Structuring Sets in Declarative Languages.

Giancarlo Succi*  
Angelo Bergamini* and Cristina Dondina*

*LII/DISA  
Università di Trento  
email: charm@lii.unitn.it  
Via Zeni 8 I-38068 Rovereto

*Università di Genova  
Dipartimento di Informatica Sistemistica e Telematica  
email: pegaso@dist.unige.it  
Via Opera Pia 11A, 16145 Genova, Italy

Abstract  
This paper presents an abstract machine derived from the classical WAM, designed to execute SEL programs. The focus of the paper is on the new machine instruction set and memory supports added to the WAM structure to manage sets in a clean and efficient way. Comparison will be made between two implementation based on different technique for representing sets (viz. linked lists vs. hash tables).

Keywords: Set, Dynamic Hashing, Abstract Machine

1 Introduction

Set theory is used in many different fields to represent problems as relations between sets; moreover unions and intersections are not so rare in real life problems. However it is hard to find a good implementation that resembles the features of sets, especially those regarding the random access to the elements, and that allows an efficient implementation of union and intersection. The gap between unordered and ordered structures, i.e., sets and computer memory, should be filled to allow a wide use of sets in different programming paradigms. In this paper the way we try to reduce this gap is depicted following this organization: section 2 is a brief informal overview about languages that work on sets; section 3 is devoted to the description of the classical WAM and the type of hashing method; section 4 describes the new memory supports added to the WAM’s ones to manage sets; section 5 presents a simple example that shows how the new structures are used; section 6 presents the hashing functions used in the abstract machine; section 7 introduces the new machine instructions set; section 8 outlines the results obtained with the new abstract machine and section 9 draws some conclusions and evidence open problems for further work.
2 Languages Overview

Sets are clear and simple data structures that can be easily used in the resolution of many practical problems. Unfortunately, their use as a programming tool has been limited by the complexity of mapping their "wild" structure into fit-to-computer form. Generally, sets are represented with linked lists which impose an arbitrary order on the elements. That causes a computational extra overhead. Only relatively few programming languages provide sets as primitive objects. Among them, the procedural language SETL [FSS83, SSS81, SDG+79] and the functional languages MIRANDA and ME TOO. In the last few years, the evolution of logic programming languages resulted in more effort being put towards interpreting set theory into this paradigm. Indeed such languages seem good candidates for hosting set data abstractions due to their potentially high declarative nature. \{Log\}[DOPR91] is an extended logic language, that is definite Horn clauses with no extra-logical constructs. Then special set terms are added to this language to allow the extensional representation of finite sets. Very few distinguished predicates, namely set membership and equality are added to the base language, along with their negative counterparts. A new unification algorithm which can take into account the semantic properties of with (namely, right permutativity and right absorption) is developed and proved to terminated.

SETL is the first successful attempt of using sets as a core data structure. It is a very high-level language whose syntax and semantics are based on the standard set theoretical diction of mathematics. It is designed to allow complex algorithms and large-scale programs to be written succinctly and rapidly, in a form closely resembling their abstract mathematical statement. Set is the most important data type and a whole set of operations is provided.

The SETL optimizing compiler allows an automatic data representation selection in order to improve execution efficiency. Different kinds of representation are possible for sets depending on the operations performed on them. SEL [SM91a, Suc91], which is the language we refer to, introduced the subset assertion, used whenever the argument of a function is a set. It, is therefore a more complete tool for those programmers having experiences in representing problems with relations between sets. It is a powerful logic language like Prolog but substitutes unification and backtracking with matching operations. An Abstract Machine based on WAM [AK90] has been implemented and will be described in this paper. One of the remarkable features of this language is the ability of exploiting data parallelism in a simple way [SM91b].

3 Extending the WAM

In this paragraph only data structures will be discussed, later on the new instructions adopted to manage and store sets will be described.

