are true.

From Q(X,A) and R(A,Y) follows P(X,Y).

The ‘*DELAY*/2 clause makes the procedural reading of the example in Figure A1–2. 13 different from the procedural reading of the example in Figure A1–2. 12. There is now a constraint on when the ‘R’/2 subgoal may be attempted:

To solve problem P(X,Y), solve the subproblems Q(X,A) and R(A,Y). Delay any attempt to solve R(A,Y) as long as A is an unbound variable.

Since the first argument of ‘R’/2, in the clause for ‘P’/2, can only be bound before attempting to solve ‘R’/2 if ‘Q’/2 has been solved, this ‘*DELAY*/2 constraint on ‘R’/2 has the effect of ordering attempts at solving the subgoals of ‘P’/2.

Thus the difference between the declarative readings and the procedural ones is that the latter not only define the logical relations between the head of the clause and the goals in the body, but also (possibly) the *order* in which goals are processed.

2.3.1. A formalization of declarative meaning. The declarative meaning of programs determine whether a given goal is true, and if so, for what values of variables it is true. To precisely define the declarative meaning we need to introduce the concept of "instance" of a clause. An instance of a clause C is the clause C with each of its variables substituted by some term.

A goal G is true (that is, satisfiable, or logically follows from the program) if and only if

(1) there is a clause C in the program such that

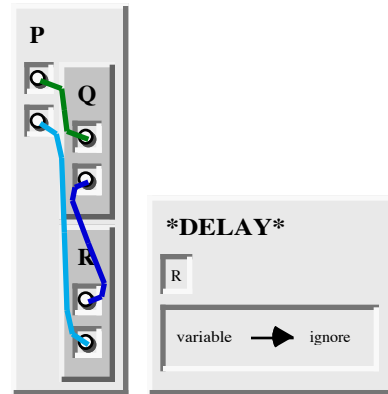


Figure A1–2. 13: ‘P’/2 example with two literals.

(2) *there is a clause instance I of C such that*
 (a) *the head of I is identical to G,*
 (b) *all of the goals in the body of I are true,*
 (c) *the instantiation of C to create I does not*
violate any partitioning constraints in C or I

This definition extends to SPARCL questions as follows. In general, a question to the SPARCL system is a set of goals. A set of goals is true if all of the goals in the set are true for the same instantiation of variables and that instantiation of variables does not violate any (implicit) partitioning constraints.

A set of goals, as just described, denotes the "conjunction" of those goals; they *all* have to be true. There is a built-in predicate in SPARCL which implements "disjunction"; 'ordered_disjunction'/2. This built-in is used

in the clause for 'P'/0 in Figure A1-2. 14. The declarative reading of this is "P is true if Q or R is true." The procedural reading of this is ordered: To solve problem P, first try to solve subproblem Q. If this succeeds, then P is solved. Otherwise, if solving subproblem Q fails, then solve subproblem R. If this succeeds then P is solved.

The 'P'/0 clause with the 'ordered_disjunction'/2 literal in Figure A1-2. 14 has the same declarative meaning as the two 'P'/0 clauses in Figure A1-2. 15. The difference in meaning is only procedural. With these two clauses, there is no guarantee which ordering of the P clauses the SPARCL interpreter will use when attempting to solve the P problem.

2.4: The procedural meaning of SPARCL. The procedural meaning specifies *how* SPARCL answers questions. To answer a question means to try to satisfy a set of goals. They can be satisfied if the variable that occur in the goals can be instantiated in such a way that the goals logically follow from the program. Thus the procedural meaning of SPARCL is a procedure for executing a set of goals with respect to a given program. To "execute goals" means try to satisfy them.

Let us call this procedure "execute set". The inputs and the outputs for this procedure are:

input: a program, a goal set, and a set of partitioning constraints

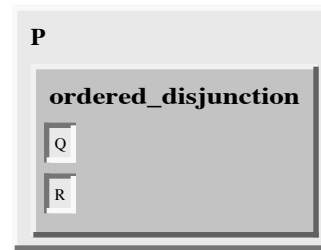


Figure A1-2. 14: 'P'/0 ordered_disjunction example.

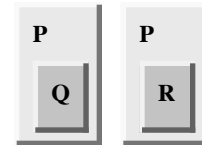


Figure A1-2. 15: 'P'/0 example of disjunction implemented using two clauses.

output: a success/failure indicator and an instantiation of variables

The meaning of the two output results is as follows:

- (1) The success/failure indicator is “yes” if the goals are satisfiable and “no” otherwise. We say that “yes” signals a successful termination and “no” a failure.
- (2) An instantiation of variables is only produced in the case of a successful termination; in the case of failure there is no instantiation.

In section 1, we discussed informally what procedure “execute set” does, under the heading “How SPARCL answers questions.” What follows in this section is a more formal and systematic description of this process.

To execute a set of goals $\{G1, \dots, Gm\}$ with partitioning constraints P the procedure “execute” does the following:

- Order the goal set to create a list $\{G1', \dots, Gm'\}$
- invoke "execute list".

To execute a LIST of goals $[G1', \dots, Gm']$ with partitioning constraints P the procedure "execute list" does the following:

- If the goal list is empty then if the delayed goals list is empty terminate with "success", else terminate with "failure".
- If the goal list is not empty then divide the list into the first goal, $G1'$, and the OtherGoals.
- If $G1'$ is the special "marker" goal 'end_body', then replace it with the DelayedGoalsList ($[DG1, \dots, DGk]$) to create $[DG1, \dots, DGk, G2', \dots, Gm']$ and recursively invoke execute_list with this new goal list and an empty delayed goals list. The result of the recursive invocation is the result of this invocation.
- Else, if $G1'$ is a goal which should be delayed according to the *DELAY* definitions in the program then add $G1'$ to the DelayedGoals and recursively invoke execute_list with the OtherGoals ($[G2', \dots, Gm']$) and the extended DelayedGoals. The result of the recursive invocation is the result of this invocation.
- Otherwise (i.e. if GoalList is not empty, $G1'$ is not 'end_body', and $G1'$ does not need to be delayed) continue with (the following) operation called "SCANNING".
- SCANNING: Scan through the clauses in the program in any order until a

clause, C , is found such that the head of C matches the first goal $G1'$ without violating the current partitioning constraints P . If there is no such clause then terminate with "failure".

If there is such a clause C with head H and body goals $\{B1, \dots, Bn\}$, then replace the variables of C with new variables (essentially, rename the variables of C) to obtain a variable C' of C , such that C' and the list $G1', \dots, Gm'$ have no common variables. Let C' have head H' and body $\{B1', \dots, Bn'\}$. Let $G1'$ match H' ; let the resulting instantiation of variables be S and extend partitioning constraints P to be P' .

Order the goal set $\{B1', \dots, Bn'\}$ to create the goal list

$[B1'', \dots, Bn'']$.

In the goal list $[G1', \dots, Gm']$, replace $G1'$ with the list $[B1'', \dots, Bn'']$, obtaining a new list

$[B1'', \dots, Bn'', G2', \dots, Gm']$.

(Note that if C is a fact then $n = 0$ and the new goal list is shorter than the original one; such shrinking of the goal list may eventually lead to the empty list and thereby a successful termination.)

Substitute the variables in this new goal list with new values as specified in the instantiation S , obtaining another goal list

$[B1''', \dots, Bn''', G2'', \dots, Gm'']$

- Execute (recursively with procedure "execute list") this new goal list. If the execution of this new goal list terminates with success then terminate the execution of the original goal list also with success. If the execution of the new goal list is not successful then abandon this new goal list and go back to SCANNING through the program. Continue the scanning with any untried clause and try to find a successful termination using some other clause.

This procedure can be written in a Pascal-like notation as shown in Figure A1–2. 16. Several additional remarks are in order here regarding the procedures "execute_set" and "execute_list" as presented. First, it was not explicitly described how the final resulting instantiation of variables is produced. It is the instantiation S which led to a successful terminate, and was possibly further refined by additional instantiations that were done in the nested recursive calls to "execute_list".

Whenever the recursive call within SCANNING to "execute_list" fails, the execution returns to SCANNING, continuing at the program that had been last used before. As the application of the clause C did not lead to a successful termination SPARCL has to try an alternative clause to proceed. What effectively happens is that SPARCL abandons this whole part of the unsuccessful execution and backtracks to the point (clause C) where this failed branch of the execution was started. When the procedure backtracks to a certain point, all of the variable

```

procedure execute_set (Program, GoalSet, Constraints, Success)
begin
  OrderedGoalsList := order_goal_set(GoalSet);
  execute_list(Program, OrderedGoalsList, [], Constraints, Success);
end;

procedure execute_list (Program, GoalList, DelayedGoals,
  Constraints, Success)
begin
  if empty(GoalList) then
    begin
      if empty(DelayedGoals) then
        Success := true
      else Success := false
      end
    end
  else
    begin
      Goal := head(GoalList);
      OtherGoals := tail(GoalList);
      if Goal = end_body then
        begin
          NewGoals := append(DelayedGoals, OtherGoals);
          execute_list(Program, NewGoals, [], Constraints, Success);
        end
      else if delay(Goal, Program) then
        begin
          NewDelayedGoals := append(DelayedGoals, [Goal]);
          execute_list(Program, OtherGoals, NewDelayedGoals,
            Constraints, Success);
        end
      else
        begin
          Satisfied := false;
          while not Satisfied and "more clauses in program" do
            begin
              Let next clause in Program be
                head H and body {B1, ..., Bn}.
              Construct a variant of this clause
                head H' and body {B1', ..., Bn'}.
              match(Goal, H', Constraints, MatchOK, Instant,
                MatchConstraints);
              if MatchOK then
                begin
                  OrderedBodyGoals := order_goal_set({B1', ..., Bn'});
                  ExtendedBodyGoals := append(OrderedBodyGoals,
                    [end_body]);
                  NewGoals := append(ExtendedBodyGoals, OtherGoals);
                  NewGoals := substitute(Instant, NewGoals);
                  execute_list(Program, NewGoals, MatchConstraints,
                    Satisfied);
                end
              end;
            end
            Success := Satisfied
          end
        end
      end;
    end
  end;
end;

```

Figure A1–2. 16: execute_set and execute_list procedures.

instantiations that were done after that point are undone. This ensures that SPARCL systematically examines all of the possible alternative paths of execution until one is found that eventually succeeds, or until all of them have been shown to fail.

The actual implementation of SPARCL adds many refinements to the execute procedures. One of them is to reduce the amount of scanning through the program clauses to improve efficiency. So SPARCL will not scan through all of the clauses of the program, but will only consider the clauses about the relation in the current goal.

3: Presentation of an application of SPARCL

This section presents an application of SPARCL, "Column Sum". This program takes a function term table and a domain value and produces the sum of all of the range values for that domain value in the function term table. This is analogous to finding the sum of the values of a given column of a spreadsheet. This program demonstrates the use of several aspects of SPARCL including term tables, intensional sets, multisets, and arithmetic. We demonstrate the creation of the clause defining the 'Column Sum'/3 predicate and a "query" clause using this predicate and explain various aspects of these clauses and their use.

First we will create the "Column Sum" clause. This one clause is the entire definition of the "Column Sum" predicate. This clause with empty arguments and an empty body is shown in Figure A1–3. 1.

Step 1: Create the "Column Sum" clause.

Step 1.1.

DO: Create a new clause named "Column Sum" with 3 arguments in program "Column Sum"

BY: Select the "Create Clause" option in the popup menu for the program.
Add 3 arguments.

Step 2: Create the comments and variables of the arguments of the "Column Sum" clause.



Figure A1–3. 1:
Clause for
'Column Sum'/3
with empty arguments and empty body.

There are three arguments to this clause. Each of these arguments will be given a variable and a comment describing the purpose of the variable. The steps for creating and editing a comment, the sub-steps of step 2.1, are very similar to those for creating and editing an ur constant.

Step 2.1: Create a new comment "Function Table" in the first argument of the clause.

Step 2.1.1.

DO: Create a new comment in the first argument of the clause.

BY: Select the "Create Comment" option in the popup menu for the argument.

Step 2.1.2.

DO: Edit the value of "open" comment "Function Table".

BY: Enter the characters of the new value.

Step 2.1.3.

DO: Close the current edit "box" for program "Column Sum".

BY: Click in the program window anywhere outside of the box.

Step 2.2.

DO: Create a new variable in the second argument of the clause.

BY: Select the "Insert:Variable" option in the popup menu for the argument.

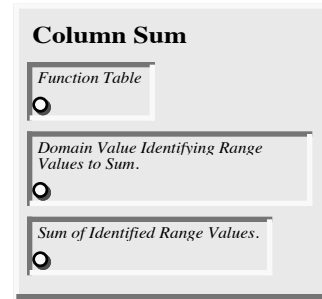


Figure A1-3. 2: Clause for 'Column Sum'/3 with arguments filled in and an empty body.

Steps 2.3 through 2.6 repeat the basic process of steps 3.2.1 and 3.2.2 to fill in the second and third arguments of the 'Column Sum'/3 clause. The result is shown in Figure A1-3. 2.

The next major step, 3, creates the single literal using 'is'/2 that is the body of the clause. This literal does the arithmetic to determine the sum. One of the terms in the second argument of this literal collects together the values which are to be summed.

Step 3: Create the literal of the "Column Sum" clause.

In steps 3.1 and 3.2 we create the literal with empty arguments and fill in the first argument. The first argument is the the result of the arithmetic evaluation of the expression in the second argument.

Step 3.3.1.

DO: Create a new literal with name "is" and 2 arguments in the clause.

BY: Select the "Create Literal" option in the popup menu for the clause.

Step 3.3.2.

DO: Create a new variable in the first argument.

BY: Select the "Insert:Variable" option in the popup menu for the argument.

The next steps build the expression which is the sum of the range values of the specified domain value. The "sum" part of this expression is represented by a '+' and the set of the range values is represented by an intensional (multi)set. The expression is an ordered pair with '+' as its first argument and the intensional multiset as its second argument. To build this N-tuple (2-tuple), we first create what will be the first element of the N-tuple, the '+' ur constant.

Step 3.3: Create a new ur constant of value "+" in the second argument.

The next step achieves two purposes: it replaces the '+' ur constant with an 2-tuple which has that ur constant as its first element, and it creates an empty

intensional set as the second element of the new 2-tuple. The interaction for this step is shown in Figure A1–3. 3. The result is shown in Figure A1–3. 4.

Step 3.4.

DO: Create an N-tuple with the ur constant “+” as the first element and create a new intensional set term as the second term.

BY: Select the "Create NTuple:IntenSet" option in the popup menu for “+”.

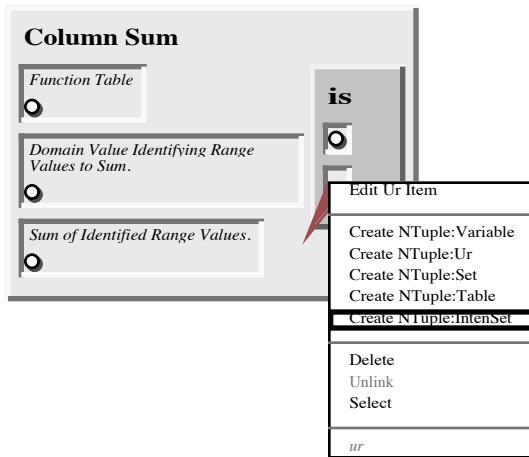


Figure A1–3. 3: Interaction for step 3.4 to convert the ‘+’ ur constant to a 2-tuple with ‘+’ as first element and an empty IntenSet as the second element.

An intensional set, such as shown in Figure A1–3. 4 as the second argument of a 2-tuple, is a term that specifies the set of all terms which have the given property. The type of the intensional set may be either "set" or "multiset". A multiset intensional set (or "intensional multiset") is the multiset of all terms which have the given property. The representation of an intensional (multi)set has two parts, the template and the body. The body is a set of literals.

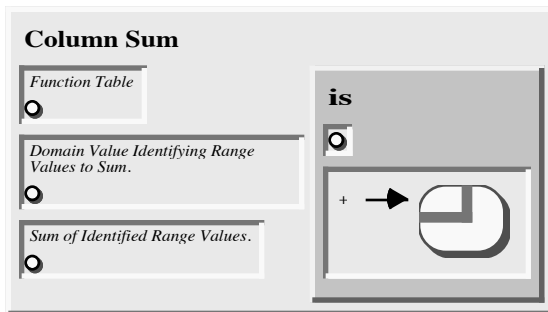


Figure A1–3. 4: Result of step 3.4.

The template contains a term (which almost always contains at least one variable) that is instantiated once for every "way" in which the literals of the body are true. The "result" of the intensional (multi)set is the (multi)set of all of these instantiations of the template.

We are using the intensional multiset to collect together all of the values of the given domain of the given function table. Since we want to sum all of these values, we want to keep the duplicates. Thus, we want the result type of the intensional set to be “multiset.” The next step changes the type appropriately. The interaction is shown in Figure A1–3. 5. The result is shown in Figure A1–3. 6.

Step 3.5.

DO: Set the "result type" of the intensional_set to multiset.

BY: Select the "Result Type:Multiset" option in the popup menu for the `intensional_set`.

The template we want is simply a single variable. This will be connected to the range term of each function pair with the given domain term in the given table.

Step 3.6.

DO: Create a new variable in the `intensional_set_template`.

BY: Select the "Insert:Variable" option in the popup menu for the `intensional_set_template`.

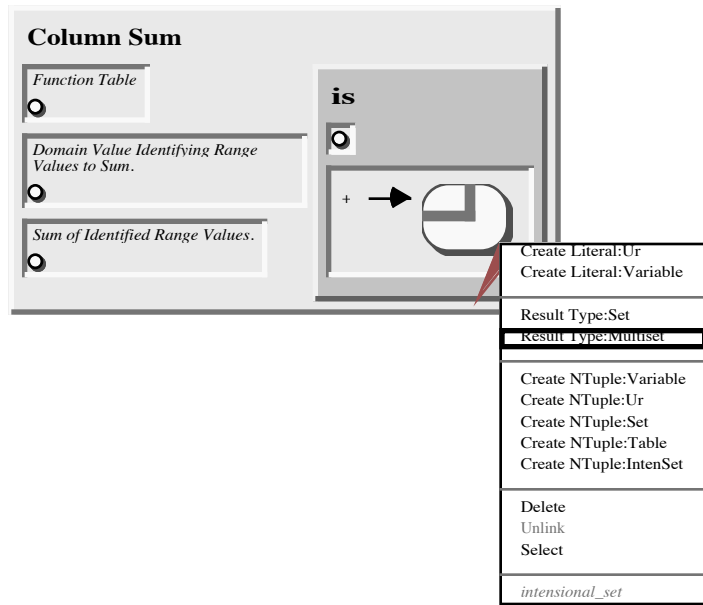


Figure A1-3. 5: Interaction for step 3.5 to convert an intensional set to an intensional multiset.

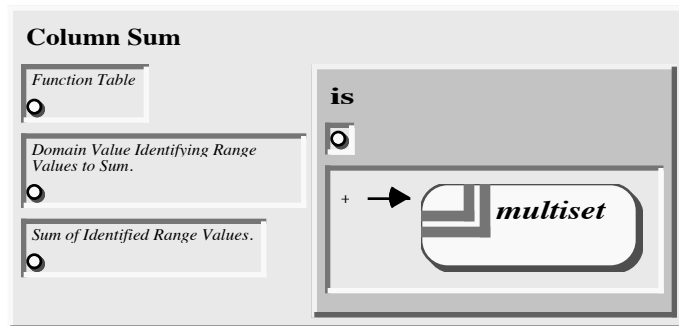


Figure A1-3. 6: Result of step 3.5 converting an intensional set to an intensional multiset.

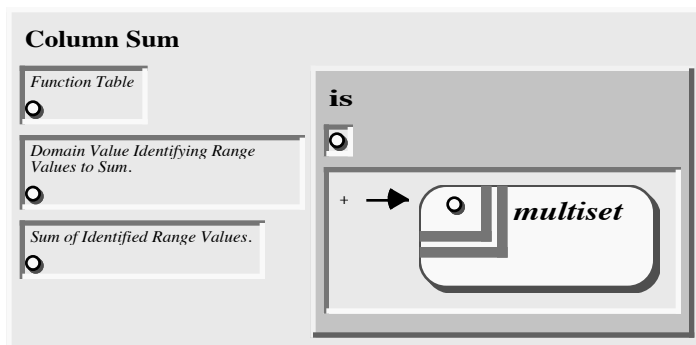


Figure A1-3. 7: Result of step 3.6 to create a variable in the template of the intensional multiset.

Next we create a "unify" literal. This will give us all possible unifications of a function pair with the given function table. The steps for editing the new literal ur in an intensional are similar to those for editing a

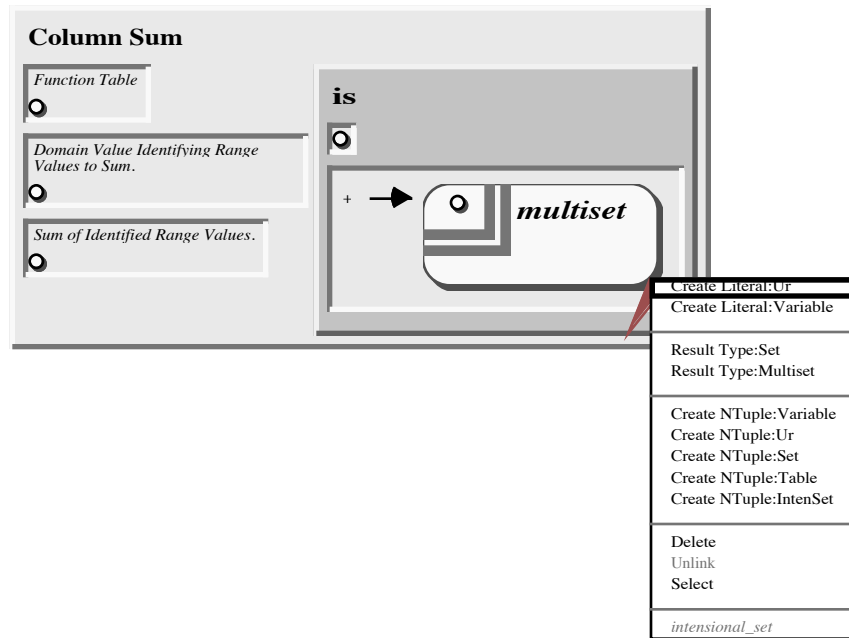


Figure A1-3. 8: Interaction for creating a new literal in an intensional (multi)set.

new ur constant. The interaction for creating the new literal ur is shown in Figure A1-3. 8. The result of creating the literal ur and editing the literal name to be "unify" is shown in

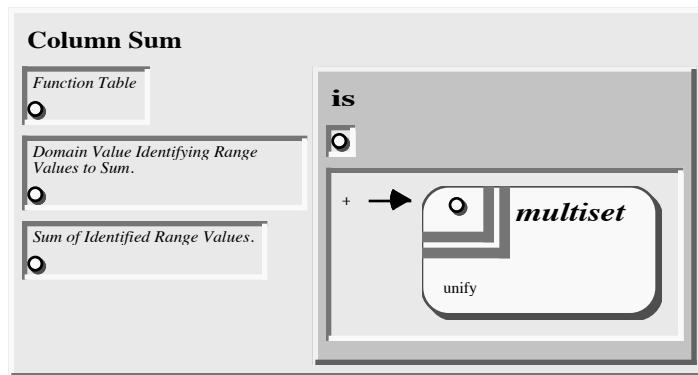


Figure A1-3. 9: Result of creating new literal "unify" ur.

Step 3.7.

DO: Create a new literal with an ur constant predicate name in the intensional_set.

BY: Select the "Create Literal:Ur" option in the popup menu for the intensional_set.

Step 3.8.

DO: Edit the value of the "open" ur constant to "unify".

BY: Enter the characters of the new value.

Step 3.9.

DO: Close the current edit "box" for program "Column Sum temp".

