SET REPRESENTATIONS IN A
SUBSET-EQUATIONAL LANGUAGE

by

Giancarlo Succi

A thesis submitted to the Faculty of the Graduate School
of the State University of New York at Buffalo
in partial fulfillment of the requirements
for the degree of
Master of Science

January 1991
Acknowledgments

I would like to thank my thesis advisor, Dr. B. Jayaraman, for his patience and his help and ideas.

Also a big thanks goes to all the people of the Computer Science Department, especially to Dr. Ken Regan, who always challenged my mind, to Mrs. Ellie Benzol, who helped me through the bureaucracy, and to all my colleagues.

A special thanks goes also to the Fulbright organization and to the Italian commission for the cultural exchanges between Italy and USA which gave me the such a great opportunity.

Last but not the least, I would like to express my thank to Prof. G. Marino who never got bored of my questions.

Set Representations in a Subset-Equational Language

by

Giancarlo Succi

Submitted to the Faculty of the Graduate School of State University of New York at Buffalo on January 21, 1991, in partial fulfillment of the requirements for the degree of Master of Science

Abstract

This thesis investigates alternative set representations in the subset-equational language SEL. The current SEL implementation represents sets using unordered lists for the sake of simplicity and for compatibility with the Warren Abstract Machine (WAM). This thesis explores a potentially better representation, namely, hash tables, following some of the techniques proposed in the imperative, set-oriented language SETL. We examine the way in which sets are usually generated in SEL, and then propose a suitable hash table implementation. We compare the two approaches (lists vs hash tables), and examine the ease of performing optimizations in both approaches. We also explore a technique for estimating object sizes in SEL, based on program analysis, and mention its relevance for efficient code generation. Finally, we show how the current abstract machine for SEL can be modified to accommodate a hash table representation for sets.

Thesis Supervisor: Bharat Jayaraman
Title: Associate Professor
Contents

1 Introduction
   1.1 Declarative Programming  9
   1.2 Set-based Languages    10
   1.3 This Thesis            10

2 Subset-Equational Programming
   2.1 Equational and Subset Assertions  12
   2.2 Set Construction         16
   2.3 Distribution Over Union  17

3 The Hash Table Representation
   3.1 Hash Tables            19
   3.2 Base Sets              20
   3.3 Pattern Matching with Hash Tables  21
   3.4 Distribution Over Union 23

4 Object Size Analysis
   4.1 A Metric for Sizes      25
   4.2 Data Dependency Graph    27
      4.2.1 Building the Data Dependency Graph  27
      4.2.2 Size Relations       28
   4.3 Difference Equations     29
      4.3.1 Non Recursive Clauses 30
      4.3.2 Recursive Clauses    30

4.4 Subset Assertions       31
4.5 A Final Example         33

5 The Virtual Machine
   5.1 The Prolog WAM         35
   5.2 The SEL-WAM            36
   5.3 SEL-WAM with Hash Tables 38
   5.4 A Sample Compilation   41

6 Conclusions and Further Research  43

A Sample Programs
   A.1 Huffman Code          45
   A.2 Topological Sorting   47
   A.3 Minimum Distance     48
   A.4 Cyclic Graphs        49

B The SEL Environment      52
Chapter 1

Introduction

1.1 Declarative Programming

Analyzing the various paradigms of computing, Kowalski [Kow79] coined the equation

\[ \text{Algorithm} = \text{Logic} + \text{Control} \]

meaning that an algorithm consists of two independent parts: the abstract idea of what one wants to compute, i.e., the logic, and the concrete sequence of operations needed to carry out the idea, i.e. the control. Thus, we can partition the programming languages into two classes:

1. imperative class: viz. the class of languages that explicitly specify the control, and leave the logic implicit in the mind of the programmer or attached as comments to various control points in the program;

2. declarative class: viz. the class of those languages that explicitly specify the logic and leave the control implicit.

Examples of the first class are assembly languages, Fortran, Algol, Pascal, Ada, C, C++, and even the Turing Machine control sequence format. Pure Lisp, FP, Haskell, \( \lambda \)-calculus, pure Prolog, CLP, EqL, SEL, and SuRE fall in the second class. It may be noted that, although these two classes differ completely in their approach to algorithm
specification, they are equivalent under computability properties. Sometimes further classifications may be made within these two groups: e.g., declarative languages may be separated into functional and logic languages.

While the declarative paradigm is still not as widely used as the imperative paradigm (although the situation is rapidly changing), it nevertheless offers several advantages over the imperative paradigm:

- **Implicit Parallelism.** Since no explicit control is imposed on the program, parallelism can be easily automatically exploited by the compiler (this form of parallelism is termed transparent parallelism).

- **Abstract Analysis.** Explicitly specifying only the logic of the program has the consequence that various forms of abstract manipulation are intrinsically simpler, e.g., mode analysis, grainsize analysis, strictness analysis.

- **Program Verification.** The absence of explicit control and assignment statements make declarative programs closer to their specifications, thus simplifying the task of program verification.

- **Fast Prototyping.** As the user has only to specify the logic underlying a program, declarative programs tend to be much shorter than their imperative counterparts, and hence declarative languages are good tools for rapid prototyping.

Therefore, it seems feasible in the near future that, in order to write an efficient program, one just defines its logic, then verifies its correctness with respect to some specifications, then employs abstract analyzers to transform it into code to be executed on a fast, multiprocessor machine.

### 1.2 Set-based Languages

In the family of programming languages, there is a class of languages that emphasizes a particular data-structure. For example, LISP emphasizes lists, APL emphasizes arrays, etc. In the mid seventies, a group of researchers from New York University [SSS81] designed an imperative language based on the set construct, called SETL. Although this was not the first time sets were used as a data structure in a programming language—for example, the language Pascal has sets—never before did sets play so primary a role as in SETL. Furthermore, SETL provided the programmer to indicate the kind of set representation most appropriate to the problem, and also a mechanism for automatically choosing a representation according to some predefined rules.

In 1987, Jayaraman and Plaisted developed a logic programming language called SEL (Subset Equational Language)[JP87]. While programming with equations is a well-known paradigm, subset assertions were a new idea and had been introduced in order to provide a more efficient treatment of set-valued functions than existing approaches[Jay90a]. Furthermore, this paradigm of computing has easy and simple declarative semantics, based on the concept of logical consequences of the completion of the program. Its operational semantics is based on set-matching and innermost rewriting of expressions, features that lend themselves to efficient implementation.

### 1.3 This Thesis

The topic of this thesis belongs to the conjunction of the two aforementioned areas: declarative languages and set-based languages. More specifically, it addresses the problem of implementing sets in SEL. In the current implementation of SEL, sets are represented using unordered lists. Although much work has been done in studying properties of set-valued functions that allow more efficient execution, not much has been done in potential alternatives to the list representation. We propose to investigate a hash table representation for sets. The SETL project is relevant in our research since a significant part of SETL work was aimed at hash table implementation of sets. The approach of this thesis is to investigate a broad range of issues relating to the set representation problem, rather than to concentrate on any issue in great detail. Thus we examine how sets are constructed typically in SEL programs; we examine how pattern-matching with sets and the property of functions distributing over
union in a particular argument are compatible with the hash table representation. We sketch a method of estimating the size of objects and sets (to guide hash table storage allocation) and also abstract machine instruction sets for a hash table based SEL implementation.