In the first version of the modified WAM sets are stored with the same technique used for lists in the classical WAM. Two contiguous cells are used to address the current element and the remainder of the set. Operations like iteration on the elements of a set can take advantage of this implementation of sets, but those one requiring direct access to the elements (e.g. pattern matching and membership test) are slowed down, especially when the size of sets grows. Since both these kind of operations are frequently performed working with sets a new method to manage them is needed to improve performances. Among the different possible solutions the use of hashing techniques has been selected:
it speeds up direct access of one complexity degree (from linear to constant) and slows down linear access of some extent (but not one degree).

The choice of the more suitable hashing and collision management techniques is based upon the following assumption:

- the amount of elements of a set is unknown at all and it is subject to great fluctuations;
- the access must be as faster as possible;

Dynamic hashing [Har88, ED88, CW79] resembles these requirements: it allows to adjust the size of the table according to the number of elements of the set and to have a direct access to the elements reserving one cell of the hash table for each one. This is accomplished expanding the table after each collision, avoiding the problem of a possibly long search in a list of synonyms.

It is necessary to underline the difference between relative and absolute synonyms:

- relative synonyms have the last \(m\) right-most bits equal;
- absolute synonyms have the same string of bits (e.g. equal hash code).

The size of the table is always a power of 2, that is the table is doubled whenever an expansion takes place. The address of the table where the reference to the element is to be inserted is obtained applying the function \(\mod\) to the hash code previously computed:

\[
<\text{address}>> = <\text{hash code}> \mod <\text{current size of the table}>
\]

That is the hash code is considered as a string of bits (e.g. its binary representation) and only a part of them is used to generate the address: if the size of the table is \(2^n\) then only the \(n\) right-most bits will produce the address.

The relative synonyms can be handled with the expansion of the table because the \((m+1)\)-th right-most bit is different and sufficient to discriminate between the two elements; the absolute synonyms must be handled with a linked list of synonyms and their frequency is strongly related to the choice of the hash function. Obviously the table cannot grow indefinitely and an upper bound must be posed on its size; relative and absolute synonyms are both handled with chaining when this limit is reached.

4 Structure of the Extended WAM

In this section there is the discussion on how the WAM has been modified to support sets with hash tables. It is necessary to add a new memory space devoted to the hash tables of the sets and one to store the information regarding each set. Furthermore a new area has been used to store the elements of the sets and it has the same shape as the heap. The new structures of the modified SAM are:

1. Set Table: is a table made by cells in which the information concerning the sets are memorized,

2. HT Space: is an area in which the hash tables are cut out, one for each set.

\[\text{It is not really necessary here but it is been introduced to make easier the use of hashing in the data-parallel implementation of the modified WAM.}\]
3. **Data Space**: is an area in which the elements of the sets are stored.

Each cell of the *Set Table* contains:

- *size* of the hash table of the set,
- *address* of the hash table in the HT Space needed to identify the set considered,
- *number of the elements* contained in the set,
- *hash code* of the set considered as element of another set.

Each cell of the *HT Space* contains:

- *hash code* of the datum,
- *address* of Data Space where the element is stored.

The first two fields should be stored together with the element they are referred to, leaving HT Space with just one field per cell. In this way the copying of a table during the expansion would be speed up, but, on the other side, the control of the expansion would be slow down since it needs these information.

Data Space is handled almost in the same way as the heap and contains the *datum* itself if it is simple (viz. a constant or an integer value), its memory location if it is complex. The memory location can address three different memory spaces:

1. if the datum is a set the corresponding address must be used to access the Set Table;
2. if the datum is a list the corresponding address indicates the heap location where the first element of the list is stored;
3. if the datum is a functor, the address is interpreted as a Data Space address.

5 **An Example**

In this section a simple example is presented to clarify how the previously depicted structures can be used. At the start one cell of Set Table is reserved for the whole set. The first free HT Space cell address is inserted in its field and a two cells table is located on HT Space (since it is likely that 2 is the minimum set dimension). The storage of an element consists of these steps:

1. the data are stored in suitable memory areas (as pointed before),
2. the element hash code is computed,
3. the cell of the hash table of the considered set where the reference to the element should be stored is reached through the hash code:
   - if the element is already present, the memory space used in step 1 is recovered,
   - if the element is not present, the reference is inserted.