BY: Click in the program window anywhere outside of the box.

At this point, we have a ‘unify’/0 literal. To make a ‘unify’/2 literal, we need two arguments added to ‘unify’/0. This is represented by an N-tuple of three elements, the first element is the "unify" constant and the second and third elements are the two arguments. The intensional set is a shorthand for the *meta-predicate* ‘setof’/3. A

meta-predicate is a predicate that takes terms that are interpreted as literals in one or more of its arguments. A term to be interpreted as a literal is a constant or an N-tuple. A constant is interpreted as a literal of no arguments, a N-tuple is interpreted as a literal of

(N-1) arguments. In the next step we replace the “unify” ur constant with an N-tuple of two elements: the “unify” ur constant and a variable. The interaction is shown in Figure A1–3. 10. The result is shown in Figure A1–3. 11.

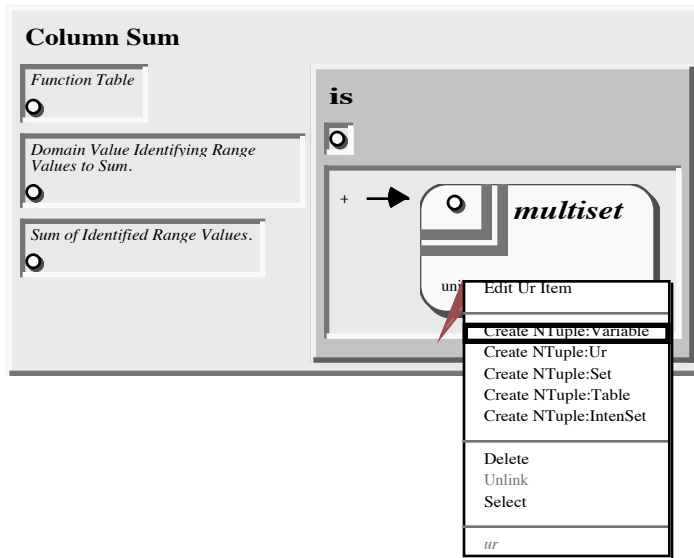


Figure A1–3. 10: Interaction to create an ordered pair (2-tuple) with “unify” as first element and a variable as the second element.

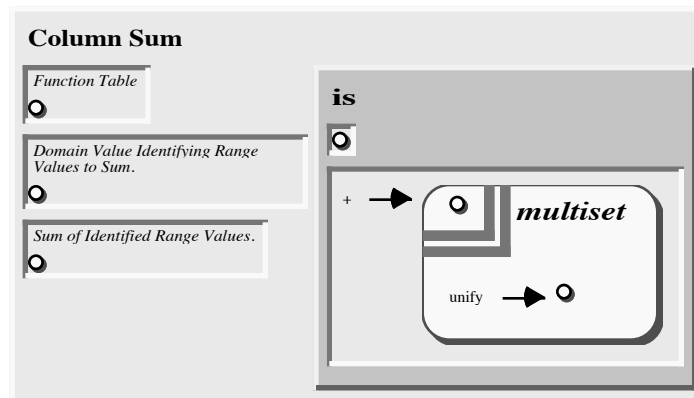


Figure A1–3. 11: Result of interaction to create an ordered pair with “unify” as the first element and a variable as the second element.

Step 3.10.

DO: Create an N-tuple with ur constant “unify” as the first element and create a new variable term as the second term. (Call the new N-tuple object “N-tuple 2”, call the new variable object “variable 3.”)

BY: Select the "Create NTuple:Variable" option in the popup menu for ur constant “unify”.

Now we have an 2-tuple of "unify" and a variable. To complete the "unify" literal, we extend this 2-tuple with another element, an empty set. The interaction is shown in Figure A1–3. 12 and the result is shown in Figure A1–3. 13.

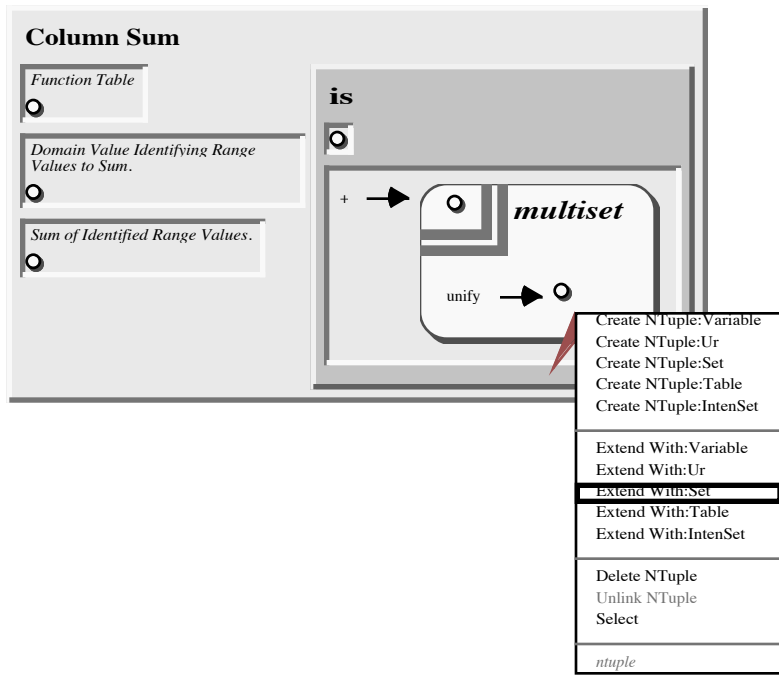


Figure A1–3. 12: Interaction to extend an ordered pair (2-tuple) to a 3-tuple with an empty set as the third element.

Step 3.11.
DO: Extend N-tuple 2, creating a new set term as the new last element. (Call the new object "set 1".)
BY: Select the "Extend With: Set" option in the popup menu for the N-tuple.

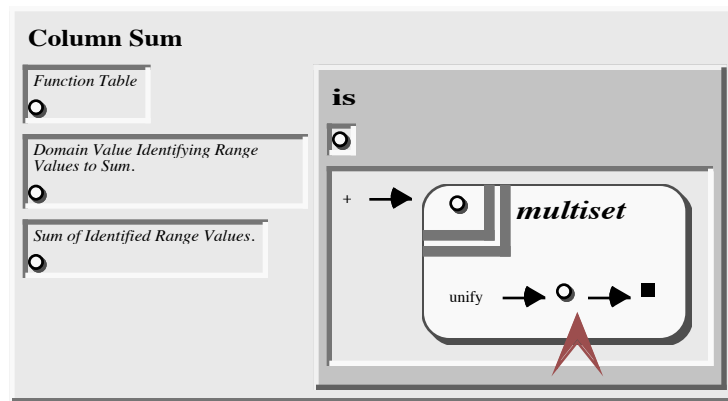


Figure A1–3. 13: Result of interaction to extend an ordered pair (2-tuple) to a 3-tuple with an empty set as the third element.

Eventually, we will connect the variable that is the first argument to "unify" with the given function table. The second argument to "unify", which is currently an empty set, we are going to turn into a pattern term which can match (or unify) with any function pair (domain value/range value) in a function table. Since this unify literal is in the body of an intensional set, evaluation of the intensional set will produce all possible unifications of the pattern term with the given

table.

The pattern term is an ordered-pair (2-tuple) within a part of a two-part partitioning of a set (the "inner" set), which is in turn within a part of a two-part partitioning of a set (the "outer" set). The ordered pair will unify with any ordered pair in a function (which is a set of ordered pairs where no two first elements are the same). The two parts of the "inner" will unify with any table where one part contains one ordered pair and the other part contains any number of terms (and is possibly empty). This partitioned "inner" set will unify with any function (or row) of the given table. The two parts of the "outer" set will unify with any table where one part contains the "inner" set (a row of the table) and the other part contains any number of terms (and is possibly empty).

Our first step in creating this pattern term is to create an empty set within the existing empty set. The existing empty set will be converted to a partitioned set of one part. This is the "outer"

set. The newly created empty set will become the "inner" set. The interaction is shown in Figure A1-3. 14 and the result is shown in

Figure A1-3. 15.

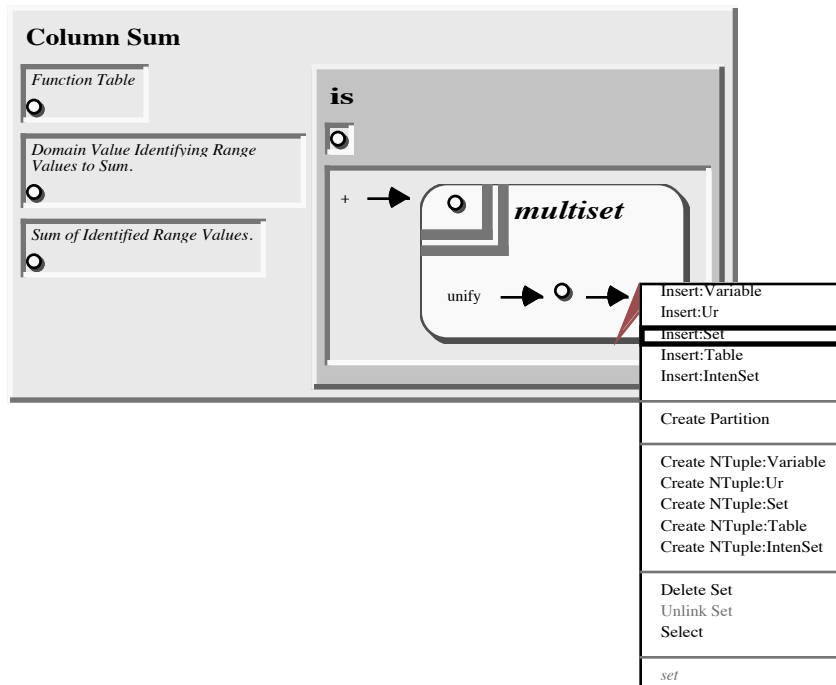


Figure A1-3. 14: Interaction to convert an empty set to a partitioned set of one part and create an empty set inside that new part.

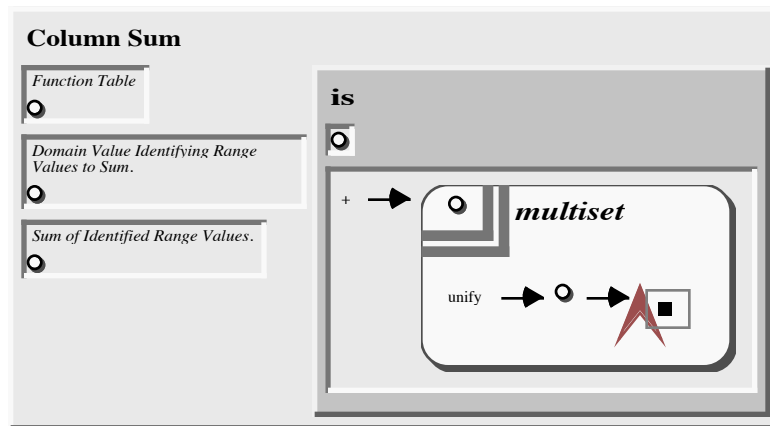


Figure A1-3. 15: Result of interaction to convert an empty set to a partitioned set of one part and create an empty set inside that new part.

Step 3.12.

DO: Create a new set in set 1. (Call the new object "set 2".)

BY: Select the "Insert:Set" option in the popup menu for the set.

Next we add a "hollow" second part to the "outer" set partitioning. The interaction is shown in Figure A1-3. 16, and the result is shown in Figure A1-3. 17.

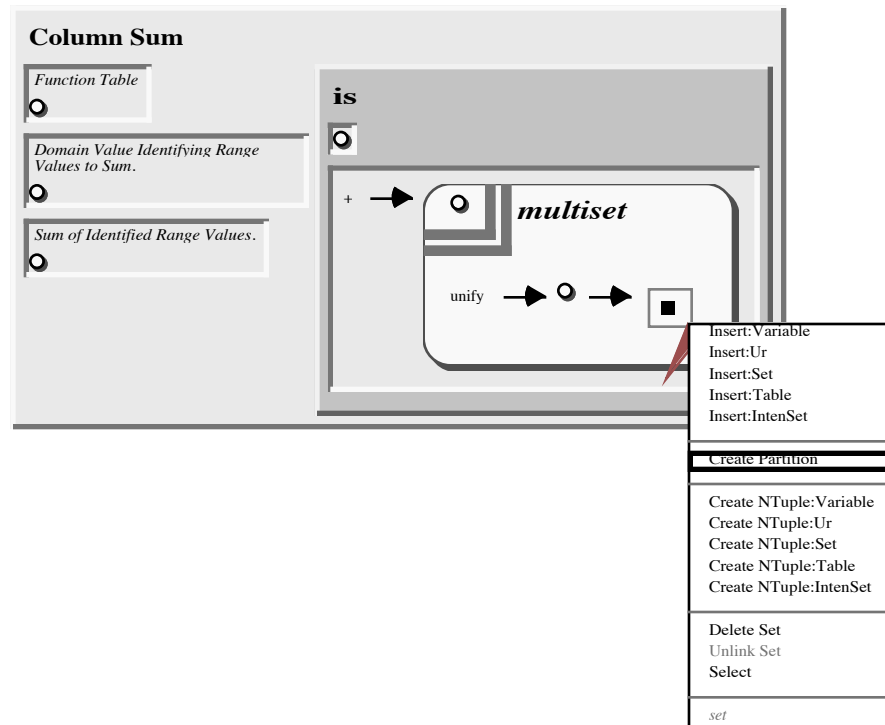


Figure A1-3. 16: Interaction to create a “hollow” part in a partitioned set.

Step 3.13.

DO: Create a new partitioned set part in the set 1.

BY: Select the "Create Partition" option in the popup menu for set 1.

Now we build the ordered pair inside of the "inner" set. First, we create a variable inside the "inner" set. This will convert the "inner" set from an empty set to a

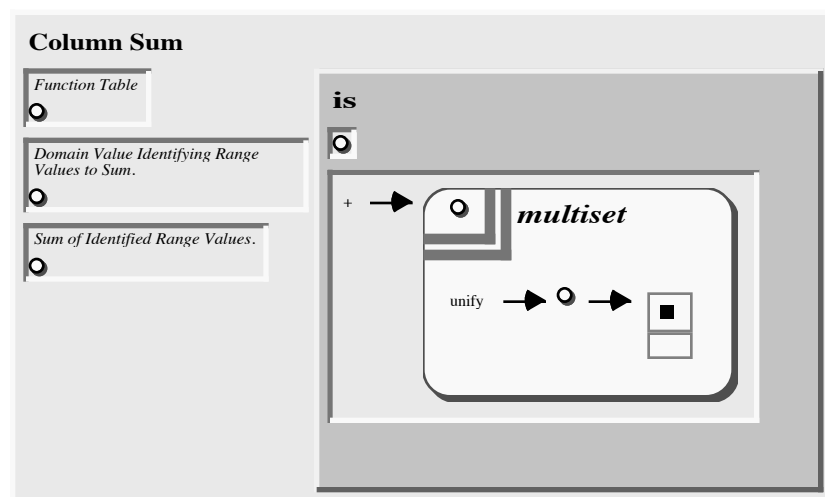


Figure A1-3. 17: Result of interaction to create a “hollow” part.

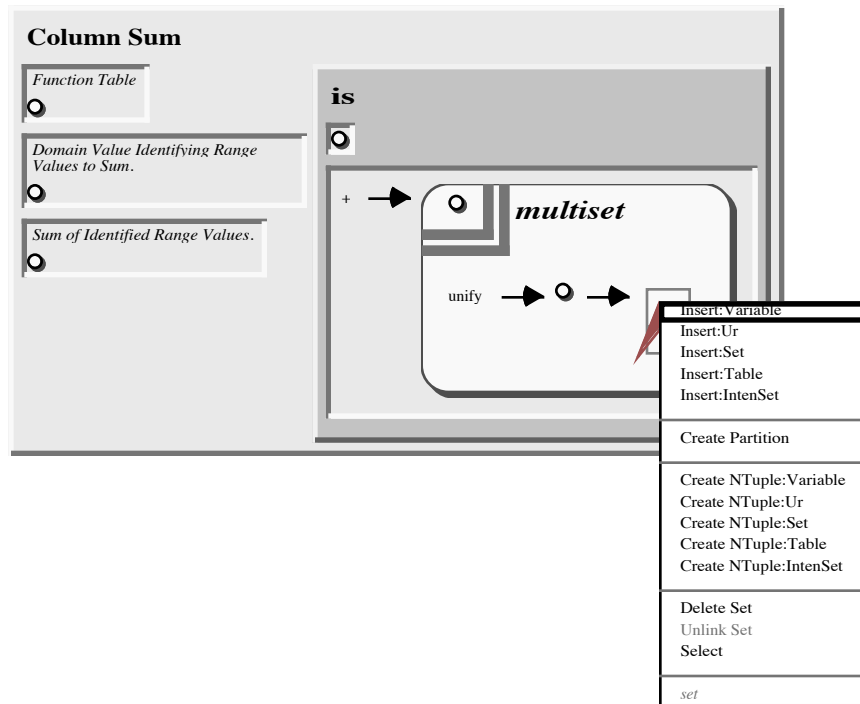


Figure A1-3. 18: Interaction to create a variable.

partitioned set of one part, and place a variable inside this one part. The interaction is shown in

Figure A1-3. 18 and the result is shown in

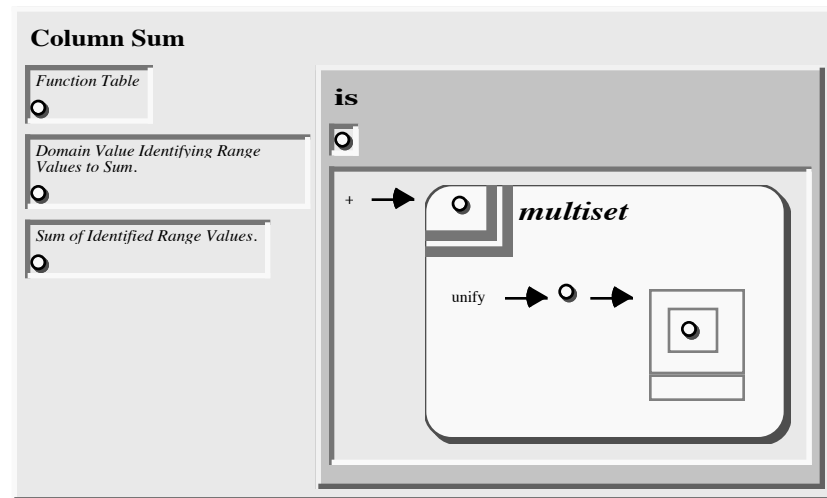


Figure A1-3. 19: Result of creation of a variable.

Step 3.14.

DO: Create a new variable in set 1. (Call this object "variable 4".)

BY: Select the "Insert:Variable" option in the popup menu for the set.

This variable is the generic domain value variable. We replace this variable by an

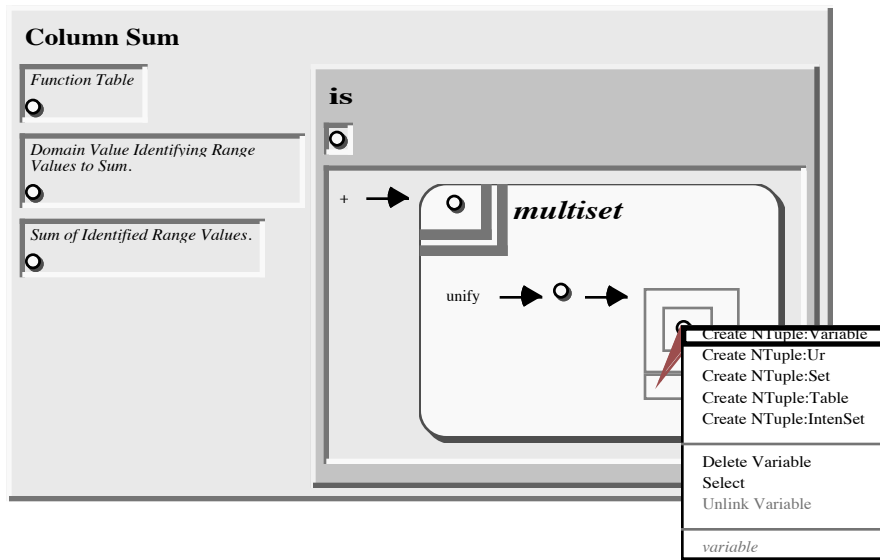


Figure A1-3. 20: Interaction to convert a variable to a 2-tuple of two variables.

N-tuple with this variable as its first element and another variable as its second element. This is shown in Figure A1-3. 20 and Figure A1-3. 21 .

Step 3.3.15.

DO: Create an N-tuple with variable 4 as the first element and create a new variable term as the second term. (Call this new N-tuple object “N-tuple 3” and the new variable object “variable 5.”)

BY: Select the "Create NTuple:Variable" option in the popup menu for the variable.

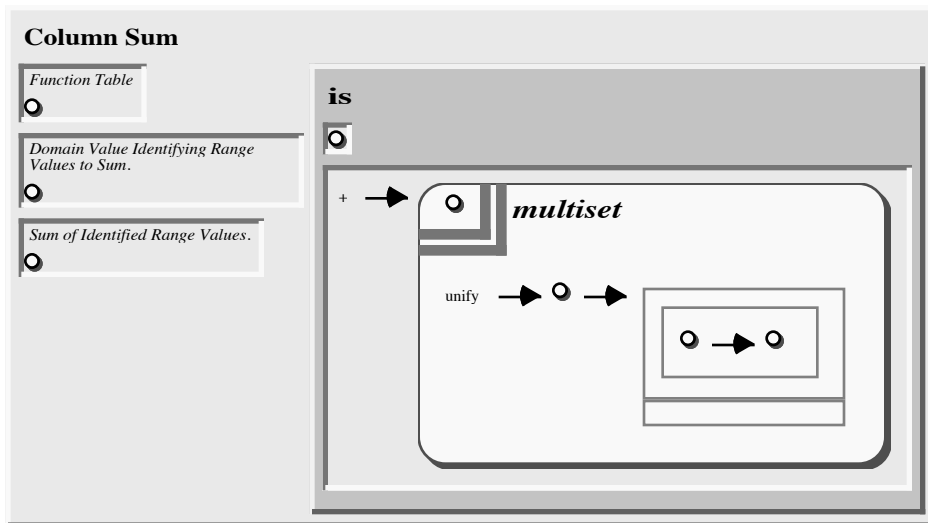


Figure A1-3. 21: Result of interaction to convert a variable to a 2-tuple of two variables.

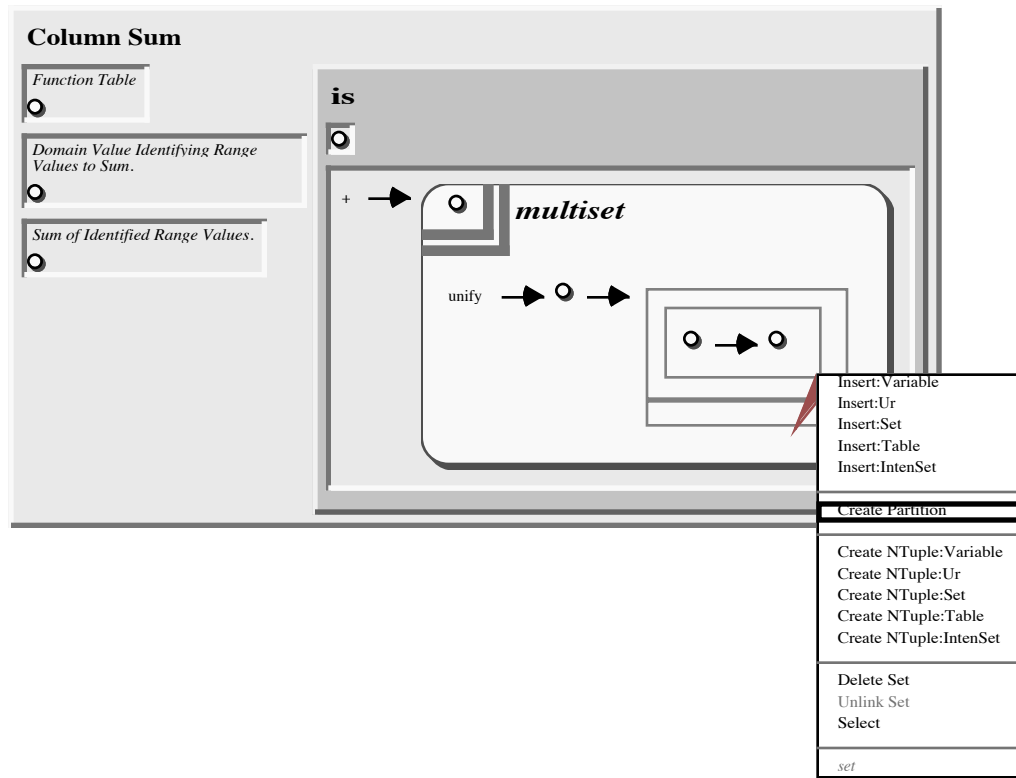


Figure A1-3. 22: Interaction to create a “hollow” part in a partitioned set.