The rest of this document is organized as follows. Chapter 2 gives a brief introduction to SEL and programming with sets; chapter 3 presents a new scheme, for set implementation, based on hash tables and analyzes its advantages and disadvantages; chapter 4 describes an approach for object size analysis, in order to estimate the amount of storage needed for objects, especially sets; chapter 5 shows how to change the current SEL virtual machine to support hash tables. The appendices are a collection of SEL programs to well-known problems and a introduction to the SEL environment.

We assume that the reader has some familiarity with logic programming and its implementation.

Chapter 2

Subset-Equational Programming

This chapter gives a brief introduction to the paradigm of subset-equational programming. The purpose is not to give a detailed description of the language, its semantics and its implementation strategies, for which the reader can consult the references [Jay91, Jay90a].

2.1 Equational and Subset Assertions

A SEL program is a collection of two kind of assertions:

\[
\begin{align*}
 f(\text{terms}) &= \text{expression.} \\
 f(\text{terms}) &\supset \text{expression.}
\end{align*}
\]

All variables appearing on the right-hand side (rhs) of an assertion must also appear on the left-hand side (lhs). The first is an equational assertion, while the second is a subset assertion. The declarative meaning of the above equational assertion (respectively, subset assertion) is that, for all ground instances of it, i.e., replacing variables by ground terms, the function \( f \) applied to the corresponding ground instance of \( \text{terms} \) is equal to (respectively, contains) the ground term denoted by the corresponding ground instance of \( \text{expression} \). In general, multiple equational and subset assertions, as well as a combination of the two, can be used to define some function \( f \). Subset assertions incorporate a collect-all assumption, so that the result
of a applying a function \( f \) to argument ground terms \( t \) is equal to the union of the respective sets denoted by the rhs expressions of those subset assertions whose lhs terms match \( t \).

In the SEL environment the user is prompted for a ground expression:

\[ ? \ \text{ground}\_\text{expression} \]

Expressions are evaluated in innermost order, i.e., inner subexpressions are evaluated before outer subexpressions. When more than one subset assertion matches a subexpression, these assertions are attempted one at a time; i.e., there is a depth-first evaluation order. In legal SEL programs, at most one equational assertion is allowed to match a subexpression.

A simple example of a SEL program using equational assertions is the following:

\[
\begin{align*}
\text{father('John')} &= \text{'Anthony'}, \\
\text{father('Anthony')} &= \text{'Peter'}, \\
\text{father('Peter')} &= \text{'Bob'}, \\
\text{pat.grandfather}(x) &= \text{father(father}(x))
\end{align*}
\]

Now we can have this SEL session:

\[
\begin{align*}
? \ \text{father('John')} . \\
\quad \text{'Anthony'} \\
? \ \text{pat.grandfather('John')} . \\
\quad \text{'Peter'} \\
? \ \text{pat.grandfather('Peter')} \\
\end{align*}
\]

The first query is about the father of John, and hence the reply is Anthony. The next query asks for the paternal grandfather of John, and the reduction sequence is:

\[
\text{pat.grandfather('John')} \rightarrow \text{father(father('John'))} \rightarrow \text{father('Anthony')} \rightarrow \text{'Peter'}.
\]

The last query \( \text{pat.grandfather('Peter')} \) has the undefined element \( ? \), as a result since we have not defined who is the father of Bob.

Now we add the following assertions to the preceding programs:

\[
\begin{align*}
\text{mother('John')} &= \text{'Mary'}, \\
\text{father('Mary')} &= \text{'Ted'}, \\
\text{pat.grandfather}_s &= \text{father(mother}(x)). \\
\text{grandfathers}(x) &= \text{pat.grandfather}_s(x). \\
\text{grandfathers}(x) &= \text{pat.grandfather}_s(x).
\end{align*}
\]

Note that the keyword \( \text{contains} \) is the equivalent of \( \exists \). Here we define two subset assertions for \( \text{grandfathers} \), to state that \( \text{grandfathers} \) includes the father of the father and the father of the mother. By virtue of the collect-all assumption, no other elements are included. Suppose now that we want the set of both grandfathers of John. We might enter the following query:

\[
? \ \text{grandfathers('John')} \\
\quad \{('Bob', 'Ted')\}
\]

To compute the above answer, SEL looks for all assertions that can match \( \text{grandfathers('John')} \), and reduces the corresponding rhs one at a time to a resulting set. On the other hand, consider the query:

\[
? \ \text{grandfathers('Mary')} . \\
\end{align*}
\]

Note that \( \text{pat.grandfather('Mary')} \) and \( \text{pat.grandfather('Mary')} \) reduce to \( ? \), therefore by virtue of the emptiness as failure assumption of SEL we have that \( {?} = \{\} \). Hence the overall result is \( \{\} \). Easy and succinct programs can be written in SEL, such as the following:

\[
\begin{align*}
\text{setproduct} &\{x, y\} \text{ contains } \{x, y\} \\
\text{intersect} &\{x, y\} \text{ contains } \{x\}.
\end{align*}
\]
union(s1, s2) contains s1
union(s1, s2) contains s2

From the examples above it can be seen that the use of set terms is an important feature of this language. Despite its similarity to the usual list terms, it is semantically very different. To get a better sense of how it works consider matching \{a, b, c\} against the pattern \{x \perp t\}. We obtain three different matches for x and t, as follows:

- \(x \leftarrow a, t \leftarrow \{b,c\}\)
- \(x \leftarrow b, t \leftarrow \{a,c\}\)
- \(x \leftarrow c, t \leftarrow \{b,a\}\)

The above three matches correspond to the three different ways of selecting one element from the set \{a, b, c\} and forming the corresponding remainder. This behavior is what we call implicit iteration, viz., selecting the elements of a set one at a time. Sometimes recursive definitions can be transformed in non-recursive ones using set-matching, as illustrated by the definitions of setproduct, intersection, and union.

More complicated programs can also be written, as illustrated by the following program for solving the well-known \(N\)-Queens problem:

```prolog
solve(n) = queens(n,1,\{},iota(n)).
quen\(\text{s}(n, \text{col}, \text{safeset}, \text{rows}) =
  \text{if eq(col, n+1) then } \text{safeset}
  \text{else placequeen(n, col, safeset, rows, rows)}.
placequeen(n, col, safeset, [row|\{}), rows) contains
  \text{if safe([col|row], safeset)
     then queens(n, col+1, [col|row|safeset], rows)
     else }\{}.
safe([c1|r1], \{\}) = \text{true}.
safe([c1|r1], [[c2|r2] r]) = (r1 \leftarrow r2) \text{ and } (abs(c1-c2)}
```

\(<\text{set}(r1-r2)\) and safe([c1|r1], t).
iota(n) = \text{if eq(n, l) then } \{i\} \text{ else } [\text{iota}(n-1)]

2.2 Set Construction

Understanding the way in which sets are generated in subset-equational programming is crucial to properly designing their implementation. Hence we devoted some effort analyzing programs to determine some patterns occurring in the set construction. Considering the programs of the previous subsection as well as the programs in the appendix, we can partition the ways in which sets are built up in three groups.

a. sets generated from non-sets, element by element, e.g.:
   \(iota, \text{baseFreq, cyc180d, elem, cyclePred, elim uncyclic}\)

b. sets generated from other sets using mappings, e.g.:
   \(\text{setproduct, finalCode, addStar, totFreq, diff, startTag, noPred, doTag}\)

c. sets generated by joining other sets, e.g.:
   \(\text{union, makeCode, nodes, intersect, grandfathers.}\)

The number of examples belonging to the union of the two last categories is generally larger than those of the first, since all the relative set constructs widely used in functional languages fall in this class. It is clear that in cases b. and c. we may save space, since we might think of a mechanism of defining the new one in terms of the old ones, without a full reconstruction. However, taking this approach it is not clear whether there will be a saving in terms of time, since this operation may turn out to be more time consuming because of the need to re-construct new elements from the old ones. Anyway, these ideas will be clarified in more details in the next chapters.
2.3 Distribution Over Union

There is a need to eliminate duplicates when constructing a set because the language permits one to perform global operations on a set, such as finding its cardinality, etc. This need creates one of the biggest costs of programming with sets, since checking for duplicates is an operation which is quadratic in the size of the sets, if sets are represented as unordered lists.