\(^2\)Again for compatibility with the data-parallel implementation.
Let us suppose to store the set \{couple(1,2) , [3,4] , {"dog" , "cat"} \} is shown.
Let’s suppose that the hash codes of the elements are as follow:

- couple(1,2) has hash code 1.
- [3,4] has hash code 2.
- {"dog" , "cat"} has hash code 3.

that the hash code of the strings dog and cat is the address of the Tables of Constants
where they are stored and the hash code of a set is the sum of the hash code of its
elements. The Table of Constants contains all the strings present in a SEL program.
The first free Set Table cell is assigned to the current set and its fields are updated step
by step. The address of the first free HT Space cell is inserted in the corresponding field
and a two cells hash table is reserved for this set setting to two the value of table size.
The set cardinality field and the set hash code are set to 1. The reference to the functor
is stored in the hash table after inserting the functor and its arguments in Data Space
because it is necessary to know the Data Space address of the functor and to perform
a duplicate check. At this point the corresponding field in the hash table associated to
the set can be modified. The storage technique used for a functor is the same as the one
adopted in the WAM. The field corresponding to the number of set elements in Set Table
is fixed to 2 and the one corresponding to set hash code is set to 3 when the list [3,4] is
stored. Besides the field Value of the first free Data Space cell is set to heap address at
which the first element of the list has been stored. As the element to be memorized is a
list, the storage technique is the same as the one adopted in the WAM.
Finally the last element must be memorized. The set element is stored performing the
same steps: a new cell of Set Table is assigned, the elements are stored in Data Space and
their references are put in a new hash table.
The new address of Set Table is inserted in Data Space, together with the tag SET, to
identify the set element. An expansion of the first hash table takes place when the address
of Data Space, where the last element has been stored, must be inserted in it. It happens
because the hash table has been completely filled. In this particular case a displacement
of the original table is necessary because the adjacent cells are used for another table.

6 Hash Functions

The choice of a good hash function is a very critical phase when these methods are
applied; moreover in our application there is an additional problem: the elements of a set
in a program can be very different one from another and their complexity is not fixed.
Set elements can be divided into three main categories:

- simple data like strings, numbers and atoms,
- complex data like lists, sets and functors,
- structured data viz. complex data containing complex elements.

For this reason, the hash function must behave in different ways depending on data and
must work on a restricted part of the element if it is deeply or widely nested. If not, access
time would grow making useless the hashing technique. Moreover, it is necessary to select
those parts of the structured element which are likely to produce as few synonyms as possible. The hash function of a list, for example, can be based both on its length and on the hash code of its first i-elements while for a functor can be significant the name, the arity and some of its arguments.

Sets, instead, bind the hash function to compute the hash code using the information about all of their elements; since any order cannot be imposed on the elements of a set, two sets with the same elements ordered in different ways could produce different results if only parts of the available data were used. In this case the operation executed on the hash code of the element must be simple and fast.

Strings appearing in a SEL program can be stored in a special area called Table of Constants since they are already known at compile time. This allows the use of a perfect hashing function [Cic80] for this kind of datum. A perfect hashing function is a hashing function "trained" to generate different values for each string. It uses the following algorithm to compute the hash code of a string: hash code is the sum of the length of the string plus the sum of all of the values associated to the characters of the string. The training of perfect hashing function consists in tuning the values associated to each character so that the hash code of each string is unique.

7 The Abstract Machine Instruction Set

In this section the instruction set adopted in the modified WAM will be described. At first a brief discussion about the similarities and differences between the original and the modified instruction set will be made, then the new instructions will be illustrated.

The argument loading and analysis instructions (PUT and GET) are preserved and two new one have been added: put_set and get_set perform the put and get operations on sets.