The new variable in Figure A1-3. 21 is the generic range value variable. This completes the function pair N-tuple. To finish this "pattern" term, we will create a second "hollow" part in the "inner" set. The interaction and result are shown in Figure A1-3. 22 and Figure A1-3. 23.

Step 3.16.

DO: Create a new partitioned set part in the set 2.

BY: Select the "Create Partition" option in the popup menu for the set.

We complete the intensional set by connecting the template variable to the generic range value variable. This is shown in Figure A1-3. 24 and Figure A1-3. 25.

Step 3.17.

DO: Create a coreference link including the intensional set template variable and variable 5.

BY: With "connect tool" as the current tool, depress the mouse button while the cursor is over one of the variables and drag the cursor until it's over the other variable, then release the mouse button.

To complete the "Column Sum" clause, we need only to connect some terms. We

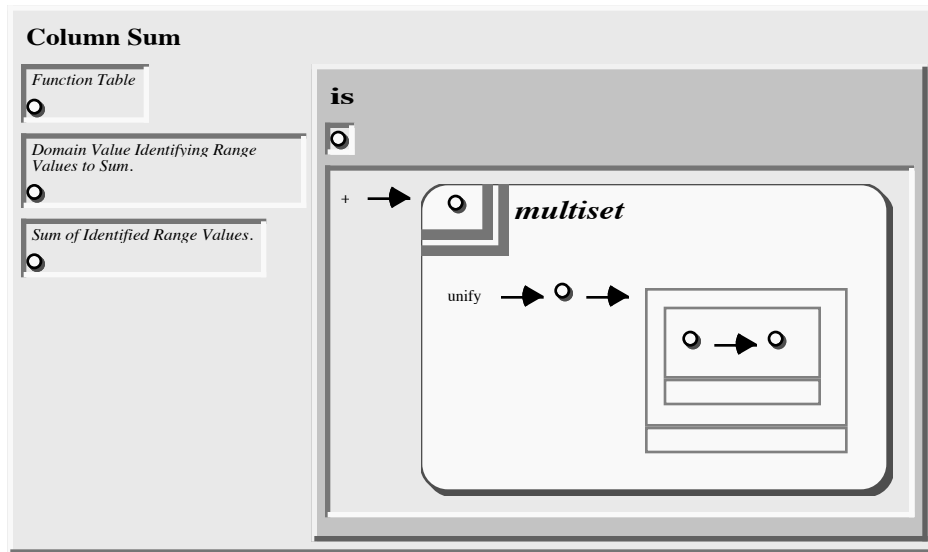


Figure A1-3. 23: Result of creating “hollow” part in partitioned set.

connect the "given table" argument variable to the intensional set unification table variable. This specifies that the intensional set "ranges" over the given table. This is shown in Figure A1-3. 25 and Figure A1-3. 26.

Step 3.18.

DO: Create a coreference link including the variable in the first argument of the clause and variable 3.

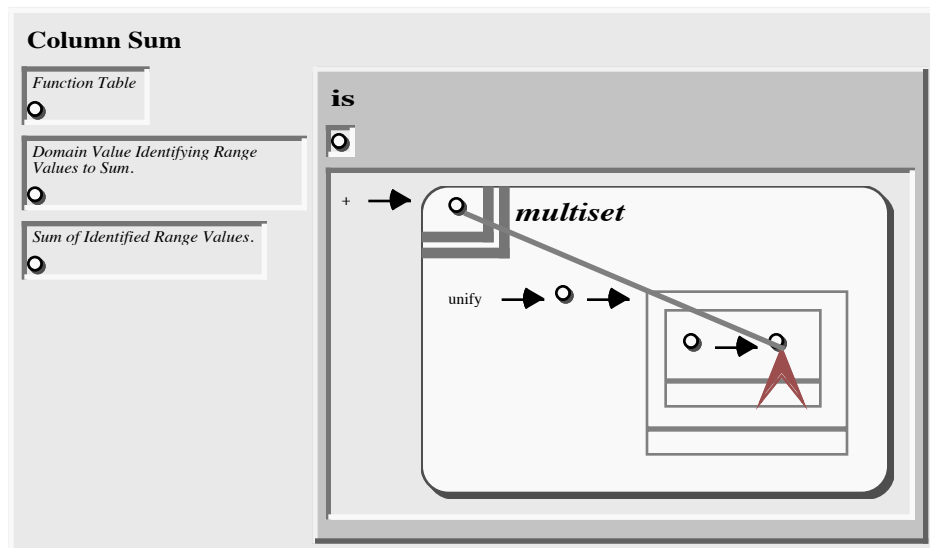


Figure A1-3. 24: Interaction (in “connect” mode) to create a link between the template variable and the range variable of the “pattern”.

BY: With "connect tool" as the current tool, depress the mouse button while the cursor is

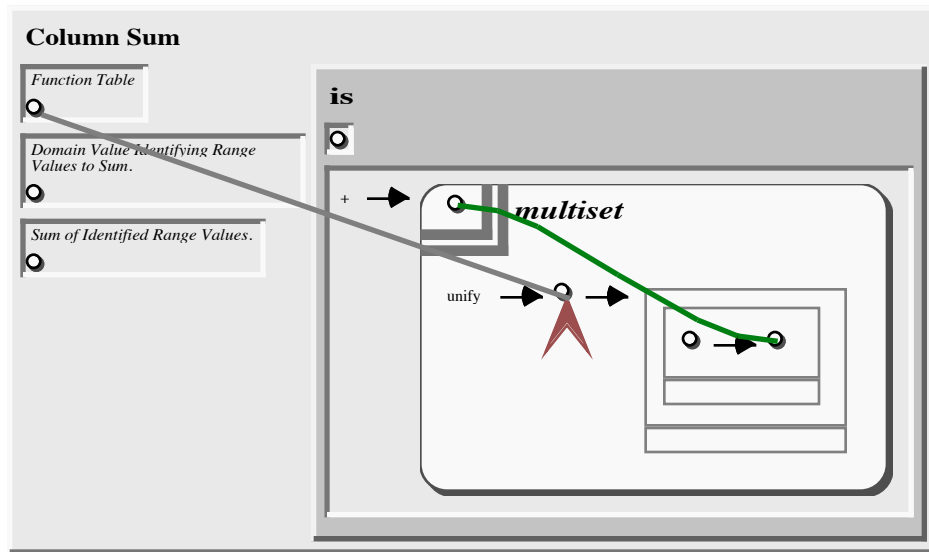


Figure A1-3. 25: Result of linking the template and pattern “range value” variables. Interaction to create a link between the “function table” argument variable and the first argument variable of the intensional set ‘unify’/2 literal 3-tuple.

over one of the variables and drag the cursor until it's over the other variable, then release the mouse button.

We connect the "given domain value" argument variable to the first element of the ordered pair in the intensional set unification. This specifies that only range values

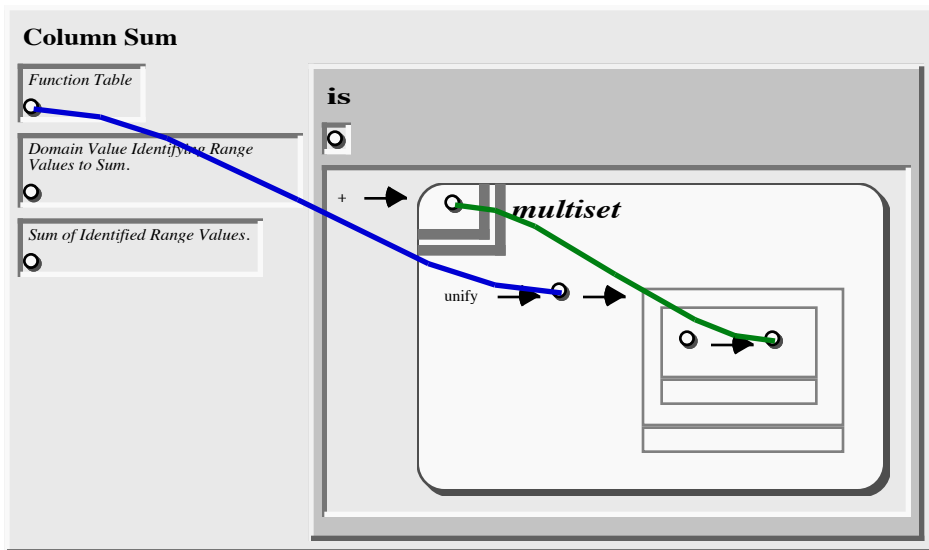


Figure A1-3. 26: Result of creating the “function table” link.

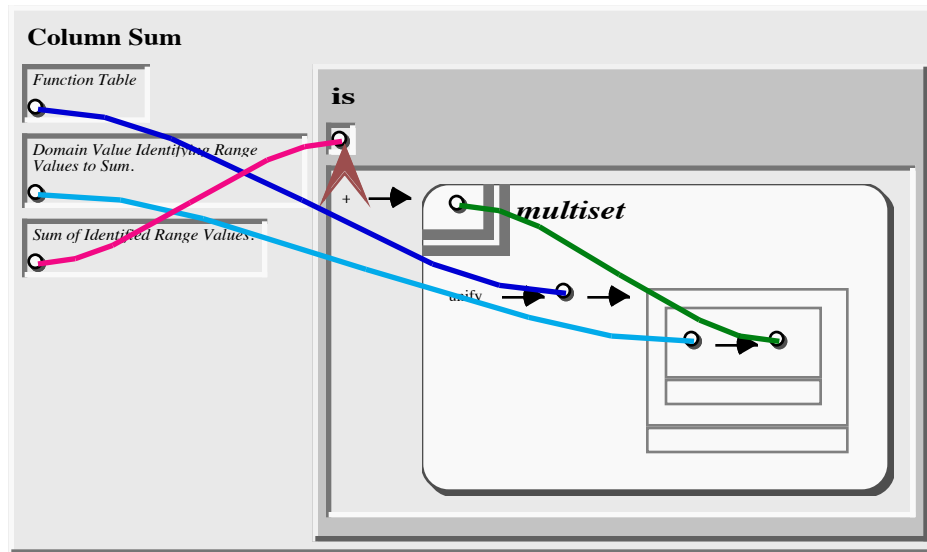


Figure A1-3. 27: Complete clause defining the 'Column Sum'/3 predicate.

paired with a domain value the same as the "given" domain value will be collected in the intensional set result.

Step 3.19.

DO: Create a coreference link including the variable in the second argument of the clause and variable 4.

BY: With "connect tool" as the current tool, depress the mouse button while the cursor is over one of the variables and drag the cursor until it's over the other variable, then release the mouse button.

The last connection makes the sum of the intensional multiset the result of the clause.

Step 3.20

DO: Create a coreference link including the variable in the third argument of the clause and the variable in the first argument of the literal.

BY: With "connect tool" as the current tool, depress the mouse button while the cursor is over one of the variables and drag the cursor until it's over the other variable, then release the mouse button.

Having finished creating the clause which defines the 'Column Sum'/3 predicate as shown in Figure A1-3. 27, we will create a clause defining the predicate 'Column Sum Query'/1 for querying this predicate. This clause will provide test data to 'Column Sum'/3 and return the calculated sum.

3.1.2. Create the ‘Column Sum Query’/1 predicate.

The section describes the creation of the clause to be used to query the ‘Column Sum’/3 clause built in the previous section. This is step 4 of the “Column Sum” example.

Step 4: Create the "Column Sum Query" clause.

The first two sub-steps of step 4 create the clause for the ‘Column Sum Query’/1 predicate with an empty body.

Step 4.1.

DO: Create a new clause named "Column Sum Query" with 1 argument in program (window) "Column Sum".

BY: Select the "Create Clause" option in the popup menu for the program. Remove 2 arguments from the default 3 arguments. Edit the default name ("Column Sum") to change it to "Column Sum Query".

Step 4.2: Create the argument term for the "Column Sum Query" clause.

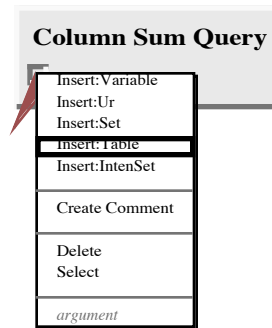


Figure A1-3.4. 1: Interaction to create a table representation of a set.

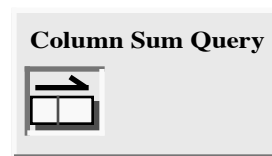


Figure A1-3.4. 2: Result of creating a new table.

The term in the argument of this query clause gives shape to the result of the query. The query clause that we will create provides a table of data and the name of a column to be summed in that table. The result of the query is a table of two rows: one row shows the test data and the other row shows the name of the column being summed and the sum for that column. The next step creates an empty table which is a set of one 2-tuple row. The interaction and result are shown in Figure A1-3.4. 1 and Figure A1-3.4. 2.

Step 4.3.

DO: Create a new term table in the argument of the “Column Sum Query” clause.

BY: Select the "Insert:Table" option in the popup menu for the argument.

The first column of this table is used to label the parts of the answer. The first row will hold the input “data”. Step 4.4 creates an ur constant “data” in the first column of the first row. Step 4.5 creates a variable which will corefer with the input table that will be placed in the first argument of the literal to be created in the query clause.

Step 4.4: Create a new ur constant of value “data” in the left term_table_cell.

Step 4.5.

DO: Create a new variable in the left term_table_cell.

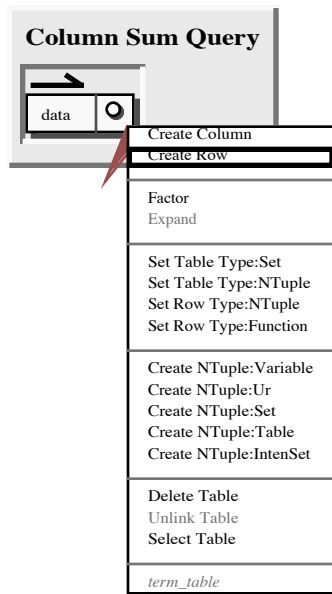


Figure A1-3.4. 3: Interaction to add a second row to a term table.

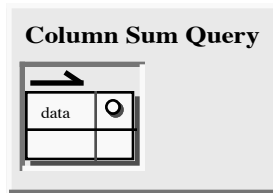


Figure A1-3.4. 4: Result of adding a second row to a term table.

BY: Select the "Insert:Variable" option in the popup menu for the left `term_table_cell`.

Step 4.6 creates a second row in the result table. This row will hold the sum of the selected column of the input data. The interaction and result of this step are shown in

Figure A1-3.4. 3 and Figure A1-3.4. 4.

Step 4.6.

DO: Create a row at the bottom of the `term_table`.

BY: Select the "Create Row" option in the popup menu for the `term_table`.

We want to ensure that the "data" row is displayed before (above) the

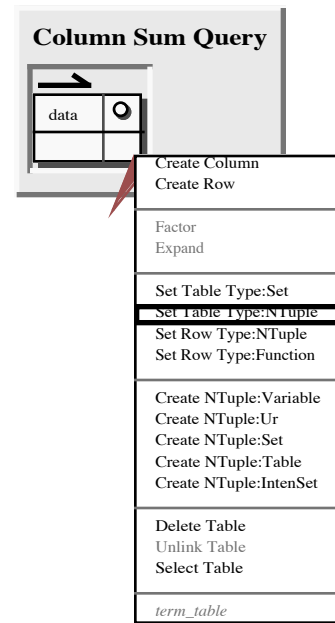


Figure A1-3.4. 5: Interaction to set the type of a term table to "NTuple".

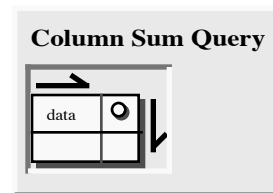


Figure A1-3.4. 6: Result of setting the type of a term table to "NTuple".

"total" row, so we will make the result table an ordered table—one which is an N-tuple of rows instead of a set of rows. The interaction and result of this step are shown in Figure A1-3.4. 5 and Figure A1-3.4. 6.

Step 4.7.

DO: Set the "table type" of the `term_table` to N-tuple.

BY: Select the "Set Table Type:NTuple" option in the popup menu for the `term_table`.

Now we put the "label" for the second row in place. The interaction for starting this is shown in Figure A1-3.4. 7. The resulting table is shown in Figure A1-3.4. 8.

Step 4.8: Create a new ur constant of value "Calculation" in the `term_table_cell`.

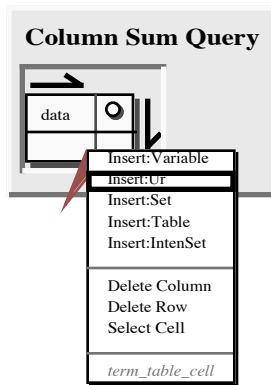


Figure A1-3.4. 7: Interaction to insert an ur constant in a term table cell.

The calculation from the ‘Column Sum’/3 predicate will be shown using a function term table with the column titles from the input data. It will have one row with the first column being the ur constant “Total” and the second column being the calculated total. The first step in constructing this new term table is

shown in Figure A1-3.4. 9 and Figure A1-3.4. 10.

Step 4.9.

DO: Create a new term table (call it term table 2) in the term_table_cell of row 2, column 2 of the table.

BY: Select the "Insert:Table" option in the popup menu for this term_table_cell.

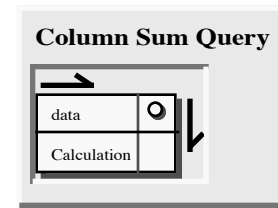


Figure A1-3.4. 8: Result of inserting “Calculation” in a term table cell.

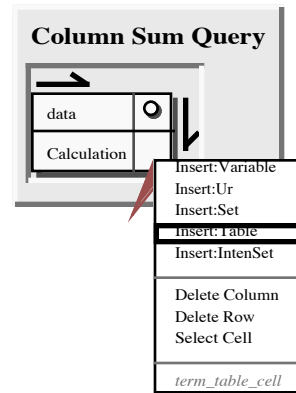


Figure A1-3.4. 9: Interaction to create term table in a term table cell.

The next step change the row type of the new term table to “Function”, as shown in Figure A1-3.4. 11 and Figure A1-3.4. 12.

Step 4.10.

DO: Set the "row type" of term_table 2 to function.

BY: Select the "Set Row Type:Function" option in the popup menu for this term_table.

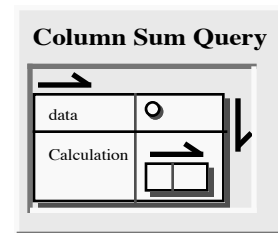


Figure A1-3.4. 10: Result of creating a term table in a term table cell.

Step 4.11 creates a variable in the first column of the function table, as shown in Figure A1-3.4. 13.

Step 4.11.

DO: Create a new variable in the term_table_cell (1,1) of table 2.

BY: Select the "Insert:Variable" option in the popup menu for this term_table_cell.

Steps 4.12, 4.13, and 4.14 create two more variables and the ur constant “Total” in the function table, as shown in Figure A1-3.4. 14.

Step 4.12.

DO: Create a new variable in the term_table_cell (1,2) of table 2.

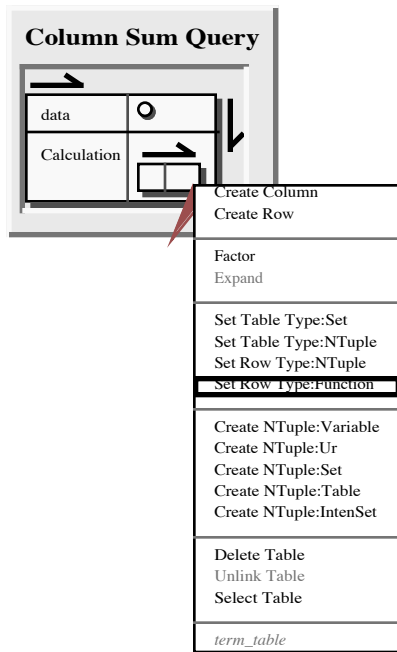


Figure A1-3.4. 11: Interaction to set the row type of a term table to "Function".

Sum Query'/1 predicate is creating the 'Column Sum'/3 literal with appropriately filled in arguments.

Step 4.15: Create the literal of the "Column Sum Query" clause.

The first sub-step is to create the literal with empty arguments.

Step 4.15.1.

DO: Create a new literal with name Column Sum and 3 arguments in the clause.

BY: Select the "Create Literal" option in the popup menu for the clause.

Having created the 'Column Sum'/3 literal, we now fill in its arguments. In the first argument we create a function table, put domain values in the "header" (the first row) of the table and fill in two "body" rows of data. First we'll create the empty function table, then we'll explain what a function table represents in SPARCL.

Step 4.15.2 creates a term table, shown in Figure A1-3.4. 15.

Step 3.4.15.2.

DO: Create a new term table (call it table 3) in the first argument of the literal.

BY: Select the "Insert:Variable" option in the popup menu for this term_table_cell.

Step 4.13: Create a new ur constant of value "Total" in the term_table_cell (2,1) of table 2.

Step 4.14.

DO: Create a new variable in the term_table_cell (2,2) of table 2.

BY: Select the "Insert:Variable" option in the popup menu for this term_table_cell.

The remaining

major part of creating the clause defining the 'Column

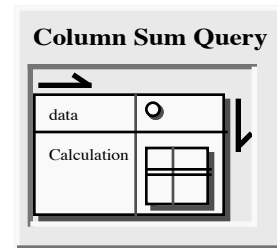


Figure A1-3.4. 12: Result of setting the row type of a term table to "Function".

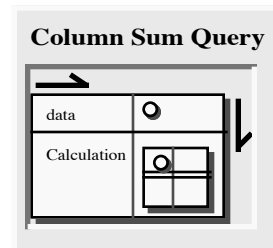


Figure A1-3.4. 13: Result of creating a variable in the first column of the header of the function

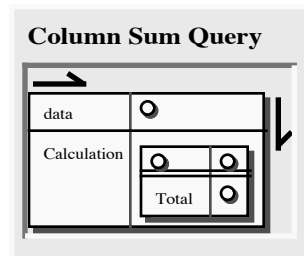


Figure A1-3.4. 14: Complete result term in the argument of the clause defining the 'Column Sum Query'/1 predicate.

BY: Select the "Insert:Table" option in the popup menu for the argument.

The next step takes this N-tuple-row-type table and converts it to a function-row-type table, shown in Figure A1–3.4. 16.

Step 3.4.15.3.
DO: Set the "row type" of term_table 3 to function.
BY: Select the "Set Row Type:Function" option in the popup menu for this term_table.

This conversion has added a row to the table so that it has a "header" row and one "body" row. The double horizontal line between the first and second row is the indication in the visual representation that this table is now a function table. It has the appearance of a table with named columns. A function table representation can be used for a set of "functions", where all of the functions in this set are finite and have the same domain values. These domain values are the column names of the function table. The column names are placed at the top of the table, which we will do now in steps 4.15.4 and 4.15.5. The result is shown in Figure A1–3.4. 17.