Much effort was placed in trying to solve this problem and some useful results were developed, especially the property of distributing over union [Jay90a].

Definition 1 (Distribution over union in the $i^{th}$ argument) A set valued function $f(\ldots, x_i, \ldots)$ distributes over union in its $i^{th}$ argument, iff:

$$f(\ldots, x_i^1 \cup x_i^2, \ldots) = f(\ldots, x_i^1, \ldots) \cup f(\ldots, x_i^2, \ldots)$$

Definition 2 (Distribution over union) A set valued function distributes over union iff it distributes over union on all its set valued arguments.

Functions that use a set as a global entity, such as power set, usually do not distribute over union, while the ones that compute element-based properties, such as the set-product or intersection, usually do. If we know that a function distributes over union in an argument, we can defer checking for duplicates in that argument. Since $f(\ldots, x_i^1 \cup x_i^2, \ldots) = f(\ldots, x_i^1, \ldots) \cup f(\ldots, x_i^2, \ldots)$, it follows that

$f(\ldots, x, \ldots) = \{f(\ldots, [x], \ldots) : x \in s\}$.

Abstract analysis techniques can be helpful in automatically determining whether this optimization can be applied, but the current SEL compiler leaves it to the user to specify with annotations when this kind of optimization should be performed [JN88, Nai88]. The potential danger of this optimization is that, when the argument set does have duplicates, we might over-compute by operating element-at-a-time. However, usually argument sets do not have duplicate elements, so the use of this kind of optimization does indeed lead to substantial savings in space as well as time. Space is saved because large intermediate sets need not be formed when composing several such set-valued functions together to generate a result. This is a form of futility in the computation [Jay91], similar to the futility of some functional languages. Time is saved because the check for duplicate elements is avoided.
Chapter 3

The Hash Table Representation

The current SEL implementation uses linked lists to represent sets for the sake of simplicity in implementation, but this representation has the disadvantage that checking for duplicates and test for membership are expensive operations. In considering other representations, we find that Modula-2 [Wir85] implemented sets by means of bitvectors. However, this bitvector representation assumes a fixed maximum size for sets (the size of the vector), and hence is inapplicable to the SEL language. This chapter describes another approach, viz. SETL's remote base representation of sets. It also examines how this representation can be used to implement pattern-matching in SEL and also considers the compatibility of this representation with the distribution-over-union optimization.

3.1 Hash Tables

A hash table is a means of storing elements with the capability of retrieving them in constant-time on the average. The positions of elements in a hash table are determined by a function (called hashing function) over the elements themselves or over portions of them (called keys). For instance, if we want to store numbers in a hash table with 10 positions, we can place each number \( n \) in the position determined by the function \( f(n) = n \mod 10 \). If an element is to go into an already occupied place, we have to find another unique position for it. This can be done in two ways:

- Starting with the position given by the hash function, scan the table until an empty location is found. If the table is full, abort the computation.
- Assuming that each position in the hash table is a pointer to a linked list, store the element in the linked list, perhaps in some order. In this approach there is no limitation on the number of elements that can be stored in the table. Henceforth we will assume this approach.

Compared with linked lists, the two critical operations on sets, duplicates-checking and set-membership, are speeded up in this framework: The problem of duplicates-checking essentially reduces to the set-membership problem which entails either a look up of the element position, which is determined by the hashing function, or of scanning the linked list in this position. There are two critical issues of this approach: the size of the table, and the hashing function. It has been proved [Fe88] that under some circumstances (such as the number of elements being less than the 60% of the number of entries in the table), the access time can be made a constant. Determining a good hashing function is not easy, especially when the range of values to be considered is not fixed in advance, as in SEL. Our suggested approach is to define it for simple types (like integer, ...) and then, for composite ones (lists, sets, ...), to use combination of the hash values of the constituent elements. In the remaining of the thesis we will assume that we can define properly a hashing function.

3.2 Base Sets

The main cost in using hash tables generally is the cost of computing the hashing function. For example, if an element \( x \) is accessed 10 times, the hashing function would have to applied 10 times. Clearly, it would be more efficient to remember the position of \( x \) in the table and avoid the subsequent hashing. Another noteworthy point is that often several related sets tend to have many elements in common; in these cases, the cost of hashing can be minimized by maintaining all these related sets together in some way. It is this last observation that motivates the use of a base
set, which is defined as the values taken by a specified set of variables during program execution. Each of the related sets is thus some subset of the base set.

The simplest way of using this technique in conjunction with hash tables is referred to as local basing (in SETL terminology). Here, each element inserted into the hash table is assigned a unique numerical index, and a bitvector is allocated with each element to indicate in which of the related sets it is a member. The disadvantages of this approach are: (a) it is not convenient to make a set represented in this way a part of another structure, (b) common operations like union, intersection, ..., require element-by-element consideration and hence are not as fast as one would prefer, and (c) iteration over the set requires iteration over the entire base.

To overcome some of the above problems, the technique of remote basing was proposed [SSS81]. Here, elements are inserted into the hash table as before, but one bitvector is maintained for each of the related sets (instead of maintaining one bitvector per element). This allows fast union and intersection operations, taking advantage of machine-level logical operations, and also allows a set to participate in other sets conveniently. However, iteration over the set is still not easy to perform as it entails traversal over the entire base.

3.3 Pattern Matching with Hash Tables

One of the most widely used features in subset-equational programming is pattern matching, since the selection of every clause to be executed is determined by this operation. In order to determine its compatibility with respect to a hash table implementation, we can examine two different situations:

1. when the pattern to match is ground, and

2. when the pattern is not ground

If the pattern is ground, e.g.,

```plaintext
foo('Grass', {'Grass'1.}) = true
```

then the hash table implementation is inherently faster than the list-based one, since all that needs to be done is to compute a hash value (for 'Grass' in the above example) and check for the presence of the value in the set. This is because we are essentially making membership tests, which is on the average a constant-time operation with hash tables, whereas it is a linear-time operation for unordered lists.

When the pattern is not ground, we may consider two different cases:

1. free matching, viz. matching scalar objects, and

2. nested matching, viz. matching nested structures.