Given the specifications of SEL., these instructions do not unify but only match their arguments. By the same way the unify instructions have been replaced by store and match instructions.

The new instructions introduced to manage sets can be grouped into three main categories:

- **map** and **search**
- **fold**
- **match**

The map operations are used when an operation is to be performed on all the elements of a set in a subset assertion like

\[
\text{incr_set}\{X\} \text{contains}\{X + 1\}.
\]

which increments by one all the elements of the argument set. This assertion is translated in the following instructions for the WAM:
[1] get_set $X_8$, $A_1$
[2] get_variable $X_7$, $A_2$
[3] map_over $X_8$, $X_6$, $X_5$ [10]
[.] <increment the value contained in $X_5$>
[.] <store the result in $A_2$>
[9] end_map_over $X_6$, $X_5$, [4]
[10] ...

where map_over and end_map_over cooperate to select and store in the proper registers the elements of the set. Registers $X_5$ and $X_6$ are used to contain information for indexing the set. [10] and [4] are the addresses where it is necessary to jump when, respectively, the set is empty and there are still elements to work on.

The map operation is specialized to face three particular cases:

- when the remainder of a set needs to be explicitly generated, as in the assertion

  \[
  \text{sub_from_set}(X, \{X|T\}) \text{ contains } \{T\}.
  \]

  or in

  \[
  \text{all_sum}(\{X,Y\_{..}\}) \text{ contains } \{X+Y\}.
  \]

  the map\_generating\_copy instruction is used;

- when the data to work on is a part of a complex element as in

  \[
  \text{sum\_functor}(\{f(X)|_{..}\}) \text{ contains } \{X+1\}.
  \]

  the map\_over\_matching instruction is used;

- for those assertions in which both the previous cases are present the map\_generating\_matching instruction has been implemented.

When it is necessary to perform a match before using the element a continue_map\_* instruction is used. An example of this case will be shown later.

This class of instructions is the only one that takes no advantages on hashing technique because, in the sequential version, it is inherently sequential.

The search operations are used in equational assertion in place of the map instruction because it is necessary to perform an operation on just one element.

The fold operation is a machine instruction to which this couple of assertions is reduced to:

\[
\begin{align*}
  f(\{\}) &= \text{constant}, \\
  f(\{X|T\}) &= g(h(X), f(T)).
\end{align*}
\]

It allows to turn a recursive assertion into an iterative one reducing both the use of stack for the environments and time to build and erase them.

The match instructions are used whenever it is necessary to find a pattern in a set and are frequently used inside a map\_* -- end_map\_* cycle; hashing can reduce these instructions to constant time operation, allowing a great performance improvement. A

---

3 Obviously, for each one there is the corresponding end\_* instruction.
4 The data-parallel version of the modified WAM is able to perform the same operation on the whole set in just one step.
simple example of the use of this instruction is in the assertion that returns the intersection between two sets:

\[ \text{intersect}\{X\}, \{X\} \text{ contains } \{X\}. \]

that is compiled in:

1. \text{get_set } X_{10} A_1
2. \text{get_set } X_9 A_2
3. \text{get_variable } X_8 A_3
4. <\text{prepare the matching support structures}>
5. \text{map_over } X_{10} X_5 X_6 [11]
6. \text{match_set } X_9 X_7 X_4
7. \text{continue_map } X_4 X_5 X_6 [7] [11]
8. <\text{store the matching term}>
9. \text{end_map_over } X_5 X_6 [7]
10. ... 

Moreover, a complete set of instructions for storing sets, both arguments and results, has been developed which guarantees, with a relative low cost, that there are no duplicated elements.

8 Conclusions

In this paper the use of hash tables for implementing sets is presented. The results seem to be interesting even if obtained with very simple hashing functions, however, before we can give a definite evaluation of the system we need some more tests. Many are the open problems on this specific subject of this fields as the use of perfect hashing functions and the use of bases for representing sets.

Acknowledgements

Some of the ideas presented in this paper has been discussed with B. Jayaraman.

References