Step 4.15.4. Create a new ur constant of value "Item" in term_table_cell (1,1) of table 3.
Step 4.15.5: Create a new ur constant of value "Value" in term_table_cell (1,2) of table 3.

Each "body" row plus the "header" row represents a function. A function in SPARCL is a set of ordered pairs (2-tuples), where the first elements of the ordered pairs are the domain values and the second elements are the range values. Further, no two ordered pairs in a "function" set have the same first element.

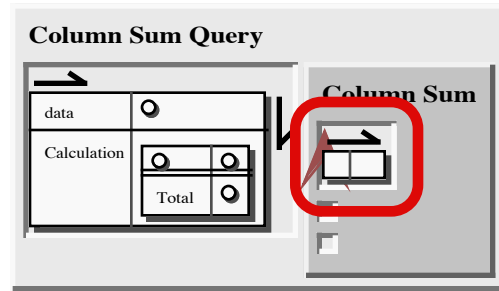


Figure A1–3.4. 15: Result of creating a term table in first argument of literal.

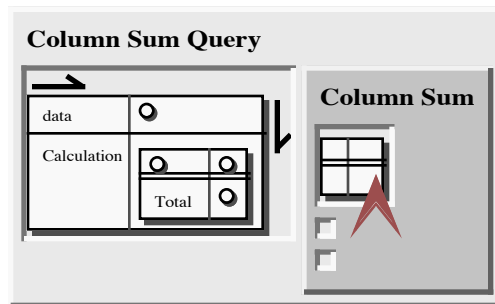


Figure A1–3.4. 16: Result of setting the literal term table row type to "Function."

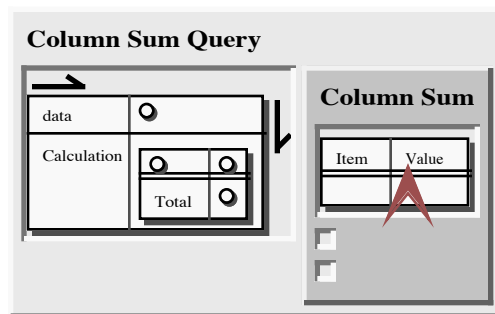


Figure A1–3.4. 17: Result of adding the ur constants "Item" and "Value" as column headers to the function table in the literal.

The range values of a function are placed here by steps 4.15.6 and 4.15.7. The result is shown in Figure A1–3.4. 18.

- Step 3.4.15.6: Create a new ur constant of value “Sheet Music” in term_table_cell (2,1) of table 3.
- Step 3.4.15.7: Create a new ur constant of value “25.00” in term_table_cell (2,2) of table 3.

Column Sum	
Item	Value
Sheet Music	25.00

Figure A1–3.4. 18: Result of adding the ur constants “Sheet Music” and “25.00” to the function table in the literal.

In step 4.15.8 we add another row to the table. “Adding” a row always puts it at the end of the table. For a table which is a set of rows (as opposed to an N-tuple of rows), this is semantically sufficient for available editing operations because the order of the rows doesn't make any semantic difference. But for a table which is an N-tuple of rows, this can be an awkward interface because there is no tool to place a new row at a particular place in the order of rows. To place a new row somewhere other than at the end, one would have to delete the rows after the place where the new row is wanted, add the new row, then rebuild the deleted rows.

- Step 4.15.8.
- DO: Create a row at the bottom of term_table 3.
- BY: Select the "Create Row" option in the popup menu for this term_table.

Next we fill in this last "body" row.

- Step 4.15.9: Create a new ur constant of value “Arm Chair & Stool” in term_table_cell (3,1) of table 3.
- Step 3.4.15.10: Create a new ur constant of value “75.00” in term_table_cell (3,2) if table 3.

This completes the test data table. Now we put variables in the other two arguments. The result of this (overlain by a linking interaction) is shown in Figure A1–3.4. 19.

- Step 4.15.11.
- DO: Create a new variable in the second argument of the literal.
- BY: Select the "Insert:Variable" option in the popup menu for this argument.
- Step 4.15.12.
- DO: Create a new variable in the third argument of the literal.
- BY: Select the "Insert:Variable" option in the popup menu for this argument.

Rather than put the same constant in the literal twice, we write it once and put a variable at the other location. Then we connect these two terms. In step 4.15.13, the title of the second column of the test data is connected to the variable in the literal argument for the name of the column to be summed. The interaction is shown in Figure A1–3.4. 19. The result (overlain by the next interaction) is shown in Figure A1–3.4. 20.

Step 3.4.15.13.

DO: Create a coreference link including the variable in the second argument of the literal and “Value”.

BY: With "connect tool" as the current tool, depress the mouse button while the cursor is over the variable and drag the cursor until it's over “Value”, then release the mouse button.

The entire table of test data is linked to the variable in the second column of the first row. The interaction is shown in Figure A1–3.4. 20 and the result is shown in Figure A1–3.4. 21.

Step 4.16.

DO: Create a coreference link including the variable in cell (1,2) of table 1 and all of table 2.

BY: With "connect tool" as the current tool, depress the mouse button while the cursor is over the variable and drag the cursor until it's over the table, then release the mouse button.

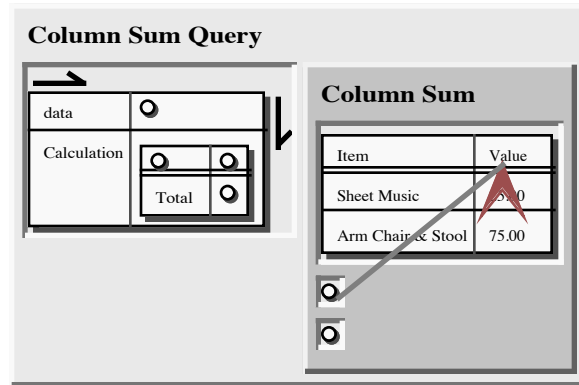


Figure A1–3.4. 19: Interaction for linking “Value” to the argument for the name of the column to be summed.

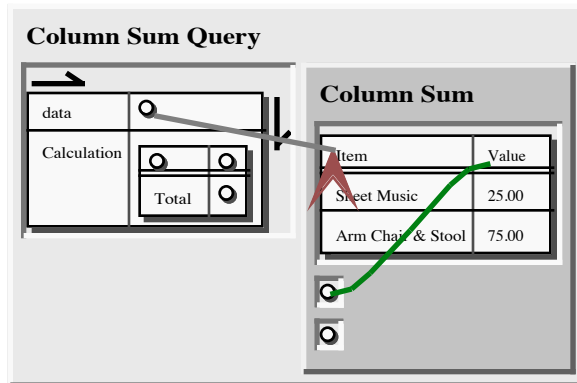


Figure A1–3.4. 20: Result of linking “Value”.

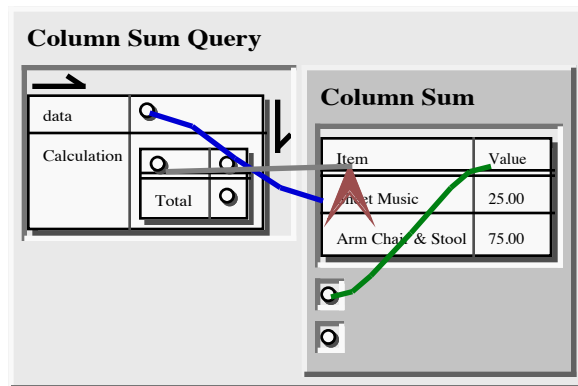


Figure A1–3.4. 21: Result of linking the test data table, and interaction for linking “Item”.

The name of the column not being summed, “Item”, is linked to the head of the

appropriate column in the result table's function table. The interaction is shown in Figure A1-3.4. 21 and the result is shown in Figure A1-3.4. 22.

Step 4.17.

DO: Create a coreference link including the variable in cell (1,1) of table 2 and "Item".

BY: With "connect tool" as the current tool, depress the mouse button while the cursor is over the variable and drag the cursor until it's over "Item", then release the mouse button.

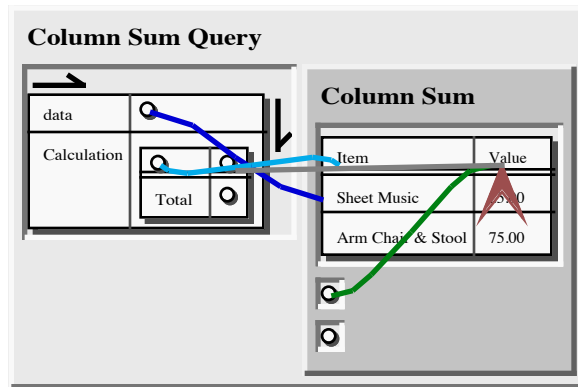


Figure A1-3.4. 22: Result of linking "Item" to the to the head of the result data column *not* being summed, and interaction for linking "Value" to the result data column for the sum.

The name of the column being summed, "Value", is linked to the head of the appropriate column in the result table's function table. The interaction is shown in Figure A1-3.4. 22 and the result is shown in Figure A1-3.4. 23.

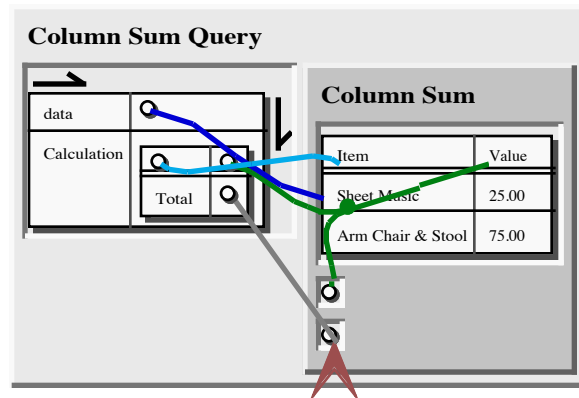


Figure A1-3.4. 23: Result of linking "Value" to the result data column for the sum, and interaction for linking the column sum result variable to the result data column sum result value variable.

Step 4.18.

DO: Create a coreference link including the variable in cell (1,2) of table 2 and "Value".

BY: With "connect tool" as the current tool, depress the mouse button while the cursor is over the variable and drag the cursor until it's over "Value", then release the mouse button.

The final connection is from the result variable for the 'Column Sum'/3 literal and the result sum variable in the second column of the "Total" row of the function table of the result table. The interaction is shown in Figure A1-3.4. 23 and the result (plus the next interaction) is shown in Figure A1-3.4. 24.

Step 3.4.19.

DO: Create a coreference link including the variable in cell (2,2) of table 2 and the variable in the third argument of the literal.

BY: With "connect tool" as the current tool, depress the mouse button while the cursor is over one

variable and drag the cursor until it's over the other variable, then release the mouse button.

Finally, we will run the test query. The query interaction is shown in Figure A1-3.4. 24. The result of the query is a 2-tuple with the name of the query clause as the first element and the bound value of the argument to the query clause as the second element. The query interaction is shown in Figure A1-3.4. 24 and the result is in Figure A1-3.4. 25.

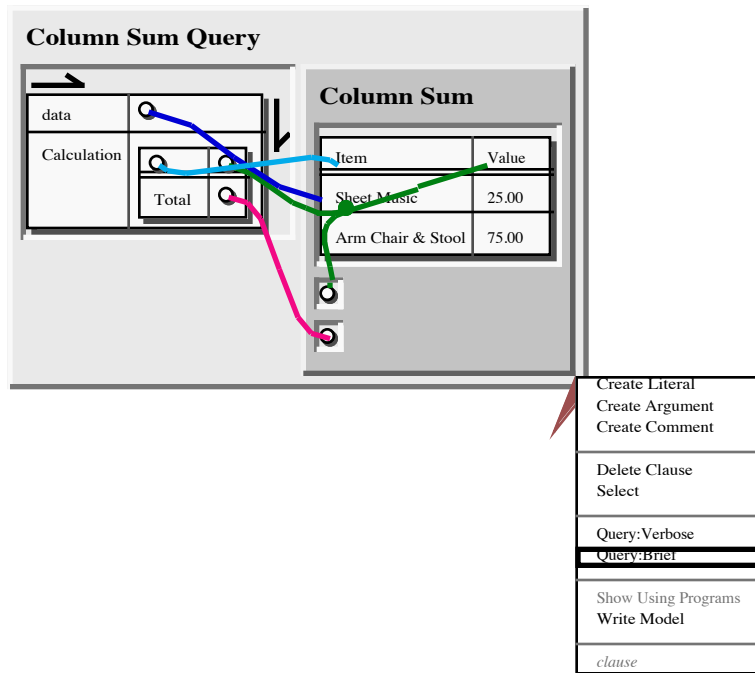


Figure A1-3.4. 24: Interaction to query a 'Column Sum Query'/1 clause.

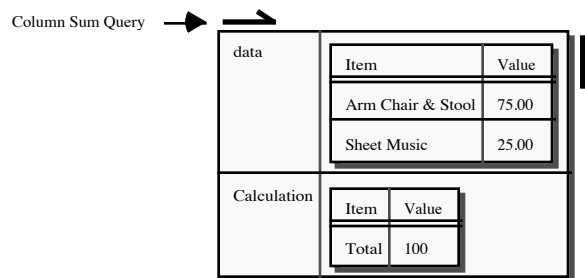


Figure A1-3.4. 25: Result of the query of the clause for 'Column Sum Query'/1.

Step 3.5.

DO: Execute a query of the "Column Sum Query" clause, with tracing information suppressed.

BY: Select the "Query:Brief" option in the popup menu for the clause.

This query result in Figure A1-3.4. 25 shows the sum of the items in the test data is 100.

1. Exercise for section 3

1. Implement a "Tournament Scores" predicate which scores players based on their performance in multiple rounds of a tournament. In preparation for implementing this predicate (in part 3, below), you are first asked to implement "Maximal Range Value" in part 1, and then you are asked to

implement "Maximal Pairs" in part 2. These predicates will make use of ordered pairs (2-tuples), partitioned sets, intensional sets (and intensional multisets), function tables, "*DELAY*" specifications, and the fails/2 meta-predicate (a meta-predicate is a predicate which takes a literal as an argument and invokes the given literal).

Write a "Maximal Range Value" predicate of two arguments.

1. Write a program for the predicate "Maximal Range Value" with two arguments. The first argument is a set of pairs, with the second elements (the range values) being numbers. The second argument is a number which is maximal with respect to the range values of this set, i.e. a number such that no range value is greater than it.

1. HINT:

Use a fails/1 literal and use two literals in its argument, a unify/2 literal and a less/2 literal. The unify/2 literal unifies given the pair set with a partitioned set with an ordered pair of two variables in one part and nothing in the other part. The less/2 literal compares the second argument term with the second element of the ordered pair of the unify/2 set.

2. Write a "Maximal Range Value Query" predicate of no arguments which tests the "Maximal Range Value/2" predicate by giving it the set "{a=>3,b=>4,c=>5}" and 5 as the maximal range value.

2. Write a "Maximal Pair" predicate of three arguments and a test predicate.

1. The "Maximal Pairs" predicate has a set of ordered pairs as its first argument, its second argument is the set of maximally range-valued pairs among the first argument's set. Thus, for the first argument set "{a=>2, b=>1, c=>2, d=>0}", the second argument set of maximal pairs is "{a=>2, c=>2}".

1. HINT:

This is implemented by a single clause, plus a "helper" predicate. The second argument to "Maximal Pairs" should be an intensional set. The template is an ordered pair

(which will become the maximal ordered pairs). The body has two literals. One of these is a unify/2 literal which unifies a 2-part partitioning with a variable. The first part contains a 2-tuple, the second part is hollow. The second literal is the "helper" predicate, "Maximal Range Value" of two arguments as implemented in part 1 of this exercise. For "Maximal Range Value"/2 to work correctly in "Maximal Pairs", there must be two `"*DELAY*"` specifications for the "Maximal Range Value" predicate of two arguments such that "Maximal Range Value" delays if either argument is a variable (i.e. a delay specification for `"variable=>ignore"` and another for `"ignore=>variable"`).

2. Write a "Maximal Pairs Query" predicate of one argument to test the "Maximal Pairs" predicate. The "Maximal Pairs Query" predicate should return the maximal pairs found by "Maximal Pairs", given the test set `"{a=>2, b=>1, c=>2, d=>0}"`.

3. Write a "Tournament Scores" predicate and a predicate to test it.

1. The "Tournament Scores" predicate has two arguments. The first argument is the tournament rounds scores and the second argument is a function from player to overall score for that player. The tournament rounds scores are a function table, where each column is a different player in the tournament and each row is a set of scores for a round of the tournament. The overall scoring of a player for the tournament is the number rounds in which that player was among those with the highest score.

1. HINT:

The second argument of "Tournament Scores" is an intensional multiset, where the template is a variable (which will be the player) and the body contains two literals. One of these literals is a unify/2 literal which extracts a row from the rounds table. The other literal is a "Maximal Pairs"/2 literal which finds the players with maximal scores for that round. The second argument to this

"Maximal Pairs" literal should be a partitioned set of two parts. One of these parts is an ordered pair with the first element of this pair being connected to the template variable.

2. Write a "Tournament Scores Query" predicate of one argument which returns the overall player scores from "Tournament Scores", given a table of:

"{{a=>1, b=>2, c=>0}, {a=>2, b=>2, c=>1}, {a=>1, b=>0, c=>2}}".

Appendix 2

Source Metrics for the Implementation of SPARCL

This appendix contains details of the measurements of the various parts of the implementation of SPARCL.

Totals for selected columns in Table A2– 1:
 Procedure count: 3667,
 Clause count: 5618,
 Goal count: 23032,
 Size: 191012

Logical source line count = Clause count + Goal count = 28650 logical source lines.

Table A2– 1: Metrics for the modules implementing SPARCL. The modules are sorted by (Halstead) volume.

<i>Program</i>	<i>G</i>	<i>C</i>	<i>P</i>	<i>GC</i>	<i>GP</i>	<i>Vol</i>	<i>S</i>	<i>Voc</i>	<i>UOtr</i>	<i>UOnd</i>	<i>TOtr</i>	<i>TOnd</i>
SPARCL Script	: 1878	434	202	4.3	9.2	154419	14745	1421	28	1393	7399	7346
SPARCL to 3D Model	: 1453	362	203	4.0	7.1	151150	14612	1300	27	1273	7335	7277
SPARCL Dev Env	: 2041	432	281	4.7	7.2	137103	12931	1555	28	1527	6433	6498
SPARCL Display	: 1453	264	197	5.5	7.3	136636	13635	1039	27	1012	6885	6750
SPARCL Log Analysis	: 1603	393	287	4.0	5.5	117633	11765	1023	34	989	5996	5769
SPARCL Interpreter	: 1143	291	197	3.9	5.8	113421	11484	940	33	907	5790	5694
SPARCL QD3D Model	: 739	171	48	4.3	15.3	61061	6908	458	17	441	3410	3498
SPARCL DE Database	: 1064	280	231	3.8	4.6	60384	6543	600	24	576	3340	3203
SPARCL DE Program	: 882	215	127	4.1	6.9	60186	6298	753	22	731	3141	3157
SPARCL Interp Unify	: 803	235	114	3.4	7.0	58094	6437	521	21	500	3322	3115
SPARCL Dialogs	: 679	116	76	5.8	8.9	52973	5621	687	22	665	2810	2811
SPARCL Halstead	: 552	131	98	4.2	5.6	52519	5821	520	22	498	2912	2909
Display Graph	: 299	69	47	4.3	6.3	44950	5012	501	17	484	2466	2546
Graph Averaging Layo	: 468	109	72	4.2	6.5	44313	4910	521	25	496	2473	2437
SPARCL DE Create Obj:	629	140	109	4.4	5.7	41085	4684	437	21	416	2361	2323
vectors	: 425	110	57	3.8	7.4	30363	3855	235	20	215	1942	1913
SPARCL DE Edit Objec:	458	79	66	5.7	6.9	26987	3224	331	16	315	1644	1580
SPARCL Visual Transf	: 344	86	53	4.0	6.4	25770	3141	295	13	282	1625	1516
String Utilities	: 349	93	56	3.7	6.2	23889	2851	333	21	312	1455	1396
SPARCL Linear Transf	: 298	67	46	4.4	6.4	23557	2896	281	17	264	1478	1418
SPARCL Objects Kerne	: 189	155	154	1.2	1.2	22288	2790	254	9	245	1390	1400
My Conversion Suppor	: 372	150	130	2.4	2.8	21998	2635	326	41	285	1302	1333
SPARCL 3D Model Util:	265	58	34	4.5	7.7	21750	2608	324	18	306	1265	1343
SPARCL Inventor Mode:	322	81	36	3.9	8.9	21044	2579	286	16	270	1282	1297
Containment Tree DB	: 218	60	52	3.6	4.1	19678	2597	191	18	173	1302	1295
Graph Utilities	: 330	81	48	4.0	6.8	19539	2439	258	17	241	1265	1174
SPARCL Display Util	: 323	78	49	4.1	6.5	19507	2379	294	21	273	1191	1188
SPARCL DE Factor Tab:	295	58	43	5.0	6.8	19147	2418	242	13	229	1226	1192
SPARCL Readable Line:	309	97	48	3.1	6.4	18353	2304	250	23	227	1200	1104
<i>Program</i>	<i>G</i>	<i>C</i>	<i>P</i>	<i>GC</i>	<i>GP</i>	<i>Vol</i>	<i>S</i>	<i>Voc</i>	<i>UOtr</i>	<i>UOnd</i>	<i>TOtr</i>	<i>TOnd</i>

Table A2– 1 (continued): Metrics for the modules implementing SPARCL. The modules are sorted by (Halstead) volume.

<i>Program</i>	<i>G</i>	<i>C</i>	<i>P</i>	<i>GC</i>	<i>GP</i>	<i>Vol</i>	<i>S</i>	<i>Voc</i>	<i>UOtr</i>	<i>UOnd</i>	<i>TOtr</i>	<i>TOnd</i>
SPARCL DE Object Util:	219	50	18	4.3	12.1	14686	1878	226	14	212	951	927
SPARCL DE Menus :	298	55	42	5.4	7.0	13353	1731	210	13	197	870	861
List Utilities :	168	71	37	2.3	4.5	10755	1577	113	16	97	826	751
portray :	178	43	25	4.1	7.1	10567	1423	172	18	154	715	708
SPARCL Trans Disp Ob:	138	46	27	3.0	5.1	9886	1339	167	13	154	681	658
SPARCL POV Model :	209	44	25	4.7	8.3	9752	1287	191	15	176	620	667
Browser :	123	30	14	4.1	8.7	8562	1161	166	9	157	577	584
SPARCL Interp Kernel :	162	46	31	3.5	5.2	8272	1149	147	14	133	592	557
Arithmetic Utilities :	140	40	27	3.5	5.1	8000	1143	128	22	106	583	560
SPARCL Interaction L :	165	30	28	5.5	5.8	7362	965	198	17	181	483	482
SPARCL Dev Env Util :	127	29	25	4.3	5.0	7255	1012	144	18	126	496	516
Display Lines :	74	19	14	3.8	5.2	4425	669	98	13	85	338	331
Binary Tree :	77	19	10	4.0	7.7	4240	641	98	17	81	316	325
SPARCL Dialog Kernel :	64	7	7	9.1	9.1	3605	523	119	15	104	262	261
SPARCL Menus Kernel :	108	32	32	3.3	3.3	3487	547	83	12	71	277	270
Queue Utilities :	37	22	16	1.6	2.3	2700	463	57	8	49	238	225
Preference Utilities :	78	14	14	5.5	5.5	2556	401	83	13	70	199	202
SPARCL Pgm Wdw Na:	69	13	12	5.3	5.7	2296	376	69	10	59	189	187
File Utilities :	61	14	13	4.3	4.6	2206	350	79	9	70	177	173
SPARCL Pjct&Pgm U :	56	15	15	3.7	3.7	1892	312	67	14	53	159	153
curve_to_lines :	34	9	7	3.7	4.8	1866	316	60	9	51	159	157
SPARCL Picture DB :	30	10	8	3.0	3.7	1405	239	59	10	49	121	118
SPARCL DE Cls DB :	35	7	6	5.0	5.8	1247	213	58	13	45	110	103
garbage_collecting_c :	38	12	10	3.1	3.8	891	180	31	8	23	92	88
SPARCL Output Wdw :	26	5	5	5.2	5.2	870	148	59	14	45	74	74
SPARCL Database Kern:	25	11	11	2.2	2.2	781	144	43	10	33	73	71
Point-Line Distance :	19	5	5	3.8	3.8	742	146	34	6	28	73	73
Picture DB :	22	7	7	3.1	3.1	687	130	39	7	32	65	65
Term Utilities :	17	5	3	3.4	5.6	528	103	35	11	24	53	50
straight :	13	3	3	4.3	4.3	516	100	36	10	26	49	51
Picture DB Utilities :	13	4	3	3.2	4.3	485	98	31	8	23	50	48
Error Handling :	13	3	3	4.3	4.3	388	80	29	9	20	40	40
Clause Utilities :	13	3	3	4.3	4.3	210	46	24	8	16	24	22

<i>Program</i>	<i>G</i>	<i>C</i>	<i>P</i>	<i>GC</i>	<i>GP</i>	<i>Vol</i>	<i>S</i>	<i>Voc</i>	<i>UOtr</i>	<i>UOnd</i>	<i>TOtr</i>	<i>TOnd</i>
----------------	----------	----------	----------	-----------	-----------	------------	----------	------------	-------------	-------------	-------------	-------------

GoalCount=G ClauseCount=C ProcedureCount=P

AvgGoalsPerClause=GC AvgGoalsPerProcedure=GP Volume=Vol

Size=S Vocabulary=Voc UniqueOperators=UOtr

UniqueOperands=UOnd TotalOperators=TOtr TotalOperands=TOnd

Table A2– 2: Metrics for the modules implementing SPARCL, sorted by module name..