For free matching we can again identify two cases:

i. when there is a matching of a rule without repeated variables in the lhs:

```plaintext
doubles([a1..]) contains [2*a]
```

Here pattern matching reduces to iteration over the elements set, and this operation is generally faster with a list representation than with our hash table representation because the remote base representation requires iteration of the entire base rather than the set at hand.

ii. when we deal with rules with repeated variables in the lhs:

```plaintext
intersection([{a1..}], [a1..]) contains [a]
```

In this case the first pattern is of case i. above, while the second pattern becomes a case of simple ground matching after the first pattern is performed, and hence using hash table is again a constant time operation.

We can extend the foregoing ideas to the case of nested matching. The most common situation is exemplified by the following rule, since repeated nested patterns are rather rare:

```plaintext
foo([[Grass'[a]]]) contains [a]
```
In this situation, one problem is that we must be able to apply the hashing function to a partially defined object, like ['Grape' 'ia'], so that if the element was present, it would be in a certain range of cells of the hash table. This is similar to saying that if we are summing a list of n positive numbers and the first one is a 5 then the sum of the elements will be greater or equal to 5. However a proper handling of this fact seems to slow down the computation rather than speeding it up, so we prefer not to address it. Therefore there is no significant difference between the free and the nested case, and hence in this situation the results determined previously for free matching still hold.

3.4 Distribution Over Union

As noted earlier, knowing that a function distributes over union produces, in the present (list-based) implementation of SEL, significant advantages both in terms of space and time:

- (space) we can avoid constructing intermediate sets, and
- (time) we can postpone the check for duplicates.

Note that this property would be insignificant if we didn't know that a wide range of functions do have this property; however, an examination of several SEL programs seems to suggest that it is quite wide. So the question now is whether or not a list-based representation which uses this property could outperform a hash table representation which may or may not use it. We now develop some criteria which should help arrive some definite conclusions after a wider analysis of SEL programs.

Using a hash table representation, we can still take advantage of this property since:

- (space) we can think of some set-less transformation that uses intermediate sets without really constructing them, exactly in the same way as it is sometimes done for lists (e.g., generating one element at a time and so on);

- (time) we can postpone again the check for duplicates even if this check will be performed not on all the table but only in each bucket.

However, the impact of this optimization seems to be reduced, since in some cases intermediate sets can be represented with bitvectors, thus saving a lot of space. Furthermore, the check for duplicates is performed only on each bucket hence the impact depends heavily on its size. Therefore, an issue raised here is the size of the table.

These considerations lead us to the conclusion that the impact of this property in a hash table representation is less than in a list-based one. It is not clear whether it would be convenient to use it at all if we were adopting a hash table scheme, since we do not know whether the extra overhead introduced in terms of extralogue annotations to be supplied by the programmer, implementation complexity of the language, and execution slow down due to the effect of lazy evaluation would be offset by the improvements of the performance. Hence, before judging whether or not a list-based implementation (which uses the distribute over union optimization) is better than a hash table one, we have to decide whether it is worthwhile to use the distribute-over-union optimization at all in a hash table approach. Altogether, our impression is that the distribute-over-union optimization is not strictly necessary for a hash table approach.

Ideally, an integrated scheme using remote basing techniques, local ones and lists should be designed, using program analysis to choose the one which is more suited in each different situation.
Chapter 4

Object Size Analysis

We now focus our attention on a slightly different problem, namely, estimating the sizes of the data objects in a SEL program. Knowing them at compile time would allow us to optimize the execution by allocating properly the needed storage. In our present context, this information would allow us to allocate hash tables of appropriate sizes. However, this problem is in general undecidable, therefore we seek a way of statically analyzing a program for determining approximations of the sizes of the objects involved in the program. Our approach is based on the work of Debray, Lin and Hermenegildo [DLH90] for a relational Prolog-like language; we adapt their techniques to SEL [Jay90b]. We first discuss equational assertions, and subsequently subset assertions.

4.1 A Metric for Sizes

We start by defining what is meant by the size of an object and how to measure it. Since our definition is operational in nature, measuring sizes is greatly simplified.

Basically, the size of a ground term is the number of constant and constructor symbols in it. However, sometimes it is necessary to define different measures for different kinds of terms, e.g. length for lists, depth for trees, etc. Let $S$ be the class of functions that can map terms to their sizes. Then $\forall f \in S$ we have that $f : \mathcal{H} \rightarrow \mathcal{N}$, where $\mathcal{H}$ is the (Herbrand) universe of ground terms, and $\mathcal{N}$ is the set of natural numbers. Examples of these functions are the function \texttt{list.length} which maps each list to its length, the function \texttt{term.depth} which maps every ground term to the depth of its tree representation and the function \texttt{term.size} which maps every ground term to the number of constants and function symbols appearing in it.

Given a set of terms $T$, a substitution $\theta$ is said to be $T$-grounding iff $\forall t \in T$, $\theta(t)$ is a ground term. Now, for each function $f$ of $S$ we can define the function $\text{size}_f : \mathcal{H} \rightarrow \mathcal{N}_\lambda$, where $\mathcal{H}$ is the Herbrand universe, as before, and $\mathcal{N}_\lambda$ is the set of natural numbers augmented with the bottom ($\bot$) element (which stands for the undefined value).

$$
\text{size}_f(t) = \begin{cases} 
  n & \text{if } f(\theta(t)) = n \text{ for every } \{t\} \text{- grounding substitution } \theta \\
  \bot & \text{otherwise} 
\end{cases}
$$

Therefore, if, for instance, we have the set of terms $T = \{Y, f(Y,Z)\}$, and the substitution $\theta = \{Y \leftarrow a, Z \leftarrow b\}$, then $\theta$ is $T$-grounding, since $T\theta = \{a, f(a,b)\}$ does not contain any non ground term; however, given $S = \{f(X,Y,Z)\}$ and $\sigma = \{X \leftarrow c\}$, $\theta$ is not $S$-grounding, and $\sigma$ is not $T$-grounding, while $\sigma\theta$ is both $T$-grounding and $S$-grounding.

Analogously, we can define the function $\text{diff}_f : \mathcal{H} \times \mathcal{H} \rightarrow \mathcal{N}_\lambda$, which gives the differences between two measure $t_1$ and $t_2$ using $f$ as measuring function.

$$
\text{diff}_f(t_1, t_2) = \begin{cases} 
  n & \text{if } f(\theta(t_1)) - f(\theta(t_2)) = d \text{ for every } \{t_1, t_2\} \text{- grounding substitution } \theta \\
  \bot & \text{otherwise} 
\end{cases}
$$

Hence we will have the following relations:

$$
\begin{align*}
\text{diff}_{\text{list.length}}([b|T], [b, c|T]) &= 1, \\
\text{diff}_{\text{term.depth}}(f(a, b(X)), X) &= 2, \\
\text{diff}_{\text{term.size}}(f(a, X), Y) &= \bot.
\end{align*}
$$
4.2 Data Dependency Graph

The first step in our analysis consists of flattening all SEL assertions. Briefly, flattening is the process of transforming an unconditional assertion into a conditional one with the same meaning, but not having any function subexpressions. For example, flattening the following nrev definition:

\[
\text{nrev}(I) = I.
\]

\[
\text{nrev}([HT]) = \text{append}(\text{nrev}(T), [H]).
\]

we obtain

\[
\text{nrev}(I) = I.
\]

\[
\text{nrev}([HT]) = W := Z = \text{nrev}(T), W = \text{append}(Z, [H]).
\]

Note that the order of the equalities to the right of := reflects the innermost reduction order for subexpressions. For the sake of clarity, in the following we will call head of the clause the literal in the leftmost position, result of a clause the literal between the \(-\) and the \(\text{:-}\) symbols and body of a clause the sequence of literals after the \(\text{:-}\). Considering the flattened nrev clause, \(\text{nrev}([HT])\) is the head, \(W\) is the result and \(Z = \text{nrev}(T), W = \text{append}(Z, [H])\) are the body. Each equality in the body has the form variable = literal where literal is a function application whose subterms do not contain any function symbol. Under this representation the head and the result could be thought as special cases of literals, having respectively the variable and the literal part empty.