<i>Program</i>	<i>G</i>	<i>C</i>	<i>P</i>	<i>GC</i>	<i>GP</i>	<i>Vol</i>	<i>S</i>	<i>Voc</i>	<i>UOtr</i>	<i>UOnd</i>	<i>TOtr</i>	<i>TOnd</i>
Arithmetic Utilities :	140	40	27	3.5	5.1	8000	1143	128	22	106	583	560
Binary Tree :	77	19	10	4.0	7.7	4240	641	98	17	81	316	325
Browser :	123	30	14	4.1	8.7	8562	1161	166	9	157	577	584
Clause Utilities :	13	3	3	4.3	4.3	210	46	24	8	16	24	22
Containment Tree DB :	218	60	52	3.6	4.1	19678	2597	191	18	173	1302	1295
curve_to_lines :	34	9	7	3.7	4.8	1866	316	60	9	51	159	157
Display Graph :	299	69	47	4.3	6.3	44950	5012	501	17	484	2466	2546
Display Lines :	74	19	14	3.8	5.2	4425	669	98	13	85	338	331
Error Handling :	13	3	3	4.3	4.3	388	80	29	9	20	40	40
File Utilities :	61	14	13	4.3	4.6	2206	350	79	9	70	177	173
Graph Averaging Layo :	468	109	72	4.2	6.5	44313	4910	521	25	496	2473	2437
Graph Utilities :	330	81	48	4.0	6.8	19539	2439	258	17	241	1265	1174
garbage_collecting_c :	38	12	10	3.1	3.8	891	180	31	8	23	92	88
List Utilities :	168	71	37	2.3	4.5	10755	1577	113	16	97	826	751
My Conversion Suppor :	372	150	130	2.4	2.8	21998	2635	326	41	285	1302	1333
Picture DB :	22	7	7	3.1	3.1	687	130	39	7	32	65	65
Picture DB Utilities :	13	4	3	3.2	4.3	485	98	31	8	23	50	48
Point-Line Distance :	19	5	5	3.8	3.8	742	146	34	6	28	73	73
Preference Utilities :	78	14	14	5.5	5.5	2556	401	83	13	70	199	202
portray :	178	43	25	4.1	7.1	10567	1423	172	18	154	715	708
Queue Utilities :	37	22	16	1.6	2.3	2700	463	57	8	49	238	225
SPARCL 3D Model Util:	265	58	34	4.5	7.7	21750	2608	324	18	306	1265	1343
SPARCL Database Kern:	25	11	11	2.2	2.2	781	144	43	10	33	73	71
SPARCL DE Cls DB :	35	7	6	5.0	5.8	1247	213	58	13	45	110	103
SPARCL DE Create Obj:	629	140	109	4.4	5.7	41085	4684	437	21	416	2361	2323
SPARCL DE Database :	1064	280	231	3.8	4.6	60384	6543	600	24	576	3340	3203
SPARCL DE Edit Objec:	458	79	66	5.7	6.9	26987	3224	331	16	315	1644	1580
SPARCL DE Factor Tab:	295	58	43	5.0	6.8	19147	2418	242	13	229	1226	1192
SPARCL DE Menus :	298	55	42	5.4	7.0	13353	1731	210	13	197	870	861
SPARCL DE Object Uti :	219	50	18	4.3	12.1	14686	1878	226	14	212	951	927
SPARCL DE Program :	882	215	127	4.1	6.9	60186	6298	753	22	731	3141	3157
SPARCL Dev Env :	2041	432	281	4.7	7.2	137103	12931	1555	28	1527	6433	6498
SPARCL Dev Env Util :	127	29	25	4.3	5.0	7255	1012	144	18	126	496	516
SPARCL Dialog Kernel :	64	7	7	9.1	9.1	3605	523	119	15	104	262	261
SPARCL Dialogs :	679	116	76	5.8	8.9	52973	5621	687	22	665	2810	2811
SPARCL Display :	1453	264	197	5.5	7.3	136636	13635	1039	27	1012	6885	6750
SPARCL Display Util :	323	78	49	4.1	6.5	19507	2379	294	21	273	1191	1188
SPARCL Halstead :	552	131	98	4.2	5.6	52519	5821	520	22	498	2912	2909
SPARCL Interaction L :	165	30	28	5.5	5.8	7362	965	198	17	181	483	482
SPARCL Interp Kernel :	162	46	31	3.5	5.2	8272	1149	147	14	133	592	557
SPARCL Interpreter :	1143	291	197	3.9	5.8	113421	11484	940	33	907	5790	5694
SPARCL Interp Unify :	803	235	114	3.4	7.0	58094	6437	521	21	500	3322	3115
SPARCL Inventor Mode:	322	81	36	3.9	8.9	21044	2579	286	16	270	1282	1297
SPARCL Linear Transf :	298	67	46	4.4	6.4	23557	2896	281	17	264	1478	1418
SPARCL Log Analysis :	1603	393	287	4.0	5.5	117633	11765	1023	34	989	5996	5769
SPARCL Menus Kernel :	108	32	32	3.3	3.3	3487	547	83	12	71	277	270
SPARCL Objects Kerne :	189	155	154	1.2	1.2	22288	2790	254	9	245	1390	1400
SPARCL Output Wdw :	26	5	5	5.2	5.2	870	148	59	14	45	74	74

Table A2– 2 (continued): Metrics for the modules implementing SPARCL, sorted by module name.

SPARCL Pgm Wdw Na:	69	13	12	5.3	5.7	2296	376	69	10	59	189	187
SPARCL Picture DB :	30	10	8	3.0	3.7	1405	239	59	10	49	121	118
SPARCL POV Model :	209	44	25	4.7	8.3	9752	1287	191	15	176	620	667
SPARCL Pjct&Pgm U :	56	15	15	3.7	3.7	1892	312	67	14	53	159	153
SPARCL QD3D Model :	739	171	48	4.3	15.3	61061	6908	458	17	441	3410	3498
SPARCL Readable Line:	309	97	48	3.1	6.4	18353	2304	250	23	227	1200	1104
SPARCL Script :	1878	434	202	4.3	9.2	154419	14745	1421	28	1393	7399	7346
SPARCL Trans Disp Ob:	138	46	27	3.0	5.1	9886	1339	167	13	154	681	658
SPARCL to 3D Model :	1453	362	203	4.0	7.1	151150	14612	1300	27	1273	7335	7277
SPARCL Visual Transf :	344	86	53	4.0	6.4	25770	3141	295	13	282	1625	1516
String Utilities :	349	93	56	3.7	6.2	23889	2851	333	21	312	1455	1396
straight :	13	3	3	4.3	4.3	516	100	36	10	26	49	51
Term Utilities :	17	5	3	3.4	5.6	528	103	35	11	24	53	50
vectors :	425	110	57	3.8	7.4	30363	3855	235	20	215	1942	1913

<i>Program</i>	<i>G</i>	<i>C</i>	<i>P</i>	<i>GC</i>	<i>GP</i>	<i>Vol</i>	<i>S</i>	<i>Voc</i>	<i>UOtr</i>	<i>UOnd</i>	<i>TOtr</i>	<i>TOnd</i>
----------------	----------	----------	----------	-----------	-----------	------------	----------	------------	-------------	-------------	-------------	-------------

GoalCount=G ClauseCount=C ProcedureCount=P

AvgGoalsPerClause=GC AvgGoalsPerProcedure=GP Volume=Vol

Size=S Vocabulary=Voc UniqueOperators=UOtr

UniqueOperands=UOnd TotalOperators=TOtr TotalOperands=TOnd

Appendix 3

PROLOG and LISP Solutions to Programming Problems

This appendix contains the source for the PROLOG and LISP solutions of the *ID3* programming problem and the PROLOG solution of the *WARPLAN* problem.

ID3

PROLOG.

```

/*
Sample query:

:-id3([color, size],
      decision,
      [[color-red, size-big, decision-yes],
       [color-red, size-small, decision-no],
       [color-blue, size-big, decision-yes]],
      Tree
      )

No.1 : Tree = [node(top, size, [node(big, yes), node(small, no)])]

*/

id3(Attributes, Decision, Examples, Tree) :-
    id3_tree(Attributes, [top-Examples], Tree, Decision).

id3_tree(Attributes, ClassifiedExamples,
         [node(Value, SelectedAttribute, Subtree)|OtherNodes], Decision) :-
    choose(ClassifiedExamples, Value-ValueExamples,
           OtherClassifiedExamples),
    heterogeneous_examples(ValueExamples, Decision),
    select_attribute(Attributes, ValueExamples, Decision,
                    SelectedAttribute, SubclassifiedExamples,
                    OtherAttributes),
    id3_tree(OtherAttributes, SubclassifiedExamples, Subtree, Decision),
    id3_tree(Attributes, OtherClassifiedExamples, OtherNodes, Decision).

id3_tree(Attributes, ClassifiedExamples, [node(Value, DecisionValue)|OtherNodes],
         Decision) :-
    choose(ClassifiedExamples, Value-ValueExamples,
           OtherClassifiedExamples),
    homogeneous_examples(ValueExamples, Decision, DecisionValue),
    id3_tree(Attributes, OtherClassifiedExamples, OtherNodes, Decision).

```

```

id3_tree(_, [], [], _).

heterogeneous_examples(ValueExamples, Decision) :-
    choose_trim(ValueExamples, Example, OtherExamples),
    choose(Example, Decision-DV1),
    choose(OtherExamples, OtherExample),
    choose(OtherExample, Decision-DV2),
    DV1 \= DV2.

homogeneous_examples([Example|OtherExamples], Decision, DV) :-
    choose(Example, Decision-DV),
    forall(choose(OtherExamples, OtherExample),
        choose(OtherExample, Decision-DV)).

select_attribute(As, ValueExamples, Decision, Attribute,
    ClassifiedExamples, OAs) :-
    determine_entropies(ValueExamples, Decision, As, AttributeEntropies),
    choose(AttributeEntropies,
        entropy(Attribute, Entropy, ClassifiedExamples),
        OtherAttributeEntropies),
    \+ (choose(OtherAttributeEntropies, entropy(_, OtherEntropy, _)),
        Entropy > OtherEntropy
    ),
    setof(TA, X^Y^choose(OtherAttributeEntropies, entropy(TA, X, Y)), OAs).

determine_entropies(Examples, Decision, [A|OAs],
    [entropy(A, E, ClassifiedExamples)|OtherInfos]) :-
    determine_entropy(A, Examples, Decision, E, ClassifiedExamples),
    determine_entropies(Examples, Decision, OAs, OtherInfos).

determine_entropies(_, _, [], []).

determine_entropy(A, Examples, Decision, E, ClassifiedExamples) :-
    classify_examples(Examples, A, ClassifiedExamples),
    de_attribute_values(ClassifiedExamples, Decision, 0, E).

de_attribute_values([], _, E, E).

de_attribute_values([_Examples|OtherClassifiedExamples], Decision, E0, E1) :-
    de_attribute_value(Examples, Decision, EV),
    EN is E0 + EV,
    de_attribute_values(OtherClassifiedExamples, Decision, EN, E1).

de_attribute_value(Examples, Decision, E) :-
    length(Examples, ExampleCount),
    classify_examples(Examples, Decision, ClassifiedExamples),
    de_attribute_decisions(ClassifiedExamples, ExampleCount, 0, NE),

```

```

E is -1*NE.

de_attribute_decisions([], _, Entropy, Entropy).

de_attribute_decisions([ValueExamplesPair|OtherValueExamplesPairs], ExampleCount,
    EntropyIN, EntropyOUT) :-
    de_attribute_decision(ValueExamplesPair, ExampleCount,
        EntropyIN, EntropyNEXT),
    de_attribute_decisions(OtherValueExamplesPairs, ExampleCount,
        EntropyNEXT, EntropyOUT).

de_attribute_decision(_Examples, OuterExampleCount, EntropyIN, EntropyOUT) :-
    length(Examples, ExamplesCount),
    Fraction is ExamplesCount / OuterExampleCount,
    EntropyPart is Fraction * ln(Fraction),
    EntropyOUT is EntropyIN + EntropyPart.

% short version.
classify_examples(Examples, ClassifyingAttribute, ClassifiedExamples) :-
    setof(V-Es,
        setof(XE,
            (member(XE, Examples),
             member(ClassifyingAttribute-V, XE))
        ),
        Es
    ),
    ClassifiedExamples).

/*
% fast version.
% this version of classify_examples/3 should be much faster than the "setof"
version.

classify_examples(Examples, ClassifyingAttribute, ClassifiedExamples) :-
    classify_examples1(Examples, ClassifyingAttribute, ValueExamplePairs),
    sort(ValueExamplePairs, SortedPairs),
    assemble_value_example_pairs(SortedPairs, ClassifiedExamples).

classify_examples1([], _, []).

classify_examples1([Example|OtherExamples],
    ClassifyingAttribute,
    [Value-Example|OtherValueExamplePairs]) :-
    choose(Example, ClassifyingAttribute-Value),
    classify_examples1(OtherExamples, ClassifyingAttribute,
        OtherValueExamplePairs).

assemble_value_example_pairs([Value-Example|OtherPairs],
    [Value-[Example|OtherValueExamples]
    | OtherClassifiedExamples]) :-

```

```

assemble_value_example_pairs(OtherPairs, Value,
                             OtherValueExamples,
                             OtherClassifiedExamples).

assemble_value_example_pairs([], _, [], []).

assemble_value_example_pairs([Value-Example|OtherPairs],
                             RefValue, ValueExamples,
                             ClassifiedExamples) :-
    (Value = RefValue
    -> ValueExamples = [Example|OtherValueExamples],
        ClassifiedExamples = OtherClassifiedExamples
    ;
    _ = RefValue,
    ValueExamples = [],
    ClassifiedExamples = [Value-[Example|OtherValueExamples]
                        | OtherClassifiedExamples]
    ),
    assemble_value_example_pairs(OtherPairs, Value,
                                OtherValueExamples,
                                OtherClassifiedExamples).

*/

/*
choose([H|_], H).
choose([_|T], X) :-
    choose(T, X).

choose([H|T], H, T).
choose([H|T], X, [H|R]) :-
    choose(T, R).

choose_trim([H|T], H, T).
choose_trim([H|T], X, R) :-
    choose(T, R).

length([], 0).
length([_|T], N) :-
    length(T, K),
    N is K + 1.

*/

```

LISP.

```
(defun find-tree (examples)
  (cond ((constant-decision examples)
        (get-decision-value (get-decision (first examples))))
        (t (split examples))))

(defun constant-decision (examples)
  (if (null examples)
      t
      (constant-decision-util
       (get-decision-value (get-decision (first examples)))
       (rest examples))))

(defun constant-decision-util (value examples)
  (if (null examples)
      t
      (if (equal value (get-decision-value (get-decision (first examples))))
          (constant-decision-util value (rest examples))
          nil)))

;;; split takes a list of examples and determines which attribute to use
;;; to "split" the examples using the entropy calculation across the
;;; attributes in the examples. It returns a list of 2 elements:
;;; (splitting-attribute-name split-examples)
;;; where the split-examples is an assoc list of
;;; (value trimmed-examples) triples,
;;; where the trimmed-examples are those examples having that value, minus
;;; the attribute being used for the split. split-recurse is called by split
;;; to find-tree the examples which have been classified by the "current"
;;; split. split uses a shortcut in selecting the appropriate attribute:
;;; instead of using attribute with the largest information gain, it uses
;;; the attribute with the smallest conditional entropy. This will always
;;; select the same attribute as the largest information gain would have.

(defun split (examples)
  (let ((attribute-names (get-attribute-names
                          (get-attributes (first examples))))
        (best-entropy 10000) ;;; assume that entropy is always < 10000
        (best-attribute-name nil)
        (best-split nil))
    (do ((attribute-name nil)
        (other-attribute-names attribute-names)
        ((null other-attribute-names))
        (setf attribute-name (first other-attribute-names))
        (setf other-attribute-names (rest other-attribute-names))
        (let ((y (split-attribute attribute-name examples)))
          (cond ((< (second y) best-entropy)
                 (setf best-entropy (second y))
                 (setf best-attribute-name (first y))
                 (setf best-split (third y))))))
      (list best-attribute-name (split-recurse best-split))))

(defun split-recurse (splits)
```

```

(if (null splits)
    nil
    (let* ((split (first splits))
           (value (first split))
           (examples (second split)))
      (cons (list value (find-tree examples))
            (split-recurse (rest splits))))))

(defun split-attribute (name examples)
  (let* ((split (classify name examples nil))
         (entropy-split (calculate (length examples) split 0 nil)))
    (list name (first entropy-split) (second entropy-split))))

(defun classify (name examples classes)
  (cond ((null examples) classes)
        (t (classify name
                      (rest examples)
                      (extend classes
                           name
                           (get-attribute-value name (first examples))
                           (trim-example name (first examples)))))))

(defun extend (classes name value example)
  (cond ((null classes)
        (list (list value
                    (list example)
                    (count-decisions (list example) nil))))
        ((equal value (first (first classes)))
         (cons (list value
                     (cons example (second (first classes)))
                     (count-decisions (list example)
                                       (third (first classes))))
               (rest classes)))
        (t (cons (first classes)
                  (extend (rest classes) name value example)))))

(defun count-decisions (examples counts)
  (cond ((null examples) counts)
        (t (count-decisions
            (rest examples)
            (extend-counts
             counts
             (get-decision-value (get-decision (first examples))))))))

(defun extend-counts (counts value)
  (cond ((null counts) (list (list value 1)))
        ((equal value (first (first counts)))
         (cons (list value (+ 1 (second (first counts)))) (rest counts)))
        (t (cons (first counts) (extend-counts (rest counts) value)))))

;;; calculate determines the conditional entropy for the given split and
;;; returns this entropy and the split list with the decision counts removed.

(defun calculate (number-of-examples split sum trimmed-split)
  (if (null split)

```

```

(list sum trimmed-split)
(let* ((class (first split))
      (class-size (length (second class))))
  (calculate number-of-examples
    (rest split)
    (+ sum (/ (* class-size
                  (entropy class-size (third class) 0))
              number-of-examples))
    (cons (list (first class) (second class)) trimmed-split))))

(defun entropy (class-size decision-counts sum)
  (if (null decision-counts)
      sum
      (let* ((decision-count (second (first decision-counts)))
            (ratio (/ (float decision-count) (float class-size)))
            (logratio (- (log2 decision-count)
                          (log2 class-size))))
        (entropy class-size
          (rest decision-counts)
          (- sum (* ratio logratio))))))

(defun get-attributes (example) (rest example))
(defun get-decision (example) (first example))

(defun get-attribute-value (name attributes)
  (attribute-match name attributes))

(defun attribute-match (name attributes)
  (cond ((equal name (first (first attributes))) (second (first attributes)))
        (t (attribute-match name (rest attributes)))))

(defun get-attribute-names (attributes)
  (if (null attributes)
      nil
      (cons (first (first attributes))
        (get-attribute-names (rest attributes)))))

(defun get-decision-value (decision) (second decision))
(defun get-decision-name (decision) (first decision))

(defun trim-example (name example)
  (cond ((equal name (first (first example))) (rest example))
        (t (cons (first example) (trim-example name (rest example))))))

```


WARPLAN programming problem.

There is no LISP version of this problem. The WARPLAN algorithm was designed to fit logic programming semantics well, and as a consequence it is awkward to implement in LISP.

PROLOG.

WARPLAN was written by David H. D. Warren. This version is taken from the book [Coelho&Cotta 1988]. The variables have been descriptively renamed.

There are facts and actions. Goals are the facts which are the desired state. A state is a conjunction of facts, expressed using the '&' operator. A plan is a sequence of actions, expressed using the '=>' operator. A fact *F* is "preserved" by an action *A* if and only if *F* is not added by *A* and *F* is true in a state resulting from *A* whenever *F* is true in the state the application of *A*.

WARPLAN solves problems in a some "world". The state of a world is a conjunction of facts. A world is defined by defining the set of possible actions, the set of "always" true facts for the world, the set of "impossible" conjunctions of facts (which defines a set of impossible states of the world), and the "given" initial state of the world.

An action is defined by a set of preconditions, deletions, and additions.

Schema for defining an action:

```
can(Action, PreconditionFacts1) :- PreconditionBody1.  
...  
can(Action, PreconditionFactsJ) :- PreconditionBodyJ.  
del(DeleteFact1, Action) :- DeleteBody1.  
...  
del(DeleteFactN, Action) :- DeleteBodyN.  
add(AddFact1, Action) :- AddBody1.  
...  
add(AddFactK, Action) :- AddBodyK.
```

Preconditions of an action are a conjunction of facts, defined by the `can/2` procedure: `can(Action, Facts)`. A deletion of a fact (i.e. a fact which is not preserved by an action) is defined by the `del/2` procedure: `del(Fact, Action)`. Multiple facts may be deleted by the same action, this is indicated by having multiple `del/2` facts for the same action. The addition of a fact is defined by the `add/2` procedure: `add(Fact, Action)`. Multiple facts may be added by the same action, this is indicated by having multiple `add/2` facts for the same action.

An "always" true fact is indicated by: `always(Fact)`. An "impossible" conjunction of facts is indicated by: `imposs(Facts)`. A "given" fact F for initial state S is indicated by: `given(S, F)`.

```
:- op(700, xfy, &).
:- op(650, yfx, =>).

plan(Goals, Given, PlanOUT) :-
    plan(Goals, true, Given, PlanOUT).

plan(FirstGoal & OtherGoals, ProtectedFactsIn, PlanIn, PlanOut) :-
    !,
    solve(FirstGoal, ProtectedFactsIn, PlanIn, ProtectedFactsNext, PlanInterim),
    plan(OtherGoals, ProtectedFactsNext, PlanInterim, PlanOut).

plan(Goal, ProtectedFacts, PlanIn, PlanOut) :-
    solve(Goal, ProtectedFacts, PlanIn, _, PlanOut).

solve(Goal, ProtectedFactsIn, PlanIn, ProtectedFactsIn, PlanIn) :-
    (def(always),
     always(Goal)
    )
    ;
    (def(Goal),
     call(Goal)
    ).

solve(Goal, ProtectedFactsIn, PlanIn, ProtectedFactsOut, PlanIn) :-
    holds(Goal, PlanIn),
    and(Goal, ProtectedFactsIn, ProtectedFactsOut).

solve(Goal, ProtectedFactsIn, PlanIn, Goal&ProtectedFactsIn, PlanOut) :-
    add(Goal, AddingAction),
    achieve(Goal, AddingAction, ProtectedFactsIn, PlanIn, PlanOut).
```

```

achieve(_, ActionToAchieve, ProtectedFacts, PlanIn,
        PlanInterim => ActionToAchieve) :-
    preserves(ActionToAchieve, ProtectedFacts),
    can(ActionToAchieve, Goals),
    consistent(Goals, ProtectedFacts),
    plan(Goals, ProtectedFacts, PlanIn, PlanInterim),
    preserves(ActionToAchieve, ProtectedFacts).

achieve(Goal, ActionToAchieve, ProtectedFacts,
        PlanIn => RetracedAction, PlanOut => RetracedAction) :-
    preserved(Goal, RetracedAction),
    retrace(ProtectedFacts, RetracedAction, ProtectedFactsNext),
    achieve(Goal, ActionToAchieve, ProtectedFactsNext, PlanIn, PlanOut),
    preserved(Goal, RetracedAction).

/* Possible changes due to intervening variable bindings require the repeated
preserved/2 check.
*/

holds(Fact, _ => LastAction) :-
    add(Fact, LastAction).

holds(Fact, PrecedingActions => LastAction) :-
    !,
    preserved(Fact, LastAction),
    holds(Fact, PrecedingActions),
    preserved(Fact, LastAction).

holds(Fact, PrecedingActions) :-
    given(PrecedingActions, Fact).

preserves(Action, Fact & OtherFacts) :-
    preserved(Fact, Action),
    preserves(Action, OtherFacts).

preserves(_, true).

preserved(Fact, Action) :-
    numbertvars(Fact & Action, 0, _),
    del(Fact, Action),
    !,
    fail.

preserved(_, _).

retrace(ProtectedFactsIn, Action, ProtectedFactsOut) :-
    can(Action, Facts),
    retracel(ProtectedFactsIn, Action, Facts, ProtectedFactsInterim),
    consistent(Facts, ProtectedFactsInterim), % LLS addition.
    append_facts(Facts, ProtectedFactsInterim, ProtectedFactsOut).

```

```

retracel(ProtectedFactIn & OtherProtectedFactsIn, Action, Facts,
         ProtectedFactsOut) :-
    add(FactAddedByAction, Action),
    equiv(ProtectedFactIn, FactAddedByAction),
    !,
    retracel(OtherProtectedFactsIn, Action, Facts, ProtectedFactsOut).

retracel(ProtectedFactIn & OtherProtectedFactsIn, Action, Facts,
         ProtectedFactsOut) :-
    elem(Fact, Facts),
    equiv(ProtectedFactIn, Fact),
    !,
    retracel(OtherProtectedFactsIn, Action, Facts, ProtectedFactsOut).

retracel(ProtectedFactIn & OtherProtectedFactsIn, Action, Facts,
         ProtectedFactIn & OtherProtectedFactsOut) :-
    retracel(OtherProtectedFactsIn, Action, Facts, OtherProtectedFactsOut).

retracel(true, _, _, true).

consistent(Facts, ProtectedFacts) :-
    numbervars(Facts & ProtectedFacts, 0, _),
    imposs(ImpossibleFacts),
    % The double "not" avoids new bindings.
    \+(\!(intersect(Facts, ImpossibleFacts))),
    implied(ImpossibleFacts, Facts & ProtectedFacts),
    !,
    fail.

consistent(_, _).

and(NewFact, Facts, Facts) :-
    elem(OldFact, Facts),
    equiv(NewFact, OldFact),
    !.

and(Fact, FactsIn, Fact & FactsIn).

append_facts(Fact1 & OtherFacts1, Facts2, Fact1 & OtherFactsCombined) :-
    !,
    append_facts(OtherFacts1, Facts2, OtherFactsCombined).

append_facts(Fact, Facts, Fact & Facts).

elem(Fact, FirstFacts & _) :-
    elem(Fact, FirstFacts).

elem(Fact, _ & OtherFacts) :-
    !,
    elem(Fact, OtherFacts).

elem(Fact, Fact).

```

```

implied(FirstImpliedFacts & OtherImpliedFacts, AcceptedFacts) :-
    !,
    implied(FirstImpliedFacts, AcceptedFacts),
    implied(OtherImpliedFacts, AcceptedFacts).

implied(ImpliedFact, AcceptedFacts) :-
    elem(ImpliedFact, AcceptedFacts).

implied(ImpliedFact, _) :-
    def(ImpliedFact),
    call(ImpliedFact).

intersect(Facts1, Facts2) :-
    elem(CommonFact, Facts1),
    elem(CommonFact, Facts2).

not_equal(Term1, Term2) :-
    \+(Term1 = Term2),
    \+(Term1 = '$VAR'(_)),
    \+(Term2 = '$VAR'(_)).

equiv(Fact1, Fact2) :-
    \+(nonequiv(Fact1, Fact2)).

/* nonequiv(Fact1, Fact2) must be used carefully, as any variables in Fact1 and
Fact2 are bound to dummy symbols of the form '$VAR'(_) by numbervars. Thus,
nonequiv/2 is used as the argument of a not/1 call.
*/

nonequiv(Fact1, Fact2) :-
    numbervars(Fact1 & Fact2, 0, _),
    Fact1 = Fact2,
    !,
    fail.

nonequiv(_, _).

```

Coelho&Cotta 1988 *Prolog By Example: How to Learn, Teach, and Use It* by Helder Coelho and
Jose' C. Cotta, Springer-Verlag, 1988

Appendix 4

Interaction Log Data

This appendix contains detailed data and analyses for the interaction logs.

Figure 1 is a histogram of the frequencies of interactions grouped by their durations. The duration of an interaction is the time from the recording of the interaction to the time of the recording of the next interaction. These durations are more precisely the interaction intervals.

Figure 1: Histogram of durations in seconds for all interactions.

Each "x" is a count of 10 .

```

0|xxxxxx (69)
1|xxxxxxxxxxxxxxxx (154)
2|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (283)
3|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (573)
4|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (519)
5|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (480)
6|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (430)
7|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (373)
8|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (239)
9|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (231)
10|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (195)
11|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (161)
12|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (144)
13|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (123)
14|xxxxxxxxxx (82)
15|xxxxxx (69)
16|xxxxxx (66)
17|xxxxxx (61)
18|xxxxx (57)
19|xxxx (47)
20|xxxxx (53)
21|xxxx (44)
22|xxx (34)
23|xx (25)
24|xxx (35)
25|xx (20)
26|xx (24)
27|xx (25)
28|xx (21)
29|xx (20)
30| (9)
31|x (16)
32|x (12)
33|xx (24)

```

Figure 1(continued): Histogram of durations in seconds for all interactions.

34	x	(14)
35	x	(13)
36		(9)
37		(8)
38	x	(11)
39	x	(10)
40		(8)
41	x	(12)
42		(9)
43	x	(10)
44	x	(12)
45		(7)
46	x	(13)
47		(5)
48		(4)
49		(7)
50		(8)
51		(9)
52		(4)
53		(4)
54		(3)
57		(6)
58		(2)
59		(8)
60		(6)
61		(1)
62		(6)
63		(3)
64		(7)
66		(3)
67		(5)
68		(8)
69		(6)
70		(6)
71		(4)
72		(5)
73		(1)
74		(1)
75		(1)
76		(3)
77		(6)
78		(5)
79		(5)
80		(4)
81		(1)
82		(4)
83		(2)
84		(2)
85		(1)

Figure 1(continued): Histogram of durations in seconds for all interactions.

86		(3)
87		(1)
88		(5)
89		(2)
90		(3)
91		(2)
92		(4)

Skipping 79 data points spread over 36312 values.

Table 1 presents duration statistics for each action major and minor type. The median durations for the various types are generally much less than the average durations, and the standard deviations are typically greater than the average durations. This indicates that for most interaction types, the durations are skewed toward the lower values. The maximum duration is typically many times larger than the average.

There are some types of interactions which we expect were generally part of a sequence of operations, where the user followed the interaction with another interaction as rapidly as the system allowed. Most of the Argument Ops, Clause Ops, Set Ops, Term Table Cell Ops, Literal Ops, Ur Ops, Partition Ops, Variable Ops, Term Table Ops, and NTuple Ops interactions are of this type, with the exception of the Clause Ops/Create Comment and Clause Ops/Query:Brief interactions. Creating a comment was generally followed by a significant period of time editing the comment being created. Executing a query could take an arbitrarily long period of time. The median durations for these “sequence” interaction types vary between 6 and 10 seconds, with the exception of the NTuple Ops interactions, which have a median of 15 seconds.

Table 1 (part 1 of 3): Duration statistics for all participants organized by action type. Interaction "types" which account for less than 4 interactions are "suppressed". (The "===" entries are only for formatting, they indicate the end of a major action type.) Each line reports statistics for the durations of the interactions of the indicated type. "Count" is the number of interactions of that type, "Total" is the total of the durations of these interactions, "Avg" is the average (arithmetic mean) of these durations, "Dev" is the standard deviation of these durations, "Med" is the median value of these durations, "Min" is the minimum value of these durations, and "Max" is the maximum value of these durations.

<i>Interaction Type (Major/Minor)</i>	<i>Count</i>	<i>Total</i>	<i>Avg</i>	<i>Dev</i>	<i>Med</i>	<i>Min</i>	<i>Max</i>
1. script_control	3227	37468	11.6	26.2	6	0	853
1. step	3113	34583	11.1	25.4	6	0	853
2. runvb	38	1579	41.6	47.9	20	0	208
3. runbf	35	1101	31.5	41.4	14	0	190
4. registered_scripts	26	164	6.3	8.1	5	0	43
5. done	15	41	2.7	2.9	2	0	12
6. ===							
2. general_tool	459	11506	25.1	142.2	7	1	2861
1. activate	334	10229	30.6	166.3	6	1	2861
2. close_edit	125	1277	10.2	8.4	7	2	50
3. ===							
3. window	363	6929	19.1	55.7	9	0	935
1. activate	363	6929	19.1	55.7	9	0	935
2. ===							
4. Argument Ops	212	2190	10.3	12.8	7	1	148
1. Insert:Variable	149	1391	9.3	14.1	7	1	148
2. Insert:Ur	31	362	11.7	5	11	5	31
3. Delete	16	179	11.2	8.2	9	3	35
4. Insert:Set	7	74	10.6	5.7	10	3	22
5. Insert:Table	6	75	12.5	7.5	11	4	25
6. **** SKIP 1 types. ****	3						
7. ===							
5. Clause Ops	194	2855	14.7	21.3	11	1	236
1. Create Literal	82	1295	15.8	13.6	13	3	92
2. Create Argument	51	483	9.5	7.2	6	2	39
3. Query:Brief	41	694	16.9	36.3	8	1	236
4. Create Comment	10	308	30.8	33.2	22	4	128
5. Select	4	19	4.8	3.7	4	2	11
6. **** SKIP 2 types. ****	6						
7. ===							
6. connect_tool	115	1248	10.9	28	5	0	289
1. activate	115	1248	10.9	28	5	0	289
2. ===							

Table 1 (continued, part 2 of 3): Duration statistics for all participants organized by action type.

1. File	105	41183	392.2	3531.6	28	0	36404
1. Record Comment about SPARCL...	74	40274	544.2	4197.3	36	11	36404
2. Quit	17	390	22.9	84.4	0	0	360
3. Open Program...	7	269	38.4	18.6	33	18	70
4. **** SKIP 5 types. ****	7						
5. ====							
2. link	77	1113	14.5	18.3	8	2	119
3. Program Ops	54	662	12.3	9.7	9	3	57
1. Create Clause	53	653	12.3	9.8	9	3	57
2. ====							
4. Clause Name Ops	41	438	10.7	5.4	10	2	24
1. Edit Clause Name	36	399	11.1	5.6	10	2	24
2. Select	5	39	7.8	2.5	8	5	11
3. ====							
5. Windows	38	215	5.7	10.7	1	0	43
1. Exercise 1.2	5	5	1	0	1	1	1
2. Parent	4	46	11.5	18.2	1	1	43
3. Geometry Example	4	40	10	9.4	15	1	23
4. Family Relationships	4	4	1	0	1	1	1
5. Exercise 1.1	4	3	0.8	0.4	1	0	1
6. **** SKIP 11 types. ****	17						
7. ====							
6. Tutorial Scripts	36	559	15.5	24.2	8	1	119
1. 3.0.Application of SPARCL	6	81	13.5	5.7	14	4	23
2. 2.1.Data objects	4	22	5.5	2.9	8	1	8
3. **** SKIP 11 types. ****	26						
4. ====							
7. Literal Ops	29	251	8.7	7.2	6	2	37
1. Create Argument	17	123	7.2	3.9	6	3	18
2. Delete	10	123	12.3	10	9	3	37
3. **** SKIP 1 types. ****	2						
4. ====							
8. DISALLOWED	29	173	6	6.5	3	0	22
1. File	21	146	7	7	5	0	22
2. **** SKIP 3 types. ****	8						
3. ====							
9. Set Ops	24	324	13.5	9.6	10	4	45
1. Insert:Ur	10	117	11.7	7.3	10	6	31
2. Create NTuple:Set	4	72	18	7.6	24	9	27
3. Create NTuple:Ur	4	45	11.2	5.7	9	7	21
4. **** SKIP 5 types. ****	6						
5. ====							

Table 1 (continued, part 3 of 3): Duration statistics for all participants organized by action type.

1. Term Table Cell Ops	19	194	10.2	5.2	8	5	26
1. Insert:Ur	15	137	9.1	3.8	8	5	17
2. **** SKIP 2 types. ****	4						
3. ===							
2. START LOGGING	17	1539	90.5	97.2	67	8	442
1. K... C...	5	368	73.6	10.8	69	62	88
2. **** SKIP 7 types. ****	12						
3. ===							
3. Ur Ops	15	368	24.5	31.3	9	2	92
1. Delete	8	146	18.2	27.9	8	6	92
2. Edit Ur Item	4	103	25.8	31	12	2	79
3. **** SKIP 1 types. ****	3						
4. ===							
4. Partition Ops	12	104	8.7	2.7	8	6	16
1. Delete Partition	5	41	8.2	2.2	8	6	12
2. **** SKIP 4 types. ****	7						
3. ===							
5. Variable Ops	12	74	6.2	5	6	1	22
1. Unlink Variable	7	29	4.1	1.6	4	1	6
2. Delete Variable	5	45	9	6.5	6	5	22
3. ===							
6. Edit	9	279	31	38.6	6	1	107
1. Undo Most Recent	5	163	32.6	32.5	11	2	77
2. **** SKIP 4 types. ****	4						
3. ===							
7. Term Table Ops	7	121	17.3	13.8	8	5	40
1. **** SKIP 5 types. ****	7						
2. ===							
8. NTuple Ops	7	101	14.4	5.8	15	6	25
1. **** SKIP 5 types. ****	7						
2. ===							
9. **** SKIP 4 types. ****	8						

The following histograms (Figure 2, Figure 3, Figure 4, and Figure 5) show the frequencies of the types of interactions found in the usability study participant logs. The four histograms show different collections of the interactions.

Figure 2: Histogram of interaction type frequencies. Frequencies calculated by major action type for all types of interactions for the entire time spent on the study.

Each "x" is a count of 100 .

1:script_control	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	(3218)
2:general_tool	xxxx	(458)
3>window	xxx	(360)
4:Argument Ops	xx	(212)
5:Clause Ops	x	(194)
6:connect_tool	x	(115)
7:File	x	(105)
8:link		(77)
9:Program Ops		(54)
10:Clause Name Ops		(41)
11:Windows		(38)
12:Tutorial Scripts		(36)
13:Literal Ops		(29)
14:DISALLOWED		(29)
15:Set Ops		(24)
16:Term Table Cell Ops		(19)
17:START LOGGING		(17)
18:Ur Ops		(15)
19:Variable Ops		(12)
20:Partition Ops		(12)
21:Edit		(9)
22:Term Table Ops		(7)
23:NTuple Ops		(7)
24:Specify Preferences		(3)

Skipping 5 data points spread over 3 values.

Figure 3: Histogram of interaction type frequencies. Frequencies calculated by major^minor action type for all types of interactions for the entire time spent on the study. (The major action type's have been abbreviated.)

Each "x" is a count of 100 .

1:scrcnt^step	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (3106)
2:wnd^activate	xxx (360)
3:gnrt^activate	xxx (333)
4:ArgOp^Insert:Variabl	x (149)
5:gnrt^close_edit	x (125)
6:cnnt^activate	x (115)
7:ClsOp^Create Literal	(82)
8:link	(77)
9:Fl^Record Comment ab	(74)
10:PrgOp^Create Clause	(53)
11:ClsOp^Create Argumen	(51)
12:ClsOp^Query:Brief	(41)
13:scrcnt^runvb	(38)
14:ClsNmOp^Edit Clause	(36)
15:scrcnt^runbf	(35)
16:ArgOp^Insert:Ur	(31)
17:scrcnt^registered_sc	(24)
18:LtrOp^Create Argumen	(17)
19:Fl^Quit	(17)
20:START LOGGING	(17)
21:ArgOp^Delete	(16)
22:TrmTblCllOp^Insert:U	(15)
23:scrcnt^done	(15)
24:DIS^File ^New Progra	(11)
25:StOp^Insert:Ur	(10)
26:LtrOp^Delete	(10)
27:ClsOp^Create Comment	(10)
28:UrOp^Delete	(8)
29:VrbOp^Unlink Variabl	(7)
30:Fl^Open Program...	(7)
31:ArgOp^Insert:Set	(7)
32:TtrScr^3.0.Applicati	(6)
33:ArgOp^Insert:Table	(6)
34:Wnd^Exercise 1.2	(5)
35:VrbOp^Delete Variabl	(5)
36:PrtOp^Delete Partiti	(5)
37:Edt^Undo Most Recent	(5)
38:ClsNmOp^Select	(5)
39:Wnd^Parent	(4)
40:Wnd^Geometry Example	(4)
41:Wnd^Family Relations	(4)
42:Wnd^Exercise 1.1	(4)
43:UrOp^Edit Ur Item	(4)
44:TtrScr^2.1.Data obje	(4)
45:StOp^Create NTuple:U	(4)
46:StOp^Create NTuple:S	(4)
47:DIS^File ^Open Progr	(4)
48:ClsOp^Select	(4)

Figure 3 (continued): Histogram of interaction type frequencies

49:Wnd^Parent Query	(3)
50:Wnd^Exercise 2.1	(3)
51:UrOp^Create NTuple:U	(3)
52:TtrScr^Run the Tutor	(3)
53:TtrScr^3.1.b.Exercis	(3)
54:TtrScr^2.4.Procedura	(3)
55:TtrScr^2.3.Declarati	(3)
56:TtrScr^1.4.Declarati	(3)
57:TrmTblOp^Set Table T	(3)
58:TrmTblCllOp^Insert:T	(3)
59:SpcPrf^User Name...	(3)
60:PrtOp^Insert:Ur	(3)
61:DIS^Specify Preferen	(3)
62:DIS^Program Ops^Crea	(3)
63:CmmOp^Edit	(3)
64:ClsOp^Query:Verbose	(3)
65:ClsOp^Delete Clause	(3)
66:ArgOp^Create Comment	(3)

Skipping 68 data points spread over 53 values.

Figure 4: Histogram of interaction type frequencies. Frequencies calculated by major action type for selected types of interactions for the entire time spent on the study. The excluded interaction types are: script_control^_, ^activate, 'Tutorial Scripts'^_, DISALLOWED^_, and 'START LOGGING'. This basically selects those interactions which occurred during work on exercises.

Each "x" is a count of 10 .

1:Argument Ops	xxxxxxxxxxxxxxxxxxxxxxx (212)
2:Clause Ops	xxxxxxxxxxxxxxxxxxxxxxx (194)
3:general_tool	xxxxxxxxxxxxx (125)
4:File	xxxxxxxxxxxxx (105)
5:link	xxxxxxx (77)
6:Program Ops	xxxxxx (54)
7:Clause Name Ops	xxxxx (41)
8:Windows	xxx (38)
9:Literal Ops	xx (29)
10:Set Ops	xx (24)
11:Term Table Cell Ops	x (19)
12:Ur Ops	x (15)
13:Variable Ops	x (12)
14:Partition Ops	x (12)
15:Edit	(9)
16:Term Table Ops	(7)
17:NTuple Ops	(7)
18:Specify Preferences	(3)

Skipping 5 data points spread over 3 values.

Figure 5: Histogram of interaction type frequencies. Frequencies calculated by major^minor action type for selected types of interactions for the entire time spent on the study. The excluded interaction types are: script_control^_, ^activate, 'Tutorial Scripts'^_, DISALLOWED^_, and 'START LOGGING'. This basically selects those interactions which occurred during work on exercises.

Each "x" is a count of 10 .

1:ArgOp^Insert:Variabl	xxxxxxxxxxxxxxx	(149)
2:gnrt^close_edit	xxxxxxxxxxxxxxx	(125)
3:ClsOp^Create Literal	xxxxxxx	(82)
4:link	xxxxxxx	(77)
5:Fl^Record Comment ab	xxxxxxx	(74)
6:PrgOp^Create Clause	xxxxx	(53)
7:ClsOp^Create Argumen	xxxxx	(51)
8:ClsOp^Query:Brief	xxxx	(41)
9:ClsNmOp^Edit Clause	xxx	(36)
10:ArgOp^Insert:Ur	xxx	(31)
11:LtrOp^Create Argumen	x	(17)
12:Fl^Quit	x	(17)
13:ArgOp^Delete	x	(16)
14:TrmTblCllOp^Insert:U	x	(15)
15:StOp^Insert:Ur	x	(10)
16:LtrOp^Delete	x	(10)
17:ClsOp^Create Comment	x	(10)
18:UrOp^Delete		(8)
19:VrbOp^Unlink Variabl		(7)
20:Fl^Open Program...		(7)
21:ArgOp^Insert:Set		(7)
22:ArgOp^Insert:Table		(6)
23:Wnd^Exercise 1.2		(5)
24:VrbOp^Delete Variabl		(5)
25:PrtOp^Delete Partiti		(5)
26:Edt^Undo Most Recent		(5)
27:ClsNmOp^Select		(5)
28:Wnd^Parent		(4)
29:Wnd^Geometry Example		(4)
30:Wnd^Family Relations		(4)
31:Wnd^Exercise 1.1		(4)
32:UrOp^Edit Ur Item		(4)
33:StOp^Create NTuple:U		(4)
34:StOp^Create NTuple:S		(4)
35:ClsOp^Select		(4)
36:Wnd^Parent Query		(3)
37:Wnd^Exercise 2.1		(3)
38:UrOp^Create NTuple:U		(3)
39:TrmTblOp^Set Table T		(3)
40:TrmTblCllOp^Insert:T		(3)
41:SpcPrf^User Name...		(3)
42:PrtOp^Insert:Ur		(3)
43:CmmOp^Edit		(3)
44:ClsOp^Query:Verbose		(3)
45:ClsOp^Delete Clause		(3)
46:ArgOp^Create Comment		(3)

Skipping 49 data points spread over 40 values.

Table 2 gives the frequencies with which the various types of interactions occurred among the seven participants of the usability study. The interaction types are hierarchical. For instance, consider the most common interaction “minor” type (with a count of 177). The action portion of the type is a “step” subtype of the “script_control” type. There were 3226 interactions of the “script_control” type, and 3112 interactions of the “step” subtype of the “script_control” type. The “script_control:step” interaction occurred with “<< (Please read the Script Commentary) >>” in the “Next Step” field of the control window and “Rules - extending the Parents program” as the current script. There were 1065 interactions of the “script_control:step” action type with “<< (Please read the Script Commentary) >>” in the “Next Step” window. Of these interactions, there were 177 with “Rules - extending the Parents program” as the current script.