4.2.1 Building the Data Dependency Graph

The Data Dependency Graph (DDG) of a flattened SEL program is a digraph \((V, E)\), where \(V = \{s, e\} \cup N\), \(s\) represents the head of a clause, \(e\) the results, and \(N\) is a set of nodes corresponding to the literals in the body. \(E\) is a set of oriented edges connecting each node containing a variable assignment to a node using this variable in its body. The DDG makes evident the control flow in the program.

In figure 4-1 it is shown the DDGs for the two clauses defining nrev. Note that the head is the only node with a matching operation performed in its literal.

4.2.2 Size Relations

From the DDG we can define size relations between the terms of our program, which will be later used for estimating the sizes of the objects. The relations can be partitioned in two kinds:

1. the ones inside a node, between the terms in the literal and the variable;
2. the ones among depending nodes, between the variables in the predecessors and the terms in the literals of the successors.

Let \(\text{head}\), be the \(i\)th term in the head, \(\text{result}\), be the \(i\)th term in the result, \(\text{literal}_i\), be the \(i\)th term of the \(j\)th literal and variable, be the \(i\)th variable. Furthermore, let \(\psi(s_1, s_2, \ldots s_n)\) be our abstract approximation of the size of the result of applying the function \(f\) (having arity \(n\)) to arguments having sizes \(s_1, s_2, \ldots s_n\). An instance of the above DDG (for the nrev clauses) with the literals and their terms are shown in figure 4-2. Below we can define the size relation for our example:

\[
\text{size}(\text{literal}_1) = \text{size}(T)
\]
4.3 Difference Equations

Now we consider how to handle the above defined relations either to determine the size with a closed formula, or at least to constrain it with a difference equation. We have two different case depending on whether the clause is:

- recursive, both simple and mutual,
- non recursive.

4.3.1 Non Recursive Clauses

Non-recursive clauses need just a standard analysis in order to obtain a closed formula, although this does not mean that we are able to fully define their sizes, since the formula can contain a reference to the size of a recursive predicate. For instance consider the following definition of foo:

$$\text{foo}(x, y) = \text{append}(x, y).$$

The closed formula for $\text{foo}$, is:

$$\Psi_{\text{foo}}(z_1, z_2) = \Psi_{\text{append}}(z_1, z_2)$$

However, we cannot define it fully since we still need a closed formula for append, which happens to be recursive. (We later show that one can give a closed formula of append, but this need not always be the case for an arbitrary function.)

4.3.2 Recursive Clauses

To resolve the size function for recursive clauses, we need to perform the normalisation of formulas. Roughly speaking it consists of propagating the partial results about the sizes of the variables until we have obtained a complete formula for the size of the result. We define $F^i$ as the set of the relations we have obtained at the $i$th step of the process of normalization, $DS$ as the database of the previously obtained results, $TR$ the collection of the starting formulas for the DDG of the clause (or of the clauses if we are considering mutually recursive definitions). Then the process is the following:

- Let $F^0 = TR$, $i=0$;
- repeat
  - $i := i+1$
  - traverse the DDG node by node in topological order analyzing the corresponding formula;
- if there exists an equation of \( DB \) or of \( F^{(i-1)} \) whose lhs matches a portion of the rhs of the node, then substitute properly the portion of the rhs with the rhs of the equation and put this new equation in \( F \); else put the original equation in \( F \);

\* until \( F^n = F^{(i-1)} \)

We can apply this process to \( \text{assrev} \), knowing that \( DB \) is \( \{ \Psi_{\text{assrev}}(s_1, s_2) = s_1 + s_2 \} \) and we obtain:

\[
\begin{align*}
\text{size(literal}_1) &= \text{size(head}_1 - 1) \\
\text{size(literal}_2) &= \Psi_{\text{assrev}}(\text{size(head}_1 - 1)) \\
\text{size(literal}_3) &= 1 \\
\Psi_{\text{assrev}}(\text{size(head}_1)) &= \Psi_{\text{assrev}}(\text{size(head}_1 - 1)) + \
\end{align*}
\]

The problem now is to give a good approximation of the solution of a difference equation, rather than to solve the equation. Debray, Lin and Hermenegildo were interested in exploiting intrinsic parallelism of logic programs, hence they examined upper bounds of the solutions. Since we are interested in knowing how much storage to allocate for our data structure, we also are interested in upper bounds. Unfortunately we have not yet found good and efficient ways of approximating the difference equations unless we pose very strong constraints on the form of the equation.

### 4.4 Subset Assertions

We now extend the previously computed results to subset assertions, showing that almost nothing will change in the results. We consider two main situations:

\* iteration over the elements of a set with three different forms (\( s \) below is set-valued function):

\[
\begin{align*}
\text{a} \quad & p([x_1]) \text{ contains } s(x) \\
\text{b} \quad & p([l_1]) \text{ contains } s(t) \\
\text{c} \quad & p([l_2]) \text{ contains } s(x, t) \\
\end{align*}
\]

\* multiple subset assertions with the same lhs, e.g.

\[
\begin{align*}
\text{p(x)} \text{ contains } q(x) \\
\text{p(x)} \text{ contains } r(x) \\
\end{align*}
\]

Considering iteration on the elements of a set, we can give a simple approximation of the sizes of the results as follows:

\[
\begin{align*}
\text{a} \quad & \Psi_x(n) = n \times \Psi_x(k) \\
\text{b} \quad & \Psi_x(n) = n \times \Psi_x(n - 1) \\
\text{c} \quad & \Psi_x(n) = n \times \Psi_x(k, n - 1) \\
\end{align*}
\]

where \( k \) is the size of the elements of the set argument of \( p \). In this case we might overestimate the size since we are not considering possible duplicates elements.

Consider now the clauses

\[
\begin{align*}
\text{p(x)} \text{ contains } q(x) \\
\text{p(x)} \text{ contains } r(x) \\
\end{align*}
\]

We once again approximate with \( \Psi_p(s) = \Psi_q(s) + \Psi_r(s) \), recognizing that the resulting size is being overestimated.

Other situations can appear dealing with sets, like the one in the \texttt{iota} function we introduced in chapter 2:

\[
\text{iota}(n) = \begin{cases} 
\text{if } \text{eq}(n, 1) \text{ then } \{1\} \text{ else } \{n \mid \text{iota}(n-1)\}
\end{cases}
\]

However these can be handled considering the sets just as any other data structure (e.g., lists), without any peculiar reference to their being sets.

To sum up, to compute the size relations:

\* first define the size relations for the equational assertion
• then define the size relations for subset assertions;
• finally go back to the process of normalization and/or solution.

What seems clear is that the subset-equational paradigm of computing is very suited for this kind of analysis, which may turn out to be really useful in the process of compiling it.

### 4.5 A Final Example

In order to illustrate our method, for subset assertion, we refer to the previously given definition of \(_{\text{perms}}_{\text{dist}}\):

\[
\begin{align*}
\text{distr}(x, \{\}) &= \{\}.
\text{distr}(x, \{y, z\}) &\text{ contains } \{x, y\}.
\text{perms}(\{\}) &= \{\}.
\text{perms}(\{x, y\}) &\text{ contains } \text{distr}(x, \text{perms}(t))
\end{align*}
\]

Here the best choice for the size function is the cardinality. It is immediate that:

\[
\Psi_{\text{distr}}(n) = n \times \text{size}([[x, y]])
\]

which can be reduced to:

\[
\Psi_{\text{distr}}(n) = n
\]

Note that the size of the first argument of \(_{\text{distr}}_{\text{}}\) is not taken into account here since we are just interested in finding a formula for the resulting set cardinality in terms of the cardinality of the input set.

Now we can analyze \(_{\text{perms}}_{\text{}}\); obviously, from the first equation of \(_{\text{perms}}_{\text{}}\) we have that:

\[
\Psi_{\text{perms}}(0) = 1
\]

and from the second one:

\[
\Psi_{\text{perms}}(n) = n \times \Psi_{\text{distr}}(\Psi_{\text{perms}}(n - 1))
\]

Thus, using the previously constructed formula for \(_{\text{distr}}_{\text{}}\), we obtain:

\[
\begin{align*}
\Psi_{\text{perms}}(0) &= 1
\Psi_{\text{perms}}(n) &= n \times \Psi_{\text{perms}}(n - 1)
\end{align*}
\]

which is a difference equation whose solution is the factorial function, which is correct since the set of all the permutations of a given set has cardinality which is factorial in the cardinality of the argument set.
Chapter 5

The Virtual Machine

This chapter gives some suggestions for designing a virtual machine for SEL with sets implemented with hashing tables. First, we define what we mean by the term virtual machine. We divide the process of writing the compiler in two phases:

- design of the compiler as if it were the compiler for a dedicated machine for our language (virtual machine);
- implementation of the virtual machine on a conventional sequential architecture.

This approach has advantages of portability and modularity: changing the (real) architecture needs only little changes of the simulator; furthermore the user can make efficient the executor without altering the compiler. Also, certain changes to the compiler, e.g., type inference, can be made leaving the executor completely unaltered.

We do not plan to use specialized hardware for our implementation, since experience shows that special-purpose machines, such as Lisp machines, Prolog machines, etc., were equalled in performance in a few years by good compilers on conventional machines, which were also able to perform other operations that are difficult on the special ones.

One of the most interesting virtual machines is the WAM (Warren Abstract Machine) developed by D.H.D. Warren[War83] for Prolog. It is a register-based machine with essentially three stacks, two for control and one for data, and performs certain key optimizations, like clause indexing, last call optimization, and environment trimming. It has become a standard reference point for any logic language implementation[AK90]. We first give a brief description of the original WAM, presenting its instructions, data areas and controls. Then we present the virtual machine for a SEL implementation by showing how the WAM can be adapted to the special needs of SEL. Finally, we present our hash table extensions.

5.1 The Prolog WAM

The WAM runtime system consists of three stacks, a pushdown list, a code area and some registers, while the instructions are a good compromise between clarity and efficiency.

Memory Areas

The memory areas are used to store data, code and environment. They can be partitioned as:

- fast access memory areas (registers);
- slow access memory areas (stacks)

The fast access memory areas consist of registers which are used both for storing data and for maintaining the status of the computation. The most used ones are: $P$: program pointer; $CP$: continuation program pointer; $E$: last environment; $B$: last backtrack point; $TR$: top of the trail; $TH$: top of the heap; $HB$: heap backtrack point; $S$: structure pointer; $A_1, \ldots, A_n$: argument registers; and $X_1, \ldots, X_n$: temporary variables.

The slow access memory areas consist of the data needed for the computation plus some information about control. They are:

- code area, containing the compiled code for the program;
- local stack, for environments and choice points (also called control stack);
global stack for storing terms constructed during the execution (also called heap);
- trail, for holding pointers to variables that must be reset upon backtracking;
- pushdown list, for recursive traversal of terms during unification.

Instructions
The WAM philosophy is to have one instruction per Prolog symbol. The instructions format is:

```
opcode operand₁ operand₂ operand₃ operand₄
```

where the opcode depends on the Prolog symbols and on the environment and operand₃ can be an argument register, a variable, etc. Instructions can be partitioned as:

- instructions for handling unification (put_*, get_* and unify_*);
- instruction for managing the memory arcs (allocate, deallocate .);
- instructions for indexing and branching (try_match, switch_on_term, .);
- instructions for dealing with the cut symbol (stack cut, set_level and cut)

Optimizations
As we have already mentioned, several optimization strategies are implemented in the WAM, and in the next three paragraph we will briefly mention the most relevant.

Last Call Optimization: It is an extension of the tail recursion for logic languages and consists of overwriting the current frame on the local stack with the frame created in response to the last call of a clause. It can be applied if some environment conditions are satisfied and allows to have constant space computation for iterative computations, saving time and space.

Clause Indexing: Often the various clauses defining a predicate can be partitioned on the basis of the first argument in the head of the clauses. Clause indexing is the term used to refer to the selection of the appropriate clause at run time by inspecting the first argument of a call.

Environment Trimming: It can be viewed as an extension of last-call optimization in that variables in a clause are discarded even before control reaches the end of the clause, if no further reference to them is possible. It is up to the compiler to order variable cells so that there can be maximum trimming.

5.2 The SEL-WAM

Since the SEL-WAM is very similar to the WAM, we will just concentrate on the differences between the two. We first transform SEL clauses by the process of flattening (see chapter 4) so that it is in a form that is more tailored for the compilation into WAM-like instructions.

Set Matching

A major difference from Prolog is that there is no need for full unification in SEL since arguments to function must be ground. This fact has two major consequences:

(i) there is no need of trail, since variables cannot be conditionally bound; and (ii) the unify instructions can be substituted by the store and match equivalents. However, unlike Prolog, set matching can have multiple outcomes, hence there is a need to augment the choicpoint record with so-called branchpoints, to keep track of the branching within set matching.

Patterns of the kind {one|remainder} in general incur a space which is Θ(n²) (where n is the size of the set). However under some circumstances it can be reduced to be Θ(n) or even Θ(1)—the latter occurs when remainder is not used on the rhs of the clause. A complete discussion on this topic can be found in [Jay91]; how this can be accomplished in a hash table representation is described in the next section.
The Execution Model

There are also some differences in the execution model between the SEL-WAM and the WAM, due to the different languages they are tailored to:

Success Backtracking: The only situation in which the WAM backtracks is when there is a failure, but SEL-WAM also must backtrack upon success: each time the control reaches the end of a subset assertion, control must transfer to the most recent choice point, together with joining the newly computed set with the previous ones[Jay91].

Collecting Elements: The SEL-WAM instruction `collect` is placed at the end of any compiled code for subset assertions, and checks a call-mode register to determine whether to exit or to start success backtracking. In the former case the environment record is deleted, if it is not needed later; in the latter, a check for duplicates is performed on the newly created set.