Interaction types which account for less than four interactions have been “skipped” in this table to improve its readability.

Table 2 (part 1 of 18): Interaction type counts grouped by major action type, minor action type, first argument, and second argument. Interaction "types" which account for less than 4 interactions are "suppressed".

script_control	3226	
step	3112	
<< (Please read the Script Commentary) >>	1065	
Rules - extending the Parents program		177
This script discusses *how* SPARCL works.		164
Set types discussion		96
This script creates clauses for querying the paren		81
Create the literal of the "Column Sum" clause.		64
Discuss term matching.		58
The declarative meaning of SPARCL		48
Term types discussion		48
General comments about SPARCL and the tutorial scr		46
Create a program which describes the "parent" rela		41
This script describes what SPARCL programs mean.		35
The procedural meaning of SPARCL		32
Discussion of simple geometry representation.		32
Create a program of six clauses. These clauses des		26
Presentation of an application of SPARCL		20
Create the literal of the "Column Sum Query" claus		18
Chapter of tutorial detailing SPARCL representatio		15
Create the "Column Sum" clause.		12
Tutorial introduction to SPARCL		8
Rules Exercise (2.1)		8
Introduction to SPARCL		8
Rules Exercise (1.2)		7
Exercise 1.1		6
Exercise 3.1		5
Create the argument term for the "Column Sum Query		5
Create the comments and variables of the arguments		4
**** SKIP 1 types. ****		1
Create a variable.	63	
Create a literal with two arguments.		63

Table 2 (part 2 of 18): Interaction type counts

Close the current edit "box" for program "Parent Q	63	
Create a new ur constant of value Pam in the argum		7
Create a new comment "'Who is Pam's child?'" in th		7
Create a new comment "'Who is Liz's parent?'" in t		7
Create a new comment "'Who are two people related		7
Create a new ur constant of value Liz in the argum		6
Create a new comment "'Who is a grandparent of Jim		6
Create a new ur constant of value Jim in the argum		5
Create a new ur constant of value Bob in the argum		4
Create a new ur constant of value Ann in the argum		4
Create a new comment "'Who is a common parent of A		4
**** SKIP 2 types. ****		6
Close the current edit "box" for program "Parent".	60	
Create a new ur constant of value Bob in the argum		6
Create a new ur constant of value Ann in the argum		6
Create a new ur constant of value Tom in the argum		5
Create a new ur constant of value Tom in the argum		5
Create a new ur constant of value Pat in the argum		5
Create a new ur constant of value Pam in the argum		5
Create a new ur constant of value Liz in the argum		5
Create a new ur constant of value Bob in the argum		5
Create a new ur constant of value Bob in the argum		5
Create a new ur constant of value Bob in the argum		5
Create a new ur constant of value Pat in the argum		4
Create a new ur constant of value Jim in the argum		4
Use existing arguments in an existing object.	52	
Use existing arguments in an existing object.		25
Create a clause with arguments Tom and Liz in prog		6
Create a clause with arguments Bob and Ann in prog		6
Create a clause with arguments Tom and Bob in prog		5
Create a clause with arguments Pat and Jim in prog		5
Create a clause with arguments Bob and Pat in prog		5
Create a literal with two arguments.	47	
Create a "query" clause with two arguments and two		26
Create a "Parent Query" clause for asking the ques		7
Create a "Parent Query" clause for asking the ques		7
Create a "Parent Query" clause for asking the ques		7
Make program "Parent Query" the front window.	35	
This script creates clauses for querying the paren		35
Close the current edit "box" for program "Column S	33	
Create the literal of the "Column Sum" clause.		4
Create a new ur constant of value data in the term		4
Create a new ur constant of value + in the argumen		4
Create a new comment "Function Table" in the argum		4
**** SKIP 10 types. ****		17
Stop establishing arguments.	30	
Use existing arguments in an existing object.		25
Add arguments to an existing object.		5
Create an ur object.	30	
Create a literal with two arguments.		30
Make program "Predecessor" the front window.	22	
Rules - extending the Parents program		15
This script discusses *how* SPARCL works.		7

Table 2 (part 3 of 18): Interaction type counts

Close program "Predecessor".	18	
Rules - extending the Parents program		15
**** SKIP 1 types. ****		3
Create a "query" clause with two arguments and two	15	
Create a "Parent Query" clause for asking the ques		7
Create a "Parent Query" clause for asking the ques		5
**** SKIP 1 types. ****		3
Open the program in the file ":Predecessor".	13	
Rules - extending the Parents program		8
This script discusses *how* SPARCL works.		5
Open the program in the file ":Parent".	13	
Rules - extending the Parents program		8
This script discusses *how* SPARCL works.		5
Create a new literal with name Parent and 2 argume	11	
Create a literal with two arguments.		11
Create a new argument in the clause with ID Parent	10	
Add arguments to an existing object.		10
Add arguments to an existing object.	10	
Create a clause with arguments Pam and Bob in prog		5
Add arguments to an existing object.		5
Introduction to SPARCL	9	
Chapters of the SPARCL tutorial.		9
Chapters of the SPARCL tutorial.	9	
Tutorial introduction to SPARCL		5
(start)		4
Term types discussion	8	
Data objects section		8
Set types discussion	8	
Data objects section		8
Rules Exercise (2.1)	8	
Data objects section		8
Rules Exercise (1.2)	8	
Rules - extending the Parents program		8
Rules - extending the Parents program	8	
Introduction to SPARCL		7
**** SKIP 1 types. ****		1
Open the program in the file ":Sex - long".	8	
Rules - extending the Parents program		8
Open the program in the file ":Set Types".	8	
Set types discussion		8
Open the program in the file ":Geometry Example".	8	
Discussion of simple geometry representation.		8
Make program "Set Types" the front window.	8	
Set types discussion		8
Make program "Geometry Example" the front window.	8	
Discussion of simple geometry representation.		8
Discussion of simple geometry representation.	8	
Data objects section		8
Create a program which describes the "parent" rela	8	
Introduction to SPARCL		7
**** SKIP 1 types. ****		1
Create a new program (and window) named "Exercise	8	
Rules Exercise (2.1)		8

Table 2 (part 4 of 18): Interaction type counts

Create a new program (and window) named "Exercise 1.1"	8	6
**** SKIP 1 types. ****		2
Create a new literal with name Parent and 2 arguments.	8	
Create a literal with two arguments.		8
Close program "Sex - table".	8	
Rules - extending the Parents program		8
Close program "Parent Query".	8	
Create a program which describes the "parent" relation		8
Close program "Parent Query RESULT".	8	
Create a program which describes the "parent" relation		8
Close program "Geometry Example".	8	
Discussion of simple geometry representation.		8
Close program "Exercise 1.1".	8	
Exercise 1.1		6
**** SKIP 1 types. ****		2
Open the program in the file ":Term Types".	7	
Term types discussion		7
Open the program in the file ":Predecessor, restriction"	7	
Rules - extending the Parents program		7
Open the program in the file ":Offspring".	7	
Rules - extending the Parents program		7
Open the program in the file ":Family Relationship"	7	
Rules - extending the Parents program		7
Open the program in the file ":Fallible Socrates".	7	
This script discusses *how* SPARCL works.		7
Make program "Term Types" the front window.	7	
Term types discussion		7
Make program "Sex - long" the front window.	7	
Rules - extending the Parents program		7
Make program "Offspring" the front window.	7	
Rules - extending the Parents program		7
Make program "Family Relationships" the front window.	7	
Rules - extending the Parents program		7
Make program "Fallible Socrates" the front window.	7	
This script discusses *how* SPARCL works.		7
Execute a query of clause with ID Parent Query:40,	7	
This script creates clauses for querying the parent		7
Execute a query of clause with ID Parent Query:25,	7	
This script creates clauses for querying the parent		7
Execute a query of clause with ID Parent Query:13,	7	
This script creates clauses for querying the parent		7
Execute a query of clause with ID Parent Query:1,	7	
This script creates clauses for querying the parent		7
Execute a query of clause with ID Fallible Socrates	7	
This script discusses *how* SPARCL works.		7
Edit the value of ur with ID Parent Query:10 to "Pam"	7	
Create a new ur constant of value Pam in the argument		7
Edit the value of comment with ID Parent Query:4 to	7	
Create a new comment "Who is Pam's child?" in the		7
Edit the value of comment with ID Parent Query:29	7	
Create a new comment "Who are two people related		7

Table 2 (part 5 of 18): Interaction type counts

Edit the value of comment with ID Parent Query:16	7	
Create a new comment ""Who is Liz's parent?"" in t	7	7
Create a new variable in the argument with ID Pare	7	
Create a variable.	7	7
Create a new variable in the argument with ID Pare	7	
Create a variable.	7	7
Create a new variable in the argument with ID Pare	7	
Create a variable.	7	7
Create a new variable in the argument with ID Pare	7	
Create a "Parent Query" clause for asking the ques	7	7
Create a new variable in the argument with ID Pare	7	
Create a "Parent Query" clause for asking the ques	7	7
Create a new variable in the argument with ID Pare	7	
Create a "Parent Query" clause for asking the ques	7	7
Create a new variable in the argument with ID Pare	7	
Create a variable.	7	7
Create a new variable in the argument with ID Pare	7	
Create a "Parent Query" clause for asking the ques	7	7
Create a new ur in the argument with ID Parent Que	7	
Create a new ur constant of value Pam in the argum	7	7
Create a new ur in the argument with ID Parent Que	7	
Create a new ur constant of value Liz in the argum	7	7
Create a new ur constant of value Pam in the argum	7	
Create an ur object.	7	7
Create a new ur constant of value Liz in the argum	7	
Create an ur object.	7	7
Create a new program (and window) named "Exercise	7	
Rules Exercise (1.2)	7	7
Create a new literal with name Parent and 2 argume	7	
Create a literal with two arguments.	7	7
Create a new literal with name Parent and 2 argume	7	
Create a literal with two arguments.	7	7
Create a new literal with name Parent and 2 argume	7	
Create a literal with two arguments.	7	7
Create a new comment in the clause with ID Parent	7	
Create a new comment ""Who is a grandparent of Jim	7	7
Create a new comment in the clause with ID Parent	7	
Create a new comment ""Who are two people related	7	7
Create a new comment in the clause with ID Parent	7	
Create a new comment ""Who is Liz's parent?"" in t	7	7
Create a new comment in the clause with ID Parent	7	
Create a new comment ""Who is Pam's child?"" in th	7	7
Create a new comment ""Who is Pam's child?"" in th	7	
Create a "Parent Query" clause for asking the ques	7	7
Create a new comment ""Who is Liz's parent?"" in t	7	
Create a "Parent Query" clause for asking the ques	7	7
Create a new comment ""Who is a grandparent of Jim	7	
Create a "query" clause with two arguments and two	7	7
Create a new comment ""Who are two people related	7	
Create a "Parent Query" clause for asking the ques	7	7
Create a new clause named "Parent Query" with 2 ar	7	
Create a "Parent Query" clause for asking the ques	7	7

Table 2 (part 6 of 18): Interaction type counts

Create a new clause named "Parent Query" with 1 ar	7	
Create a "Parent Query" clause for asking the ques	7	7
Create a new clause named "Parent Query" with 1 ar	7	
Create a "query" clause with two arguments and two	7	7
Create a new clause named "Parent Query" with 1 ar	7	
Create a "Parent Query" clause for asking the ques	7	7
Create a coreference link including variable with	7	
Create a "Parent Query" clause for asking the ques	7	7
Create a coreference link including variable with	7	
Create a "Parent Query" clause for asking the ques	7	7
Create a coreference link including variable with	7	
Create a "Parent Query" clause for asking the ques	7	7
Create a coreference link including variable with	7	
Create a "Parent Query" clause for asking the ques	7	7
Create a coreference link including variable with	7	
Create a "Parent Query" clause for asking the ques	7	7
Create a "Parent Query" clause for asking the ques	7	
This script creates clauses for querying the paren	7	7
Create a "Parent Query" clause for asking the ques	7	
This script creates clauses for querying the paren	7	7
Create a "Parent Query" clause for asking the ques	7	
This script creates clauses for querying the paren	7	7
Close program "Sex - long".	7	
Rules - extending the Parents program	7	7
Close program "Set Types".	7	
Set types discussion	7	7
Close program "Offspring".	7	
Rules - extending the Parents program	7	7
Close program "Family Relationships".	7	
Rules - extending the Parents program	7	7
Close program "Fallible Socrates".	7	
This script discusses *how* SPARCL works.	7	7
Close program "Exercise 1.2".	7	
Rules Exercise (1.2)	7	7
This script creates clauses for querying the paren	6	
Create a program which describes the "parent" rela	6	6
Open the program in the file ":Sex - table".	6	
Rules - extending the Parents program	6	6
Open the program in the file ":Predecessor "How" E	6	
This script discusses *how* SPARCL works.	6	6
Open the program in the file ":Predecessor "How" E	6	
This script discusses *how* SPARCL works.	6	6
Make program "Sex - table" the front window.	6	
Rules - extending the Parents program	6	6
Make program "Predecessor "How" Example" the front	6	
This script discusses *how* SPARCL works.	6	6
General comments about SPARCL and the tutorial scr	6	
Tutorial introduction to SPARCL	6	6
Exercise 1.1	6	
Create a program which describes the "parent" rela	6	6
Execute a query of clause with ID Parent Query:78,	6	
This script creates clauses for querying the paren	6	6

Table 2 (part 7 of 18): Interaction type counts

Execute a query of clause with ID Parent Query:59,	6	
This script creates clauses for querying the paren		6
Edit the value of ur with ID Parent:6 to "Bob".	6	
Create a new ur constant of value Bob in the argum		6
Edit the value of ur with ID Parent:24 to "Ann".	6	
Create a new ur constant of value Ann in the argum		6
Edit the value of ur with ID Parent:17 to "Tom".	6	
Create a new ur constant of value Tom in the argum		6
Edit the value of ur with ID Parent Query:56 to "J	6	
Create a new ur constant of value Jim in the argum		6
Edit the value of ur with ID Parent Query:23 to "L	6	
Create a new ur constant of value Liz in the argum		6
Edit the value of comment with ID Parent Query:43	6	
Create a new comment "Who is a grandparent of Jim		6
Create a program of six clauses. These clauses des	6	
Create a program which describes the "parent" rela		6
Create a new variable in the argument with ID Pare	6	
Create a "query" clause with two arguments and two		6
Create a new ur in the argument with ID Parent:4.	6	
Create a new ur constant of value Bob in the argum		6
Create a new ur in the argument with ID Parent:15.	6	
Create a new ur constant of value Tom in the argum		6
Create a new ur in the argument with ID Parent Que	6	
Create a new ur constant of value Jim in the argum		6
Create a new ur constant of value Tom in the argum	6	
Create a clause with arguments Tom and Liz in prog		6
Create a new ur constant of value Jim in the argum	6	
Create an ur object.		6
Create a new ur constant of value Bob in the argum	6	
Create a clause with arguments Pam and Bob in prog		6
Create a new program (and window) named "Parent Qu	6	
This script creates clauses for querying the paren		6
Create a new literal with name Parent and 2 argume	6	
Create a literal with two arguments.		6
Create a new clause in program "Parent" with its t	6	
Create a clause with arguments Bob and Pat in prog		6
Create a new clause in program "Parent" with its t	6	
Create a clause with arguments Bob and Ann in prog		6
Create a new clause in program "Parent" with its t	6	
Create a clause with arguments Tom and Liz in prog		6
Create a coreference link including variable with	6	
Create a "Parent Query" clause for asking the ques		5
**** SKIP 1 types. ****		1
Create a clause with arguments Tom and Liz in prog	6	
Create a program of six clauses. These clauses des		6
Create a clause with arguments Tom and Bob in prog	6	
Create a program of six clauses. These clauses des		6
Create a clause with arguments Pat and Jim in prog	6	
Create a program of six clauses. These clauses des		6
Create a clause with arguments Bob and Pat in prog	6	
Create a program of six clauses. These clauses des		6
Create a clause with arguments Bob and Ann in prog	6	
Create a program of six clauses. These clauses des		6

Table 2 (part 8 of 18): Interaction type counts

Create a "Parent Query" clause for asking the ques	6	
This script creates clauses for querying the paren		6
Create a "Parent Query" clause for asking the ques	6	
This script creates clauses for querying the paren		6
Close program "Term Types".	6	
Term types discussion		6
Close program "Parent".	6	
Create a program which describes the "parent" rela		6
This script discusses *how* SPARCL works.	5	
Introduction to SPARCL		4
**** SKIP 1 types. ****		1
Open the program in the file ":Line Segments".	5	
Discuss term matching.		5
Make program "Predecessor "How" Ex, 2" the front w	5	
This script discusses *how* SPARCL works.		5
Make program "Line Segments" the front window.	5	
Discuss term matching.		5
Edit the value of ur with ID Parent:5 to "Pam".	5	
Create a new ur constant of value Pam in the argum		5
Edit the value of ur with ID Parent:30 to "Pat".	5	
Create a new ur constant of value Pat in the argum		5
Edit the value of ur with ID Parent:29 to "Bob".	5	
Create a new ur constant of value Bob in the argum		5
Edit the value of ur with ID Parent:23 to "Bob".	5	
Create a new ur constant of value Bob in the argum		5
Edit the value of ur with ID Parent:18 to "Liz".	5	
Create a new ur constant of value Liz in the argum		5
Edit the value of ur with ID Parent:12 to "Bob".	5	
Create a new ur constant of value Bob in the argum		5
Edit the value of ur with ID Parent:11 to "Tom".	5	
Create a new ur constant of value Tom in the argum		5
Edit the value of ur with ID Column Sum temp:45 to	5	
Create a new ur constant of value data in the term		5
Data objects section	5	
Chapter of tutorial detailing SPARCL representatio		5
Create the comments and variables of the arguments	5	
Create the "Column Sum" clause.		5
Create the argument term for the "Column Sum Query	5	
Create the "Column Sum Query" clause.		5
Create the "Column Sum Query" clause.	5	
Presentation of an application of SPARCL		5
Create a new variable in the argument with ID Pare	5	
Create a "query" clause with two arguments and two		5
Create a new variable in the argument with ID Pare	5	
Create a variable.		5
Create a new variable in the argument with ID Pare	5	
Create a variable.		5
Create a new variable in the argument with ID Pare	5	
Create a variable.		5
Create a new ur in the term_table_cell with ID Col	5	
Create a new ur constant of value data in the term		5
Create a new ur in the argument with ID Parent:9.	5	
Create a new ur constant of value Tom in the argum		5

Table 2 (part 9 of 18): Interaction type counts

Create a new ur in the argument with ID Parent:33.	5	
Create a new ur constant of value Pat in the argum		5
Create a new ur in the argument with ID Parent:3.	5	
Create a new ur constant of value Pam in the argum		5
Create a new ur in the argument with ID Parent:28.	5	
Create a new ur constant of value Pat in the argum		5
Create a new ur in the argument with ID Parent:27.	5	
Create a new ur constant of value Bob in the argum		5
Create a new ur in the argument with ID Parent:22.	5	
Create a new ur constant of value Ann in the argum		5
Create a new ur in the argument with ID Parent:21.	5	
Create a new ur constant of value Bob in the argum		5
Create a new ur in the argument with ID Parent:16.	5	
Create a new ur constant of value Liz in the argum		5
Create a new ur in the argument with ID Parent:10.	5	
Create a new ur constant of value Bob in the argum		5
Create a new ur constant of value Tom in the argum	5	
Create a clause with arguments Tom and Bob in prog		5
Create a new ur constant of value Pat in the argum	5	
Create a clause with arguments Pat and Jim in prog		5
Create a new ur constant of value Pat in the argum	5	
Create a clause with arguments Bob and Pat in prog		5
Create a new ur constant of value Pam in the argum	5	
Create a clause with arguments Pam and Bob in prog		5
Create a new ur constant of value Liz in the argum	5	
Create a clause with arguments Tom and Liz in prog		5
Create a new ur constant of value data in the term	5	
Create the argument term for the "Column Sum Query		5
Create a new ur constant of value Bob in the argum	5	
Create a clause with arguments Bob and Pat in prog		5
Create a new ur constant of value Bob in the argum	5	
Create a clause with arguments Bob and Ann in prog		5
Create a new ur constant of value Bob in the argum	5	
Create a clause with arguments Tom and Bob in prog		5
Create a new ur constant of value Ann in the argum	5	
Create a clause with arguments Bob and Ann in prog		5
Create a new term table in the argument with ID Co	5	
Create the argument term for the "Column Sum Query		4
**** SKIP 1 types. ****		1
Create a new program (and window) named "Parent".	5	
Create a program of six clauses. These clauses des		5
Create a new program (and window) named "Exercise	5	
Exercise 3.1		5
Create a new clause named "Column Sum Query" with	5	
Create the "Column Sum Query" clause.		5
Create a new clause in program "Parent" with its t	5	
Create a clause with arguments Pat and Jim in prog		5
Create a new clause in program "Parent" with its t	5	
Create a clause with arguments Tom and Bob in prog		5
Create a new clause in program "Parent" with its t	5	
Create a clause with arguments Pam and Bob in prog		5
Create a coreference link including variable with	5	
Create a "Parent Query" clause for asking the ques		5

Table 2 (part 10 of 18): Interaction type counts

Create a coreference link including variable with	5	
Create a "Parent Query" clause for asking the ques		5
Create a coreference link including variable with	5	
Create a "Parent Query" clause for asking the ques		5
Create a clause with arguments Pam and Bob in prog	5	
Create a program of six clauses. These clauses des		5
Close program "Fallible Socrates RESULT".	5	
This script discusses *how* SPARCL works.		5
This script describes what SPARCL programs mean.	4	
**** SKIP 2 types. ****		4
Set the "table type" of term_table with ID Column	4	
Create the argument term for the "Column Sum Query		4
Set the "result type" of intensional_set with ID C	4	
Create the literal of the "Column Sum" clause.		4
Presentation of an application of SPARCL	4	
Presentation of an application of SPARCL		4
Open the program in the file ":Union".	4	
Discuss term matching.		4
Open the program in the file ":Declarative Example	4	
The declarative meaning of SPARCL		4
Make program "Union" the front window.	4	
Discuss term matching.		4
Make program "Declarative Example" the front windo	4	
The declarative meaning of SPARCL		4
Extend an N-tuple, creating a new set term as the	4	
Create the literal of the "Column Sum" clause.		4
Execute a query of clause with ID Line Segments:35	4	
Discuss term matching.		4
Execute a query of clause with ID Column Sum temp:	4	
Presentation of an application of SPARCL		4
Edit the value of ur with ID Parent:36 to "Jim".	4	
Create a new ur constant of value Jim in the argum		4
Edit the value of ur with ID Parent:35 to "Pat".	4	
Create a new ur constant of value Pat in the argum		4
Edit the value of ur with ID Parent Query:88 to "A	4	
Create a new ur constant of value Ann in the argum		4
Edit the value of ur with ID Column Sum temp:22 to	4	
Create the literal of the "Column Sum" clause.		4
Edit the value of ur with ID Column Sum temp:17 to	4	
Create a new ur constant of value + in the argumen		4
Edit the value of comment with ID Parent Query:81	4	
Create a new comment "Who is a common parent of A		4
Edit the value of comment with ID Column Sum temp:	4	
Create a new comment "Function Table" in the argum		4
Create the literal of the "Column Sum" clause.	4	
Create the "Column Sum" clause.		4
Create the "Column Sum" clause.	4	
Presentation of an application of SPARCL		4
Create an N-tuple with variable with ID Column Sum	4	
Create the literal of the "Column Sum" clause.		4
Create an N-tuple with ur with ID Column Sum temp:	4	
Create the literal of the "Column Sum" clause.		4