5.3 SEL-WAM with Hash Tables

Now we examine how we can extend the SEL-WAM in order to support hash tables. We assume that program analysis will determine the base sets and estimates of their sizes as well as the types of elements in sets. Alternatively, user annotations may be used to provide this information. There are three issues to be addressed:

1. to define the runtime representation for hash tables;
2. to define suitable new instructions for hash tables;
3. to implement correctly the new primitives

Regarding runtime representation, hash tables and remote sets will be stored in the heap, along with other objects. We propose to keep a separate area to store 'static' base sets; by the term 'static' we refer to base sets that will be necessary for almost all the computation. In a strongly typed framework, we wouldn't need to specify the arity of each constructor in the stack, but, as explained earlier, we would need to store in each set which kind of set it is, in order to be able to access properly to it.

Regarding new SEL-WAM primitives, we propose the following instruction for building hash table set:

```
build_ht_set <type> <size> <pointer>
```

which builds a set of a specified type and size and returns a pointer to it. Furthermore, since sets can have different representations, we need to keep a tag for each set specifying its format; all existing primitives can be adapted thereafter. Finally we have to avoid rebuilding the hash table in recursive definitions, like the one of `perm`.

Regarding existing instructions, the behaviour of the three `addset` instructions are re-implemented as follows to make use of hash tables:

- `addset_head`: We need only a pointer which scans the bitvector until it finds the next applicable element of the set.
- `addset`: This primitive can implemented using a bitvector for representing the remainder of the set. This bitvector can be initially set to a copy of the bitvector representing the set and then each time we will take the 'head' and the remainder, we will change the bit corresponding to the head to 0.
- `addset_with_copy`: In this case we have to create a bitvector each time we take the 'head' and the 'tail' of our structure. This bitvector will be identical to the one of the previous step but the examined element, which will be turned to 0.

In our approach the different implementation of sets is transparent for `match` and `store`, unless we want to optimize their behaviour. The same can be said for `getset` and all the indexing instructions (switch.on.ground_term, ...). On the other hand, few words are necessary for the procedural instructions: as mentioned earlier, the check for duplicate in a hash table is much simpler and more efficient than with linked lists, hence an ad hoc implementations could improve the performance. Finally, the call.all, call.one and last.call.one primitives seem unnecessary since the property of distributing over union wouldn't speed up the language if we use a hash table representation for sets.
call one perms tail/2 R
put.value Y1 A1 % distr(x)
put.value Y4 A2 % v2 contains
put.value Y5 A3 % v3
last call one distr tail/3 Y3 R
collect? Y3 Y5 % v1 := v1 union v3

5.4 A Sample Compilation

We conclude our survey on design strategies, showing a possible compiled code for the perms function. Our hypothetical compiler would perform a sequence of abstract analyses, including the one described in chapter 4, and its final output might be as shown below. Note that the hash table is just built once and all the accesses to the set use on it. Note that the instruction set below might nor be the most suited for a hash table implementation; however we keep it because of its compatibility with the other SEL-WAM implementations.

perms/2:

compute_output_size X1 % Computing the size
build ht set int X1 R % Building the hash table
perms tail/2: % Recurs. call: no building of the set here.
switch on ground term C1, fail, fail, C2

C1:

gt empty set A1 % perms(()) =
store set R % []
store constant [] % []
store empty % []
proceed

C2:

allocate
get set A1 Y0 % perms( 
adj set Y0
match variable Y1 % x[]
match variable Y2 % t[]) contains
get variable Y3 R % v1
save choice point % :-
put value Y2 A1 % perms(t) contains
put variable Y4 R % v2
Chapter 6

Conclusions and Further Research

The aim of this thesis is to provide some guidelines for improving the implementation of SEL from various considerations, rather than to propose a detailed solution.

Altogether it seems that hash tables are quite suited to represent sets in the framework of SEL: common operations, such as pattern matching, test for membership, set iteration and so on appear to be faster than in the current (unordered-list based) representation. On the other hand, it seems that the language implementation would be more complicated with hash tables, and that abstract analysis techniques would play a more important role than they do now.

Further work may be directed in at least four areas.

1. It would be interesting to analyze how other commonly used data structures could implement sets: balanced trees, for instance, could be thought of as an intermediate point between the unordered list and the hash table representation.

2. Several features can be added to SEL itself for bettering its user interface, such as: a type system and type inference, allowing functors, an explicit union operator, the relative sets construct, and closure operations via stratified SEL clauses.

3. Abstract analysis is an area which seems to be one which could offer the opportunity for substantial improvement of the performance.

4. As SEL is a purely logic language, it offers great opportunities for a parallel implementation allowing implicit parallelism. This is another topic for further investigation.
Appendix A

Sample Programs

In this appendix we give the code for some SEL sample programs we developed to analyze the way in which sets are constructed in SEL. For each program, we first give its code, followed by a brief explanation.

A.1 Huffman Code

\[\text{huffmanCode}(\text{list}) = \text{finalCode}(
\text{makeCode}(\text{elem}(\text{list}),
\text{union}(\text{tag}(\text{list}), \text{cylBld}(\text{totalFreq}(\text{list}))))))\]

\[\text{finalCode}([a.]) \text{ contains } \text{tailReverse}(a).\]

\[\text{tailReverse}([a\text{[list]]) = [a\text{[reverse(1st])].}\]

\[\text{reverse}(\text{1st}) = \text{reverse2}(\text{list}, []).\]

\[\text{reverse2}([], \text{list}) = \text{list}.\]

\[\text{reverse2}([a\text{[rem]}, \text{list}) = \text{reverse2}(\text{rem}, [a\text{[list]).}\]

\[\text{makeCode}([a.], \text{codFreq}) \text{ contains } \{\text{buildCode}(a, \text{codFreq})\}.\]

\[\text{buildCode}(a, [\text{[', a,f][remFreq]) = [a]\text{[followChain}(f, \text{remFreq})].\]

\[\text{followChain}(f, [[f, e,d][remFreq]) = [d]\text{[followChain}(f, \text{remFreq})].\]

\[\text{followChain}(., .) = []\]

\[\text{tag}(\text{1st}) = \text{addStar}(\text{totalFreq}(\text{1st})).\]

\[\text{addStar}([[a,f][.]) \text{ contains } [[', a,f]].\]

\[\text{totalFreq}(\text{1st}) = \text{totFreq}(\text{1st}, \text{baseFreq}(\text{1st})).\]

\[\text{baseFreq}([[a,][]) \text{ contains } [[a,1]].\]

\[\text{baseFreq}([[,],]) \text{ contains } \text{baseFreq}(\text{1st}).\]

\[\text{totFreq}([], \text{freqSet}) = \text{freqSet}.\]

\[\text{totFreq}([a\text{[list}],[a,][\text{rem}) = \text{totFreq}(\text{1st}, [[a,(i+1)[\text{rem}].}\]

\[\text{cylBld}([], \text{huffTree}) = \text{huffTree}.\]

\[\text{cylBld}(s, \text{huffTree}) = \text{cylBld}((\text{union}(\text{diff}(\text{diff}(s, \text{minFreq}(s)),
\text{secMinFreq}(s)),[[\text{freq}(\text{minFreq}(s))\text{freq}(\text{secMinFreq}(s))],
\text{freq}(\text{minFreq}(s))\text{freq}(\text{secMinFreq}(s))),', '1'),
\text{freq}(\text{secMinFreq}(s)), '0'), \text{huffTree}).\]

\[\text{minFreq}([[a,1]]) = [a,1].\]

\[\text{minFreq}([[a,1],[b,1][s]) \text{ if } i \geq j \text{ then } \text{minFreq}([[a,1][s])\]

\[\text{secMinFreq}(s) = \text{minFreq}(\text{diff}(s, \text{minFreq}(s))).\]

\[\text{freq}([[,f]]) = f.\]

\[\text{elem}([[a,][]) \text{ contains } a.\]

\[\text{elem}([[,],]) \text{ contains } \text{elem}(\text{1st}).\]

\[\text{union}(s, \text{rem}) \text{ contains } a.\]

\[\text{union}(., b) \text{ contains } b.\]

\[\text{diff}([[x,]],[t]) \text{ contains if } \text{member}(x, t) \text{ then } \text{else } [t].\]
This program computes the Huffman code of a sequence of character stored in
a list: it uses the standard approach consisting on looking at the two symbols with
minimum frequency, grouping them and generating a new symbol having as frequency
the sum of the frequencies, until only two symbols are left.

### A.2 Topological Sorting

topSort(graph) = cyclePred(cyclePred(numPred(baseNumPred(
  nodes(graph)), {}), {}, numPred(baseNumPred(
  nodes(graph)), {}), graph).

cyclePred({}, list,..) = list.
cyclePred({n|rem}, list, predList, graph) = cyclePred(union(rem
  newNoPred(n, graph, predList)), {n|list},
  newPredList(n, graph, predList), graph).

newNoPred(n, {{c|rem1}, {{c, 1}|rem2}}) contains {c}.
newPredList(n, {{c|rem1}, {{c, 1}|rem2}}) =
  if i>1 then newPredList(n, rem1, {{c, (i-1)|rem2}})
  else newPredList(n, rem1, rem2).

numPred({{c, 0}|rem}) contains {c}.
numPred({}, numPred) = numPred.
numPred({{L, c}|rem1}, {{c, i}|rem2}) =
  numPred(rem1, {{c, (i+1)|rem2})
baseNumPred({{L, |list}) contains {{a, 0}}.
nodes({{a, |list}) contains {a}.
nodes({{L, b}|list}) contains {b}.

This program produces the topological sort of the elements of a graph using the
standard algorithm: for each element the number of predecessors is computed and
all the elements without any predecessor are stored in a resulting list; a new graph is
then obtained deleting these nodes, and the process goes on until the graph produced
is empty.

### A.3 Minimum Distance

adjmat(g) = collectdist(nodes(g), g)

startdist(x,y, [[x,y,z]|.]) = x.
startdist(x,y,.|.) = 10000.
collectdist([x[y]|.],g) contains [[x,y,startdist(x,y,g)]]
makenewdist([[x,w,d][d]],x,w) = [[x,w,d]].
makenewdist([[x, w, d1][v,z,d2][x, z, d3][rem]], x, w) =
  if (d2>0 and (d1+d2<d3)) then
    [[x, z, (d1+d2)][v, z, d2][makenewdist([[x, w, d1]
    [rem]], x, w)] else [[x, z, d3][v, z, d2]]
    makenewdist([[x, w, d1][rem], x, w])].

h2wp(distmat, notexm, x, y) = elimuncyclic(h2wp(distmat,
  notexm, x), h3wp(distmat, notexm), x, y).
h2wp(distmat, notexm, x) = makenewdist(distmat, x,
  mindistinside(notexm, distmat)).
h3wp(distmat, notexm) = diff(notexm,
  mindistinside(notexm, distmat)).
elimuncyclic(distmat, notexm, x, y) = if member(y, notexm)
  then h2wp(distmat, notexm, x, y) else distmat.
hibmp(g,x) = diff(nodes(g),x)
buildinpath(x,y,g) = elim.uncyclic(adjmat(g),hibmp(g,x),x,y).
minpath(x,y,g) = get(x,y,buildinpath(x,y,g)).
mininside([x],[g]) = if empty(g)
then id(x) else minin2([x],[g]).
minin2([x[y][rem1]],([[x,hx,1],[y,hy,j][rem2]]) = if i \leq j
    then mininside([x[y][rem1]],([[x,hx,1],[rem2]])
                   else mininside([y[y][rem1]],([[y,hy,j][rem2]])
get(x,y,[[x,y,d]],) = d.
id(x) = x.
nodes([[x,y,[j]],[[j]]) contains [x]
nodes([[x,y,[j]],[[j]]) contains [y]
member(x,[x,]) = true.
member(_,[y]) = false.
empty() = true.
empty() = false.

This program computes the minimum distance between two nodes in a given graph using the method of the adjacency matrix; two non-connected nodes are initially supposed to have infinite distance and then, step by step, the adjacency matrix is filled in with the appropriate values.

### A.4 Cyclic Graphs

hascycle(g) = isnotempty(diff(nodes(g),uncyclicpart(g)))

uncyclicpart(g) = elim.uncyclic(roots(g),{}).
    nodes(g),taggedvert(g)).
elim.uncyclic({},uncyclic,...) = uncyclic.
elim.uncyclic([x][rem1],uncyclic,[x][rem2],tagvertex,g) =
    elim.uncyclic(union(rem1, newuncyclicportion(x,{}),tagvertex,g)),
    [x][uncyclic], retag(x,{}),tagvertex,g)).
newuncyclicportion(x,nopred,{}),g) = nopred.
newuncyclicportion(x,nopred,[[y][ii][rem]],g) =
    if member([x][ii],g) and equal(x,1) then
        newuncyclicportion(x,[[nopred],[rem]],g)
    else newuncyclicportion(x,nopred,rem,g)
taggedgraph(g) = dotag(starttag(g),g).
starttag([x][])) contains ([x][00]]).
dotag([x,],) = s.
dotag([x][0]) = dotag([x][1][rem1],([x][y][rem2])) = dotag([x][1+1][rem1],rem2).
roots(g) = extractzero(taggedgraph(g)).
extractzero([x,0]]) contains [x].
retag(x,retagd,{},g) = retagd.
retag(x,retagd,[[y][ii][rem]],g) =
    if member([x][ii],g) and i > 1 then
        retag(x,[[y],[i-1]][retagd],rem,g)
    else retag(x,retagd,rem,g).

This program checks whether a given graph is cyclic. The graph is specified as list of couples [x][y], where x and y are the edges defining an arc in the graph. A reference count approach is taken. A numerical tag is assigned to each node indicating
the number of its predecessors; the nodes with tag 0 are removed. This process is repeated until it converges to a set: if it is empty then the graph is not cyclic, else it is cyclic.

Appendix B

The SEL Environment

Here below there is a sample SEL session to illustrate the main comands of its environment. We enter the SEL world calling sel. In this framework we can:

• reduce an expression, e.g., solve(4);;

• compile a file containing SEL definitions, e.g., compile('queens.sel');;

• ask for tracing the reductions using the command trace.

XX sel

SEL Version 1.0

sel> compile('queens.sel').

[iota, queens, solve, placequeen, safe, ]

sel> solve(3).

[]

sel> solve(4).

[[4|3], [3|1], [2|4], [1|2]], [[4|2], [3|4], [2|1], [1|3]]

sel> trace.

sel> solve(2).

Call to solve(2).

Call to iota(2).

Call to iota(1).