Table 2 (part 11 of 18): Interaction type counts

Create an N-tuple with ur with ID Column Sum temp:	4	
Create the literal of the "Column Sum" clause.		4
Create a row at the bottom of term_table with ID C	4	
Create the argument term for the "Column Sum Query		4
Create a new variable in the term_table_cell with	4	
Create the argument term for the "Column Sum Query		4
Create a new variable in the set with ID Column Su	4	
Create the literal of the "Column Sum" clause.		4
Create a new variable in the intensional_set_templ	4	
Create the literal of the "Column Sum" clause.		4
Create a new variable in the argument with ID Pare	4	
Create a variable.		4
Create a new variable in the argument with ID Pare	4	
Create a variable.		4
Create a new variable in the argument with ID Pare	4	
Create a variable.		4
Create a new variable in the argument with ID Pare	4	
Create a variable.		4
Create a new variable in the argument with ID Pare	4	
Create a variable.		4
Create a new variable in the argument with ID Colu	4	
Create the comments and variables of the arguments		4
Create a new variable in the argument with ID Colu	4	
Create the comments and variables of the arguments		4
Create a new variable in the argument with ID Colu	4	
Create the literal of the "Column Sum" clause.		4
Create a new ur in the argument with ID Parent:34.	4	
Create a new ur constant of value Jim in the argum		4
Create a new ur in the argument with ID Parent Que	4	
Create a new ur constant of value Ann in the argum		4
Create a new ur in the argument with ID Column Sum	4	
Create a new ur constant of value + in the argumen		4
Create a new ur constant of value Jim in the argum	4	
Create a clause with arguments Pat and Jim in prog		4
Create a new ur constant of value Ann in the argum	4	
Create an ur object.		4
Create a new ur constant of value + in the argumen	4	
Create the literal of the "Column Sum" clause.		4
Create a new set in the set with ID Column Sum tem	4	
Create the literal of the "Column Sum" clause.		4
Create a new program (and window) named "Column Su	4	
Presentation of an application of SPARCL		4
Create a new partitioned set part in the set with	4	
Create the literal of the "Column Sum" clause.		4
Create a new partitioned set part in the set with	4	
Create the literal of the "Column Sum" clause.		4
Create a new literal with ur predicate name in the	4	
Create the literal of the "Column Sum" clause.		4
Create a new literal with name is and 2 arguments	4	
Create the literal of the "Column Sum" clause.		4
Create a new comment in the clause with ID Parent	4	
Create a new comment "Who is a common parent of A		4

Table 2 (part 12 of 18): Interaction type counts

Create a new comment in the argument with ID Colum	4	
Create a new comment "Function Table" in the argum	4	4
Create a new comment "Sum of Identified Range Valu	4	
Create the comments and variables of the arguments	4	4
Create a new comment "Function Table" in the argum	4	
Create the comments and variables of the arguments	4	4
Create a new comment "Domain Value Identifying Ran	4	
Create the comments and variables of the arguments	4	4
Create a new comment ""Who is a common parent of A	4	
Create a "query" clause with two arguments and two	4	4
Create a new clause named "Parent Query" with 1 ar	4	
Create a "query" clause with two arguments and two	4	4
Create a new clause named "Column Sum" with 3 argu	4	
Create the "Column Sum" clause.	4	4
Create a coreference link including variable with	4	
Create a "Parent Query" clause for asking the ques	4	4
Create a coreference link including variable with	4	
Create the "Column Sum" clause.	4	4
Create a coreference link including variable with	4	
Create the "Column Sum" clause.	4	4
Create a coreference link including variable with	4	
Create the literal of the "Column Sum" clause.	4	4
Create a coreference link including variable with	4	
Create the "Column Sum" clause.	4	4
Close program "Union".	4	
Discuss term matching.	4	4
Close program "Line Segments".	4	
Discuss term matching.	4	4
Close program "Line Segments RESULT".	4	
Discuss term matching.	4	4
Close program "Declarative Example".	4	
The declarative meaning of SPARCL	4	4
Close program "Column Sum temp".	4	
Presentation of an application of SPARCL	4	4
Chapter of tutorial detailing SPARCL representatio	4	
**** SKIP 2 types. ****	4	4
**** SKIP 88 types. ****	159	
runvb	38	
**** SKIP 31 types. ****	38	
runbf	35	
<< (Please read the Script Commentary) >>	8	
**** SKIP 5 types. ****	8	8
**** SKIP 26 types. ****	27	
registered_scripts	26	
<< (Please read the Script Commentary) >>	15	
Tutorial introduction to SPARCL	8	
**** SKIP 7 types. ****	7	
**** SKIP 7 types. ****	11	
done	15	
<< (Please read the Script Commentary) >>	6	
**** SKIP 6 types. ****	6	6
**** SKIP 9 types. ****	9	

Table 2 (part 13 of 18): Interaction type counts

general_tool	459		
activate	334		
›Parent Query {Program}		45	
›Exercise 1.2 {Program}		33	
›Exercise 1.1 {Program}		30	
Predecessor {Program}		23	
†Parent Query {Program}		17	
›Column Sum temp {Program}		14	
›Parent {Program}		10	
Sex - long {Program}		10	
Set Types {Term Set}		10	
Geometry Example {Term Set}		10	
›Parent Query RESULT {Term Set}		9	
›Exercise 2.1 {Program}		8	
Term Types {Term Set}		8	
Sex - table {Program}		8	
Offspring {Program}		8	
Family Relationships {Program}		8	
Fallible Socrates {Program}		7	
›Fallible Socrates RESULT {Term Set}		6	
›Exercise 3.1 {Program}		6	
†Exercise 1.1 {Program}		6	
Predecessor "How" Example {Program}		6	
Predecessor "How" Ex, 2 {Program}		6	
Parent {Program}		6	
Union {Program}		5	
Line Segments {Program}		5	
›Line Segments RESULT {Term Set}		4	
›Exercise 1.1 RESULT {Term Set}		4	
›Column Sum temp RESULT {Term Set}		4	
Declarative Example {Program}		4	
**** SKIP 7 types. ****		14	
close_edit	125		
Pat		13	
Exercise 1.1			12
**** SKIP 1 types. ****			1
new_ur		11	
Exercise 2.1			6
Exercise 3.1			4
**** SKIP 1 types. ****			1
Pat	8		
Exercise 1.1			8
2		6	
Exercise 2.1			6
1		6	
Exercise 2.1			6
Happy		5	
Exercise 1.2			5
Liz		4	
Exercise 1.1			4
**** SKIP 56 types. ****		72	

Table 2 (part 14 of 18): Interaction type counts

window	360		
activate	360		
Exercise 1.1		92	
Exercise 1.2		76	
Exercise 2.1		38	
Exercise 1.1 RESULT		23	
Family Relationships		21	
Parent		18	
Exercise 3.1		16	
Exercise 1.2 RESULT		14	
Geometry Example		12	
Predecessor		10	
Column Sum temp		9	
Parent Query		7	
Parent Query RESULT		5	
Set Types		4	
**** SKIP 9 types. ****		15	
Argument Ops	212		
Insert:Variable	149		
argument(3)		9	
Exercise 1.2			5
**** SKIP 2 types. ****			4
argument(6)		8	
Exercise 1.1			4
**** SKIP 2 types. ****			4
argument(7)		6	
Exercise 1.2			4
**** SKIP 1 types. ****			2
argument(33)		5	
**** SKIP 2 types. ****			5
argument(36)		4	
**** SKIP 2 types. ****			4
argument(30)		4	
**** SKIP 2 types. ****			4
argument(22)		4	
**** SKIP 2 types. ****			4
argument(18)		4	
**** SKIP 2 types. ****			4
argument(15)		4	
**** SKIP 2 types. ****			4
**** SKIP 50 types. ****		101	
Insert:Ur	31		
**** SKIP 22 types. ****		31	
Delete	16		
**** SKIP 14 types. ****		16	
Insert:Set	7		
**** SKIP 6 types. ****		7	
Insert:Table	6		
**** SKIP 6 types. ****		6	
**** SKIP 1 types. ****		3	

Table 2 (part 15 of 18): Interaction type counts

Clause Ops	194		
Create Literal	82		
clause(1)		30	
Exercise 2.1			8
Exercise 1.2			8
Exercise 1.1			8
Exercise 3.1			6
clause(30)		8	
Exercise 1.2			4
Exercise 1.1			4
clause(51)		4	
Exercise 1.2			4
clause(34)		4	
**** SKIP 2 types. ****			4
clause(25)		4	
Exercise 1.1			4
clause(3)		4	
Exercise 1.1			4
**** SKIP 14 types. ****		28	
Create Argument	51		
clause(1)		27	
Exercise 2.1			9
Exercise 1.1			9
Exercise 1.2			5
Exercise 3.1			4
clause(30)		5	
Exercise 1.2			4
**** SKIP 1 types. ****			1
clause(51)		4	
Exercise 1.2			4
**** SKIP 9 types. ****		15	
Query: Brief	41		
clause(1)		21	
Exercise 1.1			14
Exercise 1.2			7
clause(25)		4	
Exercise 1.1			4
**** SKIP 11 types. ****		16	
Create Comment	10		
clause(1)		4	
**** SKIP 2 types. ****			4
**** SKIP 6 types. ****		6	
Select	4		
**** SKIP 2 types. ****		4	
**** SKIP 2 types. ****	6		

Table 2 (part 16 of 18): Interaction type counts

connect_tool	115			
activate	115			
Parent Query		51		
Exercise 1.2		30		
Exercise 1.1		25		
Column Sum temp		8		
**** SKIP 1 types. ****		1		
File	105			
Record Comment about SPARCL...	74			
script_control		47		
Output		6		
Exercise 2.1		4		
Exercise 1.2		4		
Exercise 1.1		4		
**** SKIP 5 types. ****		8		
Quit	17			
script_control		11		
**** SKIP 4 types. ****		5		
Open Program...	7			
**** SKIP 3 types. ****		7		
**** SKIP 5 types. ****	7			
link	77			
variable(46)	7			
variable(52)		4		
Exercise 1.2			4	
**** SKIP 3 types. ****		3		
variable(10)	6			
**** SKIP 4 types. ****		6		
variable(47)	4			
**** SKIP 3 types. ****		4		
**** SKIP 40 types. ****	60			
Program Ops	54			
Create Clause	53			
program(0)		53		
Exercise 1.1			23	
Exercise 1.2			19	
Exercise 2.1			9	
**** SKIP 1 types. ****			2	
**** SKIP 1 types. ****	1			
Clause Name Ops	41			
Edit Clause Name	36			
name(2)		10		
Exercise 1.2			4	
**** SKIP 3 types. ****			6	
**** SKIP 21 types. ****		26		
Select	5			
**** SKIP 3 types. ****		5		

Table 2 (part 17 of 18): Interaction type counts

Windows	38		
Exercise 1.2		5	
**** SKIP 3 types. ****			5
Parent		4	
**** SKIP 4 types. ****			4
Geometry Example		4	
Exercise 2.1			4
Family Relationships		4	
**** SKIP 4 types. ****			4
Exercise 1.1		4	
**** SKIP 3 types. ****			4
**** SKIP 11 types. ****	17		
Tutorial Scripts	36		
3.0.Application of SPARCL		6	
Output			4
2.1.Data objects		4	
**** SKIP 3 types. ****			4
**** SKIP 11 types. ****	26		
Literal Ops	29		
Create Argument		17	
literal(5)			4
Exercise 1.1			4
**** SKIP 8 types. ****			13
Delete		10	
**** SKIP 9 types. ****			10
**** SKIP 1 types. ****		2	
DISALLOWED	29		
File		21	
New Program...			11
script_control			9
**** SKIP 1 types. ****			2
Open Program...			4
script_control			4
**** SKIP 5 types. ****			6
**** SKIP 3 types. ****		8	
Set Ops	24		
Insert:Ur		10	
**** SKIP 6 types. ****			10
Create NTuple:Ur		4	
**** SKIP 3 types. ****			4
Create NTuple:Set		4	
**** SKIP 3 types. ****			4
**** SKIP 5 types. ****		6	
Term Table Cell Ops	19		
Insert:Ur		15	
**** SKIP 14 types. ****			15
**** SKIP 2 types. ****		4	

Table 2 (part 18 of 18): Interaction type counts

START LOGGING	17		
K... C...	5		
**** SKIP 3 types. ****		5	
**** SKIP 7 types. ****	12		
Ur Ops	15		
Delete		8	
**** SKIP 7 types. ****			8
Edit Ur Item		4	
**** SKIP 4 types. ****			4
**** SKIP 1 types. ****		3	
Variable Ops	12		
Unlink Variable		7	
**** SKIP 7 types. ****			7
Delete Variable		5	
**** SKIP 4 types. ****			5
Partition Ops	12		
Delete Partition		5	
**** SKIP 5 types. ****			5
**** SKIP 4 types. ****		7	
Edit	9		
Undo Most Recent		5	
Exercise 2.1			5
**** SKIP 4 types. ****		4	
Term Table Ops	7		
**** SKIP 5 types. ****		7	
NTuple Ops	7		
**** SKIP 5 types. ****		7	
**** SKIP 4 types. ****	8		

Table 3: Durations in minutes by section:

User	Count	Total	1.1	1.2	1.3	1.4	2.1	2.2	2.3	2.4	3.1
ag	4	46.2	29.6	13.2	1.8	1.6	-	-	-	-	-
ca	9	154.1	26.6	31.1	10.3	1.4	33.2	4.7	2.9	1.7	42.2
el	6	77.5	22.4	33.2	5.2	2	13.8	0.9	-	-	-
jp	9	81.5	36.2	11.9	9.2	1.6	7.7	2.6	1.7	0.5	10
kc	9	130.3	44.8	28.9	9.6	3.6	11.2	3.1	4.4	3.2	21.4
rv	9	186.4	62.3	18.5	21	1.6	45.2	4.6	2.7	1.1	29.5
vn	3	79.5	51.6	21.9	6	-	-	-	-	-	-

Table 4: The time in minutes spent in working on the exercises:

User	Count	Total	Exercise			
			1.1	1.2	2.1	3.1
ag	2	19.6	6.2	13.4	-	-
ca	4	101	12.6	34.8	33.1	20.4
el	3	32.3	12.5	19.6	0.2	-
jp	3	25.4	18.4	2.6	4.4	-
kc	3	45.9	11.7	13.3	20.8	-
rv	3	60.6	25.1	27.2	8.2	-
vn	2	16	14.8	1.3	-	-

Table 5: Statistics on the exercise durations:

Statistics	count	total	1.1	1.2	2.1	3.1
Count	7	7	7	7	5	1
Total	20	300.7	101.3	112.2	66.8	20.4
Avg	2.9	43	14.5	16	13.4	20.4
Dev	0.6	27.7	5.5	11.3	12.1	0
Median	3	32.3	12.6	13.4	8.2	20.4
Min	2	16	6.2	1.3	0.2	20.4
Max	4	101	25.1	34.8	33.1	20.4

Table 6: Total times

User	All Sections	All Exercises	Other	Total
ag	46.2	19.6	25.5	91.3
ca	154.1	101	8.7	263.7
el	77.5	32.3	11.4	121.2
jp	81.5	25.4	24.2	131
kc	130.3	45.9	19.9	196
rv	186.4	60.6	9	256
vn	79.5	16	20.1	115.6

Appendix 5

Response Time Analysis

This appendix describes the result of a test to determine the response time of SPARCL for common simple editing operations, and how much having other programs loaded slows down these operations. In this test, the following simple program in Figure A5 – 1 is constructed.

To test the observation that having unrelated programs loaded slows down the editing process, the above program was constructed in SPARCL 2D6 once without any other programs loaded, then a second time with the ID3 project loaded.

The Program Size Measurements for ID3 are shown in Figure A5 – 2.

In addition to the measurements, the number of “display objects” in SPARCL’s internal database was 766. Note that this is very close to the “graphical size” of 764 (sum of graphical total operators and total operands).

These tests were run on an Apple Macintosh 8500/120 with 32Meg RAM (64Meg virtual RAM). The SPARCL application was assigned 16Meg memory. The results without ID3 loaded are shown in Figure A5 – 3. The results of the test *with* ID3 loaded

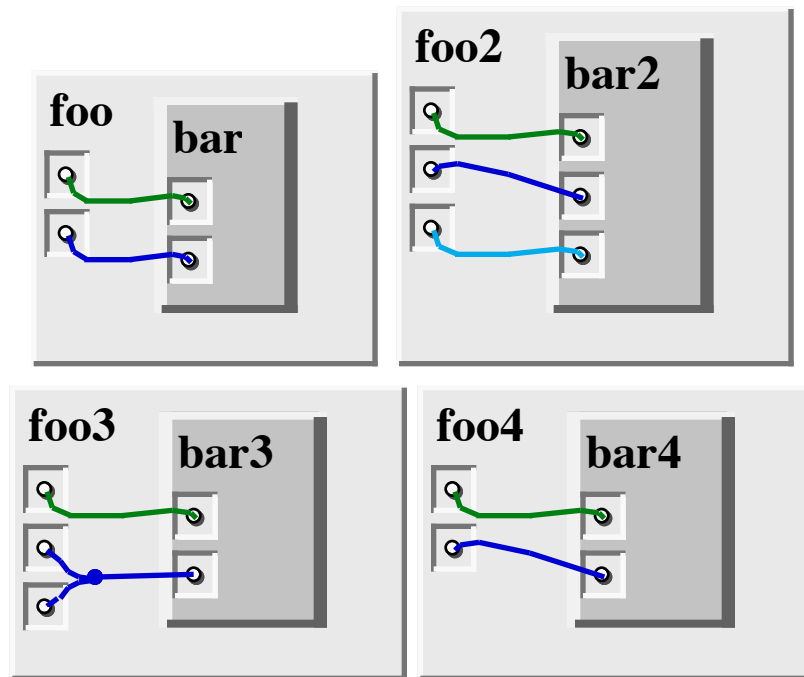


Figure A5 – 1: Program constructed for response time test.

grams: [Cardinality, Classify	unique_references : 67	max_rows : 0
nuples, DE - Attribute Decisions,	total_references : 150	enclosures : 146
- Attribute Value, DE - Attribute	unique_sets : 1	enclosure_present : 1
ies, Determine Entropies,	total_sets : 19	nodes : 159
ogeneous Examples, ID3 Tree,	max_parts : 2	node_types : 5
Trees, Select Attribute]	unique_parts : 1	edges : 163
	total_parts : 36	edge_types : 2
al_count : 34	max_parts_card : 3	textual_tokens : 177
se_count : 36	unique_urs : 19	textual_token_types : 27
edure_count : 13	total_urs : 121	adjoinments : 119
ue_operators : 23	unique_names : 18	adjoinments_present : 1
_operators : 242	total_names : 56	vocabulary : 110
ue_operands : 87	unique_vars : 1	size : 663
_operands : 421	total_vars : 150	volume_basic : 4496
me_special : 3983	unique_empty_sets : 0	program_level_estimated : 0.018
hical_unique_operators : 4	total_empty_sets : 0	effort_special : 221278
hical_total_operators : 428	unique_intensional_sets : 1	effort_special_hours : 7.68±4.61
hical_unique_operands : 32	total_intensional_sets : 7	effort_basic : 250202
hical_total_operands : 336	unique_fact_tables : 1	effort_basic_hours : 8.69±5.21
hical_volume : 3950	total_fact_tables : 8	language_level_special : 1.29
ue_ntuples : 1	max_facts : 3	language_level_basic : 1.457
_ntuples : 116	unique_term_tables : 0	
_ntuple_items : 6	total_term_tables : 0	

Figure A5 – 2: Program Size Measurements for ID3.

are shown in .

Because the two tests have some minor unintended differences, it is useful to adjust the totals for comparisons. The “without ID3” test has only 7 Literal Ops interactions while the “with ID3” test has 9 Literal Ops interactions. The “without ID3” window and general_tool types have 7 and 5 interactions, respectively, while the “with ID3” test has 6 and 6 interactions, respectively. There are 2 interactions in the final “SKIP” of the “with ID3” test and 3 interactions in the “without ID3”.

The direct comparison of the lengths of these test yields 159 seconds for the “without ID3” test versus 385 seconds for the “with ID3” test. This is a slowdown of more than a factor of two overall.

The “with ID3” number can be adjusted to approximate the same interactions as the “without ID3” test by subtracting two average Literal Ops ($2 \times 4.9 = 9.8$), subtracting one average window (3.3), and adding one average general_tool (2.8). This gives an adjusted total duration for the “with ID3” test of (approximately) 375. This is still more than twice the duration of the “without ID3” test.

<i>operation</i>	<i>count</i>	<i>total</i>	<i>avg</i>	<i>std dev</i>	<i>median</i>	<i>min</i>	<i>max</i>
Argument Ops	19	38	2	0.9	2	0	4
Insert:Variable	19	38	2	0.9	2	0	4
**** SKIP 19 types. ****	19						
Clause Ops	12	36	3	2	2	1	7
Create Argument	8	15	1.9	0.3	2	1	2
**** SKIP 4 types. ****	8						
Create Literal	4	21	5.2	2	7	2	7
**** SKIP 4 types. ****	4						
link	10	24	2.4	1.2	2	1	5
**** SKIP 10 types. ****	10						
Literal Ops	7	15	2.1	0.3	2	2	3
Create Argument	7	15	2.1	0.3	2	2	3
**** SKIP 3 types. ****	7						
window	6	13	2.2	1.1	2	1	4
activate	6	13	2.2	1.1	2	1	4
foo	6	13	2.2	1.1	2	1	4
general_tool	6	10	1.7	0.7	2	0	2
activate	4	6	1.5	0.9	2	0	2
†foo {Program}	4	6	1.5	0.9	2	0	2
**** SKIP 1 types. ****	2						
Program Ops	4	14	3.5	1.1	4	2	5
Create Clause	4	14	3.5	1.1	4	2	5
program(0)	4	14	3.5	1.1	4	2	5
foo	4	14	3.5	1.1	4	2	5
connect_tool	4	9	2.2	0.8	3	1	3
activate	4	9	2.2	0.8	3	1	3
foo	4	9	2.2	0.8	3	1	3
**** SKIP 2 types. ****	4						
Totals:	72	159					

Figure A5 – 3: Without ID3 loaded.

<i>operation</i>	<i>count</i>	<i>total</i>	<i>avg</i>	<i>std dev</i>	<i>median</i>	<i>min</i>	<i>max</i>
Argument Ops	19	127	6.7	1.4	7	5	11
Insert:Variable	19	127	6.7	1.4	7	5	11
**** SKIP 19 types. ****	19						
Clause Ops	12	78	6.5	2.9	5	4	11
Create Argument	8	36	4.5	0.7	4	4	6
**** SKIP 4 types. ****	8						
Create Literal	4	42	10.5	0.5	11	10	11
**** SKIP 4 types. ****	4						
link	10	57	5.7	1.3	5	4	9
**** SKIP 10 types. ****	10						
Literal Ops	9	44	4.9	0.9	5	4	7
Create Argument	9	44	4.9	0.9	5	4	7
**** SKIP 4 types. ****	9						
window	7	23	3.3	2.4	2	2	9
activate	7	23	3.3	2.4	2	2	9
foo	7	23	3.3	2.4	2	2	9
general_tool	5	14	2.8	1.7	2	1	6
activate	4	8	2	0.7	2	1	3
†foo {Program}	4	8	2	0.7	2	1	3
**** SKIP 1 types. ****	1						
Program Ops	4	35	8.8	1.3	10	7	10
Create Clause	4	35	8.8	1.3	10	7	10
program(0)	4	35	8.8	1.3	10	7	10
foo	4	35	8.8	1.3	10	7	10
connect_tool	4	7	1.8	0.4	2	1	2
activate	4	7	1.8	0.4	2	1	2
foo	4	7	1.8	0.4	2	1	2
**** SKIP 2 types. ****	3						
Totals:	73	385					

Figure A5 – 4: With ID3 loaded.